Win-GRAF Workbench

User Manual

(Version 1.0)

WARRANTY

All products manufactured by ICP DAS are warranted against defective materials for a period of one year from the date of delivery to the original purchaser.

WARNING

ICP DAS assumes no liability for damages consequent to the use of this product. ICP DAS reserves the right to change this manual at any time without notice. The information furnished by ICP DAS is believed to be accurate and reliable. However, no responsibility is assumed by ICP DAS for its use, nor for any infringements of patents or other rights of third parties resulting from its use.

COPYRIGHT

TRADEMARK

Names are used for identification only and may be registered trademarks of their respective companies.

CONTACT US

If you have any questions, please feel free to contact us via email at:
service@icpdas.com
service.icpdas@gmail.com

Revision

Revision	Date	Description	Author
1.0	09.11.2021	Initial version	M. K.

Contents

1 Product Overview

1.1 Introduction

Win-GRAF Workbench is the Soft PLC development environment provided by ICPDAS. Win-GRAF supports the five programming languages as defined by the IEC 61 131-3 standard. The workbench allows multitasking programming with priority settings, PLC application program download to the target runtime and online debugging by displaying runtime parameter values directly in the source code of the programming editor. HMI software provided by ICPDAS such as eLogger and Indusoft has been integrated into to the workbench. All standard Modbus protocols (TCP, RTU, ASCII) are supported. In addition real-time EtherCAT and PLCopen defined motion control is supported when using the runtime together with the EtherCAT master cards ECAT-M801/e-M901 of ICPDAS.

This manual describes the key features of the Win-GRAF workbench. Basic knowledge of the Soft PLC concept and its programming language is a prerequisite.

2 Workbench and Runtime Installation

2.1 Installing Win-GRAF Workbench

The Workbench has to be installed on a Windows PC. Before installation make sure that your PC meets the following requirements:

- Operation system: Windows 7, Windows 8, Windows 10 (32-bits or 64-bits)
- RAM: 1 GB minimum (Recommended: 2 GB or more)
- Available hard-disk space: 200 MB minimum

Installation Steps:

- ▪ Download the workbench installation file from the ICPDAS website.
    - Website: https://www.icpdas.com/
    - Enter the keyword *'Win-GRAF'* into the search box and select *'Win-GRAF workbench'* from the drop list.
- ▪ Double-click the *'Win-GRAF_Workbench_xx.xx_Setup.exe'* setup execution file and follow the execution steps.

2.2 Run Win-GRAF Workbench

After the installation process has successfully been completed the workbench is ready to be started by clicking the *'Win-GRAF Workbench xx.xx'* in the start menu. Before running the Win-GRAF workbench make sure the USB license key is plugged into your PC otherwise the workbench will run in demo mode. A PLC program which has been compiled by the demo workbench version will only run for about fifteen minutes before it will be terminate by the runtime. If the workbench has been started before the USB dongle key has been inserted then it needs to be restarted in order to run the fully licensed version.

2.3 Win-GRAF Runtime Platforms

ICPDAS provide a variety of Soft PLC hardware platforms:

- Dual PAC Redundant System: RPAC-2658M
- Win-GRAF Based ViewPAC: VP-x238-CE7, VP-x208-CE7
- Win-GRAF Based PAC: WP-9x28-CE7, WP-8x28-CE7, WP-5238-CE7
- EtherCAT Motion Controller: EMP-9xx8-xx
- EtherCAT motion control on a standard Windows PC:
    - Runtime has to be installed on a the PC and a the EtherCAT master card has to be plugged into the PCIe slot.
    - EtherCAT master card: ECAT-M801-xx

3 Workbench

The Win-GRAF Workbench is used for configuration, programming, and debugging. The workbench supports all standard soft PLC programming languages such as Structure Text, Function Blocks, Ladder, Instruction List and Sequential Function Charts. The workbench supports cold restart, hot restart and on-line changes. Multitasking programming with task priority and cycle time setting is possible. Tools are provided for event based communication between different Win-GRAF runtimes. The event are time based.

HMI communication interfaces for Indusoft and eLogger (ICPDAS developedHMI) are part of the workbench tools. Programming interfaces for c++, c#, LabVIEW enables data exchange between runtime and third party software.

The workbench include standard Modbus TCP/IP, RTU and ACSII protocols. Real-time EtherCAT communication and PLCopen defined motion function blocks are supported by the ECAT-801 PCIe card and EMP-9xx8-xx series. Separated manuals are provided for PLCopen, EtherCAT and OPC UA server.

Programming, download and debugging of application programs is done remotely via Ethernet TCP/IP.

The Win-GRAF workbench is a licensed software tool which requires a USB dongle on your Windows PC.

The main user interface (UI) of the Win-GRAF workbench is shown below (Figure 1).

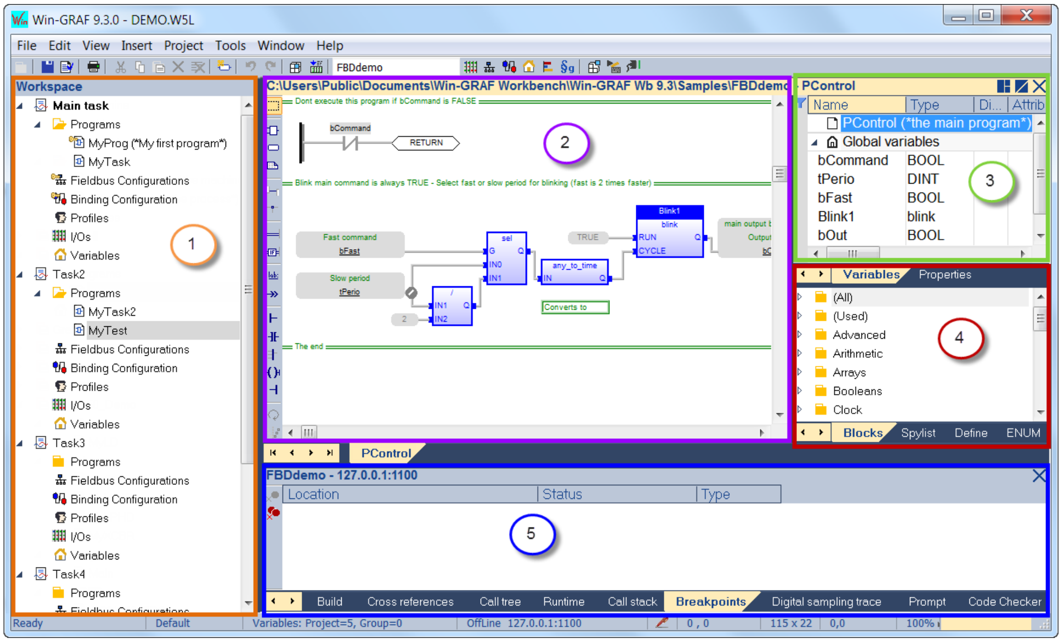


Figure 1: Win-GRAF workbench UI

The interface consists of the following parts:

Step 1: Workspace window:

This window list the following items:

- Tasks supported by the runtime in a tree. For each task several programs can be added,
- Access to the Fieldbus selection and configuration interface. The user interfaces for the HMI and shared memory are found in the fieldbus section
- Access to the I/O configuration interfaces
- Access to the variable editor window
- Access to user defined PLC libraries and data types
- Event driven data exchange between two PLC runtime configured and a communication interface with a HMI created.

Step 2: Program editor window:

Here PLC programs and modules are created and edited. The program has to be edited in one of the five IEC61131 defined languages.

Step 3: Variable editor window:

PLC variables has to be declared in the editor. Global and local defined variables and function block instances are displayed in the list. For each variable the name, data type, dimension, scope, attribute, etc. has to be entered.

Step 4: Block window:

- Library: List all the standard IEC function and function blocks. In addition function for Ethernet and Modbus communication are available. Special function blocks for ICPDAS device are part of the library, e.g. PLCopen defined functions.

- Spylist: Enables a quick dynamic view on variables during debugging. Hint: In the Global Spylist, via the column headers, you can search for defined contents or sort the list entries ascending or descending.
- ENUM: List of user defined enumerated data types.
- Graphics: Lists of all available kinds of graphic objects.

Step 5: Output window:

Shows the compiler messages and if connected to the runtime all the messages generated by the runtime and the state of each task.

Step 6: Status bar:

At the bottom of the workspace is a bar, where you get additional information. The content depends on the selected area.

3.1 Customize Toolbar and Menus

Via the configuration dialog (Figure 2) the visible toolbar and the menus commands are selected.

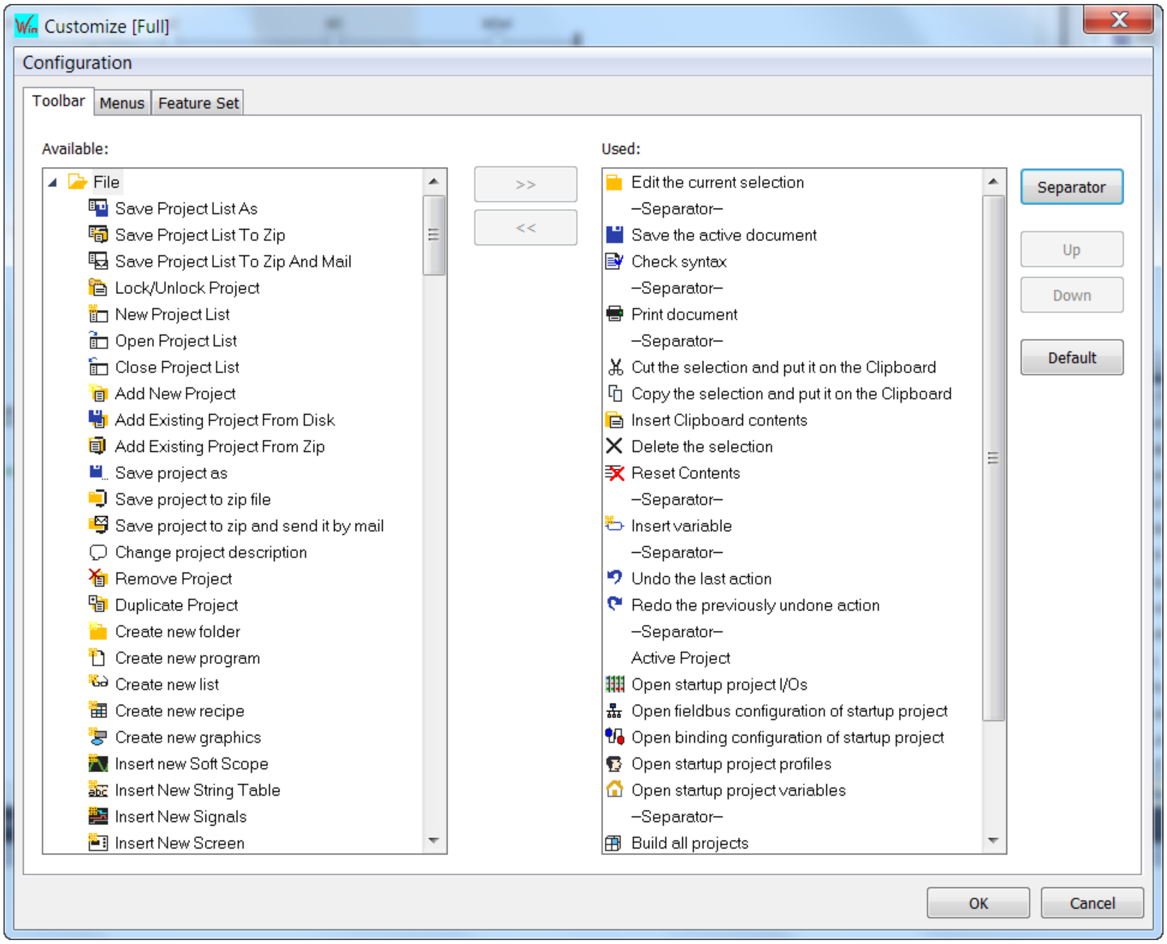
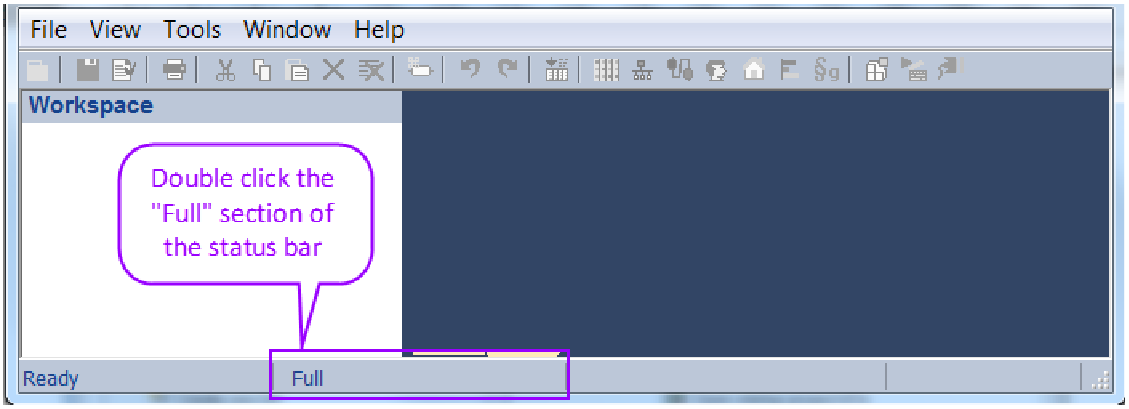


Figure 2: Toolbar and menu configuration window

The configuration dialog is opened by double clicking the *'Full'* section of the status bar (Figure 3)

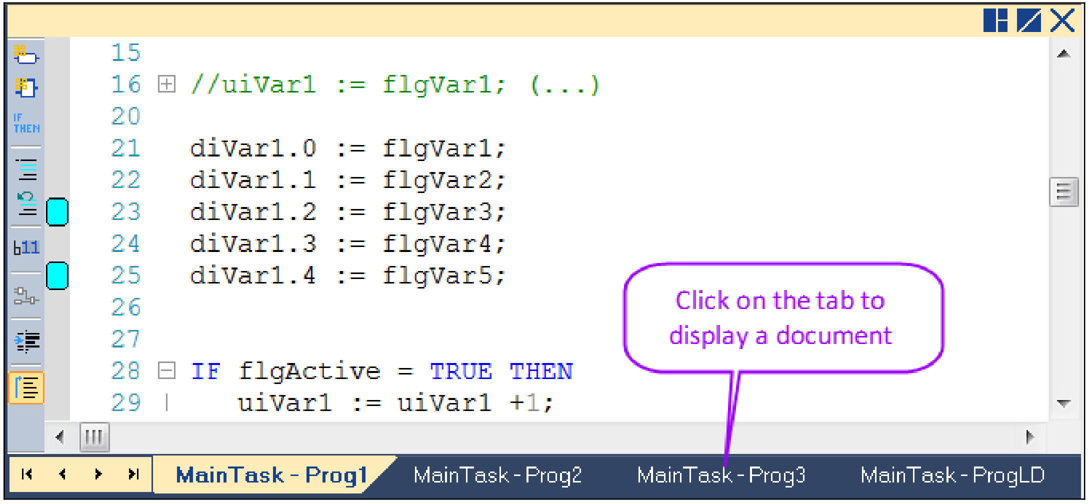


Double click the "Full" section of the status bar

Figure 3: Open the toolbar and menu configuration dialog

3.2 Main Window

In the *'Main window'* several documents can be opened at the same time. Use the tab control at the bottom of the area to display a document.



```
//uiVar1 := flgVar1; (...)

diVar1.0 := flgVar1;
diVar1.1 := flgVar2;
diVar1.2 := flgVar3;
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
```

Click on the tab to display a document

Figure 4: A separate tab is being created for each open document

Use the ✕ button in the title bar to close the active document.

The variable editor and document windows in the middle area can be maximized by clicking the ◧◪ button or double-click in their title bar.

When several documents are open in the middle area, you can lock one of them at the top or on the left of the area. For that, right click on the corresponding tab and select '*Lock*'. The same menu enables you at any time to unlock the document or lock another one:

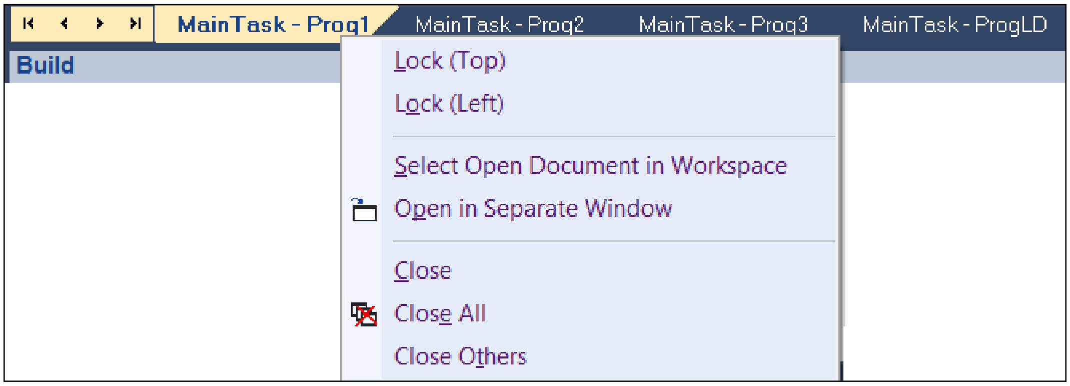


Figure 5: Lock the tab position of a document

3.3 The Workspace Window

The tasks name and it associated programs, fieldbus configuration, Spylist and global variables are shown in the workspace window in the left-hand window of the Workbench. The content of the items listed in the workspace are shown in the main window by double clicking the item. Figure 6 shows the general task items for single and multitask items.

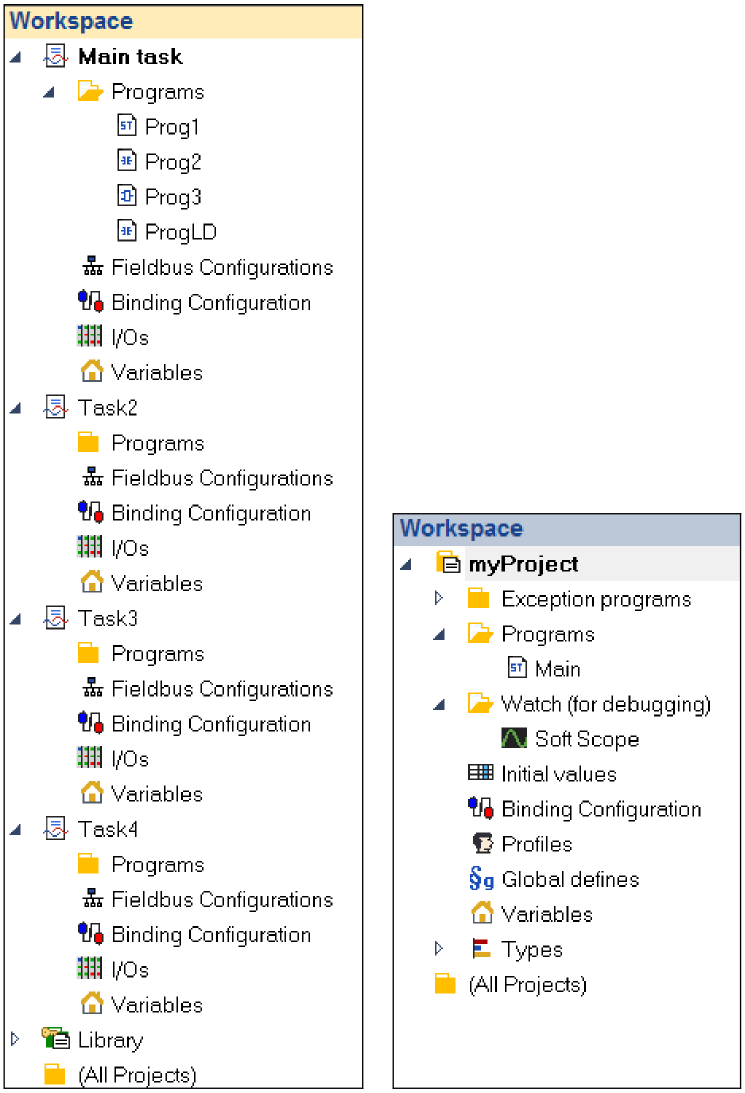


Figure 6: Workspace window with the task: multitask (left) and single task (right)

New items is added to the workspace window by right clicking the task name and selecting *'Insert new items...'* from the popup menu.

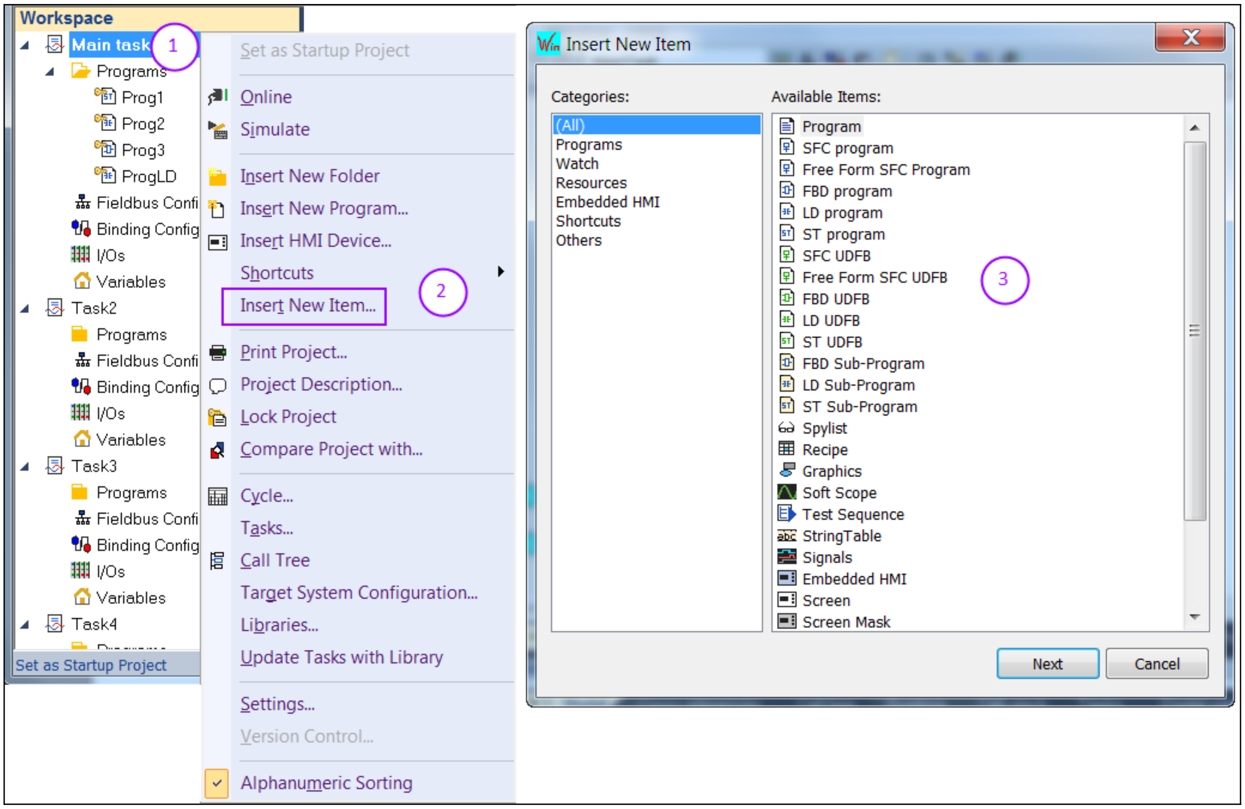


Figure 7: Adding new items to the workspace window

For each task the following items are available:

- Programs
- Recipe
- Signals
- Soft Scope
- Spy
- String Tables
- Fieldbus Configurations
- Binding Configurations
- Profiles
- Global defines
- Variables
- Types

The project workspace is stored in a file with the format *'.W5L'*. It basically stores the list of task folders and some configuration data. The workbench creates for each task a separated folder for storing the program source code, Fieldbus configuration, IO settings, etc.

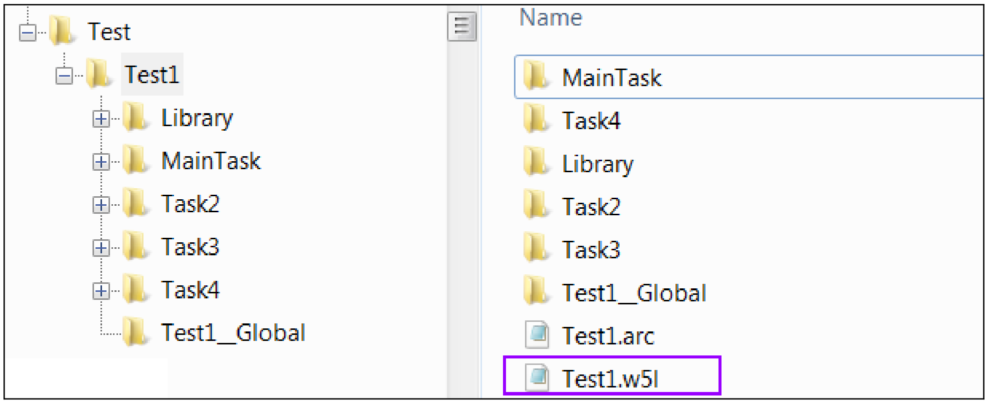


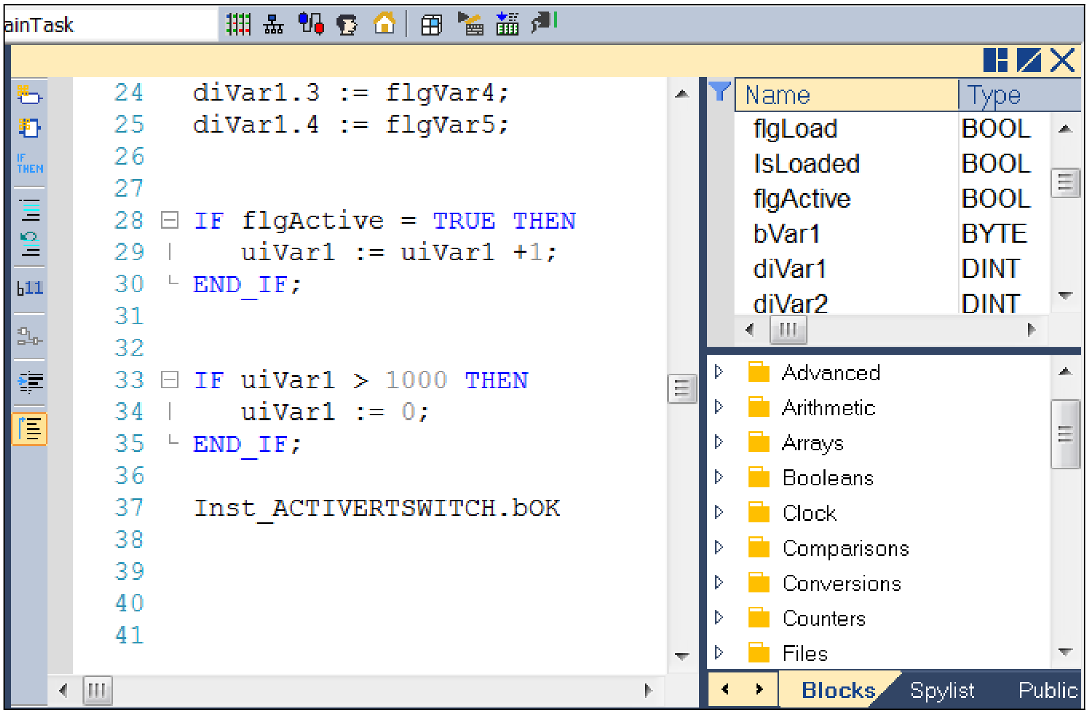
Figure 8: Project folder

Hint: It is possible, to copy items from task to another task within the workspace. This can be done by either selecting the copy command from the menu bar entries, using the shortcuts CTRL+C and CTRL+V or via drag & drop.

3.4 Program Editor

The programming environment provide editors for the following Soft PLC languages:

- Sequential Function Chart (SFC)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Structure Text (ST) and Instruction List (IL)



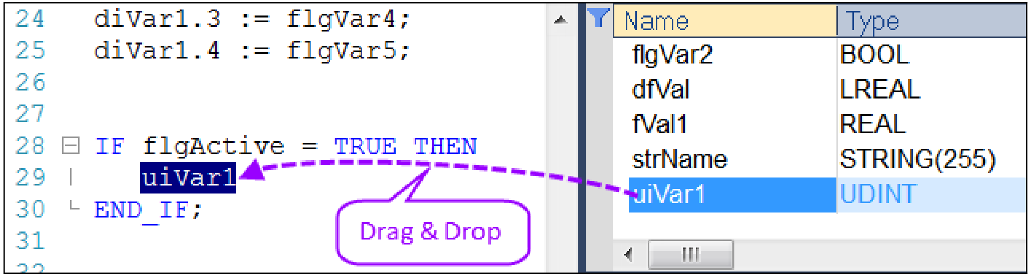
```
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
END_IF;


IF uiVar1 > 1000 THEN
    uiVar1 := 0;
END_IF;

Inst_ACTIVERTSWITCH.bOK
```

Name	Type
flgLoad	BOOL
IsLoaded	BOOL
flgActive	BOOL
bVar1	BYTE
diVar1	DINT
diVar2	DINT

Figure 9: Program editor for Structure Text

Variables, function blocks and definitions can be added to the editor via drag and drop as shown in the following examples:

- Drag a variable from the variable list to the program to insert it.

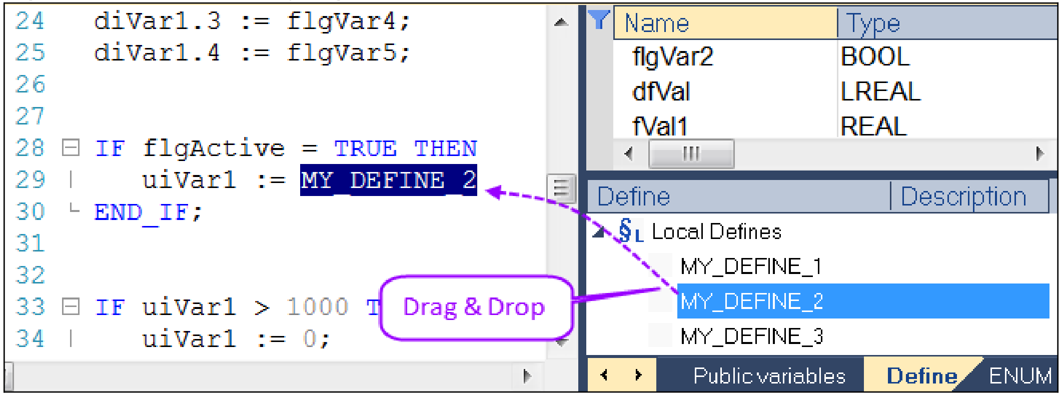


```
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1
END_IF;
```

Name	Type
flgVar2	BOOL
dfVal	LREAL
fVal1	REAL
strName	STRING(255)
uiVar1	UDINT

Figure 10: Drag variable from the variable editor to the program editor

- Drag a definition to the program to insert its name.



```
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1 := MY_DEFINE_2
END_IF;


IF uiVar1 > 1000 T
    uiVar1 := 0;
```

Name	Type
flgVar2	BOOL
dfVal	LREAL
fVal1	REAL

Define	Description
§L Local Defines	
MY_DEFINE_1	
MY_DEFINE_2	
MY_DEFINE_3	

Figure 11: Drag the definition name from the 'Define' list to the program editor

- Drag a block in the program to insert it.

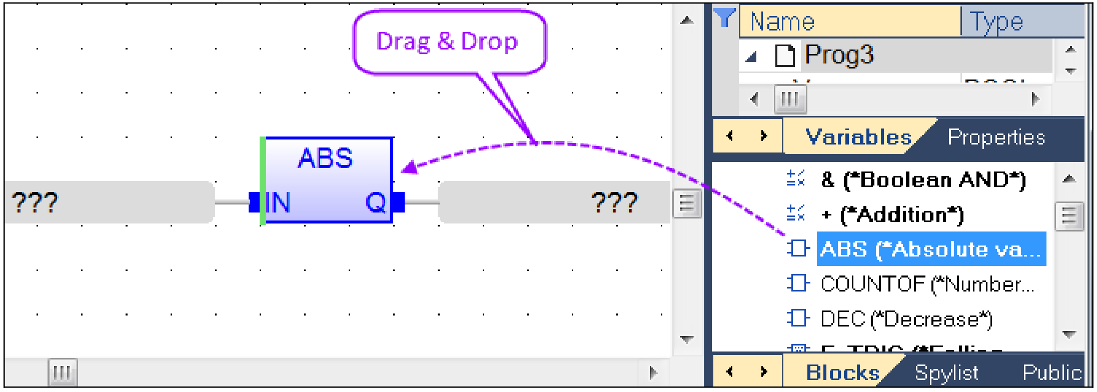
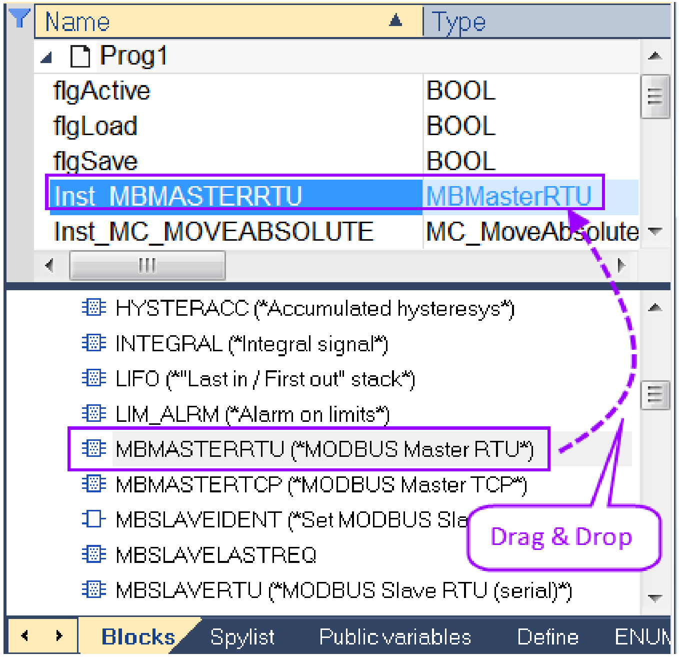


Figure 12: Drag function from the 'Blocks' list to the program editor

- Drag a function block to the variable list to declare an instance.

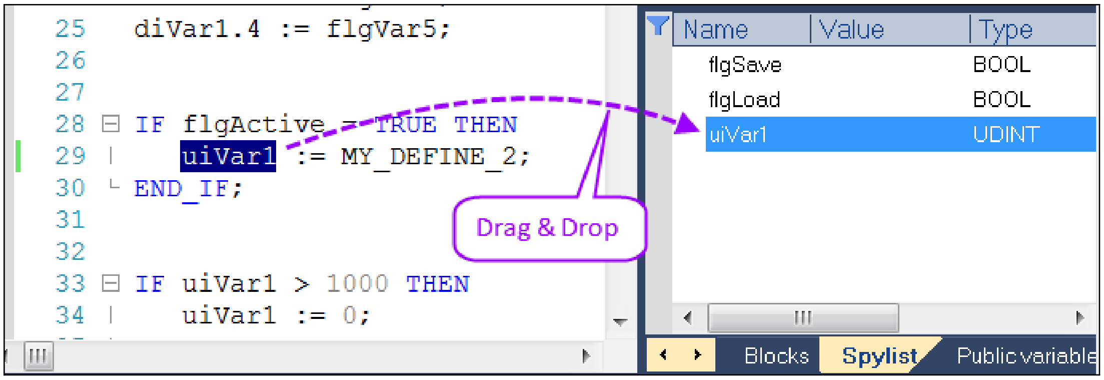


Name	Type
Prog1	
flgActive	BOOL
flgLoad	BOOL
flgSave	BOOL
Inst_MBMASTERRTU	MBMasterRTU
Inst_MC_MOVEABSOLUTE	MC_MoveAbsolute

HYSTERACC (*Accumulated hysteresys*)
INTEGRAL (*Integral signal*)
LIFO (*"Last in / First out" stack*)
LIM_ALRM (*Alarm on limits*)
MBMASTERRTU (*MODBUS Master RTU*)
MBMASTERTCP (*MODBUS Master TCP*)
MBSLAVEIDENT (*Set MODBUS Sla
MBSLAVELASTREQ
MBSLAVERTU (*MODBUS Slave RTU (serial)*)

Figure 13: Drag function block from the 'Blocks' list to the variable editor

- Drag a variable from the program or from the variable list to the spy list.



```
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1 := MY_DEFINE_2;
END_IF;


IF uiVar1 > 1000 THEN
    uiVar1 := 0;
```

Name	Value	Type
flgSave		BOOL
flgLoad		BOOL
uiVar1		UDINT

Figure 14:Drag variable from the program editor to the 'SpyList'

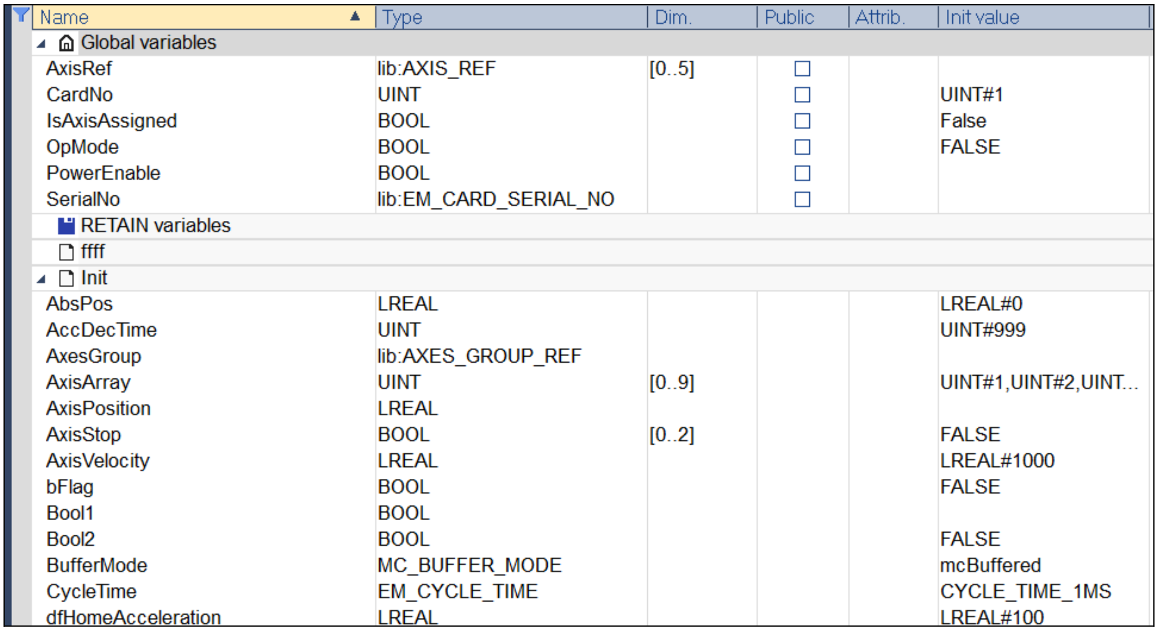
3.5 Variable Editor

Variables are declared in the top/right area of the Workbench main window. The variable editor is a grid tool that enables you to declare all variables of the application.

Variables in the editor are sorted by groups:

- Global variables.
- *'Retain'* non volatile global variables.
- I/O variables (each I/O device is a group).
- variables local to a program (including in and out parameters in case of a UDFB).

Each group is marked with a gray header in the variable list. The arrow icons ◢ ▷ on the left side of each group header can be used to expand or collapse the group:



Name ▲	Type	Dim.	Public	Attrib.	Init value
◢ Global variables					
AxisRef	lib:AXIS_REF	[0..5]	□		
CardNo	UINT		□		UINT#1
IsAxisAssigned	BOOL		□		False
OpMode	BOOL		□		FALSE
PowerEnable	BOOL		□		
SerialNo	lib:EM_CARD_SERIAL_NO		□		
RETAIN variables					
ffff					
◢ Init					
AbsPos	LREAL				LREAL#0
AccDecTime	UINT				UINT#999
AxesGroup	lib:AXES_GROUP_REF				
AxisArray	UINT	[0..9]			UINT#1,UINT#2,UINT...
AxisPosition	LREAL				
AxisStop	BOOL	[0..2]			FALSE
AxisVelocity	LREAL				LREAL#1000
bFlag	BOOL				FALSE
Bool1	BOOL				
Bool2	BOOL				FALSE
BufferMode	MC_BUFFER_MODE				mcBuffered
CycleTime	EM_CYCLE_TIME				CYCLE_TIME_1MS
dfHomeAcceleration	LREAL				LREAL#100

Figure 15: Variable editor

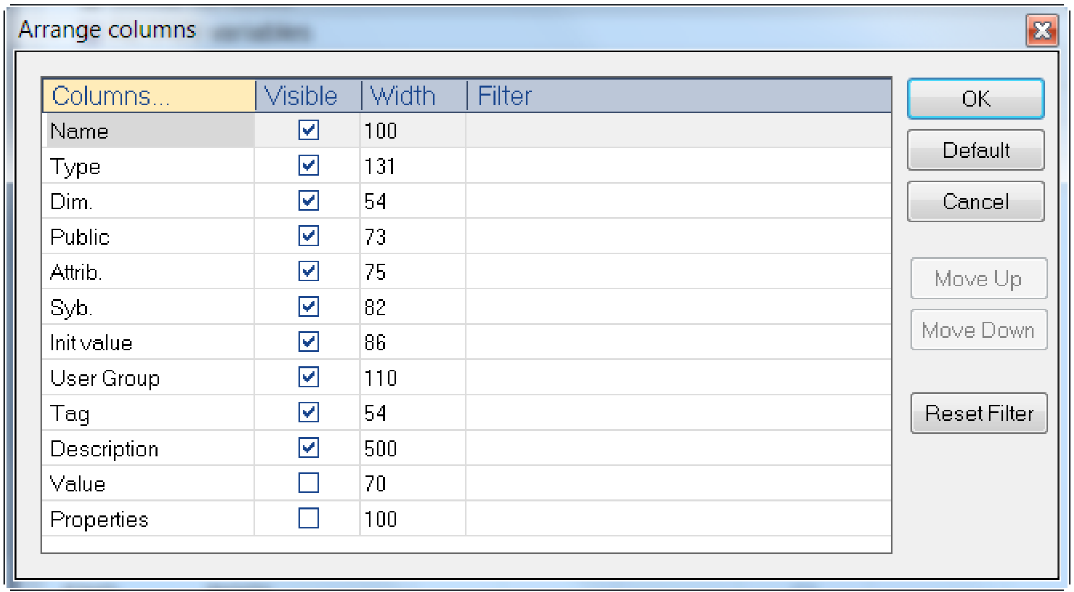
Double-click the header line enables you to sort, show or hide columns, and to apply a filter for each column. Filter is described as a text string that may contain *'?'* and *'*'* wild chars.

Each variable is described with:

- a name
- a data type and a dimension
- an attribute
- an initial value
- a tag and a description text
- OEM defined properties
- a user group

  The user group enables logical sorting of variables in the grid.

Columns of the variable editor can be rearranged or be set visible/invisible by double clicking the ▼ symbol in the editor or selecting the *'View\Columns...'* command from the dropdown menu (Figure 16).



Columns...	Visible	Width	Filter
Name	☑	100	
Type	☑	131	
Dim.	☑	54	
Public	☑	73	
Attrib.	☑	75	
Syb.	☑	82	
Init value	☑	86	
User Group	☑	110	
Tag	☑	54	
Description	☑	500	
Value	☐	70	
Properties	☐	100	

Figure 16: Variable editor column setting

3.6 Output Window View

The Output window contains the following tabs:

- Build
- Cross references

- Call tree
- Runtime
- Call stack
- Breakpoints
- Digital sampling trace
- Prompt (not supported)
- HMI (not supported)
- Code Checker

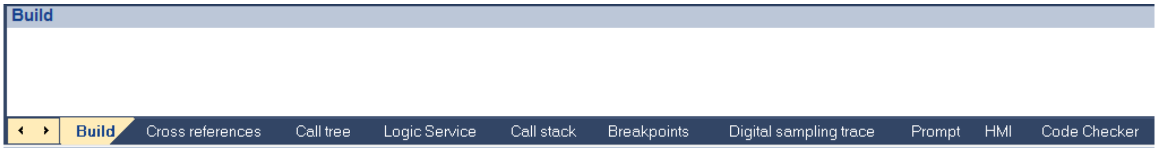
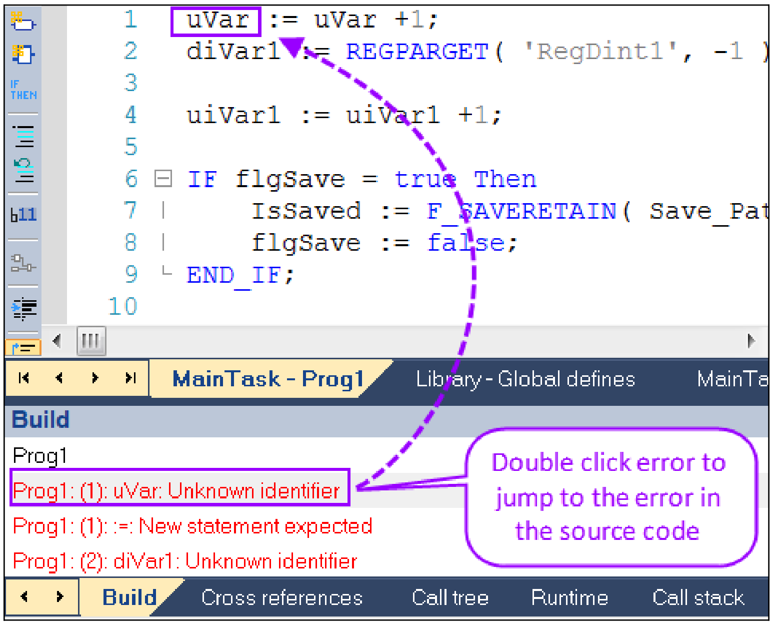


Figure 17: Output window

3.6.1 Build Output

The compiler reports messages in the build output window. If compiling errors occur, just double-click on a error line in the output window to open the position in the program code where the error occurred.



```
uVar := uVar +1;
diVar1 := REGPARGET( 'RegDint1', -1 )

uiVar1 := uiVar1 +1;

IF flgSave = true Then
    IsSaved := F_SAVERETAIN( Save_Pat
    flgSave := false;
END_IF;
```

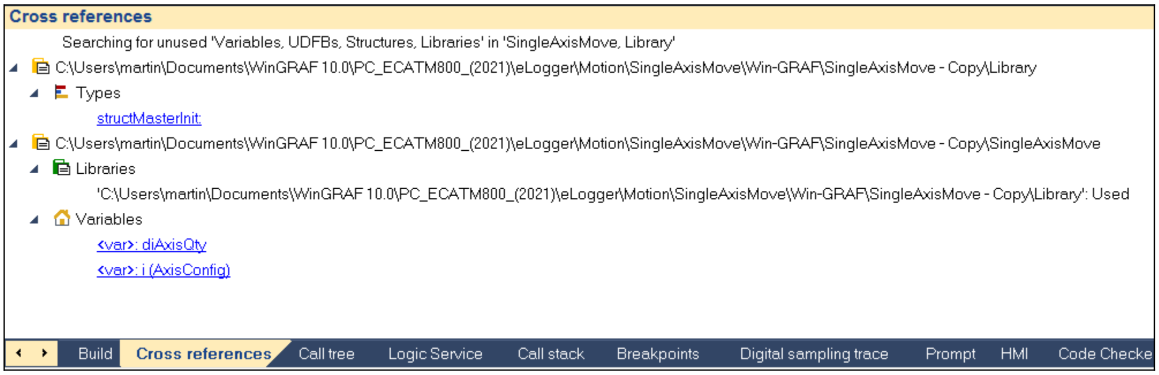
```
Prog1
Prog1: (1): uVar: Unknown identifier
Prog1: (1): :=: New statement expected
Prog1: (2): diVar1: Unknown identifier
```

Figure 18: Compiling error messages in the output window

3.6.2 Cross References

The Cross reference view allows the user to select and display one of the following information:

- Find or replace names, variables, etc.
- List unused items: It list declared variables and function blocks instance which has been declared but are not being used by the program
- List multiple variable assignments



```
Cross references
        Searching for unused 'Variables, UDFBs, Structures, Libraries' in 'SingleAxisMove, Library'
◢ C:\Users\martin\Documents\WinGRAF 10.0\PC_ECATM800_(2021)\eLogger\Motion\SingleAxisMove\Win-GRAF\SingleAxisMove - Copy\Library
   ◢ Types
         structMasterInit:
◢ C:\Users\martin\Documents\WinGRAF 10.0\PC_ECATM800_(2021)\eLogger\Motion\SingleAxisMove\Win-GRAF\SingleAxisMove - Copy\SingleAxisMove
   ◢ Libraries
         'C:\Users\martin\Documents\WinGRAF 10.0\PC_ECATM800_(2021)\eLogger\Motion\SingleAxisMove\Win-GRAF\SingleAxisMove - Copy\Library': Used
   ◢ Variables
         <var>: diAxisQty
         <var>: i (AxisConfig)
```

Figure 19: Cross reference (Example: list all unused variables)

The Cross Reference tools enables the search for a specific variables in the application. It can also be used as a powerful navigation tool for editing changes in the application programs.

The Cross Reference tools can be used via:

- the menu bar entries *'Edit/Find...'*
- the Cross references context menu in the Output window (Figure 20).

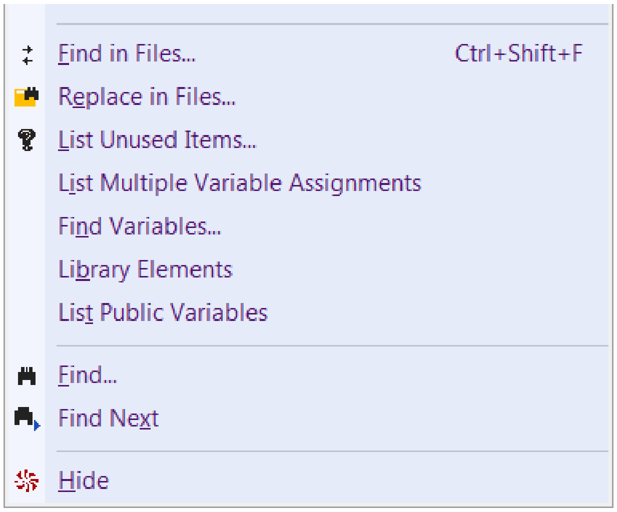
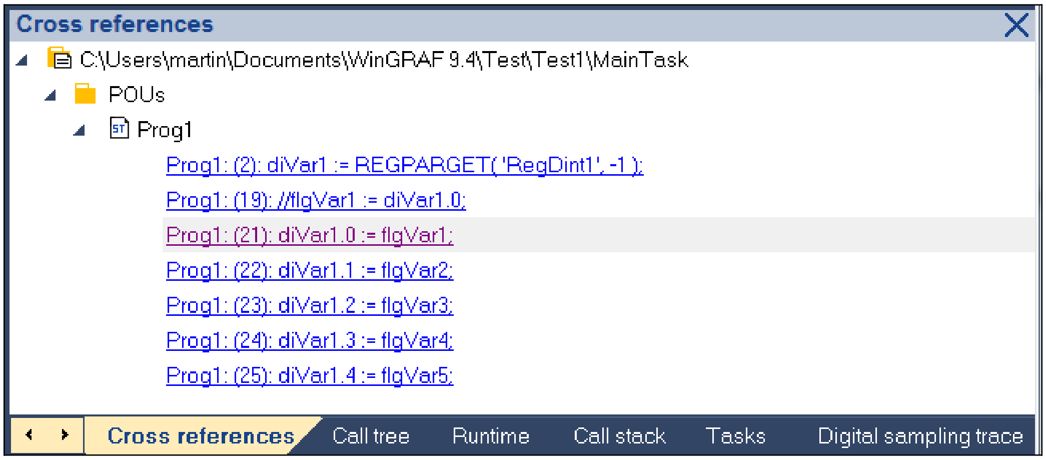


Figure 20: Cross reference commands in a popup menu

3.6.2.1 Search Text

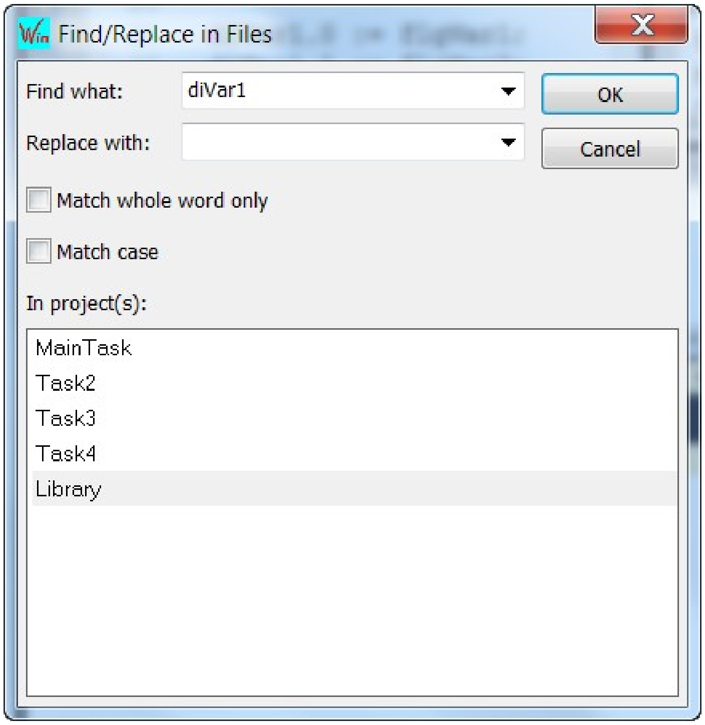
The find commands allows you to search for a text (name, variable, function, block, etc.) in all the programs of the project and list the search result in the Cross reference output window (Figure 21). By double clicking one of the items in the search list the program will be open at the position where the text occurred.



```
Prog1: (2): diVar1 := REGPARGET( 'RegDint1', -1 );
Prog1: (19): //flgVar1 := diVar1.0;
Prog1: (21): diVar1.0 := flgVar1;
Prog1: (22): diVar1.1 := flgVar2;
Prog1: (23): diVar1.2 := flgVar3;
Prog1: (24): diVar1.3 := flgVar4;
Prog1: (25): diVar1.4 := flgVar5;
```

Figure 21: List of found names

Use the '*Edit / Find / Find in Files...*' menu command or right click the Cross reference output and select the *'Find in Files...'*command from the popup menu to search for a text in all programs. Enter the search text in the popup window (Figure 22).



Find what: diVar1

Replace with:

□ Match whole word only

□ Match case

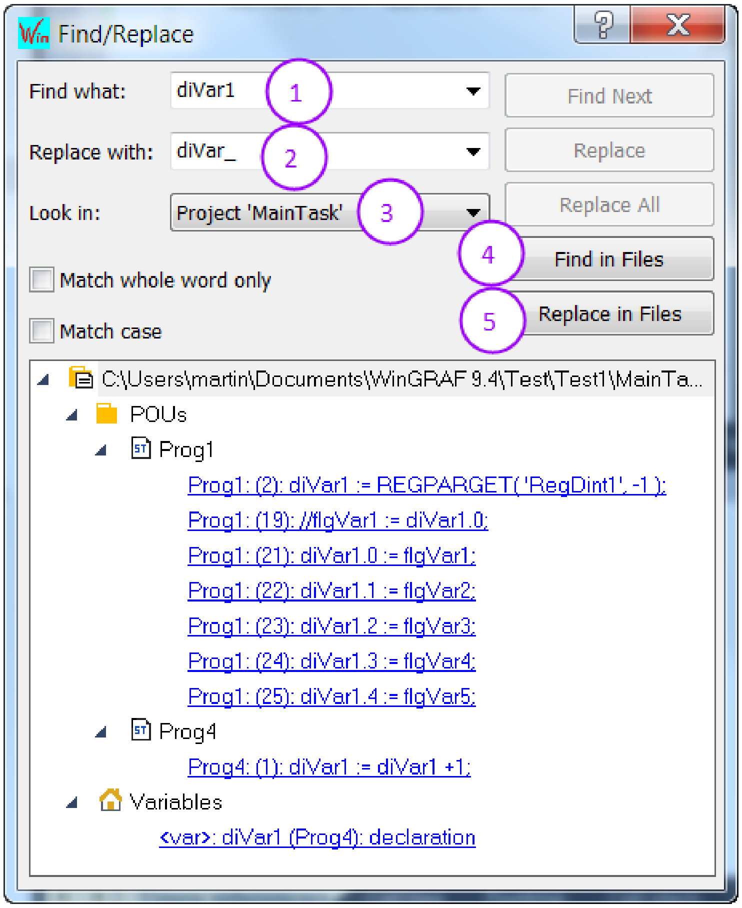
In project(s):

MainTask
Task2
Task3
Task4
Library

Figure 22: Find in all files

3.6.2.2 *Find / Replace Text*

The '*Edit / Replace in files...*' command enables you to replace a text in all the programs of the application.



Find what: diVar1

Replace with: diVar_

Look in: Project 'MainTask'

□ Match whole word only

□ Match case

```
C:\Users\martin\Documents\WinGRAF 9.4\Test\Test1\MainTa...
  POUs
    Prog1
      Prog1: (2): diVar1 := REGPARGET( 'RegDint1', -1 );
      Prog1: (19): //flgVar1 := diVar1.0;
      Prog1: (21): diVar1.0 := flgVar1;
      Prog1: (22): diVar1.1 := flgVar2;
      Prog1: (23): diVar1.2 := flgVar3;
      Prog1: (24): diVar1.3 := flgVar4;
      Prog1: (25): diVar1.4 := flgVar5;
    Prog4
      Prog4: (1): diVar1 := diVar1 +1;
  Variables
    <var>: diVar1 (Prog4): declaration
```

Figure 23: Replace text window

Procedure for replacing text:

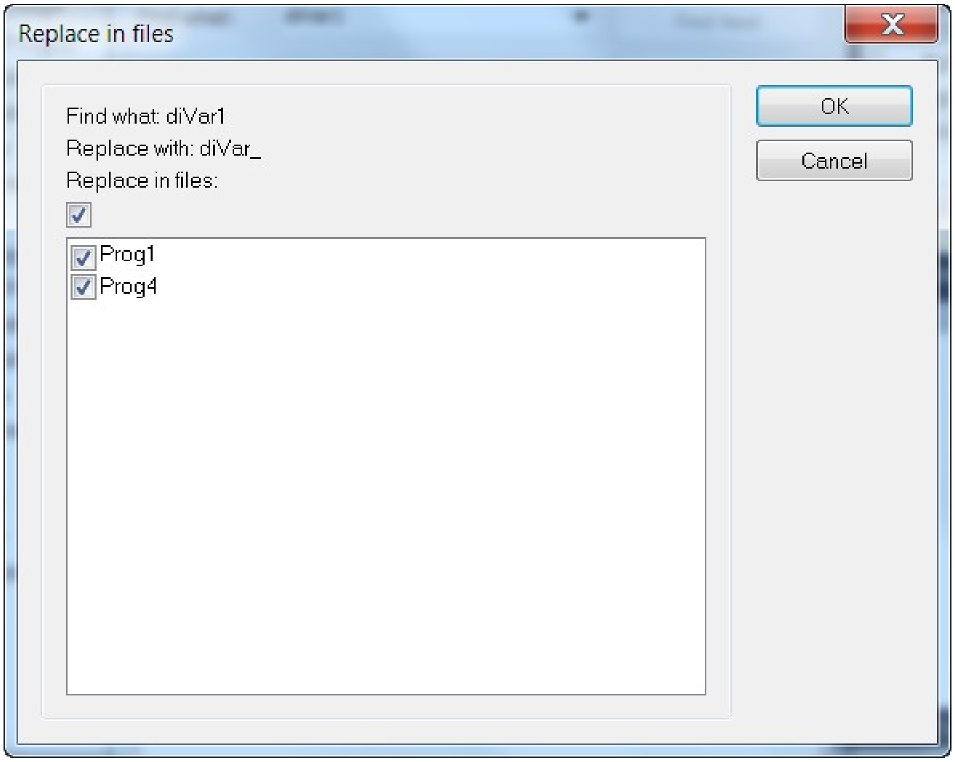
Step 1: Enter the text which needs to be replaced in the *'Find what'* editor

Step 2: Enter the new text in the *'Replace with'* editor

Step 3: Select the task where to replace the text

Step 4: Optional: Click *'Find in Files'* button to display all the location at which the text occurs. It scans all the programs in the task and list where the specified text occurs.

Step 5: Click *'Replace in Files'* button. A window pops up which allows you to select in which programs of the task to replace the text (Figure 24). Click *'OK'* to replace the text.



Find what: diVar1

Replace with: diVar_

Replace in files:

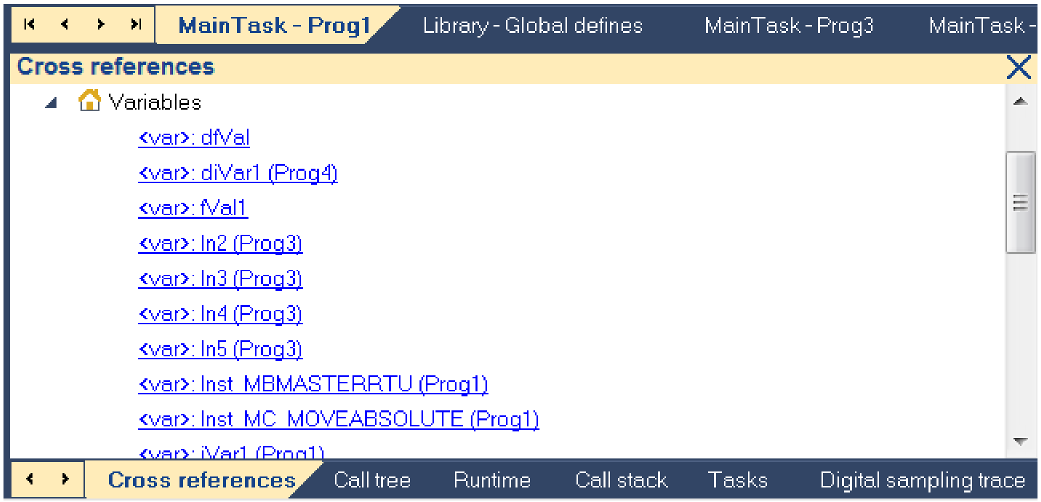
☑

☑ Prog1
☑ Prog4

Figure 24: Select the programs where to find or replace a text

3.6.2.3 List Unused Variables

Use the '*Edit / Find / List Unused Variables*' command or right click the output window and select *'List Unused items...'* to display the list of declared variables and function block instances that are not used in the programs of the task. This command is particularly useful for removing unused variables from a project.



Cross references

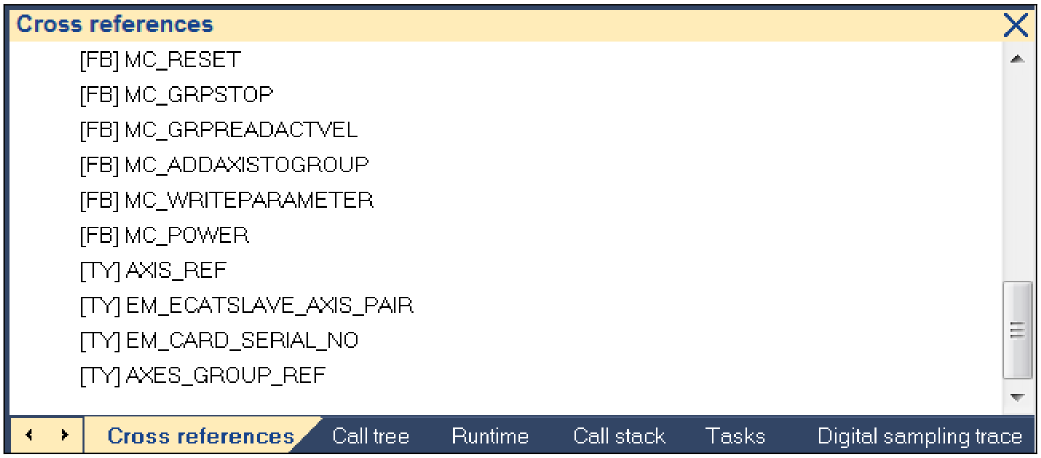
Variables

<var>: dfVal
<var>: diVar1 (Prog4)
<var>: fVal1
<var>: In2 (Prog3)
<var>: In3 (Prog3)
<var>: In4 (Prog3)
<var>: In5 (Prog3)
<var>: Inst_MBMASTERRTU (Prog1)
<var>: Inst_MC_MOVEABSOLUTE (Prog1)
<var>: iVar1 (Prog1)

Figure 25: List of unused variables and instances

3.6.2.4 List OEM Library Elements

Use the '*Edit / Find / OEM Library Elements*' command to list the I/O devices, functions and function blocks written in *'C'* that are used in your application.



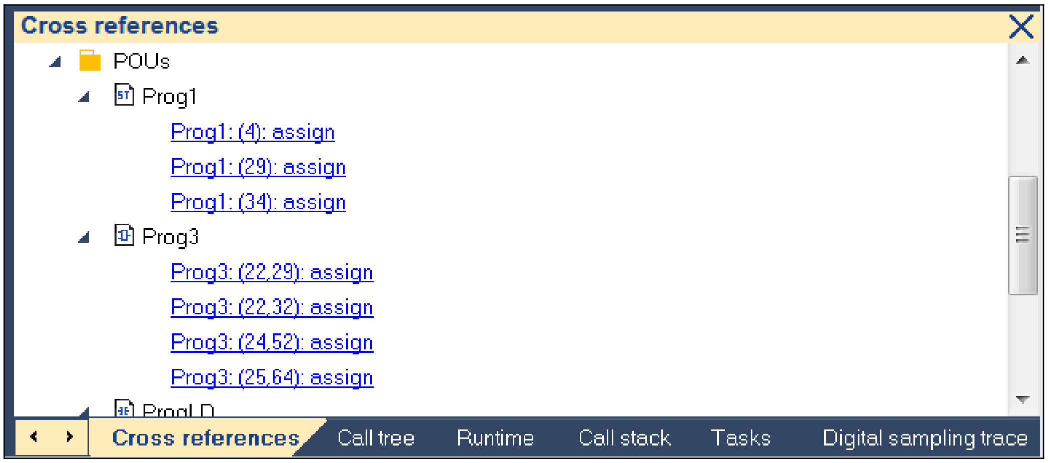
Cross references

[FB] MC_RESET
[FB] MC_GRPSTOP
[FB] MC_GRPREADACTVEL
[FB] MC_ADDAXISTOGROUP
[FB] MC_WRITEPARAMETER
[FB] MC_POWER
[TY] AXIS_REF
[TY] EM_ECATSLAVE_AXIS_PAIR
[TY] EM_CARD_SERIAL_NO
[TY] AXES_GROUP_REF

Figure 26: OEM function blocks and enumerates used by the application

3.6.2.5 List Multiple Variable Assignments

This output list all the variables which are assigned a value multiple times in the program. This function allows the user to check whether variable assignments are correct. Use the '*Edit / Find / List Multiple Variable Assignments*' command to scan all programs in a task for more than one variable assignment

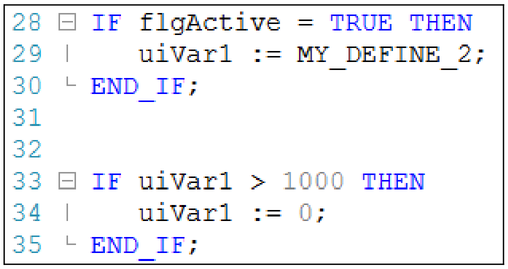


Cross references

POUs
Prog1
Prog1: (4): assign
Prog1: (29): assign
Prog1: (34): assign
Prog3
Prog3: (22,29): assign
Prog3: (22,32): assign
Prog3: (24,52): assign
Prog3: (25,64): assign
ProgLD

Figure 27: List of multiple variable assignment

Example:
The variable *'uiVar1'* in the program (Figure 28) is assigned a value in line 29 and 34. Therefore Cross reference view will list both line number.

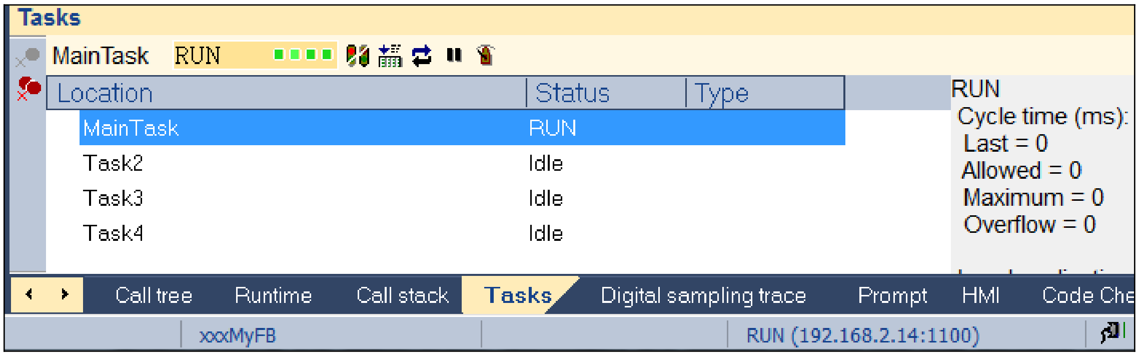


```
28  IF flgActive = TRUE THEN
29      uiVar1 := MY_DEFINE_2;
30  END_IF;
31
32
33  IF uiVar1 > 1000 THEN
34      uiVar1 := 0;
35  END_IF;
```

Figure 28: Multiple variable assignment

3.6.3 Task Status Output

The task output shows the status and the mode of all the tasks when the workbench is online. The status of a task can be changed by selecting the task in the output window and clicking on one of the command buttons in the output toolbar.



Tasks

MainTask RUN

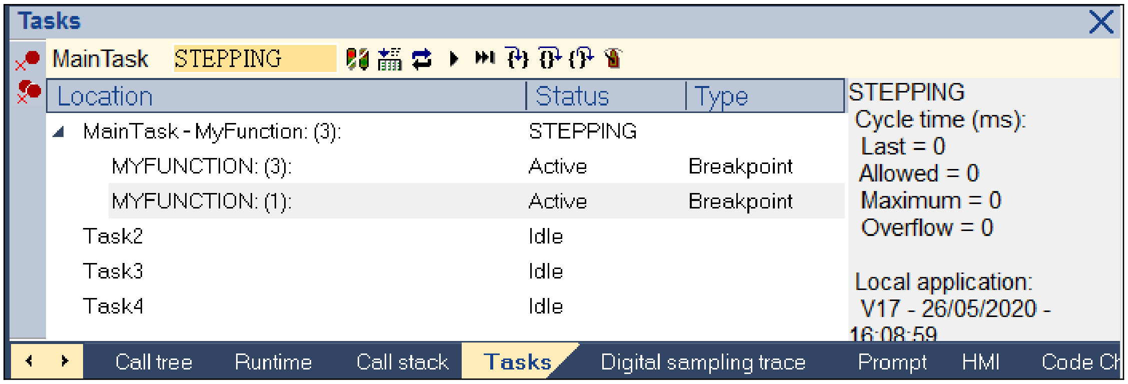
Location	Status	Type
MainTask	RUN	
Task2	Idle	
Task3	Idle	
Task4	Idle	

RUN
Cycle time (ms):
Last = 0
Allowed = 0
Maximum = 0
Overflow = 0

xxxMyFB
RUN (192.168.2.14:1100)

Figure 29: Task status

List of breakpoints are shown in the task output window and whether the program is in step by step execution mode (Figure 30). Breakpoints can be directly removed via the task output window by first selecting the break point in the output and clicking ●. Clicking the ● command removes all the breakpoints.



Tasks

MainTask STEPPING

Location	Status	Type
MainTask - MyFunction: (3):	STEPPING	
MYFUNCTION: (3):	Active	Breakpoint
MYFUNCTION: (1):	Active	Breakpoint
Task2	Idle	
Task3	Idle	
Task4	Idle	

STEPPING
Cycle time (ms):
Last = 0
Allowed = 0
Maximum = 0
Overflow = 0

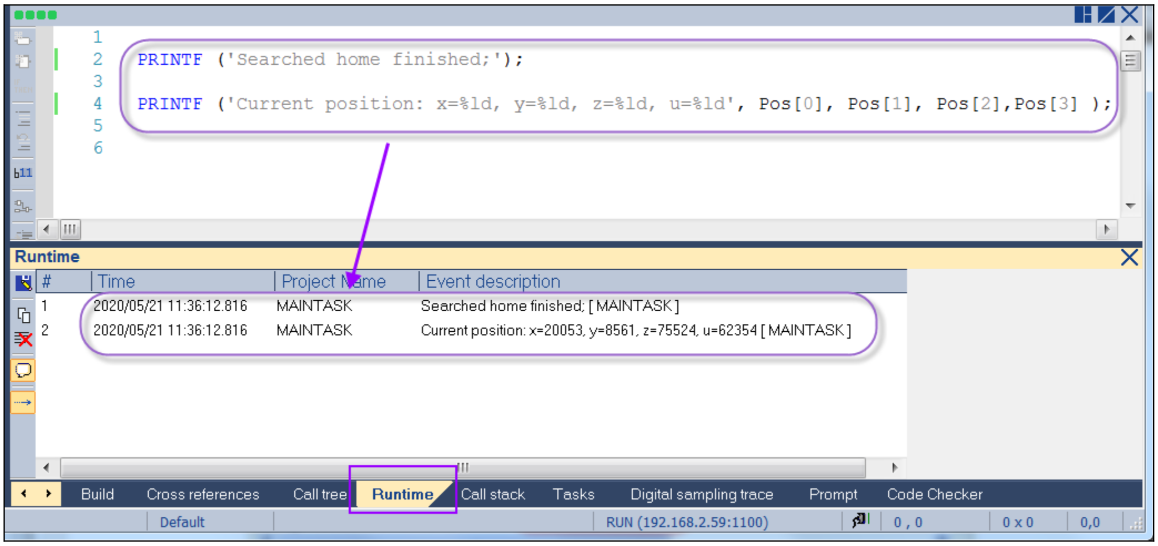
Local application:
V17 - 26/05/2020 -
16:08:59

Figure 30: Task with breakpoints

3.6.4 Runtime Messages

The Log window displays all runtime messages sent by the connected Soft PLC platform or by the simulator when testing the application. Messages are stored even if the Log window is not open.

Example:
The PRINTF function can be used to print messages from the runtime to the workbench runtime output:



```
PRINTF ('Searched home finished;');

PRINTF ('Current position: x=%ld, y=%ld, z=%ld, u=%ld', Pos[0], Pos[1], Pos[2],Pos[3] );
```

Runtime

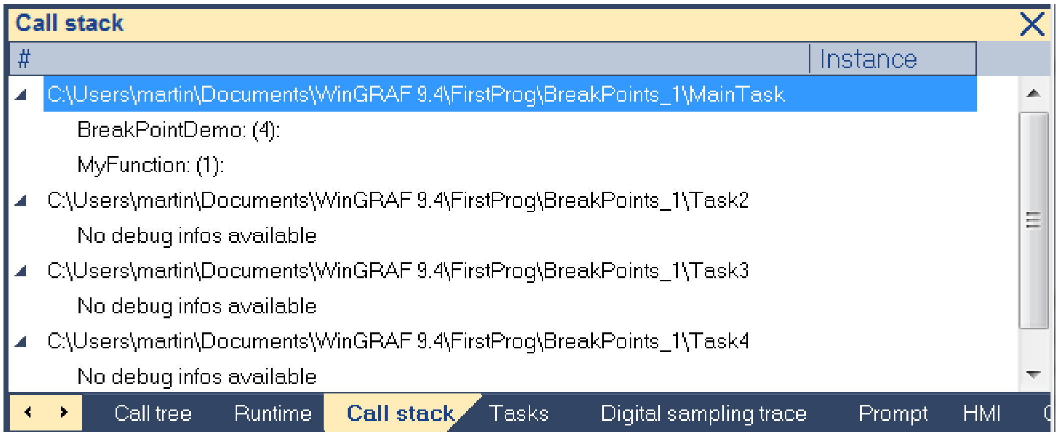
#	Time	Project Name	Event description
1	2020/05/21 11:36:12.816	MAINTASK	Searched home finished; [MAINTASK]
2	2020/05/21 11:36:12.816	MAINTASK	Current position: x=20053, y=8561, z=75524, u=62354 [MAINTASK]

Figure 31: PRINTF messages shown in the workbench runtime output

3.6.5 Call Stack View

During step by step debugging, the Call stack window shows the current call stacks.

When the workbench is in debugging mode, the Call stack view shows at which breakpoint the program has stopped. This function is only available if the application program is build in debug mode.


Call stack

#	Instance
C:\Users\martin\Documents\WinGRAF 9.4\FirstProg\BreakPoints_1\MainTask	
BreakPointDemo: (4):	
MyFunction: (1):	
C:\Users\martin\Documents\WinGRAF 9.4\FirstProg\BreakPoints_1\Task2	
No debug infos available	
C:\Users\martin\Documents\WinGRAF 9.4\FirstProg\BreakPoints_1\Task3	
No debug infos available	
C:\Users\martin\Documents\WinGRAF 9.4\FirstProg\BreakPoints_1\Task4	
No debug infos available	

Figure 32: Call stack view

3.6.6 Call Tree View

The Call Tree shows graphically the interdependency of the different programs in the project. It for example shows which program is being called by other programs. Table 1 list the different commands provided for the Call Tree.

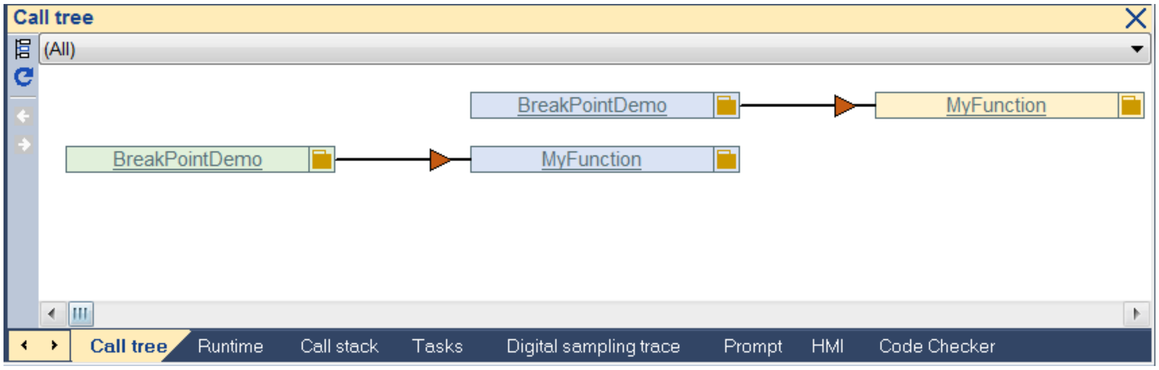


Figure 33: Call tree view

Icon	Command	Description
☰	Call Tree	Shows the Call Tree entries.
↻	Refresh	Refreshes the Call Tree view.
←	Backward	Jumps to the last Call Tree entry.
→	Forward	Jumps to the next Call Tree entry.

Table 1: Call tree view commands

3.6.7 Digital Sampling Trace

The runtime system includes a digital sampling trace recorder. The recorder is used to register periodically the state of up to 8 Boolean variables. Samples can be registered either on each cycle or according to a configurable period. The digital sampling trace is a useful tool for tracking Boolean events in the runtime application.

The sampling trace can be configured and watched from the Output window. The sampling trace is available only during simulation or on line debugging.

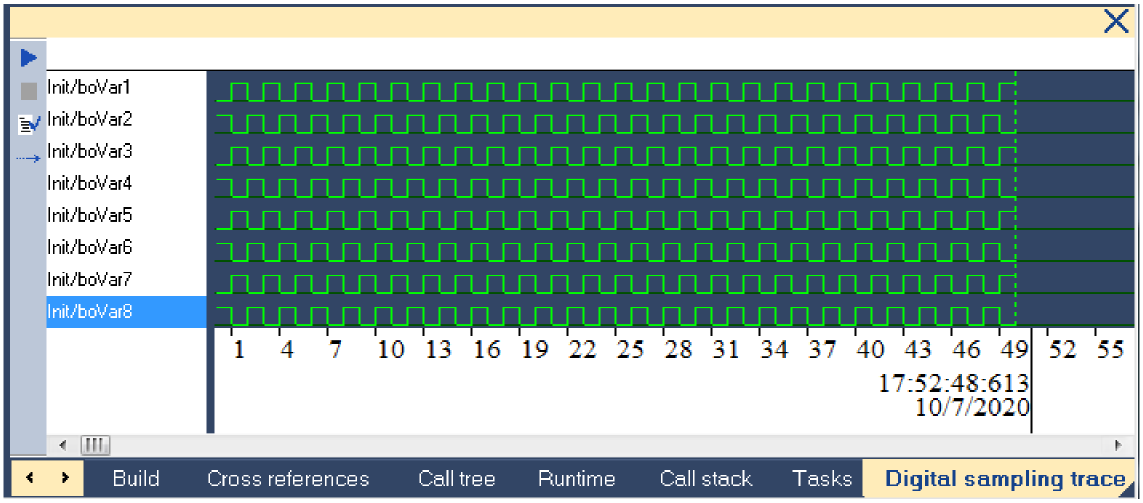


Figure 34: Output of eight Boolean variables per cycle

Attention

- The digital sampling trace is a unique resource of the runtime system. The settings of the recorder are the same for all recorded variables.
- The recording is limited to 900 samples of up to 8 BOOL variables.
- The recording of the sampling trace is time consuming and may slow down the performances of the runtime system.

Operations

Use the following commands for the Digital Sampling Trace operation:

Icon	Command	Description
▶	Start sampling	Start recording.
■	Stop sampling	Stops recording.

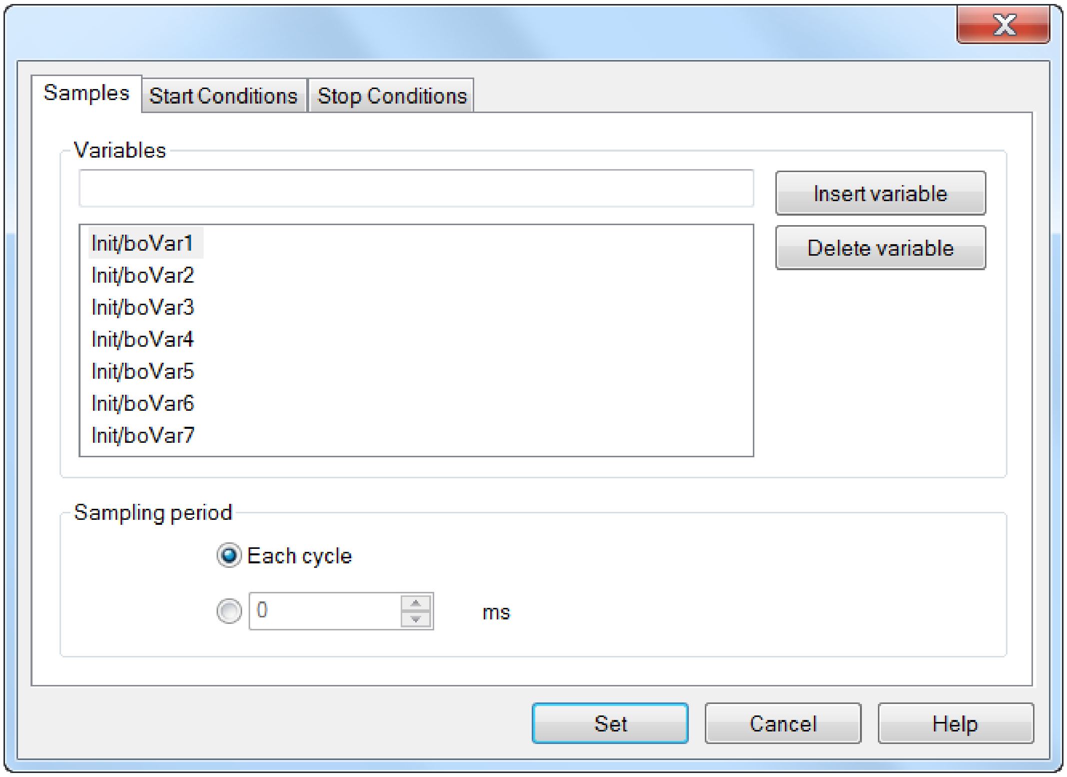
	Setup sampling	Define the variables and the settings of the sampling trace.
→	Autoscroll	Set or reset the auto-scroll mode.

Table 2: Digital sampling trace commands

3.6.7.1 Samples and Sampling Period Settings

Before starting a recording, you need to setup the parameters for the recorder in the Setup Sampling dialog (Figure 65). This includes the list of spied variables, a period (either a time or on each cycle), plus start and stop conditions. All variables must have the BOOL data type.

Note: The sampling period value indicate that the runtime wait at least the defined time interval before recording the next state. The status recording is not synchronized to the time interval which means the status is not recorded at the set time interval.



Variables

Init/boVar1
Init/boVar2
Init/boVar3
Init/boVar4
Init/boVar5
Init/boVar6
Init/boVar7

Sampling period

● Each cycle
○ 0 ms

Figure 35: Digital trace Boolean variable and sampling period

Parameters	Command	Description
Sampling Variables	Insert variable	Opens the dialog for variable selection. Select the desired variable an click on OK. The variable name will be shown in the Insert variable text field and in the Delete variable text field. Otherwise an error message occurs.

Parameters	Command	Description
	Delete variable	Removes the selected variable from the Delete variable text field.
Sampling Period	Each cycle	Sampling is done each cycle.
	Time in ms	Sampling is done in the selected time interval. Default: 0

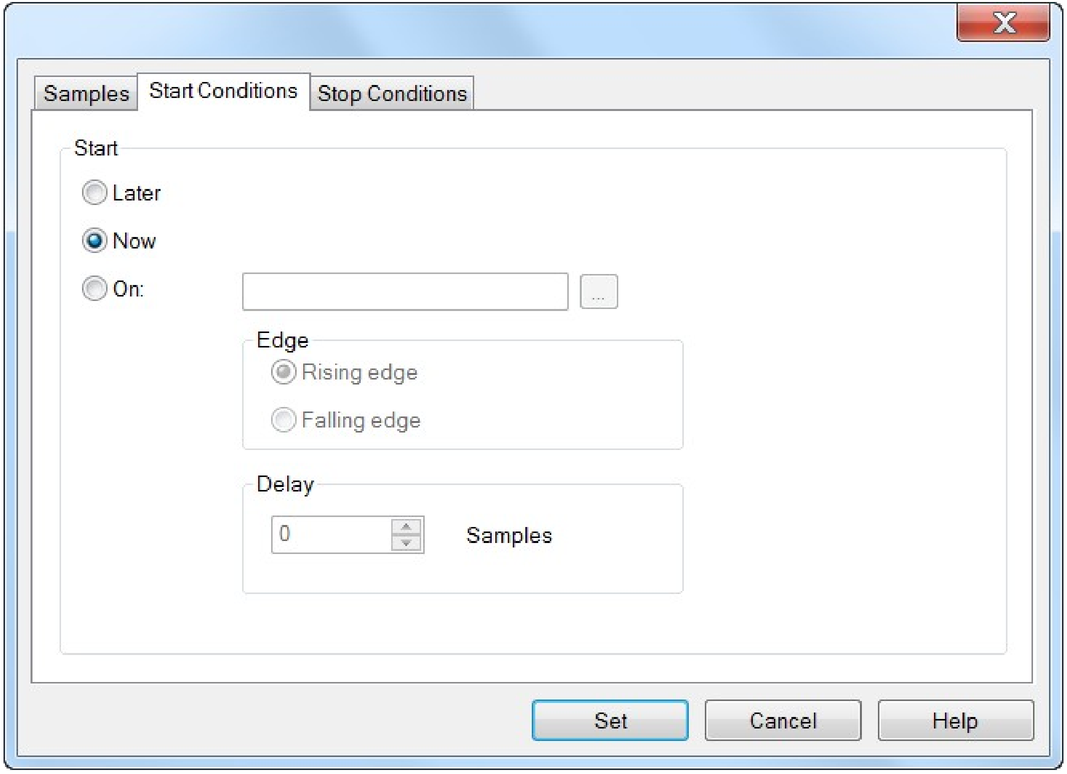
Table 3: Sample variable and period configuration interface

3.6.7.2 Start condition

The Start Condition tab of the settings box (Figure 36) enables you to define which condition will start the recording. The following choices are available:

- Later: you will have to manually start the recorder using the Start command.
- Now: tracing starts immediately after the *'Set'* has been clicked.
- On: A Boolean variable triggers the start of digital trace recording. Click on the *'...'* button to select the triggering variable. You can select the trigger condition: the rising or falling edge of the Boolean variable and the trigger delay.

The delay is expressed as a number of samples omitted after the start condition.



Start
○ Later
● Now
○ On:

Edge
● Rising edge
○ Falling edge

Delay
0 Samples

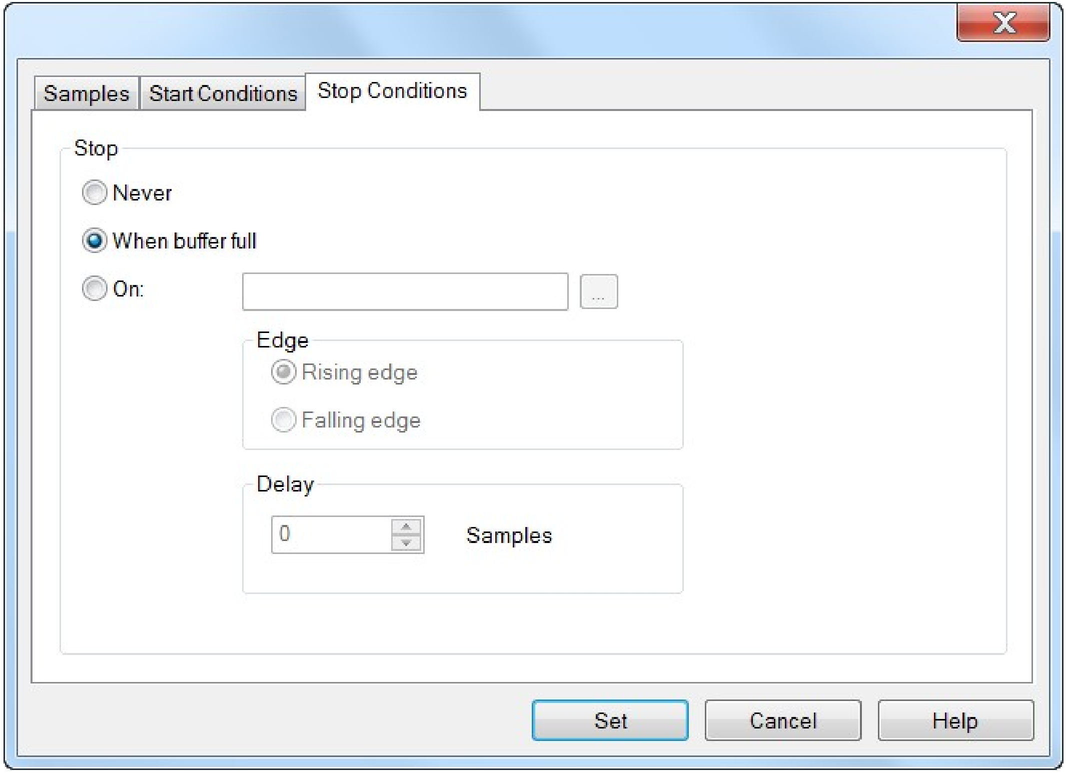
Figure 36: Digital trace start condition

3.6.7.3 Stop condition

The Stop Condition tab of the settings box (Figure 37) enables you to define which condition will stop the recording. The following choices are available:

- Never: you will have to manually stop the recorder using the Stop command.
- When the buffer is full.
- On the rising or falling edge of a BOOL variable, possibly with a delay.

The delay is expressed as a number of samples passed after the stop condition, before the recording actually stops.



Stop

○ Never

● When buffer full

○ On: ...

Edge

● Rising edge

○ Falling edge

Delay

0 Samples

Figure 37: Digital trace stop conditions

Remarks

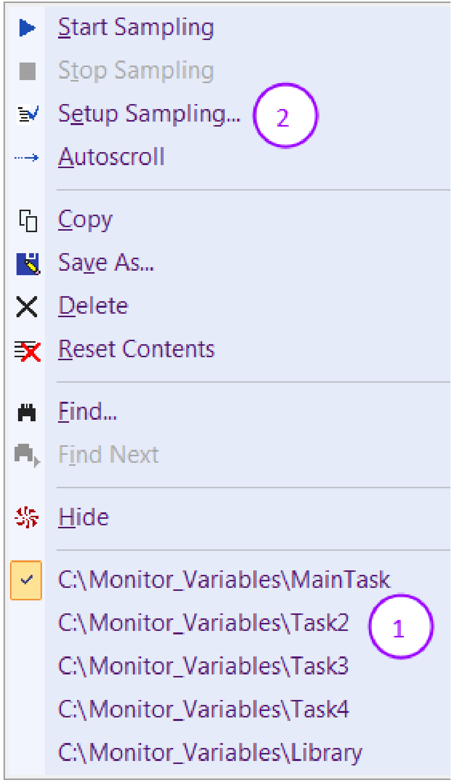
- The recorder cannot be restarted after points have been registered, even if stopped. To restart the recording, you first have to re-validate the settings.
- The sampling trace must be configured or started when the Workbench is used either for simulation or on line debugging.
- Use the File / Save As and Edit / Copy commands for exchanging recorded data with other applications such as spreadsheets.

Setting procedure:

Step 1: Download the program to the runtime and set the workbench in online mode.

Step 2: Select the task and open the configuration window.

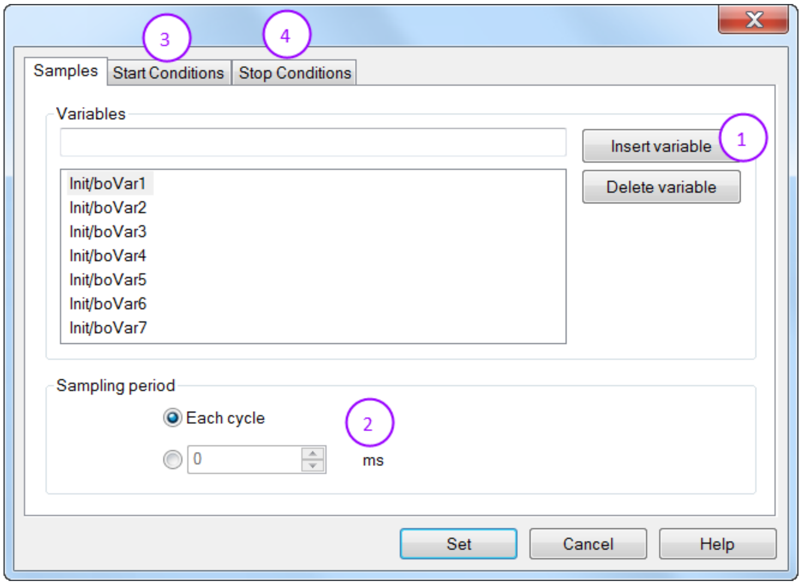
1. Right click the output window select the task to monitor
2. Click the command to open trace configuration window.


Start Sampling
Stop Sampling
Setup Sampling...
2
Autoscroll
Copy
Save As...
Delete
Reset Contents
Find...
Find Next
Hide
C:\Monitor_Variables\MainTask
C:\Monitor_Variables\Task2
1
C:\Monitor_Variables\Task3
C:\Monitor_Variables\Task4
C:\Monitor_Variables\Library

3. Optional: Select the *'Reset Contents'* command from the popup menu to clear the output window content.

Step 3: Set the digital sampling trace conditions.

1. Select the Booleans variable to trace.
2. Set the sampling period.
3. Set the start conditions.
4. Set the stop conditions.
5. Press the *'Set'* button to validate the settings.



Step 4: Start sampling by pressing the ▶ button in the output window.

3.6.8 Code Checker

The Code Checker tool performs a scan of the project declarations and programs, in order to check conformity to a set of rules, motivated by integrity, safety and portability of the code. It is run from the *'Code Checker'* tab of the Output window.

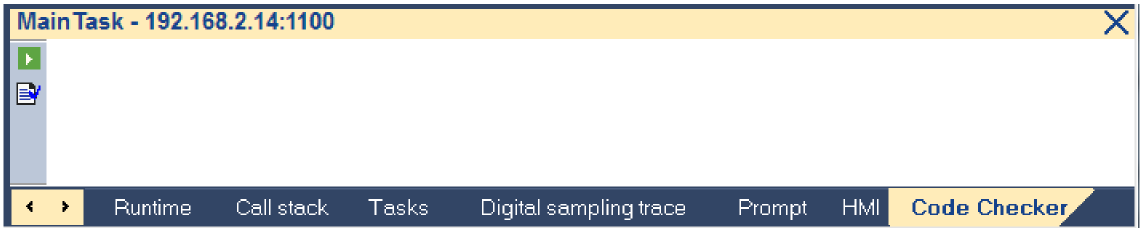
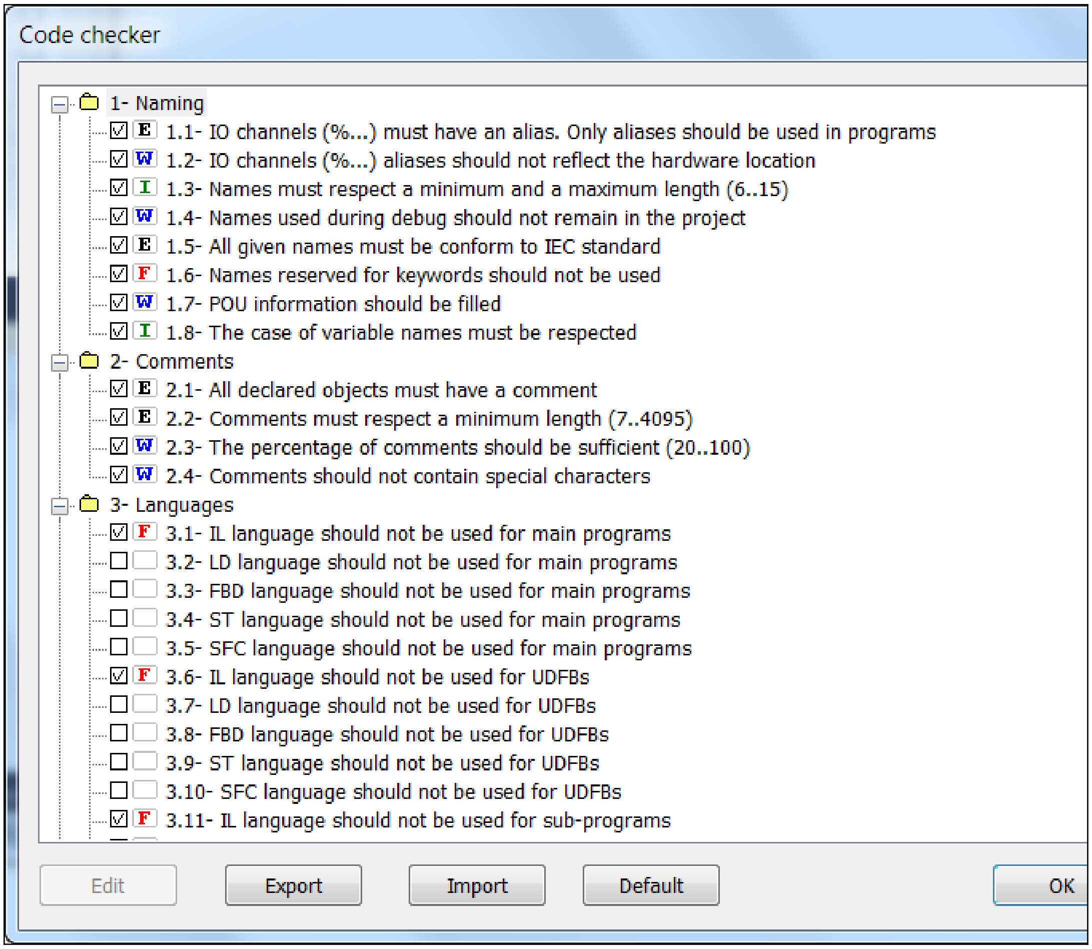


Figure 38: Code Checker output window

Use the *'Settings'* button 📝 to open the configuration window (Figure 39) and select/configure the rules to be checked for violations.



Code checker

- 1- Naming
  - ☑ E 1.1- IO channels (%...) must have an alias. Only aliases should be used in programs
  - ☑ W 1.2- IO channels (%...) aliases should not reflect the hardware location
  - ☑ I 1.3- Names must respect a minimum and a maximum length (6..15)
  - ☑ W 1.4- Names used during debug should not remain in the project
  - ☑ E 1.5- All given names must be conform to IEC standard
  - ☑ F 1.6- Names reserved for keywords should not be used
  - ☑ W 1.7- POU information should be filled
  - ☑ I 1.8- The case of variable names must be respected
- 2- Comments
  - ☑ E 2.1- All declared objects must have a comment
  - ☑ E 2.2- Comments must respect a minimum length (7..4095)
  - ☑ W 2.3- The percentage of comments should be sufficient (20..100)
  - ☑ W 2.4- Comments should not contain special characters
- 3- Languages
  - ☑ F 3.1- IL language should not be used for main programs
  - ☐ 3.2- LD language should not be used for main programs
  - ☐ 3.3- FBD language should not be used for main programs
  - ☐ 3.4- ST language should not be used for main programs
  - ☐ 3.5- SFC language should not be used for main programs
  - ☑ F 3.6- IL language should not be used for UDFBs
  - ☐ 3.7- LD language should not be used for UDFBs
  - ☐ 3.8- FBD language should not be used for UDFBs
  - ☐ 3.9- ST language should not be used for UDFBs
  - ☐ 3.10- SFC language should not be used for UDFBs
  - ☑ F 3.11- IL language should not be used for sub-programs

Figure 39: Checker rule configuration window

The configuration rules have to be set for each task separately. First select the task from the drop box in the toolbar before setting the checker rules (Figure 40).

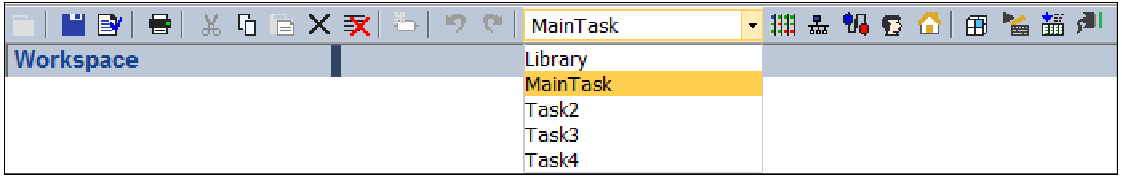
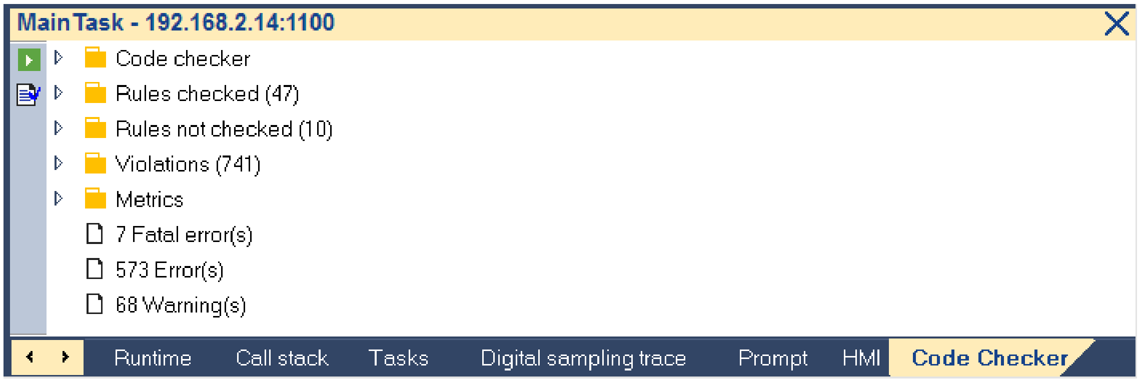


Figure 40: Task selection

Use the *'Scan'* button ▶ to start analysis. The project must be compiled without errors before being checked.

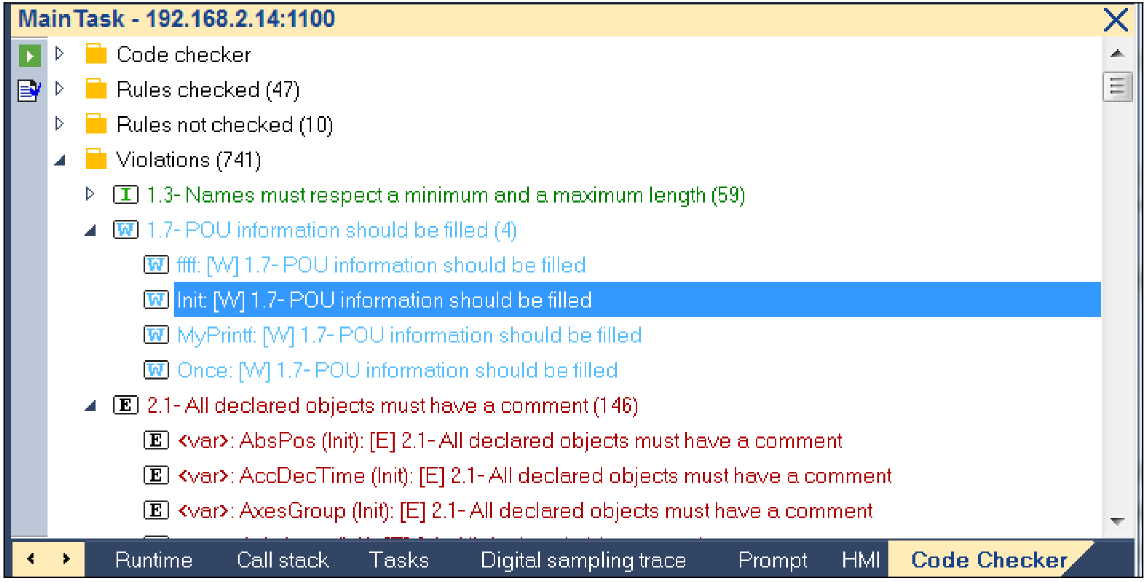


Main Task - 192.168.2.14:1100

- Code checker
- Rules checked (47)
- Rules not checked (10)
- Violations (741)
- Metrics
- 7 Fatal error(s)
- 573 Error(s)
- 68 Warning(s)

Figure 41: Checker scan result

Double-click a violation report line (Figure 42) in the output to navigate to the corresponding location of the source code.



Main Task - 192.168.2.14:1100

- Code checker
- Rules checked (47)
- Rules not checked (10)
- Violations (741)
  - I 1.3- Names must respect a minimum and a maximum length (59)
  - W 1.7- POU information should be filled (4)
    - W fff: [W] 1.7- POU information should be filled
    - W Init: [W] 1.7- POU information should be filled
    - W MyPrintf: [W] 1.7- POU information should be filled
    - W Once: [W] 1.7- POU information should be filled
  - E 2.1- All declared objects must have a comment (146)
    - E <var>: AbsPos (Init): [E] 2.1- All declared objects must have a comment
    - E <var>: AccDecTime (Init): [E] 2.1- All declared objects must have a comment
    - E <var>: AxesGroup (Init): [E] 2.1- All declared objects must have a comment

Figure 42: Violation report

Use commands of the contextual popup menu to copy or export the report.

Note: an option is available in the *'Compiler'* section of the *'Project Settings'* box to systematically run the Code Checker after any successful build.

Checker rules configuration

- Use the *'Configure'* button to select and configure rules to be checked. This is also available from the main tab of the *'Project Settings'* box. Rules are shown in a tree including check boxes. Unchecked rules will be skipped during scan. Double-click on a rule to configure it.
- You can select for each rule a severity level: *'Not checked'* or *'Info'* or *'Warning'* or *'Error'* or *'Fatal'*. You can also enter for each rule a text reference that will be

displayed in reports.

- For rules referring to metrics (e.g. name length) you can enter a minimum and maximum value.
- For rules concerning variables, you can specify a filter based on the *'User*'s group' column of the variable editor: either check only in some user's groups, or check except in some user's groups. Please enter one user's group name per line
- For rule 1.4, you have to enter in the *'Data'* box the list of forbidden names. Please enter one name per line.
- In addition the rules configuration box enables you to export or import the configuration as XML file.

3.7 Status Bar

The status bar informs about:

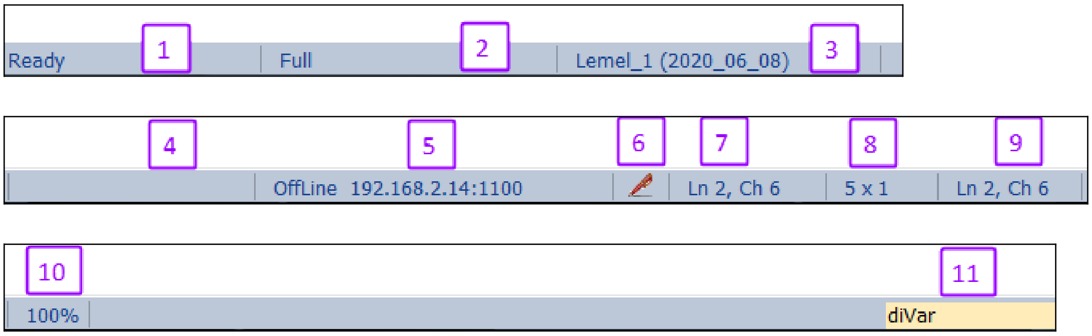


Figure 43: Workbench status bar

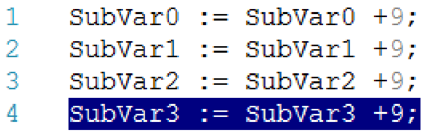
1. Workbench startup status: Indicates whether the workbench has finished starting, creating or loading a new or existing project
2. Toolbar and menu configuration: Double click this section of the status bar to configure the commands display for the toolbar and menu.
3. Target system configuration: Shows the runtime configuration being used for the current program. Double click this section to select a different configuration setting. The runtime configuration of the target system can be uploaded by right clicking the task name in the workspace window and selecting *'Target System Configuration...'* from the popup menu.
4. Empty
5. Communication settings of the target runtime (TCP/IP and port number). In offline mode, the communication parameters are displayed in the status bar. To change them, use the Tools / Communication Parameters menu command or double-click on the parameters in the status bar.
6. Workbench editing mode: program editing mode (✎) or online mode ( ). In online mode the source code of the program can not be changed, it is necessary to

first switch in editing mode before any source code modification is allowed

7. Position of the text cursor in the program editor
8. Selection size in document: The number of lines and text characters marked in the program editor.

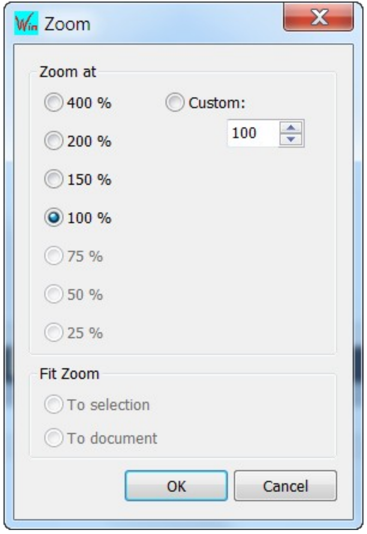
   Example:

   22 x 1 : 22 x 1 indicates that 22 character in the program editor has been marked:



```
SubVar0 := SubVar0 +9;
SubVar1 := SubVar1 +9;
SubVar2 := SubVar2 +9;
SubVar3 := SubVar3 +9;
```

9. Mouse coordinates in document: Position of the mouse cursor in the program editor.
10. Zoom: The size of the text and blocks of the program editor can be zoomed by doubled clicking the zoom section in the status bar.



Zoom

Zoom at

○ 400 %
○ Custom: 100
○ 200 %
○ 150 %
● 100 %
○ 75 %
○ 50 %
○ 25 %

Fit Zoom

○ To selection
○ To document

11. Quick Search: Simply click on the edit box, enter the text you want to search and press ENTER key.

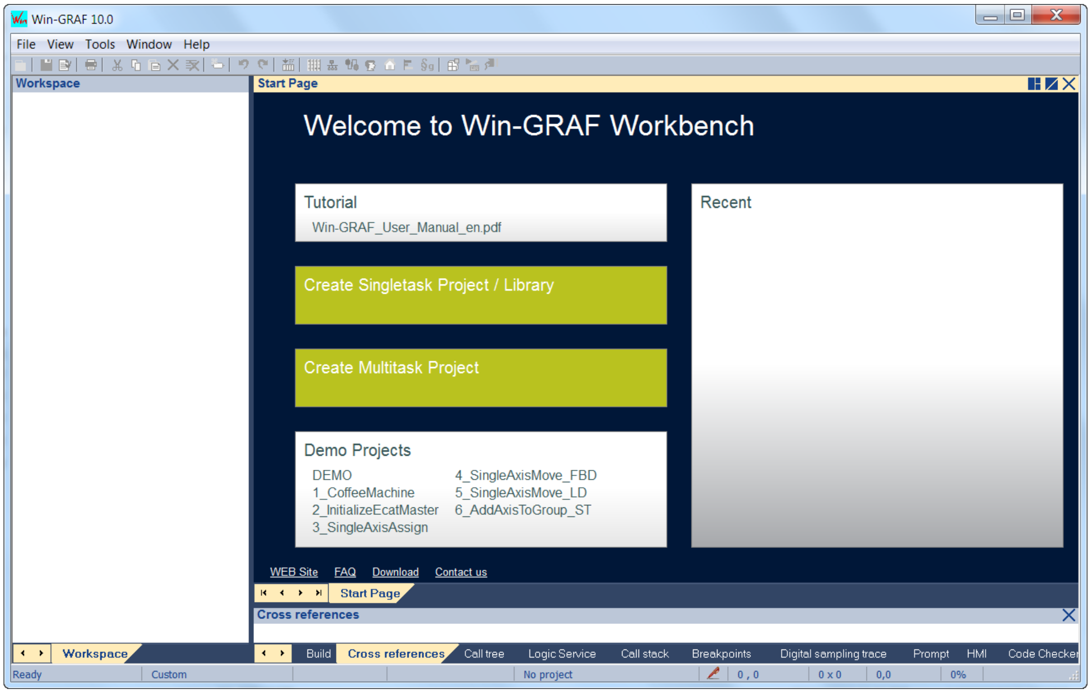
4 Single-Tasking

Depending on the target hardware platform the Win-GRAF runtime supports either single-tasking or multi-tasking application. A single-task project is allowed to run on a multi-tasking runtime but not vice versa. The workbench environment and procedure for creating a single- or multi-tasking application differs. This chapter will focus on describing the single tasking procedure.

4.1 Create a Project

The basic steps for creating a single-tasking PLC project using Win-GRAF workbench:

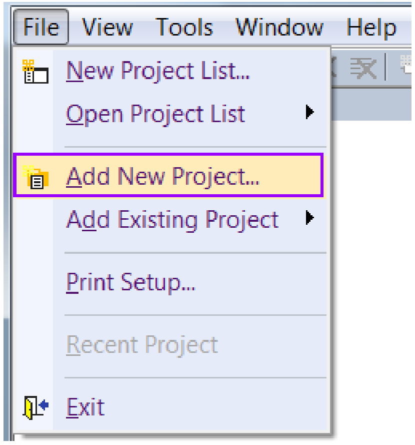
- Start the Win-GRAF workbench has been started. The *'Workspace'* is empty and the *'Start Page'* list the available manuals, demo project and the recent opened project (Figure 44). In addition two green command boxes are listed for directly creating a single- or multi-task project



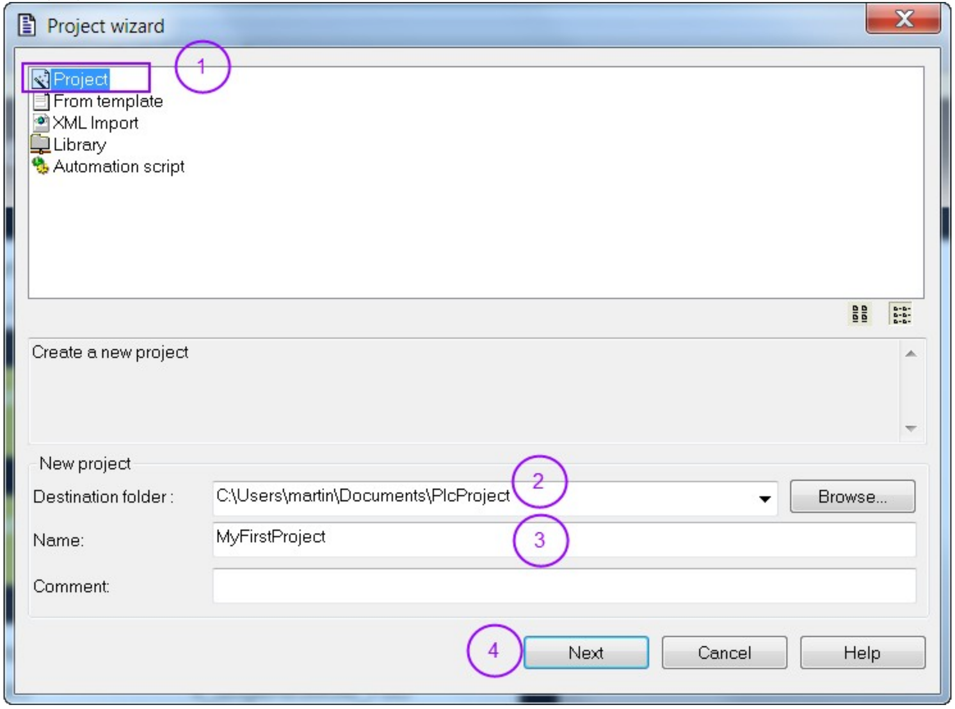
Win-GRAF 10.0
File View Tools Window Help
Workspace
Start Page
Welcome to Win-GRAF Workbench
Tutorial
Win-GRAF_User_Manual_en.pdf
Create Singletask Project / Library
Create Multitask Project
Demo Projects
DEMO
1_CoffeeMachine
2_InitializeEcatMaster
3_SingleAxisAssign
4_SingleAxisMove_FBD
5_SingleAxisMove_LD
6_AddAxisToGroup_ST
Recent
WEB Site FAQ Download Contact us
Start Page
Cross references
Workspace
Build Cross references Call tree Logic Service Call stack Breakpoints Digital sampling trace Prompt HMI Code Checker
Ready Custom No project 0 , 0 0 x 0 0,0 0%

Figure 44: Workbench with empty workspace

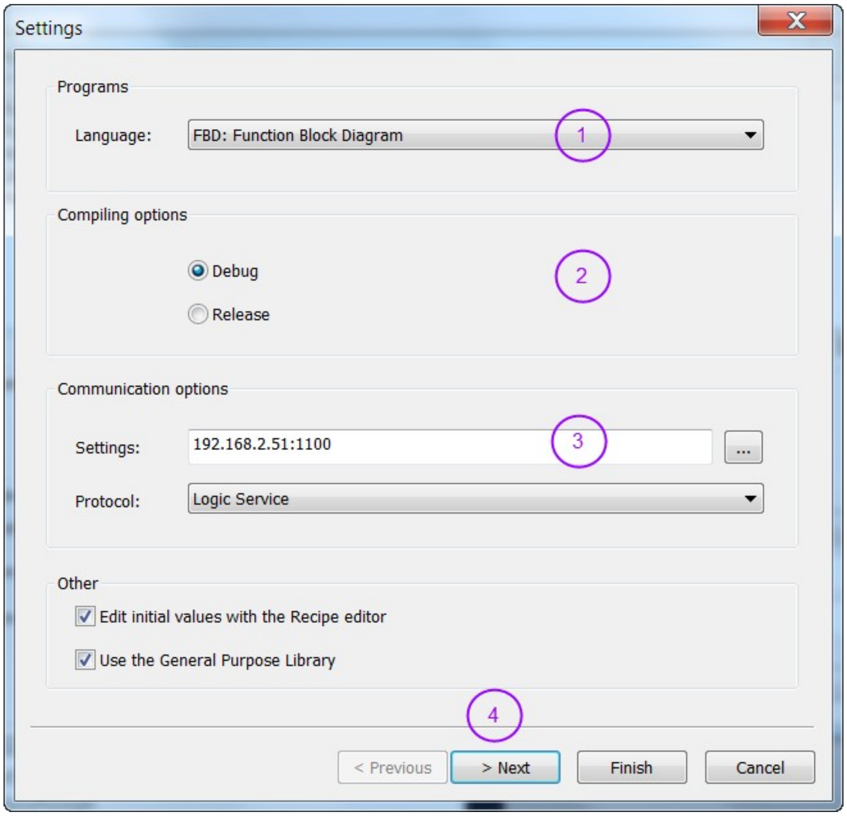
- To open the project wizard for single-task project click either the green command button in the '*Start Page*' or go to '*File\Add New Project...*'



- Click the *'Project'*, select the destination folder and enter the project name. Click *'Next'*.
  Note: For the project several folders and files are being generated. For easier maintains it is therefore suggested to create a new folder for the project or use an empty destination folder.



- Set the programming language of the first program (POU), compiling option, enter the IP address of the remote device and leave the protocol to *'Logic Service'*. All settings still can be modified after the project has been created. Although the wizard allows you to select only one programming language for the first program, additional programs for different programming language can be added later on. Click *'Next'* to continue the configuration.



Settings

Programs

Language: FBD: Function Block Diagram

Compiling options

● Debug

○ Release

Communication options

Settings: 192.168.2.51:1100

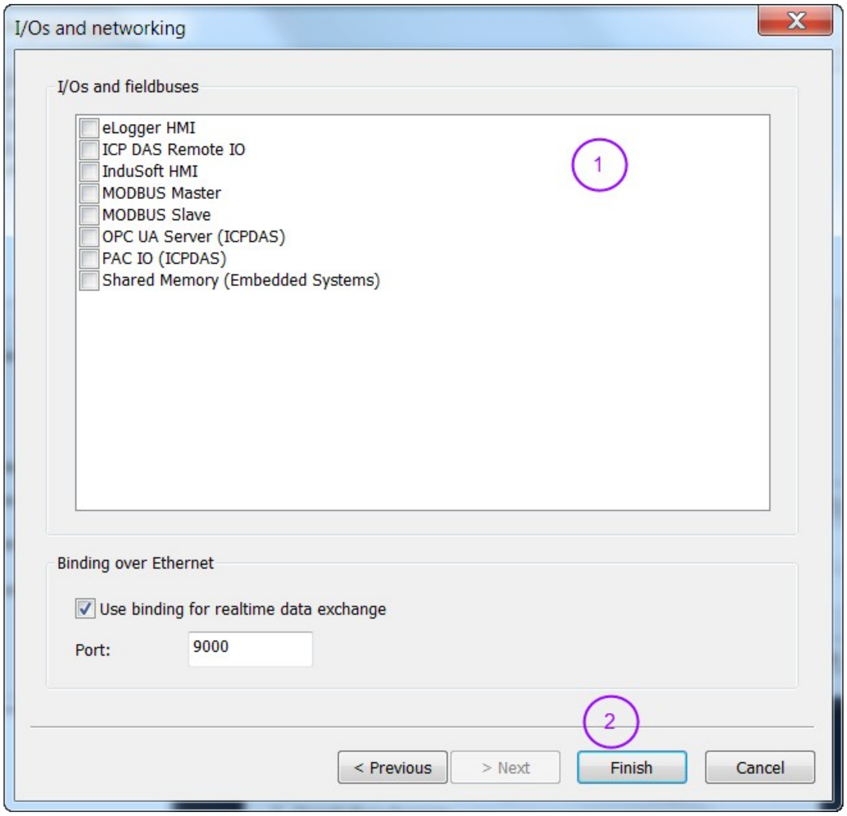
Protocol: Logic Service

Other

☑ Edit initial values with the Recipe editor

☑ Use the General Purpose Library

- Select the additional components to use. If you are unsure which components to add to the project at the current stage then leave the field unchecked. They can be manually added at any time during the project development. Do not change the '*Binding*' setting. Confirm the setting by clicking '*Finish*'. A new project with the current setting will be generated


I/Os and networking

I/Os and fieldbuses

- □ eLogger HMI
- □ ICP DAS Remote IO
- □ InduSoft HMI
- □ MODBUS Master
- □ MODBUS Slave
- □ OPC UA Server (ICPDAS)
- □ PAC IO (ICPDAS)
- □ Shared Memory (Embedded Systems)

Binding over Ethernet

☑ Use binding for realtime data exchange

Port: 9000

- Figure 45 shows the '*Workspace*' setting of a new single-tasking project. Double click the '*Main*' item to open the main programming editor.

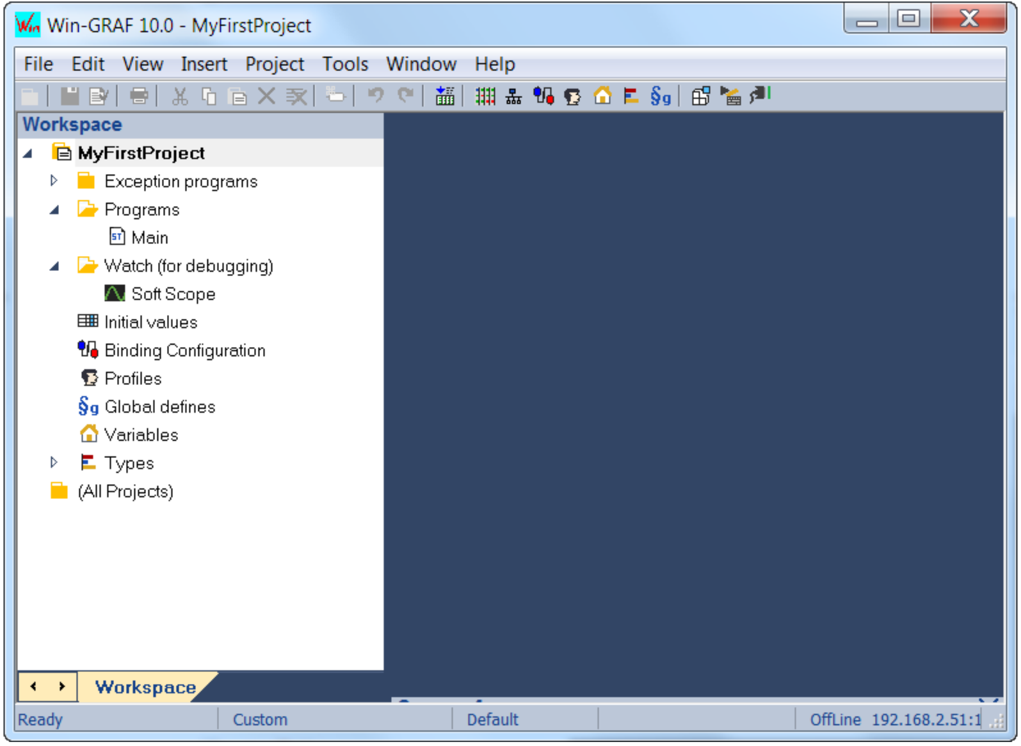


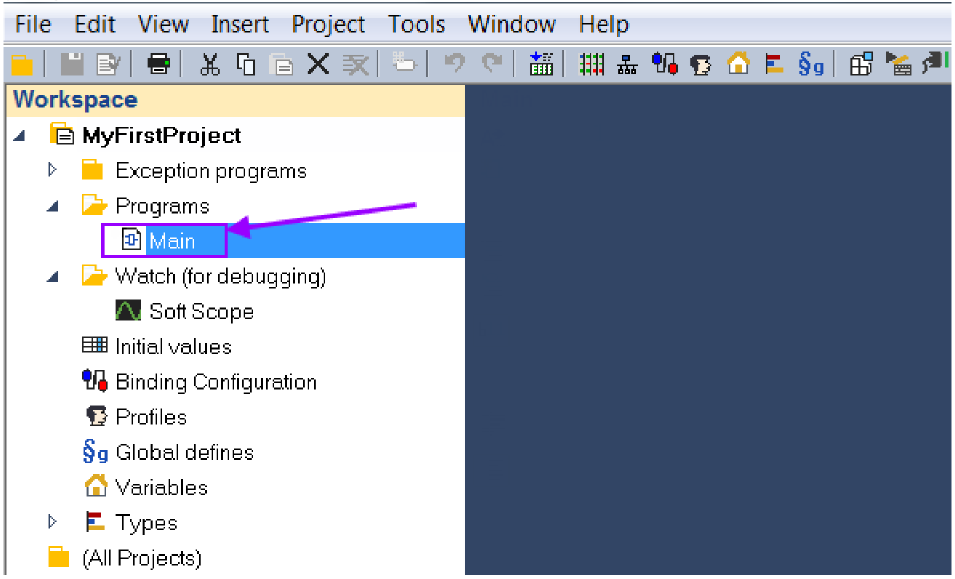
Figure 45: Single-tasking project

4.2 Edit a Program

The main focus of this section is to give a brief introduction to the user interface of the workbench and show how to use the tools provided by the workbench to edit the logic for a PLC program. The Win-GRAF workbench supports all the five PLC programming languages defined by IEC61131. For each programming language a separated editor is provided. More information about each editors toolkit is given in chapter 6.

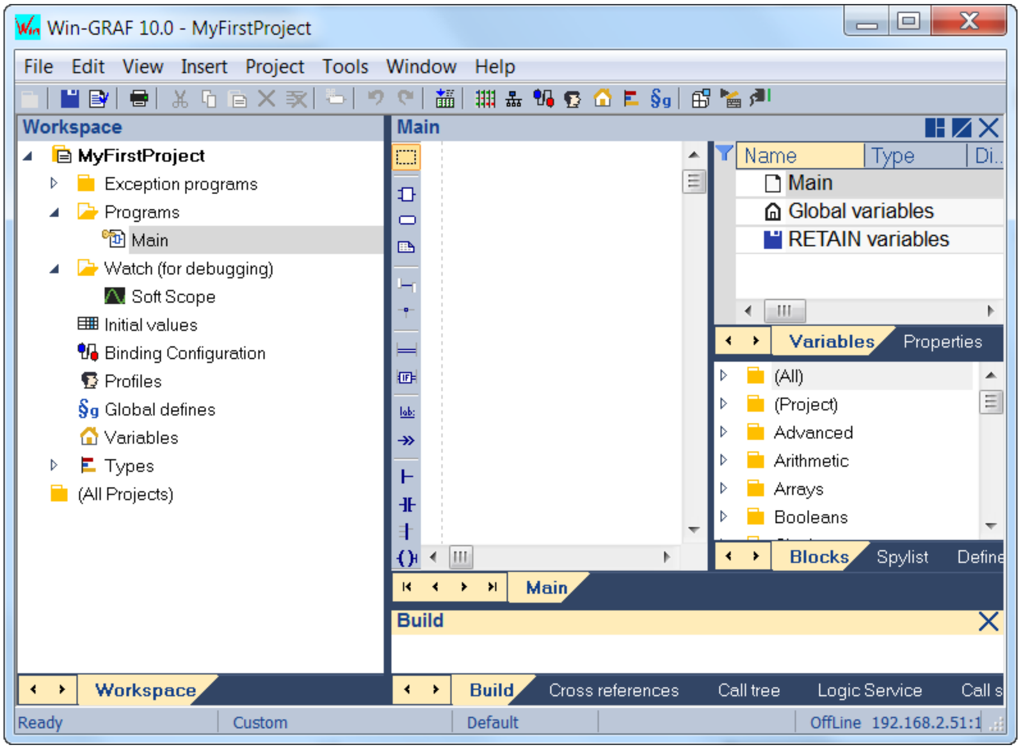
Basic procedure to declare variables and edit a function using the FBD programming editor:

Step 1: Double click the name of the '*Main*' program in the workspace to open the program editor:



Note: Double click the program name and not the icon.

The PLC logic can now be edited using the FBD language. In the following steps demonstrats how to add a function block to the editor and declare its in- and output variables.

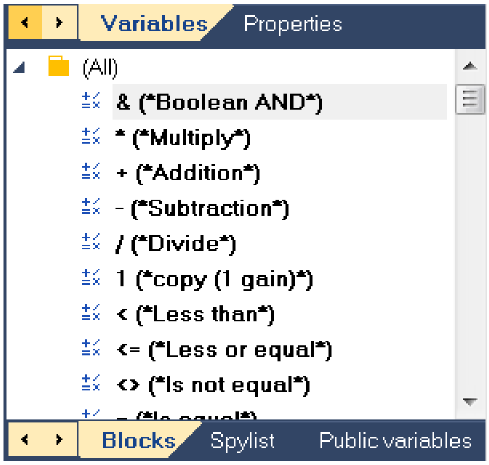


Step 2: Add the '*AND*' function block:

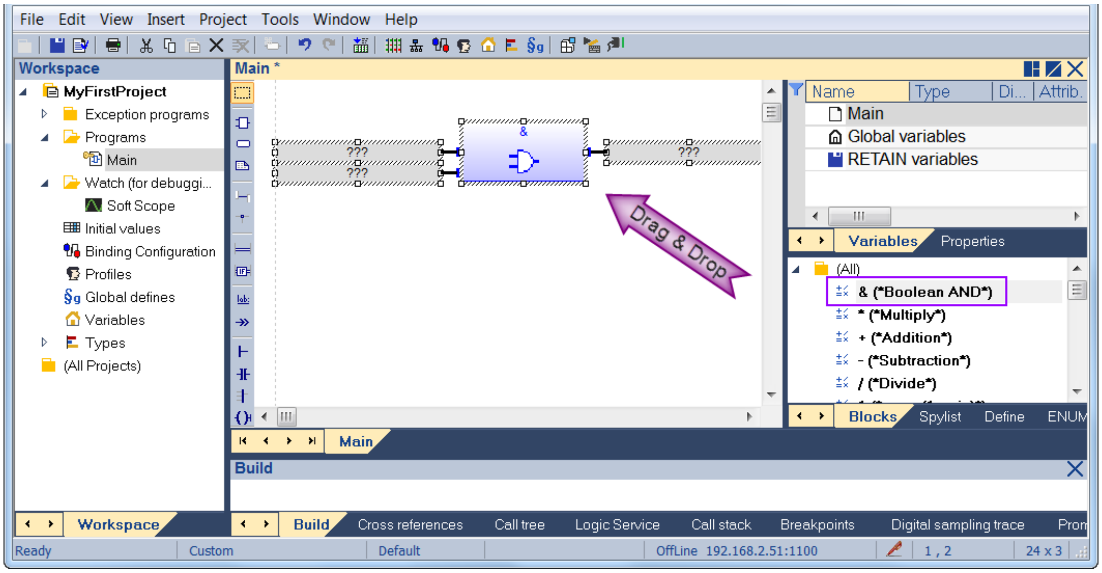
All the supported function block are listed in the '*Blocks*' tab of the Info window on the right. If the Info window is not visible then go to '*View/Infos Tab2*' in the menu bar to display the window.

The function blocks are listed according to different categories. The '*(All)*' category list all the supported function blocks

Click the '*(All)*' tree node to display all function blocks.



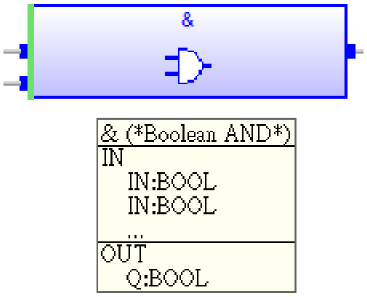
Click the *'&(*Boolean AND*)'* function block and drag it onto the editor area.



The *'AND'* function block has got two input and one output variable. The *'???'* at the inputs and output indicate that no variable has been assigned yet.

The size of the function block can be changed by clicking once on the block and pressing the *'+'* or *'-'* key on the keyboard.

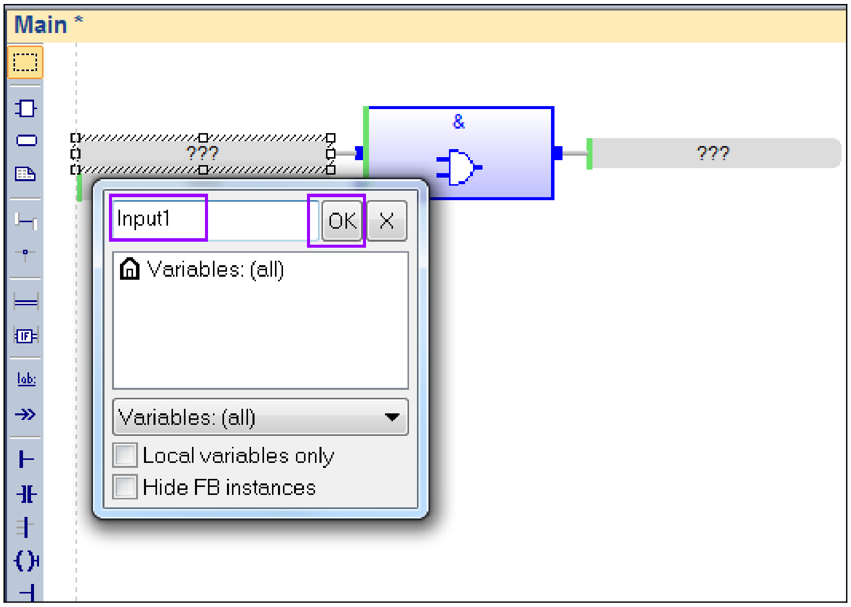
Moving the mouse pointer over the function displays the in- and output data type required:



```
& (*Boolean AND*)
IN
   IN:BOOL
   IN:BOOL
   ...
OUT
   Q:BOOL
```

Step 3: Assigning variables to the function block:

- Double click on the grey input field with the question marks.
- Enter the name of the variable
- Click *'OK'*



Input1

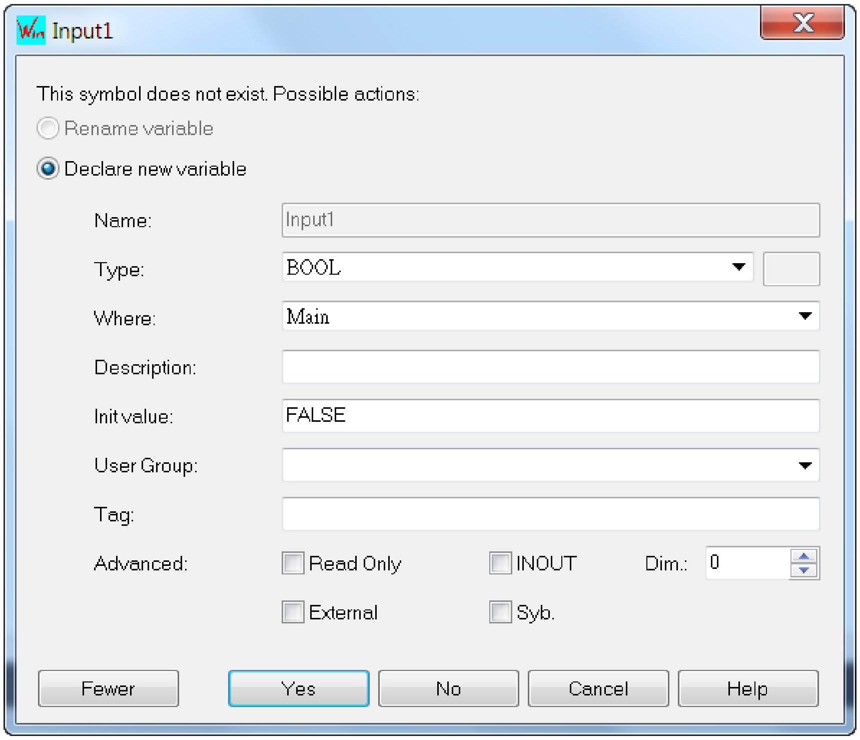
Variables: (all)

□ Local variables only

□ Hide FB instances

As the variable has not been declared in the project before a windows pops up which allows you to select the data type, initial value, etc..

- The function block input data type is BOOL, therefore select BOOL
- The initial value is set to FALSE
- Click *'Yes'* to add the variable to the project



Input1

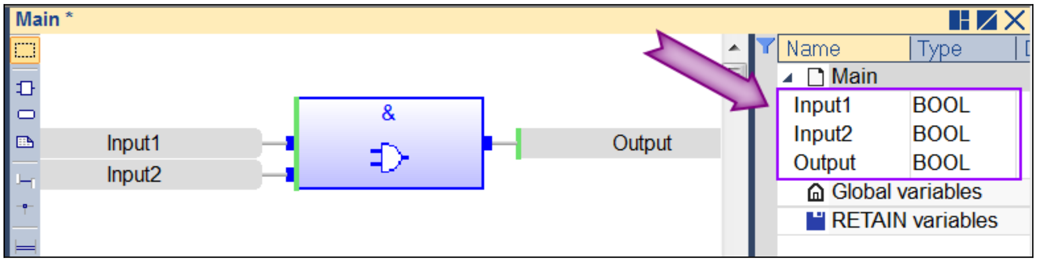
This symbol does not exist. Possible actions:

○ Rename variable

● Declare new variable

Name:	Input1
Type:	BOOL
Where:	Main
Description:	
Init value:	FALSE
User Group:	
Tag:	
Advanced:	□ Read Only □ INOUT Dim.: 0
	□ External □ Syb.

Repeat the above procedure to add the variable '*Input2*' to the second function block input and a '*Output*' variable to the output. All the newly declared variables are listed in the variables view, on the right of the screen.

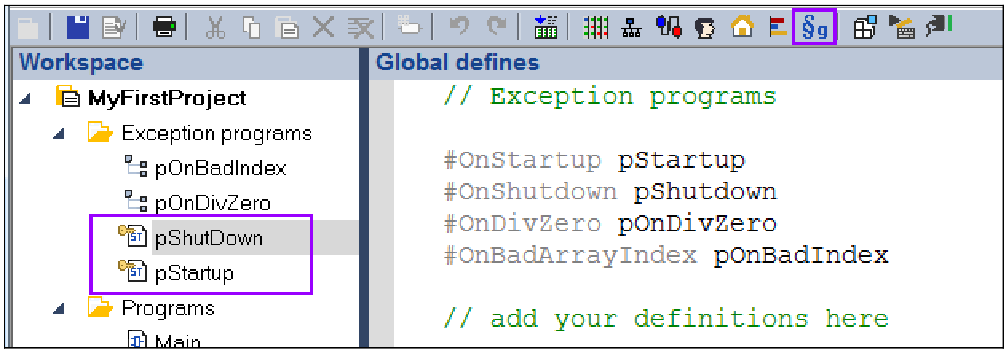


Name	Type
Input1	BOOL
Input2	BOOL
Output	BOOL

4.3 Create a Program

A task can handle several programs (POU) written in different languages. The number of programs in an application is limited to 32767.

By default the single-tasking contain two exception programs which will be called once during the PLC startup ('*pStartup*') and shutdown ('*pShutDown*') . Their purpose is to do some system initialization and cleanup. The user can edit the code inside these exception programs. If not required they can be disabled deleting their global definition. The '*Global defines*' editor has to be opened via the '*§*' command in the toolbar.



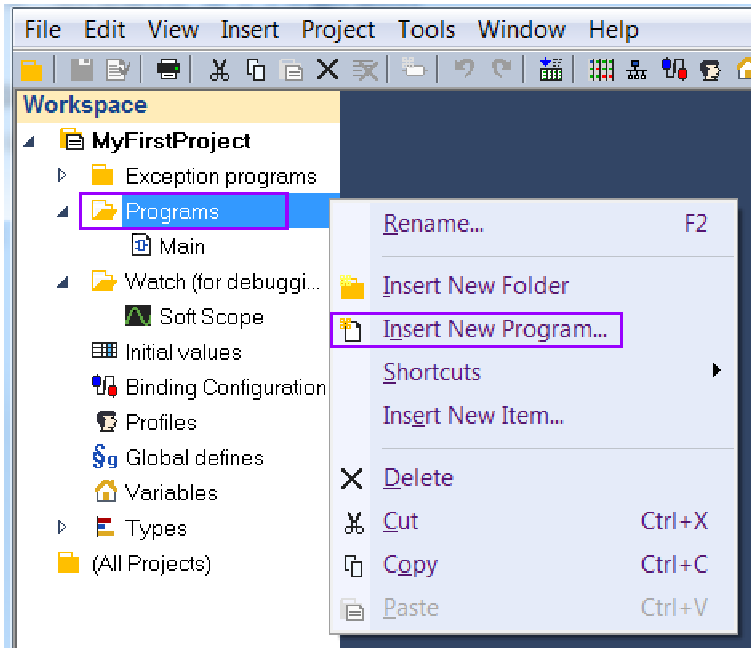
```
// Exception programs

#OnStartup pStartup
#OnShutdown pShutdown
#OnDivZero pOnDivZero
#OnBadArrayIndex pOnBadIndex

// add your definitions here
```

Figure 46: Exception programs

We will show how to add a new program to a task:

Step 1: Insert a new program. Right the program folder and select *'Insert New*

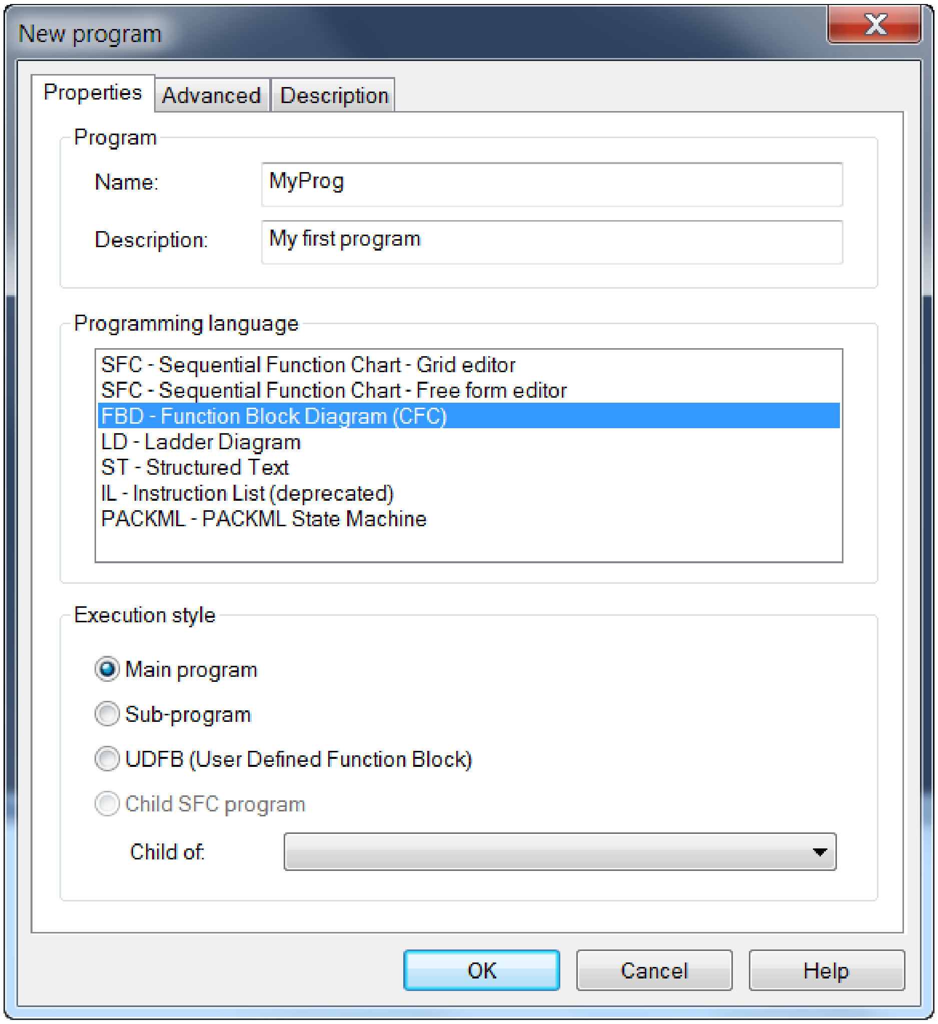
Program...' from the popup menu.



Step 2: Make the following entries:

- Enter a name for the program
- Give a short program description (optional)
- Select one of the five IEC61131 programming languages (SFC, FBD, LD, ST, IL) for the program. Remember that more than one program can be added to the task. Each program can be programmed in a different language.

For this tutorial the '*FBD-Function Block Diagram (CFC)*' is selected. Add the program to the workspace by clicking '*OK*'.



Program

Name: MyProg

Description: My first program

Programming language

- SFC - Sequential Function Chart - Grid editor
- SFC - Sequential Function Chart - Free form editor
- FBD - Function Block Diagram (CFC)
- LD - Ladder Diagram
- ST - Structured Text
- IL - Instruction List (deprecated)
- PACKML - PACKML State Machine

Execution style

- ● Main program
- ○ Sub-program
- ○ UDFB (User Defined Function Block)
- ○ Child SFC program

Child of:

Programs must have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a standard or '*C*' function or function block. A variable should not have the same name as a declared data type. The name of a program should begin by a letter or an underscore ('_') mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Two names with different cases are considered as the same.

4.4 Task Configuration

A task of a PLC application can control several IEC 61131 programs (Figure 47). The user is allowed to add several IEC programs to a task whereby each program has to be written in one of the five programming language defined by IEC61131 (ST, LD, FB, IL, SFC). Each programs of a task can be written in a different language. Programs are executed according to the order defined by the user. The number of programs in an application is limited to 32767.

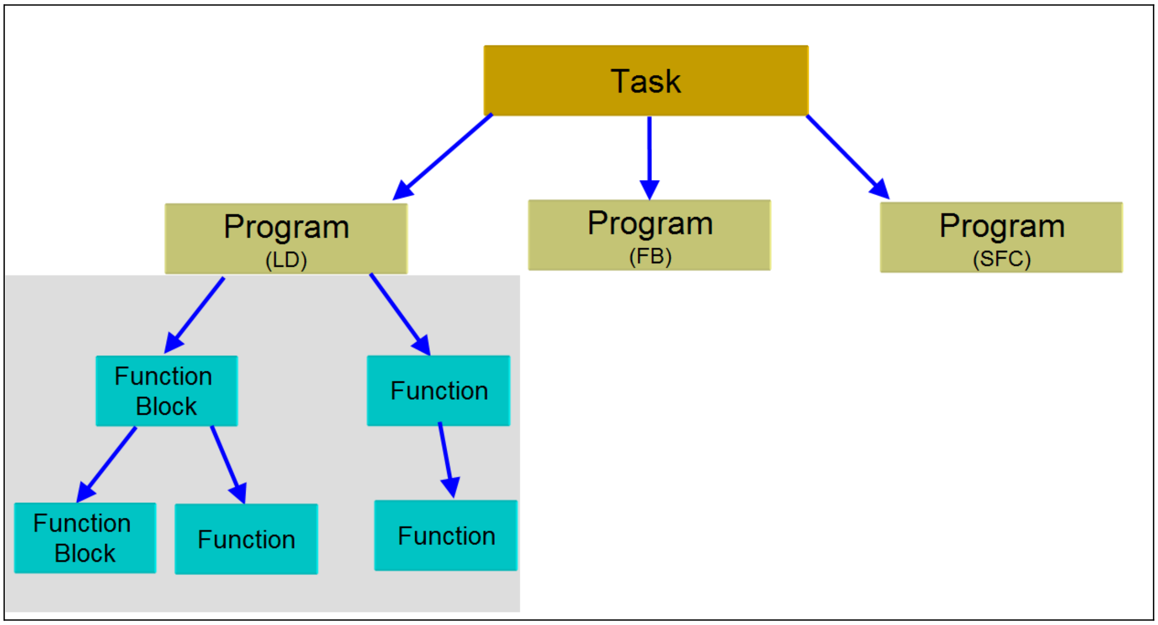


Figure 47: PLC task-program architecture

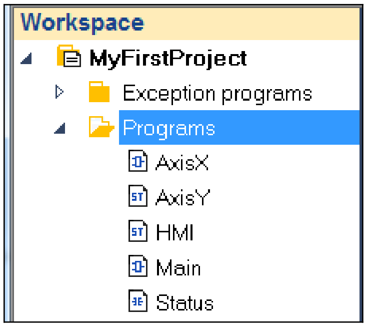


Figure 48: Task with several programs

The function of a PLC task is to control the processing of each of its IEC programs (Figure 48). In the workbench the IEC programs are listed in the workspace tree below the task. A task is defined by a name, a priority and by a type determining which condition will trigger the start of the task. You can define this condition to be either cyclic or freewheeling.

For each task, you can specify a series of program POUs that will be started by the task. The execution order and period of each program can be set.
The combination of priority and condition will determine in which chronological order the tasks will be executed.
For each task, you can configure a time control (watchdog). The possible settings depend on the specific controller platform.

4.4.1 Task Cycle Time

The Win-GRAF runtime supports two type of tasks: cyclic and freewheeling. PLC cycle time is defined as the time it takes to run the code logic from start to finish.

Programs are executed sequentially within the target cycle, according to the following model (Figure 49):

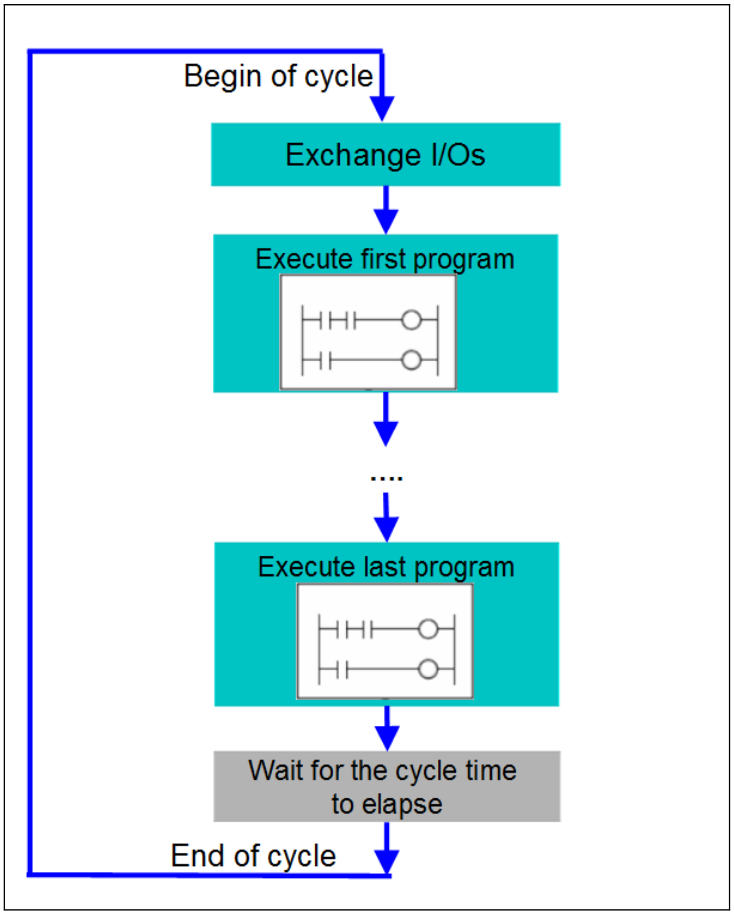
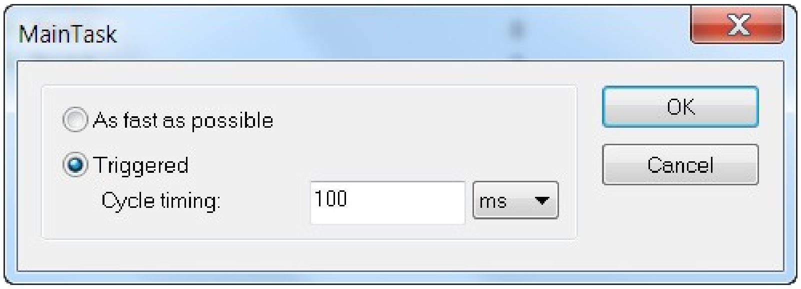


Figure 49: Task cycle execution

4.4.1.1 Cyclic Task

A Cyclic task is assigned a fixed cycle time which causes the runtime to trigger the task execution at a set fixed time interval. The next cycle is triggered, once the cycle time has elapsed.

Figure 50 shows the interface for setting the cycle time interval. The '*Cycle timing*' is the period of time, after which the task should be restarted. You can choose the desired time unit in the selection box behind the edit field: milliseconds [ms] or microseconds [µs].



MainTask

○ As fast as possible

● Triggered

Cycle timing: 100 ms

Figure 50: Cyclic task time setting

Note:

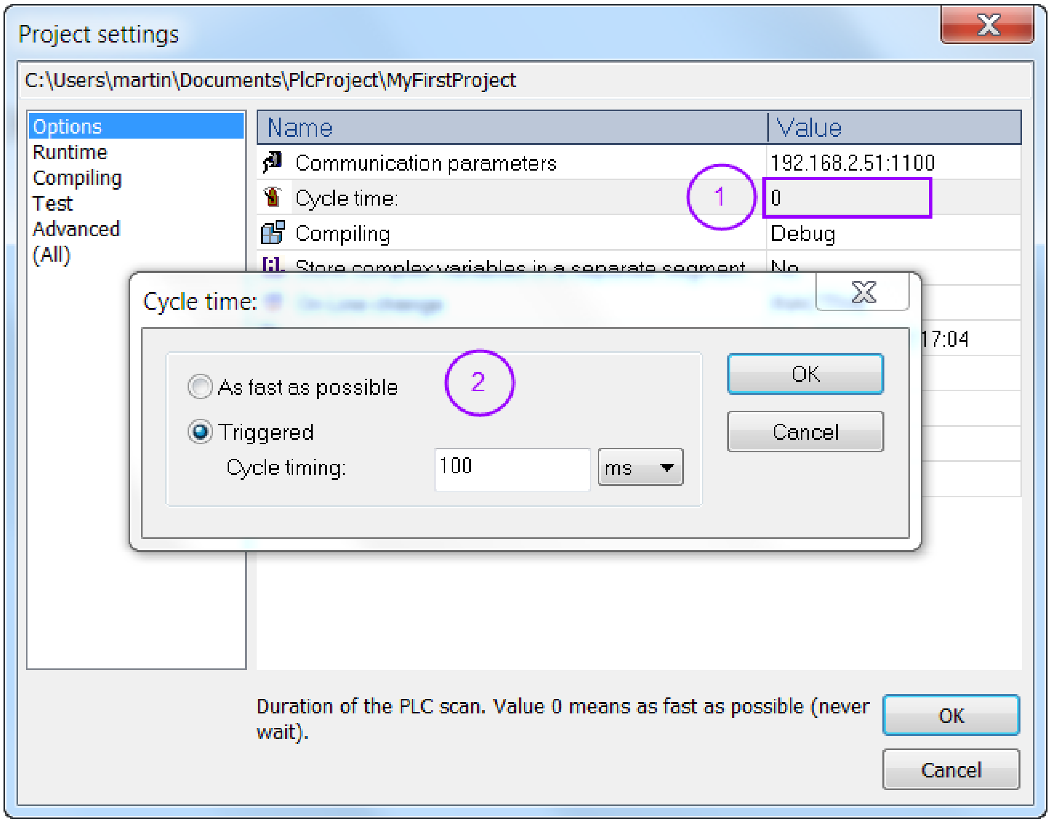
- If the execution of one cycle takes longer than the defined cycle time, the next cycle starts as soon as the previous cycle is finished without executing other lower priority tasks. This will affect the execution of all tasks and cause the runtime to generate a system watchdog exception and write a warning message to the output window.
- The runtime for Windows is not real time, therefore it is suggested to not set the time interval below 100 milliseconds. The Windows OS timer accuracy is in the range of about 100 milliseconds.

There are a few methods to decrease the PLC's response time if cycle time of a task has to be reduced. Reducing cycle time of a task

- A faster CPU will execute code faster and reduce the overall cycle time.
- Another method for reducing cycle time is optimizing the code itself. Moving pieces of code that do not have to run every cycle into a program with a higher cycle time, but may not have much effect on the maximum cycle time.
- Prevent execution peaks loading. Execution peaks occur when all programs of a task run in the same cycle. By distributing the program execution over several cycles these peaks can be prevented (see Figure 53)

Set cyclic task time:

1. Open the '*Project setting*' dialog by selecting '*Project\Settings...*' in the menu bar. Select the '*Option*' item and double click the '*Cycling time*' value to open the '*Cycle time*' configuration window.
2. Enable the '*Triggered*' option and set the '*Cyclic timing*' value. The '*Cyclic timing*' is the fixed time interval at which the task execution will be triggered.



Project settings

C:\Users\martin\Documents\PlcProject\MyFirstProject

Name	Value
Communication parameters	192.168.2.51:1100
Cycle time:	0
Compiling	Debug
Store complex variables in a separate segment	No

Cycle time:

○ As fast as possible

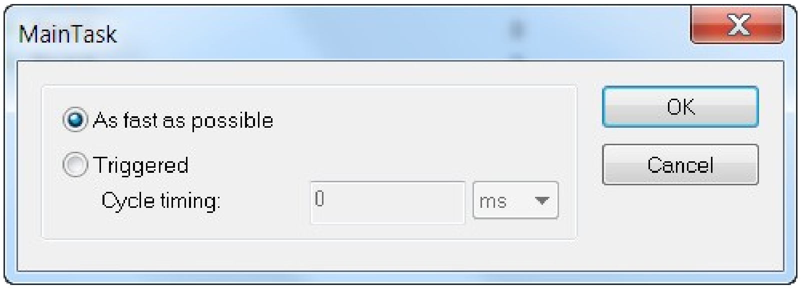
● Triggered

Cycle timing: 100 ms

Duration of the PLC scan. Value 0 means as fast as possible (never wait).

4.4.1.2 Freewheeling Task

A Freewheeling task does not have a fixed duration. In Freewheeling mode, each task begins when the previous cycle has been completed. The cycle time is not triggered at a fixed time interval.



MainTask

● As fast as possible

○ Triggered

Cycle timing: 0 ms

Figure 51: Freewheeling Task setting

Note:

- Make sure the freewheeling task has the lowest priority setting otherwise other tasks will be prevented from execution.

- Set all the freewheeling tasks to the same priority level otherwise the freewheeling task with the lower priority will have not time slot to execute.

4.4.2 Program Execution Sequence

A task may consists of several programs. The workbench allows you to set the execution order, the period and phase of each program.

For example Figure 52 shows an application where the main task consists of five programs.

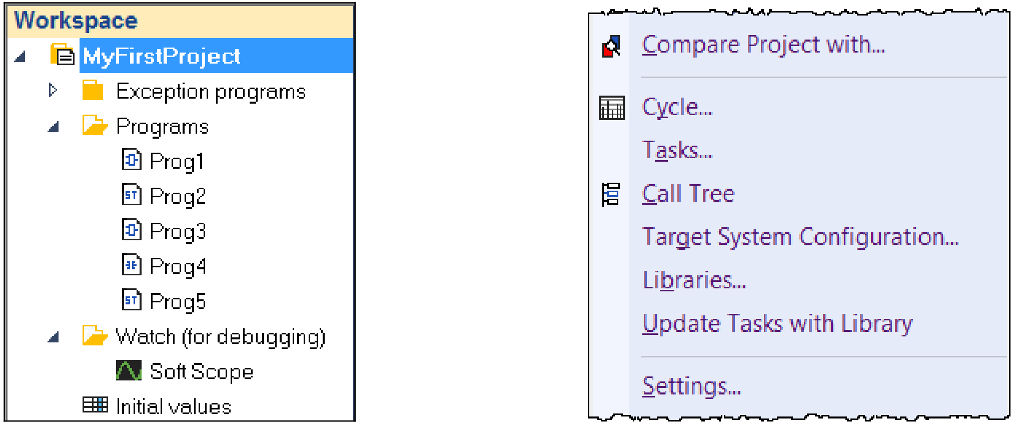
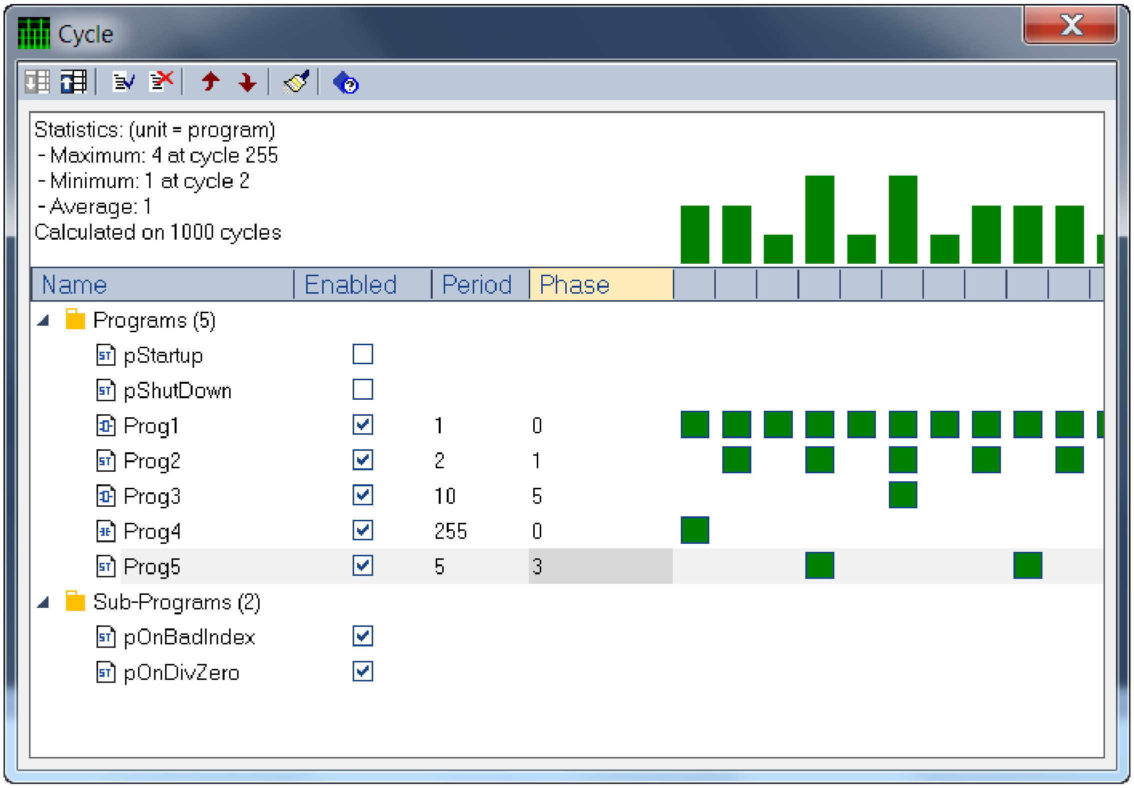


Figure 52: Main task with several programs (left); context menu (right)

Use the Cycle dialog (Figure 53) to define the execution properties of the various programs. Open the Cycle dialog by right clicking the task name and selecting *'Cycle...'* from the popup menu (Figure 52).



Statistics: (unit = program)
- Maximum: 4 at cycle 255
- Minimum: 1 at cycle 2
- Average: 1

Calculated on 1000 cycles

Name	Enabled	Period	Phase
Programs (5)			
pStartup	□		
pShutDown	□		
Prog1	☑	1	0
Prog2	☑	2	1
Prog3	☑	10	5
Prog4	☑	255	0
Prog5	☑	5	3
Sub-Programs (2)			
pOnBadIndex	☑		
pOnDivZero	☑		

Figure 53: Cycle window for program execution property setting

The Cycle dialog shows the list of the main programs, as they will be executed in runtime cycles.

Program Execution Property	Description
Program execution order	The programs will be executed in the sequence they appear in the table of Cycle dialog from top to bottom. The order of execution is determines by the vertical ordering and therefore the program execution order of Figure 53is: 1. *'Prog1'* 2. *'Prog2'* 3. *'Prog3'* 4. *'Prog4'* 5. *'Prog5'*
Period	The '*Period*' defines after how many cycles the program is executed again. It defines how many cycles are set between two executions of the same program. You can define various sampling periods for the

Program Execution Property	Description
	programs of a task. By default the value is *'1'* which means the program is executed in each task cycle. Valid range is 1 to 255. Giving a slower period (=high '*Period*' value) to some of the programs in a task is an easy way to give a higher priority to some other programs. Example: • Period = 1 --> program is executed every cycle • Period = 2 --> program is executed every second cycle • Period = 10 --> program is executed every tenth task cycle
Phase	The '*Phase*' defines the cycle where the program is executed the first time. It is an offset which enables you to dispatch slows programs among few cycles. The goal is to prevent slow program to be triggered all in the same cycle and thereby causing peak loads. The '*Phase*' setting allows you to reduce execution peak loads by postponing slow program execution. Slow programs are programs with a higher '*Period*' value setting. By default the value is '*0*' which means the program is executed with the first task cycle. Valid range is 1 to 255. Example: • A Program with Period=2 and Phase=1 is executed each **even** cycle • A Program with Period=2 and Phase=0 is executed each **odd** cycle
Program enable/ disable	*'Enabled'* indicates whether the program should be built/compiled for the application
Graph	The Cycle window shows two graphs: 1. Top graph: It shows how many programs are executed in each cycle. It allows you to determine whether execution peaks loadings occur. 2. Bottom graph: The squares indicates when a program execution will be triggered in a task. Hint: More cycles will be shown if the Cycle dialog size is increased by dragging the right edge to the left.

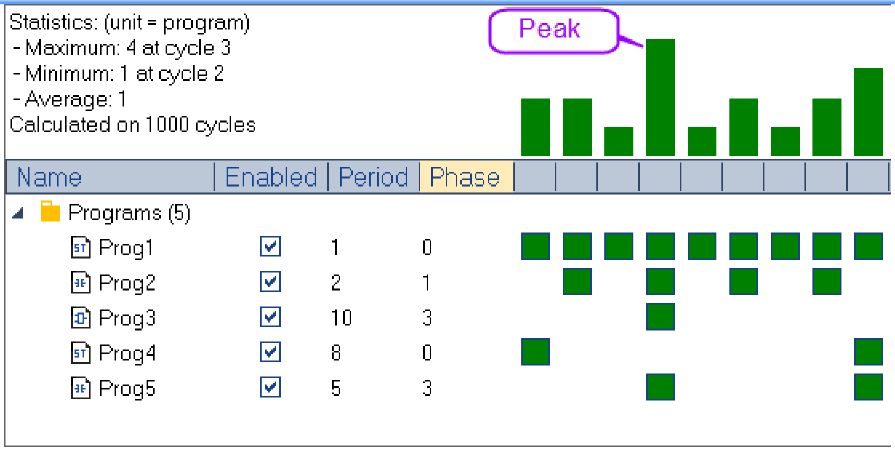
Program Execution Property	Description
	 Statistics: (unit = program) - Maximum: 4 at cycle 3 - Minimum: 1 at cycle 2 - Average: 1 Calculated on 1000 cycles Peak Name | Enabled | Period | Phase Programs (5) Prog1 ☑ 1 0 Prog2 ☑ 2 1 Prog3 ☑ 10 3 Prog4 ☑ 8 0 Prog5 ☑ 5 3

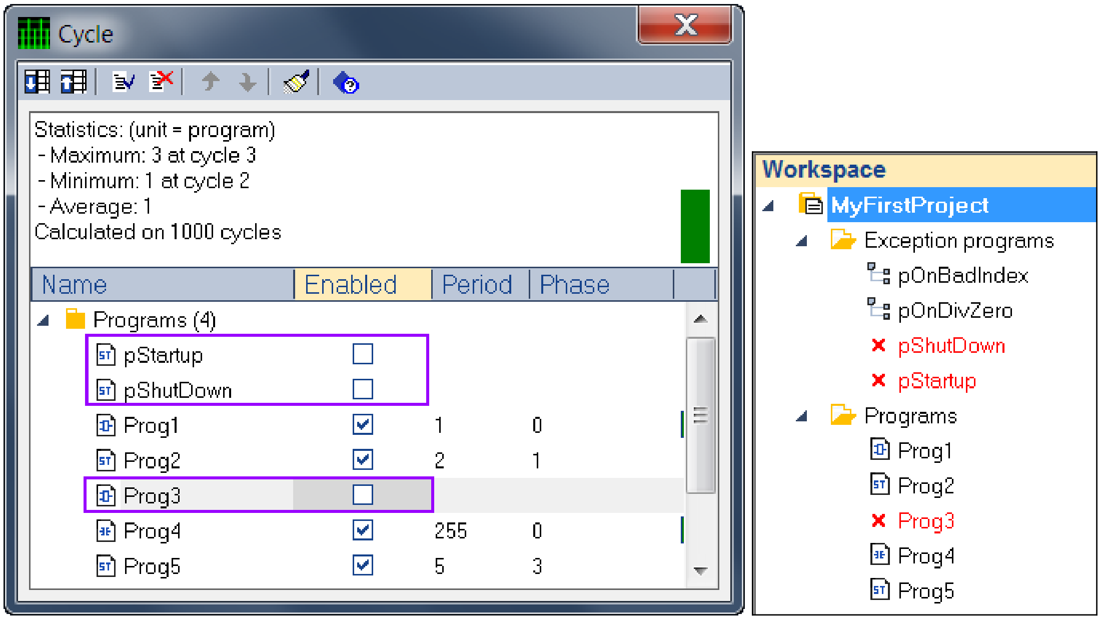
Table 4: Program execution properties

Icon	Command	Description
⇩	Move down	Move the selected program one line down
⇧	Move up	Move the selected program one line up
✓	Program enable	Include the program to the application. Program will be included in the built/compiling process
✗	Program disable	Exclude the program from the application. Program will not be included in the built/compiling process. A red cross will be shown in the workspace window next to the program name
↑	Increment	Increment the period and phase value by one of selected program
↓	Decrement	Decrement the period and phase value by one of selected program
✎	Default setting	Set the default values for selected program
?	Help	Open help documentation

Table 5: Cycle window commands

You can:

- use the Move buttons ⇩ ⇧ to change the program execution order within the task cycle.
- use the Select buttons ✓ ✗ to specify if the program must be called in the cycle. Unselected programs are ignored at compiling time, and are shown with a red cross icon in the workspace (Figure 54).
- use the Increase/Decrease buttons ↑ ↓ when the Period or Phase column is selected to change the scheduling of a program. This enables you to define '*slow*' programs that are not called on every cycle. See the Program advanced properties for further details.



Statistics: (unit = program)
- Maximum: 3 at cycle 3
- Minimum: 1 at cycle 2
- Average: 1
Calculated on 1000 cycles

Name	Enabled	Period	Phase
Programs (4)			
pStartup	□		
pShutDown	□		
Prog1	☑	1	0
Prog2	☑	2	1
Prog3	□		
Prog4	☑	255	0
Prog5	☑	5	3

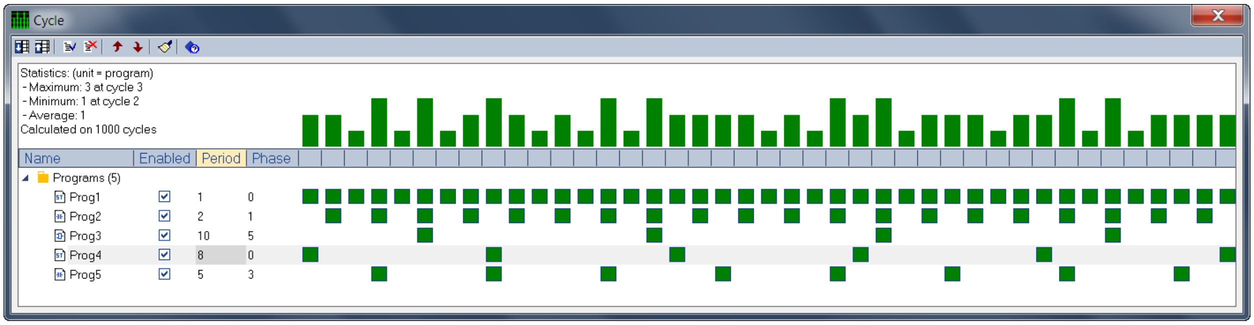
Workspace
- MyFirstProject
  - Exception programs
    - pOnBadIndex
    - pOnDivZero
    - ✗ pShutDown
    - ✗ pStartup
  - Programs
    - Prog1
    - Prog2
    - ✗ Prog3
    - Prog4
    - Prog5

Figure 54: Unselect a program for the compiling process

Example:
The following table (Table 6) shows for a task which executes five program the period and phase settings. After the data has been set in the Cycle dialog (Figure 55) the top bar chart in the Cycle dialog shows the number of programs to be executed in each cycle and the bottom graph shows at which cycle the execution of each individual program will be triggered.

Period	Phase	Description
1	0	- Period = 1: Program *'Prog1'* is executed every cycle. Therefore the phase has to be set to zero. - Phase = 0: *'Prog1'*starts to executed in the first cycle
2	1	- Period = 2: Program *'Prog2'* is executed every second cycle. The phase can be set either to zero or to one. - Phase = 1: *'Prog2'*starts to executed in the second cycle
10	5	- Period = 10: Program *'Prog3'* is executed every tenth task cycle. Valid phase range 0 to 9. - Phase = 5: *'Prog3'* starts to executed in the fifth cycle
8	0	- Period = 8: Program *'Prog4'* is executed every eight task cycle. Valid phase range 0 to 7. - Phase = 0: *'Prog4'* starts to executed in the first cycle
5	3	- Period = 5: Program *'Prog5'* is executed every tenth task cycle. Valid phase range 0 to 4. - Phase = 3: *'Prog5'* starts to executed in the fifth cycle

Table 6: Example - period and phase settings



Statistics: (unit = program)
- Maximum: 3 at cycle 3
- Minimum: 1 at cycle 2
- Average: 1
Calculated on 1000 cycles

Name	Enabled	Period	Phase
Programs (5)			
Prog1	☑	1	0
Prog2	☑	2	1
Prog3	☑	10	5
Prog4	☑	8	0
Prog5	☑	5	3

Figure 55: Example - Cycle dialog

4.5 Build/Compile Application

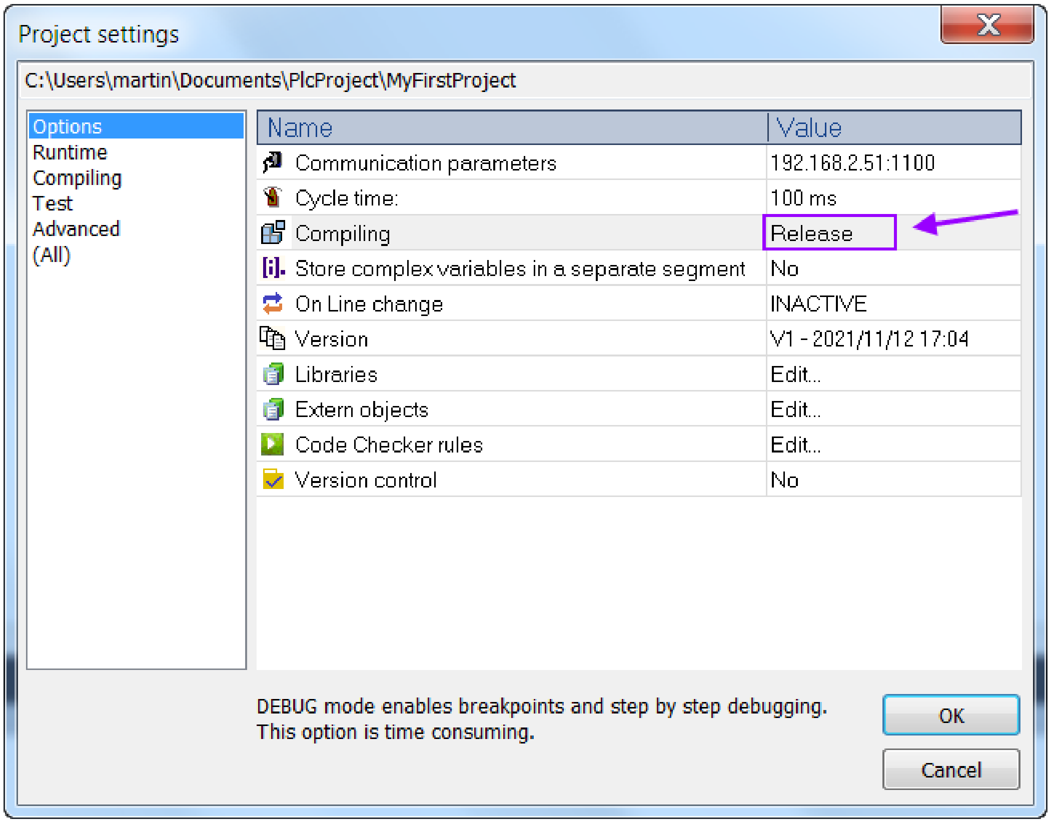
The workbench supports two types of code generation: '*Release*' and '*Debug*' mode.

Application compiled in '*Debug*' mode supports cycle by cycle execution, breakpoints and step by step debugging. Breakpoints can be placed anywhere in the source code of the application. The debugger also shows the call stack of the UDFBs and sub-programs when in step by step execution.

An application compiled in '*Debug*' mode includes additional information for stepping. This leads to bigger code size and less performances. It is recommended to compile your application in '*Release*' mode before delivering the final product to the customer.

Build application procedure:

Step 1: Determine whether to generate '*Release*' or '*Debug*' code.
Open the '*Project settings*' window by clicking the '*Project\Settings...*' command in the menu bar.
Select either '*Release*' or '*Debug*' by double clicking the 'Compiling' line in the '*Option*' category.



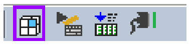
C:\Users\martin\Documents\PlcProject\MyFirstProject

Name	Value
Communication parameters	192.168.2.51:1100
Cycle time:	100 ms
Compiling	Release
Store complex variables in a separate segment	No
On Line change	INACTIVE
Version	V1 - 2021/11/12 17:04
Libraries	Edit...
Extern objects	Edit...
Code Checker rules	Edit...
Version control	No

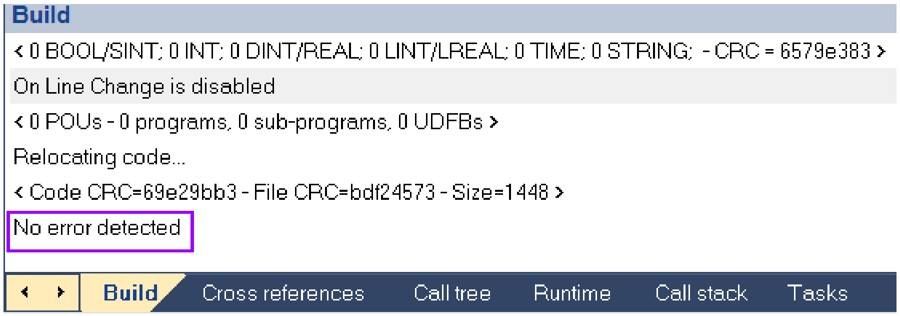
DEBUG mode enables breakpoints and step by step debugging.
This option is time consuming.

Step 2: Build the project:

- In the tool bar click the *'Build All projects'* button or
- *'Project/Build All projects'* in the menu bar or
- Press *'F7'*



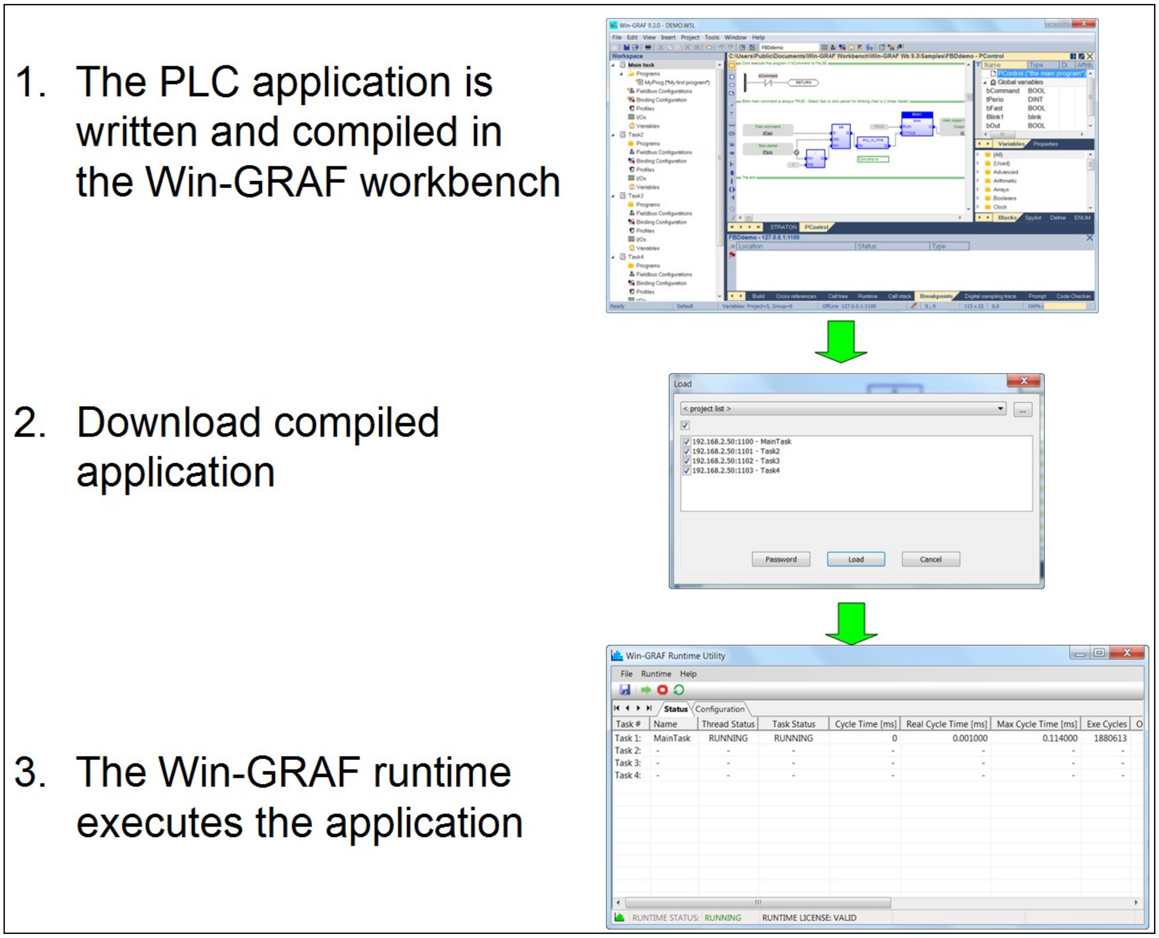
At the end of the build process the workbench indicates whether the build was successful or an error occurred:



```
< 0 BOOL/SINT; 0 INT; 0 DINT/REAL; 0 LINT/LREAL; 0 TIME; 0 STRING;  - CRC = 6579e383 >
On Line Change is disabled
< 0 POUs - 0 programs, 0 sub-programs, 0 UDFBs >
Relocating code...
< Code CRC=69e29bb3 - File CRC=bdf24573 - Size=1448 >
No error detected
```

4.6 Download Application

After the PLC application has been successfully compiled the application has to be downloaded to the runtime in order to be executed. The Win-GRAF workbench exchanges data with the runtime via TCP/IP communication.



1. The PLC application is written and compiled in the Win-GRAF workbench
2. Download compiled application
3. The Win-GRAF runtime executes the application

Load

< project list >

☑ 192.168.2.50:1100 - MainTask
☑ 192.168.2.50:1101 - Task2
☑ 192.168.2.50:1102 - Task3
☑ 192.168.2.50:1103 - Task4

Task #	Name	Thread Status	Task Status	Cycle Time [ms]	Real Cycle Time [ms]	Max Cycle Time [ms]	Exe Cycles
Task 1:	MainTask	RUNNING	RUNNING	0	0.001000	0.114000	1880613
Task 2:	-	-	-	-	-	-	-
Task 3:	-	-	-	-	-	-	-
Task 4:	-	-	-	-	-	-	-

RUNTIME STATUS: RUNNING RUNTIME LICENSE: VALID

Figure 56: Edit and download PLC program

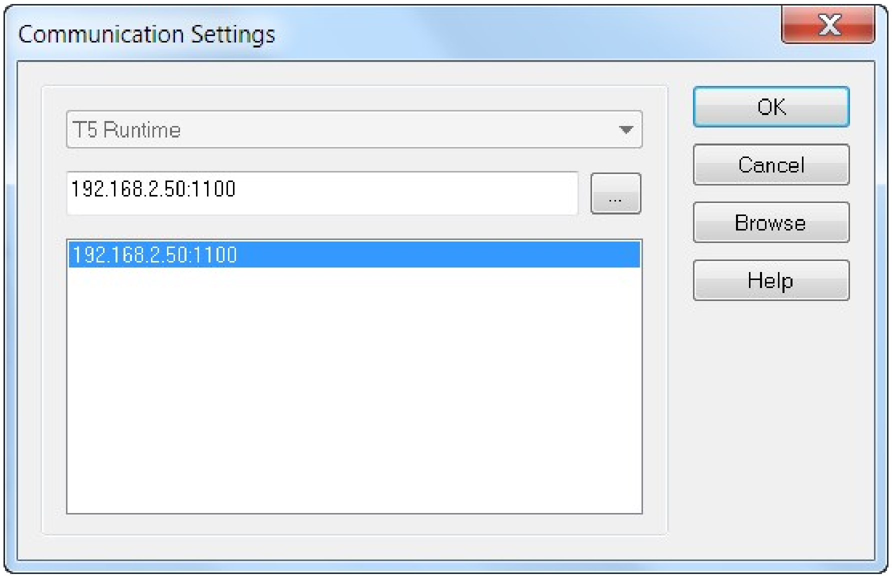
In order to establish a TCP/IP communication the workbench needs to know the IP address and the socket port number of the target runtime. Consult the user manual of the target device to determine how to set and get the communication configuration data.

Procedure for downloading the compiled application:

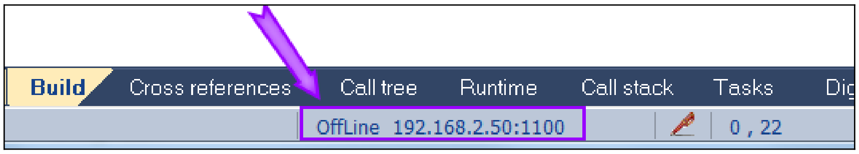
Step 1: Set the communication parameters between workbench and runtime:

1. Workbench: Set the IP address and socket port number of the target runtime.

- Select '*Tools/Communication Settings...*' from the menu or double click the '*Offline*' section in the status bar at the bottom of the window.
- Edit IP address and port number of the target runtime. Both parameters has to be separated by a colon. Only Ethernet TCP/IP communication is being supported.
- Click *'OK'*



The current communication setting is being displayed at the bottom of the screen. The setting can be directly modified by double clicking the IP address in the status bar.



Step 2: Runtime:

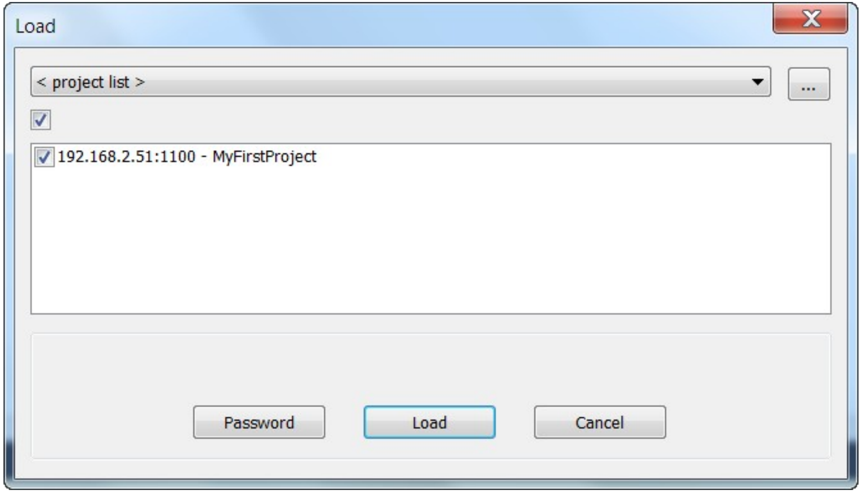
2. Make sure that the runtime on the target device has been started. Reference the device user manual regarding the runtime startup procedure setting.

3. Ensure that no IP collision exist on the network to prevent communication errors.

Step 3: Download the built plc application to the runtime:

- Click the download button in the toolbar or select '*Project\Download All Projects...*'.

- Click *'Load'* to start the download process



Wait until the download has finished.

ATTENTION:

The runtime stops running the current application before application files are downloaded.

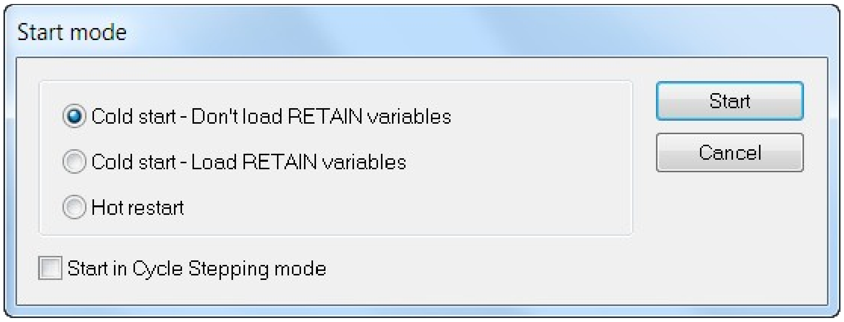
Note:

- After the download process has completed the application does not automatically restarted. This has to be done by the user. The user can decide between a cold or warm start. During a cold restart the PLC program begins again with the initial values while during a warm restart the program uses retentive data.
- Some libraries, e.g. PLCopen library, do not support warm start. Therefore make sure that all functions used in the source code support warm start before activating this start type.

Step 4: Create a online connection between workbench and the runtime by

- clicking the '*On Line*' button on the toolbar or
- enter '*Ctrl+F5*'.

Step 5: Start the PLC application in '*Cold start*' mode:
Click the '*Start*' button in the toolbar and select the '*Cold start*' option of the '*Start mode*' popup window and '*Start*' the PLC application.



Start mode

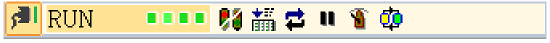
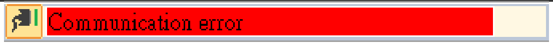
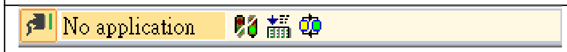
● Cold start - Don't load RETAIN variables

○ Cold start - Load RETAIN variables

○ Hot restart

□ Start in Cycle Stepping mode

Possible workbench online status:

Button	Description
 RUN	Download was successful and application runs correctly
 Communication error	• Runtime has not started • Incorrect communication parameters
 No application	• Application has not been downloaded or started yet. Output window shows more information about the cause

Function are available to directly manipulate a task:

- Stop a task and set the it again into idle mode
- Pause a task for one cycle
- Online change the cycle time for each task

Button	Description
▶	Start or stop task
⇩	Download program change
⇄	Online change
⏸	Pause (cycle to cycle)
⏱	Change cycle time

Table 7: Task commands

Variable monitoring are supported for all tasks. Next to each variable in the PLC program code the current value of the application in the runtime will be shown (Figure 57).

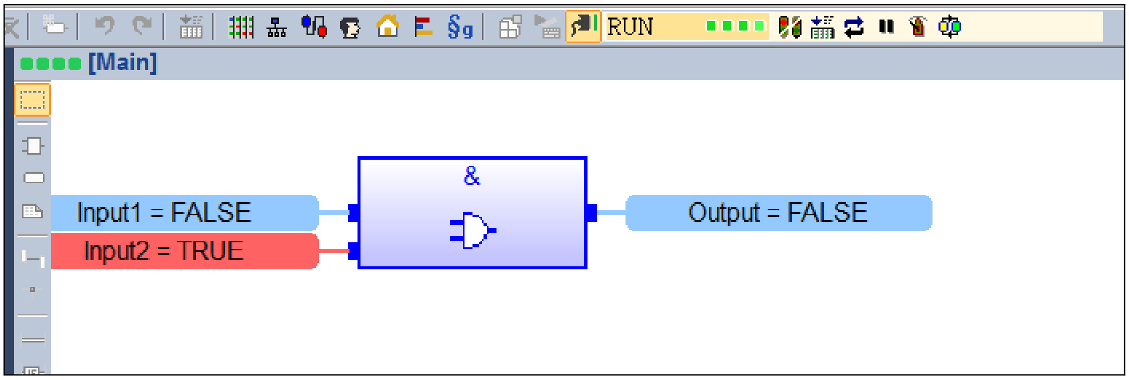
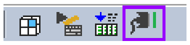


Figure 57: Workbench in online mode

4.7 Debugging

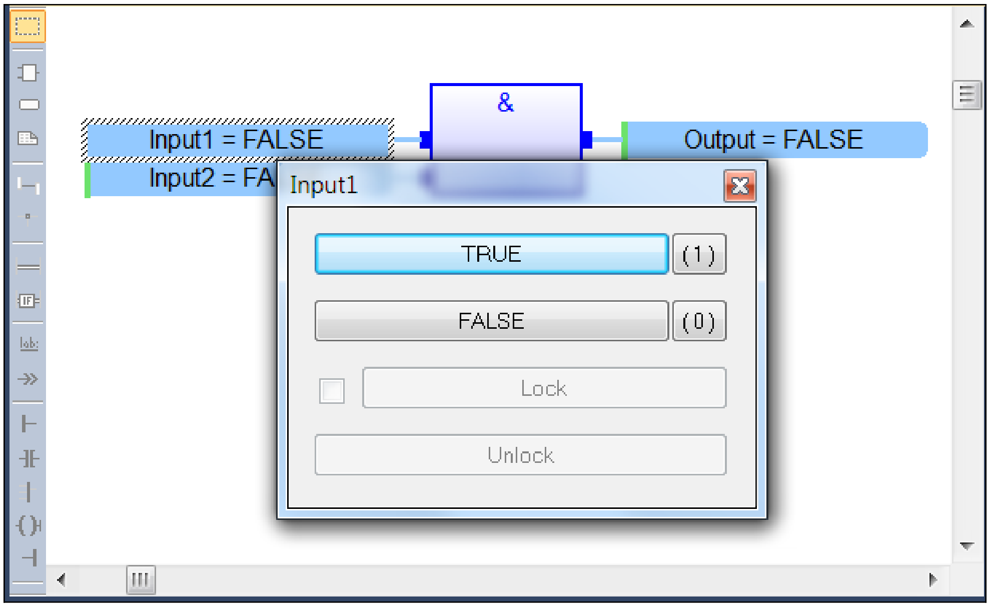
The workbench allows the user to directly change the variable values while the PLC application is running. The workbench has to be connected to the runtime to display the current variable value. This chapter describes how to monitor the PLC program and manipulate the variables via the workbench.

The following procedure describes how to directly modify a PLC variable via the workbench. It is assumed that the PLC application has already been download and is running.

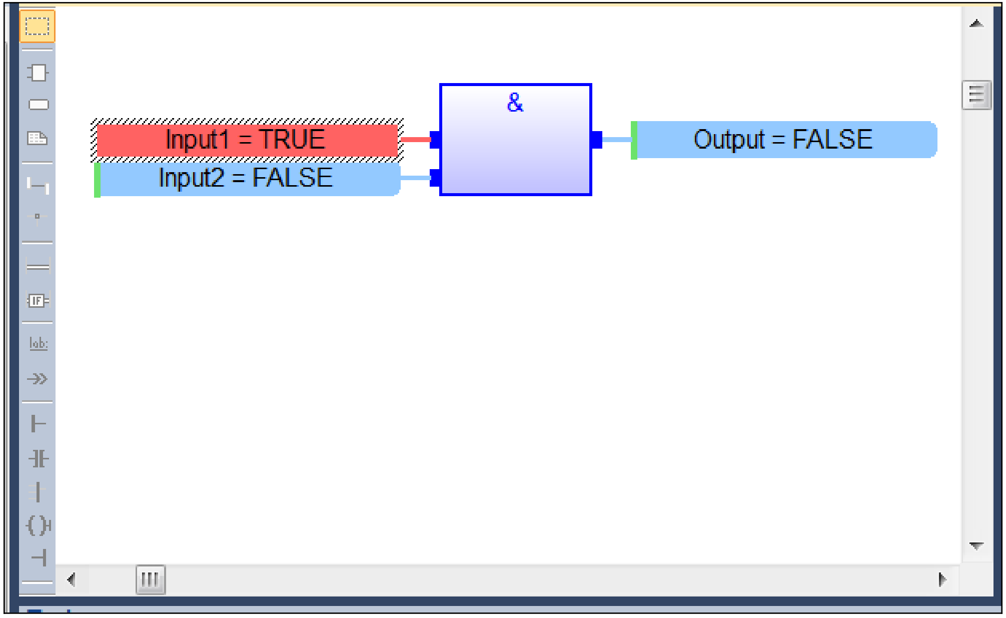
Step 1: Establish a TCP/IP connection between the workbench and runtime:
Click the '*Online*' button: 
After a connection has been established all the current values of each variables are displayed next to the variable names. These variable values are updates in each task cycle if the runtime is idling.

Step 2: Variable values can be directly changed via the workbench:

- Double click the '*Input1*' variable next to the function block and click the TRUE button of the popup window

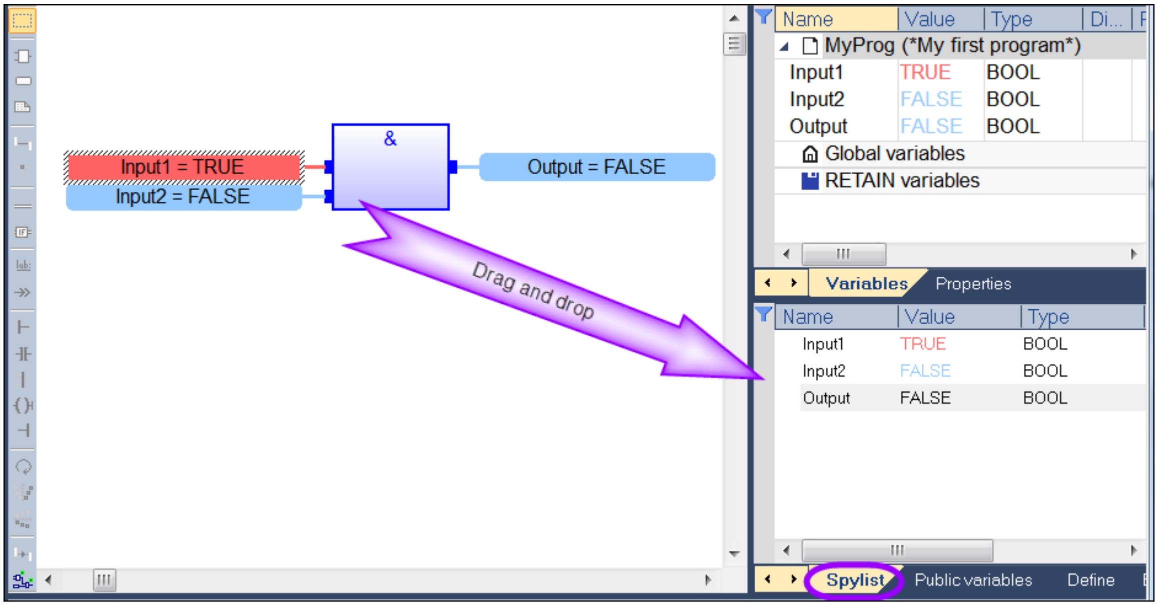


The input variable changes now from FALSE to TRUE:



All PLC data type can be manipulated in the described way. This allows direct testing of the PLC program.

The Spylist allows the user to add a number of variables to a monitoring list and thereby have a quick overview of all the relevant data. A variable is added to the Spylist by dragging it from the program editor and dropping it over the Spylist view. The Spylist shows the current variable value.

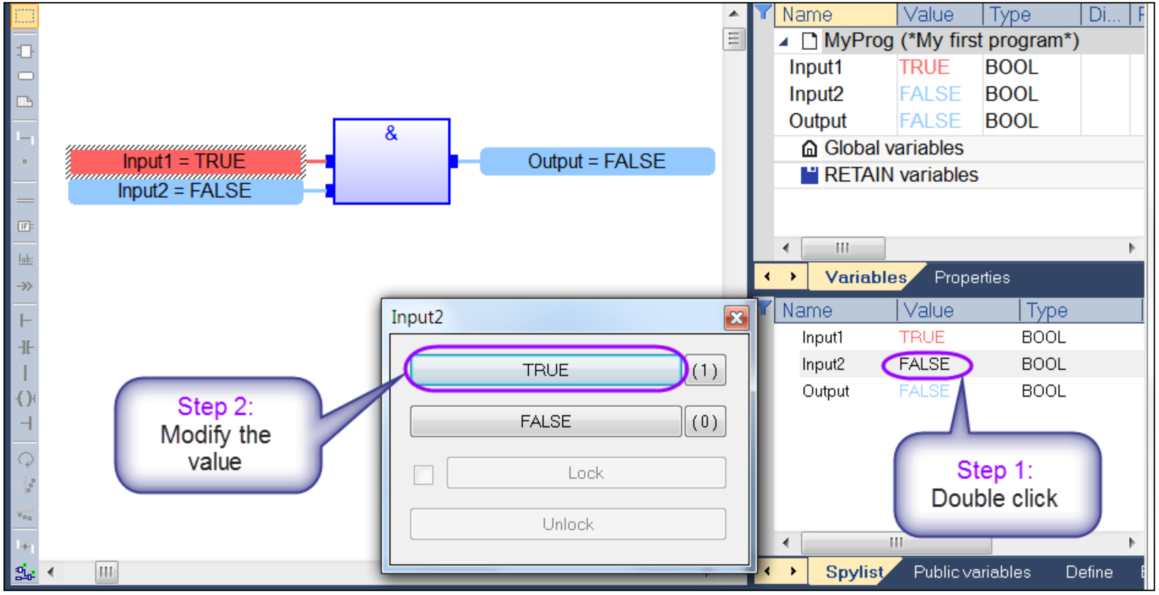


Name	Value	Type	Di...	F
MyProg (*My first program*)				
Input1	TRUE	BOOL		
Input2	FALSE	BOOL		
Output	FALSE	BOOL		
Global variables				
RETAIN variables				

Name	Value	Type
Input1	TRUE	BOOL
Input2	FALSE	BOOL
Output	FALSE	BOOL

Figure 58: Drag and drop variable to the Spylist

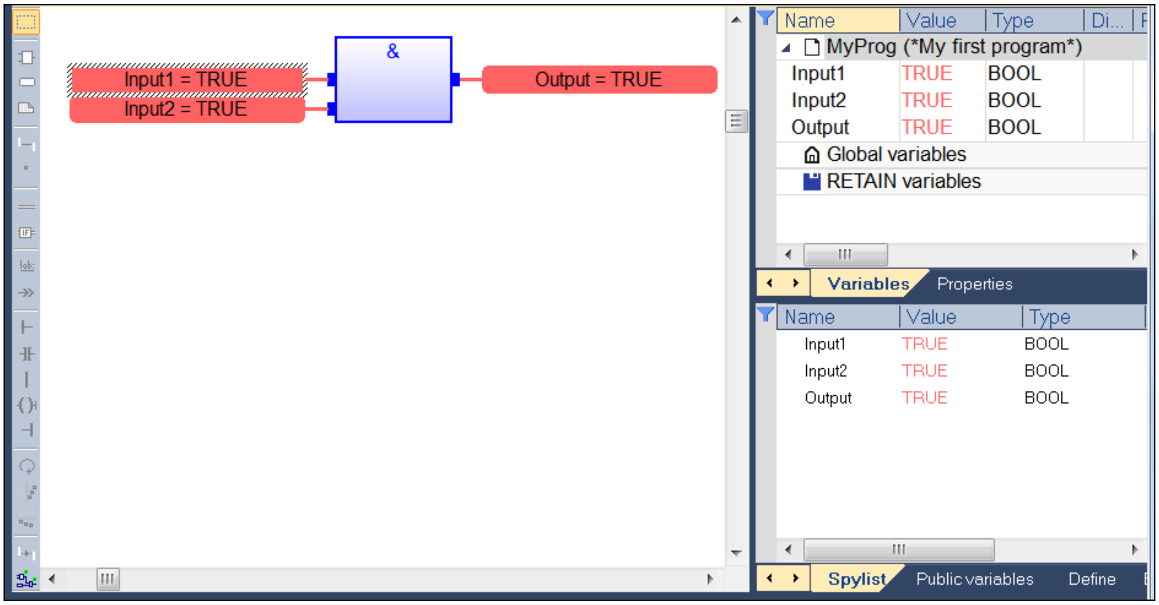
To change a variable value double click a value in the Spylist and enter a new value in the in the popup window.



Name	Value	Type	Di...	F
MyProg (*My first program*)				
Input1	TRUE	BOOL		
Input2	FALSE	BOOL		
Output	FALSE	BOOL		
Global variables				
RETAIN variables				

Name	Value	Type
Input1	TRUE	BOOL
Input2	FALSE	BOOL
Output	FALSE	BOOL

Figure 59: Change variable value via Spylist



Name	Value	Type	Di...
MyProg (*My first program*)			
Input1	TRUE	BOOL	
Input2	TRUE	BOOL	
Output	TRUE	BOOL	
Global variables			
RETAIN variables			

Name	Value	Type
Input1	TRUE	BOOL
Input2	TRUE	BOOL
Output	TRUE	BOOL

5 Multi-Tasking

The Win-GRAF runtime supports multitasking programming. The advantage of a multi-tasking projects is that different operation and action within an application can be subdivided according their execution priority and be assigned and executed by a PLC task with the required priority level. The prioritization of the different operation allows a efficient execution of the application. For example a task responsible for controlling the trajectory a servo motor has greater priority than a task which updates the HMI or OPC UA client with the current motion control status such as current position, velocity, etc.. Too many task on the other hand slows down the system due to the switching time between task. It is therefore important to find the right number of task to achieve the optimal performance of the system.
The number of task supported by the Win-GRAF varies across the ICPDAS hardware platforms. Refer to the user manual to determine the supported task number.

The user interface for the single- and multitasking environment provided by the workbench differs. For example '*Workspace*' and '*Output*' window displays different items for the different environment. Single tasking program can be edited in a multitasking environment but some single tasking functions (e.g. redundancy) are not supported.

Is important to note that

- a multi-tasking application can only run on a target platform which supports multi-tasking functionality.

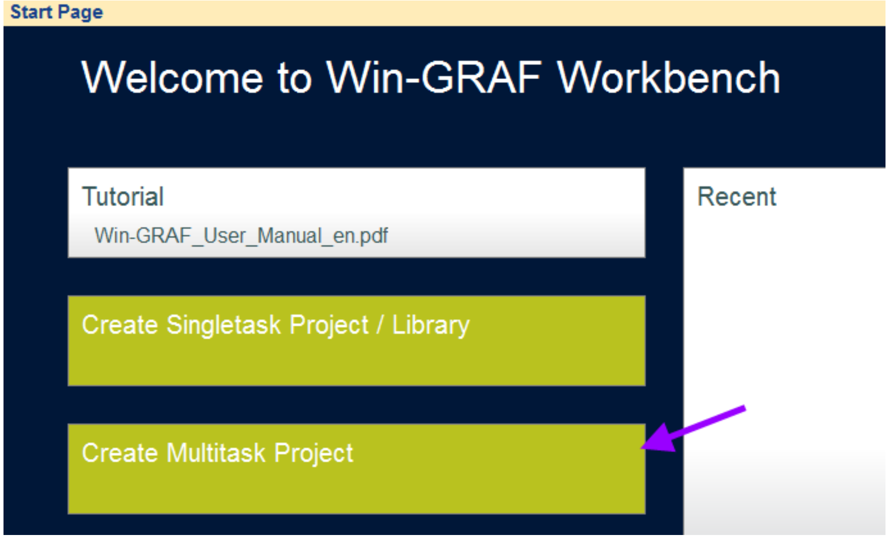
- a single-tasking application can run on a multi-tasking platform
- the multi-tasking environment does not support redundancy. Use the single-tasking workbench environment to implement redundancy functions if the target system support redundancy.
- the multi-tasking environment allows you to implement a single-tasking application without redundancy support

5.1 Create a Project

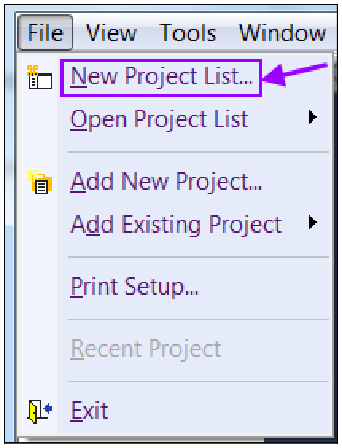
Create a multitasking project procedure:

Step 1: Open the project list to set the destination folder and name of the project and select the number of tasks. This can be done in two ways:

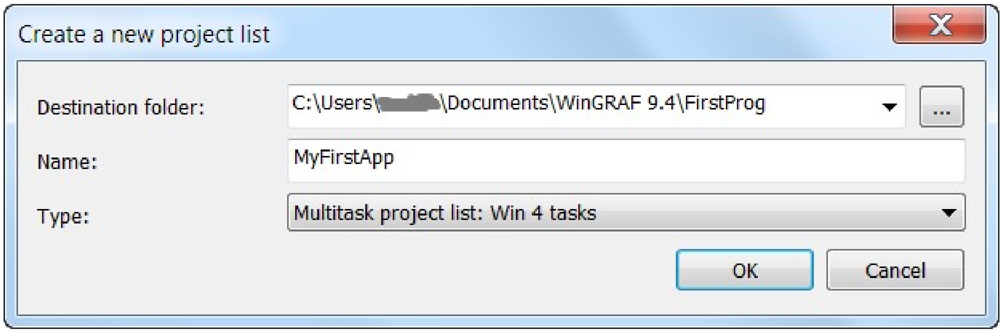
1. Via the '*Start Page*' by clicking the green '*Create Multitask Project*' command button.


Start Page
Welcome to Win-GRAF Workbench
Tutorial
Win-GRAF_User_Manual_en.pdf
Create Singletask Project / Library
Create Multitask Project
Recent

2. By executing the '*File / New project list*' menu command



Step 2: Enter the destination folder, project name and select the type of runtime. The runtime type basically determines the number of tasks used for the application. Confirm the setting with '*OK*'.

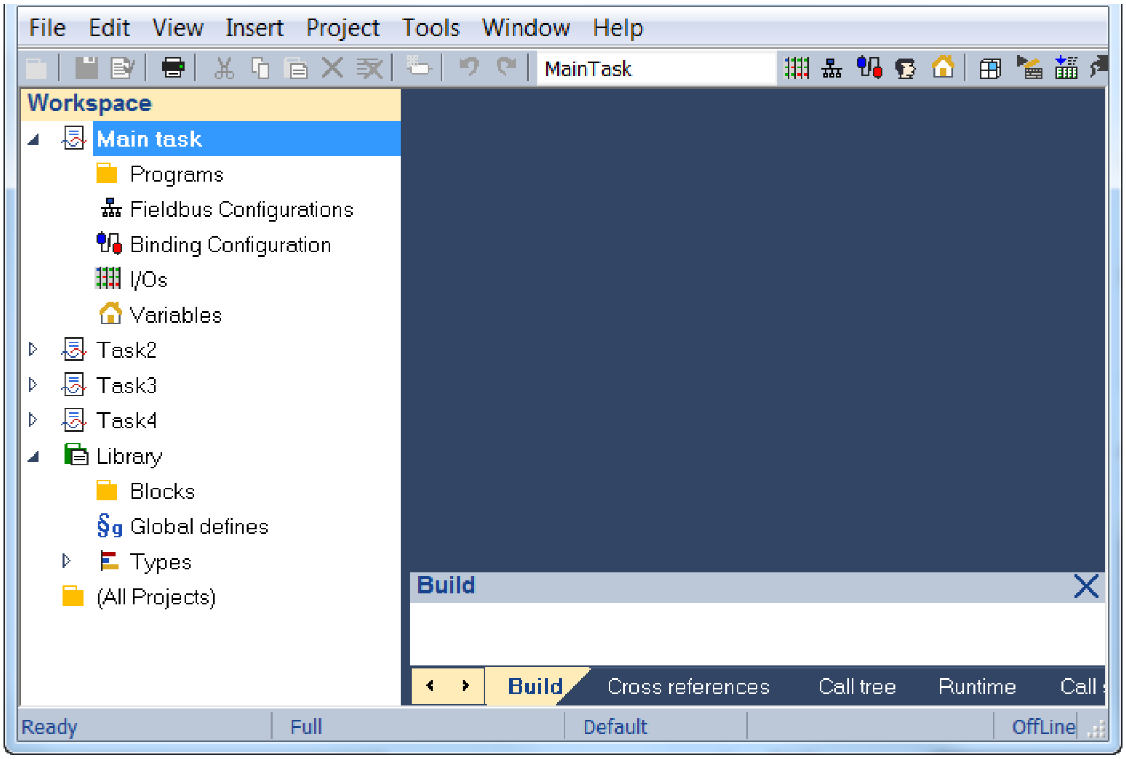

Create a new project list

Destination folder: C:\Users\[redacted]\Documents\WinGRAF 9.4\FirstProg

Name: MyFirstApp

Type: Multitask project list: Win 4 tasks

A workspace with the selected number of tasks and a shared library is created:

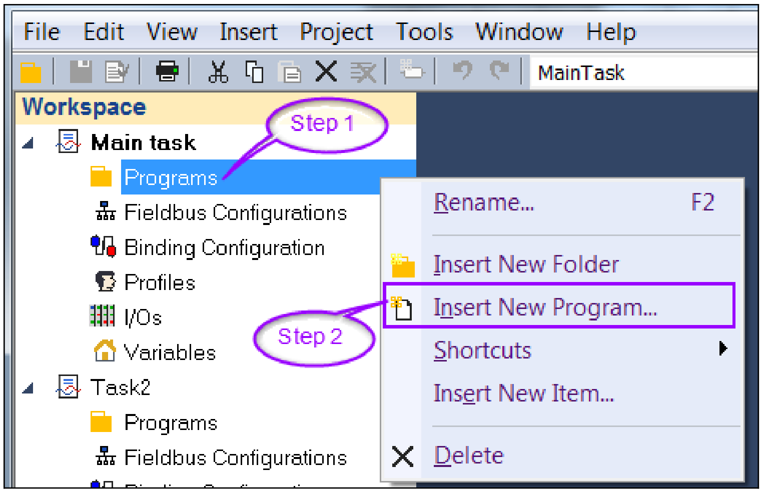


- Each '*task*' appears as a separate programming environment, including POUs, variables, I/O and Fieldbus configurations.
- You cannot remove or add tasks from the '*Workspace*' after a project has been created. Only the '*Main task*' will start automatically once the application starts executing. Within in the main task you have to programmable set the priority '*SYSCFGTASK(TaskNo)*' and start the execution '*SYSSTARTTASK(TaskNo)*' of the other tasks. Therefore if you do not want to use some tasks listed in the '*Workspace*' just do not execute it by not calling '*SYSSTARTTASK(TaskNo)*' in any of the active tasks and it will remain inactive.
- You can add a program to each task and implement the execution logic.

5.2 Create and Edit a Program

This chapter describes how to proceed to implement a PLC program using the multi-tasking environment.

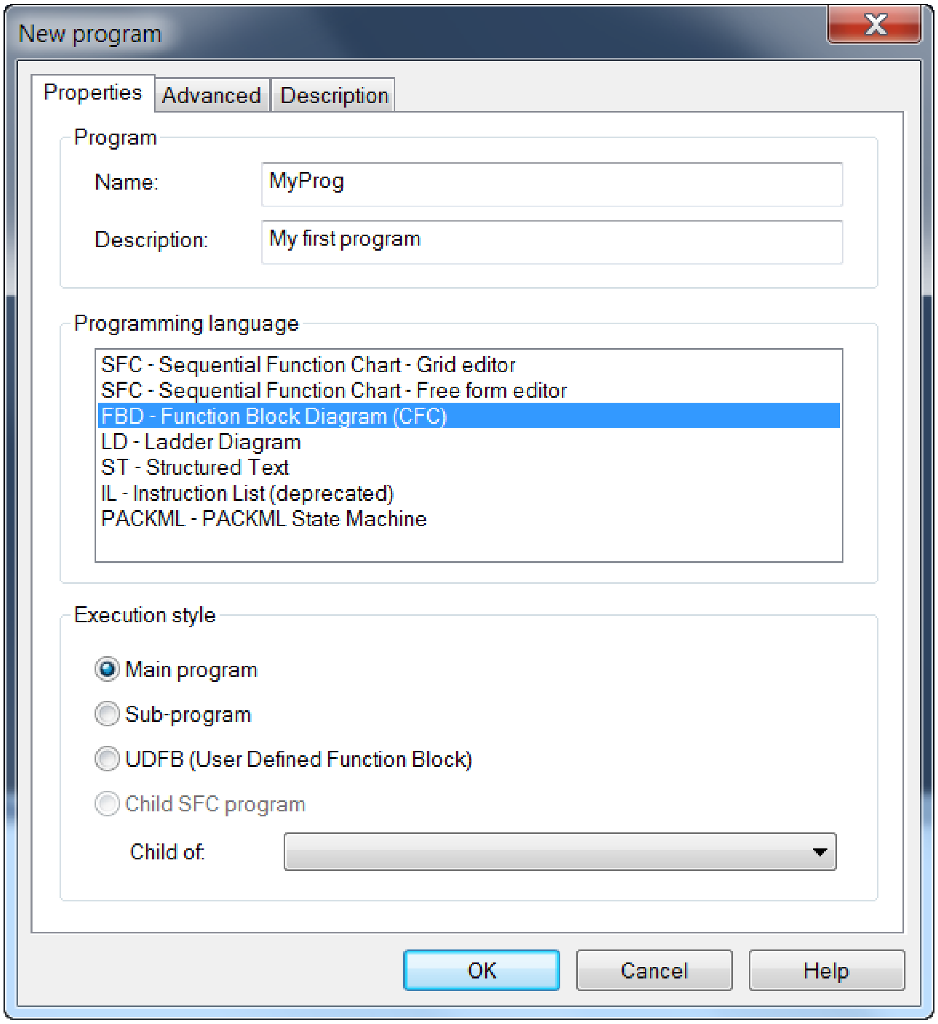
Step 1: Insert a new program to implement the PLC logic. Right click the program folder of the main task and select '*Insert New Program...*' from the popup menu.



Step 2: Make the following entries:

- Enter a name for the program
- Give a short program description (optional)
- Select one of the five IEC61131 programming languages (SFC, FBD, LD, ST, IL) for the program. Remember that more than one program can be added to the task. Each program can be programmed in a different language.

For this tutorial the '*FBD-Function Block Diagram (CFC)*' is selected. Add the program to the workspace by clicking '*OK*'.



Program

Name: MyProg

Description: My first program

Programming language

- SFC - Sequential Function Chart - Grid editor
- SFC - Sequential Function Chart - Free form editor
- FBD - Function Block Diagram (CFC)
- LD - Ladder Diagram
- ST - Structured Text
- IL - Instruction List (deprecated)
- PACKML - PACKML State Machine

Execution style

● Main program

○ Sub-program

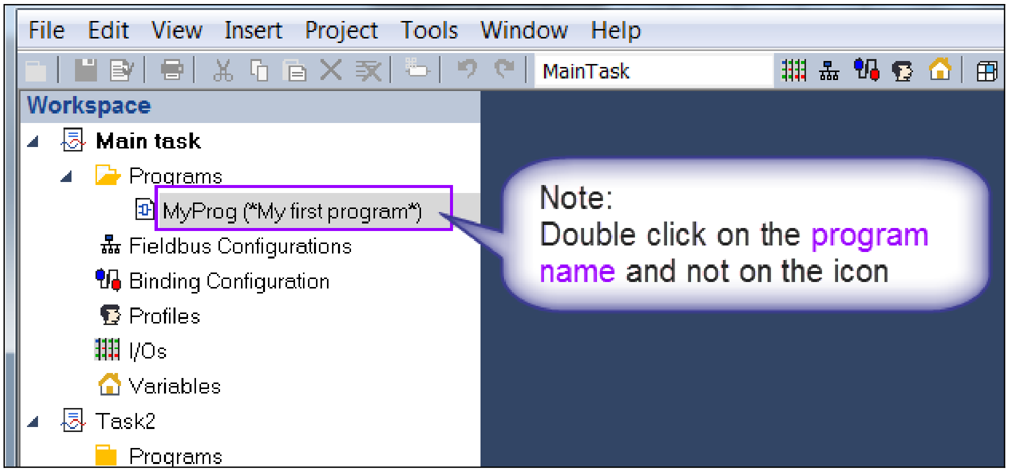
○ UDFB (User Defined Function Block)

○ Child SFC program

Child of:

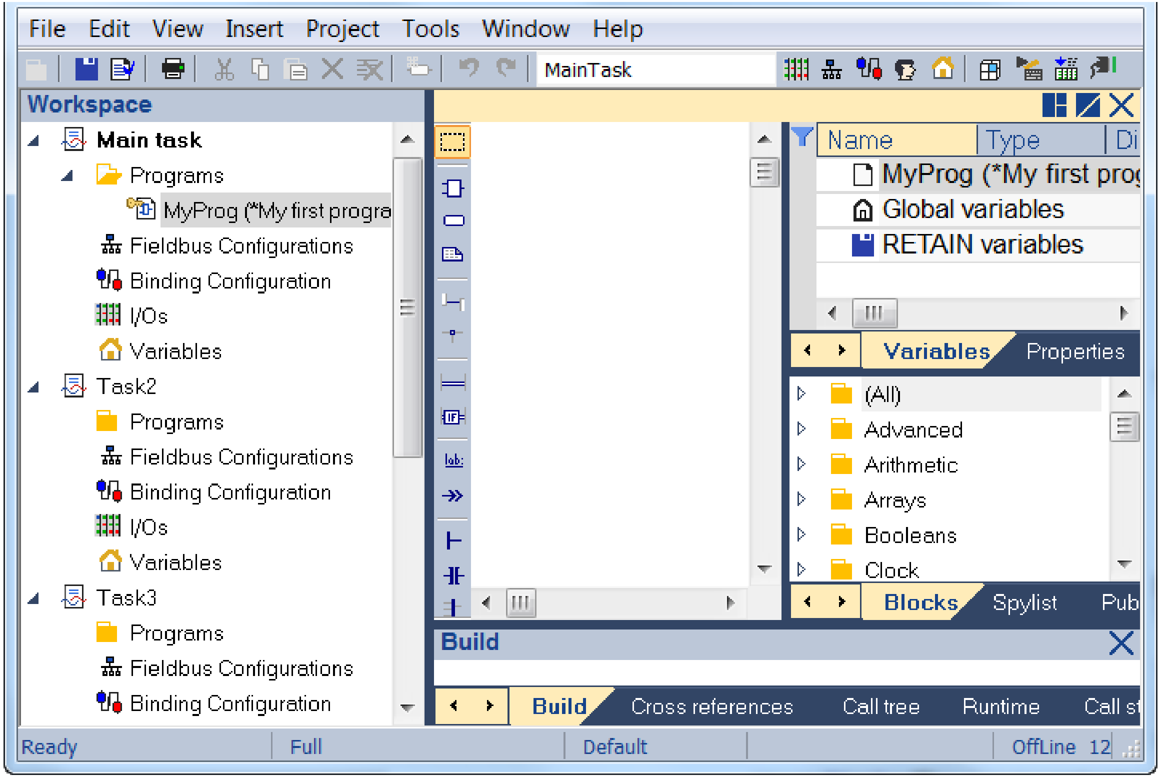
Programs must have unique names. The name cannot be a reserved keyword of the programming languages and cannot have the same name as a function or function block. A variable should not have the same name as a declared data type. The name of a program should begin either by a letter or an underscore ('_'), followed by letters, digits or underscores. It is not allowed to put two consecutive underscores within a name. Naming is case insensitive. Two names with different cases are considered as the same.

Step 3: Double click the added program in the workspace to open the program editor:

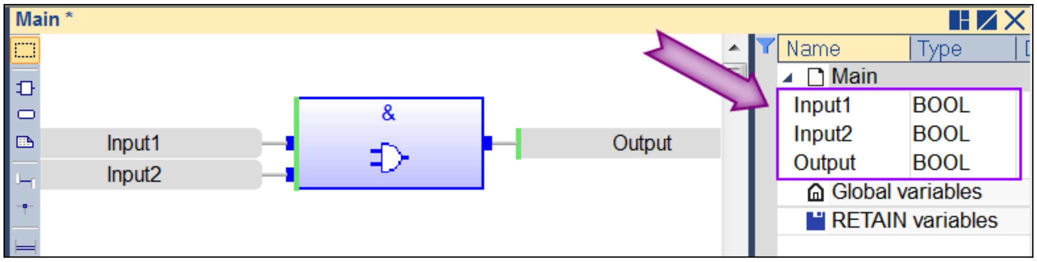


Note:
Double click on the program name and not on the icon

The PLC logic can now be edited using the FBD language. In the following steps it is demonstrated how to add a function block to the editor and declare its in- and output variables.

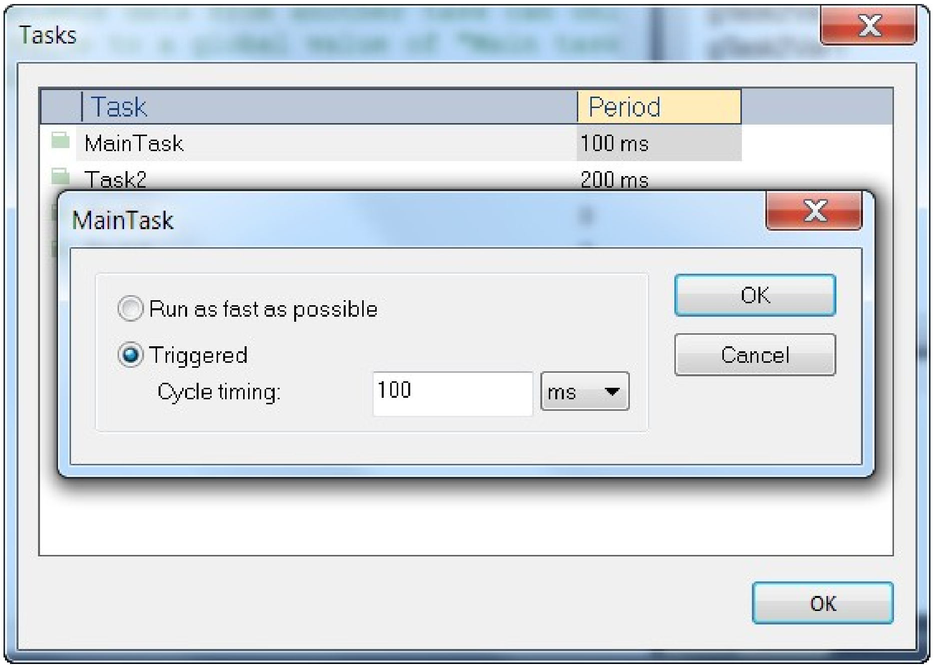


Step 4: Add programming logic to the program. The procedure for declaring variable and adding function blocks to the editor is the same as described in chapter 4.2 for single-tasking programming environment. This example only implements a '*AND*' function as shown in chapter 4.2.



5.3 Task Setting

The cycle of each task has to be directly set via the workbench by right clicking the task name and selection '*Task...*' from the pop-up menu. Double click the period column next to the task to enter the cycle time. If only one task is being used then the '*Run as fast as possible*' option can be used (Figure 60). Chapter 4.4 provides more information regarding the task and program execution settings.


Tasks

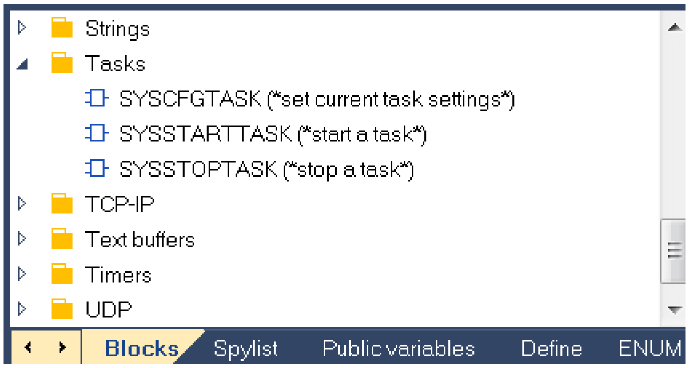
Task	Period
MainTask	100 ms
Task2	200 ms

MainTask

○ Run as fast as possible
● Triggered
Cycle timing: 100 ms

Figure 60: Task cycle time configuration

In a multi-tasking environment the '*Main task*' is the first task to be started when the PLC program starts to execute. The other tasks has to be started from the '*Main task*' or from a task which is already running by calling the `SYSSTARTTASK()` function. Additional functions are provided for configuring and terminating each task within the PLC program (Figure 61).



Strings
Tasks
SYSCFGTASK (*set current task settings*)
SYSSTARTTASK (*start a task*)
SYSSTOPTASK (*stop a task*)
TCP-IP
Text buffers
Timers
UDP
Blocks
Spylist
Public variables
Define
ENUM

Figure 61: Task dedicated functions

The task library provides the function for configuring and controlling a task:

- To start another task:
  `SYSSTARTTASK( Task(*DINT*), Warm(*BOOL*) )`
  This function should only be called once for starting a task.
- To stop a running task:
  `SYSSTOPTASK( Task(*DINT*) )`
  A task can be stopped by any running task.
- Configure a running task:
  `SYSCFGTASK( Prio(*DINT*), Opts(*STRING*) )`
  This function should only be called after a task has been started. It can only be called by a program owned by the task to be configured. It sets the task priority inside a running task. The '*Prio*' and '*Opts*' variable definition depends on the target platform.

For Windows PC:

- Prio

Windows supports the following priority classes:

Priority Classes	Description
IDLE_PRIORITY_CLASS	
BELOW_NORMAL_PRIORITY_CLASS	
NORMAL_PRIORITY_CLASS	
ABOVE_NORMAL_PRIORITY_CLASS	
HIGH_PRIORITY_CLASS	
REALTIME_PRIORITY_CLASS	Default setting of the Win-GRAF runtime

By default the priority class of the Win-GRAF runtime (PC) is set to REALTIME_PRIORITY_CLASS. This setting is fixed and can not be changed.

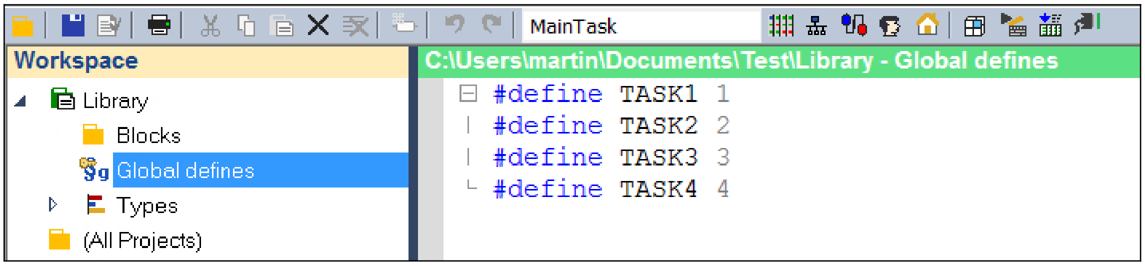
Thread Priority Level	Prio	Description
THREAD_PRIORITY_IDLE	-15	
THREAD_PRIORITY_LOWEST	-2	
THREAD_PRIORITY_BELOW_NORMAL	-1	
THREAD_PRIORITY_NORMAL	0	

THREAD_PRIORITY_ABOVE_NORMAL	1	
THREAD_PRIORITY_HIGHEST	2	
THREAD_PRIORITY_TIME_CRITICAL	15	

All tasks are created using THREAD_PRIORITY_TIME_CRITICAL. The `SYSCFGTASK` internally calls the SetThreadPriority() Windows API to adjust its priority relative to other threads in the process. To keep the priority setting simple it is suggested to just select between THREAD_PRIORITY_NORMAL for the normal task and THREAD_PRIORITY_TIME_CRITICAL for the high priority task.

- Opts:
  This parameter is not supported by the Win-GRAF runtime for Windows. Therefore just enter an empty string (' ').

Each task is identified by a number from 1 to N (1 is the main task). Predefined aliases are configured in the shared library:



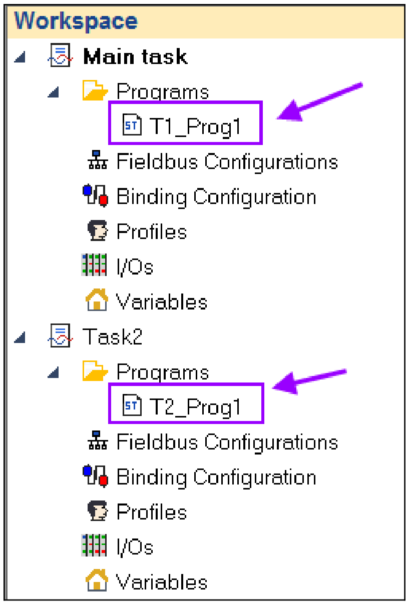
```
#define TASK1 1
#define TASK2 2
#define TASK3 3
#define TASK4 4
```

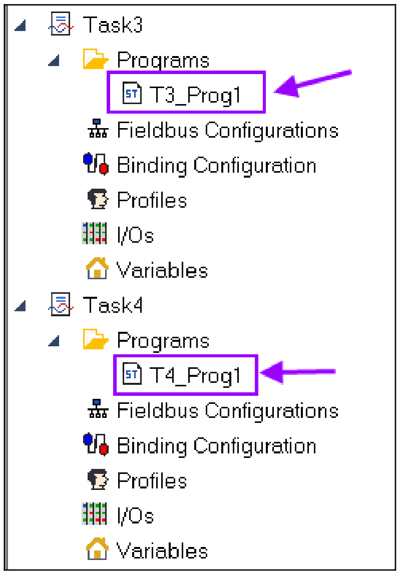
Figure 62: Task definitions

You cannot remove or add tasks. The number of available tasks is defined by the runtime system. If a task is unused, it means that it is simply not started by the main task.

Example of creating a multitasking project:
This example uses a startup exception program for starting the other task in the '*Main task*'. Task can be started and terminated from any program within the main task.

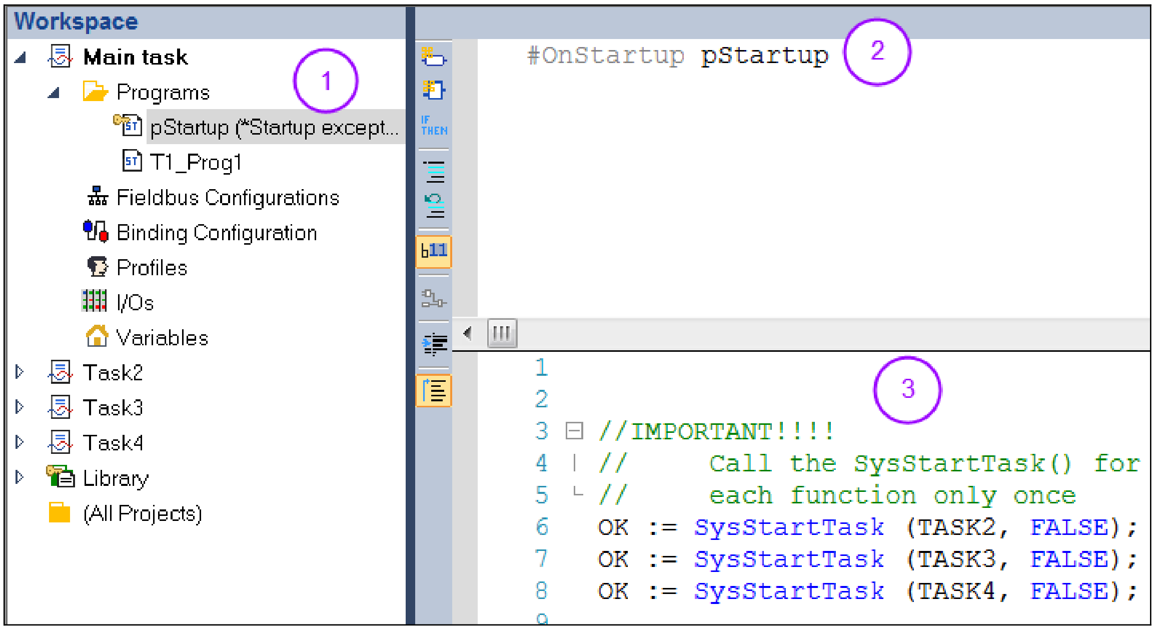
Step 1: First add program(s) to the tasks which will be used for the PLC application. It is necessary to always add at least one program to the Main task, otherwise the PLC application will not run. The other task can remain empty if they are not used. In this example a program is added to all four tasks.





Step 2: Add a Startup exception program to the '*Main task*':

1. Create an exception program in the Main task and assign it a name ('*pStartup*'). The exception program is created like a normal program and can have any name and can be programmed in any language except SFC.
2. Add the line **`#OnStartup ProgramName`** to the define editor of the startup exception program. In this example the startup program name is '*pStartup*', therefore the line '`#OnStartup pStartup`' has to be added.
3. Call the function `SYSSTARTTASK()` to start the execution of the specified task.



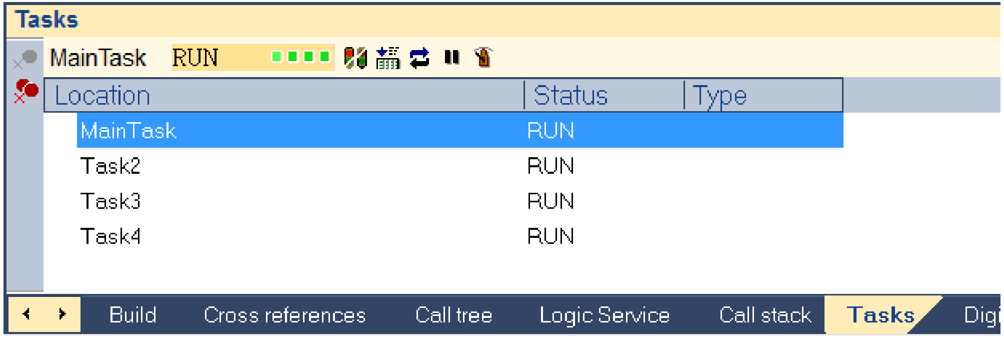
```
#OnStartup pStartup
```

```


//IMPORTANT!!!!
//      Call the SysStartTask() for
//      each function only once
OK := SysStartTask (TASK2, FALSE);
OK := SysStartTask (TASK3, FALSE);
OK := SysStartTask (TASK4, FALSE);
```

Step 3: Set the priority of each task

The task priority can only be set by a program which is controlled by the task itself. For example: In order to set the priority of the '*Task2*' it is necessary to call the SYSCFGTASK(Prio, Opts) in one of the '*Task2*' program ('T2_Prog1') once. It is possible to change the task priority again if required. The task starts with the default priority and continuous to run at this priority if not set otherwise.

The workbench displays the running task in the output window:



Tasks

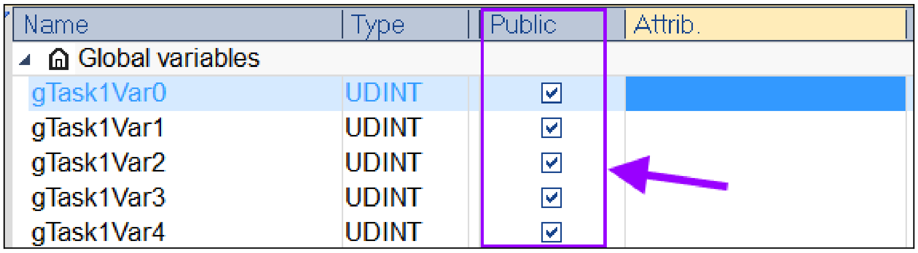
MainTask RUN

Location	Status	Type
MainTask	RUN	
Task2	RUN	
Task3	RUN	
Task4	RUN	

It is possible to manually start and stop task using the tools shown in the '*Tasks'* tab of the output window.

5.4 Data Sharing between Tasks

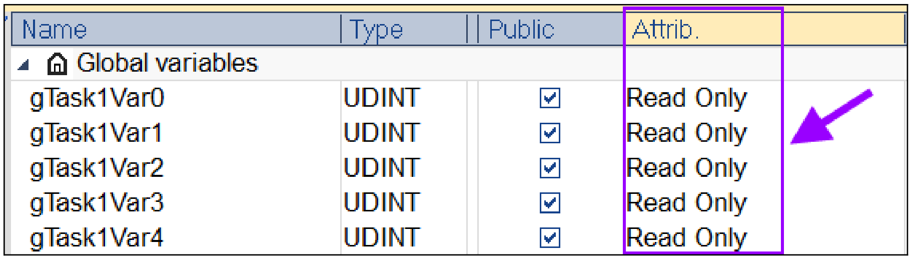
Global and retain variables can be shared among tasks by declaring it as a public variable. Enable the check box in the public column next to the declared variable in the variable editor. The shared variable can both be read and modified by all task (Figure 63).



Name	Type	Public	Attrib.
Global variables			
gTask1Var0	UDINT	☑	
gTask1Var1	UDINT	☑	
gTask1Var2	UDINT	☑	
gTask1Var3	UDINT	☑	
gTask1Var4	UDINT	☑	

Figure 63: Public variable with read and write access

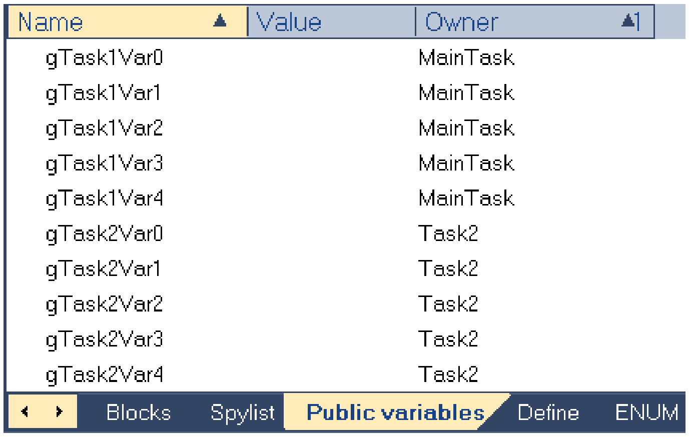
If the variable should only to be modified by the task in which it was declared then select the '*Read Only*' attribute. This attribute setting allows only the owner task to change the variable and restricts the other tasks to read access only (Figure 64).



Name	Type	Public	Attrib.
Global variables			
gTask1Var0	UDINT	☑	Read Only
gTask1Var1	UDINT	☑	Read Only
gTask1Var2	UDINT	☑	Read Only
gTask1Var3	UDINT	☑	Read Only
gTask1Var4	UDINT	☑	Read Only

Figure 64: Public variable with read access

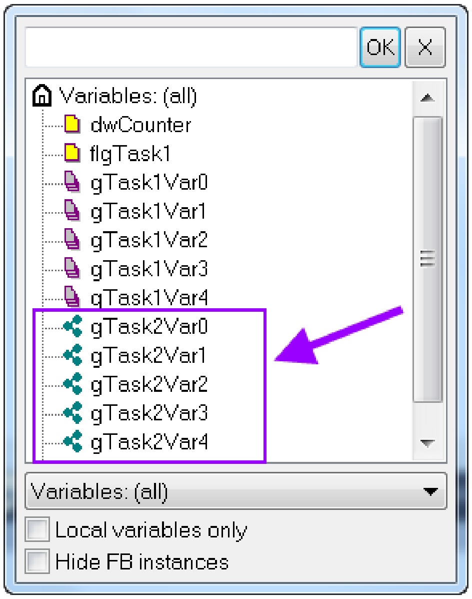
Shared variables together with the owner task are listed in the '*Public Variables*' tab of the Info window.



Name	Value	Owner
gTask1Var0		MainTask
gTask1Var1		MainTask
gTask1Var2		MainTask
gTask1Var3		MainTask
gTask1Var4		MainTask
gTask2Var0		Task2
gTask2Var1		Task2
gTask2Var2		Task2
gTask2Var3		Task2
gTask2Var4		Task2

Figure 65: List of shared variables from all task

Variables published by other tasks are also available from the variable selection box. Shared variables from other tasks are indicated by a dark cyan three dot icon:



Variables: (all)
- dwCounter
- flgTask1
- gTask1Var0
- gTask1Var1
- gTask1Var2
- gTask1Var3
- gTask1Var4
- gTask2Var0
- gTask2Var1
- gTask2Var2
- gTask2Var3
- gTask2Var4

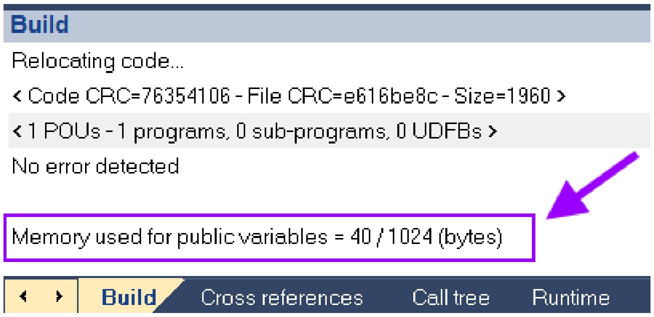
Variables: (all)

□ Local variables only

□ Hide FB instances

Figure 66:Shared variables from other

The amount of memory available for shared variables is limited to 65536 bytes for the Win-GRAF runtime. At the end of each build process the workbench shows the number of bytes used for the public variables by the PLC program:



```
Build
Relocating code...
< Code CRC=76354106 - File CRC=e616be8c - Size=1960 >
< 1 POUs - 1 programs, 0 sub-programs, 0 UDFBs >
No error detected

Memory used for public variables = 40 / 1024 (bytes)
```

Figure 67: Size of public variables in byte

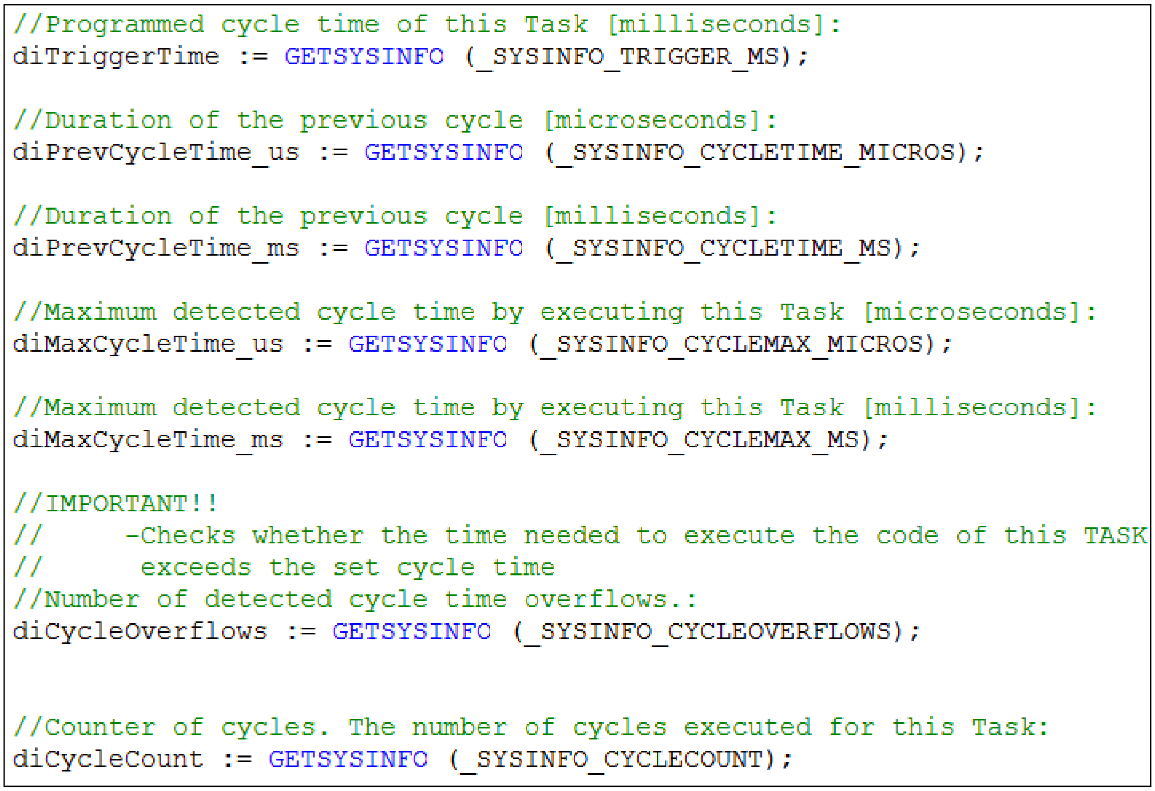
5.5 Get System Information

The '*GetSYSINFO()*' function provides additional information about the current state of the runtime and its task. Not all types are supported by every platform. Table 8 list the type of available system information and Figure 68 shows its implementation in structured text diagram.

Info Parameter	Description
_SYSINFO_TRIGGER_MICROS	Programmed cycle time in micro-seconds.
_SYSINFO_TRIGGER_MS	Programmed cycle time in milliseconds.
_SYSINFO_CYCLETIME_MICROS	Duration of the previous cycle in micro-seconds.
_SYSINFO_CYCLETIME_MS	Duration of the previous cycle in milliseconds.
_SYSINFO_CYCLEMAX_MICROS	Maximum detected cycle time in micro-seconds.
_SYSINFO_CYCLEMAX_MS	Maximum detected cycle time in milliseconds.
_SYSINFO_CYCLESTAMP_MS	Time stamp of the current cycle in milliseconds (platform dependent).
_SYSINFO_CYCLEOVERFLOWS	Number of detected cycle time overflows.
_SYSINFO_CYCLECOUNT	Counter of cycles.
_SYSINFO_APPVERSION	Version number of the application.
_SYSINFO_APPSTAMP	Compiling date stamp of the application.
_SYSINFO_CODECRC	CRC of the application code.
_SYSINFO_DATACRC	CRC of the application symbols.
_SYSINFO_FREEHEAP	Available space in memory heap (bytes)
_SYSINFO_DBSIZE	Space used in RAM (bytes)
_SYSINFO_ELAPSED	Seconds elapsed since startup
_SYSINFO_CHANGE_CYCLE	Indicates a cycle just after an On Line Change

_SYSINFO_WARMSTART	Non zero if RETAIN variables were loaded at the last start
_SYSINFO_NBLOCKED	Number of locked variables
_SYSINFO_NBBREAKPOINTS	Number of installed breakpoints
_SYSINFO_BIGENDIAN	Non zero if the runtime processor is big endian
_SYSINFO_DEMOAPP	Non zero if the application was compiled in DEMO mode
_SYSINFO_SIMUL	Returns 0 in case of a normal runtime. Returns 1 in case of a simulator.

Table 8: System information



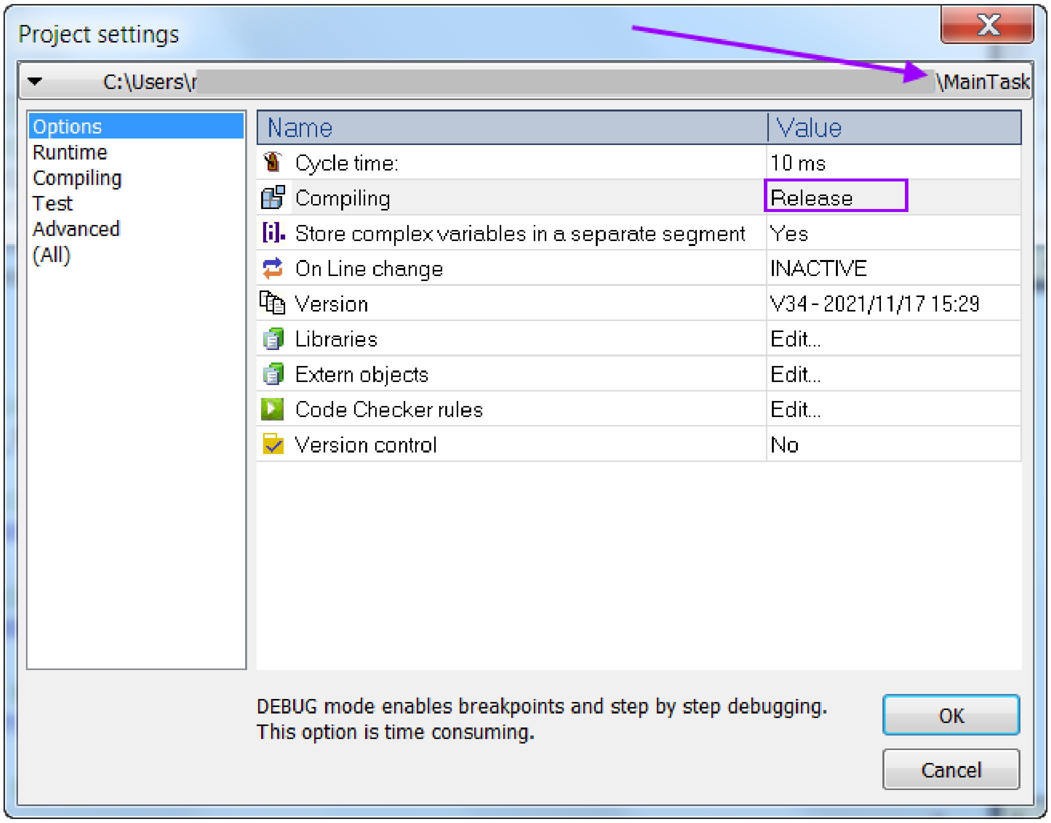
```
//Programmed cycle time of this Task [milliseconds]:
diTriggerTime := GETSYSINFO (_SYSINFO_TRIGGER_MS);

//Duration of the previous cycle [microseconds]:
diPrevCycleTime_us := GETSYSINFO (_SYSINFO_CYCLETIME_MICROS);

//Duration of the previous cycle [milliseconds]:
diPrevCycleTime_ms := GETSYSINFO (_SYSINFO_CYCLETIME_MS);

//Maximum detected cycle time by executing this Task [microseconds]:
diMaxCycleTime_us := GETSYSINFO (_SYSINFO_CYCLEMAX_MICROS);

//Maximum detected cycle time by executing this Task [milliseconds]:
diMaxCycleTime_ms := GETSYSINFO (_SYSINFO_CYCLEMAX_MS);

//IMPORTANT!!
//      -Checks whether the time needed to execute the code of this TASK
//       exceeds the set cycle time
//Number of detected cycle time overflows.:
diCycleOverflows := GETSYSINFO (_SYSINFO_CYCLEOVERFLOWS);


//Counter of cycles. The number of cycles executed for this Task:
diCycleCount := GETSYSINFO (_SYSINFO_CYCLECOUNT);
```

Figure 68: Reading system information in a FBD language

5.6 Build/Compile Application

The workbench build procedure of the multi-tasking application is nearly identical to the single-tasking application as described in chapter 4.5. The only difference is that it is necessary to configure each task separately whether it should be compiled in release or debug mode.

In the '*Project settings*' the task to be set has to be selected from the drop list before configuring the compiling mode (Figure 69).



Name	Value
Cycle time:	10 ms
Compiling	Release
Store complex variables in a separate segment	Yes
On Line change	INACTIVE
Version	V34 - 2021/11/17 15:29
Libraries	Edit...
Extern objects	Edit...
Code Checker rules	Edit...
Version control	No

DEBUG mode enables breakpoints and step by step debugging.
This option is time consuming.

Figure 69: Compiling mode setting

5.7 Download Application

Only the difference between single- and multi-tasking environment will be point out in this chapter. For more information see chapter 4.6.

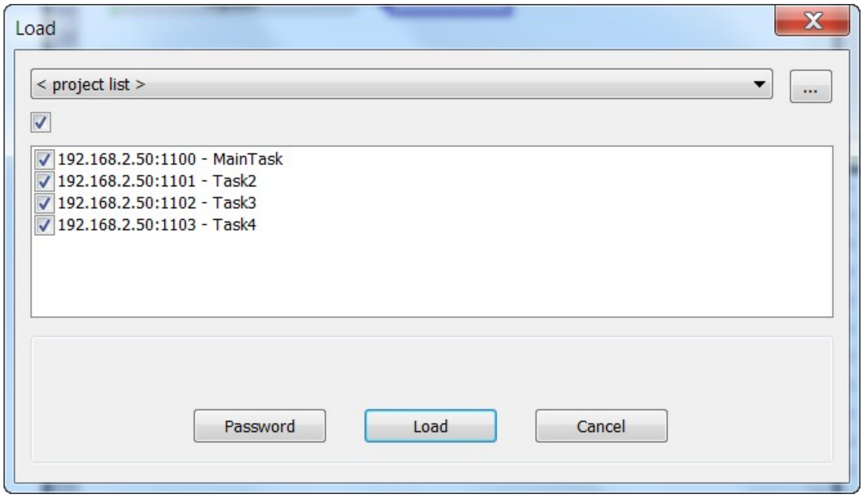
In order to establish a TCP/IP communication the workbench needs to know the IP address and the socket port number of the target runtime. Each PLC task communicate through its own port number. By default the port number of the '*Main task*' is set to 1100. The port number of the each task are incremented by one in the sequential order of the task number: port 1101 for Task2, port 1102 for Task3, etc.. If required for some platforms the port number of the '*Main task*' and thereby the other tasks can be changed via the Win-GRAF Runtime Utility.

Download a built multi-tasking application to the runtime:

Step 1: Click the download button in the toolbar or select '*Project\Download All Projects...*'.

1. The prompted dialog list all the task from the project to be downloaded.
2. Click '*Load*' button to start the download process. Wait until the download

has finished.


Load
< project list >
...
☑
☑ 192.168.2.50:1100 - MainTask
☑ 192.168.2.50:1101 - Task2
☑ 192.168.2.50:1102 - Task3
☑ 192.168.2.50:1103 - Task4
Password
Load
Cancel

ATTENTION:

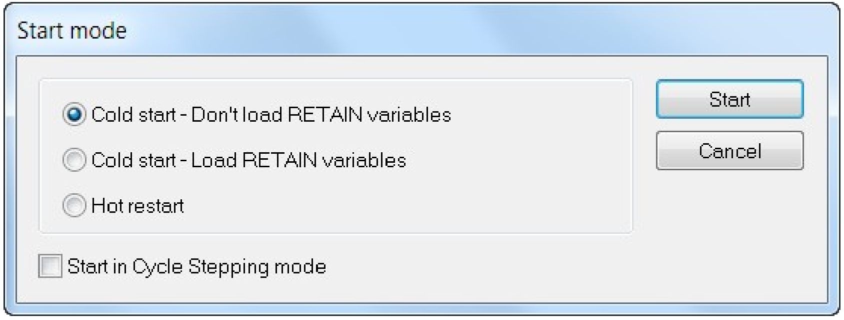
The runtime stops all running tasks before the download process starts.

Note:

- By default all tasks files are selected for download. For large programs to reduce the download time deselect the task files which have not been changed since the last download.
- After the download process has been completed the application does not automatically restarted. This has to be done by the user. The user can decide between a cold or warm restart. During a cold restart the PLC program begins again with the initial values while during a warm restart the program uses retentive data.

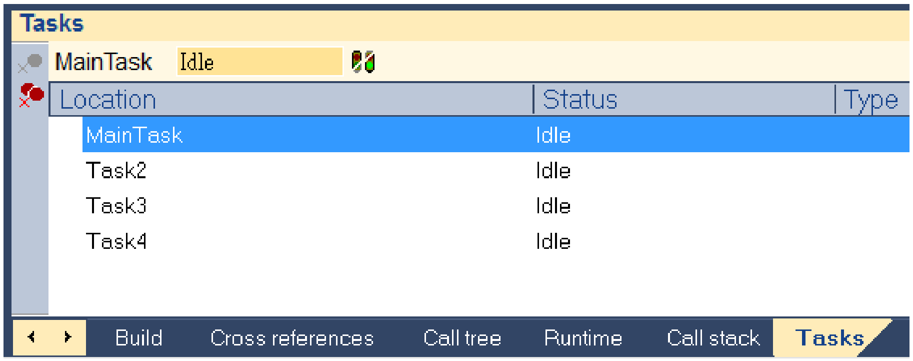
Step 2: Connect the workbench to the runtime via Ethernet by clicking the '*On Line*' button on the toolbar or using the keyboard shortcut '*Ctrl+F5*'.

Step 3: Start the PLC application in '*Cold start*' mode:
Normally the '*Start mode*' window automatically pops up right after creating a online connection. In the '*Start mode*' window select the first '*Cold start*' option and click the '*Start*' button. Now the PLC application starts running.


Start mode
● Cold start - Don't load RETAIN variables
○ Cold start - Load RETAIN variables
○ Hot restart
□ Start in Cycle Stepping mode
Start
Cancel

If the *'Start mode'* window does not pop up during the previous step, then you can open this window as follows:

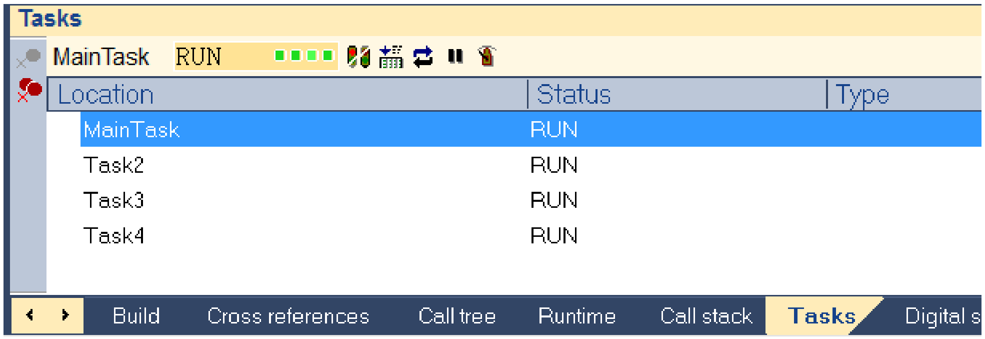
- Go to the '*Tasks*' tab of output window at the bottom. All tasks are shown in idle mode.



Location	Status	Type
MainTask	Idle	
Task2	Idle	
Task3	Idle	
Task4	Idle	

- First select the '*MainTask*' item by clicking it and click the traffic light button. Now the '*Start mode*' window should pops up, which allows you to select the start mode of the PLC application.

The main task will start executing the other tasks if the function SYSSTARTTASK() is being called from the main task. Each task can be manually started by selecting it in the '*Tasks*' tab output window and activating the traffic light button.



Location	Status	Type
MainTask	RUN	
Task2	RUN	
Task3	RUN	
Task4	RUN	

Figure 70: Tasks status tab

As for single-tasking the same function are available to directly manipulate each task:

- Stop a task and set the it again into idle mode
- Pause a task for one cycle
- Online change the cycle time for each task

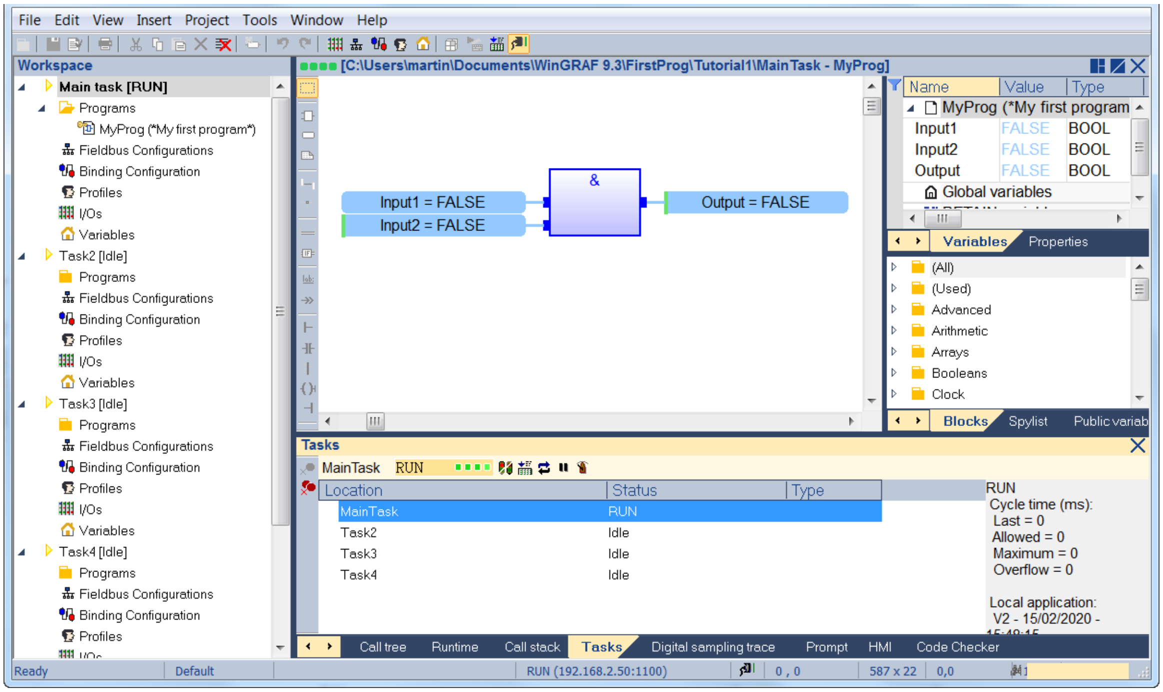
First select the task you like to control in the '*Tasks*' tab (Figure 70) and then activate one of the commands (Table 9).

Button	Description
🚦	Start or stop task
⬇	Download program change
⇄	Online change

⏸	Pause (cycle to cycle)
⏱	Change cycle time

Table 9: Task commands

Variable monitoring are supported for all tasks. In the program code the current value of each variable will be shown. For example in the FBD editor all the current in- and output variable values are shown next to the function block and in the variable view (Figure 71). Messages from each tasks are displayed in the output window.



Name	Value	Type
MyProg (*My first program		
Input1	FALSE	BOOL
Input2	FALSE	BOOL
Output	FALSE	BOOL
Global variables		

Location	Status	Type
MainTask	RUN	
Task2	Idle	
Task3	Idle	
Task4	Idle	

RUN
Cycle time (ms):
Last = 0
Allowed = 0
Maximum = 0
Overflow = 0

Local application:
V2 - 15/02/2020 -

Figure 71: Workbench in online mode

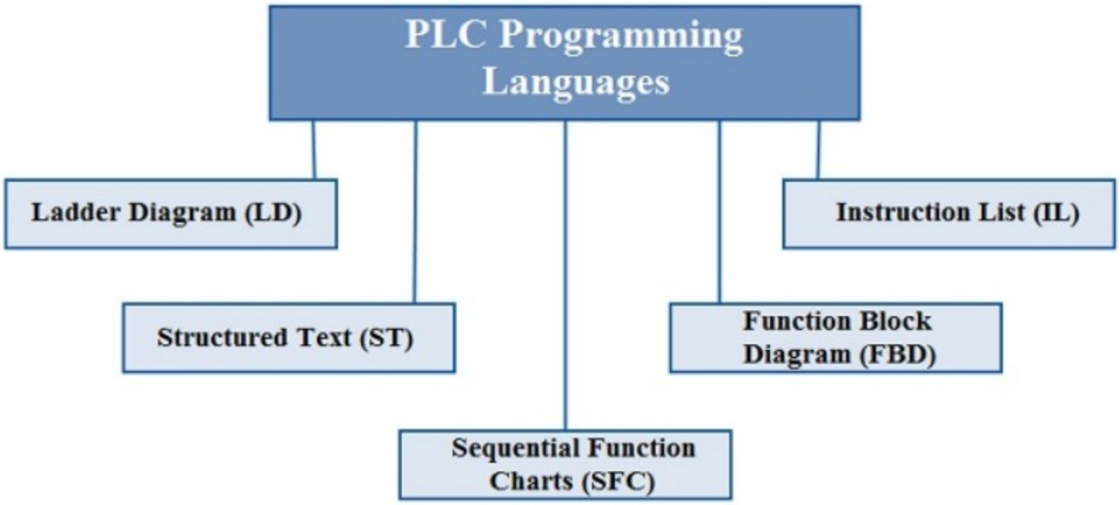
5.8 Debugging

The debugging function is identical to the single-tasking workbench environment (see chapter 4.7).

6 Editing Programs

The programming environment provide language dedicated editors for:

- Structure Text (ST)
- Function Block Diagram (FBD)
- Ladder Diagram (LD)
- Sequential Function Chart (SFC) and
- Instruction List (IL)

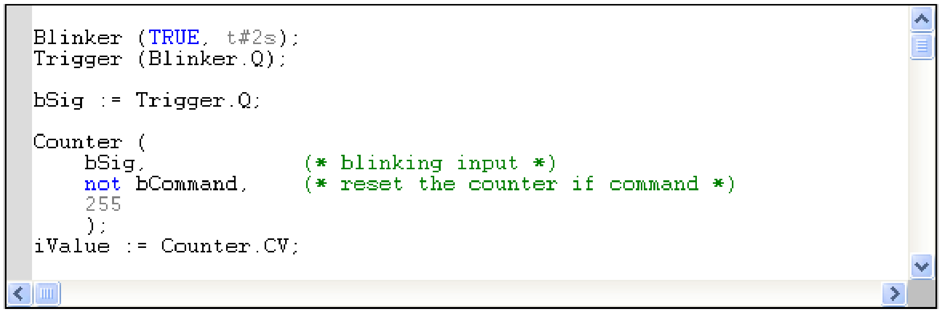

PLC Programming Languages
Ladder Diagram (LD)
Instruction List (IL)
Structured Text (ST)
Function Block Diagram (FBD)
Sequential Function Charts (SFC)

The editor provides you the ideal programming environment with drag and drop features:

- Drag a variable from the list to the program to insert it.
- Drag a definition to the program to insert its name.
- Drag a block in the program to insert it.
- Drag a function block to the variable list to declare an instance.
- Drag a variable from the program or from the variable list to the spy list.
- Double-click on a line of the output window to highlight the corresponding code.

6.1 Structured Text (ST) and Instruction List (IL) Editor

The ST / IL editor is a powerful language sensitive text editor dedicated to IEC 61131-3 languages. The editor supports advanced graphic features such as drag and drop, syntax coloring and active tooltips for efficient input and test of programs in ST and IL.

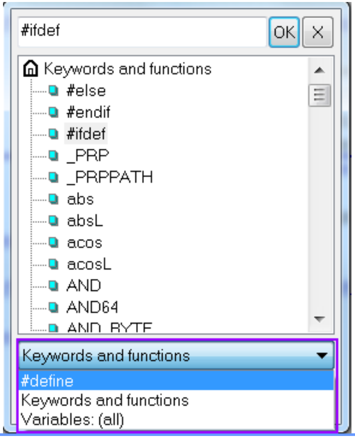


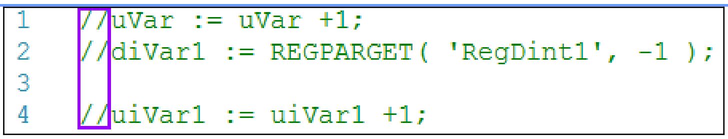
```
Blinker (TRUE, t#2s);
Trigger (Blinker.Q);

bSig := Trigger.Q;

Counter (
    bSig,              (* blinking input *)
    not bCommand,      (* reset the counter if command *)
    255
    );
iValue := Counter.CV;
```

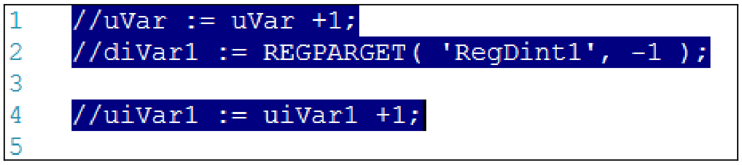
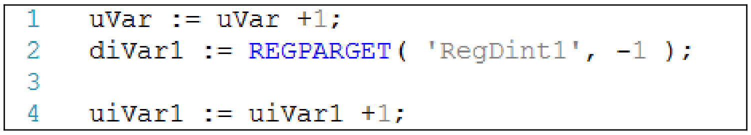
Figure 72: Structure text editor

The ST / IL editor also supports context sensitive help. Place the caret on a keyword or on the name of function or function block and hit F1 key to get help about the text.

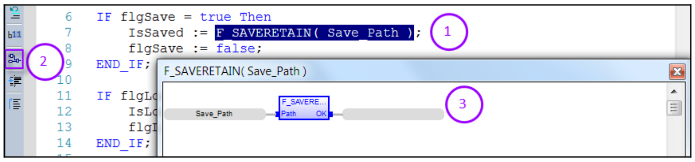
In ST and IL Language you can use the following commands of the vertical toolbar:

Icon	Function	Description
	Insert Variable	Opens the dialog to create or insert a variable at the current cursor position.
	Insert FB	Opens the **Select** dialog, to insert a function block.
IF THEN	List Key Words	Opens the dialog to select the selected content to add it to the program: - #define - Keywords and functions - Variables: (all)  #ifdef OK Keywords and functions #else #endif #ifdef _PRP _PRPPATH abs absL acos acosL AND AND64 AND_BYTE Keywords and functions #define Keywords and functions Variables: (all)
≡	Insert comment line (Ctrl+ K)	Changes an entry to an comment. Add two forward slashes '//' to the beginning of the line. For example mark the lines 1 to 4 with the mouse and click the insert comment button to add a double forward slash at the beginning of the line

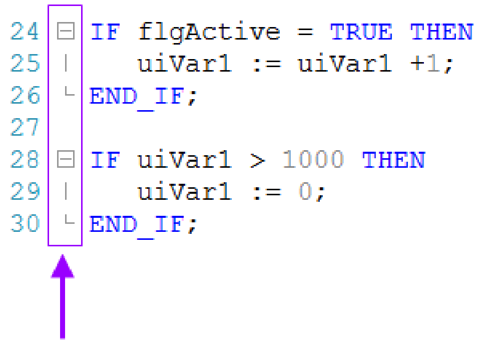
		
		`1 //uVar := uVar +1;`
		`2 //diVar1 := REGPARGET( 'RegDint1', -1 );`
		`3`
		`4 //uiVar1 := uiVar1 +1;`

	Remove comment (Ctrl + Shift + K)	Changes a comment to an normal entry.
		
		For example: Mark the line 1 to 4 and click the *'Remove comment'* button to remove the double forward slashes at the line start
		
		`1 //uVar := uVar +1;`
		`2 //diVar1 := REGPARGET( 'RegDint1', -1 );`
		`3`
		`4 //uiVar1 := uiVar1 +1;`
		`5`
		
		`1 uVar := uVar +1;`
		`2 diVar1 := REGPARGET( 'RegDint1', -1 );`
		`3`
		`4 uiVar1 := uiVar1 +1;`

	Show Value in Text	If enabled, the value of a variable is shown next to its name in the code.

	Show Expression	Allows to see a alternative graphical view for regular expressions in your code.
		1. Select the expression
		2. Press **Show Expression** to open a popup window
		3. that shows the expression in a graphical view.
		
		`6 IF flgSave = true Then`
		`7 IsSaved := F_SAVERETAIN( Save_Path );`
		`8 flgSave := false;`
		`9 END_IF;`
		`10`
		`11 IF flgLo`
		`12 IsLo`
		`13 flgI`
		`14 END_IF;`
		 1
		2
		F_SAVERETAIN(Save_Path)
		Save_Path
		F_SAVERE...
		Path
		OK
		3

	Indent text	Indents the selected text.
	Group/Ungroup Lines	Allows to group or ungroup text lines, based on used control structures (e.g. IF-ELSE) or multi-line comments. Grouped lines can be collapsed and expanded in the editors display.

```
IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
END_IF;

IF uiVar1 > 1000 THEN
    uiVar1 := 0;
END_IF;
```

Table 10: Vertical toolbar for ST and IL language

6.1.1 ST / IL Language Selection

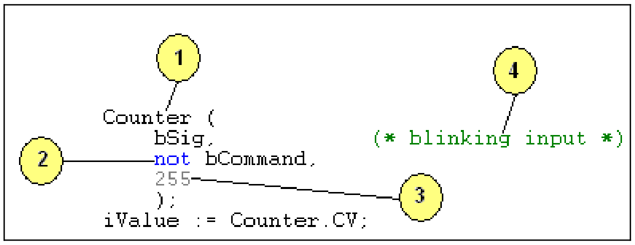
The Workbench allows you to mix ST and IL languages in textual program. ST is the default language. When you enter IL instructions, the program must be entered between *'BEGIN_IL'* and *'END_IL'* keywords, such as in the following example:

```
BEGIN_IL
   LD    var1
   ST    var2
END_IL
```

Figure 73: IL example

6.1.2 ST / IL Syntax Coloring

The ST / IL editor supports syntax coloring according to the selected programming language (ST or IL). The editor uses different colors for the following kind of words:

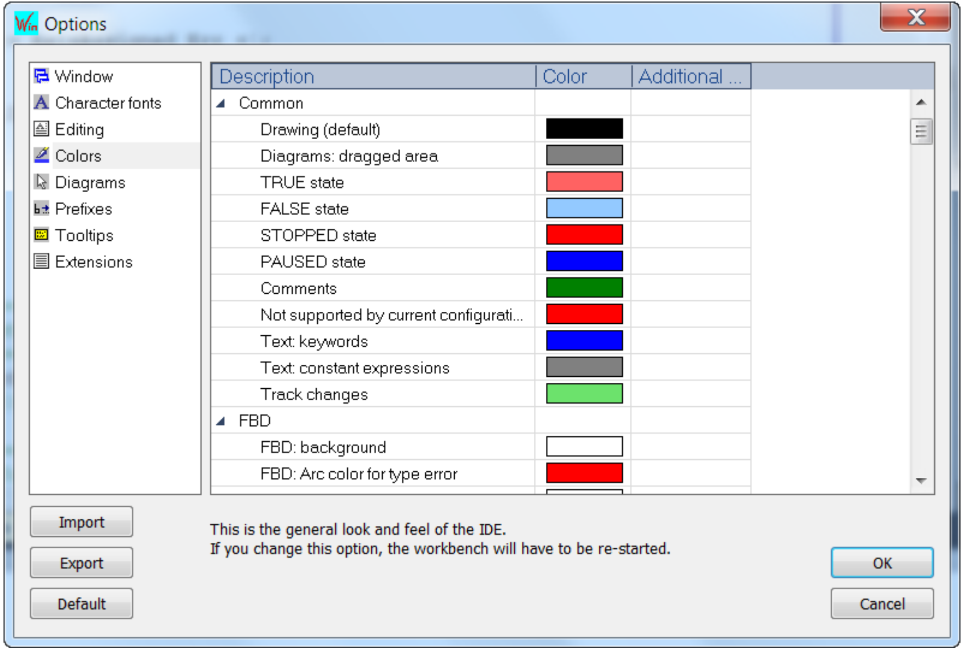


```
Counter (
    bSig,                       (* blinking input *)
    not bCommand,
    255
    );
iValue := Counter.CV;
```

Figure 74: Editor coloring

1. Default (identifiers, separators...)
2. Reserved keywords of the language
3. Constant expressions
4. Comments

The set of colors used can be changed from the Tools/Options menu command.



Description	Color	Additional ...
Common		
Drawing (default)	⬛	
Diagrams: dragged area	▨	
TRUE state	🟥	
FALSE state	🟦	
STOPPED state	🟥	
PAUSED state	🟦	
Comments	🟩	
Not supported by current configurati...	🟥	
Text: keywords	🟦	
Text: constant expressions	▨	
Track changes	🟩	
FBD		
FBD: background	⬜	
FBD: Arc color for type error	🟥	

This is the general look and feel of the IDE.
If you change this option, the workbench will have to be re-started.

Figure 75: Color setting option

6.1.2.1 Intellisense

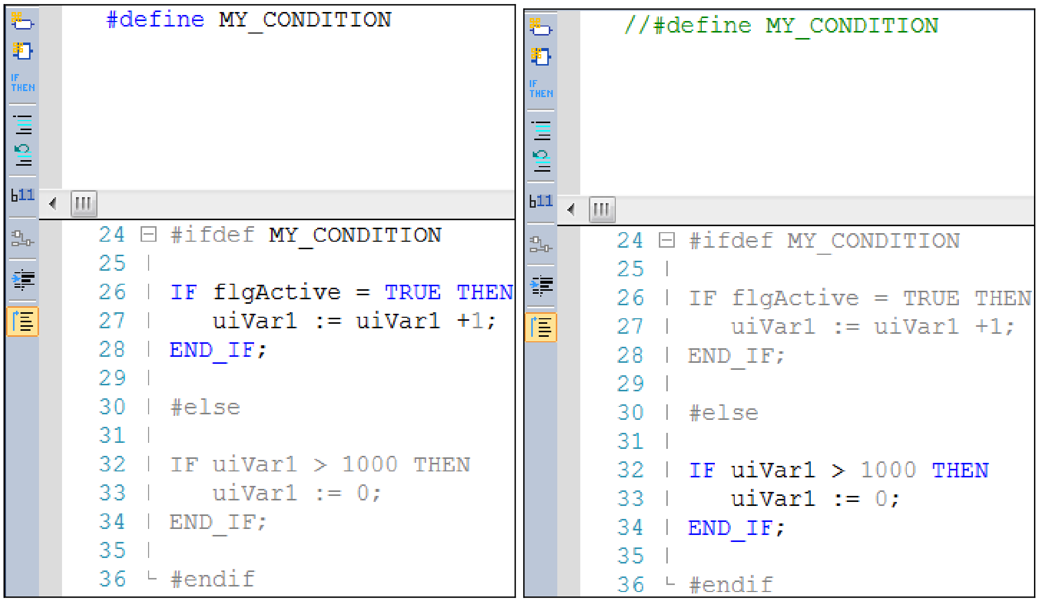
Some more features are available for smart editing and are referred to as *' IntelliSense'*. IntelliSense can be memory consuming and can be activated or deactivated from the Tools/Options menu command. After activating or de-activating the IntelliSense, you must close and reopen your project list.

Figure 76: Activate IntelliSense

The following features are available when IntelliSense is activated:

1. Conditional compiling coloring
   Parts of code which should not be compiled can be set by conditional `#ifdef` directives. Code which are shown in grey are not being compiled. Figure 77 shows

that the active part of the program changes once the conditional directive changes.



```
#define MY_CONDITION

#ifdef MY_CONDITION

IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
END_IF;

#else

IF uiVar1 > 1000 THEN
    uiVar1 := 0;
END_IF;

#endif
```

```
//#define MY_CONDITION

#ifdef MY_CONDITION

IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
END_IF;

#else

IF uiVar1 > 1000 THEN
    uiVar1 := 0;
END_IF;

#endif
```

Figure 77: Conditional directives

The `#ifdef` identifier statement is equivalent to `#ifdef 1` when identifier has been defined. It's equivalent to `#ifdef 0` when identifier has not been defined. These directives check only for the presence or absence of identifiers defined with `#define`.

The editor for defining the local preprocessor statements can be opened by right clicking the ST Editor and selecting *'Show/Hide Local Defines'* from the popup box:

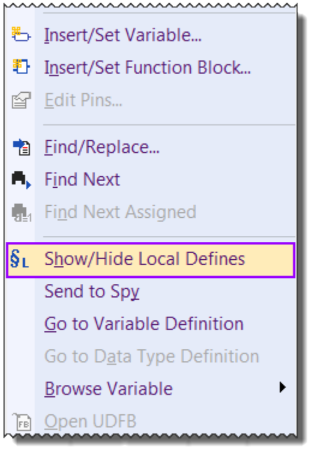
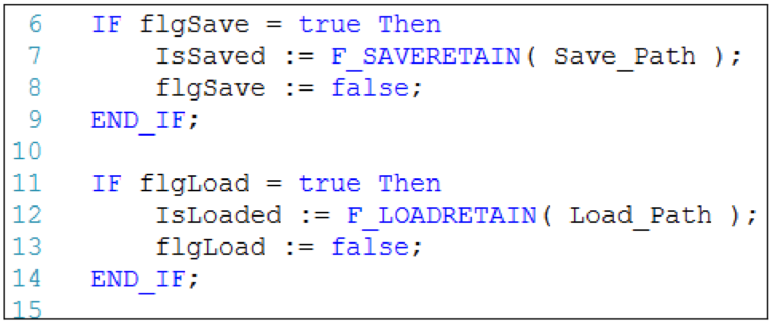


Figure 78: Showing/Hiding the editor for defining local identifiers

2. Auto-indentation

   Lines are automatically formatted (indented) on the left as you enter structured ST statements.

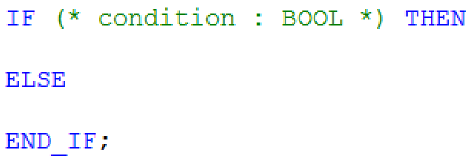


```
IF flgSave = true Then
    IsSaved := F_SAVERETAIN( Save_Path );
    flgSave := false;
END_IF;

IF flgLoad = true Then
    IsLoaded := F_LOADRETAIN( Load_Path );
    flgLoad := false;
END_IF;
```

3. Auto-completion of ST statements
   On an empty line, just enter the main keyword of a ST statement such as *'for'*, *'if'*... and immediately press the ENTER key. The whole statement will be completed including comments that will guide you through the syntax. The caret is automatically placed where you must enter the first required term or condition.

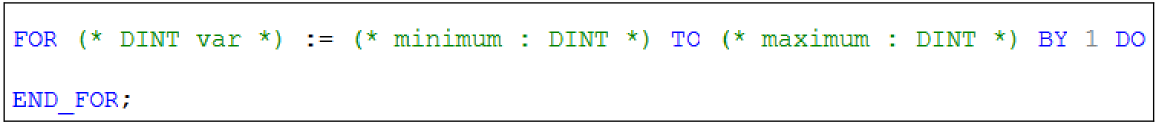
   Example:

   - Enter *'if'* and press the ENTER key:



```
IF (* condition : BOOL *) THEN

ELSE

END_IF;
```

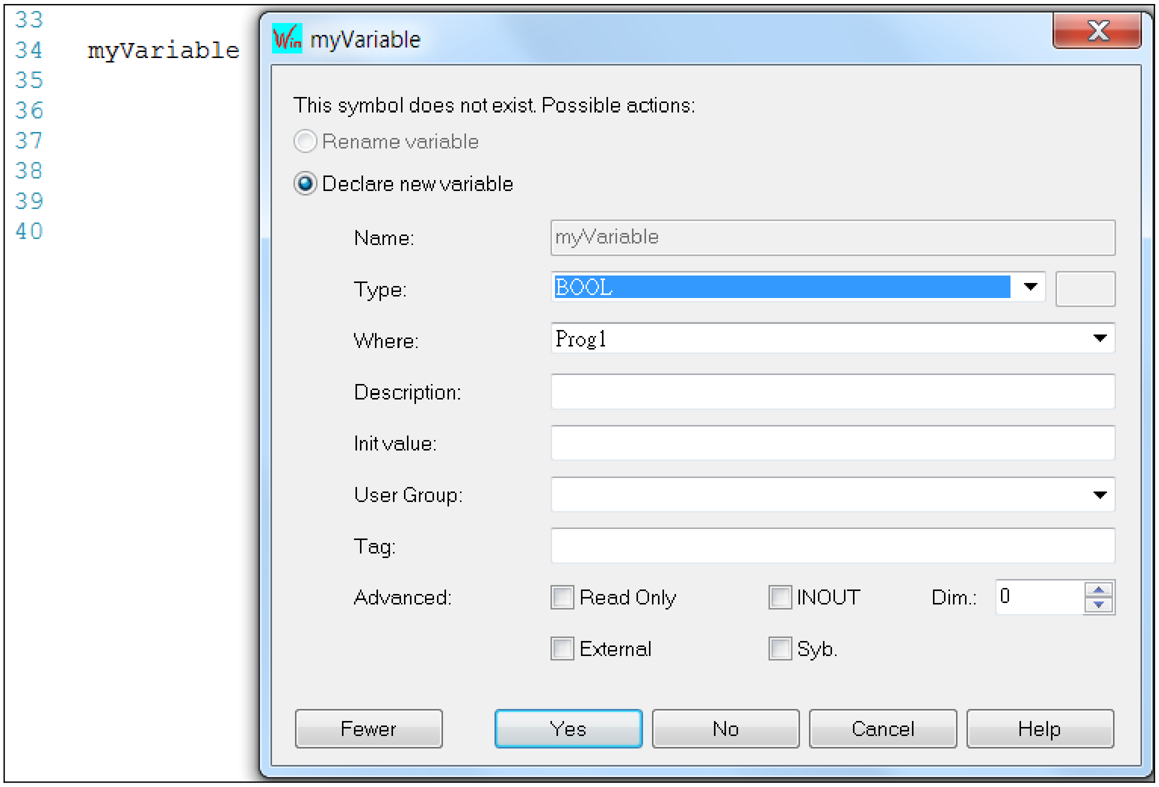
   - Enter *'for'* and press the ENTER key:



```
FOR (* DINT var *) := (* minimum : DINT *) TO (* maximum : DINT *) BY 1 DO

END_FOR;
```

4. Auto-declaration of missing symbols
   When you press ENTER at the end of a line containing an unknown variable symbol, you will be prompted for declaring it immediately.

   Example:
   Enter a variable name *'myVariable'* and press ENTER. A popup window shows up which allows you to do the necessary variable declaration:



```
myVariable
```

myVariable

This symbol does not exist. Possible actions:

○ Rename variable

● Declare new variable

Name: myVariable

Type: BOOL

Where: Prog1

Description:

Init value:

User Group:

Tag:

Advanced: □ Read Only □ INOUT Dim.: 0

□ External □ Syb.

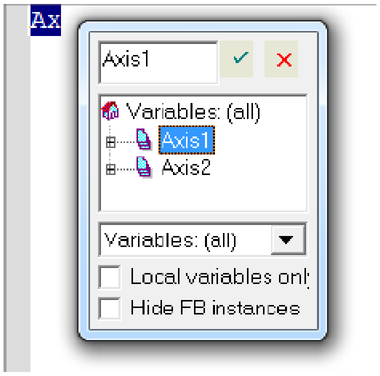
5. Line indentation
When lines are selected, you can automatically indent them. Press TAB or Shift+TAB keys to shift the lines to the left or to the right, by adding or removing blank characters on the left.

6.1.2.2 Auto Completion of Words

The ST / IL editor includes commands for automatic completion of typed words, according to declared variables and data types. The following features are available:

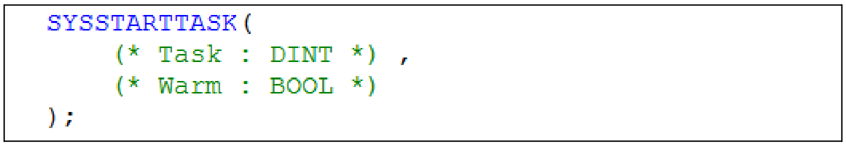
1. Auto completion of a variable name
If you enter the first letters of a variable name, you can hit the CTRL+SPACE key for automatically completing the name. A popup list is displayed with possible choices if several declared variable names match the type characters.

Example:



2. Auto-completion of function calls
   Enter the name of a function simply followed by an opening parenthesis and immediately press the ENTER key. The call will be completed with the appropriate argument list including comments and possibly default values so that you are guided through the list of values to be passed to the called function.

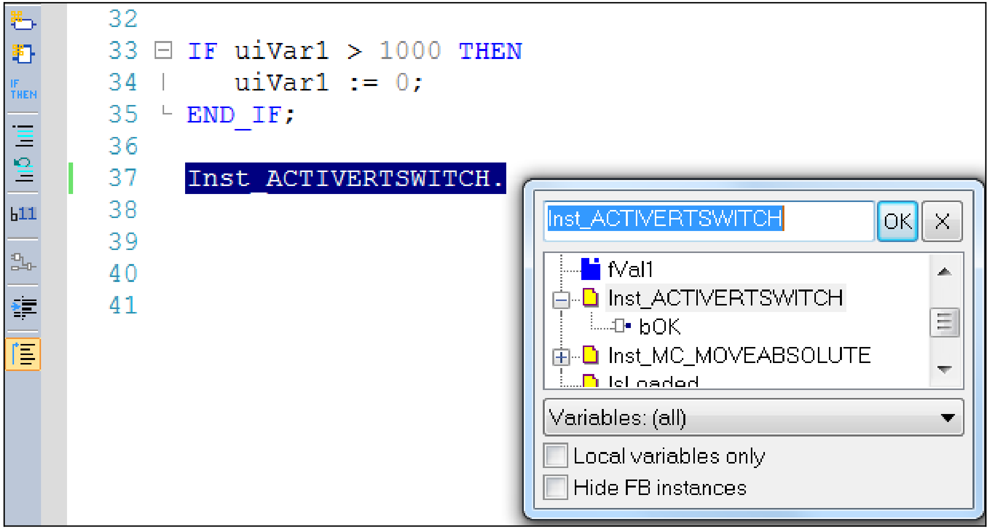
   Example:
   Enter the Function name followed by a parenthesis *'SYSSTARTTASK()'*and press enter:



```
SYSSTARTTASK(
    (* Task : DINT *) ,
    (* Warm : BOOL *)
);
```

3. Selection of FB member
   When you type the name of a function block instance (use either as an instance or a data structure), pressing the point *'.'* after the name of the instance opens a popup list with the names of possible members.

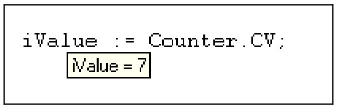
   Example:



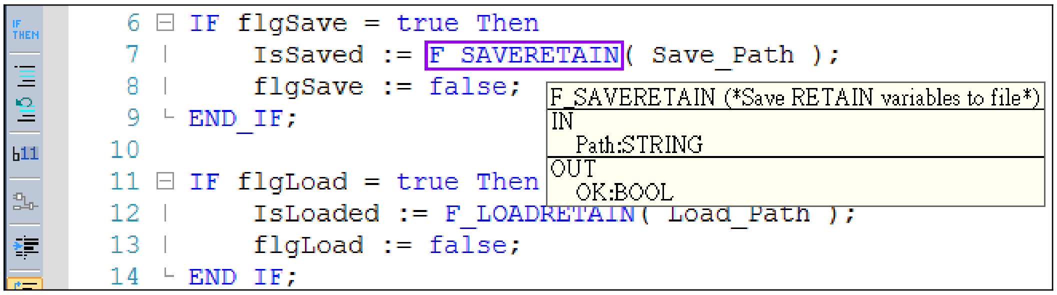
```
IF uiVar1 > 1000 THEN
    uiVar1 := 0;
END_IF;

Inst_ACTIVERTSWITCH.
```

6.1.3 Tooltips in the ST / IL Editor

During test (connected mode or simulation) of the program the ST / IL editor shows in a tooltip the current value of the variable pointed to by the mouse cursor. You do not need to run any specific command to open the tooltip. Just put the mouse on the variable symbol and wait for one second.



```
iValue := Counter.CV;
    iValue = 7
```



```
IF flgSave = true Then
    IsSaved := F_SAVERETAIN( Save_Path );
    flgSave := false;
END_IF;

IF flgLoad = true Then
    IsLoaded := F_LOADRETAIN( Load_Path );
    flgLoad := false;
END_IF;
```

```
F_SAVERETAIN (*Save RETAIN variables to file*)
IN
   Path:STRING
OUT
   OK:BOOL
```

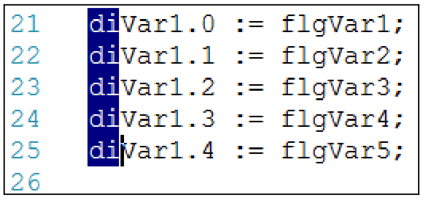
The value shown in the tooltip is automatically refreshed while the tooltip is open.

6.1.4 Shortcuts for ST and IL Editor

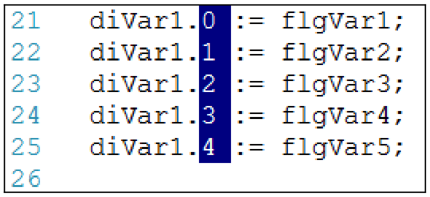
Multiple lines of the same column in the ST/IL editor can be marked and replace by text. Multiple lines at the same offset can be marked as follows:

- while pressing the Shift + Alt key the keyboard arrows can be used to select vertical text blocks
- while pressing the Alt key the mouse can be used to select vertical text blocks

So it is possible to copy existing parts of a program and modify them in a short time.



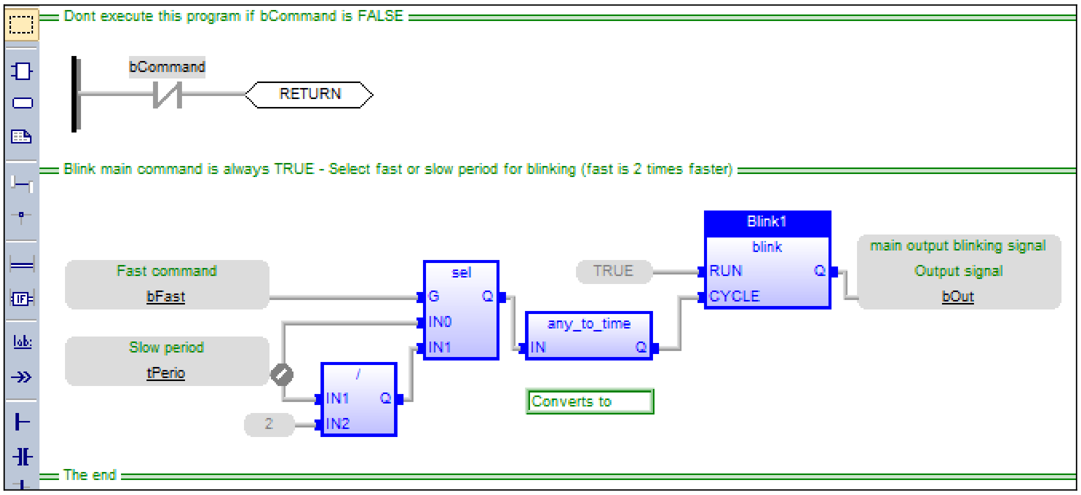
```
diVar1.0 := flgVar1;
diVar1.1 := flgVar2;
diVar1.2 := flgVar3;
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;
```



```
diVar1.0 := flgVar1;
diVar1.1 := flgVar2;
diVar1.2 := flgVar3;
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;
```

6.2 Function Block Diagram (FDB) Editor

The FBD editor is a powerful graphical tool that enables you to enter and manage Function Block Diagrams according to the IEC 61131-3 standard. The editor supports advanced graphic features such as drag and drop, object resizing and connection lines routing features, so that you can rapidly and freely arrange the elements of your diagram. It also enables you to insert in a FBD diagram graphic elements of the LD (Ladder Diagram) language such as contacts and coils.



Dont execute this program if bCommand is FALSE

Blink main command is always TRUE - Select fast or slow period for blinking (fast is 2 times faster)

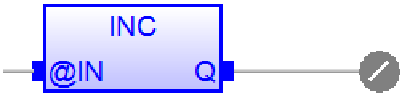
The end

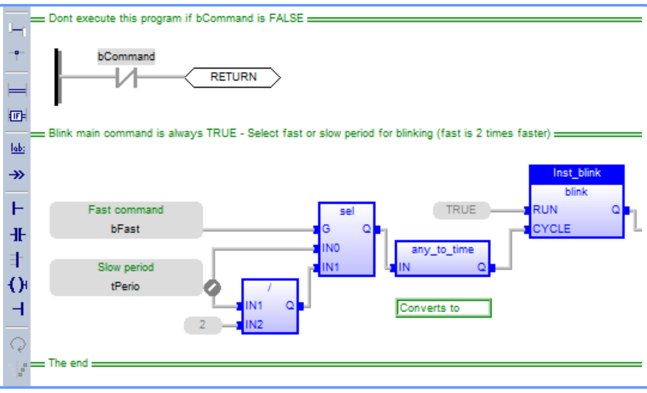
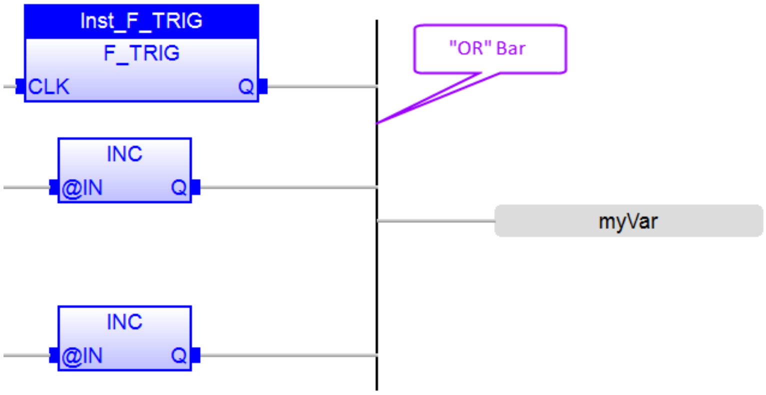
Figure 79: Function Block Editor

6.2.1 Using the FBD toolbar

The vertical toolbar on the left side of the editor contains buttons for all available editing features. Push the wished button before using the mouse in the graphic area.

Icon	Function	Description
⬚	Selection	In this mode, you cannot insert any element in the diagram. The mouse is used for selecting object and lines, select tag name areas, move or copy objects in the diagram. At any moment you can press the ESCAPE key to go back to the Selection mode.
⊡	Add Block	In this mode, the mouse is used for inserting blocks in the diagram. Click in the diagram and drag the new block to the

		wished position. The type of block that is inserted is the one currently selected in the list of the main toolbar.
	Add variable	In this mode, the mouse is used for inserting variable tags. Variable tags can then be wired to the input and output pins of the blocks. Click in the diagram and drag the new variable to the wished position.
	Add Comment	**Insert comment text:** In this mode, the mouse is used for inserting comment text areas in the diagram. Comment texts can be entered anywhere. Click in the diagram and drag the text block to the wished position. The text area can then be selected and resized.
	Add Arc	**Insert connection line:** In this mode, the mouse is used to wire input and output pins of the diagram objects. The line must always be drawn in the direction of the data flow: from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. You can change the default routing by inserting corners on lines. (see below) You also can drag a line from an output pin to an empty space. In that case the editor automatically finished the line with a user defined corner so that you can continue drawing the connection to the wished pin and force the routing while you are drawing the line. 
	Add corner	In this mode, the mouse is used for inserting a user defined corner on a line. Corners are used to force the routing of connection lines, as the FBD editor imposes a default routing only between two pins or user defined corners. Corners can then be selected and moved to change the routing of existing lines.
	Add break	Insert network break: In this mode, the mouse is used for inserting a horizontal line that acts as a break in the diagram. Breaks have no meaning for the execution of the program. They just help the understanding of big diagrams, by splitting them in a list of networks.

		 Dont execute this program if bCommand is FALSE bCommand RETURN Blink main command is always TRUE - Select fast or slow period for blinking (fast is 2 times faster) Inst_blink blink RUN CYCLE Q TRUE Fast command bFast sel G IN0 IN1 Q any_to_time IN Q Slow period tPerio / IN1 IN2 Q 2 Converts to The end
IF	Add ST condition	
lab:	Add label	In this mode, the mouse is used for inserting a label in the diagram. A label is used as a destination for jump symbols (see below).
→>	Add jump	In this mode, the mouse is used for inserting jump symbols in the diagram. A jump indicates that the execution must be directed to the corresponding label (having the same name as the jump symbol). Jumps are conditional instructions. They must be linked on their left side to a Boolean data flow.
⊢	Add left power rail	In this mode, the mouse is used for inserting a left power rail in the diagram. A left power rail is an element of the LD language, and represents a TRUE state that can be used to initiate a data flow. Power rails can then be selected and resized vertically according to the wished network height.
⊣⊢	Add direct contact	In this mode, the mouse is used for inserting in the diagram a contact as in Ladder Diagrams.
⫞	Add *'OR'* bar	In this mode, the mouse is used for inserting a rail that collects several Boolean data flows for an *'OR'* operation, in order to insert parallel contacts such as done in Ladder Diagrams.  Inst_F_TRIG F_TRIG CLK Q "OR" Bar INC @IN Q myVar INC @IN Q The OR rail has exactly the same meaning as an OR block regarding the execution of the diagram.
{}⊣	Add direct coil	In this mode, the mouse is used for inserting in the diagram a coil as in Ladder Diagrams. It is not mandatory that a coil be

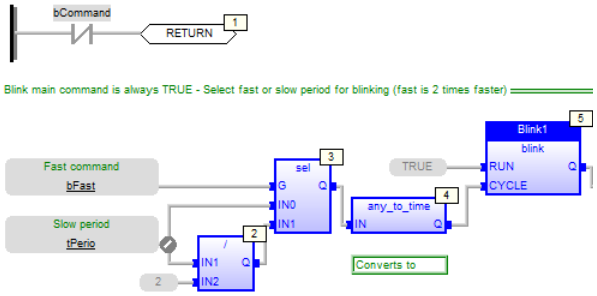
		connected on its right side.
⊣	Add right power rail	In this mode, the mouse is used for inserting a right power rail in the diagram. A right power rail is an element of the LD language, and is commonly used for terminating Boolean data flows. However it is not mandatory to connect coils to power rails. Right power rails have no meaning for the execution of the diagram.
	Show execution order	Display the execution order of the elements in the diagram. At each element a yellow box is attached which shows the execution sequence number.  bCommand RETURN 1 Blink main command is always TRUE - Select fast or slow period for blinking (fast is 2 times faster) Fast command bFast Slow period tPerio 2 / IN1 Q IN2 2 sel G Q IN0 IN1 3 TRUE any_to_time IN Q 4 Converts to Blink1 blink RUN Q CYCLE 5

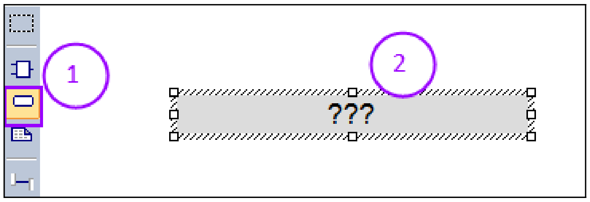
Figure 80: Function block diagram (FBD) toolbar commands

6.2.1.1 FBD Variables

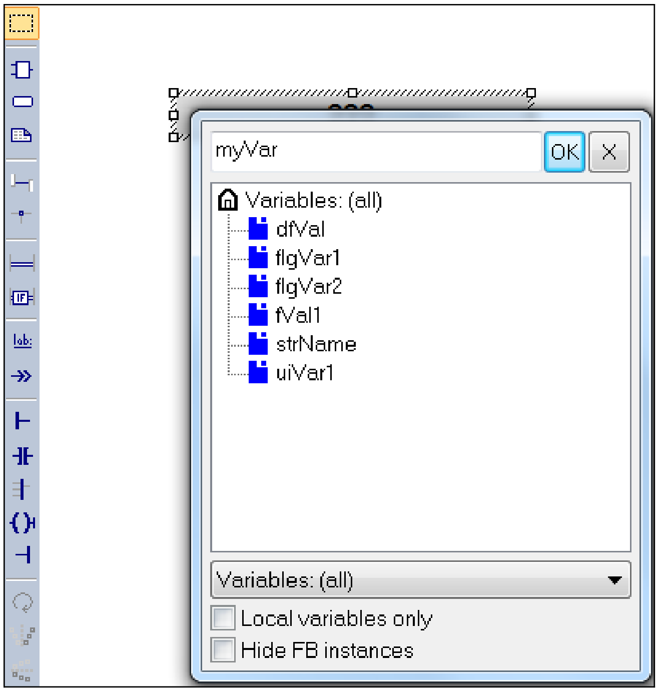
All variable symbols and constant expressions are entered in FBD diagrams using small boxes.

Step 1: Insert variable tag

1. Press the *'Insert variable'* button in the FBD toolbar
2. Click at a wished position in the FBD editor where to place the variable tag.



Step 2: Double-click on a variable tag to open the variable selection box and either select the symbol of the wished variable or enter a constant expression.



myVar

Variables: (all)

- dfVal
- flgVar1
- flgVar2
- fVal1
- strName
- uiVar1

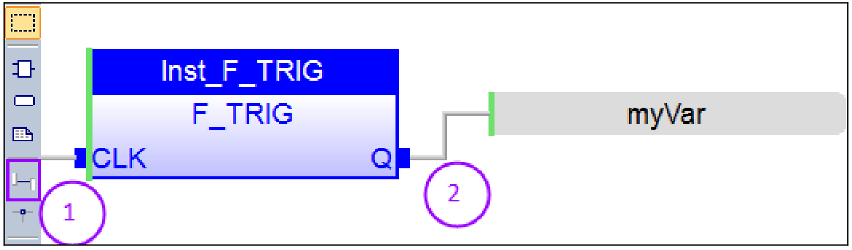
Variables: (all)

□ Local variables only

□ Hide FB instances

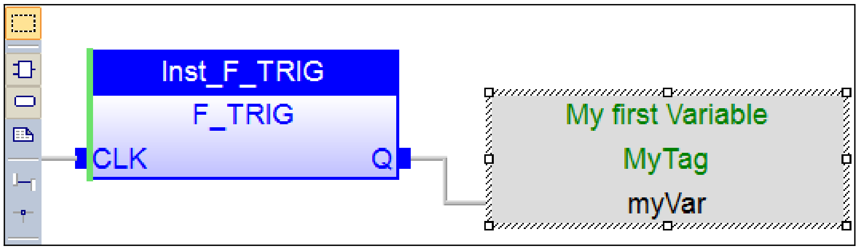
Step 3: Variables tags must then be linked to other objects such as block inputs and outputs using connection lines.

1. Click *'Add Arc'* button to insert connection line.
2. Connect the variable tag either to the input or output of a block.



Step 4: You can resize a variable box vertically in order to display together with the variable name its tag (short comment text), its description text, plus its I/O location if the variable is mapped to an I/O channel. The variable name is always displayed at the bottom of the rectangle:

- tag
- description
- % location
- name

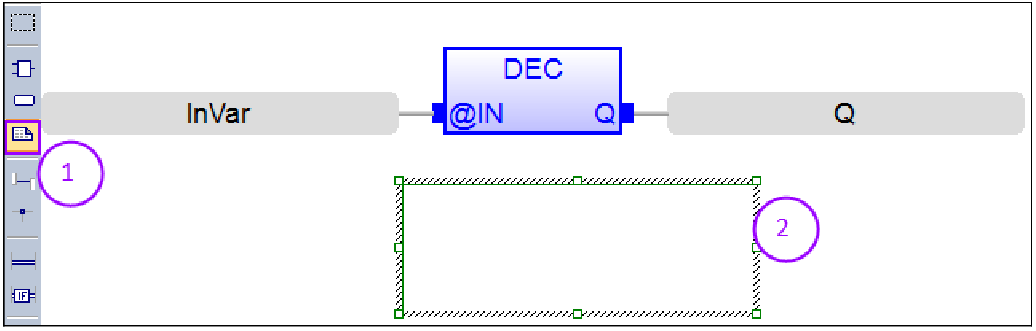


6.2.1.2 FBD Comments

Comment text areas can be entered anywhere in a FDB diagram.

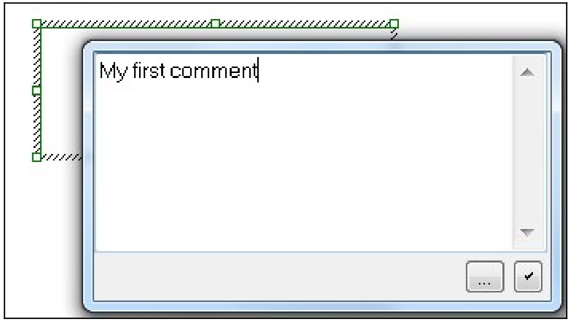
Step 1: Add a comment box to the FBD editor:

1. Press the *'Add comment'* button in the FBD toolbar for inserting a new comment area
2. Drag the comment box to the required position

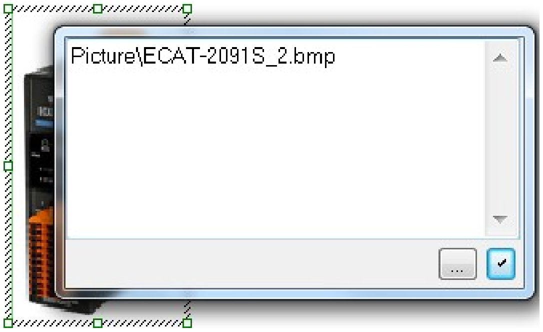


Step 2: Edit a text into the comment box:

1. Double-click on the comment area for entering or changing the attached text. You can also insert a bitmap to the comment box by entering the directory of the bitmap

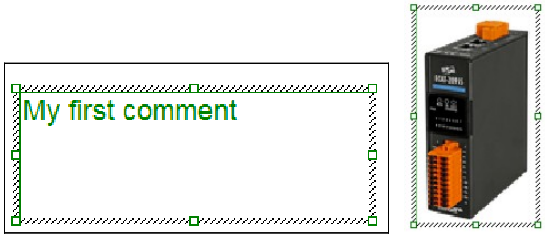


My first comment



Picture\ECAT-2091S_2.bmp

2. Confirm the setting. When selected, comment texts can be resized.



6.2.1.3 FBD Corners

Corners are used to force the routing of connection lines, as the FBD editor imposes a default routing only between two pins or user defined corners.

Example:
The FBD editor connects the FBD elements by choosing the shortest distance. The corners of the connection line are chosen by the FBD editor in such a way that the lines do not cross another function block or variable element.

Figure 81 shows the connection line generated by the FBD editor

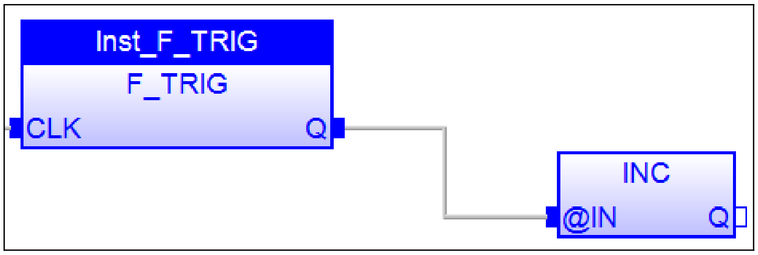


Figure 81: Connection line path generated by the FBD editor

Corners can then be selected and moved to change the routing of existing lines (Figure 82). In order to change the pathway of the line insert a corner on a line by pressing the *'Add corner'* button in the FBD toolbar and clicking the connection line. The newly added corner can now be move by the mouse to a new position and the connection line will automatically follow the new corner position.

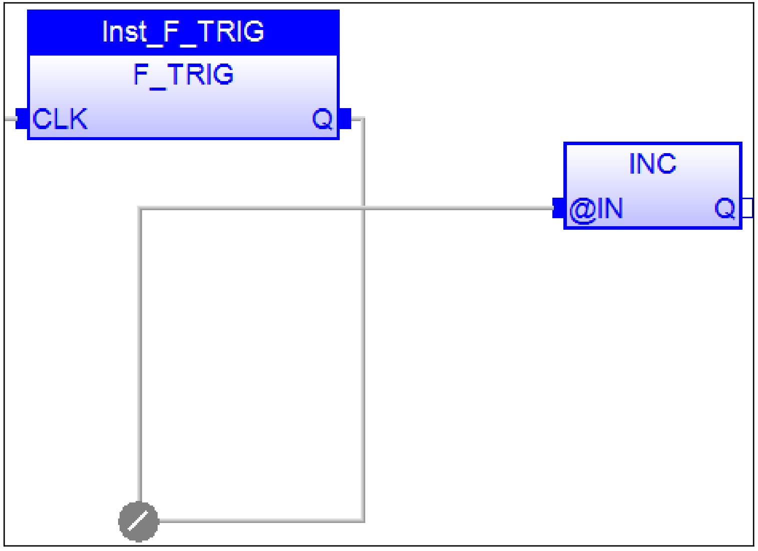
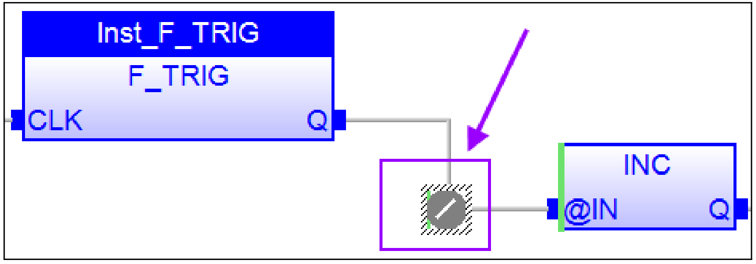
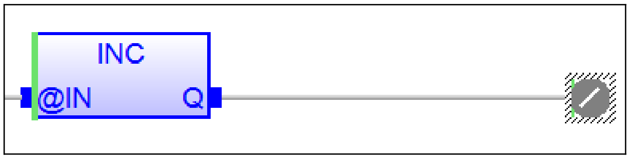


Figure 82: Connection line path manually edited by using corners

Before moving the corner make sure the corner is surrounded by a square otherwise a new corner will be generated

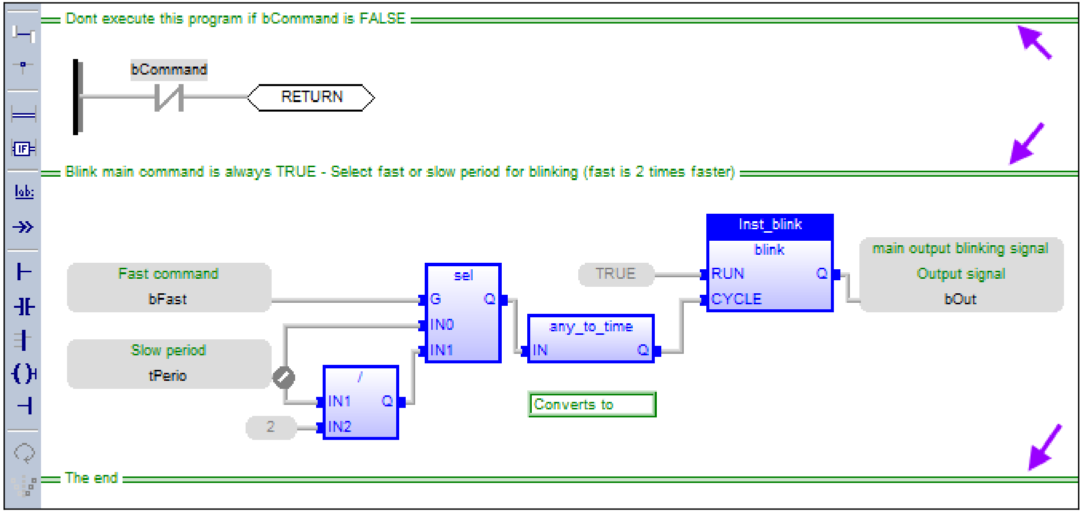


You can drag a new line from an output pin to an empty space. In that case the editor automatically finished the line with a user defined corner so that you can continue drawing the connection to the wished pin and force the routing while you are drawing the line.



6.2.1.4 FBD Network Breaks

Network breaks can be entered anywhere in a FBD diagram. Breaks have no meaning for the execution of the program. They just help the understanding of big diagrams, by splitting them in a list of networks.


Dont execute this program if bCommand is FALSE
bCommand
RETURN
Blink main command is always TRUE - Select fast or slow period for blinking (fast is 2 times faster)
Fast command
bFast
Slow period
tPerio
2
/
IN1
IN2
Q
sel
G
IN0
IN1
Q
any_to_time
IN
Q
Converts to
TRUE
Inst_blink
blink
RUN
CYCLE
Q
main output blinking signal
Output signal
bOut
The end

Press the following button in the FBD toolbar for inserting a new break:

The break line is drawn on the whole diagram width. No other object can overlap a network break. Break lines can then be selected and moved vertically to another location.

Network breaks can also be used for browsing the diagram. Press Ctrl+Page Up or Ctrl+Page Down keys to move the selection to the next or previous network break.

6.2.1.5 FBD 'OR' Vertical Rail

The FBD editor enables the drawing of LD rungs. A particular object, the *'OR'* rail can be inserted on a rung in order to connect parallel contacts together (Figure 83).

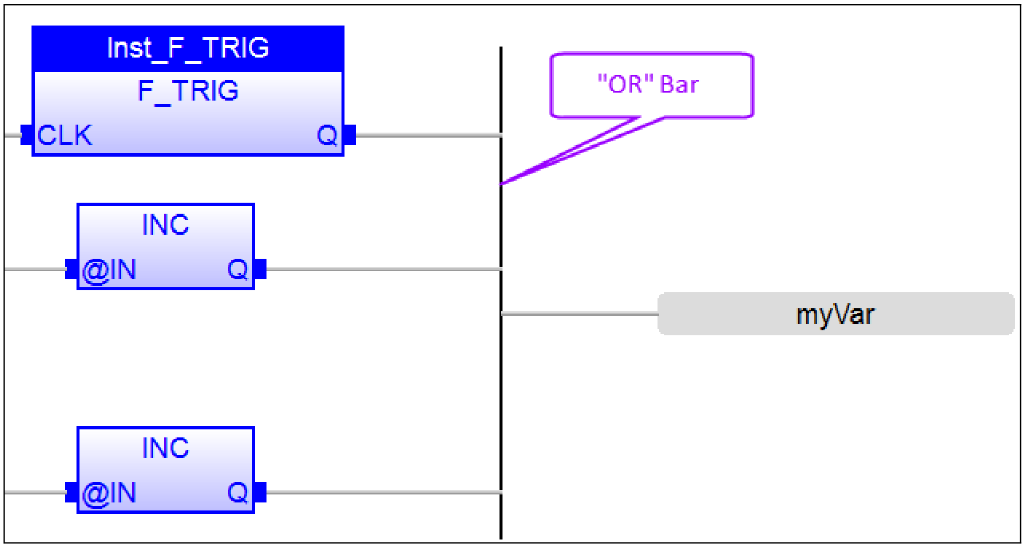


Figure 83: Example of an 'OR' bar application

The OR rail has exactly the same meaning as an OR block regarding the execution of the diagram (Figure 84).

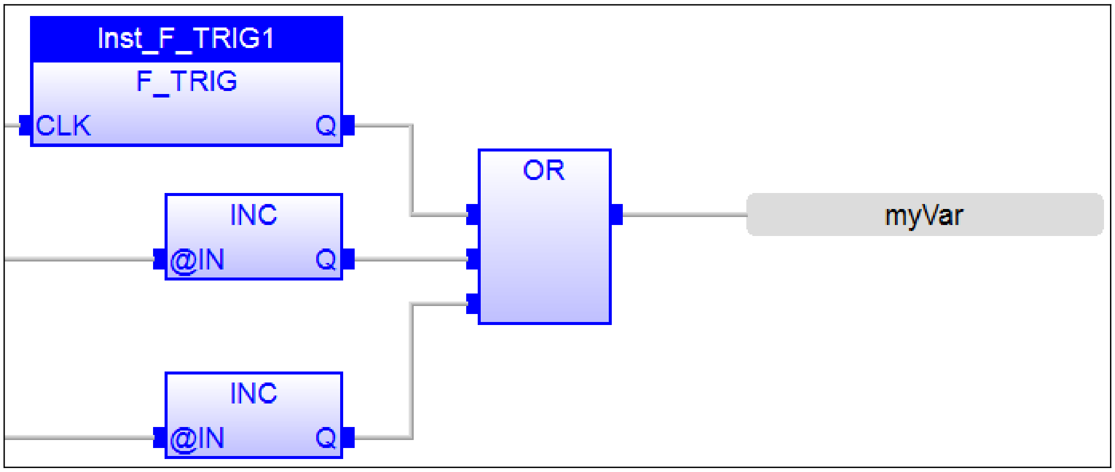


Figure 84: Example of an 'OR' function

6.2.2 Drawing FBD connection lines

The connection lines are being used to connect input and output of the objects in the FBD editor. The line must always be drawn in the direction of the data flow: from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. You can change the default routing by inserting corners on lines. Press the button before inserting a new line.

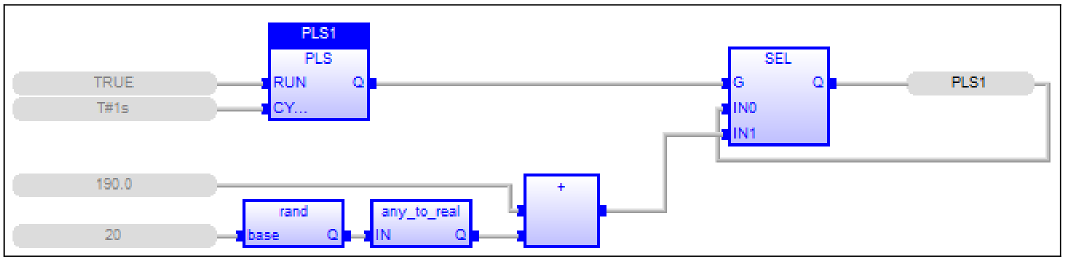


Figure 85: Connection lines indicate the data flow

Connection line is colored in red if the two linked elements are not the same data type.

The editor enables you to terminate a connection line with a Boolean negation represented by a small circle. To set or remove the Boolean negation, select the line and press the SPACE bar.

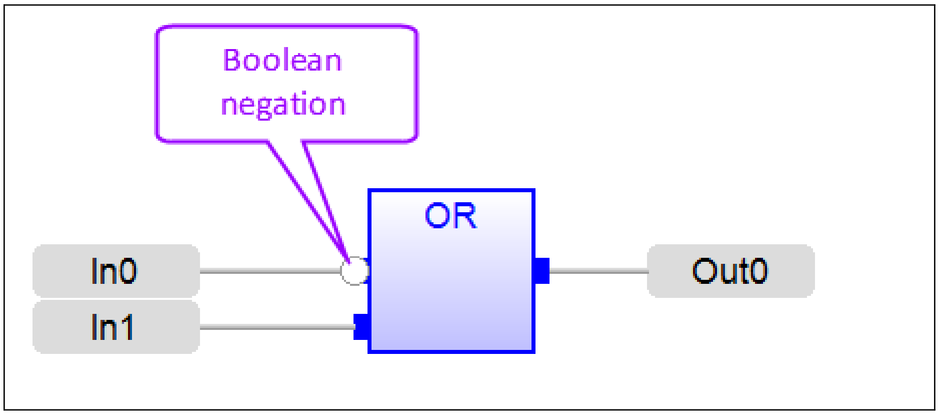


Figure 86: Connection line with a Boolean negation termination

Connection lines must always be drawn in the direction of the data flow: from an output pin to an input pin. The FBD editor automatically selects the best routing for the new line. Connection lines indicate a data flow between the following possible objects:

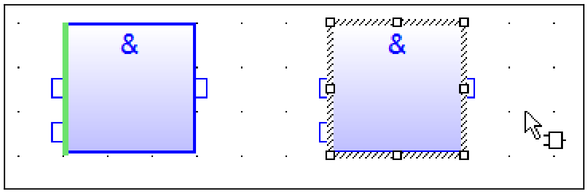
Procedure for drawing a connection line:

Step 1: Add two blocks to the FBD editor:

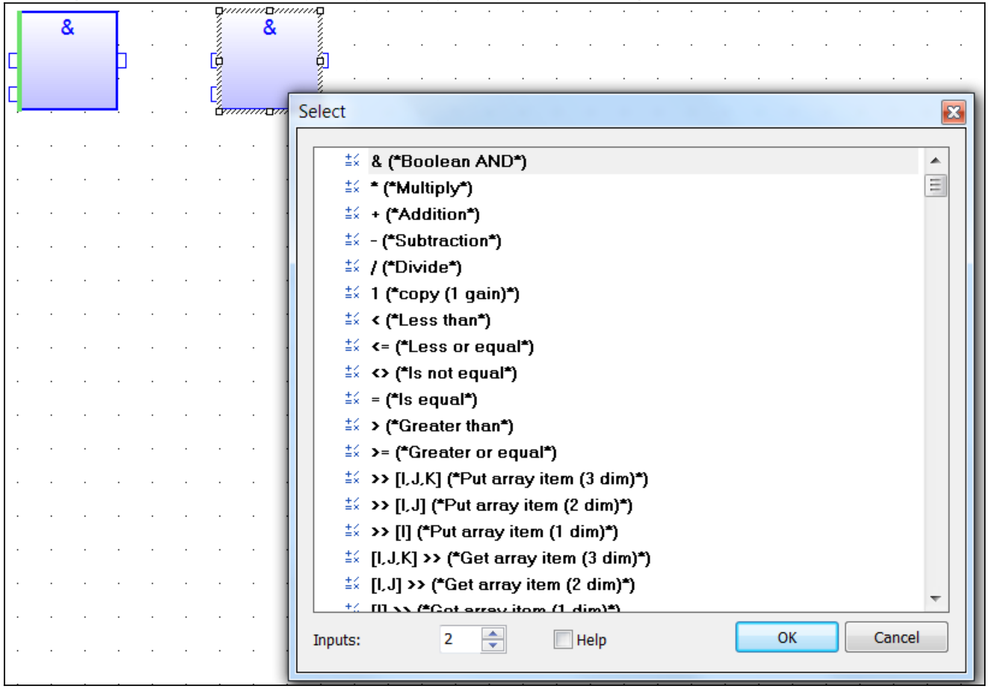
1. Click the *'Add function block'* command. Now a function block icon is displayed next to the mouse pointer

   This indicates that every mouse click drag on the FBD editor will generate a new block at the current pointer position.
2. Insert two blocks: A block is being added left clicking the mouse and

dragging it. By default the function block which appears in the first row of the selection list will be inserted.



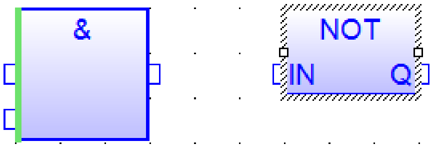
3. Change the function block type by double clicking it



& (*Boolean AND*)
* (*Multiply*)
+ (*Addition*)
- (*Subtraction*)
/ (*Divide*)
1 (*copy (1 gain)*)
< (*Less than*)
<= (*Less or equal*)
<> (*Is not equal*)
= (*Is equal*)
> (*Greater than*)
>= (*Greater or equal*)
>> [I,J,K] (*Put array item (3 dim)*)
>> [I,J] (*Put array item (2 dim)*)
>> [I] (*Put array item (1 dim)*)
[I,J,K] >> (*Get array item (3 dim)*)
[I,J] >> (*Get array item (2 dim)*)
[I] >> (*Get array item (1 dim)*)

Inputs: 2

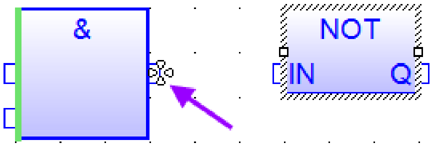
In this example the NOT function will be selected for the left block:



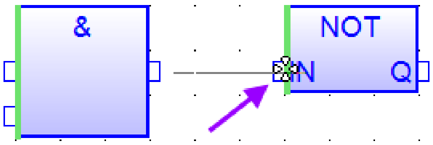
Step 2: Connect the output pin of the & function to the input pin of the NOT function

1. Click the *'Add arc'* button on the left toolbar to enter the *'insert connection line'* mode. A line icon appears next to the mouse pointer

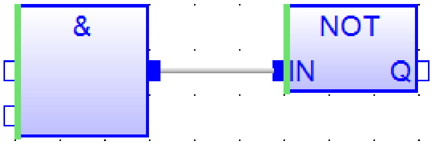
2. Move the mouse pointer over the output pin of the & block.



When a star appears the click the left mouse button and drag it to the input pin of the NOT block.



Once the star appears at the input pin release the mouse button. A connection line is drawn between the output and input.



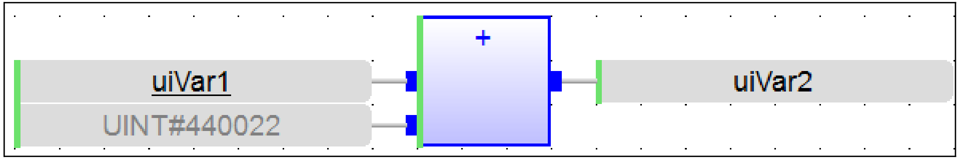
6.2.3 Selecting FBD Variables and Instances

To attached a variable to a graphic object (e.g. input and output pin of a function block), you must be in Selection mode. Simply double-click on the gray area of the variable tag box.

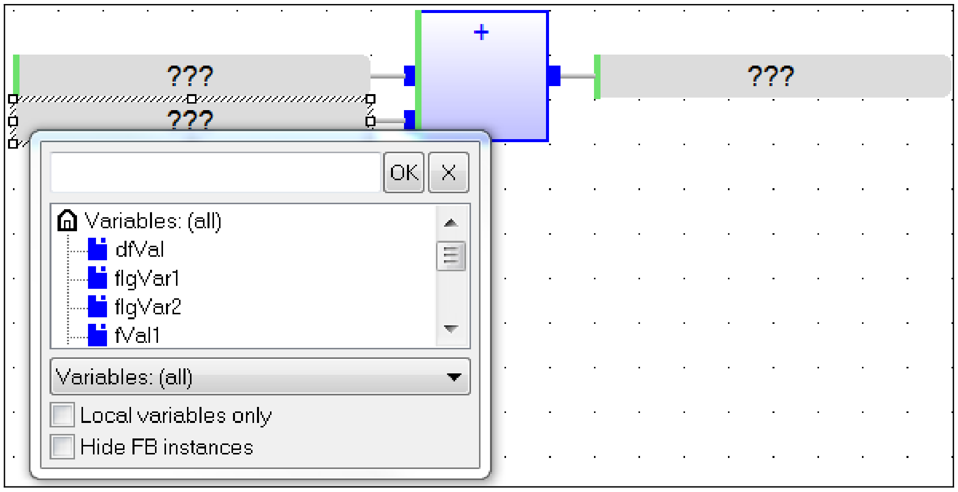
Procedure for assigning a variable to a object in the FBD editor:

Step 1: Go into selection mode by clicking the *'Selection'* button ⬚ or press the *'Esc'* key. A rectangular will be shown next to the mouse pointer.

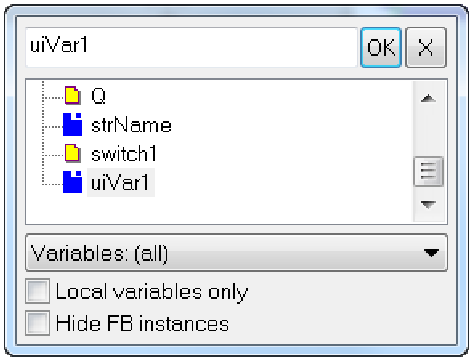
Step 2: Assign a variables:



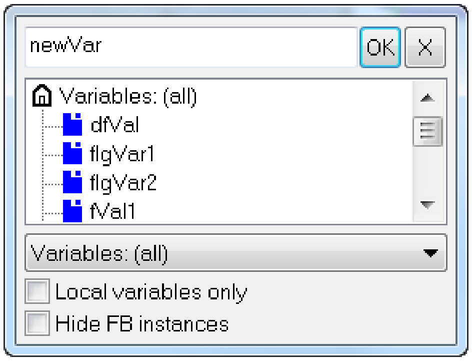
1. Double click on the tag name in the gray area. A dialog with variable list pops up.



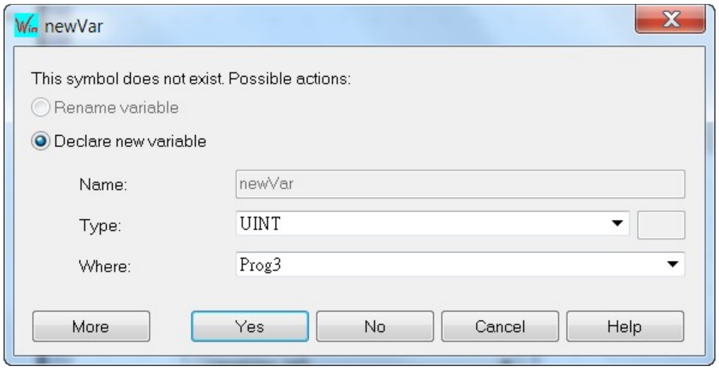
2. You have three possibilities to assign a variable:
   - Select a variable from the variable list



   - Declare a new variable in the variable list.
     The variable list can also be used as an variable editor by entering a new variable name in the list header and click *'OK'*.



A dialog box pops up which allows you to directly declare the variable.



newVar

This symbol does not exist. Possible actions:

○ Rename variable

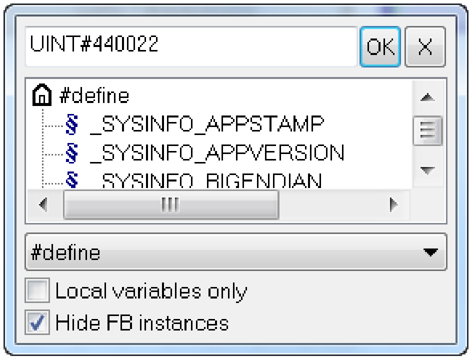
● Declare new variable

Name: newVar

Type: UINT

Where: Prog3

- Assign a constant expression (e.g. `UINT#440022`, `BOOL#FALSE`).



UINT#440022

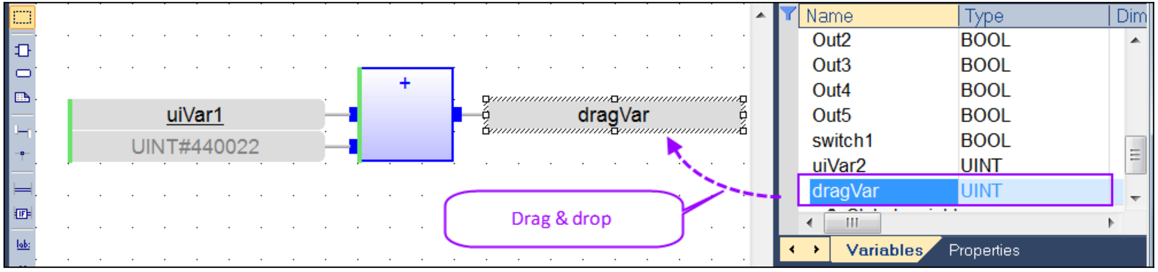
#define
§ _SYSINFO_APPSTAMP
§ _SYSINFO_APPVERSION
§ _SYSINFO_BIGENDIAN

#define

□ Local variables only

☑ Hide FB instances

Step 3: Option: It is also possible to drag a variable from the variable editor directly to the grey variable tag.



Name	Type	Dim
Out2	BOOL	
Out3	BOOL	
Out4	BOOL	
Out5	BOOL	
switch1	BOOL	
uiVar2	UINT	
dragVar	UINT	

6.2.4 Viewing FBD Diagrams

The diagram is entered in a logical grid. All objects are snapped to the grid. You can use the commands of the View menu for displaying of hiding the points of the grid.

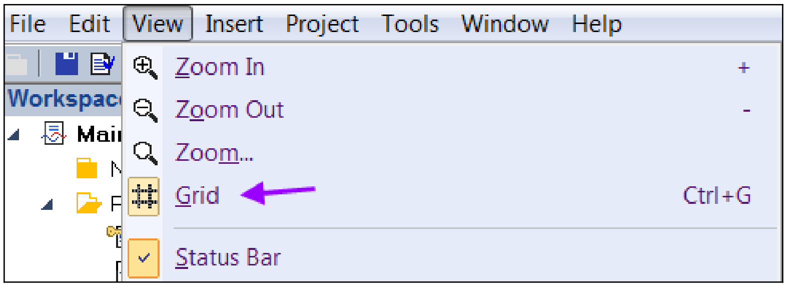


Figure 87: Viewing commands

The (x,y) coordinates of the mouse cursor are displayed in the status bar. This helps you locating errors detected by the compiler, or aligning objects in the diagram.

At any moment you can use the commands of the View menu for zooming in or out the edited diagram. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

6.2.5 Moving or Copying FBD Objects

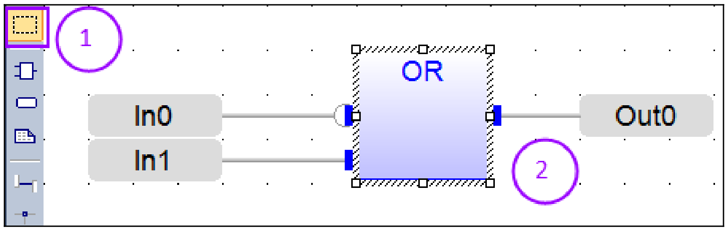
The FBD editor fully supports drag and drop for moving or copying objects.

6.2.5.1 Moving FBD Objects

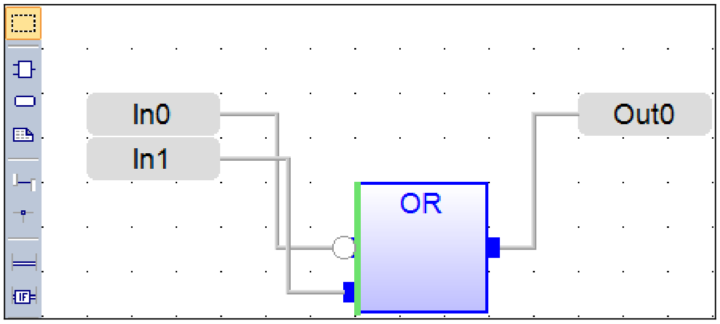
To move objects, select them and simply drag them to the wished position.

Step 1: Select an object

1. Go into selection mode by clicking the *'Selection'* button or press the *'Esc'* key.
2. Select the object by clicking on it with the mouse. The selected object will be shown in a selection frame.



Step 2: Drag the object to a new position.



6.2.5.1.1 Using the Keyboard

When graphic objects are selected, you can move them in the diagram by hitting the following keys:

Shortcut	Description
Shift + Up	Move to the top.
Shift + Down	Move to the bottom.
Shift + Left	Move to left.
Shift + Right	Move to right.

Table 11: Keyboard shortcuts for moving objects

When an object is selected, you can extend the selection by hitting the following keys:

Shortcut	Description
Shift + Control + Home	Extend to the top: select all objects before the selected one.
Shift + Control + End	Extend to the bottom: select all objects after the selected one.

Table 12: Keyboard shortcuts for selection extension

To insert or delete space in the diagram, you can simply select an object, press Shift+Control+End to extend the selection and then move selected objects up or down.

6.2.5.1.2 Auto alignment

When multiple objects are selected, the following keystrokes automatically align them:

Shortcut	Description

Control + Up	Align to the top.
Control + Down	Align to the bottom.
Control + Left	Align to left.
Control + Right	Align to right.

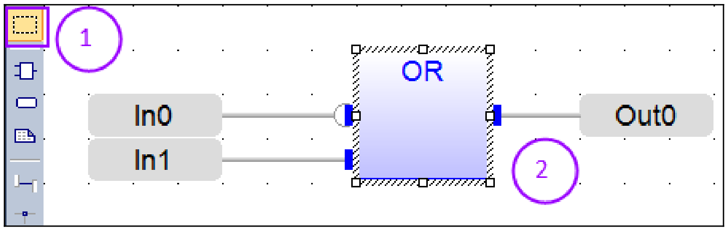
Table 13: Keyboard shortcuts for auto alignment

6.2.5.2 Copying FBD Objects

To copy objects, select the object, and just press the CONTROL key while dragging.

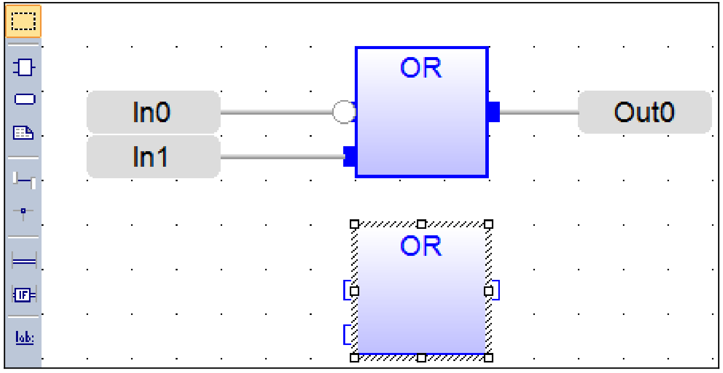
Step 1: Select an object

1. Go into selection mode by clicking the *'Selection'* button or press the *'Esc'* key.
2. Select the object by clicking on it with the mouse. The selected object will be shown in a selection frame.



Step 2: Copy the object:

1. Press the *'Ctrl'* key and drag the object to a new position.



It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen.

At any moment while dragging objects you can press ESCAPE to cancel the operation.

Alternatively, you can use classical Copy / Cut / Paste commands from the Edit menu. When you run the Paste command, the editors turns in Paste mode, with a special mouse cursor (Figure 88). Click in the diagram and move the mouse cursor to the wished position for inserting pasted objects.

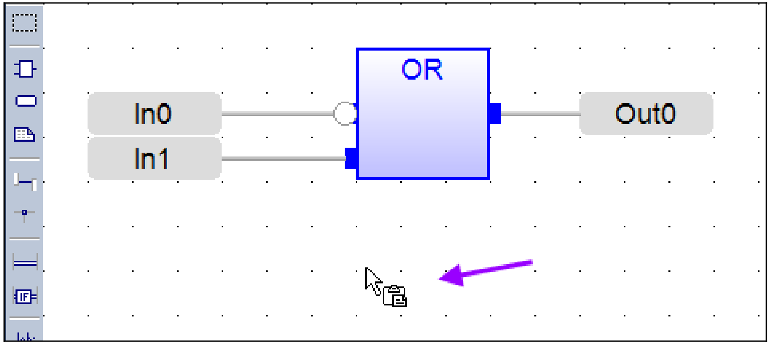


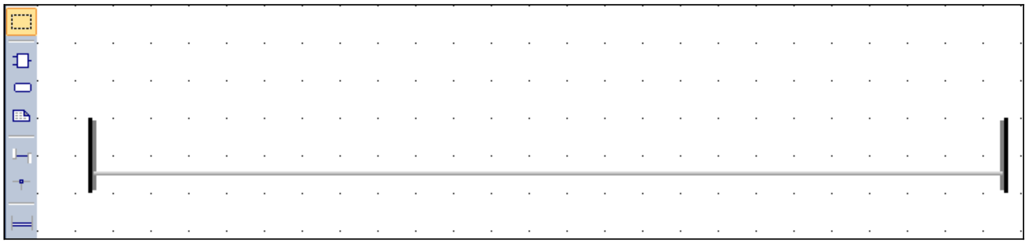
Figure 88: Mouse cursor when in paste mode

6.2.6 Inserting FBD Objects on a Line

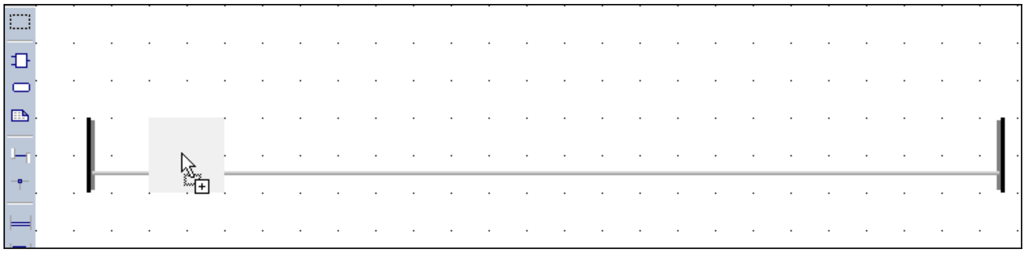
The FBD editor enables you to insert an object on an existing line and automatically connect it to the line. This feature is available for all objects having **one input pin** and **one output pin**, such as variable boxes, contacts and coils. This feature is mainly useful when entering pieces of Ladder Diagrams. Just draw a horizontal line between left and right power rails: this is the rung. Then you can simply insert contacts and coils on the line to build the LD rung.

Example:

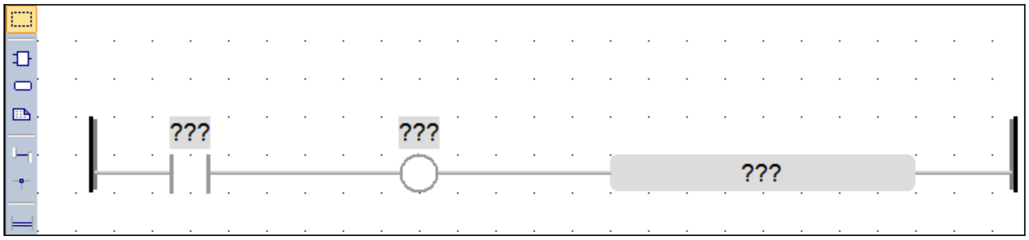
1. Add a left ⊢ and right ⊣ power rail to the editor and connect both ends with a connection line.



2. Insert a contact ⊣⊢, coil ( ) and variable ▭ on the line by dragging the objects directly over the connection line:

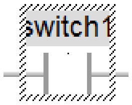
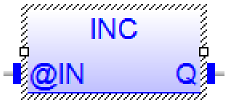
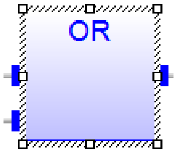


The FBD editor will automatically connect the new object to the existing connection line:



6.2.7 Resizing FBD objects

Most of the objects provide inside the FBD editor can be resized. Objects which support resizing show small square boxes after they have been selected. The square boxes indicates in which direction the objects can be resized. Click on the small square boxes for resizing the object in the wished direction.

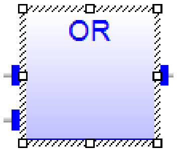
Shortcut	Description
	Object can not be resized
	Object can only be resized in horizontal direction
	Object can only be resized in all directions

Resizing procedure:

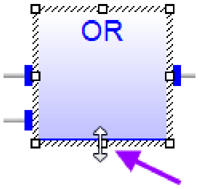
Step 1: Select an object

1. Go into selection mode by clicking the *'Selection'* button or press the *'Esc'* key.

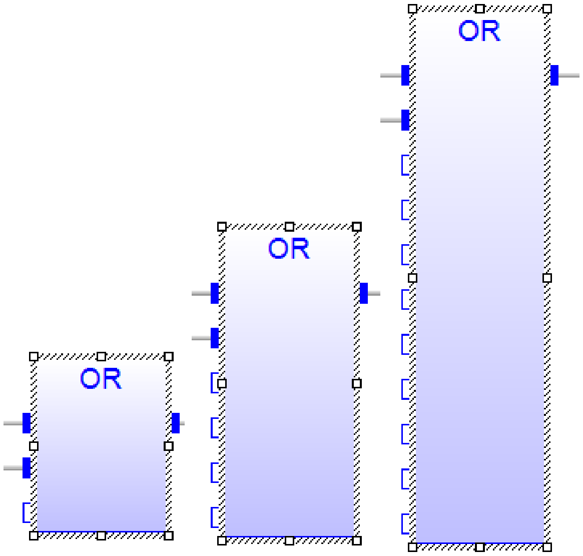
2. Select the object by clicking on it with the mouse. The selected object will be surrounded by a rectangular frame with small squares.



Step 2: Move the mouse pointer over one of the tiny squared boxes, click on it and drag the boundary to the required size.



Step 3: Optional: Some function (e.g. OR, AND, etc.) allows the user to increase the number of input pins. The number of pins can be increased/decreased by resizing the block in vertical direction:



6.3 Ladder Diagram (LD) Editor

The LD editor is a powerful graphical tool that enables you to enter and manages Ladder Diagrams according to the IEC 61131-3 standard. The editor enables quick input using the keyboards, and supports advanced graphic features such as drag and drop.

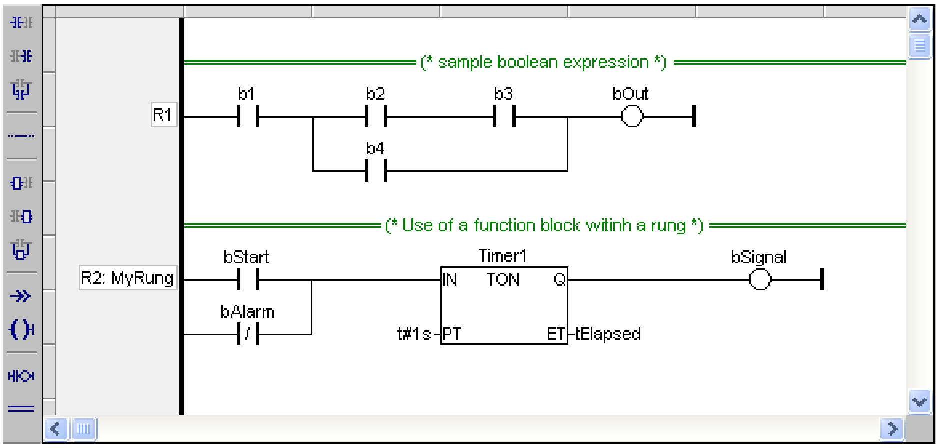

(* sample boolean expression *)
R1
b1
b2
b3
bOut
b4
(* Use of a function block witinh a rung *)
R2: MyRung
bStart
Timer1
TON
IN
Q
bSignal
bAlarm
t#1s
PT
ET
tElapsed

Figure 89: Ladder diagram (LD) editor

A Ladder Diagram is a list of rungs. Each rung represents a Boolean data flow from a power rail on the left to a power rail on the right. The left power rail represents the TRUE state. The data flow must be understood from the left to the right. Each symbol connected to the rung either changes the rung state or performs an operation. Below are possible graphic items to be entered in LD diagrams:

- Power Rails
- Contacts and Coils
- Operations, Functions and Function blocks, represented by rectangular blocks
- Labels and Jumps
- Use of ST instructions in graphic languages

6.3.1 Using the LD Toolbar

The vertical toolbar on the left side of the LD editor contains buttons for inserting items in the diagrams. Items are inserted at the current position in the diagram.

Icon	Shortcut	Description
⊣⊢	Shift+F4	Insert a contact before the selected item.
⊣⊢	F4	Insert a contact after the selected item.
⊣⊢	Ctrl+F4	Insert a contact in parallel with the selected items

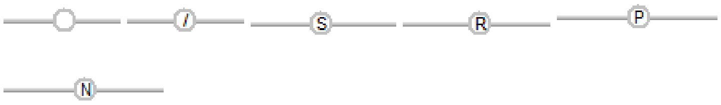
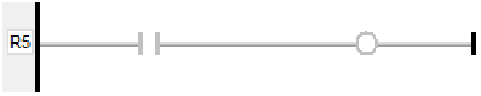
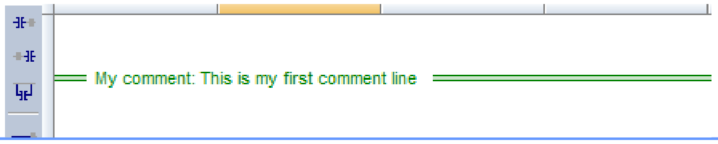
	Ctrl+Space	Insert a horizontal line before the selected item so that it is pushed to the right.
	Spacebar	Swap item style of the current cell for a contact coil 
	Shift+F8	Insert a block before the selected item.
	F8	Insert a block after the selected item.
	Ctrl+F8	Insert a block in parallel to the selected items.
	Shift+F9	Add a jump in parallel to the selected coil.
	F9	Add a coil in parallel to the selected coil or contact.
	Ctrl+R	Inserts a new rung in the diagram before the current one.  Hint: If you mark a rung and press CTRL while clicking on the vertical toolbar entry, the rung will be inserted after the marked rung.
	Ctrl+D	Insert a comment between rungs.  My comment: This is my first comment line
		Align the coils

Figure 90: Ladder diagram (LD) toolbar

6.3.2 Managing Rungs

A LD diagram is a sequential list of rungs. Each rung represents left to right Boolean power flow, that begins with a power rail, always drawn in the first column of the diagram, and finishes with a coil or a jump symbol.

Each Rung is identified by a default numbered identifier (Rnnn) displayed on the left of the power rail. The rung identifier can be used as a target for jump instructions. Alternatively you can enter a specific rung label by double-click in the rung head on the left margin.

The LD editor enables you to manipulate whole rungs by selecting only their head in the left margin. The following example shows a selected rung:

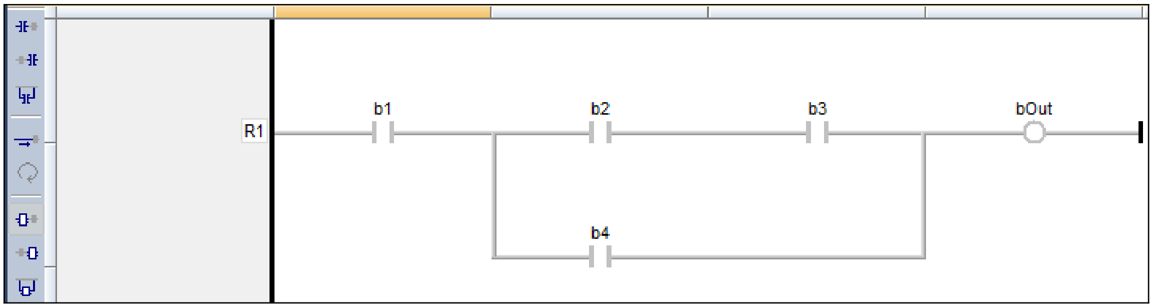


Figure 91: Rung in a ladder diagram

When a rung is selected, use the commands of the Edit menu to delete, copy or cut it.

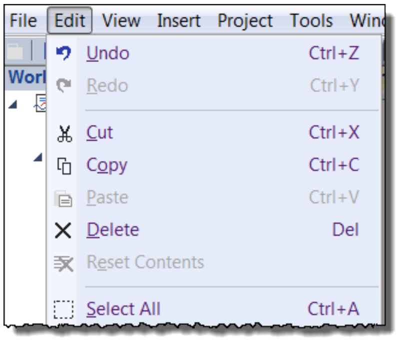
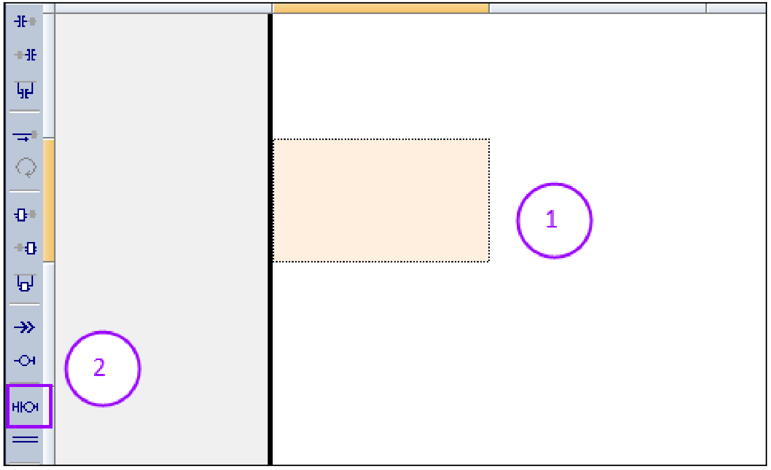


Figure 92: Copy, cut and delete commands for the ladder diagram

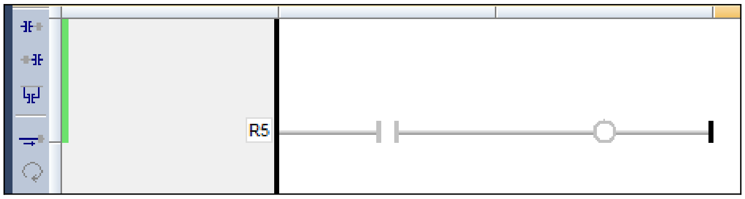
Steps for adding a new rung:

Step 1: Select a line at the which the rung has to be inserted.

Step 2: Click on the *'Insert new rung'* button on the left toolbar



A new rung is added one line above the selected line:



6.3.3 Contacts

Contacts are basic graphic elements of the LD language. A contact is associated to a Boolean variable written upon its graphic symbol. A contact sets the state of the rung on its right side, according to the value of the associated variable and the rung state on its left side.

Below are the possible contact symbols and how they change the rung state:

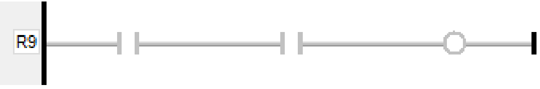
Symbol	Description
—| |—	Normal: the rung state on the right is the Boolean AND between the rung state on the left and the associated variable.
—|/|—	Negated: the rung state on the right is the Boolean AND between the rung state on the left and the negation of the associated variable.
—|P|—	Positive pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from FALSE to TRUE (rising edge).
—|N|—	Negative pulse: the rung state on the right is TRUE only when the rung state on the left is TRUE and the associated variable changes from TRUE to FALSE (falling edge).

Table 14: Contact symbols

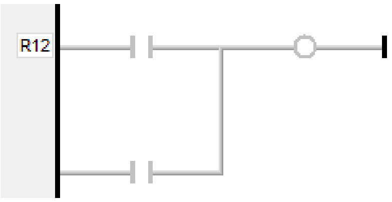
Info

When a contact or a coil is selected, You can press the SPACE bar to change its type (normal, negated, pulse...).

Two serial normal contacts represent an AND operation.



Two contacts in parallel represent an OR operation.



6.3.4 Coils

Coils are basic graphic elements of the LD language. A coil is associated to a Boolean variable written upon its graphic symbol. A coil performs a change of the associated variable according to the rung state on its left side.

Below are the possible coil symbols and how they change the rung state:

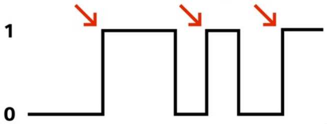
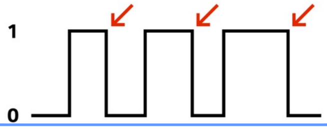
Symbol	Description
—( )—	Normal: the associated variable is forced to the value of the rung state on the left of the coil.
—(/)—	Negated: the associated variable is forced to the negation of the rung state on the left of the coil.
—(P)—	Positive Transition-Sensing Coil: A Positive Transition Contact gives a single one-shot pulse when the bit operand it is linked to **rises** from FALSE (logic 0) to TRUE (logic 1). 
—(N)—	Negative Transition-Sensing Coil: A Negative Transition Contact gives a single one-shot pulse when the bit operand it is linked to **falls** from TRUE (logic 1) to FALSE (logic 0). 
—(S)—	Set: the associated variable is forced to TRUE if the rung state on the left is TRUE. (no action if the rung state is FALSE)
—(R)—	Reset: the associated variable is forced to FALSE if the rung state on the left is TRUE. (no action if the rung state is FALSE)

Table 15: Coils symbols

Info

When a contact or a coil is selected. You can press the SPACE bar to change its type (normal, negated, pulse...).

Attention

Even though coils are commonly connected to a power rail on the right, the rung may be continued after a coil. The rung state is never changed by a coil symbol.

6.3.5 Power Rails

Vertical power rails are used in LD language for representing the limits of a rung.

The power rail on the left represents the TRUE value and initiates the rung state. The power rail on the right receives connections from the coils and has no influence on the execution of the program.

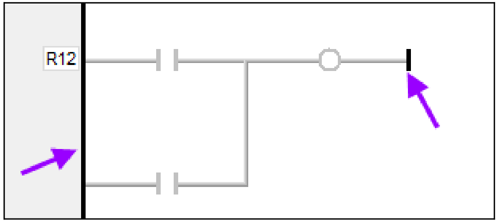


Figure 93: Vertical power rails (left and right)

Power rails can also be used in FBD language. Only Boolean objects can be connected to left and right power rails.

6.3.6 Calling a Function or Function Block

To call a function block in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

All available operators, functions and function blocks are listed in the bottom/right area of the editor. The list of available blocks is sorted into categories.

The All category enables you to see the complete list of available blocks. The Recent category contains the last used blocks. The Project category lists all UDFBs and sub-programs declared in the project.

To insert a block in a program, simply select it in the list and drag it with the mouse to the wished position in the ladder diagram.

Press the F1 key when a block is selected to have help about its function, input and output pins. In selection mode, you also can double-click the mouse on a block of the diagram to change its type, and set the number of input pins if the block can be extended.

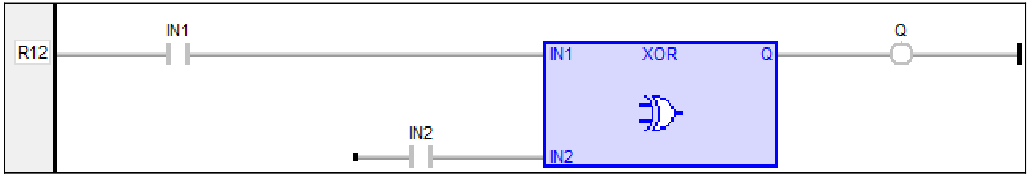
6.3.6.1 EN Input and ENO Output for Blocks

The rung state in a LD diagram is always Boolean. Blocks are connected to the rung with their first input and output. This implies that special EN and ENO input and output are added to the block if its first input or output is not Boolean.

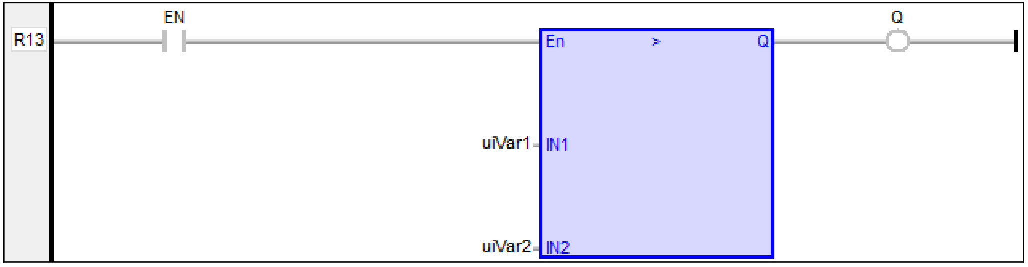
The EN input is a condition. It means that the operation represented by the block is not performed if the rung state (EN) is FALSE. The ENO output always represents the sane status as the EN input: the rung state is not modified by a block having an ENO output.

Examples:

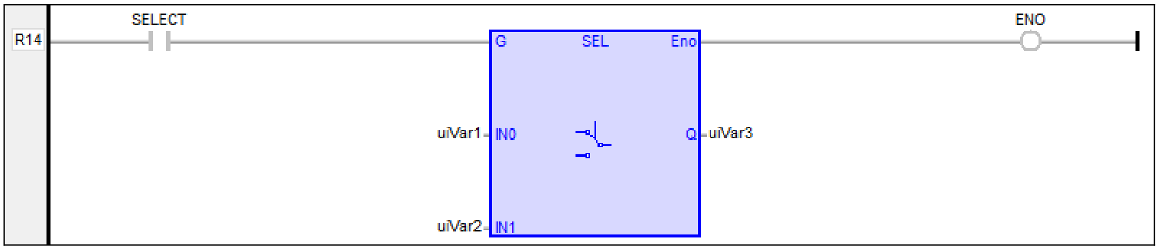
1. XOR block, having **Boolean inputs and outputs**, and requiring no EN or ENO pin. First input is the rung. The rung is the output.



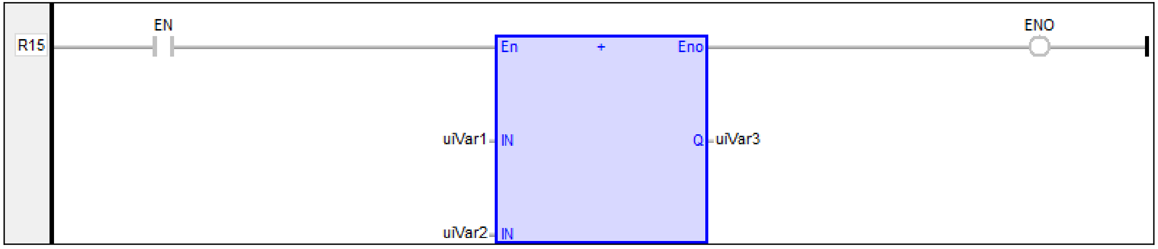
2. > (greater than) block, having **no Boolean inputs and a Boolean output**. This block has an EN input in LD language.
   The comparison is executed only if EN is TRUE.



3. The SEL function has a **Boolean input**, but **no Boolean output**. This block has an ENO output in LD language.
   The input rung is the selector. ENO has the same value as SELECT.



4. Addition, having only numerical arguments (**no Boolean inputs and no Boolean output**). This block has both EN and ENO pins in LD language.
   The addition is executed only if EN is TRUE. ENO is equal to EN.



6.3.7 Jumps - Labels

A jump to a label branches the execution of the program after the specified label. The jump is performed only if the input is TRUE. In LD language the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE.

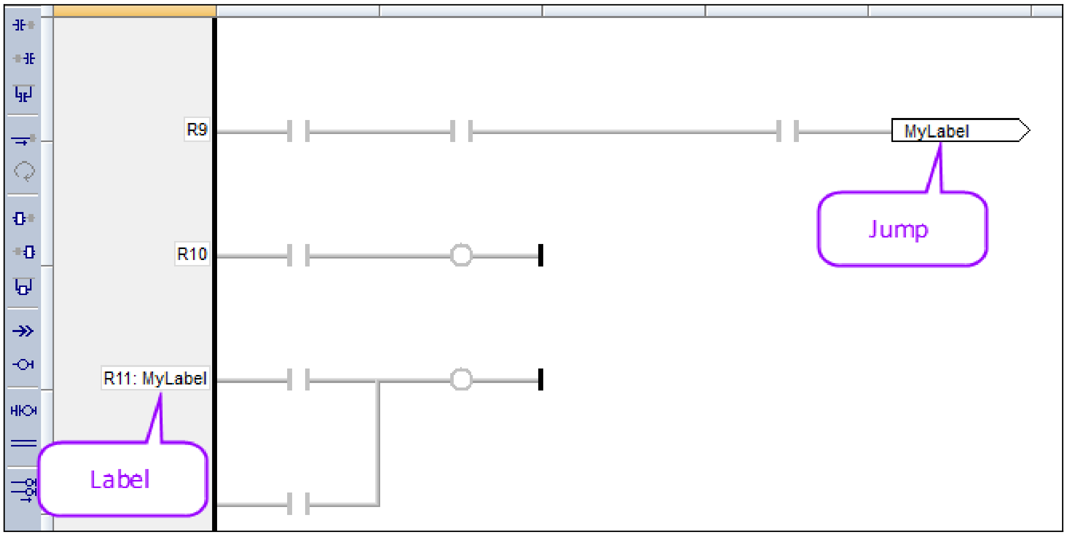


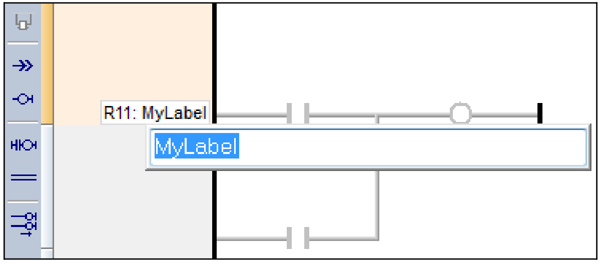
Figure 94: Jump-label in a ladder diagram

A label must be represented by a unique name and is shown on the left side of the rung.

- A jump is added by first selecting the target cell in the ladder editor and clicking the

'Insert jump' » command.

- A label is inserted to the LD editor by double clicking the first cell of a rung and entering the label name in the popup edit box.



Attention

Backward jumps may lead to infinite loops that block the target cycle.

6.3.8 Use of ST Expressions

The workbench enables any complex Structure Text (ST) expression to be associated with a graphic element in either LD or FBD language. This feature makes it possible to simplify LD and FBD diagrams when some trivial calculation has to be entered. It also enables you to use graphic features for representing a main algorithm where as text is used for detailed implementation.

Expression must be written in ST language. An expression is anything you can imagine between parenthesis in a ST program. Obviously the ST expression must meet the data type required by the diagram (e.g. an expression put on a contact must be Boolean).

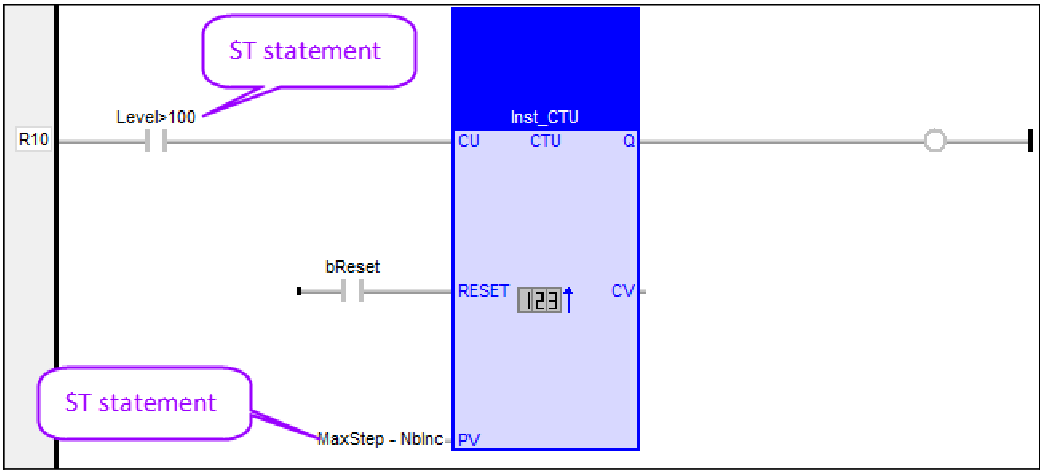


Figure 95: ST expression as input in a LD editor

6.3.9 Comments in LD Diagrams

The LD editor enables you to insert comment texts in the diagram. A comments is a single line of text inserted between two rungs. The comment text is displayed on a double line in the diagram:

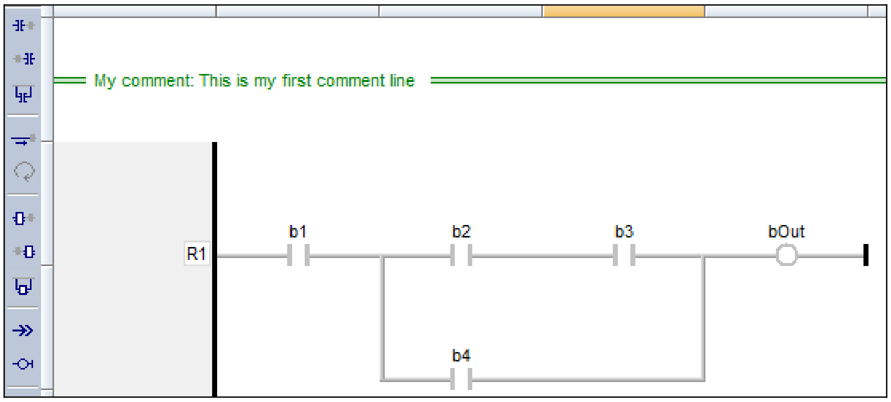


Figure 96: Comments (green) in a LD editor

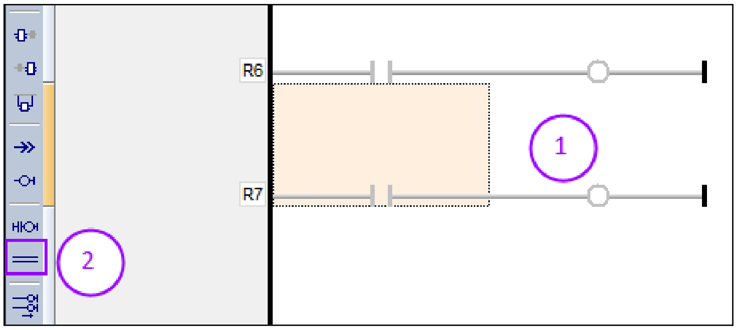
Comment texts have no meaning for the execution of the diagram. They are used to enhance the readability of the program, enabling the description of each rung.

The comment text remains visible when the diagram is scrolled horizontally. To change the text of the comment, place the selection anywhere on the comment line and hit ENTER key, or simply double-click on the comment line.

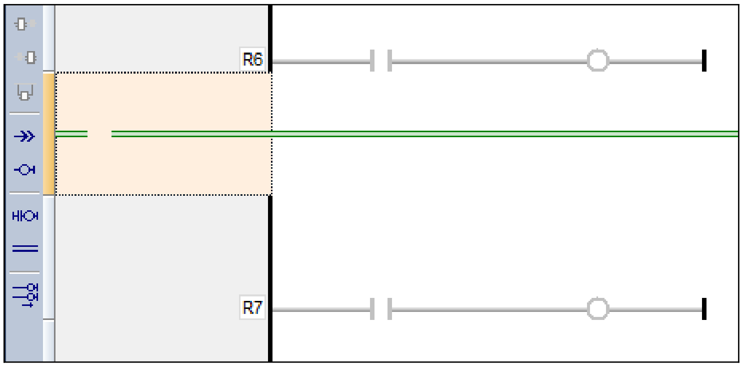
Steps for adding a comment line:

Step 1: Select a line at the which the comment has to be inserted.

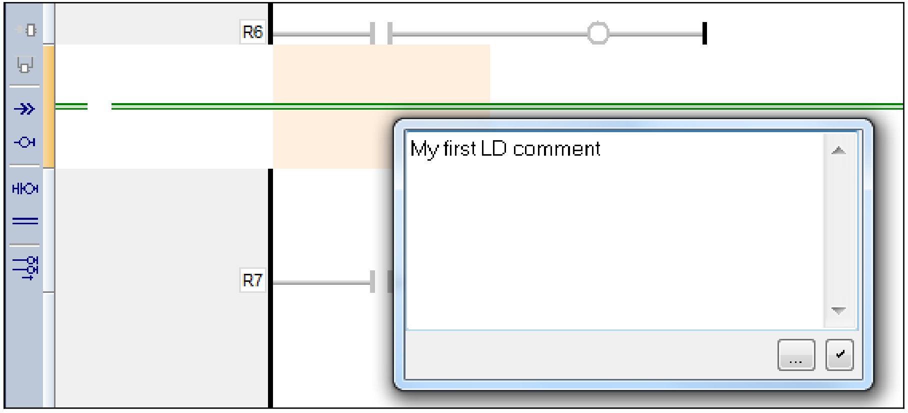
Step 2: Click on the *'Insert comment line'* button on the left toolbar



A new comment line is added one line above the selected line:

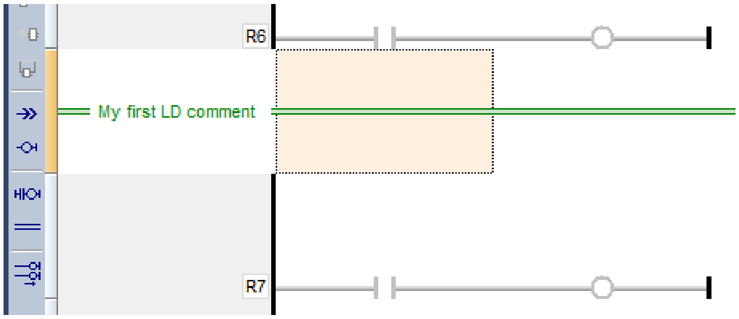


Step 3: Insert text to the comment line by double clicking the line:



My first LD comment

Step 4: Click the check button to insert the text:



My first LD comment

6.3.10 Viewing LD diagrams

The diagram is entered in a logical grid. All objects are snapped to the grid. You can use the commands of the View menu for displaying of hiding grid lines.

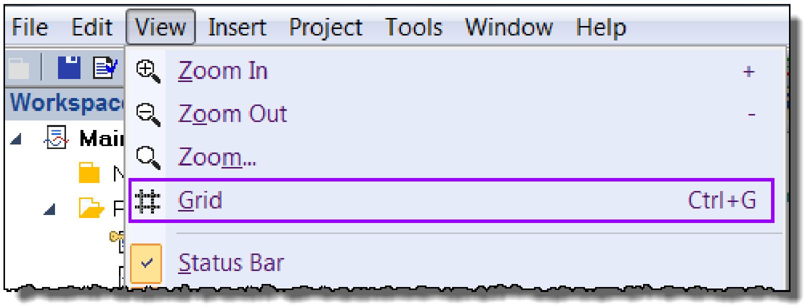


Figure 97: Displaying/hiding grid line

The (x,y) coordinates of the mouse cursor are displayed in the status bar. This helps you locating errors detected by the compiler, or aligning objects in the diagram.

At any moment you can use the commands of the View menu for zooming in or out the edited diagram. You also can press the [+] and [-] keys of the numerical keypad for zooming the diagram in or out.

You also can drag the separation lines in vertical and horizontal rulers to freely resize the cells of the grid:

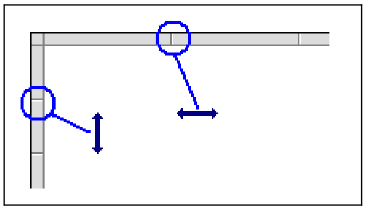


Figure 98: Resizing the cells

The LD editor adjust the size of the font according to the zoom ratio so that the name of variables associated with contacts and coils are always visible. If cells have sufficient height, variable names are completed with other pieces of information about the variable:

- Its tag (short description).
- Its description text.
- Its I/O name (%...) if the variable has a user defined name.

6.3.11 Moving and Copying LD Objects

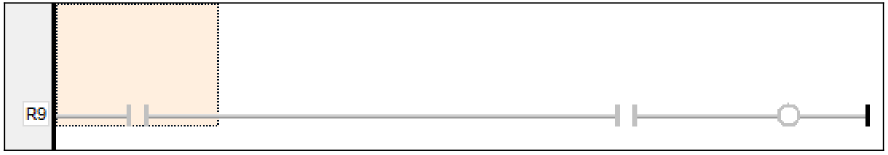
The LD editor fully supports drag and drop for moving or copying objects.

6.3.11.1 Moving LD Objects

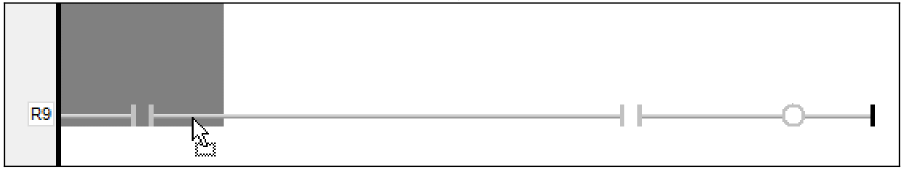
To move objects, select them and simply drag them to the wished position, in the same rung or in another rung.
Alternatively, you can use classical Cut/Paste commands from the Edit menu. Paste is performed at the current position.
At any moment while dragging objects you can press ESCAPE to cancel the operation.

Step 1: Select the item to move:

1. Click on the item to move. The background of the selected item turns yellow.

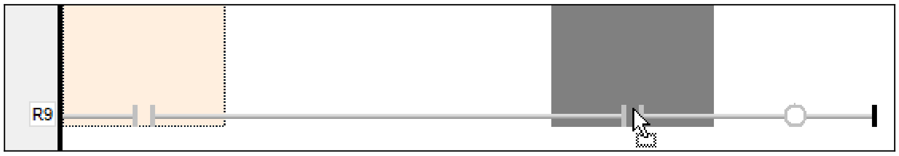


2. Now click on the item again and hold the left mouse button down until the background color turns grey. A rectangular attached to the mouse icon indicates that the item is ready to be moved.

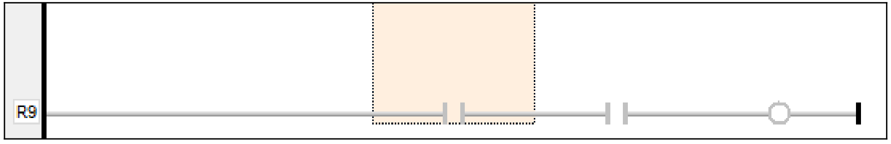


Step 2: Move the object

1. With the left mouse button still down move the item to a new position.
2. Release the mouse button when the target position has been reached. Remember that item is moved to the cell which precedes the selected cell.



After the left mouse button has been release the item is dropped to the cell which column precedes the selected target column.



Step 3: hh

6.3.11.2 Copying LD Objects

To copy objects, select them and just press the *'Ctrl'* key while dragging. It is also possible to drag pieces of diagrams from a program to another if both are open and visible on the screen. The procedure is the same as moving object except that in addition the *'Ctrl'* key has to be pressed while dragging the object.

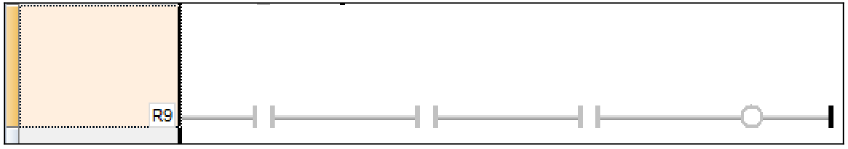
Alternatively, you can use classical Copy/Paste commands from the Edit menu. Paste is performed at the current position.

At any moment while dragging objects you can press ESCAPE to cancel the operation.

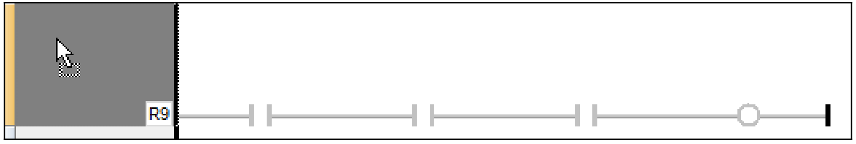
You can manipulate whole rungs by selecting only their head in the left margin (select only the cell where the rung number is displayed).

Step 1: Select the rung to copy:

1. Click on the first cell of the row to copy. The background of the selected cell turns yellow.

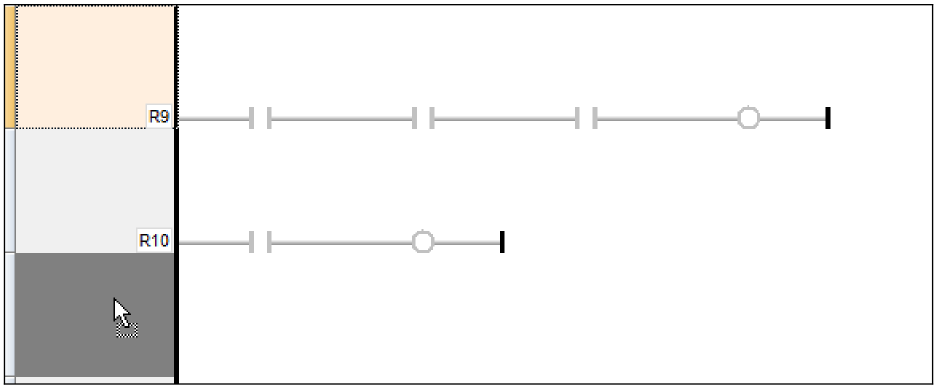


2. Now click on the cell again and hold the left mouse button down until the background color turns grey. A rectangular attached to the mouse icon indicates that the item is ready to be moved or copied.

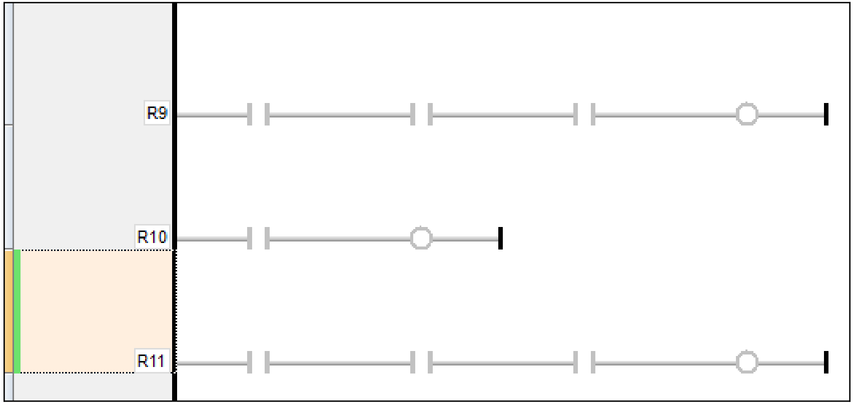


Step 2: Copy the rung:

1. With the left mouse button still down press the *'Ctrl'* key move the item to a new position.
2. Release the mouse button when the target position has been reached.

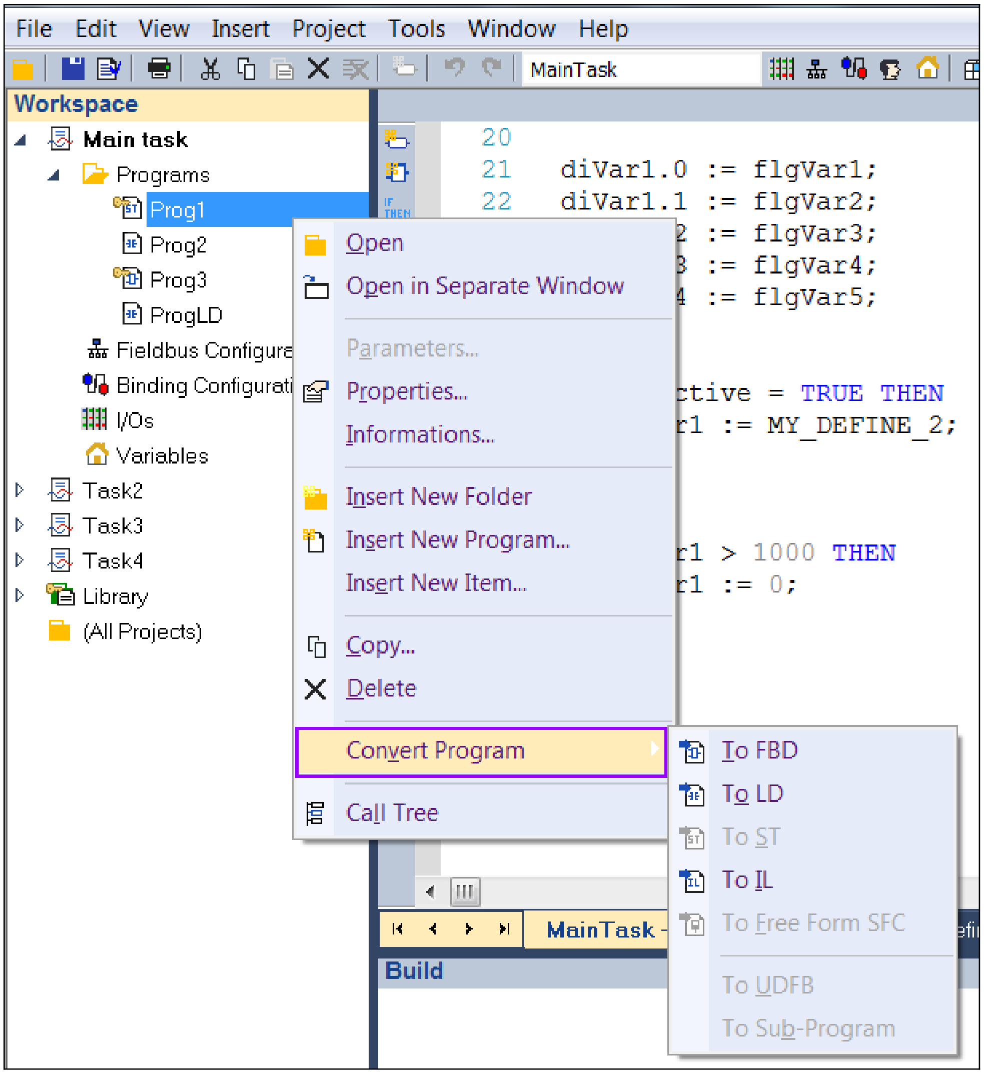


The selected rung is copied to the selected row:



6.4 Converting a Program to Another Language

The Workbench includes a feature that enables you to convert a program written in an IEC language to another language. Use the contextual popup menu in the Workspace to convert a program to another language.



```
diVar1.0 := flgVar1;
diVar1.1 := flgVar2;
       2 := flgVar3;
       3 := flgVar4;
       4 := flgVar5;

ctive = TRUE THEN
r1 := MY_DEFINE_2;

r1 > 1000 THEN
r1 := 0;
```

Figure 99: Converting a Structure Text into another language

Any conversion in between ST, IL, LD and FBD are possible. Conversion of SFC programs is not allowed. Converted POUs can be main programs, UDFBs or sub-programs.

The program must be saved and the project built without errors before converting a program.

When a program is converted to another language, its former implementation in the original language is kept until you make a change in the converted program. Thus, if you re-convert it back to the original language, you get 100% of the original contents.

Info

When a program is converted, some presentation items (such as blank lines in ST or specific alignment in FBD) are lost.

It is possible that converting a language requires some new intermediate variables to be declared. Such variables are declared locally to the program and are prefixed with '*_T*'.

6.5 Some Tips

6.5.1 Bookmarks

Bookmarks are used for navigating in a document. You can freely insert bookmarks everywhere in a document and jump from one bookmark to another with a single command. Bookmarks are supported in all program editors plus the variable editor.

Below are the available commands for using bookmarks:

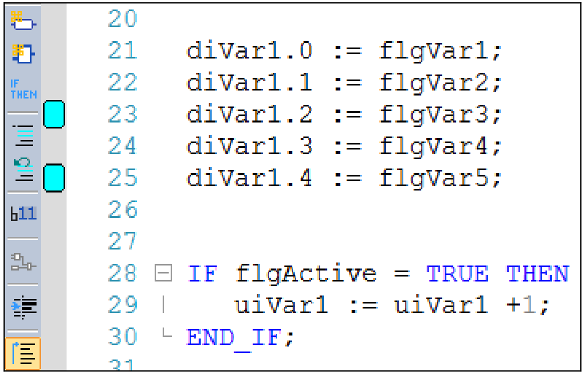
Shortcut	Description
Ctrl + F2	Toggle the bookmark at the current position.
Shift + F2	Go to the next bookmark.

Table 16: Bookmark commands

Info:
Bookmarks are valid only while the editing window is open, and are not stored in the document when the window is closed.

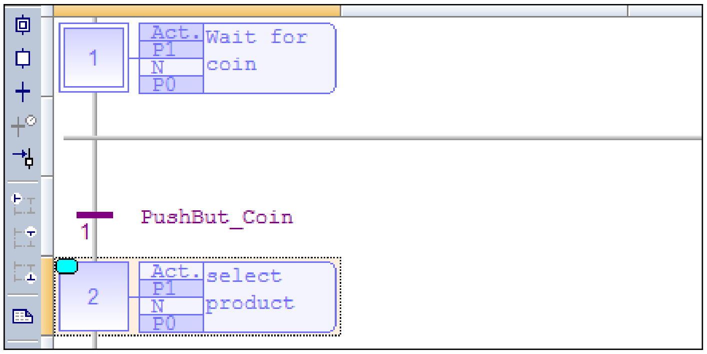
The possible locations for a bookmark are:

- In the text editor, a bookmark is placed on a line of text.

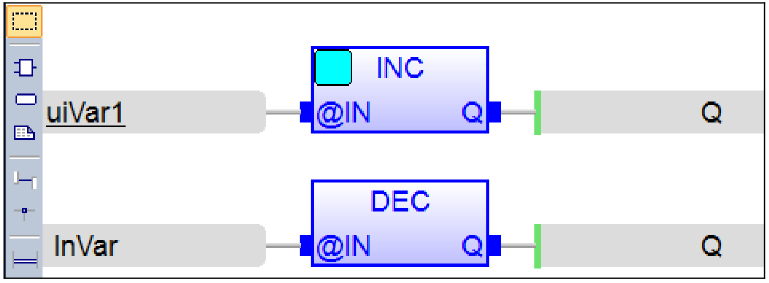


```
diVar1.0 := flgVar1;
diVar1.1 := flgVar2;
diVar1.2 := flgVar3;
diVar1.3 := flgVar4;
diVar1.4 := flgVar5;


IF flgActive = TRUE THEN
    uiVar1 := uiVar1 +1;
END_IF;
```

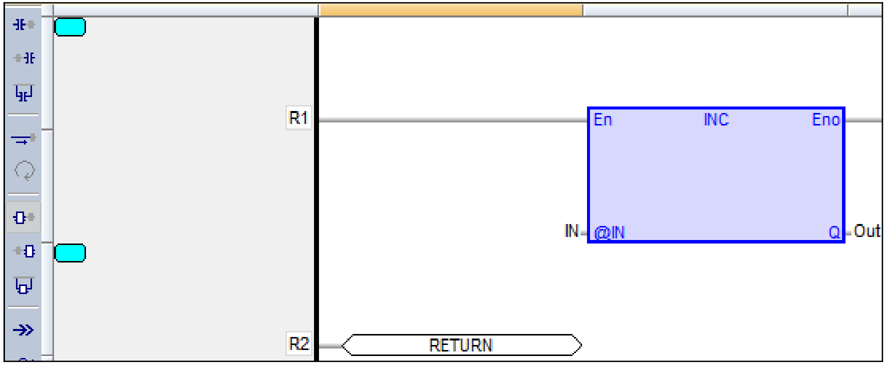
- In the SFC editor, a bookmark is placed on a SFC symbol (step, transition, jump...).



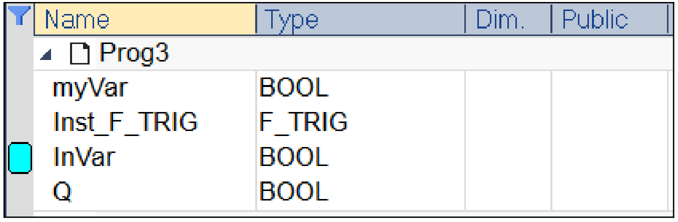
- In the FBD editor, a bookmark is placed on any FBD object (not on a line).



- In the LD editor, a bookmark is placed on a rung header.



- In the variable editor, a bookmark is placed on any line of the grid (variable or group).



Name	Type	Dim.	Public
◢ Prog3			
myVar	BOOL		
Inst_F_TRIG	F_TRIG		
InVar	BOOL		
Q	BOOL		

6.5.2 Handling Exceptions

The compiler enables you to write your own exception programs for handling particular system events. The following exceptions can be handled:

- Startup (before the first cycle)
- Shutdown (after the last cycle)
- Division by zero
- Array index out of bounds

6.5.2.1 Startup

You can write your own exception program to be executed before the first application cycle is executed. Add the following command to the to define editor of the startup exception program:

```
#OnStartup ProgramName
```

The *'ProgramName'* has to be replaced by the name of the exception program.

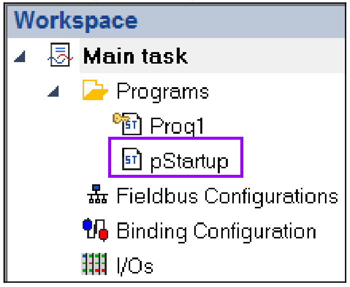
The startup exception program is executed before all other programs within the first cycle. The program is called only once before the first cycle. This implies that the cycle timing may be longer during the first cycle. You cannot put breakpoints in the Startup program.

In the startup program you can system initialization:

- Start other tasks executions
- Initialization of complex data structures
- Initialize variables from recipe: *'Initial values'*
    `ApplyRecipeColumn ('Initial values', 0);`
- Fieldbus intialization

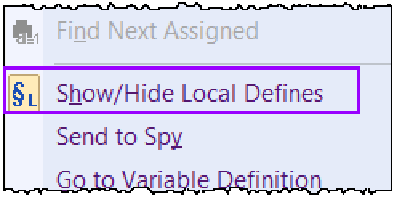
Procedure for adding an exception program:

Step 1: Create a new program that will handle the exception. It cannot be a SFC program. The exception program can have any name. In this example it is called *'pStartup'*.



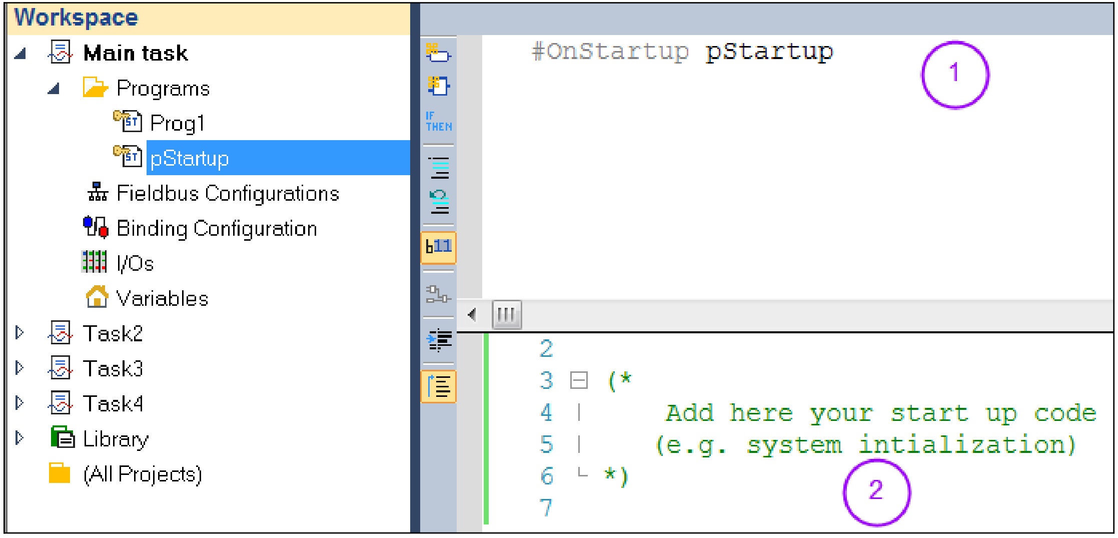
Step 2: Edit code to the exception program:

1. Inform the compiler that the program is an exception program:
   - Open the define editor by right clicking the program editor and selecting *'Show/Hide Local Defines'*



   - Insert the following line in the:
     `#OnStartup pStartup`

2. Edit the source code for the exception program



```
#OnStartup pStartup
```

```
(*
       Add here your start up code
     (e.g. system intialization)
*)
```

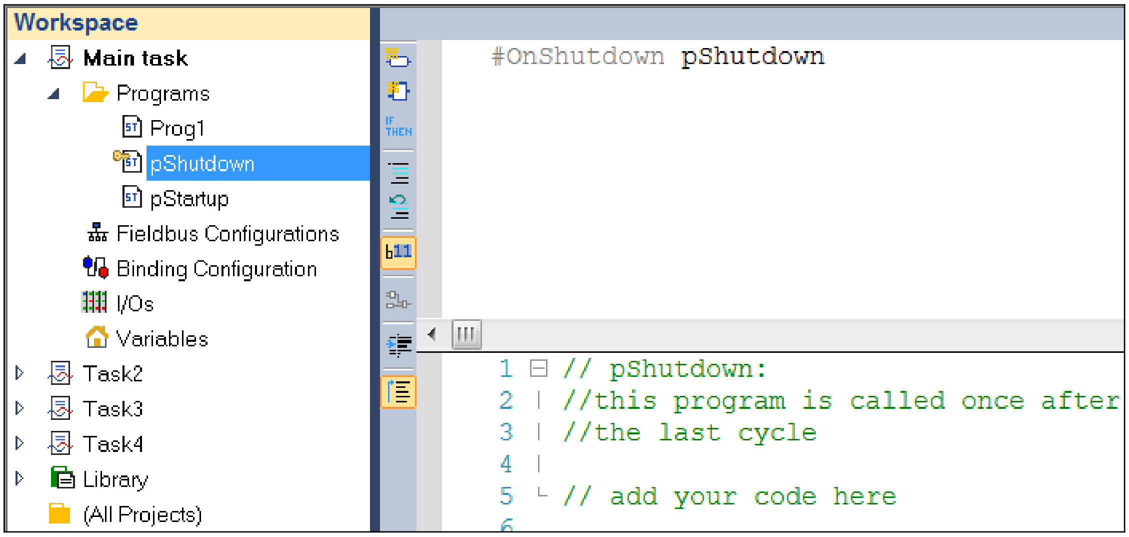
6.5.2.2 Shutdown

You can write your own exception program to be executed after the last application cycle when the runtime system is cleanly stopped.

Create a new program that will handle the shutdown exception. It cannot be a SFC program. In the local define editor of the shutdown program, insert the following line:

```
#OnShutdown ProgramName
```

The *'ProgramName'* should to be replaced by the name of the exception program (Figure 100).
You cannot put breakpoints in the Shutdown program.



```
#OnShutdown pShutdown
```

```
// pShutdown:
//this program is called once after
//the last cycle

// add your code here
```

Figure 100: Shutdown exception program

6.5.2.3 Division by Zero

You can write your own exception program for handling the *'Division by zero'* exception. Create a new sub-program without any parameter that will handle the exception. In the define section of the subprogram, insert the following line:

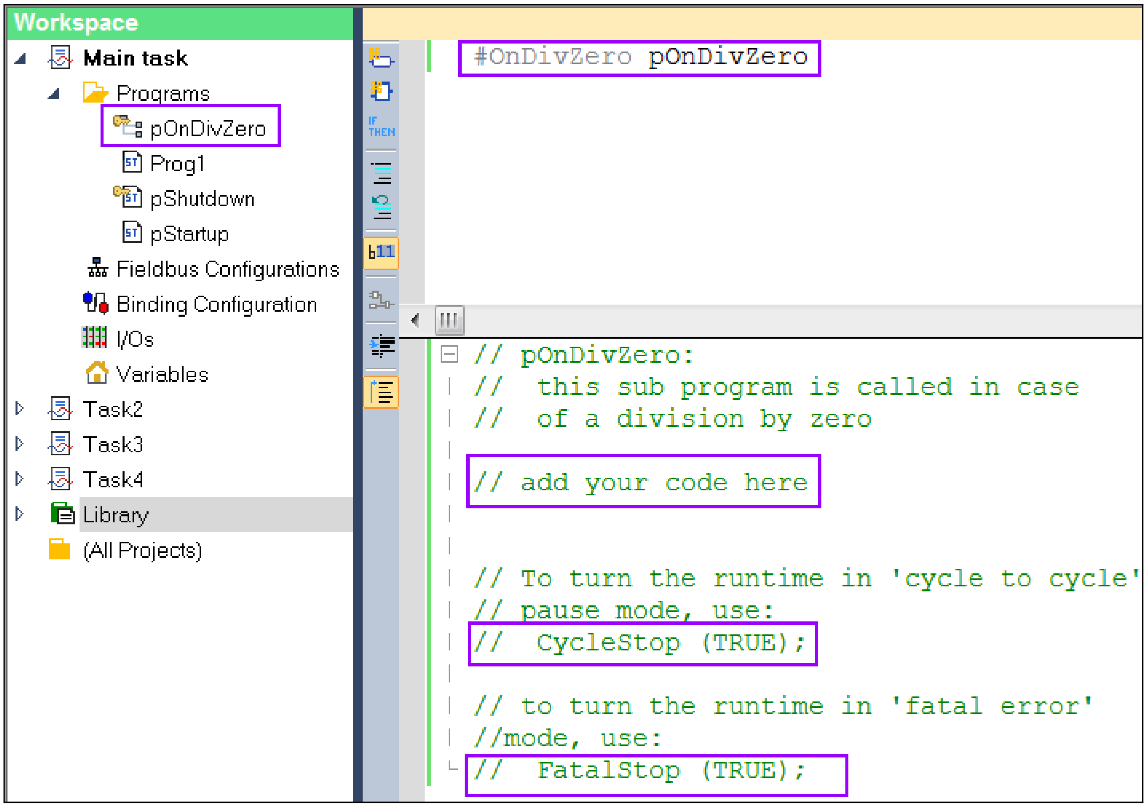
```
#OnDivZero SubProgramName
```

The *'SubProgramName'* should to be replaced by the name of the exception sub-program.

In the sub-program that handles the exception you can perform any safety or trace operation. You then have the selection between the following possibilities (Figure 101):

- Return without any special call. In that case the standard handling will be performed: a system error message is generated, the result of the division is replaced by a maximum value and the application continues.
- Call the **FatalStop** function. The runtime then stops immediately in Fatal Error mode.
- Call the **CycleStop** function. The runtime finishes the current program and then turns in cycle setting mode.

Handlers can also be used in DEBUG mode for tracking the bad operation. Just put a breakpoint in your handler. When stopped, the call stack will show you the location of the division in the source code of the program.



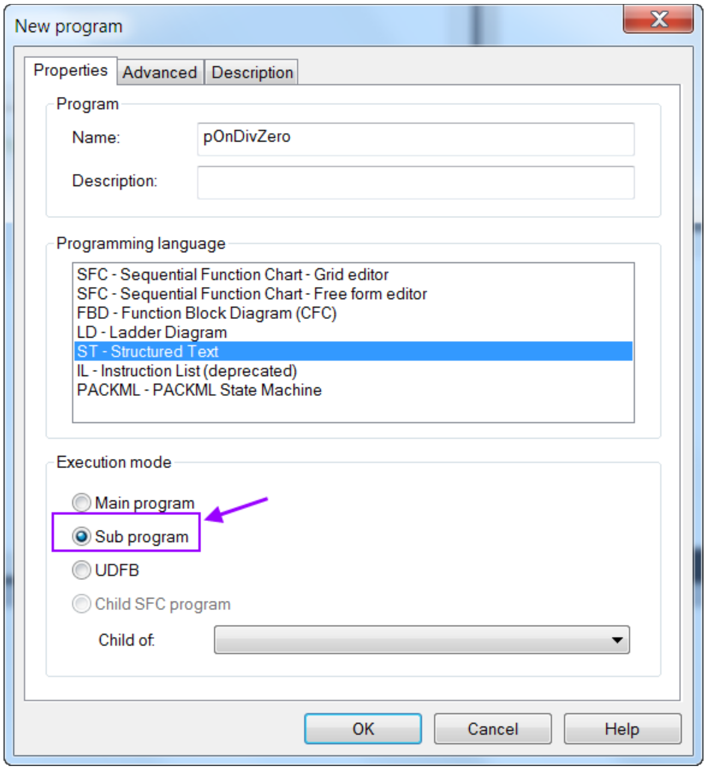
```
#OnDivZero pOnDivZero
```

```
// pOnDivZero:
//   this sub program is called in case
//   of a division by zero

// add your code here


// To turn the runtime in 'cycle to cycle'
// pause mode, use:
//   CycleStop (TRUE);

// to turn the runtime in 'fatal error'
//mode, use:
//   FatalStop (TRUE);
```

Figure 101: 'Divide by zero'- exception sub-program

Below is the procedure you must follow for setting an exception handler:

Step 1: Create a sub-program. In this example the name of the sub-program is *'pOnDivZero'*.



Program

Name: pOnDivZero

Description:

Programming language

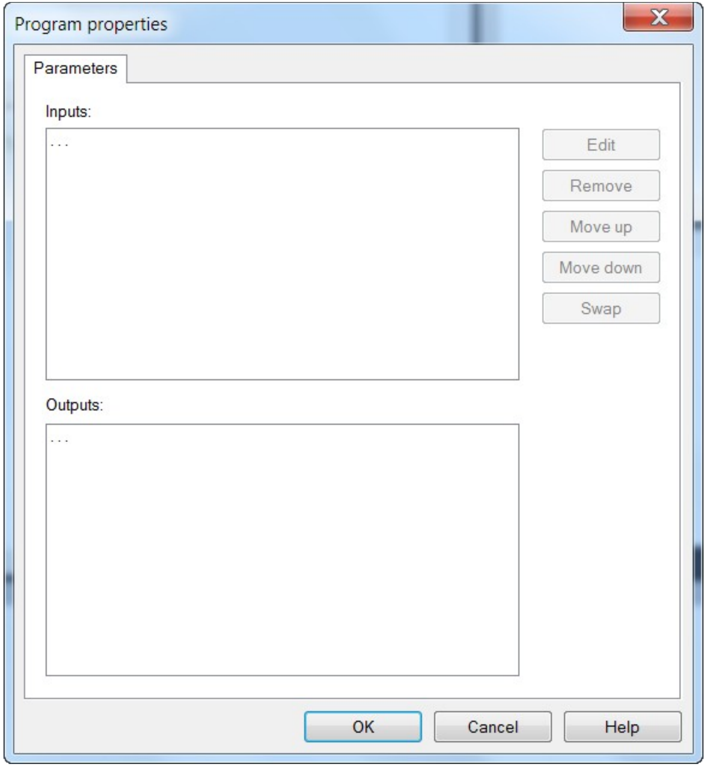
SFC - Sequential Function Chart - Grid editor
SFC - Sequential Function Chart - Free form editor
FBD - Function Block Diagram (CFC)
LD - Ladder Diagram
ST - Structured Text
IL - Instruction List (deprecated)
PACKML - PACKML State Machine

Execution mode

○ Main program
● Sub program
○ UDFB
○ Child SFC program

Child of:

Step 2: Do not define any in/output parameters for the sub-program.



Inputs:

...

Outputs:

...

Step 3: Add the following line to the define editor (Figure 101):

```
#OnDivZero pOnDivZero
```

Step 4: Edit the source code for the exception sub-program.

6.5.2.4 Array Index Out of Bounds

You can write your own exception program for handling the *'Array index out of bounds'* exception. The procedure for the setting an exception handler is similar to the *'Division by zero'* exception (Figure 102):

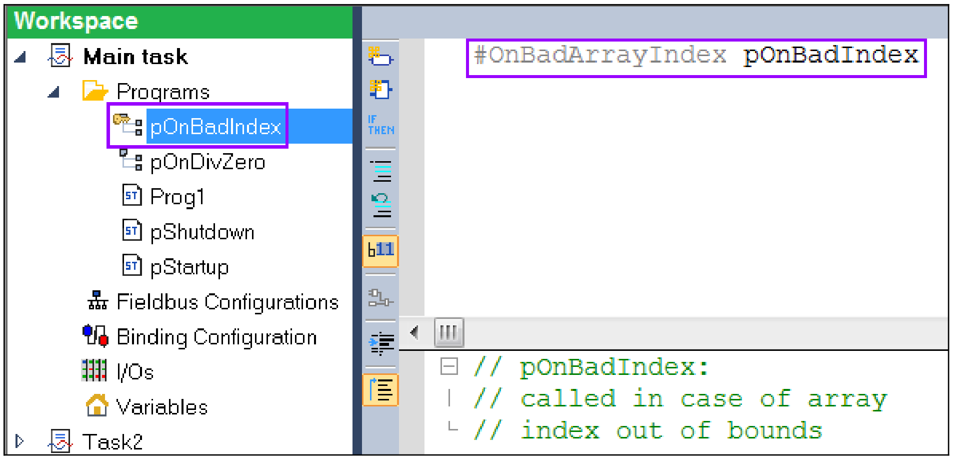
Step 1: First create a new sub-program without any parameters the exception

Step 2: In the define section of the subprogram, insert the following line:

# OnBadArrayIndex SubProgramName

The *'SubProgramName'* should to be replaced by the name of the exception sub-program.

Step 3: Edit the code of the exception program.



```
#OnBadArrayIndex pOnBadIndex
```

```
// pOnBadIndex:
// called in case of array
// index out of bounds
```

Figure 102: Array index exception sub-program

Note:

The array index out of bound error is a fatal error. If the *'Check array bounds'* compiling option is set, the runtime goes in *'fatal error'* mode after calling your sub-program.

7 Variable Monitoring (Debugging Tools)

The workbench provides several windows and tools for monitoring and manipulating variables while the PLC program is executing. The workbench has to be connected to the runtime and the application project needs to be open in order for the workbench to display the current variable values.

The following tools are available for monitoring and debugging the application program:

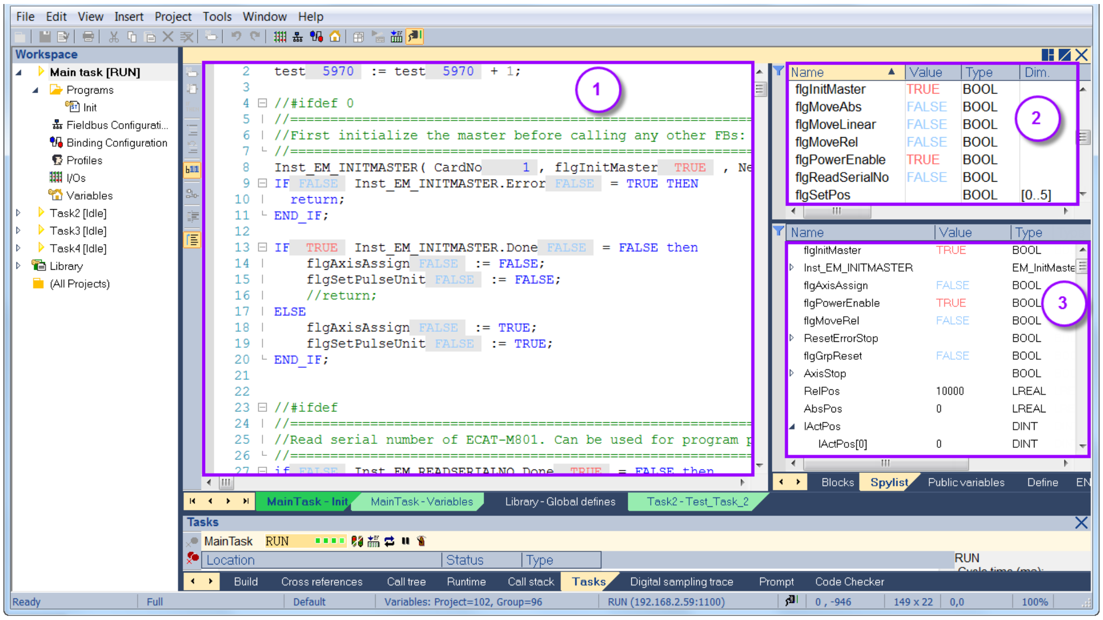
- Monitoring variables
- Diagnostic information: A diagnostic window displays messages of the runtime
- SpyList: Variable values are displayed in the program or in the lists of variable. Variable lists allow you to spy on real-time values of variables. SpyList monitors variable values of more than one task, program or function at the same time.
- Test Sequences: The workbench includes an integrated tool for designing and running automated test scenarios.
- Graphics monitoring: Graphic libraries are provided to create simple graphic interfaces to monitor and write variables. User interfaces for debugging and monitoring purpose can be done via drag and drop and linking the graphic object to a application variable.
- Step by Step debugging: When the program has reached a breakpoint you can execute the program in single steps. At each halt position you see the current value of the variables in the monitoring views. In addition to the cycle by cycle execution mode is supported which stops the program at the end of each cycle.
- Recipes: The recipe manager allows the user to force a number of variables values at the same time.
- Soft scope: An integrated oscilloscope displays variable values in a real time.

7.1 Monitoring Variable Values

When the workbench is connected to the runtime (online) you can monitor the values of the variables of the running application. It is possible to change the value of some variables while you are monitoring the variable values.

7.1.1 Inline Monitoring

If the workbench is online and the inline monitoring is activated, the inline monitoring boxes are placed behind each variable in the code, or next to the variable in the function block. The inline monitoring boxes shows the actual value of the variable in real time (Figure 103: Area 1).



```
test  5970  := test  5970  + 1;

//#ifdef 0
//====================================================================
//First initialize the master before calling any other FBs:
//====================================================================
Inst_EM_INITMASTER( CardNo     1 , flgInitMaster  TRUE  , Ne
IF  FALSE  Inst_EM_INITMASTER.Error  FALSE  = TRUE THEN
  return;
END_IF;

IF  TRUE  Inst_EM_INITMASTER.Done  FALSE  = FALSE then
    flgAxisAssign  FALSE  := FALSE;
    flgSetPulseUnit  FALSE  := FALSE;
    //return;
ELSE
    flgAxisAssign  FALSE  := TRUE;
    flgSetPulseUnit  FALSE  := TRUE;
END_IF;


//#ifdef
//====================================================================
//Read serial number of ECAT-M801. Can be used for program p
//====================================================================
if  FALSE  Inst_EM_READSERIALNO.Done  TRUE  = FALSE then
```

Name ▲	Value	Type	Dim.
flgInitMaster	TRUE	BOOL	
flgMoveAbs	FALSE	BOOL	
flgMoveLinear	FALSE	BOOL	
flgMoveRel	FALSE	BOOL	
flgPowerEnable	TRUE	BOOL	
flgReadSerialNo	FALSE	BOOL	
flgSetPos		BOOL	[0..5]

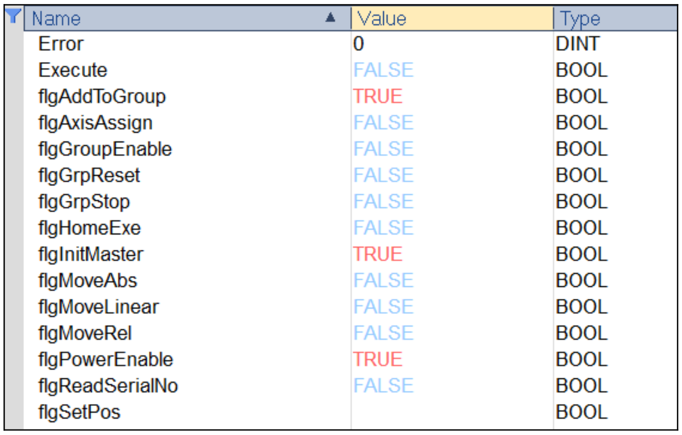
Name	Value	Type
flgInitMaster	TRUE	BOOL
▷ Inst_EM_INITMASTER		EM_InitMaste
flgAxisAssign	FALSE	BOOL
flgPowerEnable	TRUE	BOOL
flgMoveRel	FALSE	BOOL
▷ ResetErrorStop		BOOL
flgGrpReset	FALSE	BOOL
▷ AxisStop		BOOL
RelPos	10000	LREAL
AbsPos	0	LREAL
◢ lActPos		DINT
lActPos[0]	0	DINT

Figure 103: Monitoring variables

To activate or deactivate Inline monitoring click the *'Show Value in Text'* (b11) button on the left edge of the program editor window. Variable values can not be changed directly via the program editor.

7.1.2 Monitoring in the Variable Editor

The variable editor contains all the variable declared for a POU. Once the workbench is connected to a running PLC application each variable is updated with the current application value in real time (Figure 104). The current variable value is shown in the *'Value'* column. If the variable type is an array, structure or function block, then first double click the variable name to open a dialog to show the variables with its values.



Name ▲	Value	Type
Error	0	DINT
Execute	FALSE	BOOL
flgAddToGroup	TRUE	BOOL
flgAxisAssign	FALSE	BOOL
flgGroupEnable	FALSE	BOOL
flgGrpReset	FALSE	BOOL
flgGrpStop	FALSE	BOOL
flgHomeExe	FALSE	BOOL
flgInitMaster	TRUE	BOOL
flgMoveAbs	FALSE	BOOL
flgMoveLinear	FALSE	BOOL
flgMoveRel	FALSE	BOOL
flgPowerEnable	TRUE	BOOL
flgReadSerialNo	FALSE	BOOL
flgSetPos		BOOL

Figure 104: Variable editor with showing the current values

You can write and force a value by double clicking a variable in the variable editor. A dialog with the current value pops up (Figure 105). Enter a new value and click the *'Force'* button to changed the variable value while the program is running. The variable will be set to this value at the beginning of the next cycle.

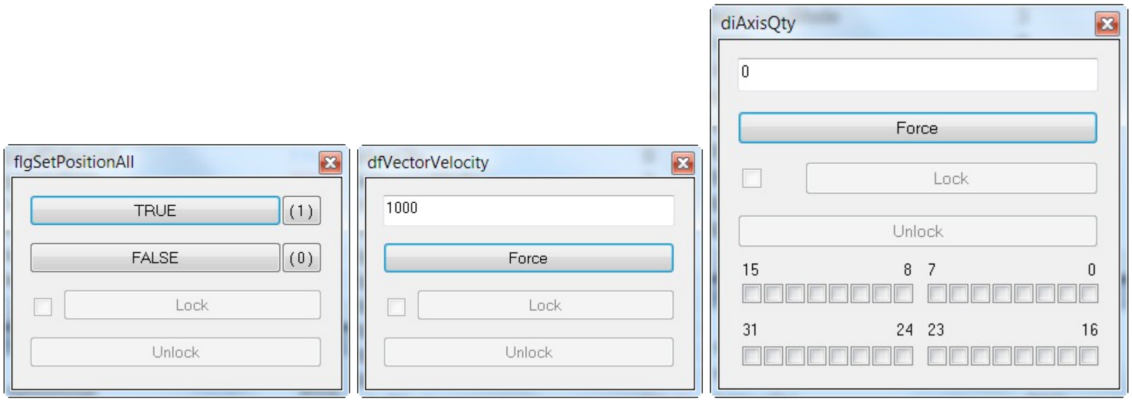


Figure 105: Force variable

7.2 SpyList

The SpyList is a monitoring tool that enables you to watch variables of the application at run time. The user can define a list of project variables that are collected in one view for the purpose of monitoring their values. Variables of simple data type as well as arrays, data structures and statements are all supported by the SpyList. In online mode, you can write and force variable values in a SpyList to actively influence the PLC application behavior. The SpyList configuration with it list of variable can be saved and be used for the next debugging session. The SpyList purpose it to help the programmer to track down errors and resolve problems.

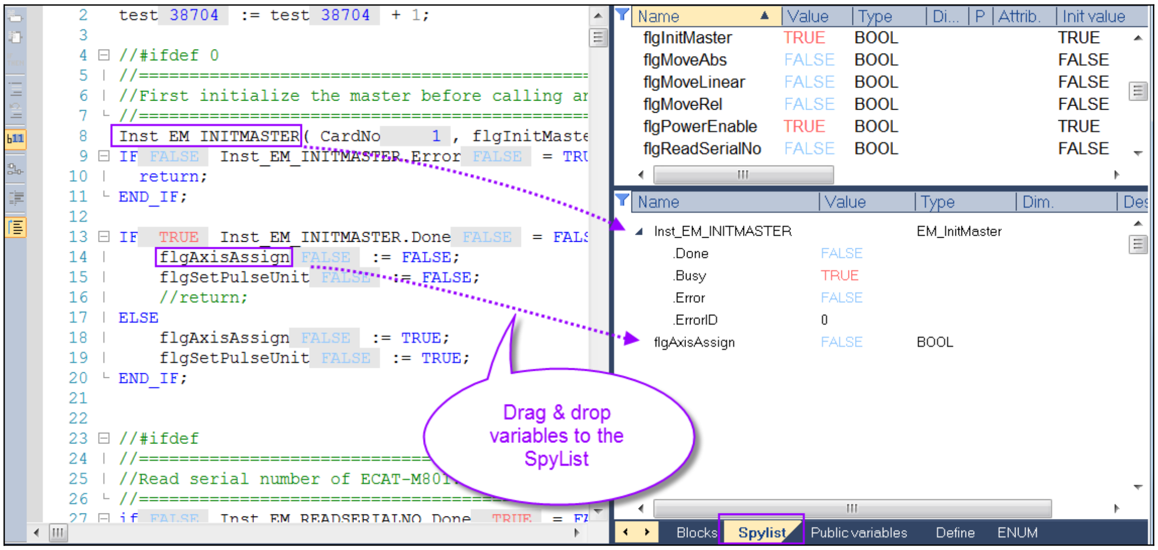
A PLC application program contains many hundreds of declared variables. The workbench allows the user to add a selected number of variables to the SpyList to monitor its current value and state in the runtime. The variables are dynamically updated by the workbench during runtime idle time or when free runtime resource are available and is not updated more than once in a task cycle. The values of each variables in the SpyList can be changed by the user.

SpyList can reference global, system, and local variables. Three type of SpyLists are provided:

- Program SpyList (Local SpyList): Only variables which are declared in the program are supported by the local SpyList.
- Task SpyList: A task may have several programs; variable of any program in the task can be added to the Task SpyList
- Multi SpyList: Any variable declared in the application can be monitored

7.2.1 Local SpyList

Each program editor has a local SpyList. If the SpyList is not visible then select in the menu *'View/Info Tab2'* to display it. The variables, structure and function block instances of the local program are added to the local SpyList by simple drag and dropping it from the program or variable editor (Figure 106). In the program editor double click the variable name to mark it and then drag the name over the SpyList window to drop it.



```
test 38704  := test 38704  + 1;

//#ifdef 0
//==============================================
//First initialize the master before calling ar
//==============================================
Inst EM INITMASTER( CardNo      1 , flgInitMaste
IF FALSE  Inst_EM_INITMASTER.Error FALSE  = TRU
  return;
END_IF;

IF  TRUE  Inst_EM_INITMASTER.Done FALSE  = FALS
    flgAxisAssign FALSE  := FALSE;
    flgSetPulseUnit FALSE  := FALSE;
    //return;
ELSE
    flgAxisAssign FALSE  := TRUE;
    flgSetPulseUnit FALSE  := TRUE;
END_IF;


//#ifdef
//==============================
//Read serial number of ECAT-M801
//==============================
if FALSE  Inst EM READSERIALNO Done  TRUE  = FA
```

Name	Value	Type	Di...	P	Attrib.	Init value
flgInitMaster	TRUE	BOOL				TRUE
flgMoveAbs	FALSE	BOOL				FALSE
flgMoveLinear	FALSE	BOOL				FALSE
flgMoveRel	FALSE	BOOL				FALSE
flgPowerEnable	TRUE	BOOL				TRUE
flgReadSerialNo	FALSE	BOOL				FALSE

Name	Value	Type	Dim.	Des
Inst_EM_INITMASTER		EM_InitMaster		
.Done	FALSE			
.Busy	TRUE			
.Error	FALSE			
.ErrorID	0			
flgAxisAssign	FALSE	BOOL		

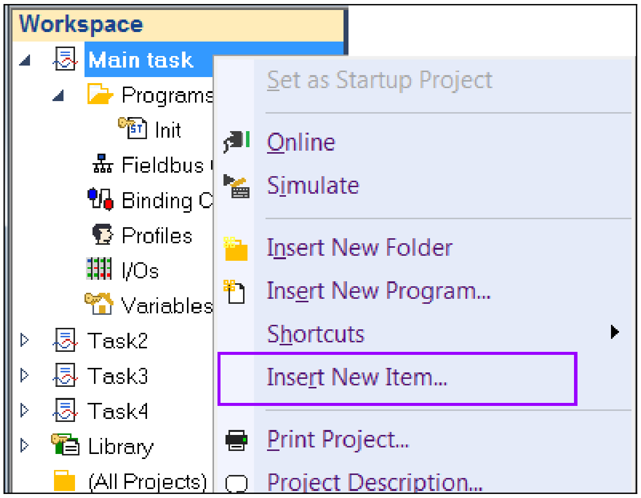
Figure 106: Local SpyList

7.2.2 Task-SpyList

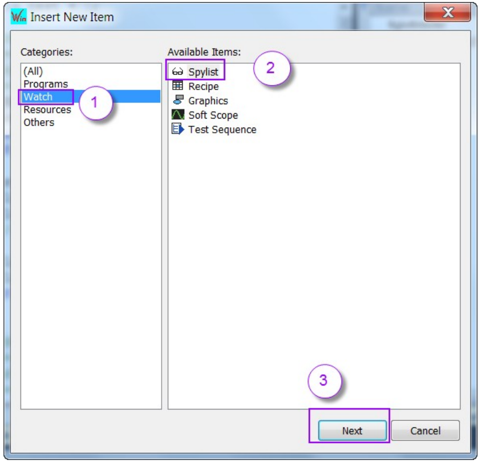
A task controls several program and sub programs. The Task-SpyList monitors all the variables declared within the task. Several SpyList items can be inserted to put the variables into different groups.

Creating a Task-SpyList and adding variables:

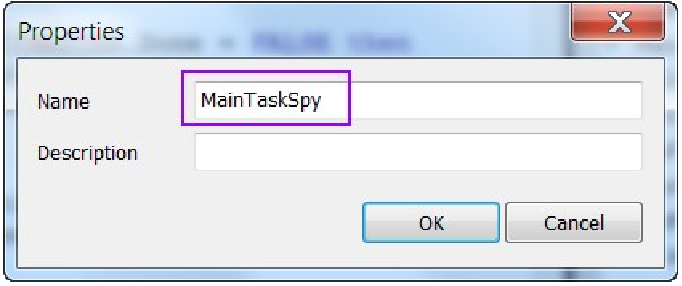
Step 1: Right-click the task name in the workspace and select *'Insert New Item...'*.



Step 2: Select SpyList from the item list



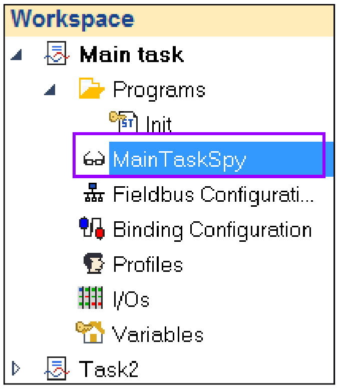
Step 3: Assign the SpyList a name and click *'OK'*



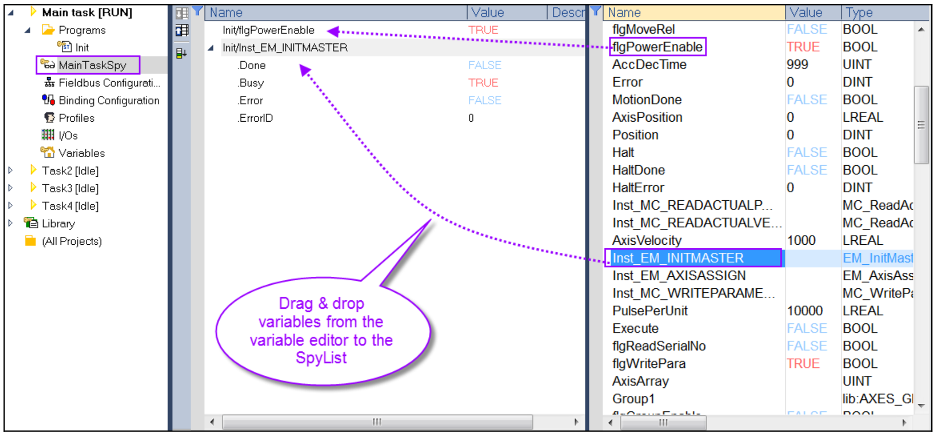
Name: MainTaskSpy

Description:

The new SpyList is being added to the task in the workspace.



Step 4: Drag a variable from the variable editor to the task SpyList. Add all successive variables that will be monitored with this list. You can change the variable order by using drag and drop operations. If the workbench is connected to the runtime then the current variable values are shown.



Name	Value	Descr
Init/flgPowerEnable	TRUE	
Init/Inst_EM_INITMASTER		
.Done	FALSE	
.Busy	TRUE	
.Error	FALSE	
.ErrorID	0	

Name	Value	Type
flgMoveRel	FALSE	BOOL
flgPowerEnable	TRUE	BOOL
AccDecTime	999	UINT
Error	0	DINT
MotionDone	FALSE	BOOL
AxisPosition	0	LREAL
Position	0	DINT
Halt	FALSE	BOOL
HaltDone	FALSE	BOOL
HaltError	0	DINT
Inst_MC_READACTUALP...		MC_ReadAc
Inst_MC_READACTUALVE...		MC_ReadAc
AxisVelocity	1000	LREAL
Inst_EM_INITMASTER		EM_InitMast
Inst_EM_AXISASSIGN		EM_AxisAss
Inst_MC_WRITEPARAME...		MC_WriteP
PulsePerUnit	10000	LREAL
Execute	FALSE	BOOL
flgReadSerialNo	FALSE	BOOL
flgWritePara	TRUE	BOOL
AxisArray		UINT
Group1		lib:AXES_GI

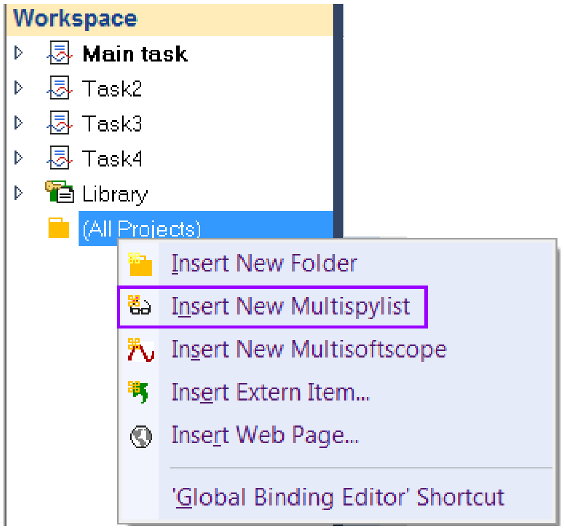
Drag & drop variables from the variable editor to the SpyList

7.2.3 Multi-SpyList

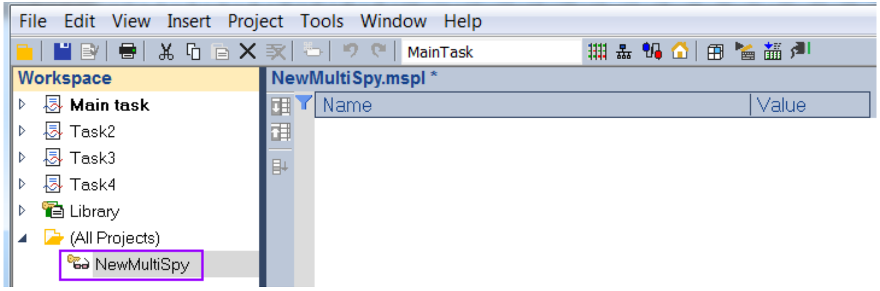
The Multi-SpyList provides the ability to watch variables within your application. Local and global declared variables are support. The variable name with its location with in the program are shown in the list.

Creating a Multi -SpyList and adding variables:

Step 1: Right-click the *'(All Project)'* in the workspace and select *'Insert New Multispylist'*.

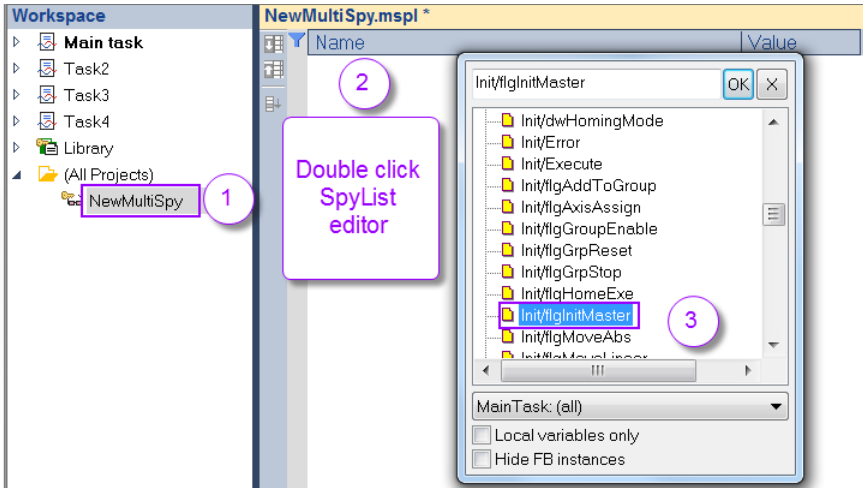


A new Multi-SpyList item will appear in the workspace:

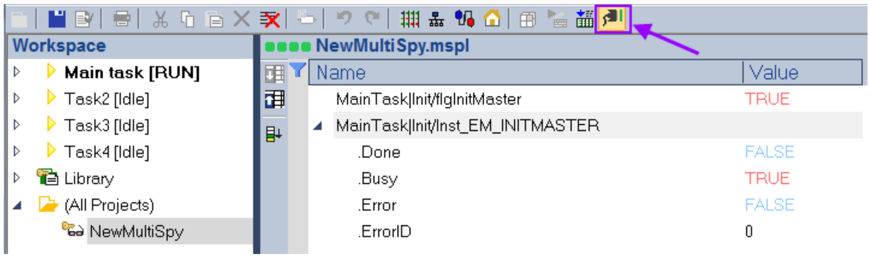


Step 2: Add a variable to the SpyList editor:

1. Open the SpyList editor by double clicking the name in the workspace
2. Double an empty space in the SpyList editor to open a variable list. The list displays all the variables declared for the application project
3. Select a variable from the list and click ok



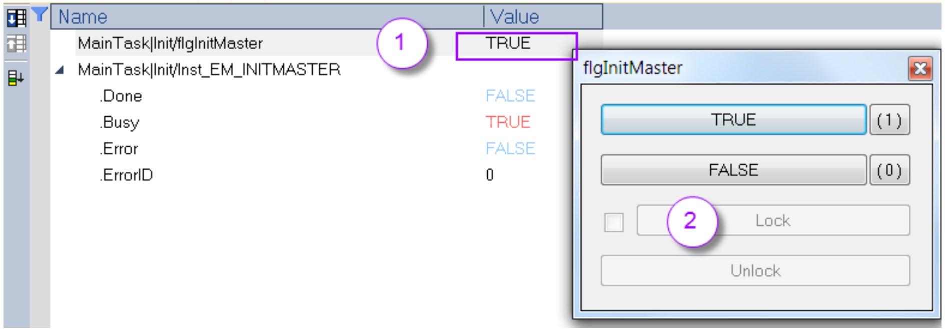
Step 3: Connect the workbench to the runtime. Current variable values are shown.



Name	Value
MainTask|Init/flgInitMaster	TRUE
MainTask|Init/Inst_EM_INITMASTER	
.Done	FALSE
.Busy	TRUE
.Error	FALSE
.ErrorID	0

Step 4: Forcing and writing variable value.

1. Double click the variable value in the *'Value'* column
2. From the popup window enter or select a new value



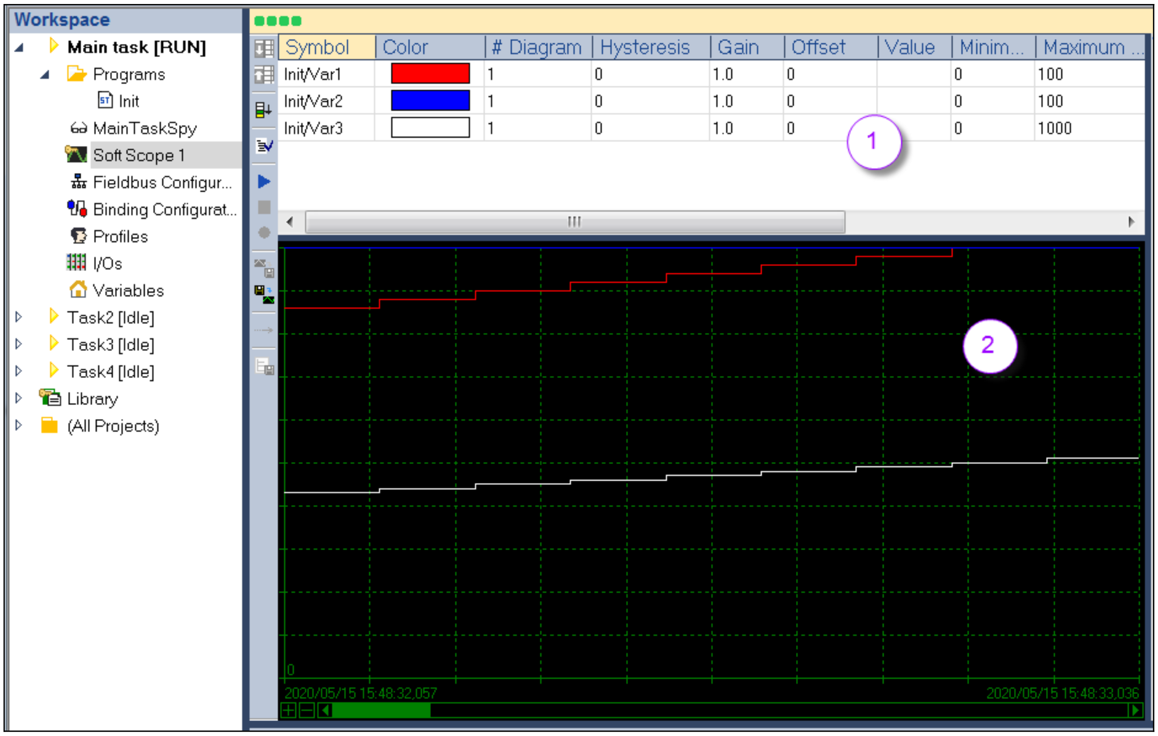
Name	Value
MainTask|Init/flgInitMaster	TRUE
MainTask|Init/Inst_EM_INITMASTER	
.Done	FALSE
.Busy	TRUE
.Error	FALSE
.ErrorID	0

7.3 Soft Oscilloscope

The soft oscilloscope is a tool which allows the user to view in a two-dimensional graph, how the value of one or more variables (vertical axis) evolves over time (horizontal axis). The soft oscilloscope enables you to track the value of boolean or numerical variables and display it in a time curve. Traced variables are tracked by the runtime, which detects changes and assign time stamps to each value record so that the trend displayed is very accurate. The soft oscilloscope is available during online debugging.

Typical applications for using Soft Oscilloscope:

- Tracing the motion path of an axis during motion control execution
- Tracing the feedback position and velocity of an axis
- Any values of digital and analog input channels (current, voltage, temperature, etc.)
- Recording a variable value change in each cycle



Symbol	Color	# Diagram	Hysteresis	Gain	Offset	Value	Minim...	Maximum ...
Init/Var1	🟥	1	0	1.0	0		0	100
Init/Var2	🟦	1	0	1.0	0		0	100
Init/Var3	□	1	0	1.0	0		0	1000

Figure 107: Soft Oscilloscope

The Soft Oscilloscope window is divided into two sections:

1. List of variables to be displayed.
2. Diagram area (oscilloscope display).

In the diagram area, the user can zoom, explore a particular time range and automatically scroll the diagrams.

To add new variables to the oscilloscope display, drag and drop them from the variable editor, or double-click an empty line in the list area. These new variables can be added in both online or offline modes.

The variable list supports the following configuration:

Parameter	Description
Symbol	Name of the traced variable.
Color	Color used to draw the curve.
#Diagram	Index of the diagram pane - default is 1. You can define up to 30 panes.
Hysteresis	Hysteresis to apply for change detection of analog values. The hysteresis is entered as an absolute value.
Value	The current value of the variable is refreshed in this column.
Minimum/Maximum	Range of the Y axis.
Time	Time and date of the last change.

Description	Free description text.

Table 17: Variable display configuration for the Soft Oscilloscope

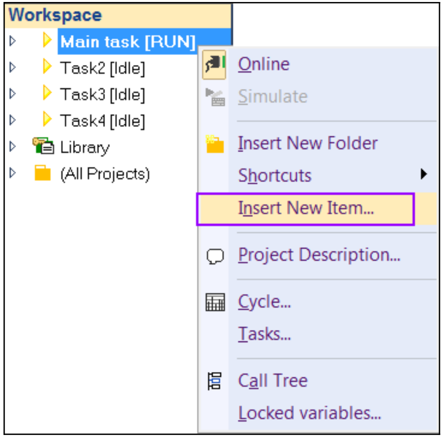
The following commands are available from the Soft Oscilloscope toolbar:

Icon	Description
	Move the selected variable up or down in the list.
	Sort variables according to alphabetic order in the list.
	Set refresh rate and Setup time ranges.
▶	Start the oscilloscope.
■	Stop the oscilloscope.
●	Start recording.
	Save record to the file.
	Reload record from file.
→	Auto-scroll mode (toggle).

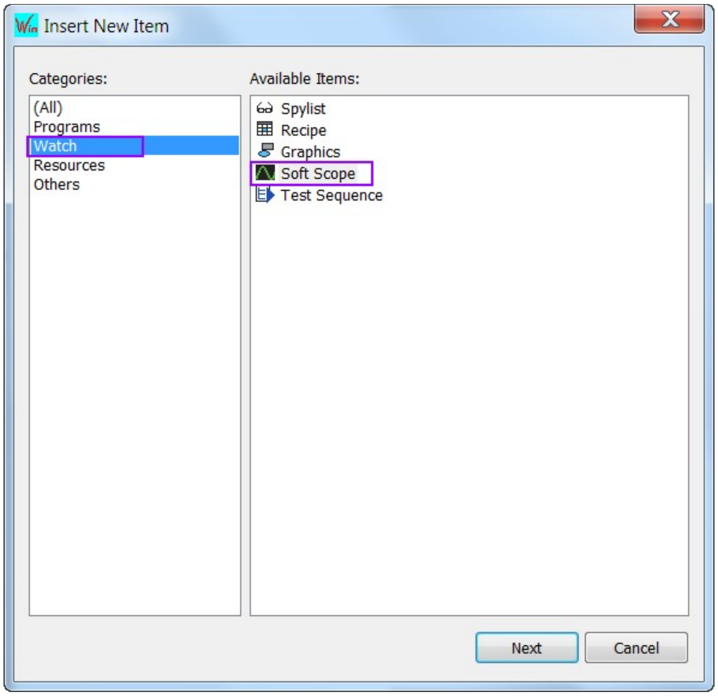
Table 18: Soft Oscilloscope toolbar

Steps for creating and configuring the Soft Oscilloscope:

Step 1: Right click a task name in the workspace and select *'Insert New Item...'* from the pop menu.



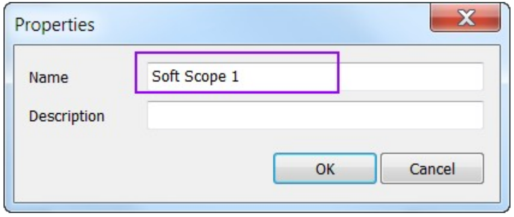
Step 2: Select *'Soft Scope'* from the item list and click *'next'*


Insert New Item

Categories: (All), Programs, Watch, Resources, Others

Available Items: Spylist, Recipe, Graphics, Soft Scope, Test Sequence

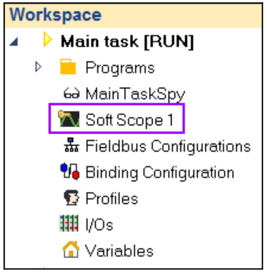
Step 3: Enter a name for the Soft Oscilloscope and click *'OK'*.


Properties

Name: Soft Scope 1

Description:

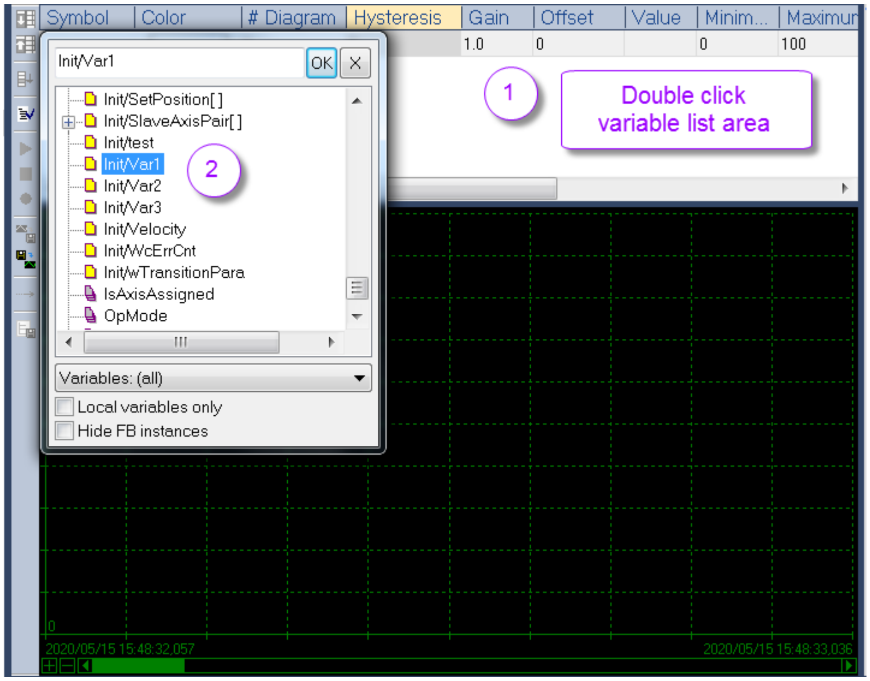
A new Soft Oscilloscope item with the defined name appears in the workspace:


Workspace
- Main task [RUN]
  - Programs
  - MainTaskSpy
  - Soft Scope 1
  - Fieldbus Configurations
  - Binding Configuration
  - Profiles
  - I/Os
  - Variables

Double click the oft Oscilloscope item to open its editor and display window.

Step 4: Select a variable to be displayed in the Soft Oscilloscope:

1. Double click the white area of the variable list. A window pops up which list all the variables declared in the task.
2. Select a variable from the list click *'OK'*



Symbol	Color	# Diagram	Hysteresis	Gain	Offset	Value	Minim...	Maximur
				1.0	0		0	100

Init/Var1

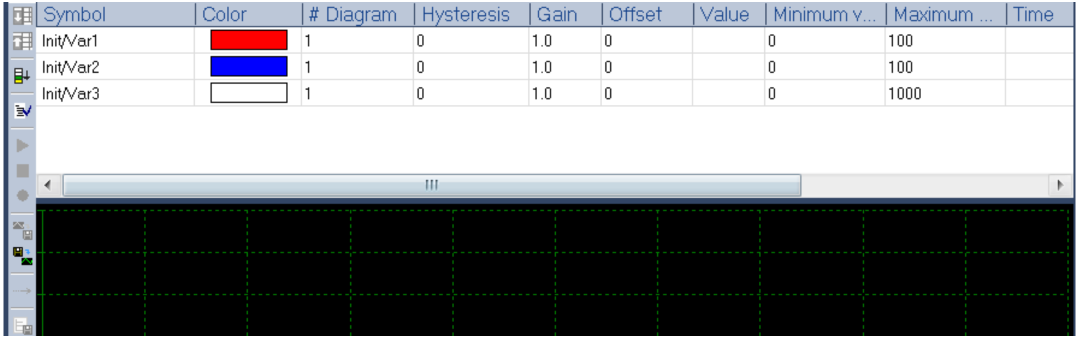
- Init/SetPosition[]
- Init/SlaveAxisPair[]
- Init/test
- Init/Var1
- Init/Var2
- Init/Var3
- Init/Velocity
- Init/WcErrCnt
- Init/wTransitionPara
- IsAxisAssigned
- OpMode

Variables: (all)

□ Local variables only

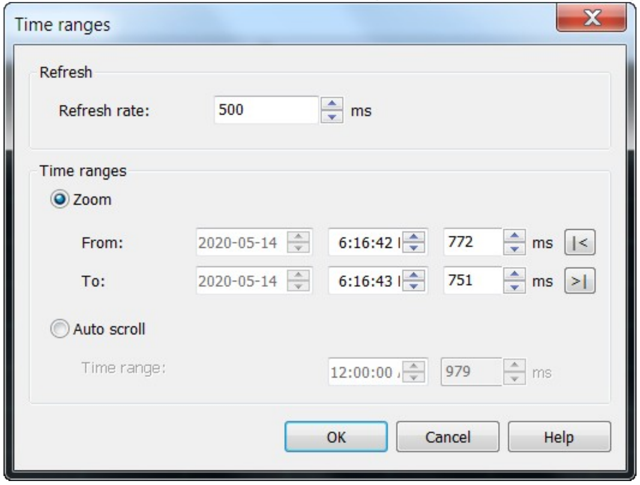
□ Hide FB instances

Step 5: Select the color for displaying the variable curve. Do all the setting described in Table 17. Repeat this process for each variable.



Symbol	Color	# Diagram	Hysteresis	Gain	Offset	Value	Minimum v...	Maximum ...	Time
Init/Var1	🟥	1	0	1.0	0		0	100	
Init/Var2	🟦	1	0	1.0	0		0	100	
Init/Var3	⬜	1	0	1.0	0		0	1000	

Step 6: Set the refresh and time range by double clicking the button:



Time ranges

Refresh

Refresh rate: 500 ms

Time ranges

● Zoom

From: 2020-05-14 6:16:42 772 ms

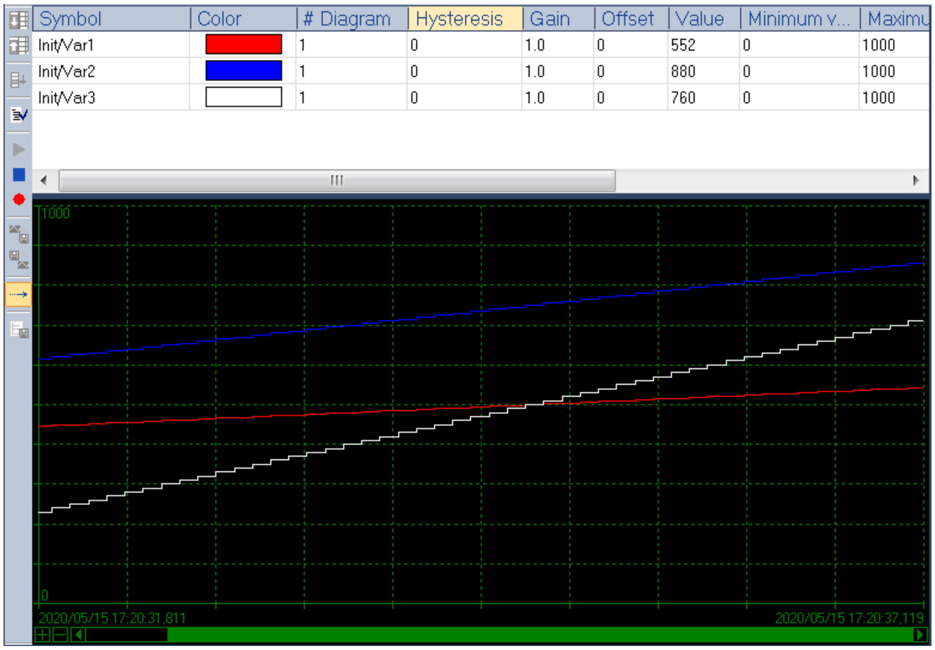
To: 2020-05-14 6:16:43 751 ms

○ Auto scroll

Time range: 12:00:00 979 ms

Step 7: Start the sampling process by clicking ▶.

When sampling is active, you can start recording all events from now by clicking on corresponding (red) button ●. You must specify a csv file where samples will be recorded. All events on all symbols will be recorded in this file until you uncheck recording by clicking the button again.

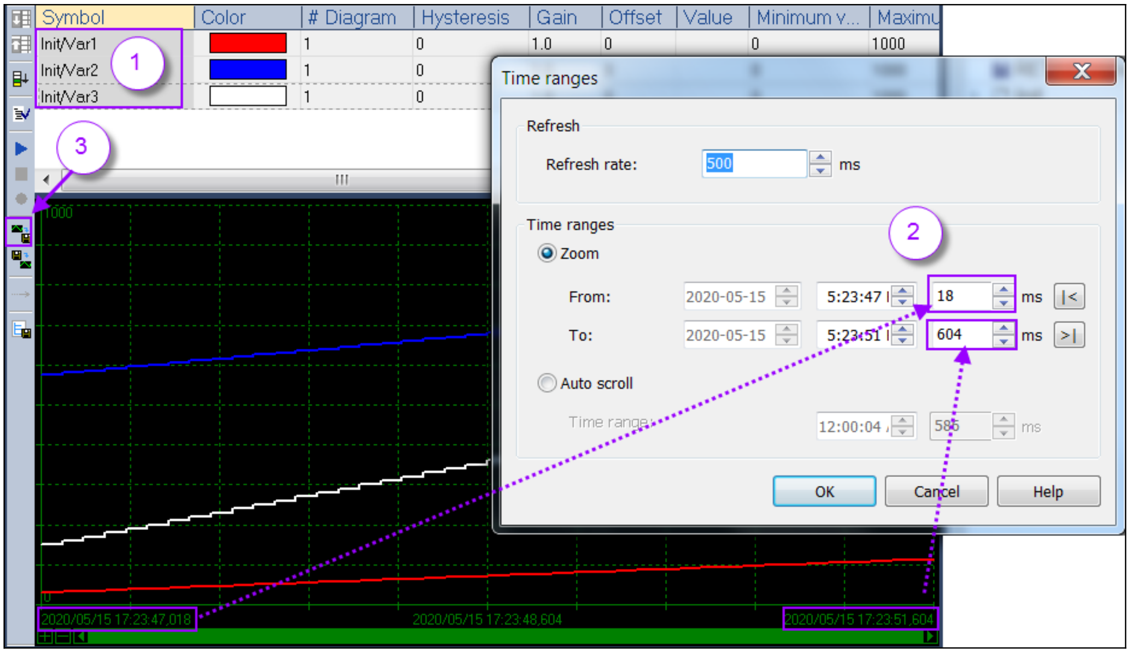


Symbol	Color	# Diagram	Hysteresis	Gain	Offset	Value	Minimum v...	Maximu
Init/Var1	🟥	1	0	1.0	0	552	0	1000
Init/Var2	🟦	1	0	1.0	0	880	0	1000
Init/Var3	⬜	1	0	1.0	0	760	0	1000

Step 8: Save display to file:

When sampling is inactive, you can save particular parts of the diagrams to file.

1. Select the variable which curve needs to be saved. Select multiple variables by pressing the *'Ctrl'* and clicking the variable with the mouse.
2. Select the time range to save.
3. Store the file as a .rec format by clicking the *'Save selected diagrams'* button. The file can be loaded by clicking the *'Restore diagrams'* button.



Symbol	Color	# Diagram	Hysteresis	Gain	Offset	Value	Minimum v...	Maximu
Init/Var1	🟥	1	0	1.0	0		0	1000
Init/Var2	🟦	1	0					
Init/Var3	⬜	1	0					

Time ranges

Refresh

Refresh rate: 500 ms

Time ranges

● Zoom

From: 2020-05-15 5:23:47 18 ms

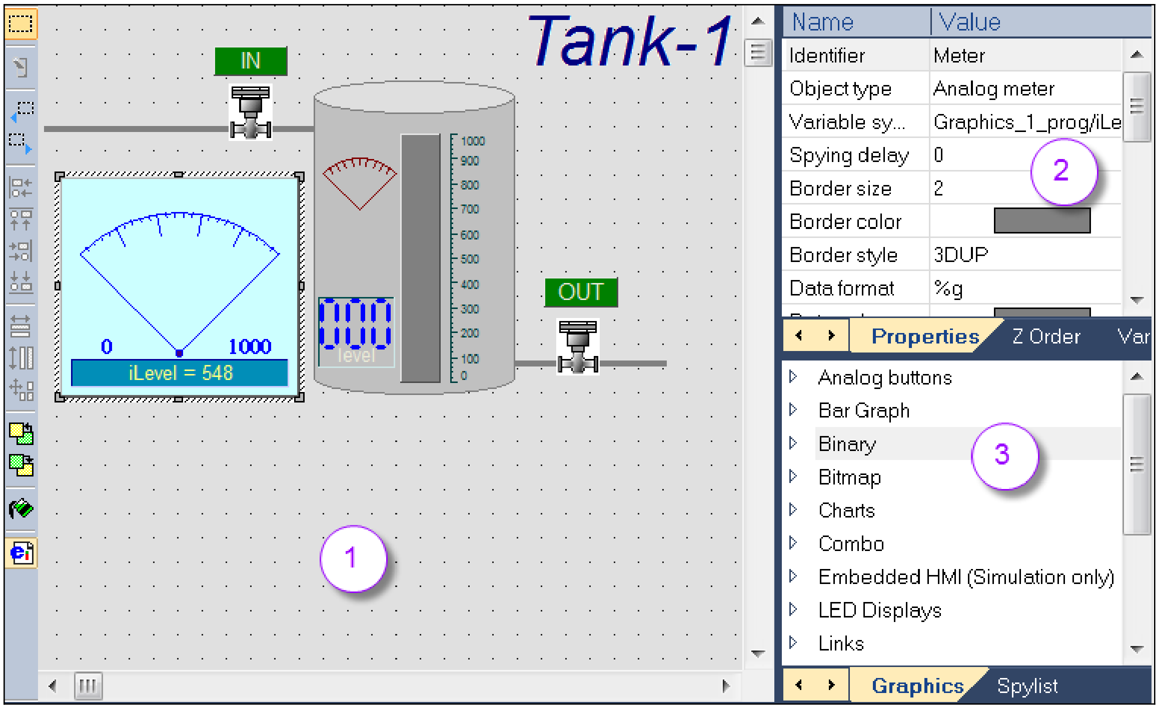
To: 2020-05-15 5:23:51 604 ms

○ Auto scroll

Time range: 12:00:04 585 ms

7.4 Control Panel for Debugging

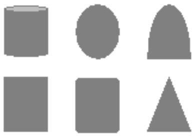
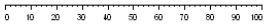
The control panel tool enables you to create graphic user interface for the workbench. When the workbench is connected to the runtime (online mode) the graphic user interface is updated in real time with the current values of the project. Via this interface variables of the PLC application can me directly manipulated by modifying its value or status. The control panel tools assist the user to create a HMI for debugging purpose to operate the PLC application. The HMI can only run together with the workbench or the X5Viewer tool but it can not act as a stand alone HMI. The X5Viewer tool which is installed with the Workbench runs and displays the control panel.

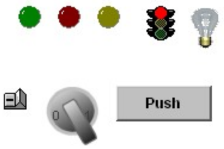
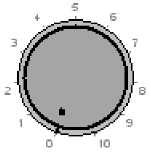
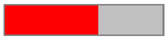
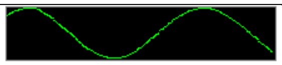
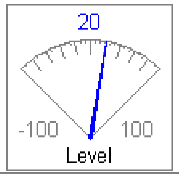
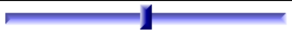
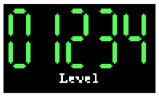
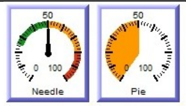


Name	Value
Identifier	Meter
Object type	Analog meter
Variable sy...	Graphics_1_prog/iLe
Spying delay	0
Border size	2
Border color	▆
Border style	3DUP
Data format	%g

Figure 108: Control panel design interface (1- graphic area, 2- graphic object property, 3- available graphic objects)

Table 19 list all the graphic objects available for designing the control panel.

Graphic Objects		Description
Basic shapes		A collection of basic drawings is available. Each object may be either static, or linked to a variable used to enable its visibility (show/hide).
Bitmaps		Bitmap file (BMP, GIF, JPG) can be inserted in the graphic area.
Scales		Scales are static drawings representing a X or Y axis, generally used to document other objects such as trend charts or bargraphs.
Text boxes		Static, animated or edit text boxes are available for displaying / forcing variables. For edit boxes at runtime, double-click on the object to enter the value and then hit ENTER to validate the input.

Graphic Objects		Description
Switches and 2-state displays		Buttons, switches and 2-state displays are used for control or display of a boolean variable.
Analog buttons		Analog buttons are used for setting the value of an integer or real variable. Mouse is used for setting the value.
Bargraphs		Bargraphs are rectangles filled according to the value of an analog variable. Bargraphs can be horizontal or vertical.
Trend charts		Trend charts enable the tracing of a variable as with an oscilloscope.
Analog meters		Analog meters provide a graphical display of an analog value.
Sliders		Sliders are used for entering an analog value with a horirontal or vertical mouse driven cursor.
Digital meters		Digital meters (digits) display the value of a variable with the same aspect as a digital clock.
Links	Back to main page	Links are mouse driven hyperlinks that are used as shortcuts for opening another graphic document. Using links enable the design of multi page animated applications.
Connection status		Connection status is a box actuated with the current status of the connection and the connected runtime application. It is aimed for diagnostic.
Gauge		Analog view meter.

Graphic Objects		Description
	50 0 100 Ring	

Table 19: Graphic objects provided for the control panel

Table 20 list the toolbar commands available for the graphic editor.

Icon	Description
	Set Operate or Edit mode. The Operate button is used to enable/disable changes in the graphic when the workbench is online. When the operate mode is selected, no change can be made. In that mode, the mouse can be used for driving active objects such as buttons.
	Select the previous item in the graphic area.
	Select the next item in the graphic area.
	Align selected items on the left. Select items with CTRL key pressed.
	Align selected items on the top.
	Align selected items on the right.
	Align selected items on the bottom.
	Makes all selected items the same width (*).
	Makes all selected items the same height (*).
	Makes all selected items the same width and height (*).
	Send to front: move the selected item to the top in Z order.
	Send to back: move the selected item to the bottom in Z order.
	Define the background color for the graphic area.
	Export graphics • Export graphic for display in the X5Viewer tool • Export graphic as HMTL5 file (not supported by the runtime)

Table 20: Graphic editor toolbar

The Z-order tab in the property area shows the list of the graphic items sorted according to their Z order. You can simply move objects in that list to change the Z-order and thus arrange overlapping items.

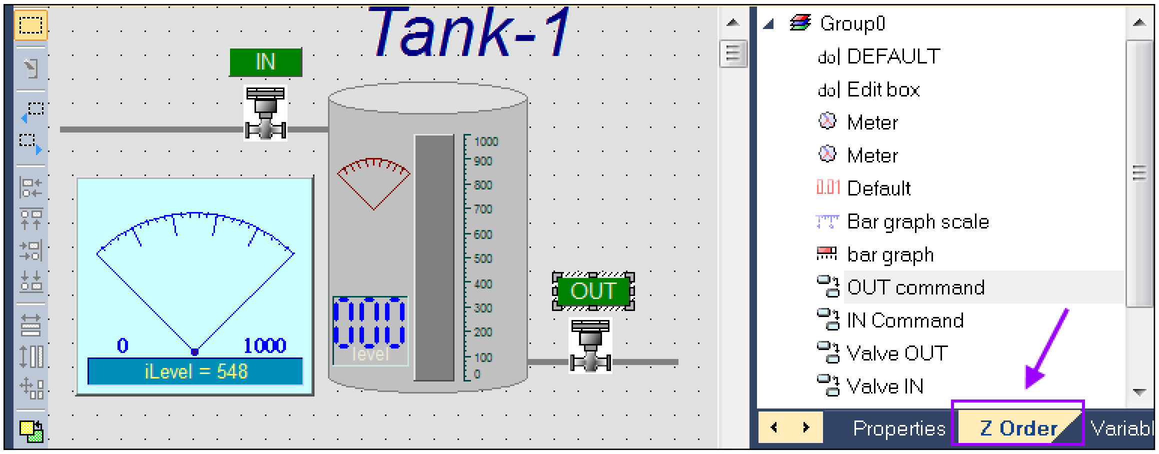
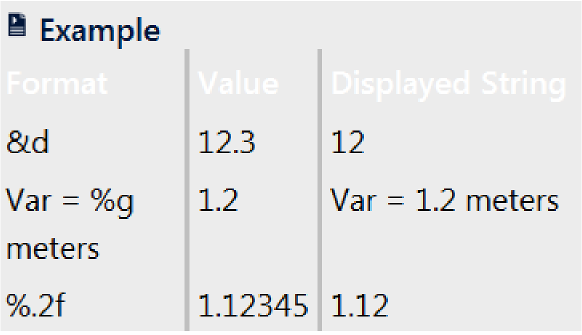


Figure 109: Z-order property

Table 21 details all possible properties for graphic objects.

Graphic Object Properties	Description
Identifier	You can freely attach a text identifier to each graphic object inserted in a document. Identifiers are useful for arranging overlapped objects as they appear in the Z-order list.
Variable symbol	This is the full name of the application variable connected to the graphic object. In case of a local variable, its symbol must be prefixed with the parent program name, separated with '/'. Example: '`MyProg/MyVar`'.
Spying delay	This is the minimum period for actuating the value of the connected variable, expressed as a number of milliseconds. If the delay is not specified or equal to 0, refresh is done as fast as possible.
Border size	This property indicates the width of the border drawn around the object, as a number of pixels. If this property is 0, then no border is drawn.
Border color	This property indicates the color of the border drawn around the object.
Border style	This property indicates the possible 3D effect used for drawing the border around the object. Possible values are: • FLAT = no 3D effect • 3DUP = depressed 3D effect • 3DDOWN = pressed 3D effect • 3D = default 3D effect
Text color	This property indicates the color used for drawing texts in the graphic object.
Text mode	This property indicates the font effect used for drawing texts in the graphic object. Possible values are: • HIDE = text is not displayed • NORMAL = normal font • BOLD = bold text • ITALIC = italic text • UNDERLINE = underlined text
Font name	This property indicates the name of the character font used for drawing texts in the graphic object.
Font size	This property indicates the size of the character font used for drawing

Graphic Object Properties	Description
	texts in the graphic object. The size is expressed as a percentage of the actual height of the object. Maximum possible value is 100. This ensures that the ratio is kept when the object is resized.
Background color	This property indicates the color used for filling the background of the object. In case of a bitmap, it specifies the color that should not be drawn if the TRANS (transparent) background mode is specified.
Background mode	This property indicates whether the background of the object must be filled or not. If this property is OPAQUE, then the background is filled with the specified background color. If this property is TRANS (transparent) then background is not filled. Transparent drawing mode may be useful in case of overlapping objects. **<u>Attention</u>** Specifying the TRANS (transparent) mode for large bitmaps is time consuming and will affect the real time performances of graphic updates.
Data format	If defined, this property indicates that the value of the connected variable must be displayed on the graphic object. You must specify for this property a format string that indicates how the data must be formatted. **<u>Attention</u>** The text property is ignored when a data format is specified. Format string has the same format as the famous *'printf'* function of *'C'* language. It may include static characters together with one of the following possible pragmas that specify the value: • %s = default formatting according to IEC syntax • %d = integer (decimal) • %X = hexadecimal • %g = floating point • %.nf = decimal real (n is the number of displayed decimal digits)  Example Format / Value / Displayed String &d / 12.3 / 12 Var = %g meters / 1.2 / Var = 1.2 meters %.2f / 1.12345 / 1.12 **<u>Info</u>** Only one % pragma can be used in a string.
Text	If defined, this property indicates the text to be displayed on the graphic object. **<u>Attention</u>** This property is ignored when a data format is specified.

Graphic Object Properties	Description
Bitmap display mode	For bitmap based objects, this property indicates whether the attached bitmap must keep its original aspect or be stretched to the actual size of the object. Possible values are: • ORIGINAL = Keep the original aspect of the bitmap (cut if too large). • STRETCH = Stretch or shrink the bitmap for fitting the actual size of the graphic object. **Attention** Large bitmaps with STRETCH display mode are time consuming during animation and can lead to poor performances.
Minimum value	For analog animated objects (meters, bargraphs, trends...) this property indicates the minimum possible value that can be displayed. For static scales, it indicates the value of the lowest mark.
Maximum value	For analog animated objects (meters, bargraphs, trends...) this property indicates the maximum possible value that can be displayed. For static scales, it indicates the value of the highest mark.
Data color	This property indicates the color used to represent the value of connected variable within the object (for example the filled part of a bargraph).
Nb divisions (main)	For objects including a graphic scale, this property indicates the number of main division marks to be drawn in the scale.
Nb divisions (small)	For objects including a graphic scale, this property indicates the number of small division marks to be drawn in the scale, between each main division mark.
Scale color	For objects including a graphic scale, this property indicates the color used for drawing the axis, the division marks and corresponding values of the scale.
Bitmap pathname	For bitmaps, this property specifies the pathname of the bitmap to be displayed. BMP, GIF and JPG formats are supported. If no directory is specified, the specified file name is searched: • In the project folder. • In the '\BITMAP' folder of the workbench.
Bitmap for *'TRUE'* state	For 2-state objects having the CUSTOM aspect, this property specifies the pathname of the bitmap to be displayed when the value of the attached variable is TRUE (or not zero for analogs). BMP, GIF and JPG formats are supported. If no directory is specified, the specified file name is searched: • In the project folder. • In the '\BITMAP' folder of the workbench.
Bitmap for *'FALSE'* state	For 2-state objects having the CUSTOM aspect, this property specifies the pathname of the bitmap to be displayed when the value of the attached variable is FALSE (or zero for analogs). BMP, GIF and JPG formats are supported. If no directory is specified, the specified file name is searched: • In the project folder. • In the '\BITMAP' folder of the workbench.
Color when not connected	For shapes, this property indicates the color used for filling shapes when no variable is attached to the graphic object.
TRUE color	For shapes, this property indicates the color used for filling shapes when the attached variable has the TRUE state, or non zero for analogs.
FALSE color	For shapes, this property indicates the color used for filling shapes when

Graphic Object Properties	Description
	the attached variable has the FALSE state, or zero for analogs.
Direction (basic shapes)	For oriented shapes such as triangles, half ellipses or cylender, this property indicates the direction of the drawing; to the left, to the right, to the top or to the bottom.
Direction (scale)	For scales, this property indicates the direction of the axis. If LEFT, the minimum value is on the left side. If RIGHT, the minimum value is on the right side.
Placement (scale)	For scales, this property indicates the location of the scale within the object rectangle: on the left, on the right, on the top or at the bottom.
Action (text)	Indicates the possible mouse action for text boxes. Following values are possible: - STATIC = No mouse action. - EDIT = Double-click opens an edit box for entering the variable value.
Action (switch)	Indicates the possible mouse action for switches. Following values are possible: • STATIC = No mouse action. • PUSHBUTTON = The variable is forced to TRUE when pressed and to FALSE when depressed. • SWITCH = The status of the variable is inverted when the button is pressed (toggle). • ONESHOTBUTTON = Same as switch, but the display remains depressed when the mouse is released.
Direction (bargraph)	For bargraphs, this property indicates the growing direction: to the left, to the right, to the top or to the bottom.
Nb of points (trends)	For trend charts, this property indicates the maximum number of stored points. If the width of the object (in pixels) is less than this number, then oldest points are not visible.
Direction (slider)	For slider, this property indicates whether the slider is horizontal (RIGHT) or vertical (TOP).
Link	This property indicates the name of the target .GRA animated document for shortcuts. If no directory is specified in the link, then the file is searched in the project folder.
Aspect (shapes)	This property indicates the type of basic shape to be drawn. Possible aspects are: • CYLINDER = A 3D like cylinder. • ELLIPSE = An ellipse. • HALFELLPISE = One half of an ellipse. • GATE = A simple vector drawing for a valve. • RECTANGLE = A rectangle. • ROUNDRECT = A rectangle with rounded corners. • TRIANGLE = A triangle.
Aspect (switches)	This property indicates the type of switch to be drawn. Possible aspects are: • DEFAULT = A standard Windows-like push button. • CUSTOM = A button with TRUE and FALSE drawings defined with bitmaps.
Aspect (trend charts)	This property indicates the type of drawing for a trend chart. Possible aspects are: • POINT = Only relevant dots are drawn. • LINE = Lines are drawn from point to point.

Graphic Object Properties	Description
	• HISTO = Histogram style.
Aspect (digits)	This property indicates the type of drawing for a digital meter. Possible aspects are: • DEFAULT = Plain drawing. • BEZEL = All segments have a 3D effect.

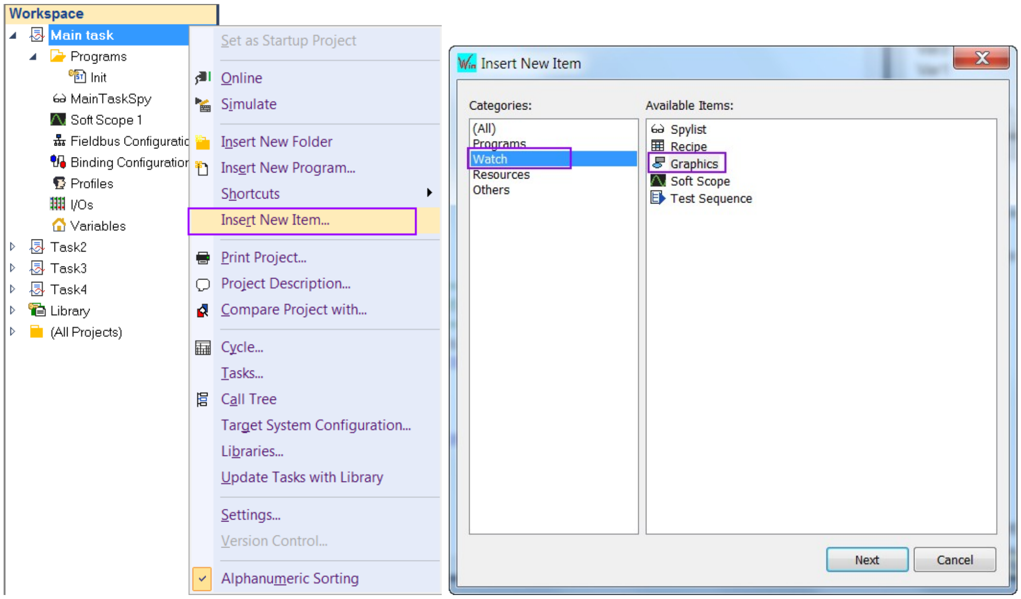
Table 21: Graphic object properties

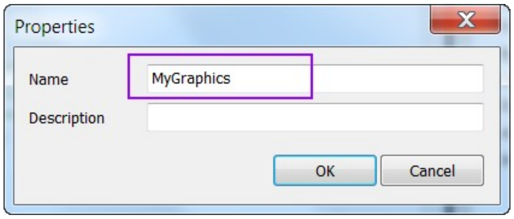
7.4.1 Create Control Panel

This section describes the procedure of creating a control panel on the workbench for debugging the PLC application in online mode.

Step 1: Add control panel:

1. Right click a task in the workspace tree and select *'Insert New Item...'* from the pop menu.
2. Select the *'Graphics'* item and click *'Next'*.
3. Assign the control panel a name and click *'OK'*

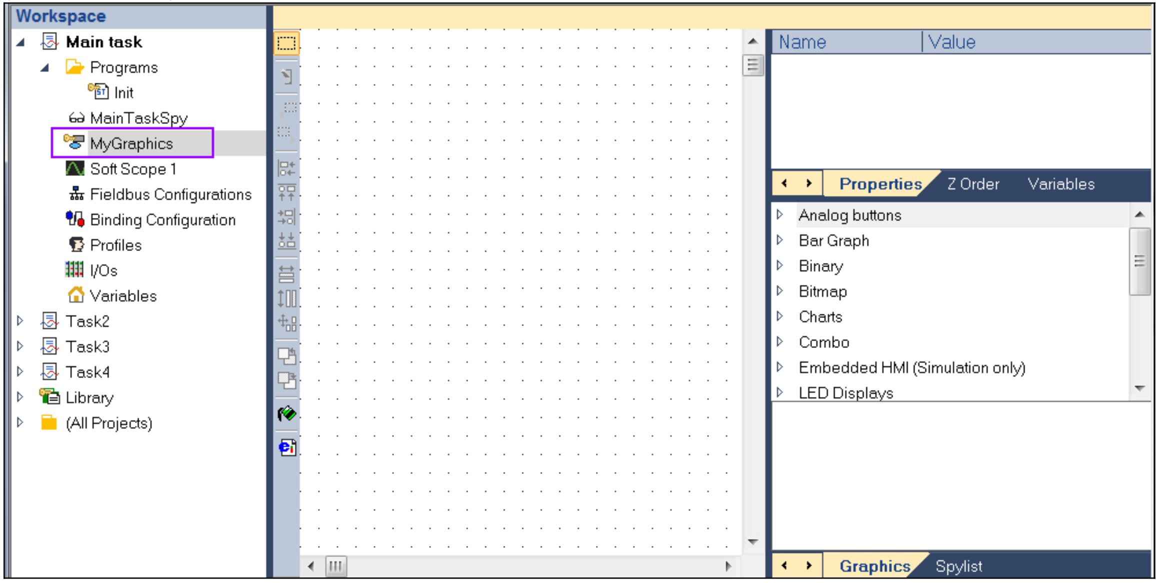




Name: MyGraphics

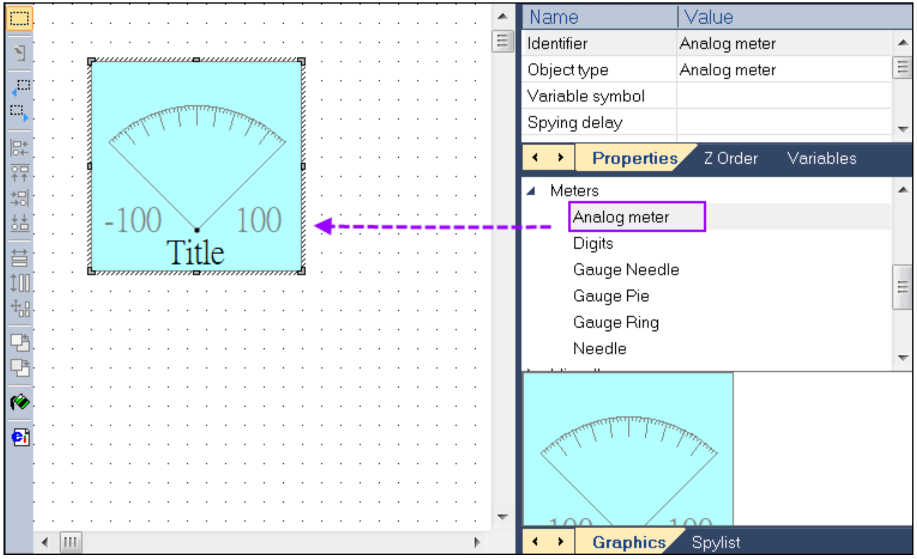
Description:

Step 2: Open the control panel graphic area by double clicking the newly added control panel name in the workspace.
The Graphics window on the right hand of the graphic area list all the graphic objects with its properties supported by the workbench. Insert a graphic object to the graphic area by simple drag it over the area and drop it where you want to place it.



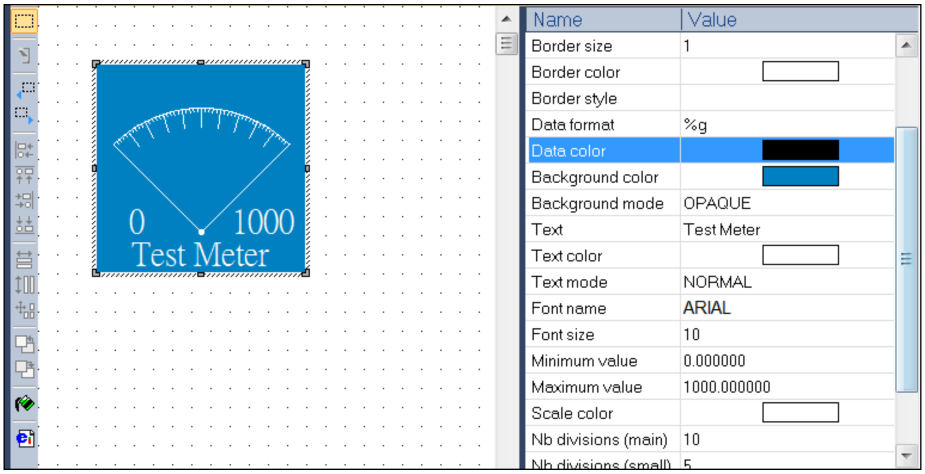
In the following it is shown how to add a meter object to the graphic area and link it to a PLC variable.

Step 3: Drag the analog meter graphic object to the graphic area:



Name	Value
Identifier	Analog meter
Object type	Analog meter
Variable symbol	
Spying delay	

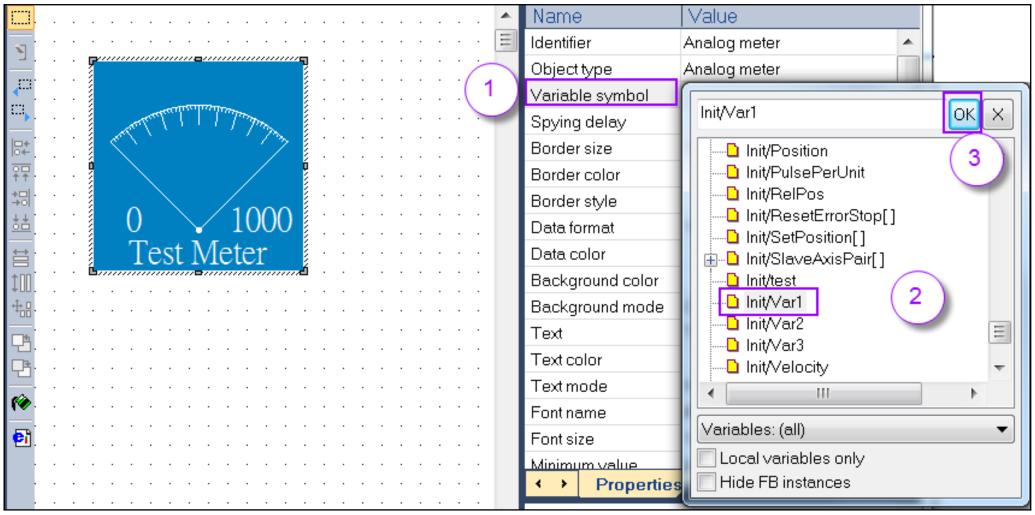
Step 4: Set the property of the graphic object (e.g. color, range, name, front style, etc.)



Name	Value
Border size	1
Border color	□
Border style	
Data format	%g
Data color	■
Background color	🟦
Background mode	OPAQUE
Text	Test Meter
Text color	□
Text mode	NORMAL
Font name	ARIAL
Font size	10
Minimum value	0.000000
Maximum value	1000.000000
Scale color	□
Nb divisions (main)	10
Nb divisions (small)	5

Step 5: Link the graphic object to a variable which value is has to represent

1. Double click the *'Variable symbol'* property
2. Select a variable from the variable list. The value of this variable will be shown by the graphic object
3. Click *'OK'* to confirm the selection



Name	Value
Identifier	Analog meter
Object type	Analog meter
Variable symbol	
Spying delay	
Border size	
Border color	
Border style	
Data format	
Data color	
Background color	
Background mode	
Text	
Text color	
Text mode	
Font name	
Font size	
Minimum value	

Init/Var1

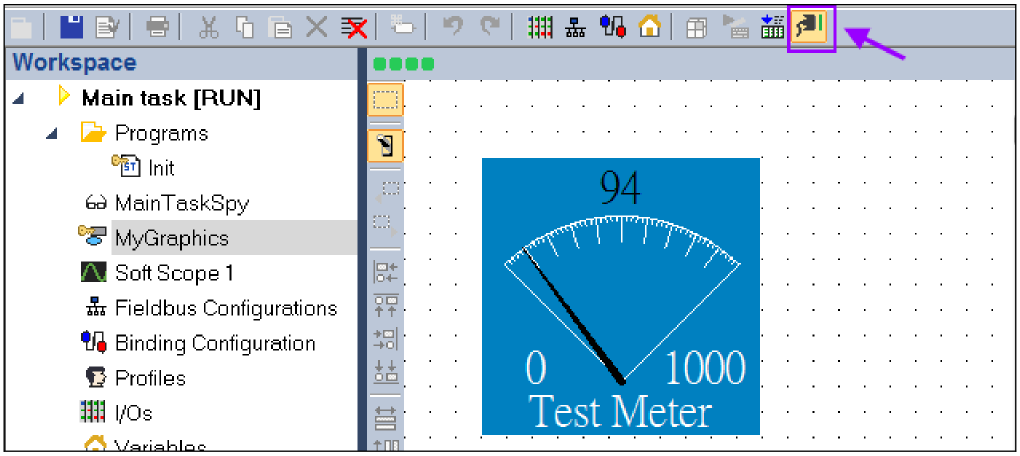
- Init/Position
- Init/PulsePerUnit
- Init/RelPos
- Init/ResetErrorStop[ ]
- Init/SetPosition[ ]
- Init/SlaveAxisPair[ ]
- Init/test
- Init/Var1
- Init/Var2
- Init/Var3
- Init/Velocity

Variables: (all)

□ Local variables only

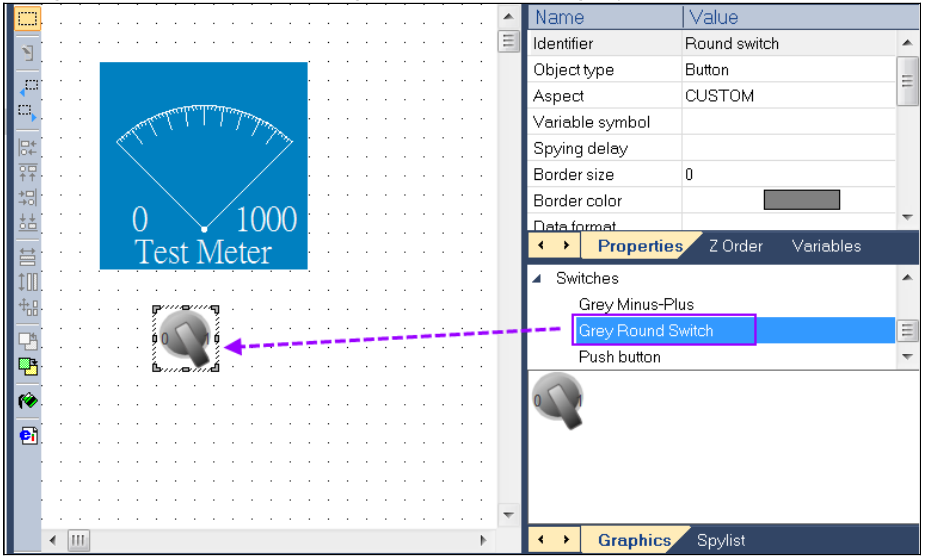
□ Hide FB instances

Step 6: Connect the workbench to the runtime. The graphic object displays the current value to which it has been linked.

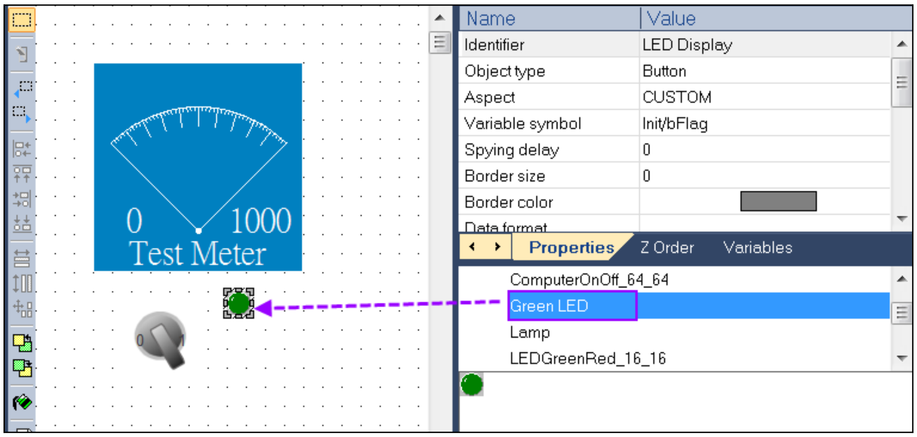


In the following steps a switch is added which serves as an example to show how to modify a PLC variable via the control panel.

Step 7: Drag a round switch and a green LED to the graphic interface.



Name	Value
Identifier	Round switch
Object type	Button
Aspect	CUSTOM
Variable symbol	
Spying delay	
Border size	0
Border color	
Data format	

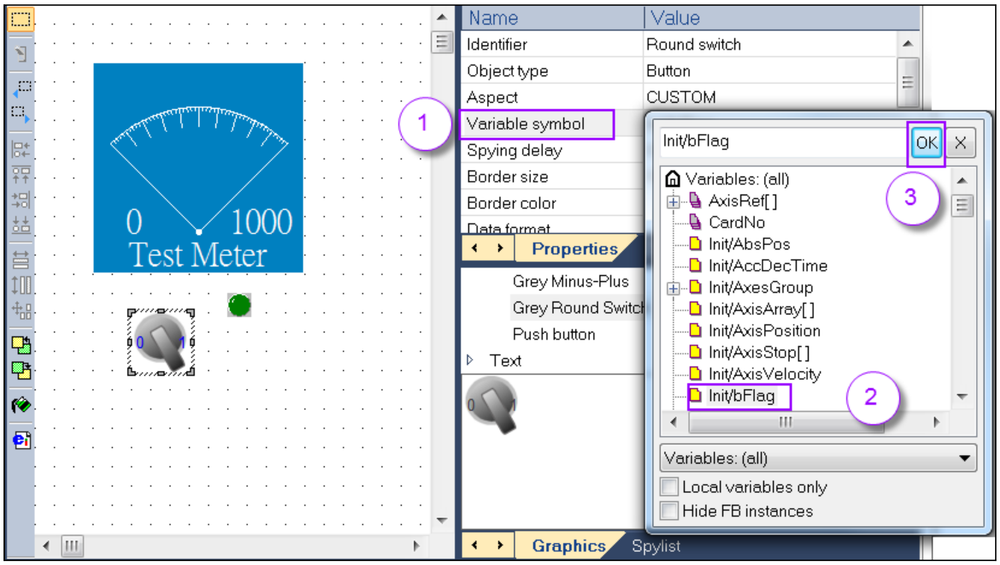


Name	Value
Identifier	LED Display
Object type	Button
Aspect	CUSTOM
Variable symbol	Init/bFlag
Spying delay	0
Border size	0
Border color	
Data format	

Step 8: Set the properties of the round switch and the green LED.

Step 9: Map both the round switch and the green LED to the same BOOL variable

1. Double click the *'Variable symbol'*
2. Select a BOOL variable from the variable list
3. Click *'OK'*



Name	Value
Identifier	Round switch
Object type	Button
Aspect	CUSTOM
Variable symbol	
Spying delay	
Border size	
Border color	
Data format	

Init/bFlag

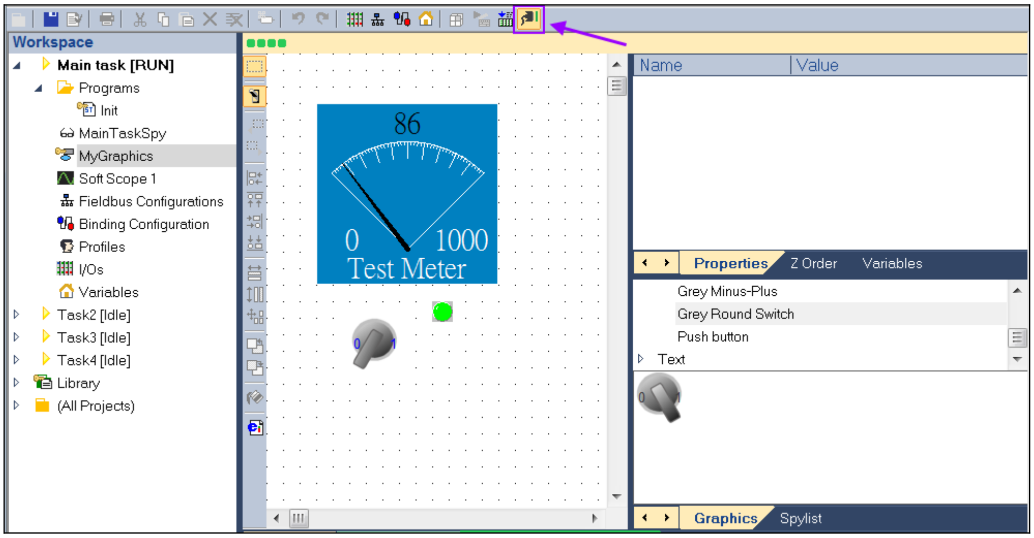
Variables: (all)
- AxisRef[ ]
- CardNo
- Init/AbsPos
- Init/AccDecTime
- Init/AxesGroup
- Init/AxisArray[ ]
- Init/AxisPosition
- Init/AxisStop[ ]
- Init/AxisVelocity
- Init/bFlag

Variables: (all)

□ Local variables only

□ Hide FB instances

Step 10: Set the workbench in online mode. By turning the switch from 0 to 1 the linked BOOL variable is forced from false to true or vice versa.



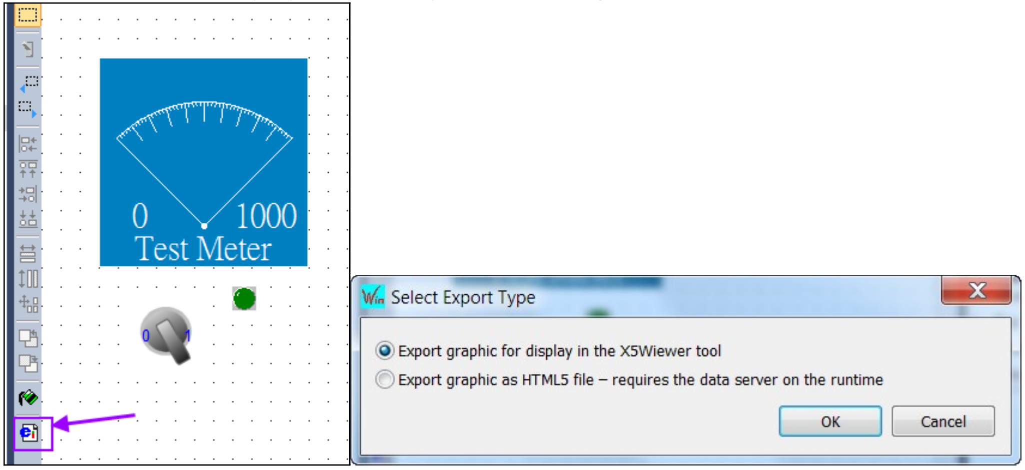
7.4.2 Exporting Control Panel to X5Viewer

The control panel developed with the workbench can also be opened and run with X5Viewer which is part of the workbench package. It is necessary to export the control panel as a .X5T file before it can be displayed by the X5Viewer tool.

The procedure of exporting the control panel and running it by X5Viewer are explained in the following steps:

Step 1: Export the graphic display

1. Click the *'Export graphic'* command in the toolbar
2. Select *'Export graphic for display in the X5Viewer tool'* option. After clicking *'OK'* a wizard window pops up.
3. Follow the steps provided by the wizard guide

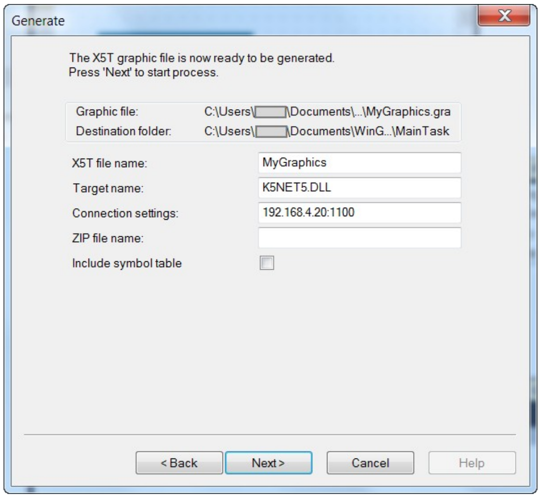


● Export graphic for display in the X5Wiewer tool

○ Export graphic as HTML5 file – requires the data server on the runtime

Step 2: Follow the steps provided by the wizard guide

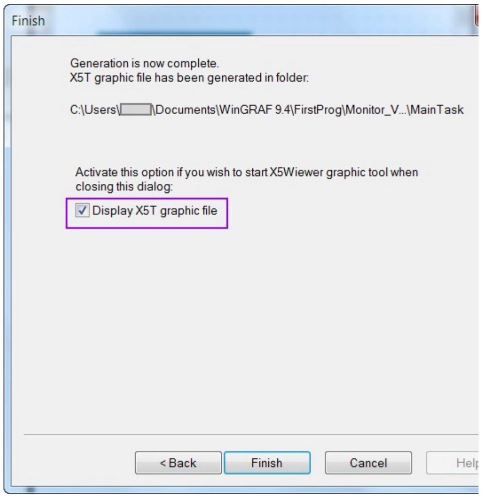
1. Select the folder where to save the .X5T file
2. Set the name of the X5T file
3. Set driver used for communication with the runtime. Specify `K5NET5.DLL` for the standard runtime.
4. Set communication parameters for connecting to the runtime. The IP address has to be followed by colons (:) and the IP port number. If not specified, the default port number used is 1100.
5. ZIP file name: This field has to be empty
6. Include symbol table: Do not check this option
7. Check the *'Display X5T graphic file'* option



The X5T graphic file is now ready to be generated.
Press 'Next' to start process.

Graphic file: C:\Users\[redacted]\Documents\...\MyGraphics.gra
Destination folder: C:\Users\[redacted]\Documents\WinG...\MainTask

X5T file name: MyGraphics
Target name: K5NET5.DLL
Connection settings: 192.168.4.20:1100
ZIP file name:
Include symbol table ☐



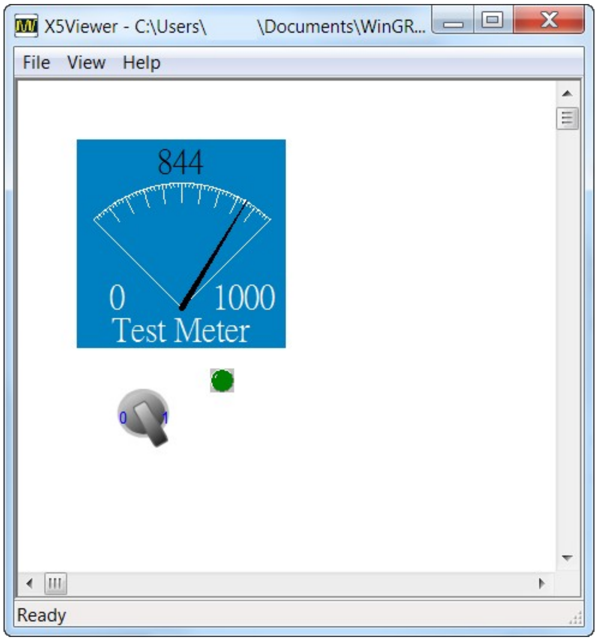
Generation is now complete.
X5T graphic file has been generated in folder:

C:\Users\[redacted]\Documents\WinGRAF 9.4\FirstProg\Monitor_V...\MainTask

Activate this option if you wish to start X5Wiewer graphic tool when closing this dialog:

☑ Display X5T graphic file

Step 3: After clicking *'Finish'* the X5Viewer with the control panel starts. The X5Viewer automatically connects to the runtime IP address set in the previous step and continuously updates the control objects with the linked variable value. Via X5Viewer variables can be forced to new values, e.g. by changing the state of a switch.



The X5Viewer execution file location is in the following directory:
`C:\Program Files (x86)\Win-GRAF Workbench\Win-GRAF Wb 9.xx`

The control panel can be directly open by the X5Viewer by clicking *'File/Open from file...'* and selecting a xxx.X5T file.

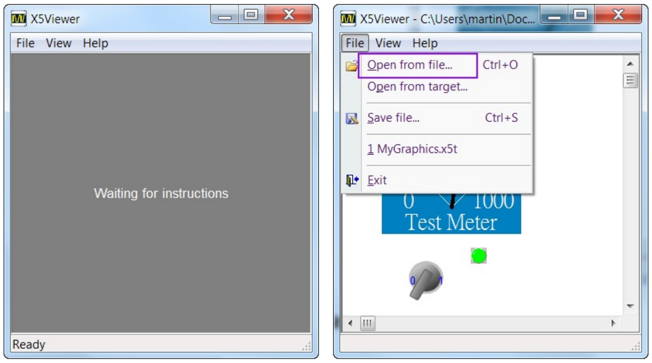
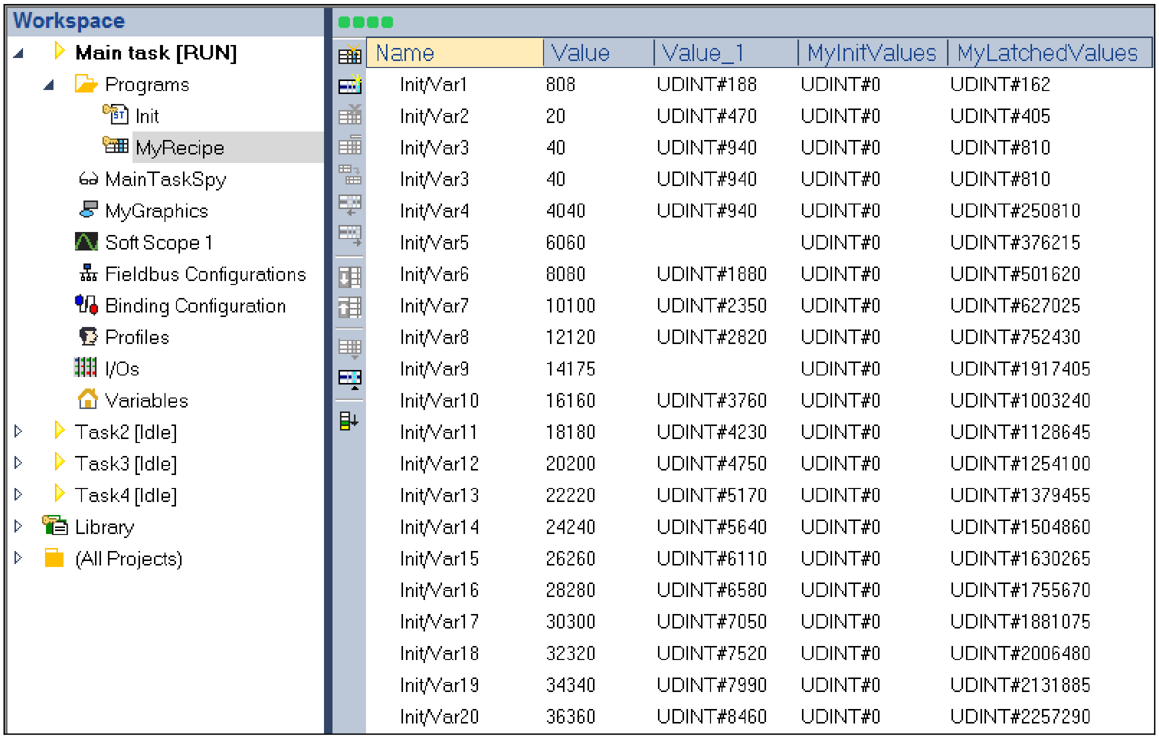


Figure 110: Open a X5T file in X5Viewer

7.5 Recipe Control

Similar to the variable editor or SpyList the recipe control enable the user monitor the values of a list of variables. In contrast to the variable editor and SpyList the recipe control has a couple of advantages:

- The values of more than one variables can be force synchronously in one cycle time. The SpyList only allows one variable value change at a time while the recipe control supports up to 50 variable value change per command by one mouse click. This ensures that all variables are written together at the same moment in the runtime, i.e. in between two cycles.
- Values of multiple variables can be saved (latched) with one command
- Several columns can be created which stores multiple variable values. The runtime can be forced to replace all current values with the values stored in one a column



Name	Value	Value_1	MyInitValues	MyLatchedValues
Init/Var1	808	UDINT#188	UDINT#0	UDINT#162
Init/Var2	20	UDINT#470	UDINT#0	UDINT#405
Init/Var3	40	UDINT#940	UDINT#0	UDINT#810
Init/Var3	40	UDINT#940	UDINT#0	UDINT#810
Init/Var4	4040	UDINT#940	UDINT#0	UDINT#250810
Init/Var5	6060		UDINT#0	UDINT#376215
Init/Var6	8080	UDINT#1880	UDINT#0	UDINT#501620
Init/Var7	10100	UDINT#2350	UDINT#0	UDINT#627025
Init/Var8	12120	UDINT#2820	UDINT#0	UDINT#752430
Init/Var9	14175		UDINT#0	UDINT#1917405
Init/Var10	16160	UDINT#3760	UDINT#0	UDINT#1003240
Init/Var11	18180	UDINT#4230	UDINT#0	UDINT#1128645
Init/Var12	20200	UDINT#4750	UDINT#0	UDINT#1254100
Init/Var13	22220	UDINT#5170	UDINT#0	UDINT#1379455
Init/Var14	24240	UDINT#5640	UDINT#0	UDINT#1504860
Init/Var15	26260	UDINT#6110	UDINT#0	UDINT#1630265
Init/Var16	28280	UDINT#6580	UDINT#0	UDINT#1755670
Init/Var17	30300	UDINT#7050	UDINT#0	UDINT#1881075
Init/Var18	32320	UDINT#7520	UDINT#0	UDINT#2006480
Init/Var19	34340	UDINT#7990	UDINT#0	UDINT#2131885
Init/Var20	36360	UDINT#8460	UDINT#0	UDINT#2257290

Figure 111: Recipe control

Several toolbar commands are available to add, delete, copy, move columns (Table 22).

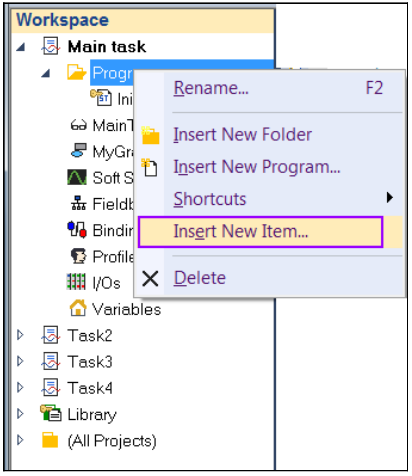
Icon	Description
	Insert a new column to the recipe table
	Add a new column behind the last column
	Remove selected column
	Rename column: Change the header name of selected column
	Copy column: copy values of selected column to another column
	Move selected column to the left
	Move selected column to the right
	Move variable one row down
	Move variable one row up
	Send recipe: sends the values of the current selected column to the runtime to replace the all the current variables values (force variable value change)
	Save recipe: copies the current variable values of the *'Value'* column to a new column.
	Sort variables in the recipe table in alphabetical order

Table 22: Recipe editor toolbar commands

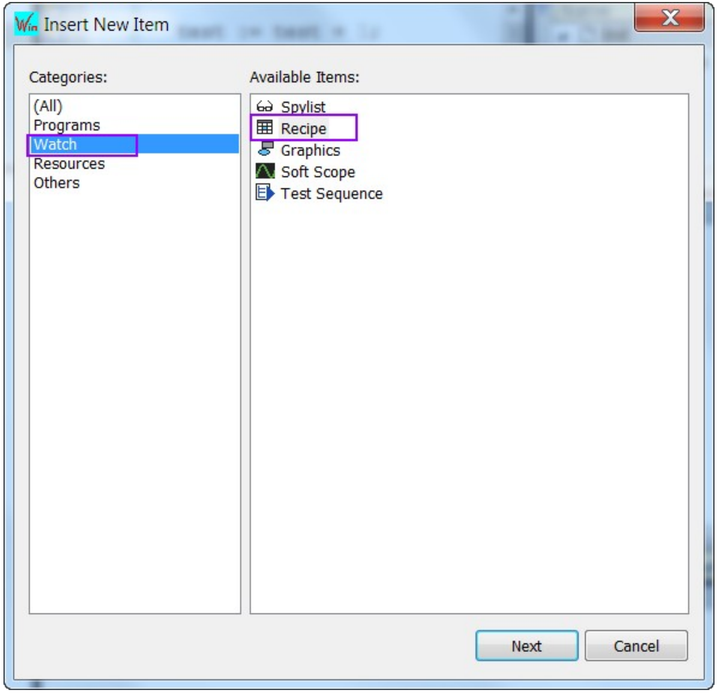
Procedure to add a new recipe control to the workbench:

Step 1: Right click a task or program item in the workspace and select *'Insert New*

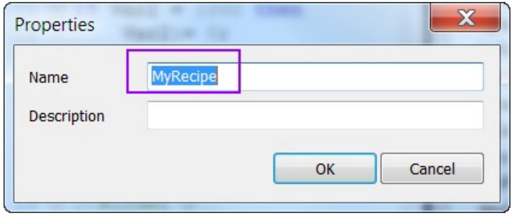
Item...' from the popup menu.



Step 2: Select *'Recipe'* and click *'Next'*



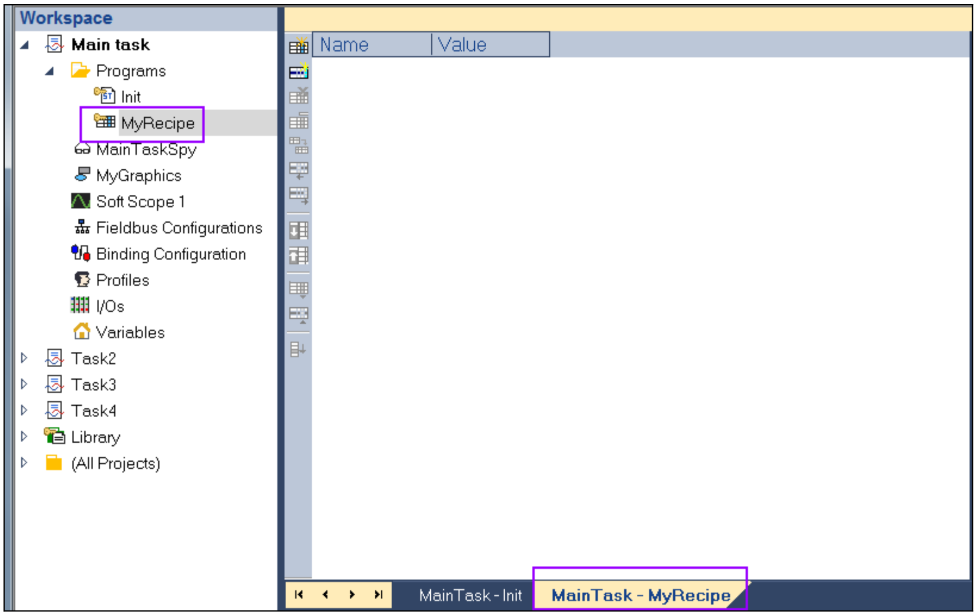
Step 3: Enter a recipe name



Name: MyRecipe

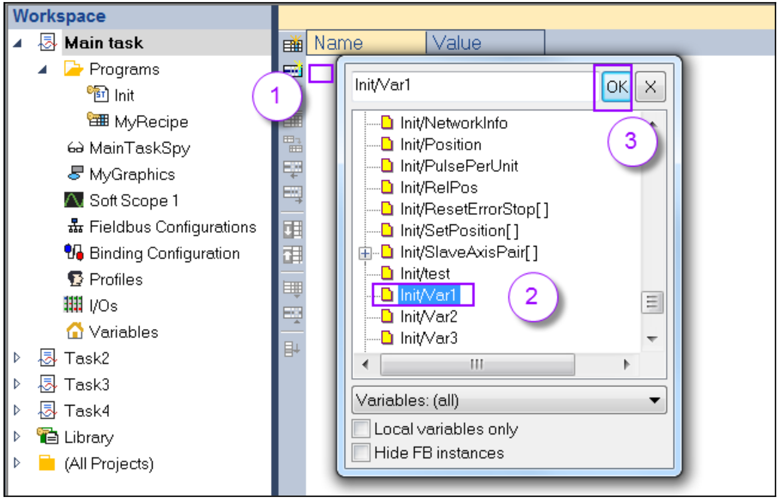
Description:

Step 4: Open the recipe editor by double clicking its name in the workspace



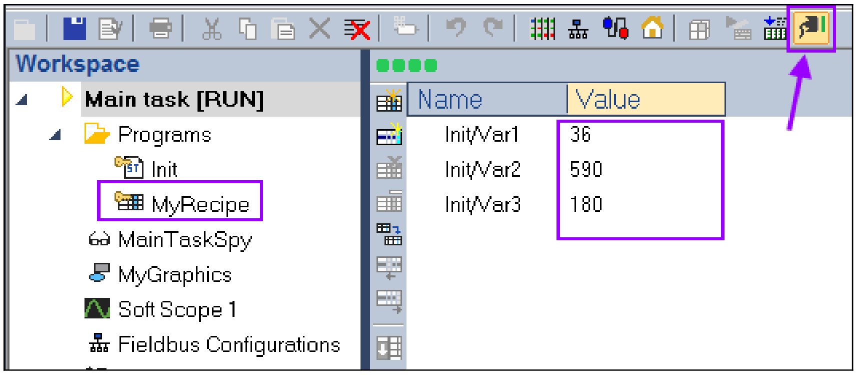
Step 5: Add variables to the recipe editor:

1. Double click the white field in the recipe editor. A variable list will pop up.
2. Select a variable from the list
3. Click *'OK'* and the variable name will appear in the first column of the recipe editor.



Repeat this step to add several variable to list which can be monitored.

Step 6: Connect the workbench to the runtime. Now the variable values are continuously being updated with the current values of the runtime.



Name	Value
Init/Var1	36
Init/Var2	590
Init/Var3	180

Application examples:

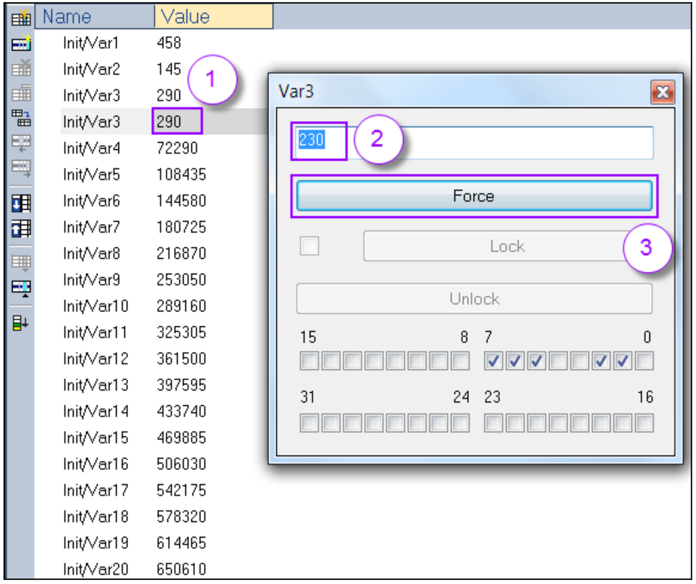
1. Forcing one variable value

   In online mode the *'Value'* column is continuously updated with the current variable values.

   Step 1: Double click a value in the *'Value'* column to open a windows which allows you to force / control the value of the selected variable. You can also first select the a value in the *'Value'* column and press the ENTER key to open the force variable window

   Step 2: Enter a new value for the variable and

   Step 3: Click *'Force'* button to replace the current value with the new value.



Name	Value
Init/Var1	458
Init/Var2	145
Init/Var3	290
Init/Var3	290
Init/Var4	72290
Init/Var5	108435
Init/Var6	144580
Init/Var7	180725
Init/Var8	216870
Init/Var9	253050
Init/Var10	289160
Init/Var11	325305
Init/Var12	361500
Init/Var13	397595
Init/Var14	433740
Init/Var15	469885
Init/Var16	506030
Init/Var17	542175
Init/Var18	578320
Init/Var19	614465
Init/Var20	650610

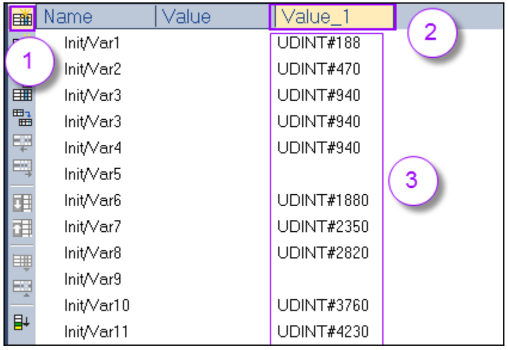
2. Forcing a column of variable values

   In the recipe editor a new column can be created and be filled with new values. With one force command all the current variable values are replaced by the new column values at the same time between two cycles. A maximum of 50 variables (or less if strings) can be sent at the same time.

   Step 1: Define a new column add fill it with new values

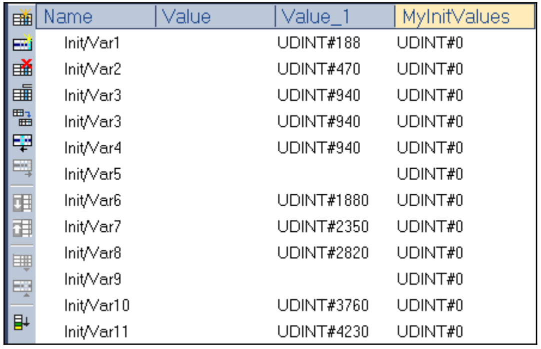
   - Click *'Insert Column'* button to add a new column to the recipe editor

- Enter a name for the column header (e.g. *'Value_1'*)
- Enter new values for each variable in the new column. If no value is assigned for a variable (e.g. Var5, Var9) then the current value for this variable in the runtime is not being replaced if the new column value is forced to replace the current values.



Name	Value	Value_1
Init/Var1		UDINT#188
Init/Var2		UDINT#470
Init/Var3		UDINT#940
Init/Var3		UDINT#940
Init/Var4		UDINT#940
Init/Var5		
Init/Var6		UDINT#1880
Init/Var7		UDINT#2350
Init/Var8		UDINT#2820
Init/Var9		
Init/Var10		UDINT#3760
Init/Var11		UDINT#4230

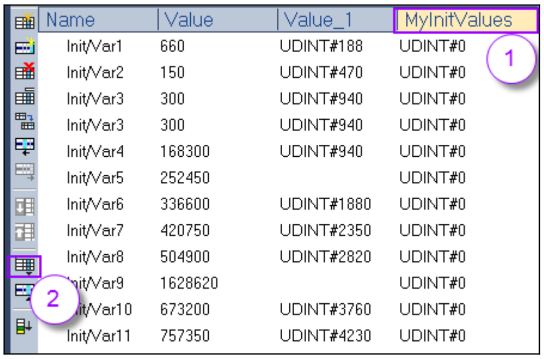
Several new columns can be created. In the figure below two columns (*'Value_1'*, *'MyInitValues'*) has been added and filled with values.



Name	Value	Value_1	MyInitValues
Init/Var1		UDINT#188	UDINT#0
Init/Var2		UDINT#470	UDINT#0
Init/Var3		UDINT#940	UDINT#0
Init/Var3		UDINT#940	UDINT#0
Init/Var4		UDINT#940	UDINT#0
Init/Var5			UDINT#0
Init/Var6		UDINT#1880	UDINT#0
Init/Var7		UDINT#2350	UDINT#0
Init/Var8		UDINT#2820	UDINT#0
Init/Var9			UDINT#0
Init/Var10		UDINT#3760	UDINT#0
Init/Var11		UDINT#4230	UDINT#0

Step 2: Set the workbench in online mode

- Select a column (e.g. *'MyInitValues'*) which variable values will be forced by the workbench to replace the values in the runtime
- Click the *'Send Recipe'* button to force a variable value replacement. All the variable values are replaced at the same time between two cycles. In the next cycle the *'Value'* column shows the replaced values.

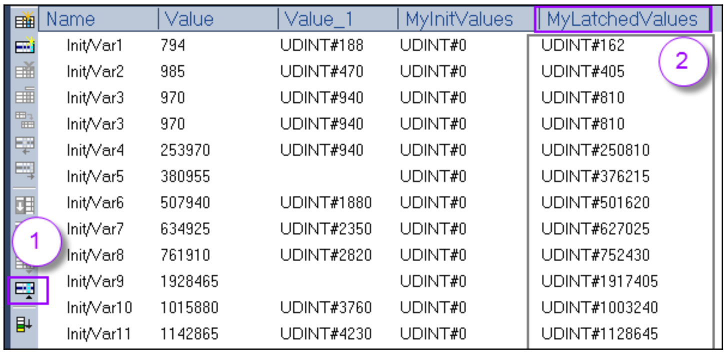


Name	Value	Value_1	MyInitValues
Init/Var1	660	UDINT#188	UDINT#0
Init/Var2	150	UDINT#470	UDINT#0
Init/Var3	300	UDINT#940	UDINT#0
Init/Var3	300	UDINT#940	UDINT#0
Init/Var4	168300	UDINT#940	UDINT#0
Init/Var5	252450		UDINT#0
Init/Var6	336600	UDINT#1880	UDINT#0
Init/Var7	420750	UDINT#2350	UDINT#0
Init/Var8	504900	UDINT#2820	UDINT#0
Init/Var9	1628620		UDINT#0
Init/Var10	673200	UDINT#3760	UDINT#0
Init/Var11	757350	UDINT#4230	UDINT#0

3. Latch all variable values in a column
   In online mode the values in the *'Value'* column are continuously being updated with the current values in the runtime. All the values listed in the *'Value'* can be latched at the same time with one command and saved to a new column.

   Step 1: Latch column values

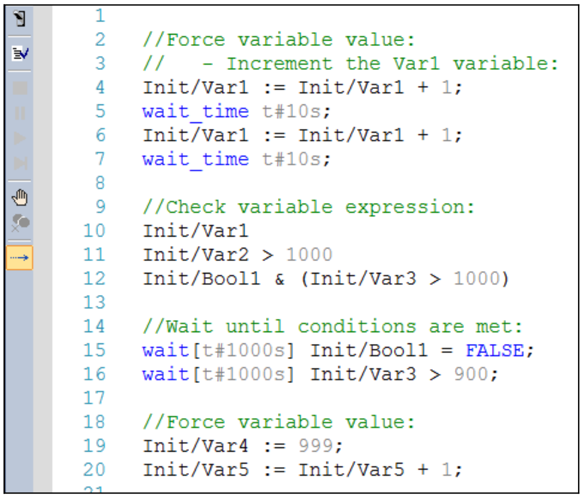
   - Click the *'Save Recipe'* button. A new column is being created and filled with the latched values (grey reactangle)
   - Enter a name for the new column



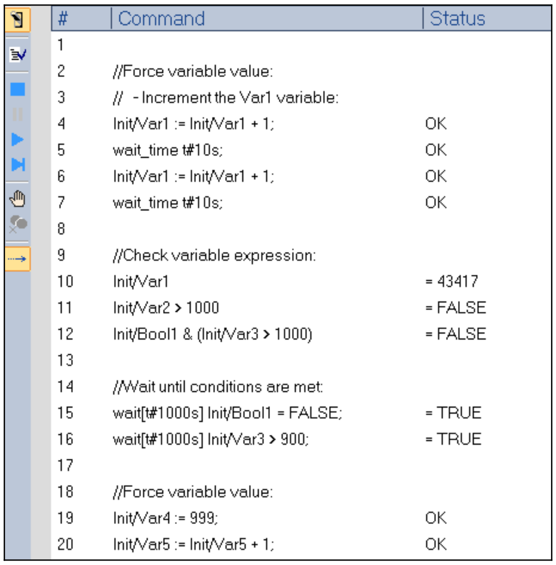
Name	Value	Value_1	MyInitValues	MyLatchedValues
Init/Var1	794	UDINT#188	UDINT#0	UDINT#162
Init/Var2	985	UDINT#470	UDINT#0	UDINT#405
Init/Var3	970	UDINT#940	UDINT#0	UDINT#810
Init/Var3	970	UDINT#940	UDINT#0	UDINT#810
Init/Var4	253970	UDINT#940	UDINT#0	UDINT#250810
Init/Var5	380955		UDINT#0	UDINT#376215
Init/Var6	507940	UDINT#1880	UDINT#0	UDINT#501620
Init/Var7	634925	UDINT#2350	UDINT#0	UDINT#627025
Init/Var8	761910	UDINT#2820	UDINT#0	UDINT#752430
Init/Var9	1928465		UDINT#0	UDINT#1917405
Init/Var10	1015880	UDINT#3760	UDINT#0	UDINT#1003240
Init/Var11	1142865	UDINT#4230	UDINT#0	UDINT#1128645

7.6 Test Sequences

The Workbench includes an integrated tool for designing and running automated test scenarios. Test scenarios are expression evaluations and forcing variable values. Test sequence does not need to be command and can be edited while the workbench is in online mode.



```

//Force variable value:
//   - Increment the Var1 variable:
Init/Var1 := Init/Var1 + 1;
wait_time t#10s;
Init/Var1 := Init/Var1 + 1;
wait_time t#10s;

//Check variable expression:
Init/Var1
Init/Var2 > 1000
Init/Bool1 & (Init/Var3 > 1000)

//Wait until conditions are met:
wait[t#1000s] Init/Bool1 = FALSE;
wait[t#1000s] Init/Var3 > 900;

//Force variable value:
Init/Var4 := 999;
Init/Var5 := Init/Var5 + 1;
```



#	Command	Status
1		
2	//Force variable value:	
3	// - Increment the Var1 variable:	
4	Init/Var1 := Init/Var1 + 1;	OK
5	wait_time t#10s;	OK
6	Init/Var1 := Init/Var1 + 1;	OK
7	wait_time t#10s;	OK
8		
9	//Check variable expression:	
10	Init/Var1	= 43417
11	Init/Var2 > 1000	= FALSE
12	Init/Bool1 & (Init/Var3 > 1000)	= FALSE
13		
14	//Wait until conditions are met:	
15	wait[t#1000s] Init/Bool1 = FALSE;	= TRUE
16	wait[t#1000s] Init/Var3 > 900;	= TRUE
17		
18	//Force variable value:	
19	Init/Var4 := 999;	OK
20	Init/Var5 := Init/Var5 + 1;	OK

Figure 112: Test sequence syntax (left) and execution result (right)

Syntax

The following syntaxes are allowed in a text sequence:

1. Comments

   Empty and comment lines are allowed. Comments may have the following syntaxes:

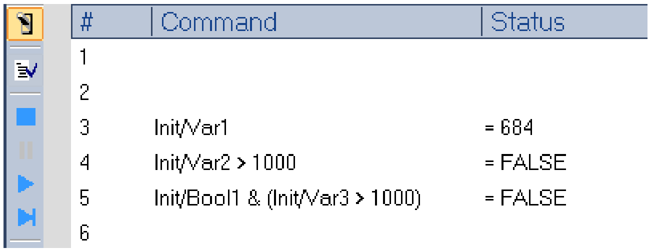
```
// comment up to the end of line
(* delimited comment *)
```

2. Evaluation of an expression

   You can evaluate a complex expression using ST operators. Function calls are not allowed within an expression.

   Examples:

```
Var1
Var2 > 1000
Bool1 & (Var3 > 1000)
```



#	Command	Status
1		
2		
3	Init/Var1	= 684
4	Init/Var2 > 1000	= FALSE
5	Init/Bool1 & (Init/Var3 > 1000)	= FALSE
6		

If an expression is entered alone on a line of a sequence, its value is simply displayed in the *'Status'* column when run.

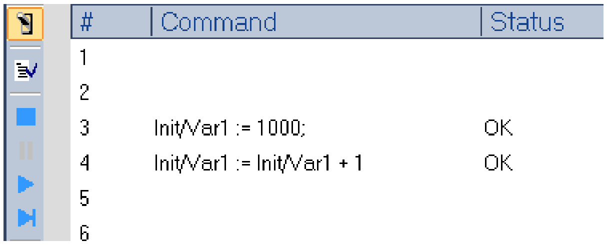
3. Forcing a variable

   A line may contain a statement to force a variable, using the syntax:

```
variable_name := expression;
```

   Examples:

```
Var1 := 1000;
Var1 := Var1 + 1;
```



#	Command	Status
1		
2		
3	Init/Var1 := 1000;	OK
4	Init/Var1 := Init/Var1 + 1	OK
5		
6		

The status column indicates whether forcing a variable was successful.

4. Waiting for an expression to be TRUE

Use the following syntax to wait for an expression to be TRUE:

```
wait expression;
```

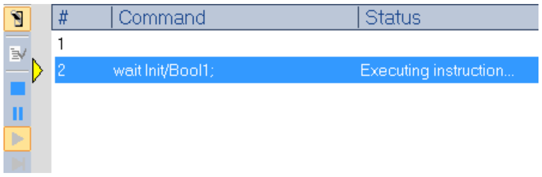
The default wait timeout is 10 seconds. A timeout can be specified in between brackets after the *'wait'* keyword:

```
wait [timeout] expression;
```

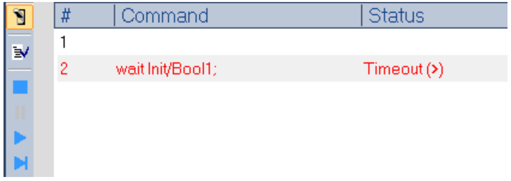
Examples:

```
wait Bool1;
```

While wait command is executing the status column displays *'Executing instruction'*. If the default wait time of 10 seconds has elapsed before the expression turned TRUE then a *'Timeout(>)'* is shown in the status column.



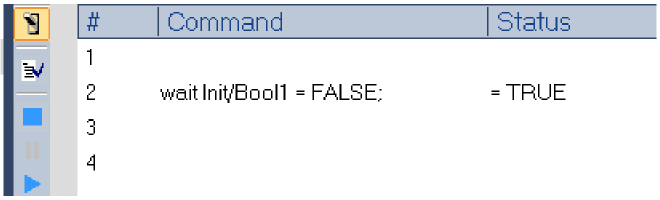
#	Command	Status
1		
2	wait Init/Bool1;	Executing instruction...



#	Command	Status
1		
2	wait Init/Bool1;	Timeout (>)

```
wait Bool1 == FALSE;
```

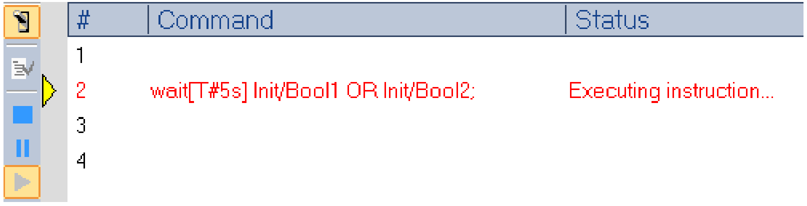
Once the expression turns TRUE the wait command ends and the status column shows TRUE.



#	Command	Status
1		
2	wait Init/Bool1 = FALSE;	= TRUE
3		
4		

```
wait [t#5s] Bool1 OR Bool2;
```

Add a wait time of 5 seconds to the wait command. If the expression does not turn TRUE within the wait time a timeout status is being shown.



#	Command	Status
1		
2	wait[T#5s] Init/Bool1 OR Init/Bool2;	Executing instruction...
3		
4		

5. Execution delay

The delay statement holds the execution of the test sequence for the set time. Delay statement syntax:

```
wait_time time_value;
```

Examples:

```
wait_time t#2s;
```

Delay the execution of the next command by 10 seconds:

```
Var1 := Var1 + 1;
wait_time t#10s;
Var1 := Var1 + 1;
```

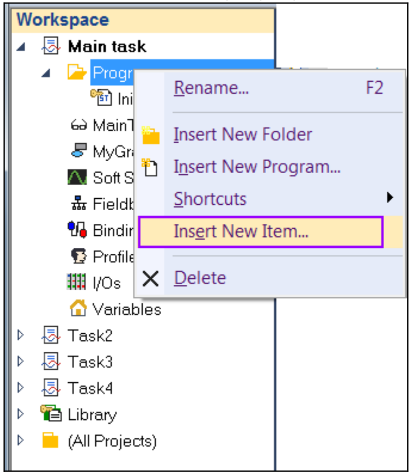
Use the editor toolbar to check and run your sequence (Table 23):

Icon	Description
▮	Use this button to activate or deactivate the sequence. The sequence must be deactivated for editing. When activated, the sequence can be run and tested.
✓	Check the syntax of the sequence.
■	Abort the sequence when running.
⏸	Pause the sequence when running.
▶	Start the execution of the sequence.
⏭	Single step: execute the selected line
✋	Set/remove a breakpoint on the current line.
✗	Remove all breakpoints.

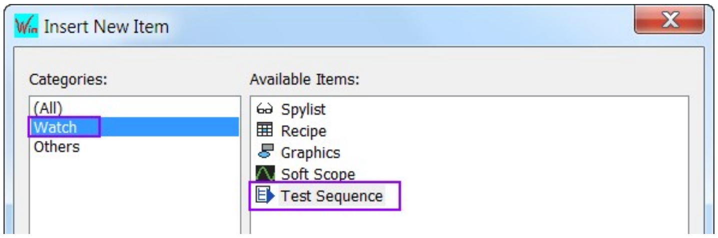
Table 23: Sequence editor toolbar commands

Steps for adding a test sequence:

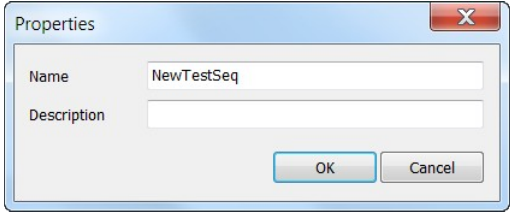
Step 1: Right click a task or program item in the workspace and select *'Insert New Item...'* from the popup menu.



Step 2: Select *'Test Sequence'* and click *'Next'*.



Enter a name for the test sequence.

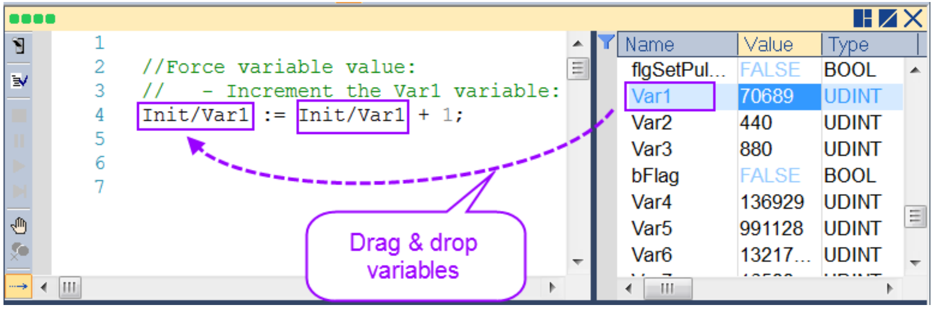


Name: NewTestSeq

Description:

Step 3: Edit test sequence commands:

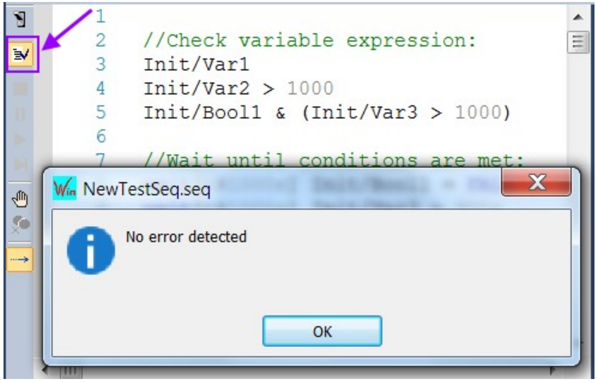
1. Open the sequence editor by double clicking its name in the workspace tree
2. Drag and drop the variable which needs to be evaluated or forced a new value from the variable editor to the sequence editor



```

//Force variable value:
//    - Increment the Var1 variable:
Init/Var1 := Init/Var1 + 1;
```

Name	Value	Type
flgSetPul...	FALSE	BOOL
Var1	70689	UDINT
Var2	440	UDINT
Var3	880	UDINT
bFlag	FALSE	BOOL
Var4	136929	UDINT
Var5	991128	UDINT
Var6	13217...	UDINT

Step 4: After the test sequence has been edited check for any syntax error by clicking the *'Check Test Sequence'* button.



```

//Check variable expression:
Init/Var1
Init/Var2 > 1000
Init/Bool1 & (Init/Var3 > 1000)

//Wait until conditions are met:
```

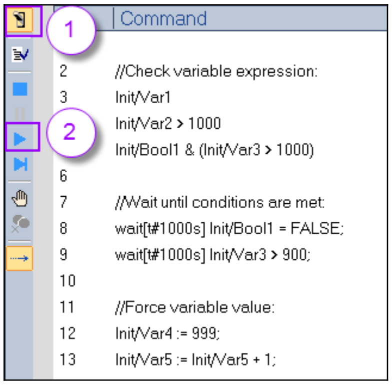
NewTestSeq.seq

No error detected

Step 5: Run the test sequence

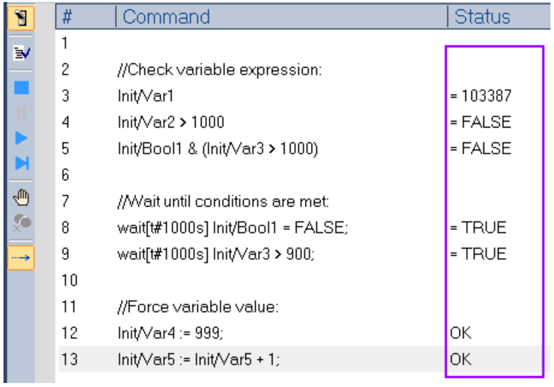
1. Make sure the workbench is online
2. Activate the test sequence. Click *'Activate/Deactivate'* button

3. Click *'Run Test Sequence'* button to execute the test



```
//Check variable expression:
Init/Var1
Init/Var2 > 1000
Init/Bool1 & (Init/Var3 > 1000)

//Wait until conditions are met:
wait[t#1000s] Init/Bool1 = FALSE;
wait[t#1000s] Init/Var3 > 900;

//Force variable value:
Init/Var4 := 999;
Init/Var5 := Init/Var5 + 1;
```

The result of the test sequence execution is shown in the status column.



#	Command	Status
1		
2	//Check variable expression:	
3	Init/Var1	= 103387
4	Init/Var2 > 1000	= FALSE
5	Init/Bool1 & (Init/Var3 > 1000)	= FALSE
6		
7	//Wait until conditions are met:	
8	wait[t#1000s] Init/Bool1 = FALSE;	= TRUE
9	wait[t#1000s] Init/Var3 > 900;	= TRUE
10		
11	//Force variable value:	
12	Init/Var4 := 999;	OK
13	Init/Var5 := Init/Var5 + 1;	OK

Note:

1. To run the test sequence the PLC program does not need to be recompiled.
2. The test sequence does not need to be compiled.
3. The test sequence can be edited while the workbench is online the and the PLC application is running.

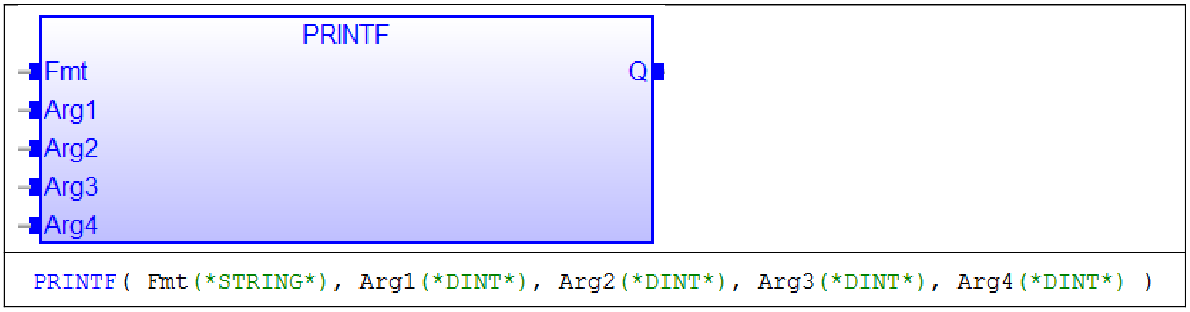
7.7 Debug Message (PRINTF)

The PRINTF function outputs a trace statement from the runtime to the workbench. It is used for debugging to trace data of the PLC application and has to be embedded in the PLC code. PRINTF give you the opportunity to print out information during application execution and allows you to trace a particular path of execution and watching the order of execution to ensure that everything is happening when you expect it to occur.

This function works similar to the *'printf'* function of the *'C'* language and supports up to four integer (DINT) arguments. You can use the following pragmas in the FMT trace message to represent the arguments according to their left to the right order:

- %ld signed value in decimal
- %lu unsigned value in decimal
- %lx value in hexadecimal

This function can be used in debug and release mode.



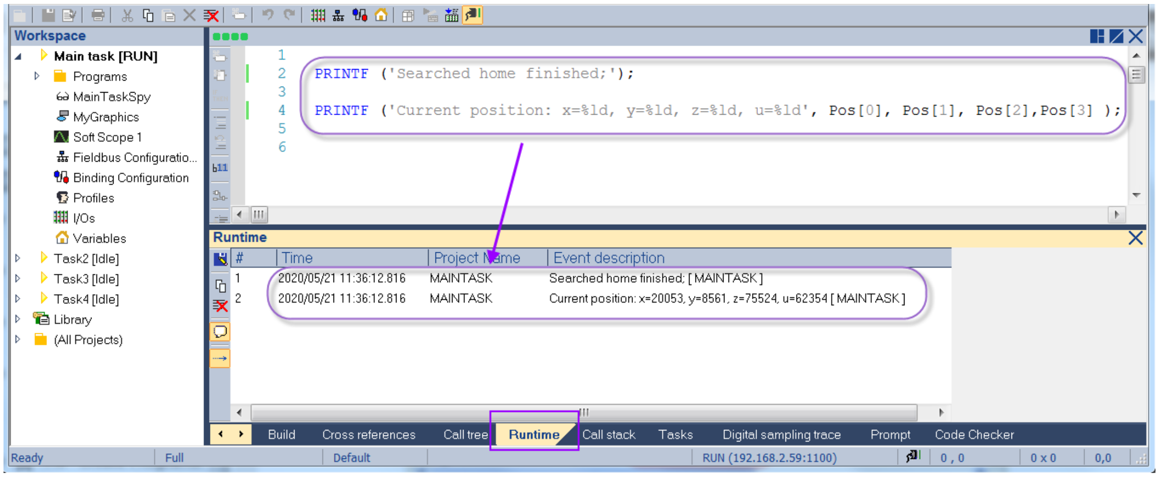
```
PRINTF( Fmt(*STRING*), Arg1(*DINT*), Arg2(*DINT*), Arg3(*DINT*), Arg4(*DINT*) )
```

Figure 113: PRINTF function

Variable		Data Type	Description
Input	Fmt	STRING	Trace message.
	Arg1	DINT	Numerical arguments to be included in the trace.
	Arg2	DINT	Numerical arguments to be included in the trace.
	Arg3	DINT	Numerical arguments to be included in the trace.
	Arg4	DINT	Numerical arguments to be included in the trace.
Output	Q	BOOL	Return check.

Table 24: PRINTF function input and output variables

The trace message is displayed in the *'Runtime'* window of the workbench (Figure 114). PRINTF is supported by the simulator. Trace messages shown in the *'Runtime'* log window can be saved to file.

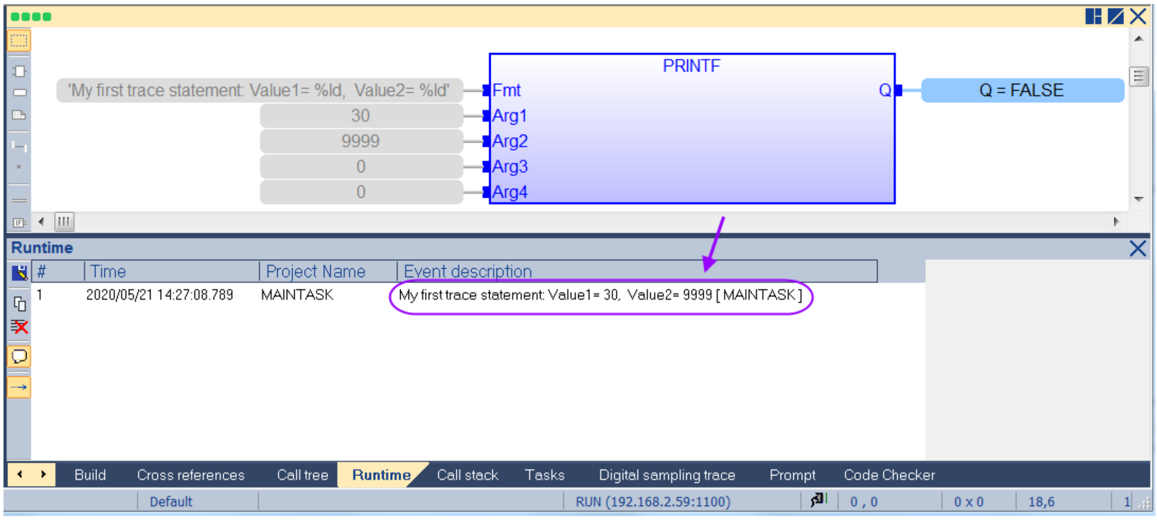


```
PRINTF ('Searched home finished;');

PRINTF ('Current position: x=%ld, y=%ld, z=%ld, u=%ld', Pos[0], Pos[1], Pos[2],Pos[3] );
```

Runtime

```
#  Time                     Project Name  Event description
1  2020/05/21 11:36:12.816  MAINTASK      Searched home finished; [ MAINTASK ]
2  2020/05/21 11:36:12.816  MAINTASK      Current position: x=20053, y=8561, z=75524, u=62354 [ MAINTASK ]
```

Figure 114: PRINTF output window using Structured Text



Runtime

```
#  Time                     Project Name  Event description
1  2020/05/21 14:27:08.789  MAINTASK      My first trace statement: Value1= 30, Value2= 9999 [ MAINTASK ]
```

Figure 115: PRINTF function call in Function Block Diagram

PRINTF function is supported by both the Debug and Release mode compiler. Conditional statement has to be added to the source code if the PRINTF function should only output trace statements when the application is compiled in debug mode. The condition '*__DEBUG'* is automatically defined by the workbench when the application is compiled in debug mode. Add the __DEBUG conditional statement in your code to prevent the PRINTF function from outputting messages in release mode.

```
#ifdef  __DEBUG
PRINTF ('Searched home finished;');
PRINTF ('Current position: x=%ld, y=%ld', Pos[0], Pos[1]);
#endif
```

Figure 116: Structure Text - Conditional compile statement

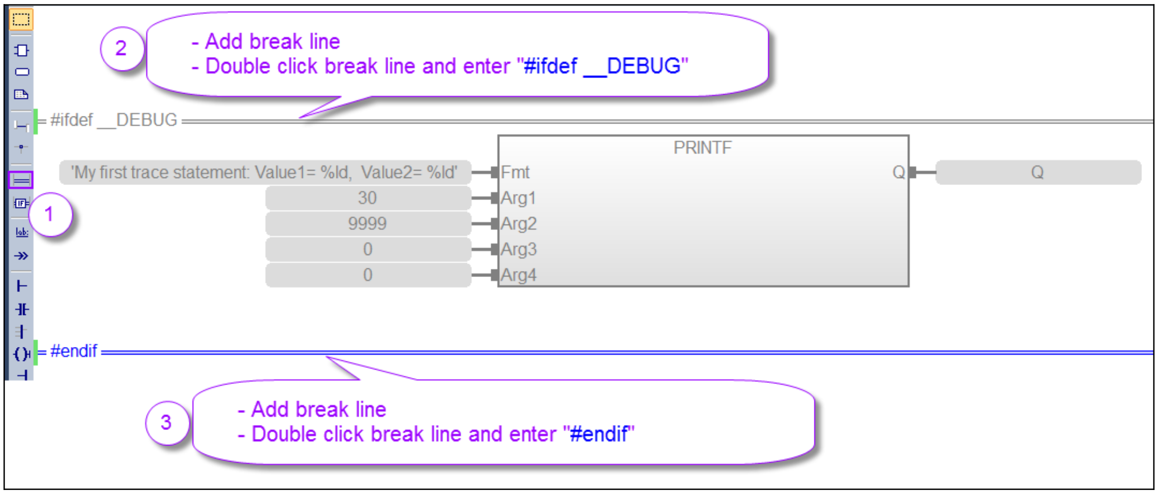

2
- Add break line
- Double click break line and enter "#ifdef __DEBUG"
#ifdef __DEBUG
PRINTF
'My first trace statement: Value1= %ld, Value2= %ld'
Fmt
30
Arg1
9999
Arg2
0
Arg3
0
Arg4
Q
Q
1
#endif
3
- Add break line
- Double click break line and enter "#endif"

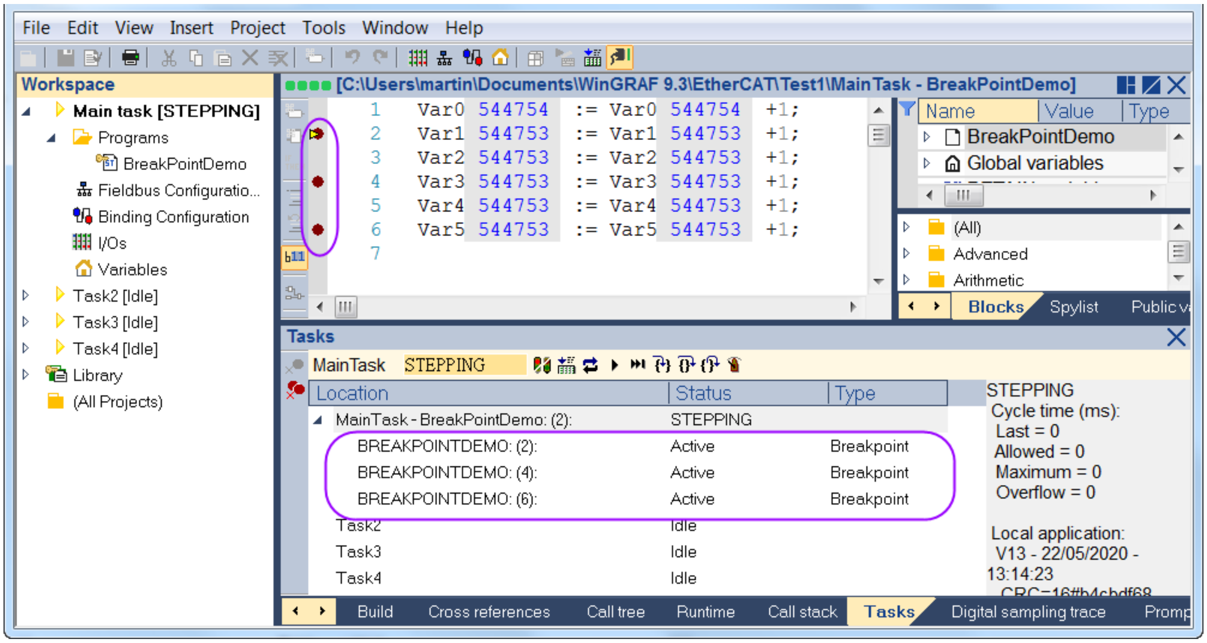
Figure 117: Function Block Diagram - Conditional compile statement

7.8 Breakpoints - Step by Step Debugging

The step by step debugging feature is enabled by setting one or more breakpoints in the source code of the application (Figure 118). Breakpoints are a marker that is set in the source code which, when reached, stops the code's execution at the location.

Step by step debugging is available:

- In ST and IL text programs (a step is a statement).
- In LD program (a step is a rung).
- In FBD (a step is a graphic symbol corresponding to an action).



[C:\Users\martin\Documents\WinGRAF 9.3\EtherCAT\Test1\MainTask - BreakPointDemo]

```
Var0 544754 := Var0 544754 +1;
Var1 544753 := Var1 544753 +1;
Var2 544753 := Var2 544753 +1;
Var3 544753 := Var3 544753 +1;
Var4 544753 := Var4 544753 +1;
Var5 544753 := Var5 544753 +1;
```

Tasks

MainTask STEPPING

Location	Status	Type
MainTask - BreakPointDemo: (2):	STEPPING	
BREAKPOINTDEMO: (2):	Active	Breakpoint
BREAKPOINTDEMO: (4):	Active	Breakpoint
BREAKPOINTDEMO: (6):	Active	Breakpoint
Task2	Idle	
Task3	Idle	
Task4	Idle	

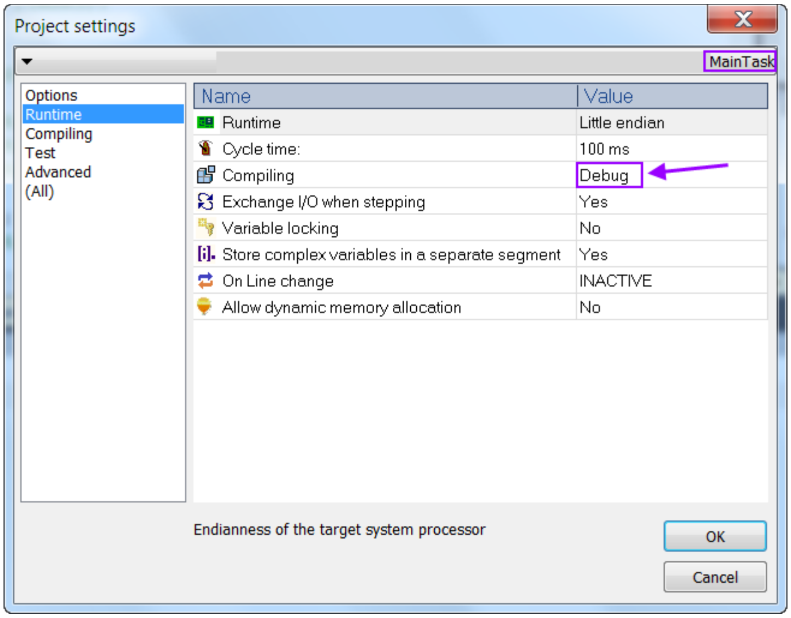
STEPPING
Cycle time (ms):
Last = 0
Allowed = 0
Maximum = 0
Overflow = 0

Local application:
V13 - 22/05/2020 - 13:14:23
CRC=16#b4cbdf68

Figure 118: Program with debug breakpoints

Attention

- Step by step debugging is available only if the project has been compiled with the debug option (Figure 119). This option can be selected from the project settings dialog box ('*Project\Settings...*'). Make sure that the compiler for the task which needs to be debugged is set into debug mode. The compiler for other task can be left in release mode.
- An application compiled in debug mode includes additional information for stepping. This leads to bigger code size and less performances. It is recommended to compile your application in release mode when the debugging is finished.



Name	Value
Runtime	Little endian
Cycle time:	100 ms
Compiling	Debug
Exchange I/O when stepping	Yes
Variable locking	No
Store complex variables in a separate segment	Yes
On Line change	INACTIVE
Allow dynamic memory allocation	No

Endianness of the target system processor

Figure 119: Set compiler in debug mode

When the program has reached a breakpoint you can execute the program in single steps. At each halt position you see the current value of the variables in the monitoring views.
When the breakpoint is reached, the execution stops at the specified location and you can step further in the program. A yellow arrow next to the breakpoint (➡) indicates at which breakpoint the execution has stopped.
Breakpoints are shown as a brown dot (dark or light) in the left frame of the program editor. Breakpoints are active (●) when the dot is dark brown and inactive (○) when the dot is light brown. Breakpoints are inactive if the workbench is not in online mode (connected to runtime) , the target application is not running or the version of running application and the source code is not identical.

7.8.1 Add a Breakpoint

To add or remove a breakpoint, click the line in the source code where to add/remove a breakpoint then right click and select *'Set/Remove Breakpoint'* from the popup menu (Figure 120) or press F9 key. Breakpoints are added to the left frame of the program editor. If the current position is not on a valid line for stepping, the breakpoint is automatically moved to the nearest valid position.

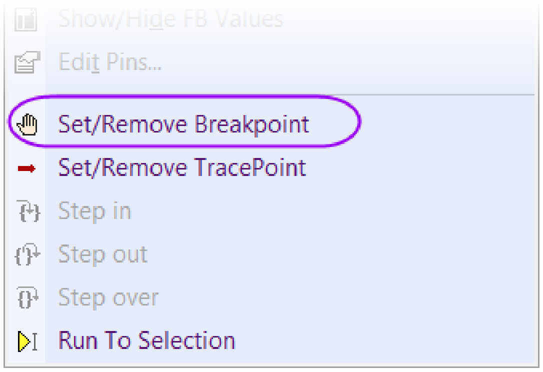
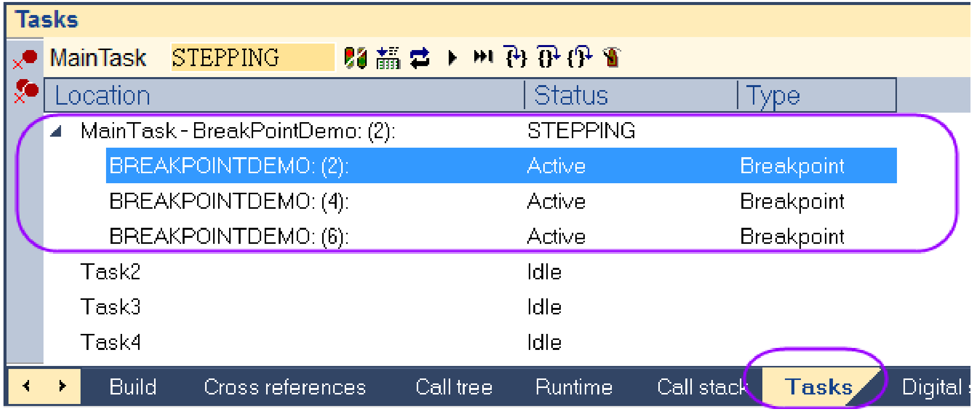


Figure 120: Set/Remove Breakpoint

Breakpoints can be placed in programs, sub-programs or UDFBs. They are not available in SFC programs.

A list of active breakpoints are shown in the *'Tasks'* Log Window (Figure 121). From here you can directly remove breakpoints in the program. Double clicking on a breakpoint in the Log window the program editor jumps to the position of the breakpoint in the program.



Location	Status	Type
MainTask - BreakPointDemo: (2):	STEPPING	
BREAKPOINTDEMO: (2):	Active	Breakpoint
BREAKPOINTDEMO: (4):	Active	Breakpoint
BREAKPOINTDEMO: (6):	Active	Breakpoint
Task2	Idle	
Task3	Idle	
Task4	Idle	

Figure 121: Tasks Log Window

Run To Selection

It is possible to run a program to a defined position. The *'Run To Selection'* function is available in the context menu (Figure 120) of the program editor, when step-by-step execution is active. This function is supported in ST, FBD and LD.

Icon	Description
{}⁺	Step over: The debugger (yellow triangle) jumps to the next source code line in the

	program editor. Now the program executes the instruction at which the debugger in the previous step was located and stops the program execution at the new debugger position. The debugger will not leave the current program editor page. If the next instruction is a function or function block call then the debugger will not enter the source code of the call, but just jumps to the next instruction line of current program editor page.
{}	Step in: If the next instruction is a function or function block call, then the debugger leaves the current program editor page and steps into the function or function block instance and stops the execution at the first line.
{}	Step out: If the current debugger position is inside a function or function block source code page, then in the next step the debugger leaves the page and jumps to the instruction which follows the function call. The program executes the function or function block from the previous debugger position up to the end of the block.

Table 25: Step by Step commands

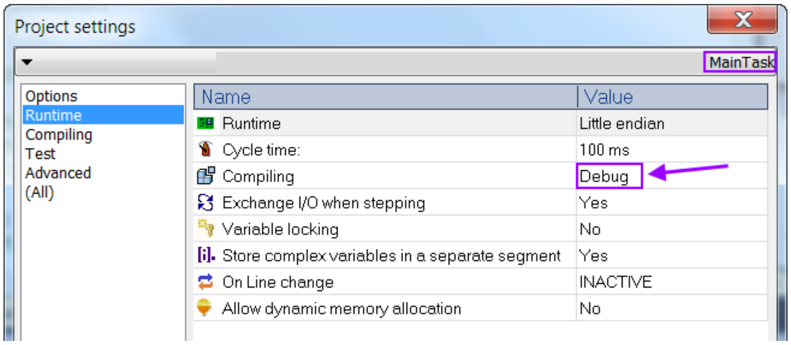
Icon	Shortcut	Command
⇄	CTRL + Alt + F4	On line change
	CTRL + F5	Debug
	F4	Pause/resume
	F5	Simulation
✋	F9	Set/Remove breakpoint
	F11	Download
	SHIFT + CTRL + F4	Start/stop application
{}	F8	Step In
{}	CTRL + F8	Step Out
{}	SHIFT + F8	Step Over
▷I	Ctrl+F10	Run to Selection

Table 26: Key shortcuts for debugging commands

7.8.2 Example

Example of a step by step debug procedure:

Step 1: Set compiler to debug mode (*'Project\Settings...'*)



Name	Value
Runtime	Little endian
Cycle time:	100 ms
Compiling	Debug
Exchange I/O when stepping	Yes
Variable locking	No
Store complex variables in a separate segment	Yes
On Line change	INACTIVE
Allow dynamic memory allocation	No

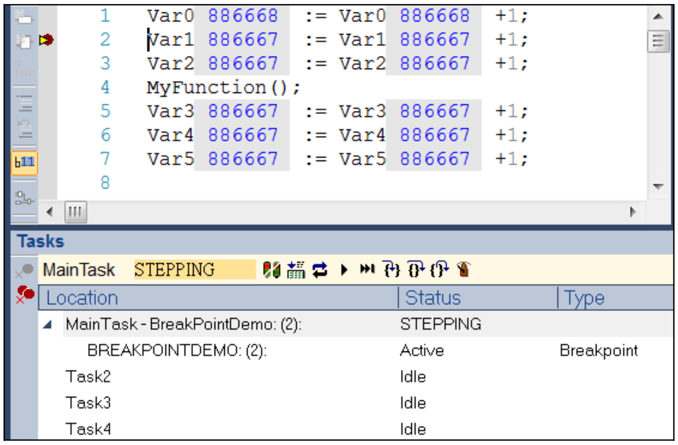
Step 2: Compile your program, download it to the target runtime.

Step 3: Set the workbench in online mode and start the application from the workbench.

Step 4: Add a breakpoint at line 2:

Set the cursor at line 2 and press F9.

Once the program hits the breakpoint the program execution stops and a yellow triangle appears next to the breakpoint. The code of line 2 will not be executed.

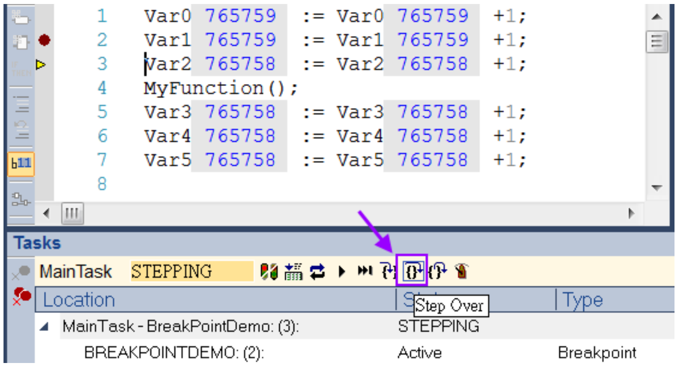


```
1  Var0 886668  := Var0 886668  +1;
2  Var1 886667  := Var1 886667  +1;
3  Var2 886667  := Var2 886667  +1;
4  MyFunction();
5  Var3 886667  := Var3 886667  +1;
6  Var4 886667  := Var4 886667  +1;
7  Var5 886667  := Var5 886667  +1;
8
```

Location	Status	Type
MainTask - BreakPointDemo: (2):	STEPPING	
BREAKPOINTDEMO: (2):	Active	Breakpoint
Task2	Idle	
Task3	Idle	
Task4	Idle	

Step 5: Run program to next line by clicking the *'Step Over'* button.

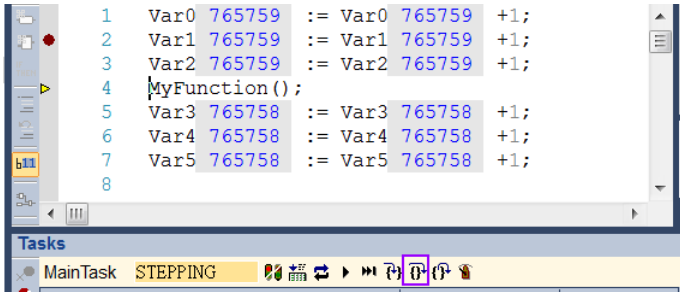
Now the program executes the instruction of line 2 and stops the program at line 3.



```
1  Var0 765759  := Var0 765759  +1;
2  Var1 765759  := Var1 765759  +1;
3  Var2 765758  := Var2 765758  +1;
4  MyFunction();
5  Var3 765758  := Var3 765758  +1;
6  Var4 765758  := Var4 765758  +1;
7  Var5 765758  := Var5 765758  +1;
8
```

Location	Status	Type
MainTask - BreakPointDemo: (3):	STEPPING	
BREAKPOINTDEMO: (2):	Active	Breakpoint

Click the *'Step Over'* button again to stop at the next line (line 4)

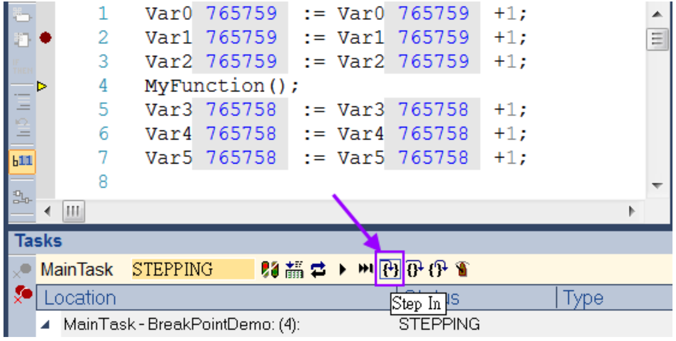


```
1   Var0 765759  := Var0 765759  +1;
2   Var1 765759  := Var1 765759  +1;
3   Var2 765759  := Var2 765759  +1;
4   MyFunction();
5   Var3 765758  := Var3 765758  +1;
6   Var4 765758  := Var4 765758  +1;
7   Var5 765758  := Var5 765758  +1;
8
```

Tasks

MainTask STEPPING

Step 6: Enter a function call:
Line 4 calls a user defined function (*'MyFunction'*). In the next step we want to enter the source code of this function. Click the *'Step In'* button to stop the program at the first instruction line of the function.



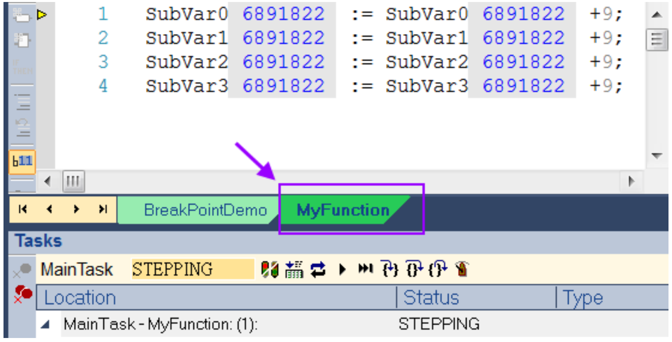
```
1   Var0 765759  := Var0 765759  +1;
2   Var1 765759  := Var1 765759  +1;
3   Var2 765759  := Var2 765759  +1;
4   MyFunction();
5   Var3 765758  := Var3 765758  +1;
6   Var4 765758  := Var4 765758  +1;
7   Var5 765758  := Var5 765758  +1;
8
```

Tasks

MainTask STEPPING

Location	Status	Type
MainTask - BreakPointDemo: (4):	STEPPING	

The workbench opens the source code page of the calling function and set the stop triangle at the first line.



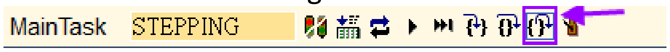
```
1   SubVar0 6891822  := SubVar0 6891822  +9;
2   SubVar1 6891822  := SubVar1 6891822  +9;
3   SubVar2 6891822  := SubVar2 6891822  +9;
4   SubVar3 6891822  := SubVar3 6891822  +9;
```

Tasks

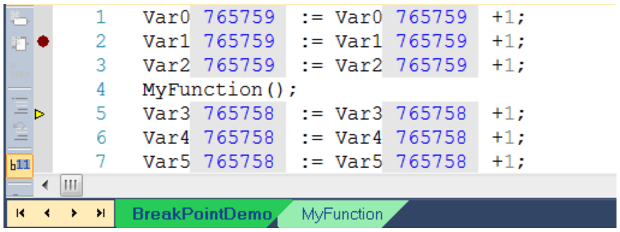
MainTask STEPPING

Location	Status	Type
MainTask - MyFunction: (1):	STEPPING	

Step 7: Leave the source code of the function by clicking the *'Step Out'* button
In this step the program should execute the function until the end and stop at the instruction following the function call.



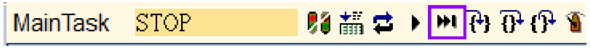
MainTask STEPPING



```
1  Var0 765759  := Var0 765759  +1;
2  Var1 765759  := Var1 765759  +1;
3  Var2 765759  := Var2 765759  +1;
4  MyFunction();
5  Var3 765758  := Var3 765758  +1;
6  Var4 765758  := Var4 765758  +1;
7  Var5 765758  := Var5 765758  +1;
```

Step 8: Run the program to the end of the cycle.

1. Remove the breakpoint in line 2 by setting the cursor to line 2 and pressing F9 key.
2. Click the *'Execute a single cycle'* button to let the program to execute all the code from the current position to the last instruction of the cycle.

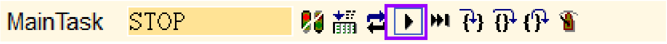


Step 9: Run a full single cycle.

If no break point exist in the program and the *'Execute a single cycle'* button is activated then the program executes one full cycle before it stops at the cycle end.

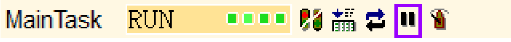
Step 10: Run the program continuously without a break (normal execution mode).

1. Remove all breakpoints. Make sure no break point exist in the program
2. Click *'Resume cycle to cycle'* mode.



The program runs now continuously without interrupt.

If you want to stop the program at the end of the cycle and go back into cycle to cycle debug mode then click the *'Pause (cycle to cycle)'* button

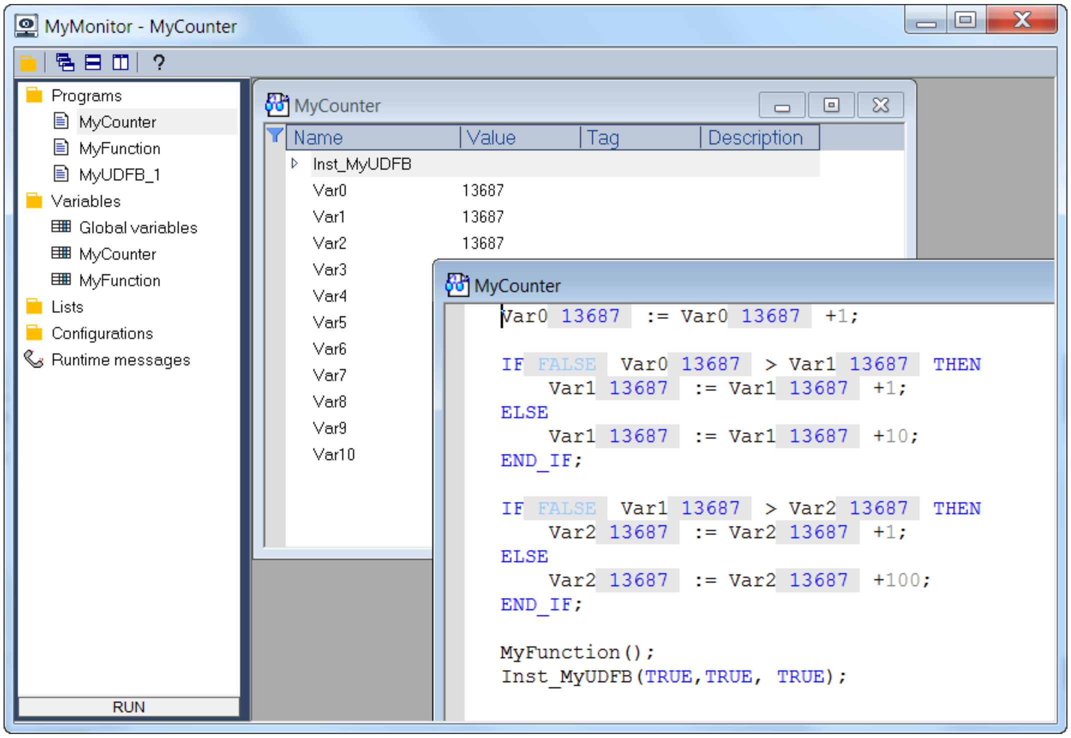


Step 11: Remember to compile the application in release mode after debugging has been finished.

7.9 W5Monitoring Utility

When the workbench is set into online mode the current variable values of the application are being shown in a box next to each variable in the source code and in the variable editor. The source code in the programming area can not be modified while the workbench is in online mode but the variable editor allows the user to force a variable value change.

The purpose of the W5Monitoring utility is to monitor variable values and support forced variable value change without using the workbench. The utility provides similar functions as the workbench when in online mode: current values are shown inline next to the variables in the source code and in the variable editor (Figure 122).



Name	Value	Tag	Description
Inst_MyUDFB			
Var0	13687		
Var1	13687		
Var2	13687		
Var3			
Var4			
Var5			
Var6			
Var7			
Var8			
Var9			
Var10			

```
Var0 13687  := Var0 13687  +1;

IF FALSE  Var0 13687  > Var1 13687  THEN
    Var1 13687  := Var1 13687  +1;
ELSE
    Var1 13687  := Var1 13687  +10;
END_IF;

IF FALSE  Var1 13687  > Var2 13687  THEN
    Var2 13687  := Var2 13687  +1;
ELSE
    Var2 13687  := Var2 13687  +100;
END_IF;

MyFunction();
Inst_MyUDFB(TRUE,TRUE, TRUE);
```

Figure 122: W5Monitoring utility

The data to be displayed in the utility has to be set via the workbench wizard. The wizard allows you to select which programs, variable list and runtime messages to be monitored by the W5Monitoring utility. Normally a PLC project exist of at least one task and several program organization units (POUs). IEC 61131-3 defines three types of POUs: programs, function blocks, and functions (Figure 123). The W5Monitoring utility can only display the POU of one task and therefore the user has to decide which task and POU to display. If inside the task more than one program, function block or functions has been defined then the user can select which one to display during execution. If required all the POU defined within one task can be selected.

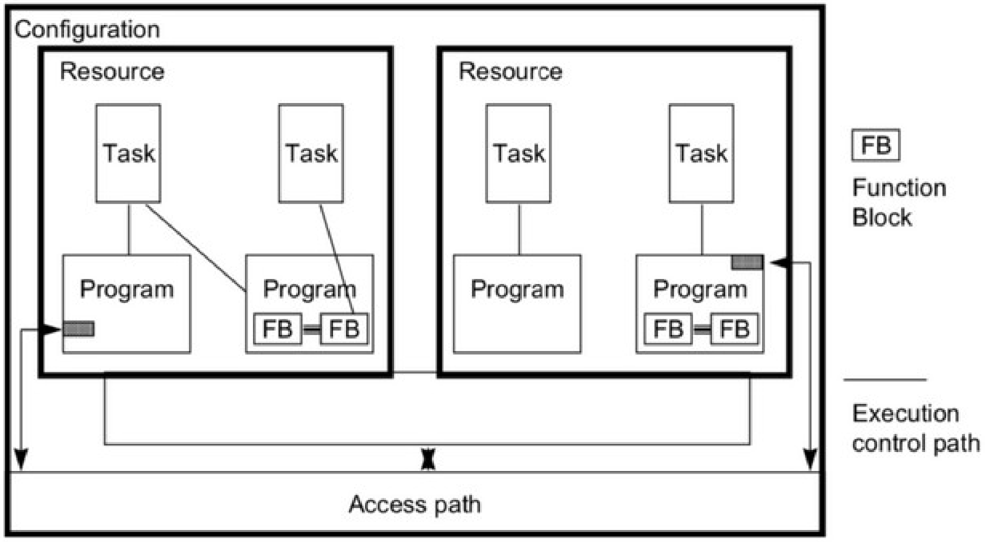


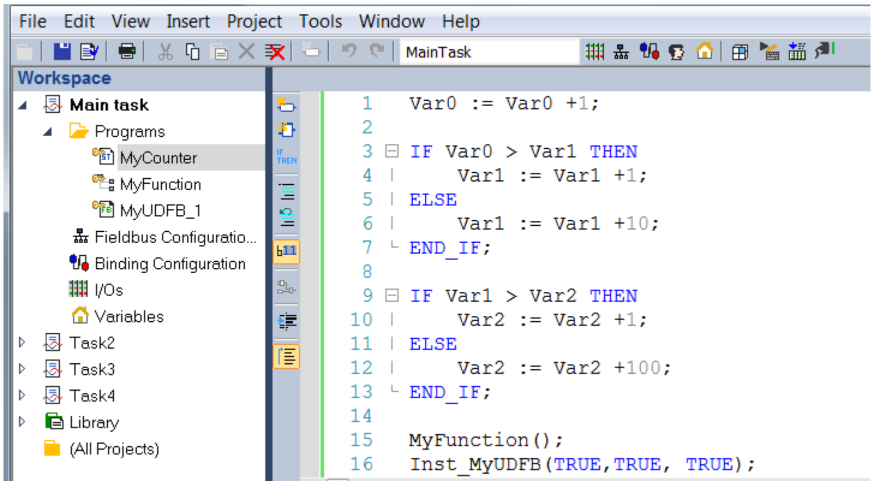
Figure 123: The IEC 61131-3 software model

All POU stored in the monitoring application are encrypted. For security reasons the W5Monitoring utility does allow any POU content to be copied or dragged to another program.

7.9.1 Create Monitoring Application File

This section describes how to use the workbench wizard to create a monitoring file for the W5Monitoring utility.

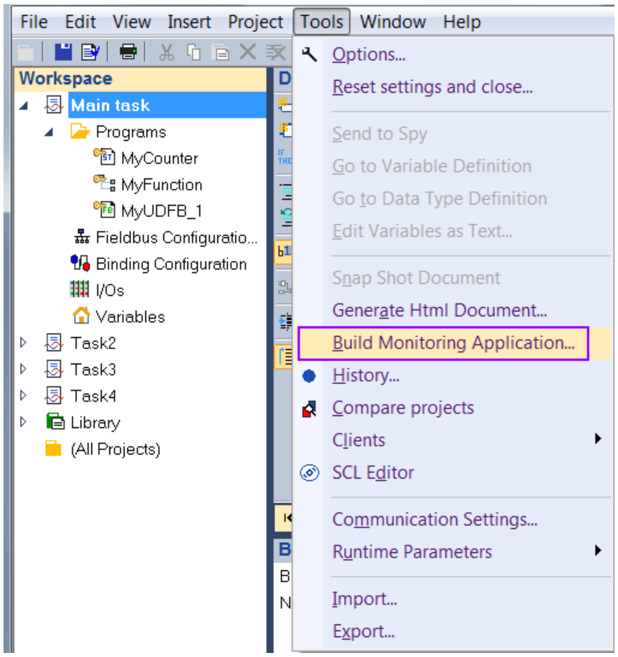
Step 1: Compile the project and download it to the runtime.
In the following steps it is shown to create a monitoring file for the main task. In this example the main task runs three POUs: a program *'MyCounter'*, a function *'MyFunction'* and a user defined function block *'MyUDFB'*. All three POUs will be exported to the W5Monitoring utility.



```
Var0 := Var0 +1;

IF Var0 > Var1 THEN
    Var1 := Var1 +1;
ELSE
    Var1 := Var1 +10;
END_IF;

IF Var1 > Var2 THEN
    Var2 := Var2 +1;
ELSE
    Var2 := Var2 +100;
END_IF;

MyFunction();
Inst_MyUDFB(TRUE,TRUE, TRUE);
```

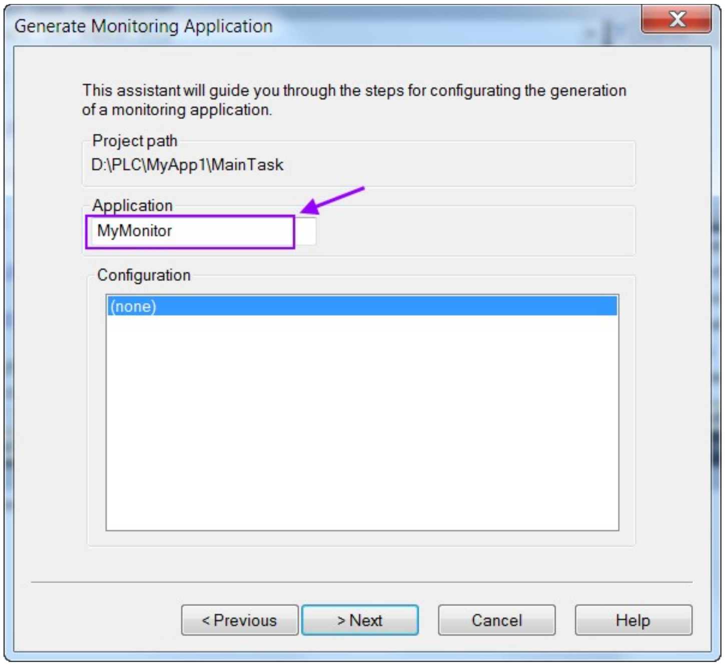
Step 2: Start the monitoring wizard:

1. Double click the *'Main task'* in the workspace to tell the workbench that the POU of this task will be exported
2. Select *'Build Monitoring Application...'*from the menu bar *'Tools'*.



Step 3: Give a name to the monitoring application.

This name will be displayed in the main title bar of the W5Monitoring utility. The configuration of a monitoring application can be saved to the workbench project. This allows you to modify the configuration at a later stage.


Generate Monitoring Application

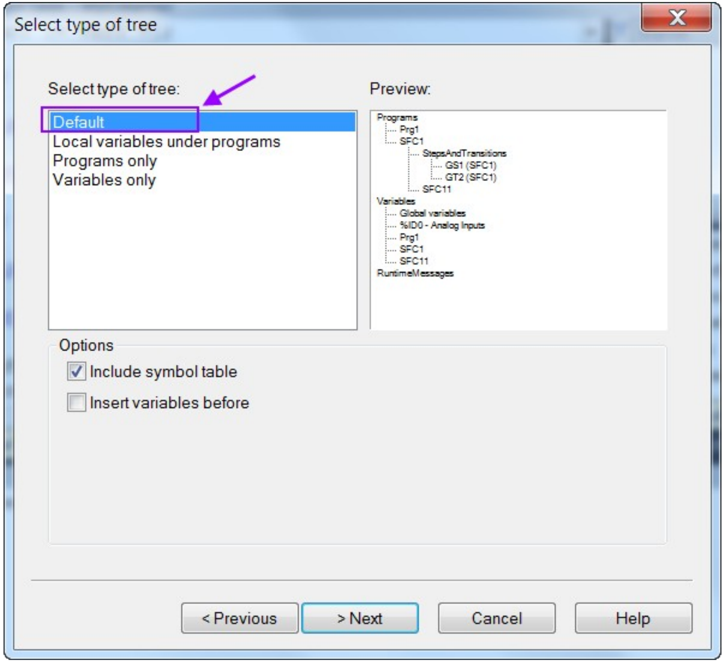
This assistant will guide you through the steps for configurating the generation of a monitoring application.

Project path: D:\PLC\MyApp1\MainTask

Application: MyMonitor

Configuration: (none)

Step 4: Select how to display the POU in the tree. In addition select to display the program editor of the POU, the variable editor, or both. Here we select both (Default).


Select type of tree

Select type of tree:

- Default
- Local variables under programs
- Programs only
- Variables only

Preview:

```
Programs
  Prg1
  SFC1
    StepsAndTransitions
      GS1 (SFC1)
      GT2 (SFC1)
    SFC11
Variables
  Global variables
  %ID0 - Analog Inputs
  Prg1
  SFC1
  SFC11
RuntimeMessages
```

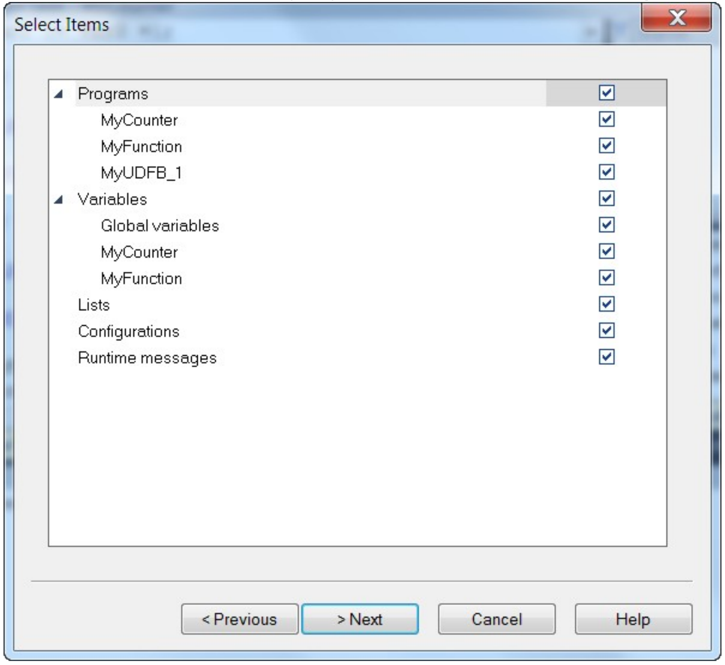
Options

☑ Include symbol table

☐ Insert variables before

If you check the *'Include symbol table'* option, the monitoring application will work on the full symbol table generated by the compiler. In this case you have to make sure that the same version of the application is running on the runtime. If you don't select this option, then the W5Monitoring utility will automatically upload symbols from the target at connection time. In this case, you have to ensure that all used symbols are embedded in the runtime application.

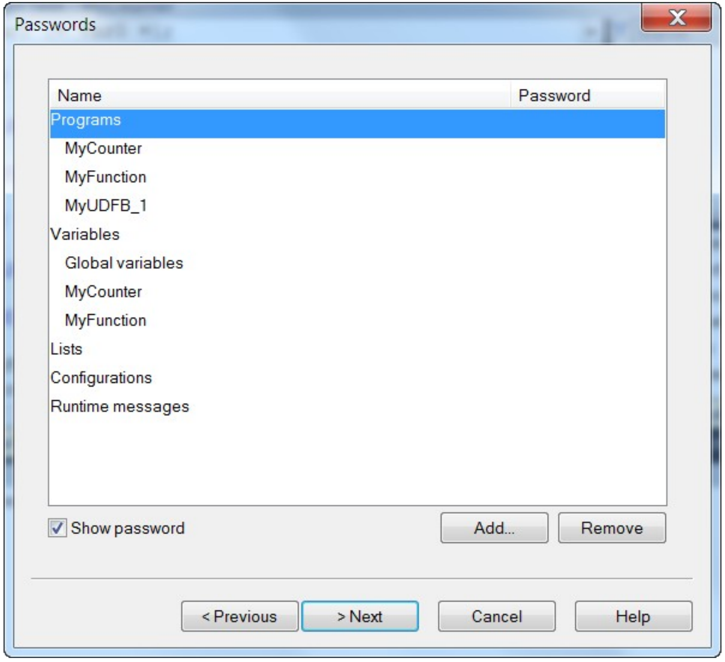
Step 5: Select the items (POU, variables) of the project that you want to include in the monitoring application. By default all items are selected. Deselect the items that should not be displayed in the W5Monitoring utility.



Select Items

- Programs ☑
  - MyCounter ☑
  - MyFunction ☑
  - MyUDFB_1 ☑
- Variables ☑
  - Global variables ☑
  - MyCounter ☑
  - MyFunction ☑
- Lists ☑
- Configurations ☑
- Runtime messages ☑

Step 6: Add passwords protection for selected items (optional)
For each item included in the application you can define a password so that the corresponding document will be protected in the monitoring application.



Passwords

Name	Password
Programs	
MyCounter	
MyFunction	
MyUDFB_1	
Variables	
Global variables	
MyCounter	
MyFunction	
Lists	
Configurations	
Runtime messages	

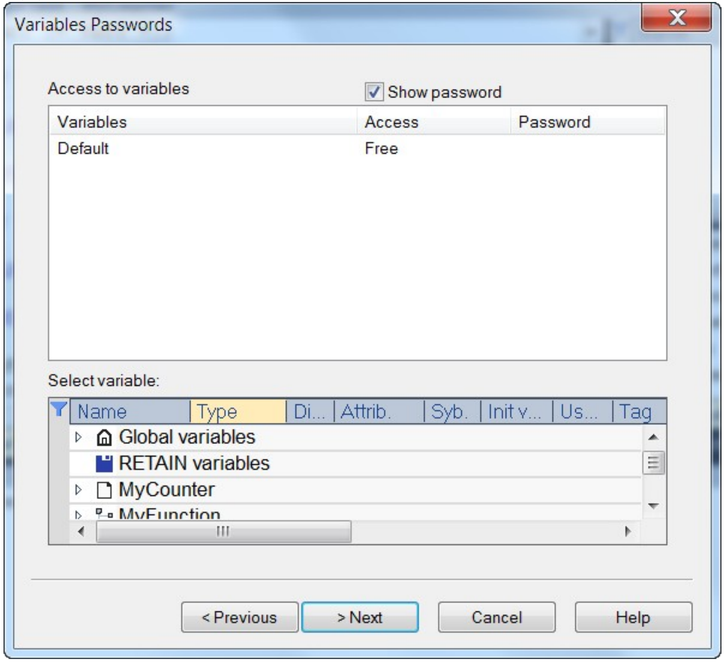
☑ Show password

Step 7: Passwords for variables

You can define passwords for the W5Monitoring utility to enable write access (forcing) to runtime variables. For each variable you can select one of the following access protection:

- **Free** : the variable can be freely forced.
- **Protected** : forcing the variable is possible with a password.
- **No** : the variable can never be forced.

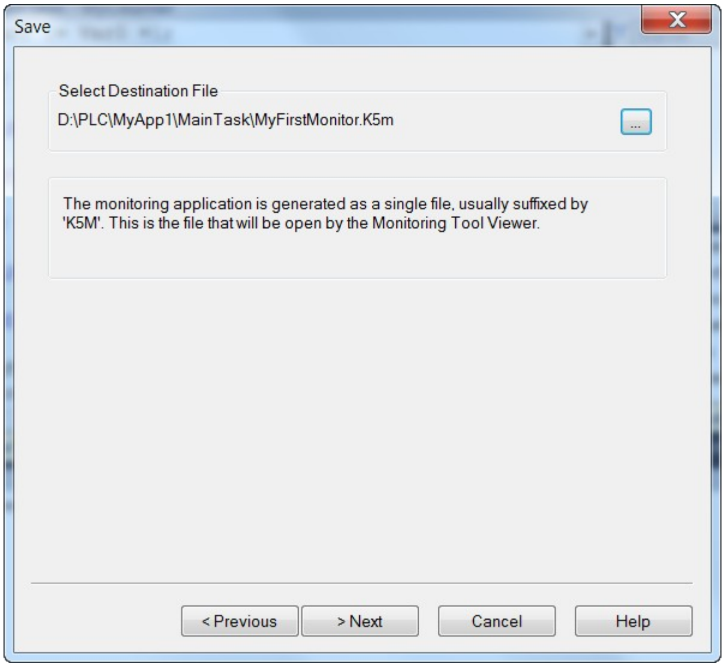
Simply drag variables from the list at the bottom to the upper list to set its protection mode. The *'Default'* choice indicates the protection mode to be to applied to all variables which hav not been dragged to the upper list.



Variables	Access	Password
Default	Free	

Step 8: Set the monitoring application file name and the directory.

Here we select *'MyFirstMonitor.K5m'* as a name. After clicking *'Next'* the monitoring application file is being generated and stored as a unique compressed file. To protect your application from piracy all POU stored in the monitoring application are encrypted.



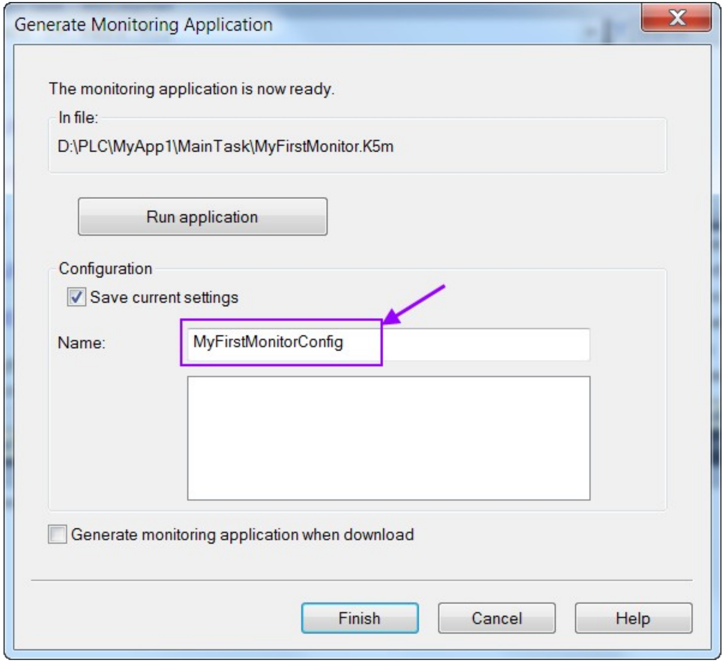
Save

Select Destination File

D:\PLC\MyApp1\MainTask\MyFirstMonitor.K5m

The monitoring application is generated as a single file, usually suffixed by 'K5M'. This is the file that will be open by the Monitoring Tool Viewer.

Step 9: Save current monitoring application configuration.
This configuring file can be loaded and modified at a later stage if required.
The configuration will be save once the *'Finish'* has been clicked.



Generate Monitoring Application

The monitoring application is now ready.

In file:

D:\PLC\MyApp1\MainTask\MyFirstMonitor.K5m

Configuration

☑ Save current settings

Name: MyFirstMonitorConfig

□ Generate monitoring application when download

7.9.2 Running Monitoring Application

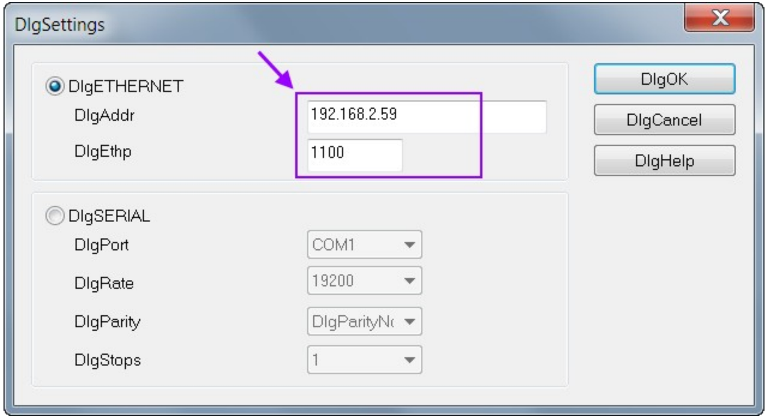
The W5Monitoring utility can be started in two ways:

- Via the Start menu:

  Step 1: WinGRAF Workbench x.x -->Tools->W5Monitoring

  Step 2: Open the monitoring file (*.Km5)

  Step 3: Enter the IP address and port number of the target runtime



DlgAddr 192.168.2.59

DlgEthp 1100

DlgPort COM1

DlgRate 19200

DlgParity DlgParityN(

DlgStops 1

Now the W5Monitoring utility shows the POU with the variable values.

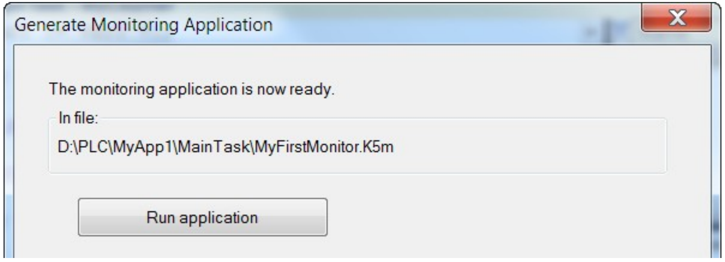
- Via the workbench wizard:

  Step 1: Make sure the communication setting is set to the IP address and port number of the target runtime (Tools-->Communication Settings...)

  Step 2: Open the wizard: *'Tool/Built Monitoring Application...'*

  Step 3: Select an application from the configuration list

  Step 4: Click *'Next'* button six times until you reach the last page of the wizard Click the *'Run application'* button.



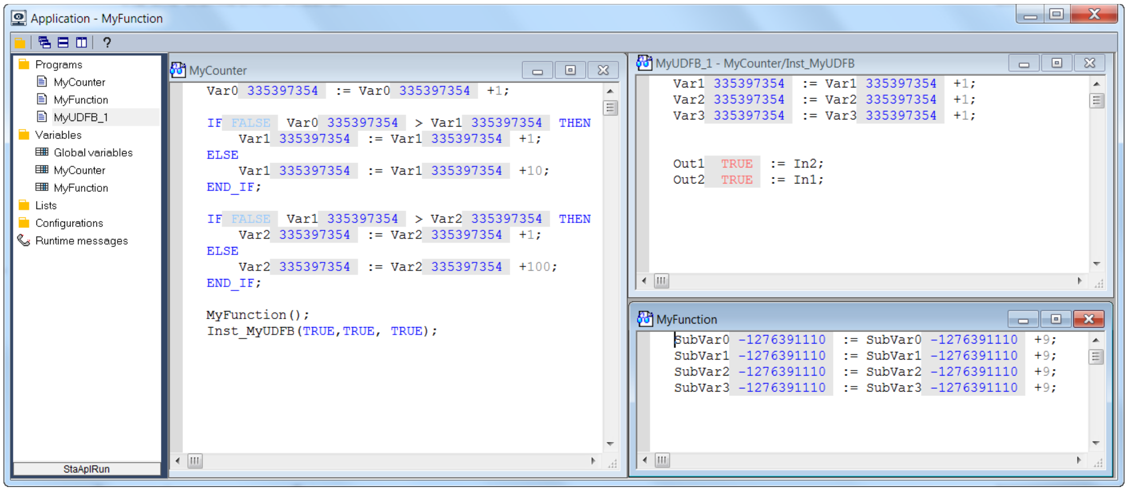
The monitoring application is now ready.

In file:

D:\PLC\MyApp1\MainTask\MyFirstMonitor.K5m

Now the W5Monitoring utility shows the POU with the variable values.

The tree view on the left list all the items (programs, variable list, etc.) which has been exported by the workbench wizard. By double clicking one of the items a window with the items data pops up. Several windows of different item can be open at the same time (Figure 124). To protect your software POU data displayed in the W5Monitoring utility can not be copied.



```
Var0 335397354  := Var0 335397354  +1;

IF FALSE  Var0 335397354  > Var1 335397354  THEN
    Var1 335397354  := Var1 335397354  +1;
ELSE
    Var1 335397354  := Var1 335397354  +10;
END_IF;

IF FALSE  Var1 335397354  > Var2 335397354  THEN
    Var2 335397354  := Var2 335397354  +1;
ELSE
    Var2 335397354  := Var2 335397354  +100;
END_IF;

MyFunction();
Inst_MyUDFB(TRUE,TRUE, TRUE);
```

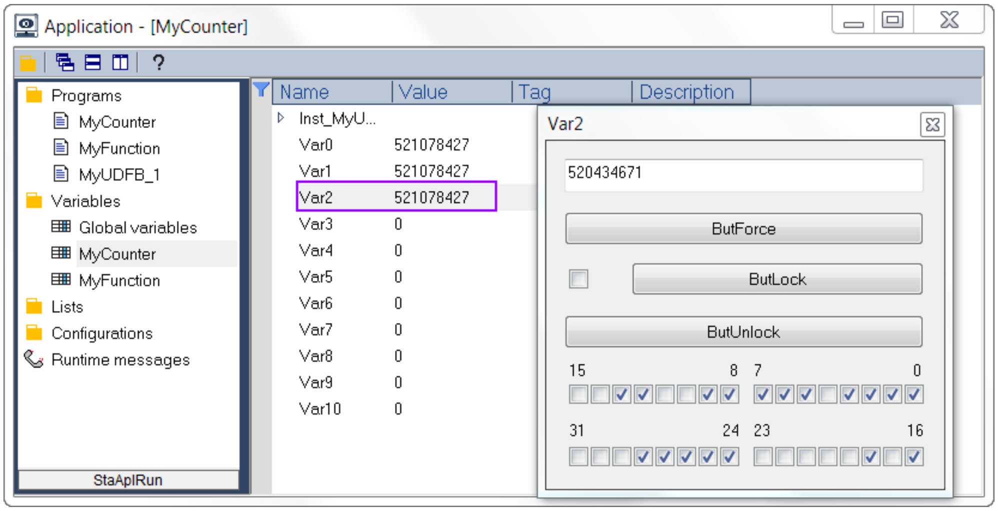
```
Var1 335397354  := Var1 335397354  +1;
Var2 335397354  := Var2 335397354  +1;
Var3 335397354  := Var3 335397354  +1;


Out1  TRUE  := In2;
Out2  TRUE  := In1;
```

```
SubVar0 -1276391110  := SubVar0 -1276391110  +9;
SubVar1 -1276391110  := SubVar1 -1276391110  +9;
SubVar2 -1276391110  := SubVar2 -1276391110  +9;
SubVar3 -1276391110  := SubVar3 -1276391110  +9;
```

Figure 124: W5Monitoring utility running a monitoring application

Items in the *'Variables'* tree view shows all the exported variable editors of the workbench. In the workbench each POU has its own variable editor and the user can decide which one to export to the monitoring application which has been described in the previous section. Depending of the protection mode set in the monitoring application file variable value change is supported (Figure 125):

- **Free**: the variable can be freely forced.
- **Protected**: forcing the variable is possible with a password.
- **No**: the variable can never be forced.



Name	Value	Tag	Description
Inst_MyU...			
Var0	521078427		
Var1	521078427		
Var2	521078427		
Var3	0		
Var4	0		
Var5	0		
Var6	0		
Var7	0		
Var8	0		
Var9	0		
Var10	0		

520434671

□ ButLock

15 … 8: □ □ ☑ ☑ □ □ ☑ ☑ 7 … 0: ☑ ☑ ☑ □ ☑ ☑ ☑ ☑

31 … 24: □ □ □ ☑ ☑ ☑ ☑ ☑ 23 … 16: □ □ □ □ □ ☑ □ ☑

Figure 125: Force variable value change

8 Online Program Change

Online Change enables you to update your PLC application on the fly, while it is running. You do not need to stop the application, download the new code and start again. You only need to modify, recompile and download the new code.
Depending on the PLC code size, the time to perform the Online Change operation can take more than one cycle. In that case, you can miss one PLC cycle before the changeover becomes effective.

Online Change functions should primarily be used for the rare cases where small modifications to the program code of an application has to be done while the application is not allowed to be halted and has to run non-stop. In general it is not recommended to do online modification and rather stop the running control application before downloading a modified version.

8.1 Online Changes Limitations

When Online Change is enabled, the following kinds of changes on the fly are supported:

- Change the code of a program.
- Change the condition of a SFC transition or the actions of a SFC step (Figure 126).
- Create, rename or delete global and local variables and function block instances ().

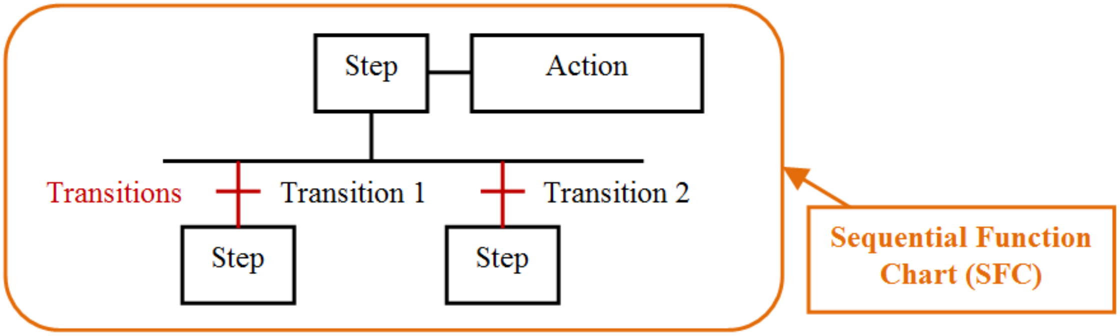
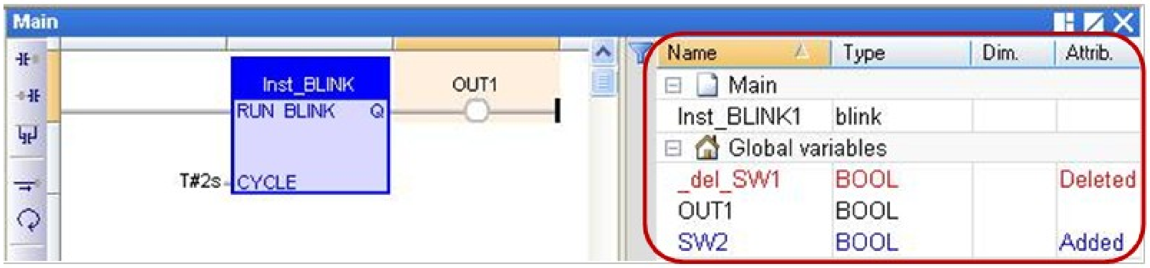


Figure 126: Condition change for SFC transition



Name	Type	Dim.	Attrib.
Main			
Inst_BLINK1	blink		
Global variables			
_del_SW1	BOOL		Deleted
OUT1	BOOL		
SW2	BOOL		Added

Figure 127: Create and delete a global variable

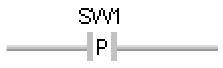
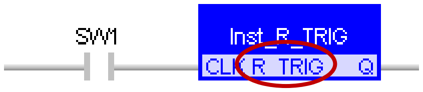
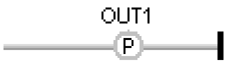
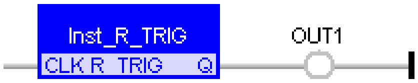
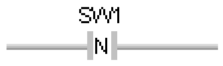
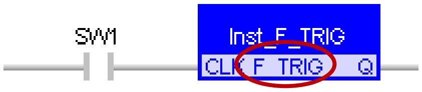
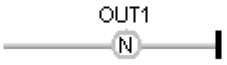
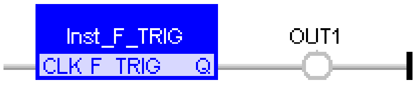
The following kind of online changes are not allowed:

- Create, delete or rename a program. (It will appear a warning message if a program is attempted to be deleted)
- Change SFC charts.
- Change the local parameters and variables of a UDFB.
- Change the type or dimension (or string length) of a variable or function block instance.
- Change the set of I/O boards.
- Change the definition of RETAIN variables.

In addition, the following programming features are not safe during a online change and therefore should not be used:

- Pulse (P or N) contacts and coils (edge detection).

✍ Instead, you must use declared instances of R_TRIG and F_TRIG function blocks.

Rising Pulse Detection		
	Before Enable	After Enable
P (False > True)		
		
Decreased Pulse Detection		
N (True > False)		
		

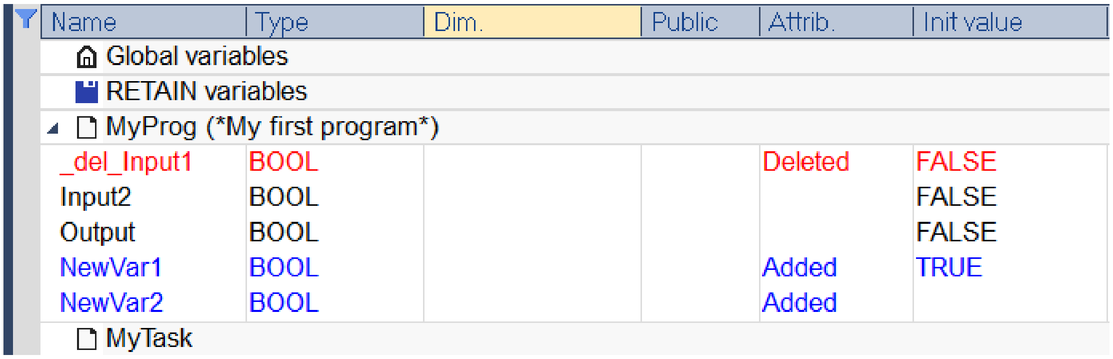
- Loops in FBD with no declared variable linked.

✍ You need to explicitly insert a variable in the loop.

8.2 Using Online Change

The normal procedure for developing a PLC application is as follows: The developer creates a new PLC application and adds logic and function blocks to the program and compiles and download it to the runtime. If necessary the developer can modify the program by changing the program logic, adding or deleting variables or function block instances, recompile and download the application. Variables which have been deleted are not shown in the variable list of the workbench any longer.

Once the Online Change for the application has been activated all the variable and function block changes made afterwards will be shown in the variable list. Newly added variables will be shown in blue and deleted variable in red. The attribute for variables and function blocks instances which have been added after the Online Change has been activate are shown as *'Added'* and deleted variables as *'Deleted'*. In addition the workbench will automatically add the prefix *'_del_'* to the names of deleted variables (Figure 128).



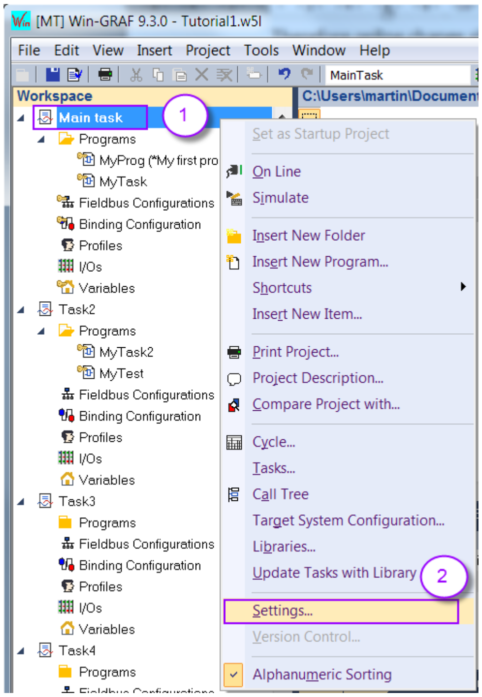
Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
RETAIN variables					
MyProg (*My first program*)					
_del_Input1	BOOL			Deleted	FALSE
Input2	BOOL				FALSE
Output	BOOL				FALSE
NewVar1	BOOL			Added	TRUE
NewVar2	BOOL			Added	
MyTask					

Figure 128: Variable changes done after online change has been activated

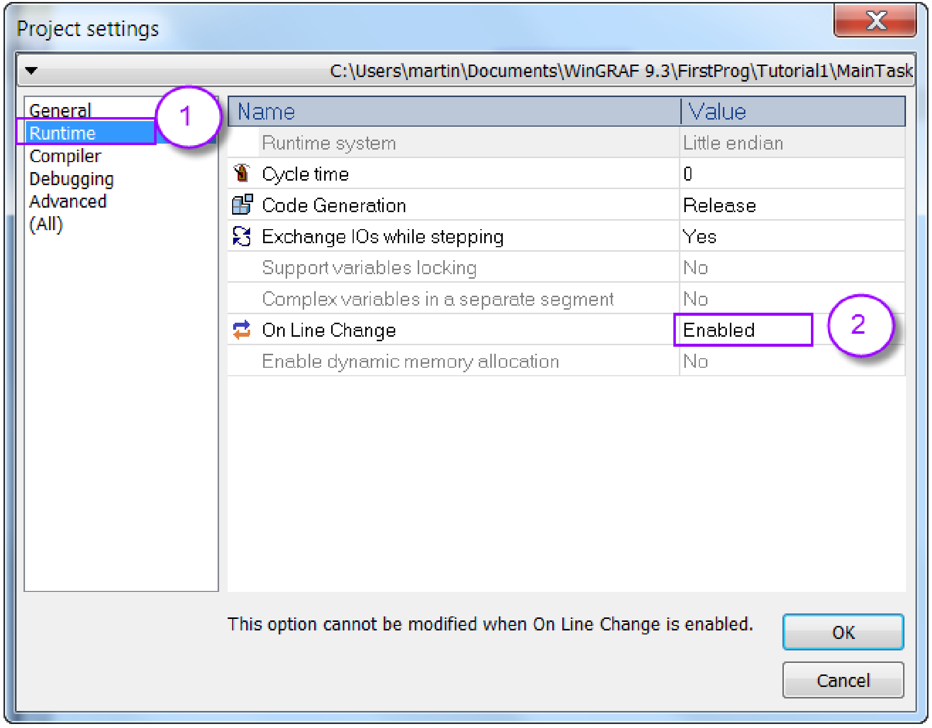
The Online Change should only be activated once the main application development has finished and it has been determined that the control logic is running fine. The purpose of the Online Change functions is just to do minor adjustments to the logic control and therefore should only be used where small modifications to the program code is necessary.

Procedure for using online change:

Step 1: Enable the online change function for the task which needs to support online change. of the workbench. This can be done as follows: Right click the task name and select '*Settings...*'



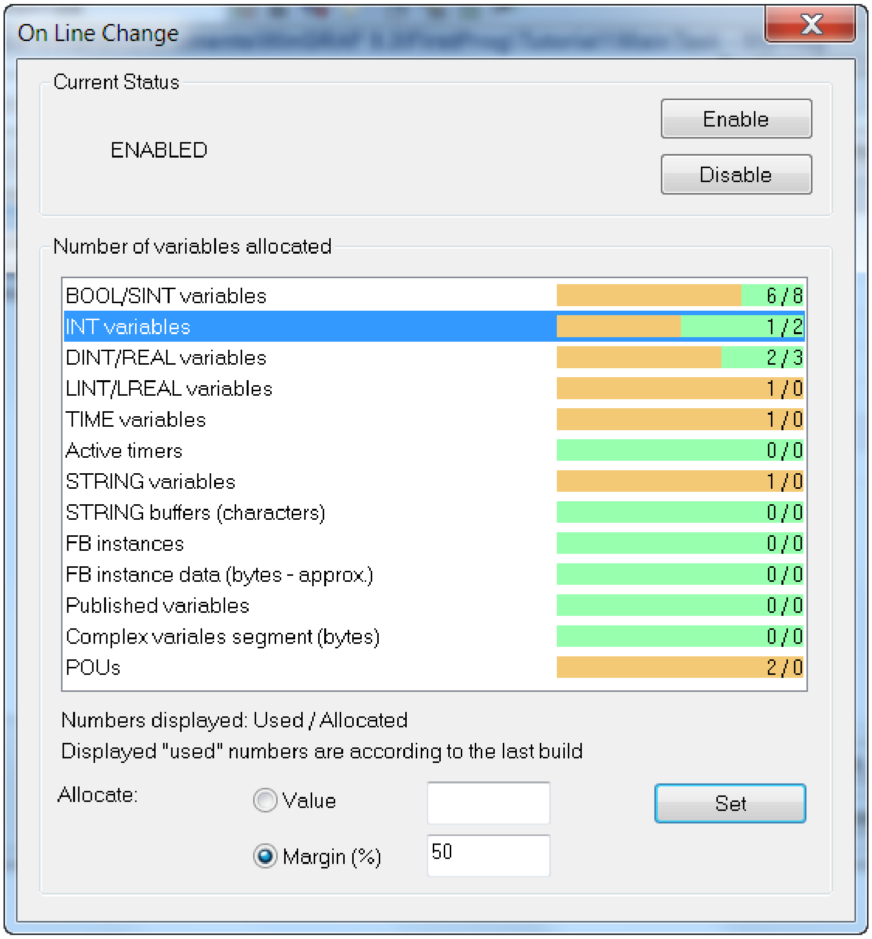
From the popup window select *'Runtime'* and double click *'On Line Change'* item to set it *'Enabled'*.



Name	Value
Runtime system	Little endian
Cycle time	0
Code Generation	Release
Exchange IOs while stepping	Yes
Support variables locking	No
Complex variables in a separate segment	No
On Line Change	Enabled
Enable dynamic memory allocation	No

This option cannot be modified when On Line Change is enabled.

Another pop-up window will appear in which you can allocate the memory to be reserved for added variables.



Current Status

ENABLED

Number of variables allocated

BOOL/SINT variables	6 / 8
INT variables	1 / 2
DINT/REAL variables	2 / 3
LINT/LREAL variables	1 / 0
TIME variables	1 / 0
Active timers	0 / 0
STRING variables	1 / 0
STRING buffers (characters)	0 / 0
FB instances	0 / 0
FB instance data (bytes - approx.)	0 / 0
Published variables	0 / 0
Complex variales segment (bytes)	0 / 0
POUs	2 / 0

Numbers displayed: Used / Allocated

Displayed "used" numbers are according to the last build

Allocate: ○ Value

● Margin (%) 50

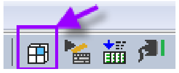
In order to allow the declaration of new variables and blocks after the Online Change function has been enabled, you have to define the amount of memory to be allocated in the target runtime for each type of data. This includes:

- The number of variable for each type (8, 16, 32 or 64 bits, character strings).
- The number of function block instances.
- The amount of memory for storing character strings.
- The amount of memory for private data of function block instances.
- The amount of variables published (with embedded symbol or profile).
- The sizing of extra segment for storing complex variables.

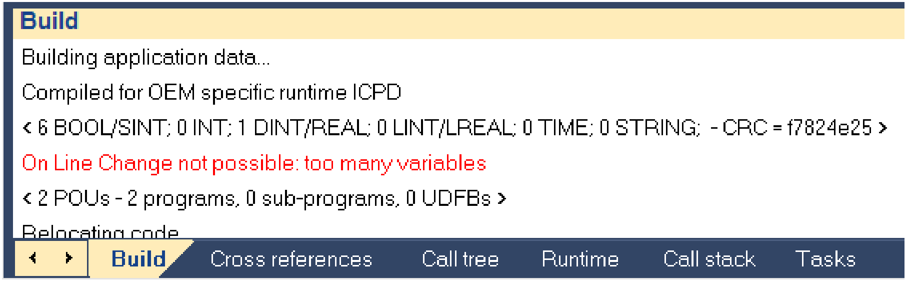
To setup a value, select the corresponding item in the list, select the ***Value*** option, enter a new number in the Value box and press the Set button. You can select several items in the list for assigning the value to any selected items.

If your project has been built, the box shows you in the list the size actually used by the application according to the last build. A progress bar shows you the percentage of used space for each item. In addition, instead of entering an absolute value, you may select to enter a percentage of the used memory to add to the used space.

Step 2: Recompile the application again:



Ignore the warning message generated by the compiler:



Build

Building application data...

Compiled for OEM specific runtime ICPD

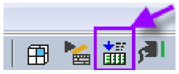
< 6 BOOL/SINT; 0 INT; 1 DINT/REAL; 0 LINT/LREAL; 0 TIME; 0 STRING; - CRC = f7824e25 >

On Line Change not possible: too many variables

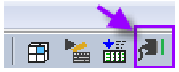
< 2 POUs - 2 programs, 0 sub-programs, 0 UDFBs >

Relocating code

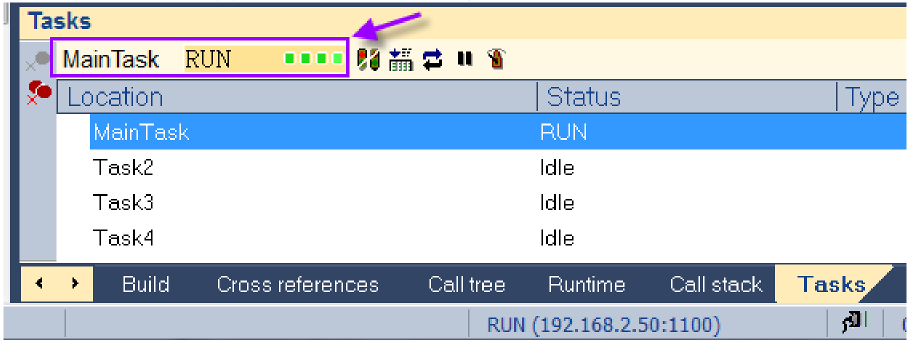
Step 3: Download the application to the runtime



Step 4: Connect to the runtime and start the application.



Make sure the application is running:



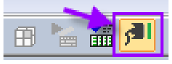
Tasks

MainTask RUN

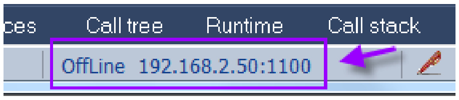
Location	Status	Type
MainTask	RUN	
Task2	Idle	
Task3	Idle	
Task4	Idle	

RUN (192.168.2.50:1100)

Step 5: Do minor changes to the logic program by adding and deleting variables. If the workbench in the online mode it shows all the current values of each variable next to the variables. In this mode the logic program can not be changed. Therefore first set the workbench in offline mode:

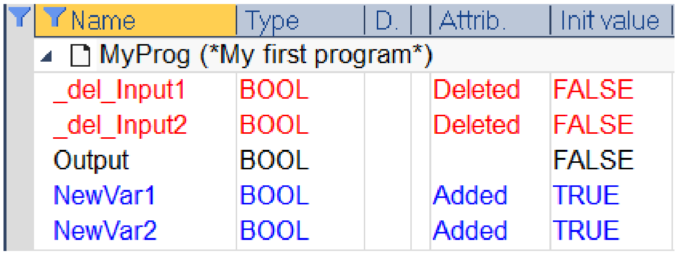


The offline mode will be displayed in the status bar at the bottom of the window:



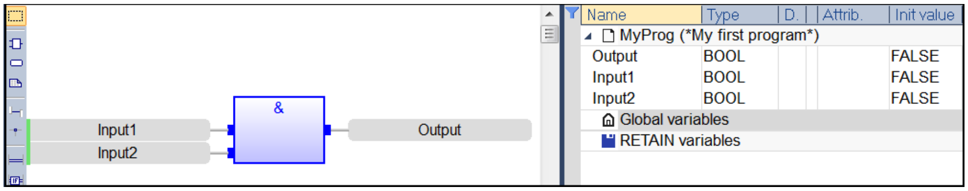
ces	Call tree	Runtime	Call stack
OffLine 192.168.2.50:1100			

Add new variables and delete variables. Deleted variables will be shown in red with a *'Deleted'* attribute and newly added variables are shown in blue with *'Added'* attribute.



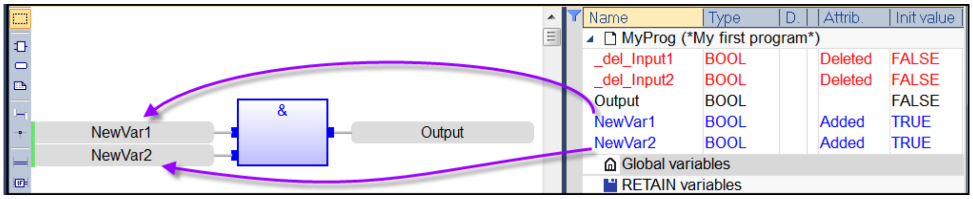
Name	Type	D.	Attrib.	Init value
MyProg (*My first program*)				
_del_Input1	BOOL		Deleted	FALSE
_del_Input2	BOOL		Deleted	FALSE
Output	BOOL			FALSE
NewVar1	BOOL		Added	TRUE
NewVar2	BOOL		Added	TRUE

In the following code the both inputs of the AND function are replace by the newly added input variables:
Input1 is replace with *NewVar1* and *Input2* is replaced with *NewVar2*.



Name	Type	D.	Attrib.	Init value
MyProg (*My first program*)				
Output	BOOL			FALSE
Input1	BOOL			FALSE
Input2	BOOL			FALSE
Global variables				
RETAIN variables				

Figure 129: Original program

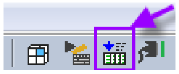


Name	Type	D.	Attrib.	Init value
MyProg (*My first program*)				
_del_Input1	BOOL		Deleted	FALSE
_del_Input2	BOOL		Deleted	FALSE
Output	BOOL			FALSE
NewVar1	BOOL		Added	TRUE
NewVar2	BOOL		Added	TRUE
Global variables				
RETAIN variables				

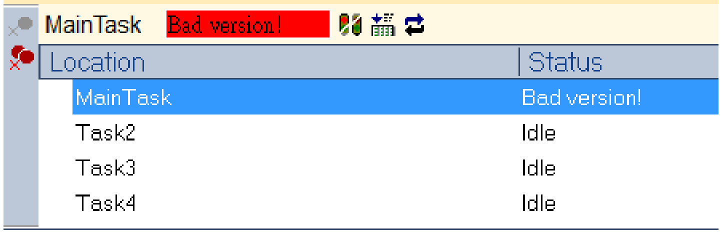
Figure 130: Modified program

Step 6: Recompile the program after the modifications have been made.

Step 7: Go into online mode

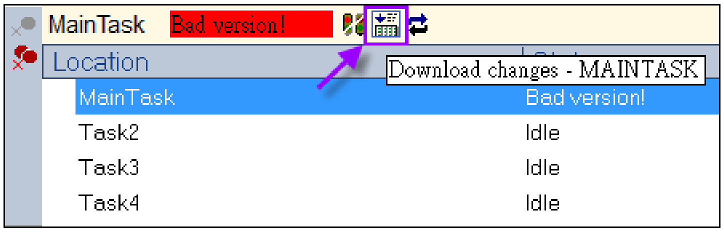


The workbench will indicate that the logic code of the application on the runtime is not identical to the code of the workbench.



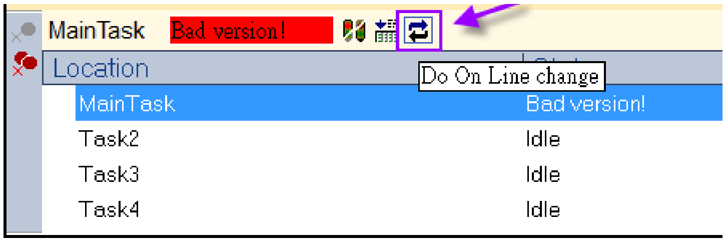
Location	Status
MainTask	Bad version!
Task2	Idle
Task3	Idle
Task4	Idle

Step 8: Download the modified application to the runtime by clicking the *'Download changes'* icon in the output window:



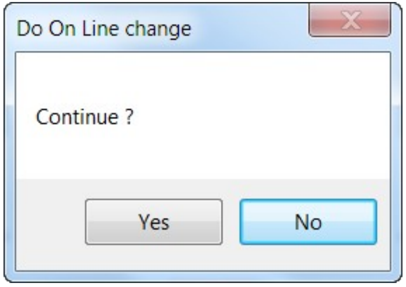
Location	
MainTask	Bad version!
Task2	Idle
Task3	Idle
Task4	Idle

Step 9: Click the Online Change icon to update your PLC application on the fly:



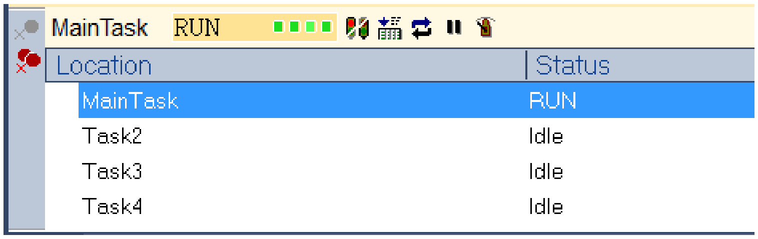
Location	
MainTask	Bad version!
Task2	Idle
Task3	Idle
Task4	Idle

A popup window appears where you have to confirm that you want to update the application. Click Yes.


Do On Line change

Continue ?

The task will switch into run status and the current values of the newly added variables will be shown next to the variable names.



Location	Status
MainTask	RUN
Task2	Idle
Task3	Idle
Task4	Idle

9 Modbus Networking

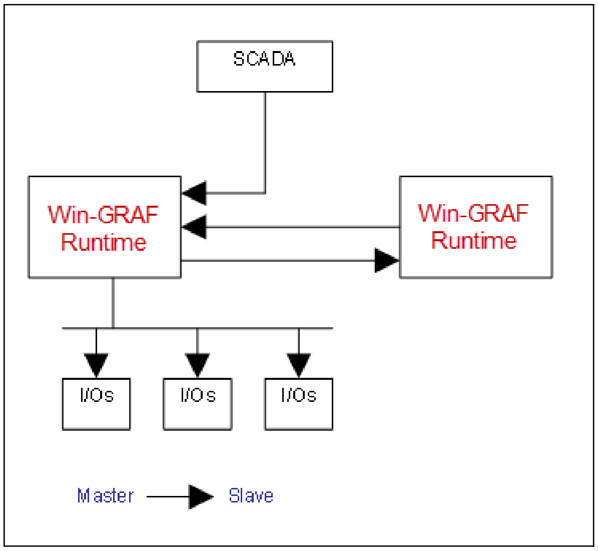
The Win-GRAF runtime includes fully integrated Modbus master and slave functions for enabling Modbus communication on serial link or Ethernet.

The following Modbus function codes are supported:

- read coils
- read bit inputs
- read holding registers
- read input registers
- write 1 coil
- write 1 register
- write N coils
- write N registers

Architecture:
The Win-GRAF runtime can be used either as a server (Modbus slave) or as a client (Modbus master). Both server and client may be active at the same time. This allows multiple applications such as:

- Slave connection to a Modbus master such as a SCADA system.
- Master connection to pilot Modbus I/Os.
- Win-GRAF to Win-GRAF communication for real time exchange of variables (binding).

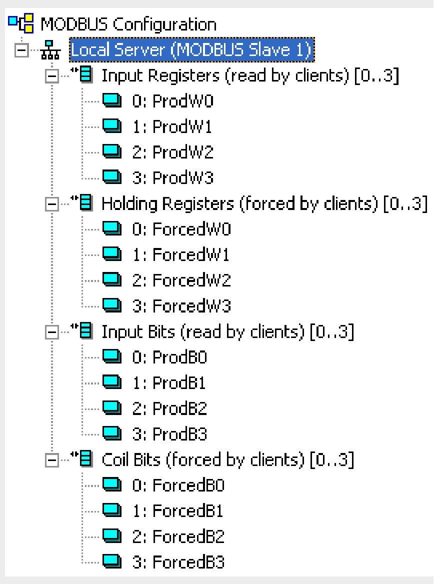


The configuration of slave and master functions is done in the Workbench.

10 Modbus Slave

This chapter describes how to setup the runtime to act as a Modbus slave. Three types of slaves can be installed: Modbus TCP, Modbus RTU and Modbus ASCII.
The Win-GRAF runtime can act as a multiple slave.

The Modbus slave of the Win-GRAF runtime first has to be configured via the workbench in order for the remote Modbus master to access it. The communication layer (Ethernet, serial), the register types and number of registers (data block) of the slave have to be set. Below is a simple example of a slave configuration (Figure 131):



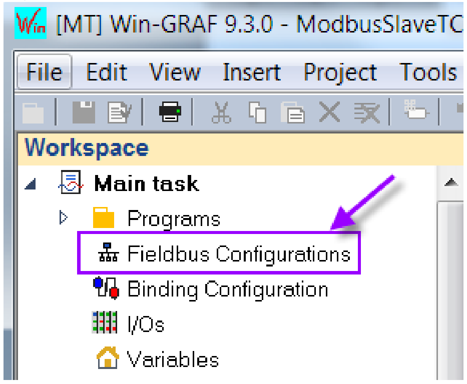
```
MODBUS Configuration
  Local Server (MODBUS Slave 1)
    Input Registers (read by clients) [0..3]
      0: ProdW0
      1: ProdW1
      2: ProdW2
      3: ProdW3
    Holding Registers (forced by clients) [0..3]
      0: ForcedW0
      1: ForcedW1
      2: ForcedW2
      3: ForcedW3
    Input Bits (read by clients) [0..3]
      0: ProdB0
      1: ProdB1
      2: ProdB2
      3: ProdB3
    Coil Bits (forced by clients) [0..3]
      0: ForcedB0
      1: ForcedB1
      2: ForcedB2
      3: ForcedB3
```

Figure 131: Modbus slave configuration

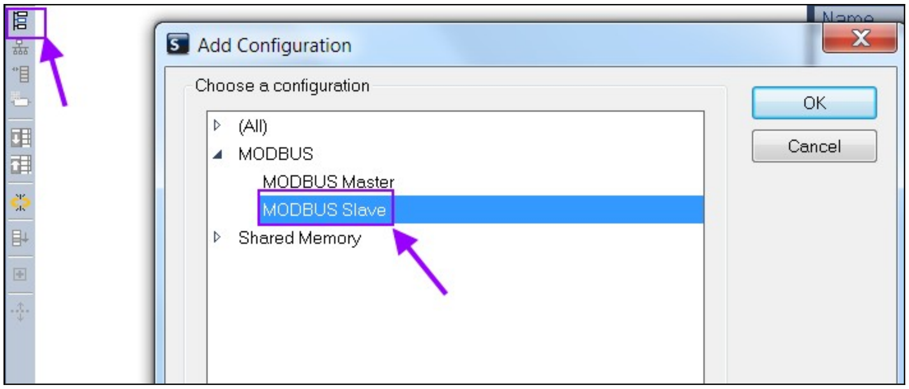
10.1 Slave Data Block Configuration

10.1.1 Selecting Slave

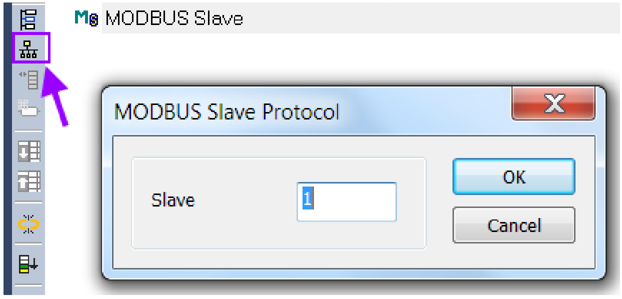
Step 1: Double click the *'Fieldbus Configuration'* tree item in one of the tasks to open the *'IO Drivers'* window.



Step 2: Click the *'Insert Configuration'* button on the left side of the *'IO Drivers'* window and then select the *'MODBUS Slave'* and click *'OK'* to enable a Modbus Slave.

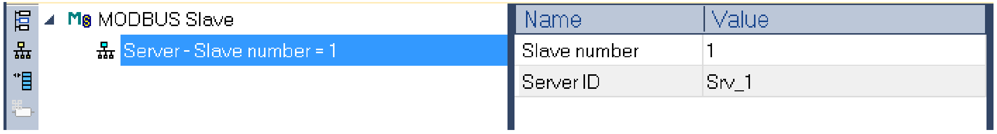


Step 3: Click the *'Insert Master/Port'* button on the left side to set the *'Slave number'* (here the value is *'1'*), and click the *'OK'* button.



Slave: 1

Step 4: If you add more than one slave then assign each slave a ID to easier identify the server in the PLC program (e.g. Srv_1)



Name	Value
Slave number	1
Server ID	Srv_1

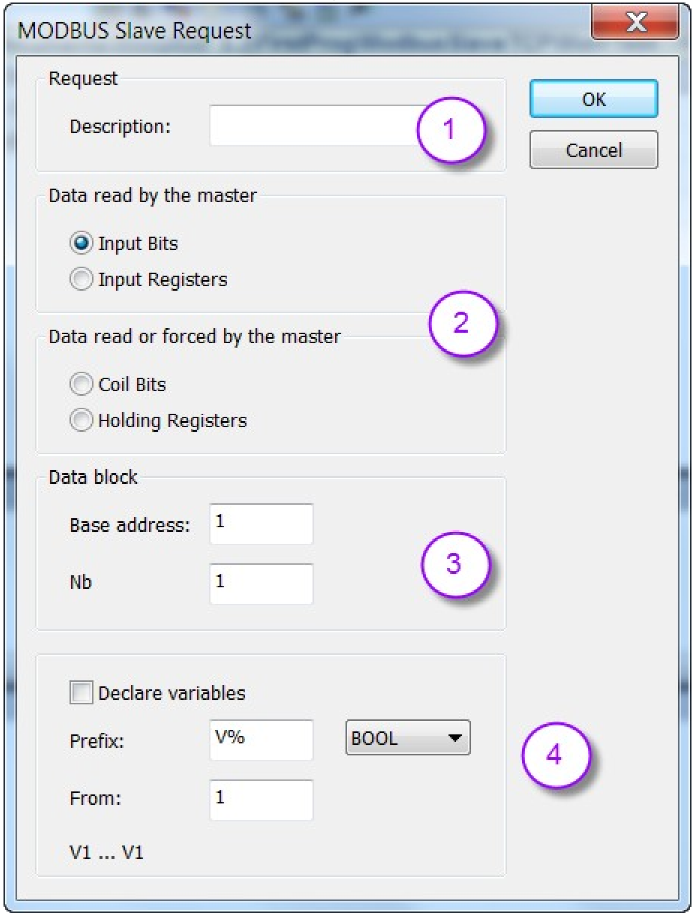
Step 5: Now define the Modbus register and coils supported by the runtime slave This will be described in the following section

10.1.2 Define Slave Register

The following standard Modbus register and coils are supported by Win-GRAF:

- Input register (read by masters)
- Holding register (read/write by master)
- Coils (read/write by master)
- Discrete inputs (read by masters)

Modbus slave register type and size has to be configured via dialog (Figure 132) which can be accessed from the menu (*'Insert'*->Insert Slave/Data block...) .



MODBUS Slave Request

Request

Description:

Data read by the master

● Input Bits

○ Input Registers

Data read or forced by the master

○ Coil Bits

○ Holding Registers

Data block

Base address: 1

Nb 1

□ Declare variables

Prefix: V% BOOL

From: 1

V1 ... V1

Figure 132: Modbus slave register configuration

Description:

1. The description field allows you to shortly describe the purpose of the data block. This field can be left empty
2. First decide whether the master should only have read access or have both read and write access to the data block. In addition decide the data format of the block.

Access	Option	Data type
Read	Input Bits	BOOL
	Input Register	BYTE, INT, DINT, REAL, etc.
Read/Write	Coil Bits	BOOL
	Holding Register	BYTE, INT, DINT, REAL, etc.

3. Set the start address (base address) of the number of register (Nb) of the data block. Depending of the selection made in 2 the register size unit is either BIT or WORD. It is recommended to set the start address to 1. If two blocks of the same register type are added make sure that the start address of the second

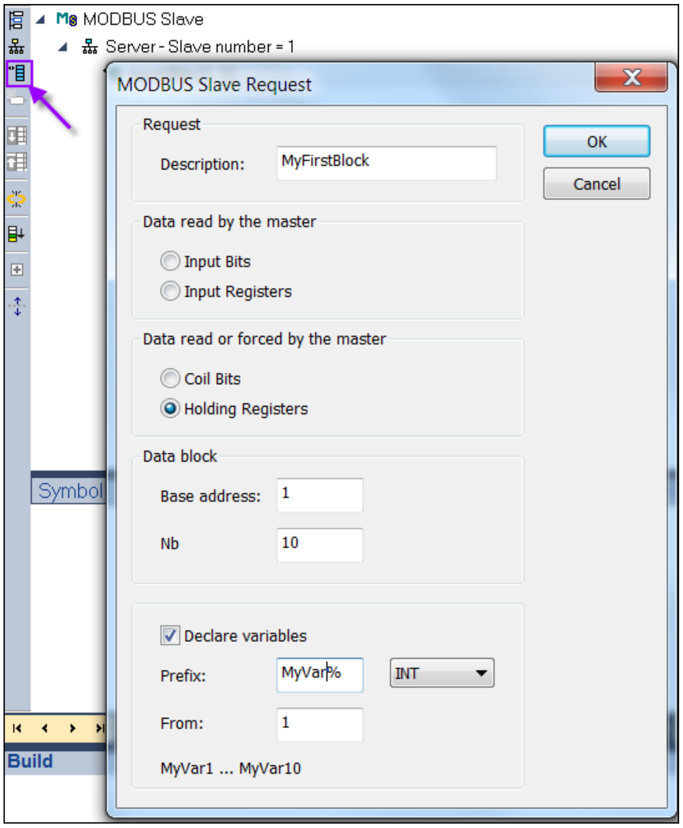
block continuous with the end address of the first block. This allows the master to access the two blocks in one datagram and decrease the number of datagram exchange between master and slave. If the data address requested from the Modbus Master (e.g., the SCADA software) is smaller than the start address or greater than the maximum address (start address + Nb -1) than the slave of the Win-GRAF runtime will not respond.

4. If required the workbench automatically declares new variables and directly map them to the newly created data block.

10.1.2.1 Define Holding Register

The following procedure describes how to add a holding register data block:

1. Add a holding register and map global variable in one step.
   Open the slave configuration window by clicking the *'Inset Slave/Data Block'* icon (Figure 133).
   - Set the name of the block to *'MyFirstBlock'*.
   - Add a data block of the holding register type which holds 10 registers (Nb=10) and the first register address starts (base address) at 1.
   - Declare a number of 10 INT variables and map it to the data block:
     - Select the check box next to *'Declare variables'*
     - Set the name of the variables to *'MyVar%'* whereby the % represents a value which starts from 1 (Set via the *'From:'* text box) and increments by one with each following variable declaration (MyVar1, MyVar2, MyVar3, ...).
     - Select INT as data type.



MODBUS Slave Request

Request

Description: MyFirstBlock

Data read by the master

○ Input Bits
○ Input Registers

Data read or forced by the master

○ Coil Bits
● Holding Registers

Data block

Base address: 1

Nb 10

☑ Declare variables

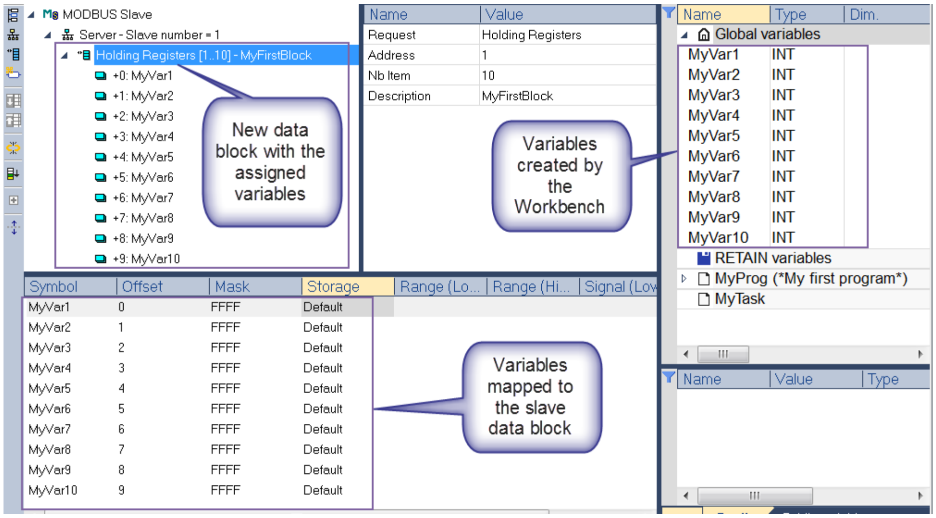
Prefix: MyVar% INT

From: 1

MyVar1 ... MyVar10

Figure 133: Define Slave data block and assign variables

After confirming the setting the workbench adds a new data block with the name *'MyFirstBlock'* to the Modbus slave and new global variables are automatically being declared and assigned to the data block (Figure 134). The PLC program has to use these new variables to access the data block.


New data block with the assigned variables
Variables created by the Workbench
Variables mapped to the slave data block

MODBUS Slave
Server - Slave number = 1
Holding Registers [1..10] - MyFirstBlock
+0: MyVar1
+1: MyVar2
+2: MyVar3
+3: MyVar4
+4: MyVar5
+5: MyVar6
+6: MyVar7
+7: MyVar8
+8: MyVar9
+9: MyVar10

Name	Value
Request	Holding Registers
Address	1
Nb Item	10
Description	MyFirstBlock

Name	Type	Dim.
Global variables		
MyVar1	INT	
MyVar2	INT	
MyVar3	INT	
MyVar4	INT	
MyVar5	INT	
MyVar6	INT	
MyVar7	INT	
MyVar8	INT	
MyVar9	INT	
MyVar10	INT	
RETAIN variables		
MyProg (*My first program*)		
MyTask		

Symbol	Offset	Mask	Storage	Range (Lo...	Range (Hi...	Signal (Low
MyVar1	0	FFFF	Default			
MyVar2	1	FFFF	Default			
MyVar3	2	FFFF	Default			
MyVar4	3	FFFF	Default			
MyVar5	4	FFFF	Default			
MyVar6	5	FFFF	Default			
MyVar7	6	FFFF	Default			
MyVar8	7	FFFF	Default			
MyVar9	8	FFFF	Default			
MyVar10	9	FFFF	Default			

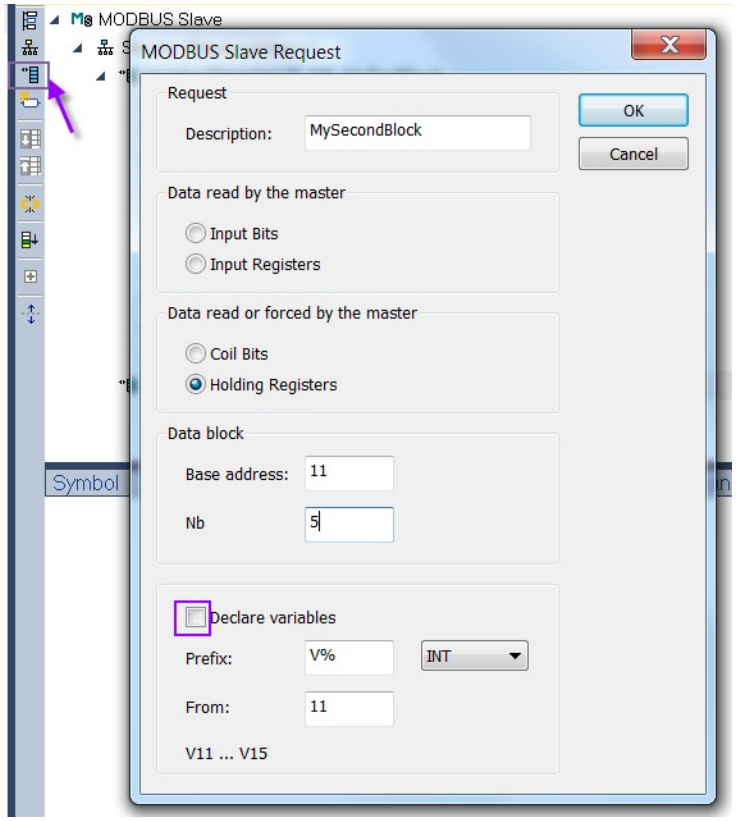
Name	Value	Type

Figure 134: Slave data block with mapped variables

2. Add a holding register and map global variable in several steps

Step 1: Click the *'Insert Slave/Data Block'* icon to open the configuration window.

1. Set the name of the block to *'MySecondBlock'*.
2. Add a data block of the holding register type which holds 5 registers (Nb=5) and the first register address starts (base address) at 11. The register address of the previous block *'MyFirstBlock'* ends with 10 therefore the first address of the second block continuous with 11 in order for the slave to have one consistence address range for the same register type (here: Holding register).
3. We want to declare and map the variable for the holding register data block by ourselves and therefore make sure the check box is unchecked.



Request

Description: MySecondBlock

Data read by the master

○ Input Bits
○ Input Registers

Data read or forced by the master

○ Coil Bits
● Holding Registers

Data block

Base address: 11

Nb 5

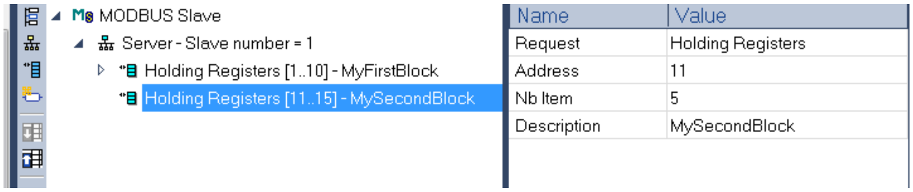
□ Declare variables

Prefix: V% INT

From: 11

V11 ... V15

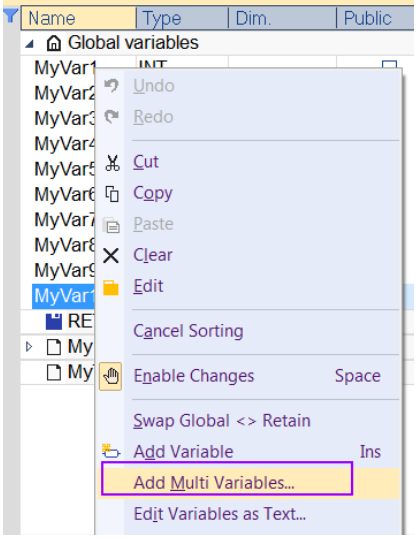
The workbench adds a empty holding register data block with the name *' MySecondBlock'* to the Fieldbus window:


MODBUS Slave
Server - Slave number = 1
Holding Registers [1..10] - MyFirstBlock
Holding Registers [11..15] - MySecondBlock

Name	Value
Request	Holding Registers
Address	11
Nb Item	5
Description	MySecondBlock

Step 2: Declare 5 global variables of INT type:

1. Right click a global variable and select *'Add Multi Variables ...'*



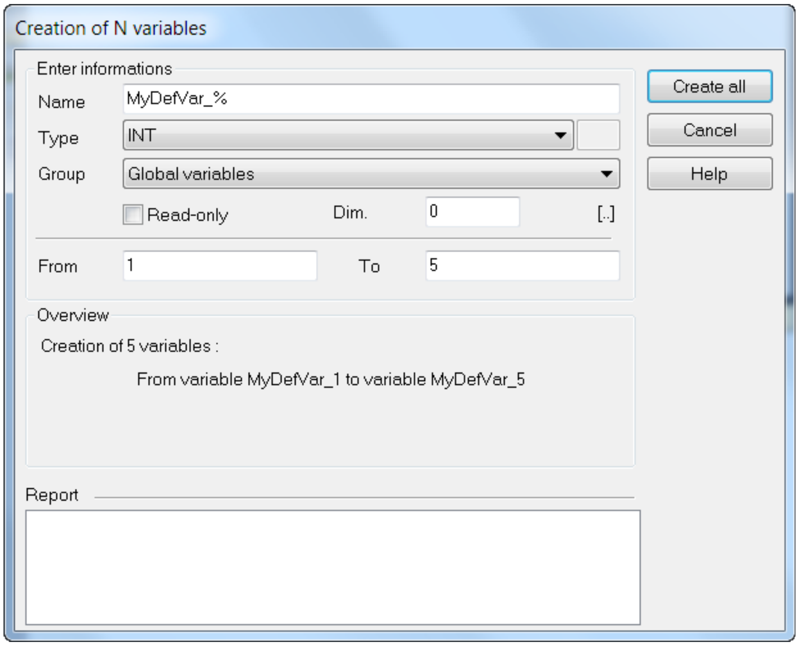
2.

i. Enter the variable name. The % symbol at the end of the name indicates that a number will be attached to the variable name at an incremental order. The start number has to be entered in the *'From'* and the end number in the *'To'* text box.

ii. Select the variable type to declare (here: INT)

iii. Declare the variables as global

iv. Click *'Create all'* to declare the variables and *'Cancel'* to close the window



Creation of N variables

Enter informations

Name: MyDefVar_%

Type: INT

Group: Global variables

□ Read-only   Dim. 0   [.]

From 1   To 5

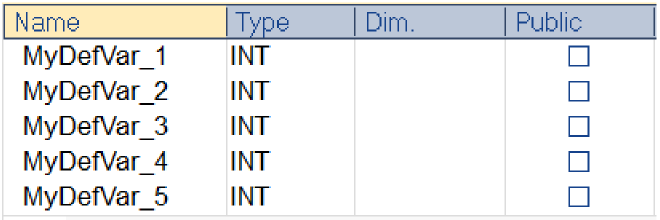
Overview

Creation of 5 variables :

From variable MyDefVar_1 to variable MyDefVar_5

Report

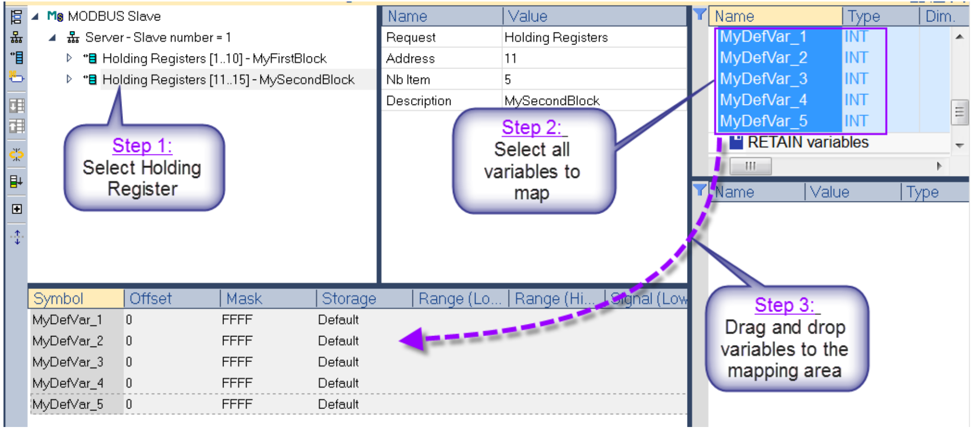
The workbench adds the declared variables to the variable editor:



Name	Type	Dim.	Public
MyDefVar_1	INT		□
MyDefVar_2	INT		□
MyDefVar_3	INT		□
MyDefVar_4	INT		□
MyDefVar_5	INT		□

Step 3: Map the newly declared variables to the Modbus register data block

1. Drag and drop the variables to the mapping area



MODBUS Slave
Server - Slave number = 1
Holding Registers [1..10] - MyFirstBlock
Holding Registers [11..15] - MySecondBlock
Step 1: Select Holding Register
Step 2: Select all variables to map
Step 3: Drag and drop variables to the mapping area
RETAIN variables

Name	Value
Request	Holding Registers
Address	11
Nb Item	5
Description	MySecondBlock

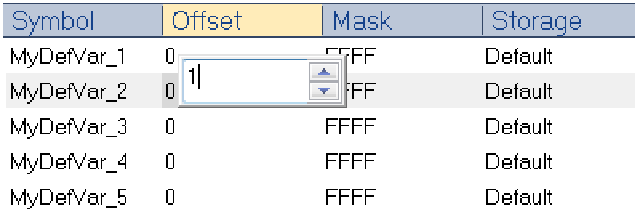
Name	Type	Dim.
MyDefVar_1	INT	
MyDefVar_2	INT	
MyDefVar_3	INT	
MyDefVar_4	INT	
MyDefVar_5	INT	

Symbol	Offset	Mask	Storage	Range (Lo...	Range (Hi...	Signal (Low
MyDefVar_1	0	FFFF	Default			
MyDefVar_2	0	FFFF	Default			
MyDefVar_3	0	FFFF	Default			
MyDefVar_4	0	FFFF	Default			
MyDefVar_5	0	FFFF	Default			

The workbench set the data block offset of each variable by default to zero which means there the variable memory are overlapping as they share the same memory area.

2. Assign each variable to a offset position in the data block.

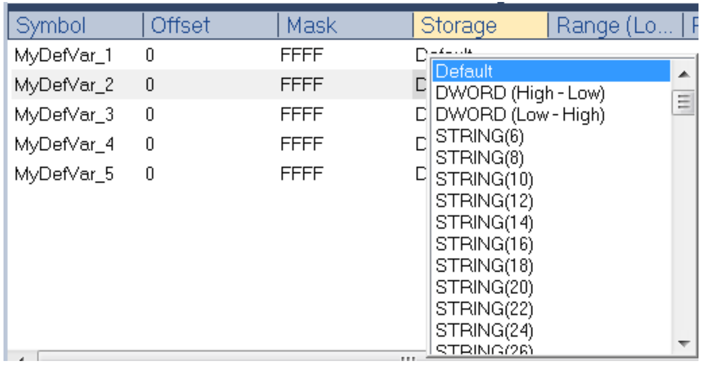
Double click the offset column next to the variable and assign is a offset position. The first variable should start at the offset at 0, the seconds at 1, etc..



Symbol	Offset	Mask	Storage
MyDefVar_1	0	FFFF	Default
MyDefVar_2	0 1	FFFF	Default
MyDefVar_3	0	FFFF	Default
MyDefVar_4	0	FFFF	Default
MyDefVar_5	0	FFFF	Default

Note:

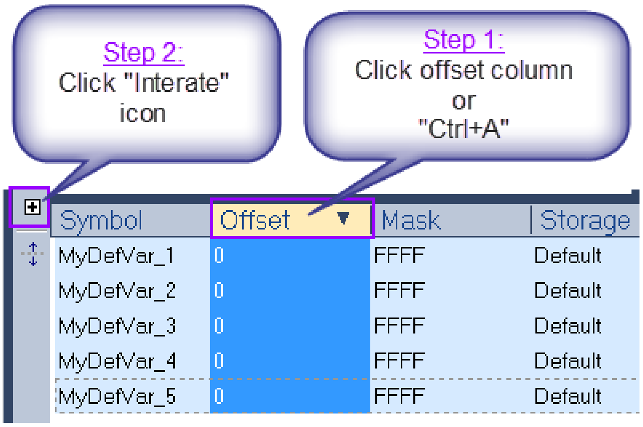
- The memory size of one Modbus holding register is 16 bits (INT), this means the offset distance is two bytes. The *'Default'* entry in the *'Storage'* column indicates that the variable needs one holding register (16 bits memory) in the data block.
- If a 32-bits or greater data type (e.g. DINT, REAL, STRING) has been assigned to the data block then the storage size has to be adjusted.
  Example:
  For a DINT or REAL variable select DWORD for storage
  For a STRING variable select the string size as storage



Symbol	Offset	Mask	Storage	Range (Lo...
MyDefVar_1	0	FFFF	Default	
MyDefVar_2	0	FFFF	D	
MyDefVar_3	0	FFFF	D	
MyDefVar_4	0	FFFF	D	
MyDefVar_5	0	FFFF	D	

Tip:
The Workbench provides a function which allows you to quickly assign memory offsets for each variable.

- Click the head line of the offset column to select all its entries.

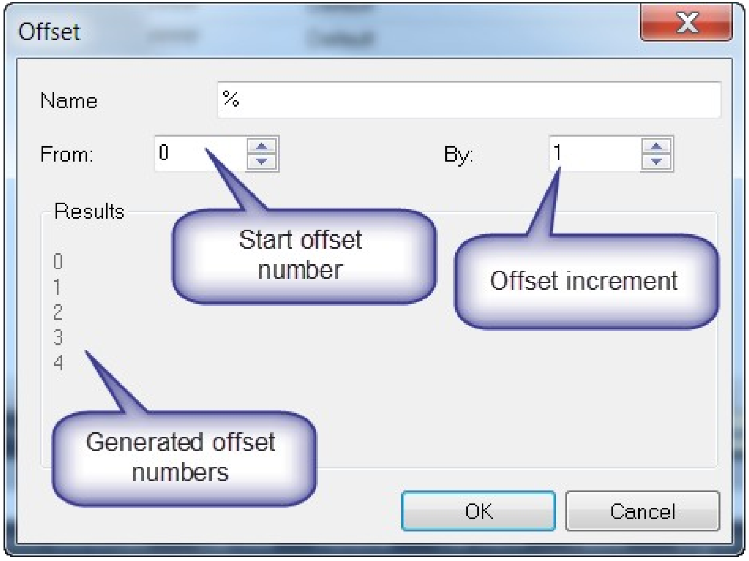


Step 2: Click "Interate" icon

Step 1: Click offset column or "Ctrl+A"

Symbol	Offset	Mask	Storage
MyDefVar_1	0	FFFF	Default
MyDefVar_2	0	FFFF	Default
MyDefVar_3	0	FFFF	Default
MyDefVar_4	0	FFFF	Default
MyDefVar_5	0	FFFF	Default

- Click the *'Iterate Property'* button on the left to open the *'Offset'* editor. Enter the start offset number and the offset increment value. Confirm the setting with entering *'OK'*.



Offset

Name %

From: 0

By: 1

Results

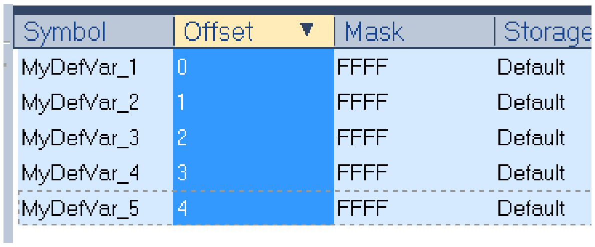
0
1
2
3
4

Start offset number

Offset increment

Generated offset numbers

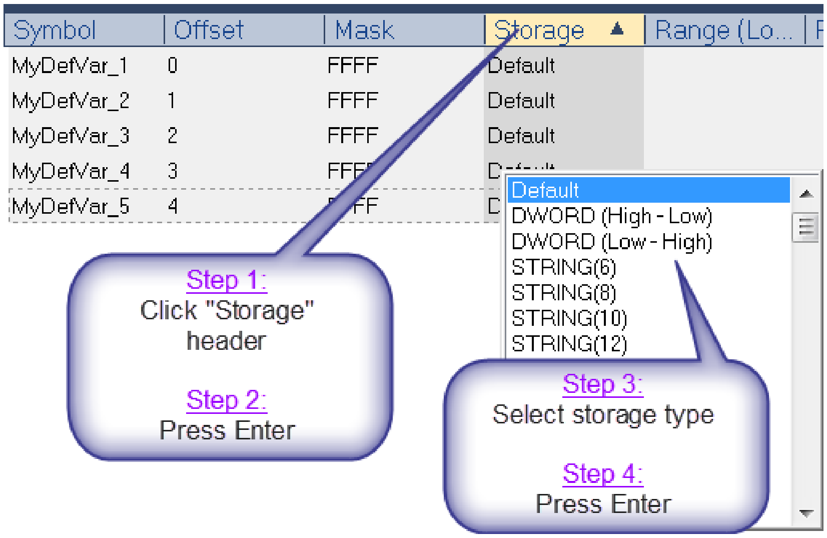
All the variables are assigned a offset value in sequential order:



Symbol	Offset	Mask	Storage
MyDefVar_1	0	FFFF	Default
MyDefVar_2	1	FFFF	Default
MyDefVar_3	2	FFFF	Default
MyDefVar_4	3	FFFF	Default
MyDefVar_5	4	FFFF	Default

The workbench allows the user to set the storage for all variable in one action:

- Click the *'Storage'* header to select the entire column and then press *'Enter'* key to display a drop-down menu.



Symbol	Offset	Mask	Storage ▲	Range (Lo...	F
MyDefVar_1	0	FFFF	Default		
MyDefVar_2	1	FFFF	Default		
MyDefVar_3	2	FFFF	Default		
MyDefVar_4	3	FFFF	Default		
MyDefVar_5	4	FFFF	D		

Step 1: Click "Storage" header

Step 2: Press Enter

Step 3: Select storage type

Step 4: Press Enter

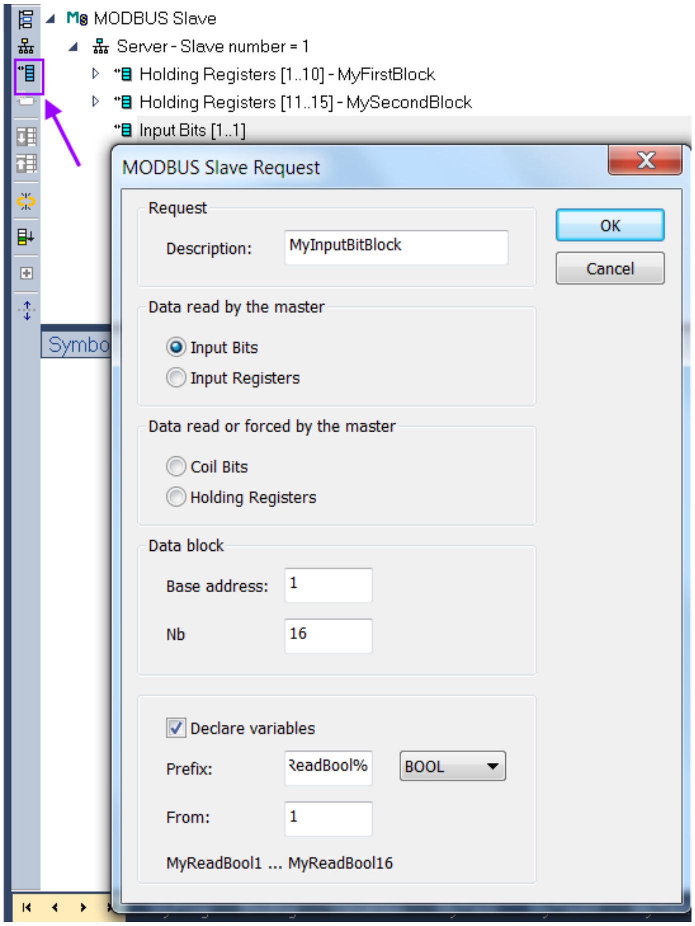
- Select a storage type and press enter

10.1.2.2 Define Input Bit Data Block

The following procedure describes how to add a input bit data block which can only be read by the master.

Open the slave configuration window by clicking the *'Inset Slave/Data Block'* icon (Figure 135).

- Set the name of the block to *'MyInputBitBlock'*.
- Add a data block which holds 16 input bit data types (Nb=16). The address starts (base address) at 1 for the first bit entry.
- Declare a number of 16 BOOL variables and map it to the data block:
    - Activate the check box next to *'Declare variables'*
    - Set the name of the variables to *'MyReadBool%'* whereby the % represents a value which starts from 1 (set via edit box *'From:'*) and increments by one for variable declaration (MyReadBool 1, MyReadBool 2, MyReadBool 3, ...).
    - Confirm the setting by clicking *'OK'*



- MODBUS Slave
  - Server - Slave number = 1
    - Holding Registers [1..10] - MyFirstBlock
    - Holding Registers [11..15] - MySecondBlock
    - Input Bits [1..1]

MODBUS Slave Request

Request

Description: MyInputBitBlock

Data read by the master

● Input Bits

○ Input Registers

Data read or forced by the master

○ Coil Bits

○ Holding Registers

Data block

Base address: 1

Nb 16

☑ Declare variables

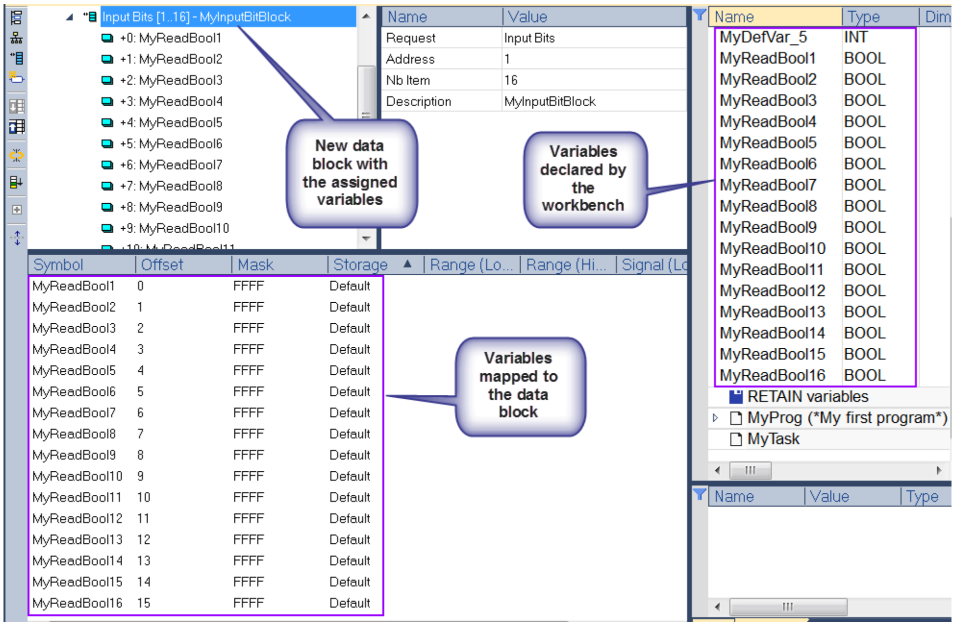
Prefix: ReadBool% BOOL

From: 1

MyReadBool1 ... MyReadBool16

Figure 135: Define Slave data block and assign variables

The workbench adds a new input bit data block with the name *'MyInputBitBlock'* to the Modbus slave Fieldbus configuration area. 16 new BOOL variables are decaled and mapped to the data block (Figure 136). The PLC program can access the Modbus data block via the newly created variables.



Input Bits [1..16] - MyInputBitBlock
- +0: MyReadBool1
- +1: MyReadBool2
- +2: MyReadBool3
- +3: MyReadBool4
- +4: MyReadBool5
- +5: MyReadBool6
- +6: MyReadBool7
- +7: MyReadBool8
- +8: MyReadBool9
- +9: MyReadBool10
- +10: MyReadBool11

Name	Value
Request	Input Bits
Address	1
Nb Item	16
Description	MyInputBitBlock

Symbol	Offset	Mask	Storage	Range (Lo...	Range (Hi...	Signal (Lo
MyReadBool1	0	FFFF	Default			
MyReadBool2	1	FFFF	Default			
MyReadBool3	2	FFFF	Default			
MyReadBool4	3	FFFF	Default			
MyReadBool5	4	FFFF	Default			
MyReadBool6	5	FFFF	Default			
MyReadBool7	6	FFFF	Default			
MyReadBool8	7	FFFF	Default			
MyReadBool9	8	FFFF	Default			
MyReadBool10	9	FFFF	Default			
MyReadBool11	10	FFFF	Default			
MyReadBool12	11	FFFF	Default			
MyReadBool13	12	FFFF	Default			
MyReadBool14	13	FFFF	Default			
MyReadBool15	14	FFFF	Default			
MyReadBool16	15	FFFF	Default			

Name	Type	Dim
MyDefVar_5	INT	
MyReadBool1	BOOL	
MyReadBool2	BOOL	
MyReadBool3	BOOL	
MyReadBool4	BOOL	
MyReadBool5	BOOL	
MyReadBool6	BOOL	
MyReadBool7	BOOL	
MyReadBool8	BOOL	
MyReadBool9	BOOL	
MyReadBool10	BOOL	
MyReadBool11	BOOL	
MyReadBool12	BOOL	
MyReadBool13	BOOL	
MyReadBool14	BOOL	
MyReadBool15	BOOL	
MyReadBool16	BOOL	

RETAIN variables
MyProg (*My first program*)
MyTask

Name	Value	Type

New data block with the assigned variables

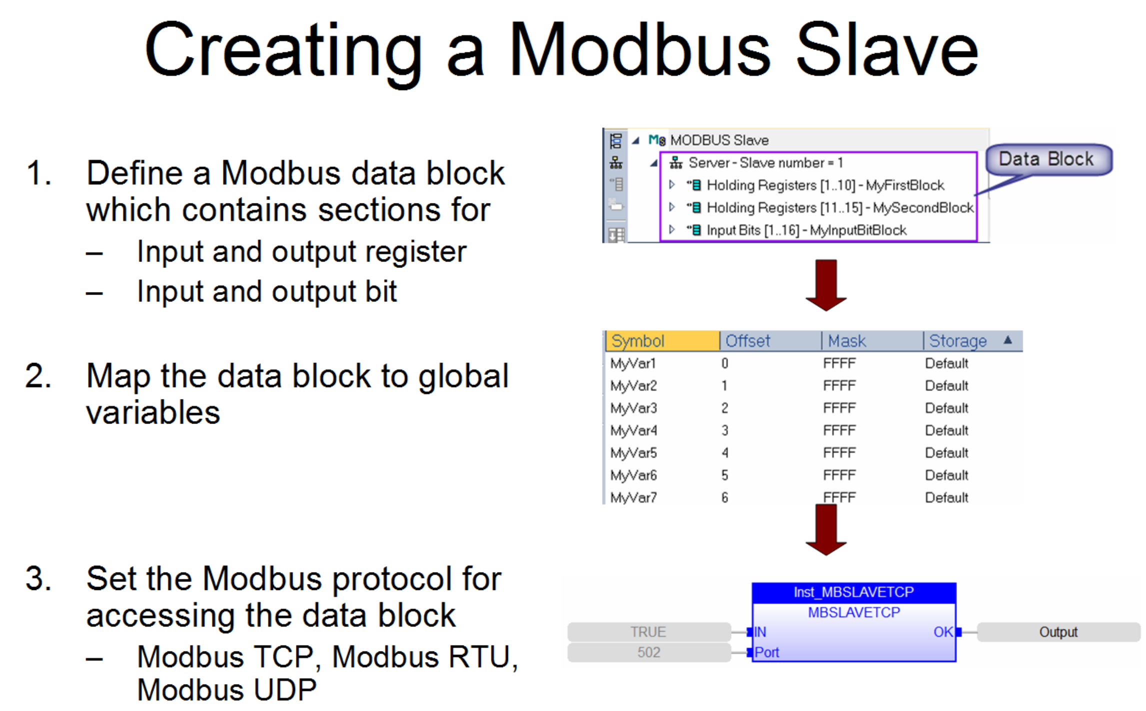
Variables declared by the workbench

Variables mapped to the data block

Figure 136: Slave data block with mapped variables

10.2 Slave Type Configuration

The Modbus data block configuration done in the previous section defines the memory size and structure of the slave. This chapter explains how to create a communication interface through which the Master can exchange data with the slave data block (Figure 137). Win-GRAF provides Modbus slave function blocks which handles the communication between master and the slave data block by processing the request received from the master. If the master request the content of the slave data block then the slave function block reads the data from the data block and write it to the response data frame to the master. Function blocks are provided which supports Ethernet and serial communication (Figure 138).



Creating a Modbus Slave

1. Define a Modbus data block which contains sections for
   - Input and output register
   - Input and output bit
2. Map the data block to global variables
3. Set the Modbus protocol for accessing the data block
   - Modbus TCP, Modbus RTU, Modbus UDP

MODBUS Slave
Server - Slave number = 1
Holding Registers [1..10] - MyFirstBlock
Holding Registers [11..15] - MySecondBlock
Input Bits [1..16] - MyInputBitBlock
Data Block

Symbol	Offset	Mask	Storage
MyVar1	0	FFFF	Default
MyVar2	1	FFFF	Default
MyVar3	2	FFFF	Default
MyVar4	3	FFFF	Default
MyVar5	4	FFFF	Default
MyVar6	5	FFFF	Default
MyVar7	6	FFFF	Default

Figure 137: Procedure for creating a Modbus slave

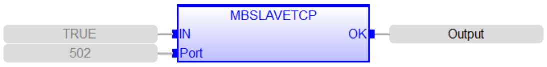
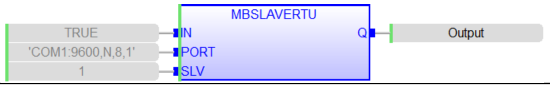
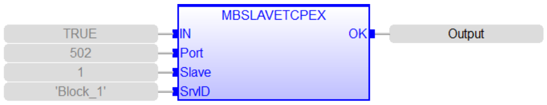
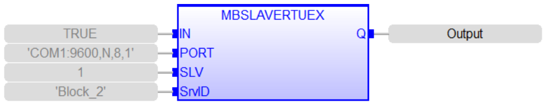
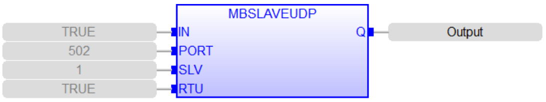
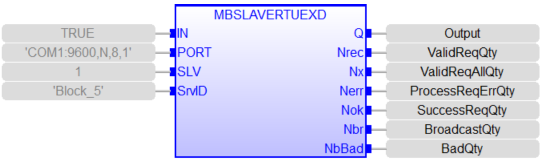
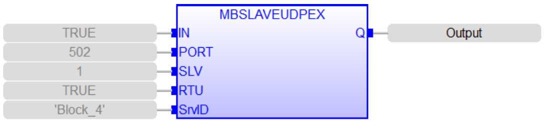
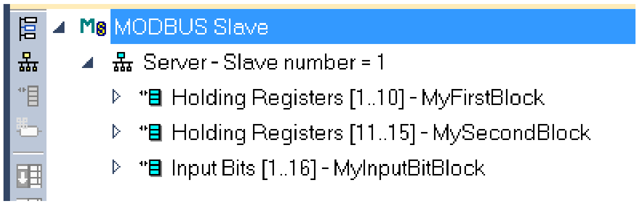
Modbus Slave Function Blocks	
Ethernet	Serial (RS232, RS485)
MBSLAVETCP 	MBSLAVERTU  MBSLAVERTU TRUE 'COM1:9600,N,8,1' 1 IN PORT SLV Q Output
MPSLAVETCPEX  MBSLAVETCPEX TRUE 502 1 'Block_1' IN Port Slave SrvID OK Output	MBSLAVERTUEX  MBSLAVERTUEX TRUE 'COM1:9600,N,8,1' 1 'Block_2' IN PORT SLV SrvID Q Output
MBSLAVEUDP  MBSLAVEUDP TRUE 502 1 TRUE IN PORT SLV RTU Q Output	MBSLAVERTUEXD  MBSLAVERTUEXD TRUE 'COM1:9600,N,8,1' 1 'Block_5' IN PORT SLV SrvID Q Nrec Nx Nerr Nok Nbr NbBad Output ValidReqQty ValidReqAllQty ProcessReqErrQty SuccessReqQty BroadcastQty BadQty
MBSLAVEUDPEX  MBSLAVEUDPEX TRUE 502 1 TRUE 'Block_4' IN PORT SLV RTU SrvID Q Output	

Figure 138: Function blocks for setting the Modbus protocol type

10.2.1 Single Data Block

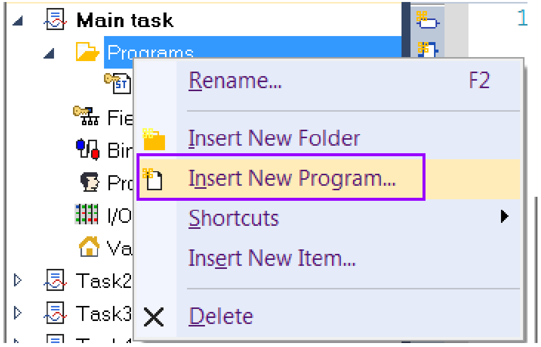
The following procedure shows how to add a Modbus slave function block to the PLC program which will allow the master to access the slave data block which has been created in the previous chapter (Figure 139).

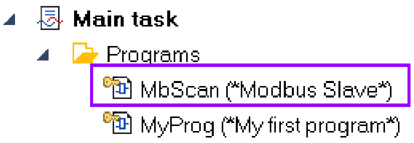


MODBUS Slave
Server - Slave number = 1
Holding Registers [1..10] - MyFirstBlock
Holding Registers [11..15] - MySecondBlock
Input Bits [1..16] - MyInputBitBlock

Figure 139: Slave data block

Step 1: Create a program called *'MbScan'*. Right click in the workspace the *'Programs'* tree item to insert a new program.

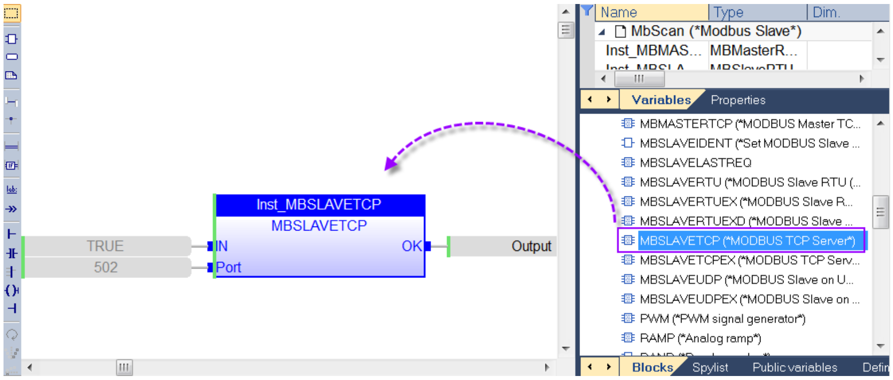




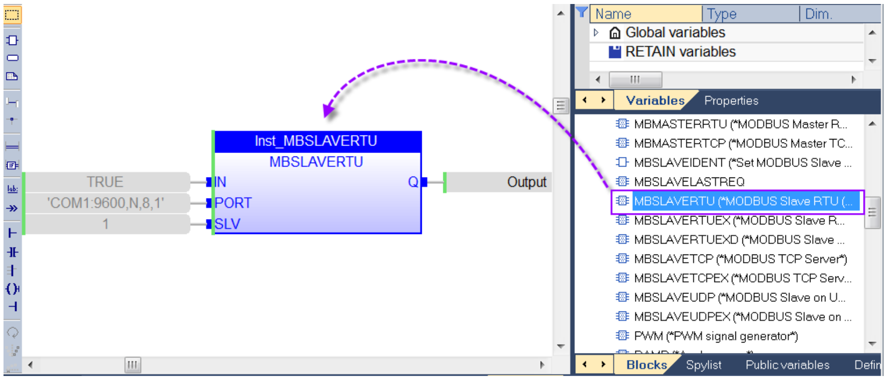
Main task
Programs
MbScan (*Modbus Slave*)
MyProg (*My first program*)

Step 2: Define the slave type. Select one of the following type.

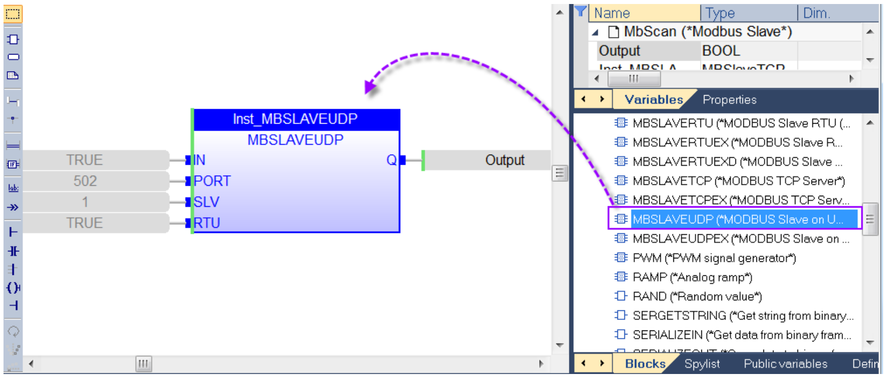
- **Modbus TCP slave:** This allows the Modbus TCP master to access the slave data block. Add a *'MBSLAVETCP'* function block to the *'MbScan'* program. The standard port number for Modbus TCP is 502. A different port number can be selected if it is being supported by the master supports. As long as the IN input is true the *'MBSLAVETCP'* function checks in every program cycle whether a new master command has arrived, process the command and responds to the master.


Inst_MBSLAVETCP
MBSLAVETCP
TRUE
502
IN
Port
OK
Output
Name
Type
Dim.
MbScan (*Modbus Slave*)
Inst_MBMAS...
MBMasterR...
Variables
Properties
MBMASTERTCP (*MODBUS Master TC...
MBSLAVEIDENT (*Set MODBUS Slave ...
MBSLAVELASTREQ
MBSLAVERTU (*MODBUS Slave RTU (...
MBSLAVERTUEX (*MODBUS Slave R...
MBSLAVERTUEXD (*MODBUS Slave ...
MBSLAVETCP (*MODBUS TCP Server*)
MBSLAVETCPEX (*MODBUS TCP Serv...
MBSLAVEUDP (*MODBUS Slave on U...
MBSLAVEUDPEX (*MODBUS Slave on ...
PWM (*PWM signal generator*)
RAMP (*Analog ramp*)
Blocks
Spylist
Public variables
Defin

- **Modbus RTU slave:** A Modbus RTU master can access the data block through one of the serial ports (RS232, RS485, RS422) on which the Win-GRAF runtime is executing. Add a instance of *'MBSLAVERTU'* function block to the *'MbScan'* program. The PORT input need a string of the serial port (*'COM1:9600,N,8,1'*) and the SLV input the Modbus slave number.

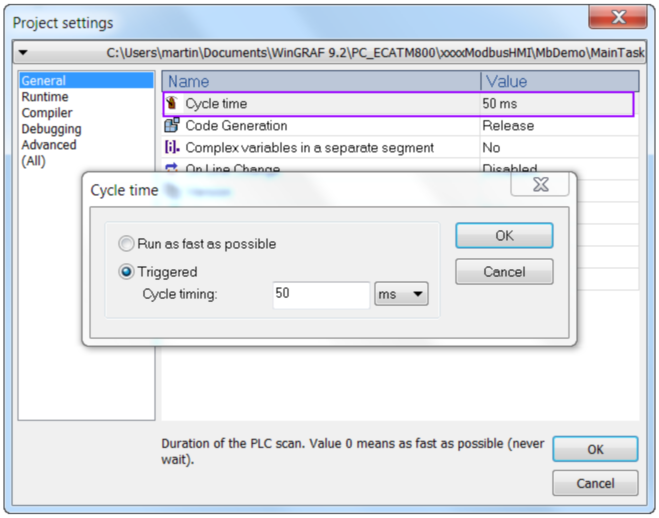

Inst_MBSLAVERTU
MBSLAVERTU
TRUE
'COM1:9600,N,8,1'
1
IN
PORT
SLV
Q
Output
Name
Type
Dim.
Global variables
RETAIN variables
Variables
Properties
MBMASTERRTU (*MODBUS Master R...
MBMASTERTCP (*MODBUS Master TC...
MBSLAVEIDENT (*Set MODBUS Slave ...
MBSLAVELASTREQ
MBSLAVERTU (*MODBUS Slave RTU (...
MBSLAVERTUEX (*MODBUS Slave R...
MBSLAVERTUEXD (*MODBUS Slave ...
MBSLAVETCP (*MODBUS TCP Server*)
MBSLAVETCPEX (*MODBUS TCP Serv...
MBSLAVEUDP (*MODBUS Slave on U...
MBSLAVEUDPEX (*MODBUS Slave on ...
PWM (*PWM signal generator*)
Blocks
Spylist
Public variables
Defin

- **Modbus UDP slave**: Modbus UDP protocol is nearly identical to Modbus TCP except that it runs connectionless on UDP/IP. Unlike TCP which is a guaranteed delivery service, when using UDP the application layer is responsible for any retries required due to possible loss of frames. The advantages of ModbusUDP are that it is in most cases faster than a TCP/IP connection.


Inst_MBSLAVEUDP
MBSLAVEUDP
TRUE
502
1
TRUE
IN
PORT
SLV
RTU
Q
Output
Name
Type
Dim.
MbScan (*Modbus Slave*)
Output
BOOL
Variables
Properties
MBSLAVERTU (*MODBUS Slave RTU (...
MBSLAVERTUEX (*MODBUS Slave R...
MBSLAVERTUEXD (*MODBUS Slave ...
MBSLAVETCP (*MODBUS TCP Server*)
MBSLAVETCPEX (*MODBUS TCP Serv...
MBSLAVEUDP (*MODBUS Slave on U...
MBSLAVEUDPEX (*MODBUS Slave on ...
PWM (*PWM signal generator*)
RAMP (*Analog ramp*)
RAND (*Random value*)
SERGETSTRING (*Get string from binary...
SERIALIZEIN (*Get data from binary fram...
Blocks
Spylist
Public variables
Defin

Step 3: Set the time interval at which the slave function block should check the port (Ethernet or serial buffer) Ethernet port for an incoming Modbus master request or command.

- Set task cycle time: Right click the task name in the tree view and select '*Settings...*'. Make sure the cycle time is shorter than the polling time of the master.



Project settings

C:\Users\martin\Documents\WinGRAF 9.2\PC_ECATM800\xxxxModbusHMI\MbDemo\MainTask

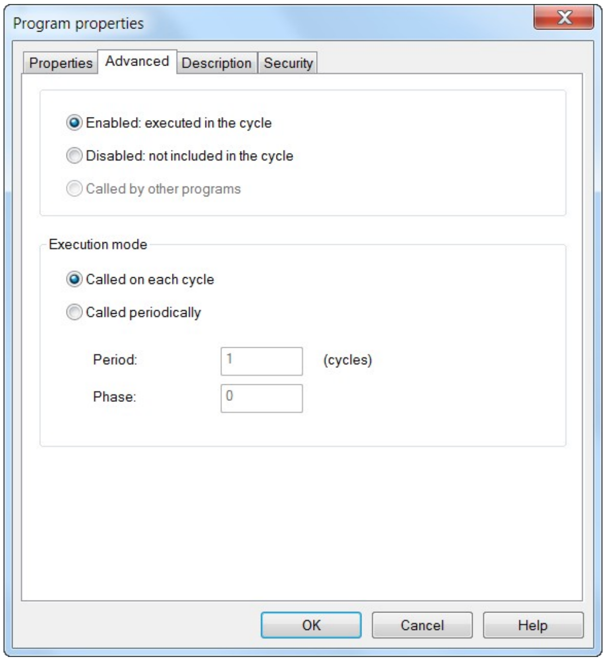
Name	Value
Cycle time	50 ms
Code Generation	Release
Complex variables in a separate segment	No
On Line Change	Disabled

Cycle time

○ Run as fast as possible
● Triggered
Cycle timing: 50 ms

Duration of the PLC scan. Value 0 means as fast as possible (never wait).

- Set execution period: Right click the program name *'MbScan'* in the tree, select *'Properties...'* and click the *'Advance'* tab. Here configure the execution mode.



Program properties

● Enabled: executed in the cycle
○ Disabled: not included in the cycle
○ Called by other programs

Execution mode

● Called on each cycle
○ Called periodically

Period: 1 (cycles)
Phase: 0

1. Example: The Modbus master polling time is 100 ms. If the task cycle time is 100 ms then the *'MbScan'* program should be called in each

task cycle.

2. Example: The Modbus master polling time is 100 ms. If the task cycle time is 10 ms then the *'MbScan'* program should be called periodically at every 5th task cycle.

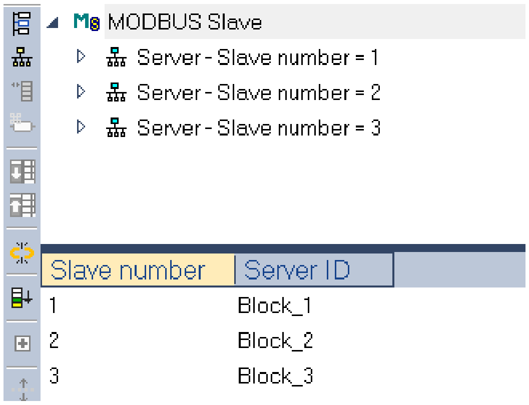
Note:

- It is important that the Modbus slave function block scans the communication buffer for a master command in a regular time interval. Make sure that the function block call interval is shorter than the master polling interval.
- If the Modbus master encounters a respond timeout error then it is necessary to increase the frequency at which the Modbus slave function block is being called in the PLC program. Another alternative is to increase the timeout setting of the Modbus master.
- The standard Modbus TCP/UDP port is 502. Only one Modbus function block instance in the PLC program is allowed to use the 502 port. In order to share Modbus data with other task use the shared memory method. If the Modbus Master supports more than one port number setting (502, 503, 504, etc.) then different ports number can be used for each task.

10.2.2 Multiple Data Block

Win-GRAF is able to process both Modbus TCP and Modbus RTU communication within one application. If the device on which the runtime is installed has multiple serial communication ports, then Win-GRAF allows the programmer to define for each COM port a separate Modbus RTU slave. In general for each communication port (serial) a Modbus slave can be created.

The best way to create multiple slaves for an PLC application is to define for each slave a separated data block in the *'Fieldbus Configuration'* area (Figure 140). The Modbus protocol for each data block is set by adding a Modbus protocol function block (Figure 138) to the programming area and linking the function block to the data block via the input *'SrvID'*.

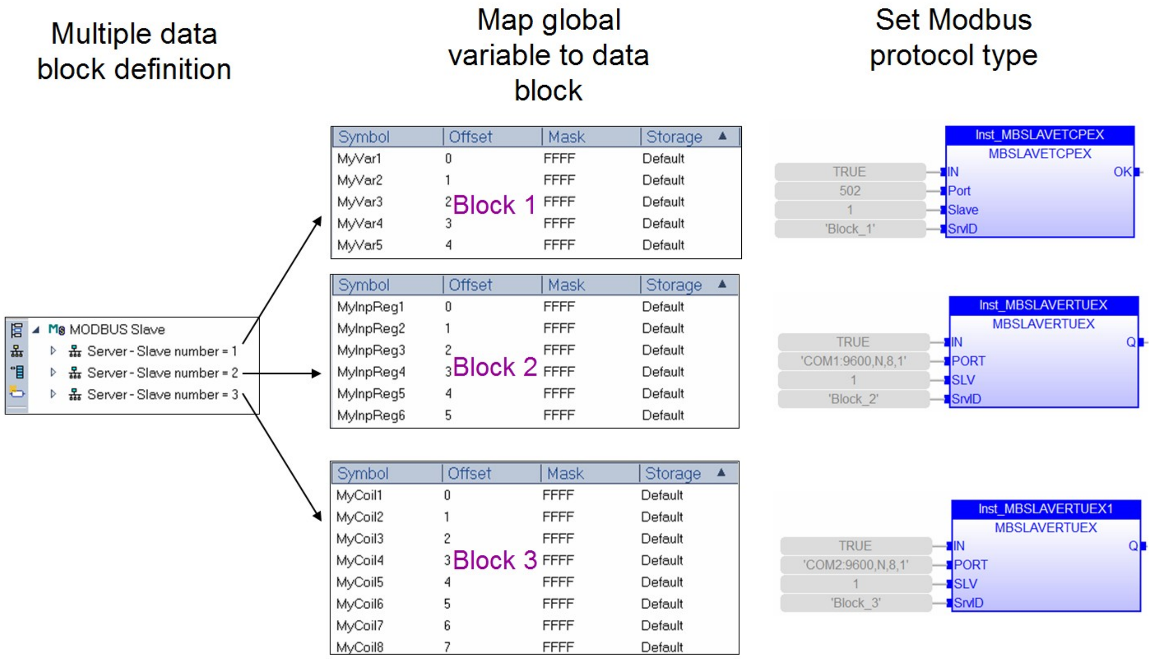


MODBUS Slave
- Server - Slave number = 1
- Server - Slave number = 2
- Server - Slave number = 3

Slave number	Server ID
1	Block_1
2	Block_2
3	Block_3

Figure 140: Multiple data block definitions

Example:



Symbol	Offset	Mask	Storage
MyVar1	0	FFFF	Default
MyVar2	1	FFFF	Default
MyVar3	2	FFFF	Default
MyVar4	3	FFFF	Default
MyVar5	4	FFFF	Default

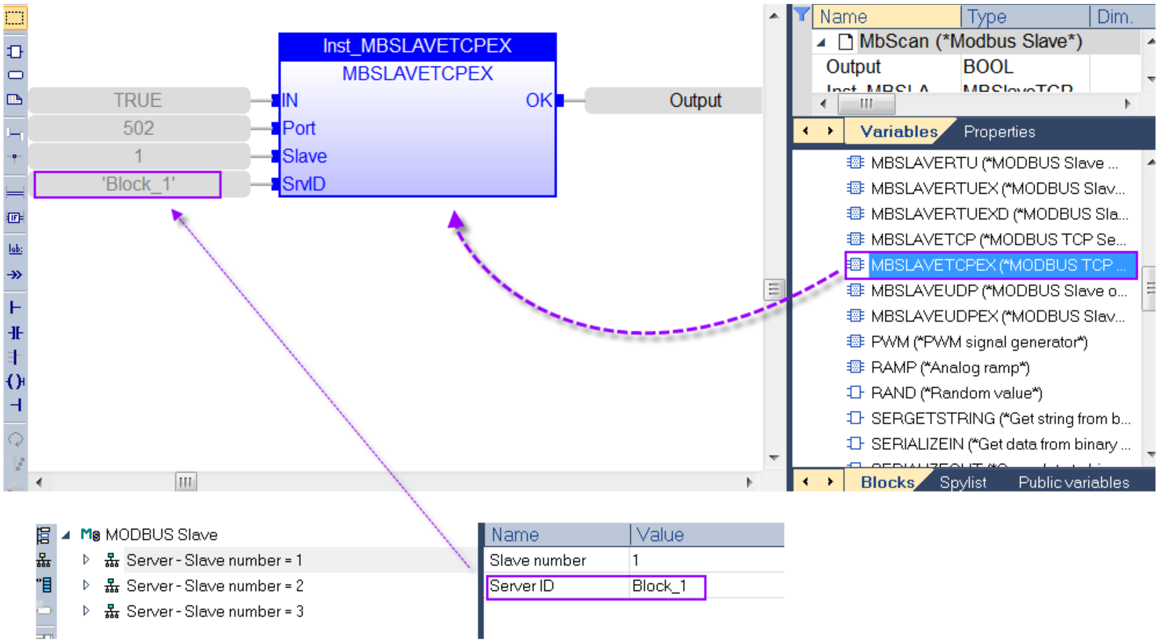
Symbol	Offset	Mask	Storage
MyInpReg1	0	FFFF	Default
MyInpReg2	1	FFFF	Default
MyInpReg3	2	FFFF	Default
MyInpReg4	3	FFFF	Default
MyInpReg5	4	FFFF	Default
MyInpReg6	5	FFFF	Default

Symbol	Offset	Mask	Storage
MyCoil1	0	FFFF	Default
MyCoil2	1	FFFF	Default
MyCoil3	2	FFFF	Default
MyCoil4	3	FFFF	Default
MyCoil5	4	FFFF	Default
MyCoil6	5	FFFF	Default
MyCoil7	6	FFFF	Default
MyCoil8	7	FFFF	Default

Figure 141: Creating multiple Modbus slaves

In this example (Figure 141) the first block will be assigned to a Modbus TCP slave, which means only a Modbus TCP master can access the information stored in the Block 1. The second block will be linked to a Modbus RTU slave which communicates via the serial COM1 port and the third block will be assigned to a Modbus RTU slave which exchanges data via the serial COM2. Extended Modbus protocol function blocks (postfix EX) has to be used to link each data block to a different Modbus protocol.

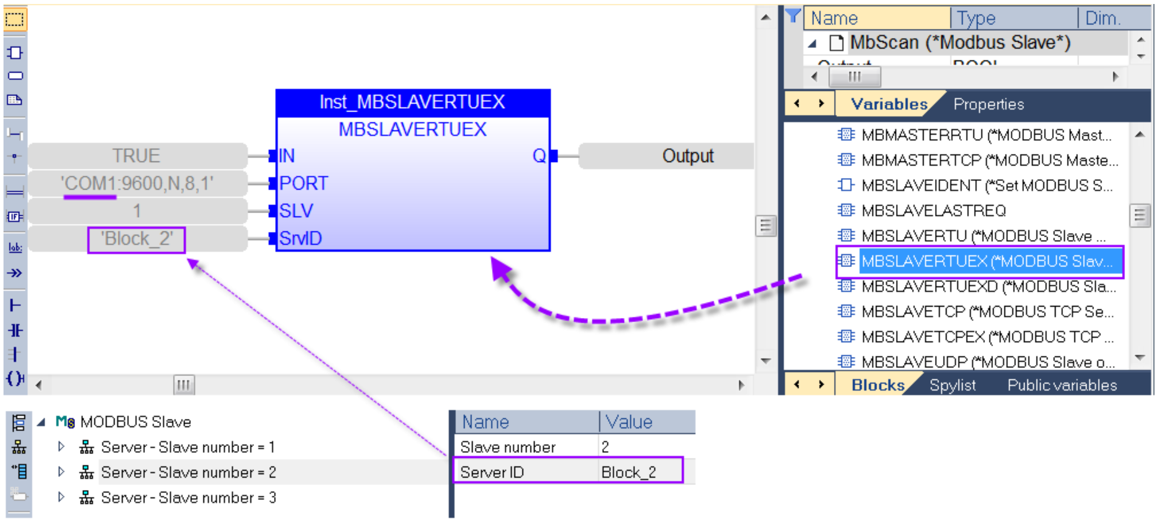
- Set block 1 as a memory area for Modbus TCP slave:


Inst_MBSLAVETCPEX
MBSLAVETCPEX
TRUE
502
1
'Block_1'
IN
Port
Slave
SrvID
OK
Output
Name
Type
Dim.
MbScan (*Modbus Slave*)
Output
BOOL
Variables
Properties
MBSLAVERTU (*MODBUS Slave ...
MBSLAVERTUEX (*MODBUS Slav...
MBSLAVERTUEXD (*MODBUS Sla...
MBSLAVETCP (*MODBUS TCP Se...
MBSLAVETCPEX (*MODBUS TCP ...
MBSLAVEUDP (*MODBUS Slave o...
MBSLAVEUDPEX (*MODBUS Slav...
PWM (*PWM signal generator*)
RAMP (*Analog ramp*)
RAND (*Random value*)
SERGETSTRING (*Get string from b...
SERIALIZEIN (*Get data from binary ...
Blocks
Spylist
Public variables
MODBUS Slave
Server - Slave number = 1
Server - Slave number = 2
Server - Slave number = 3

Name	Value
Slave number	1
Server ID	Block_1

Figure 142: Configure data block 1 as Modbus TCP server

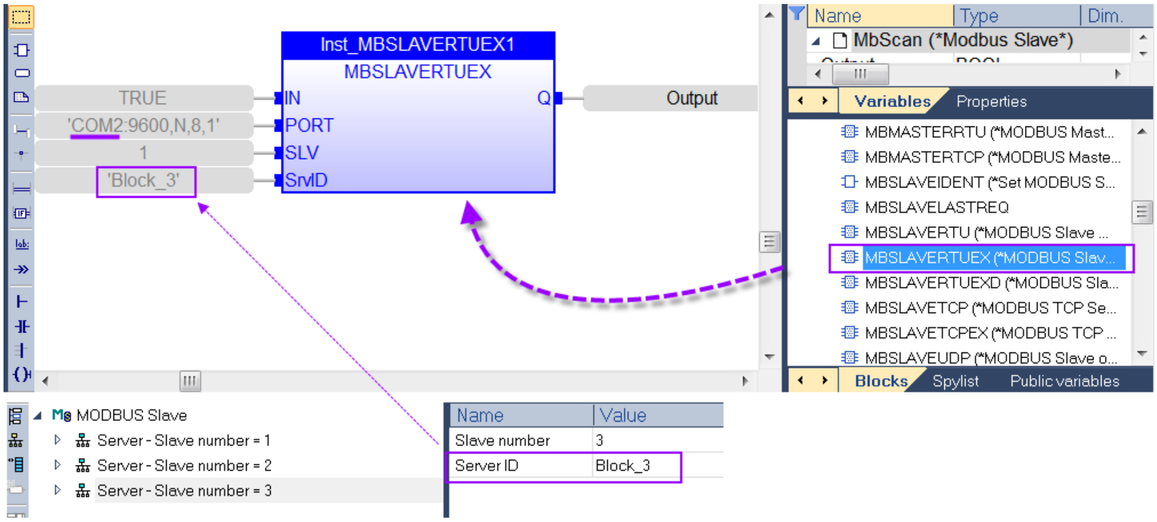
- Configure data block 2 as a Modbus RTU slave with slave number 1 and communication via COM1 port (Figure 143):


Inst_MBSLAVERTUEX
MBSLAVERTUEX
TRUE
'COM1:9600,N,8,1'
1
'Block_2'
IN
PORT
SLV
SrvID
Q
Output
Name
Type
Dim.
MbScan (*Modbus Slave*)
Variables
Properties
MBMASTERRTU (*MODBUS Mast...
MBMASTERTCP (*MODBUS Maste...
MBSLAVEIDENT (*Set MODBUS S...
MBSLAVELASTREQ
MBSLAVERTU (*MODBUS Slave ...
MBSLAVERTUEX (*MODBUS Slav...
MBSLAVERTUEXD (*MODBUS Sla...
MBSLAVETCP (*MODBUS TCP Se...
MBSLAVETCPEX (*MODBUS TCP ...
MBSLAVEUDP (*MODBUS Slave o...
Blocks
Spylist
Public variables
MODBUS Slave
Server - Slave number = 1
Server - Slave number = 2
Server - Slave number = 3

Name	Value
Slave number	2
Server ID	Block_2

Figure 143: Set data block 2 as a Modbus RTU slave

- Configure data block 3 as a Modbus RTU slave with slave number 1 and communication via COM2 port(Figure 144).

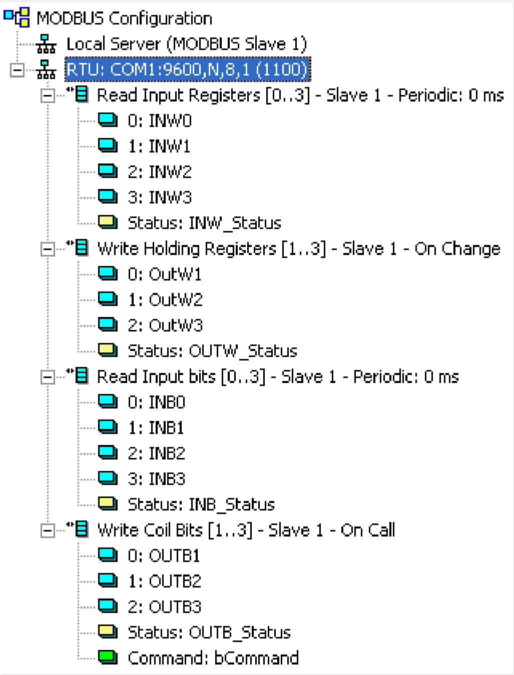

Inst_MBSLAVERTUEX1
MBSLAVERTUEX
TRUE
'COM2:9600,N,8,1'
1
'Block_3'
IN
PORT
SLV
SrvID
Q
Output
Name
Type
Dim.
MbScan (*Modbus Slave*)
Variables
Properties
MBMASTERRTU (*MODBUS Mast...
MBMASTERTCP (*MODBUS Maste...
MBSLAVEIDENT (*Set MODBUS S...
MBSLAVELASTREQ
MBSLAVERTU (*MODBUS Slave ...
MBSLAVERTUEX (*MODBUS Slav...
MBSLAVERTUEXD (*MODBUS Sla...
MBSLAVETCP (*MODBUS TCP Se...
MBSLAVETCPEX (*MODBUS TCP ...
MBSLAVEUDP (*MODBUS Slave o...
Blocks
Spylist
Public variables
MODBUS Slave
Server - Slave number = 1
Server - Slave number = 2
Server - Slave number = 3

Name	Value
Slave number	3
Server ID	Block_3

Figure 144: Set data block 3 as Modbus RTU slave

11 Modbus Master

Win-GRAF Modbus master supports three communication protocols: Modbus TCP, Modbus RTU and Modbus ASCII. Several Modbus masters can be implemented by the same PLC application. Only one master (Modbus RTU, Modbus ASCII) can be created per serial communication port (RS232, RS485, RS422) and several Modbus TCP (Ethernet) masters.



```
MODBUS Configuration
  Local Server (MODBUS Slave 1)
  RTU: COM1:9600,N,8,1 (1100)
    Read Input Registers [0..3] - Slave 1 - Periodic: 0 ms
      0: INW0
      1: INW1
      2: INW2
      3: INW3
      Status: INW_Status
    Write Holding Registers [1..3] - Slave 1 - On Change
      0: OutW1
      1: OutW2
      2: OutW3
      Status: OUTW_Status
    Read Input bits [0..3] - Slave 1 - Periodic: 0 ms
      0: INB0
      1: INB1
      2: INB2
      3: INB3
      Status: INB_Status
    Write Coil Bits [1..3] - Slave 1 - On Call
      0: OUTB1
      1: OUTB2
      2: OUTB3
      Status: OUTB_Status
      Command: bCommand
```

Figure 145: Modbus RTU master

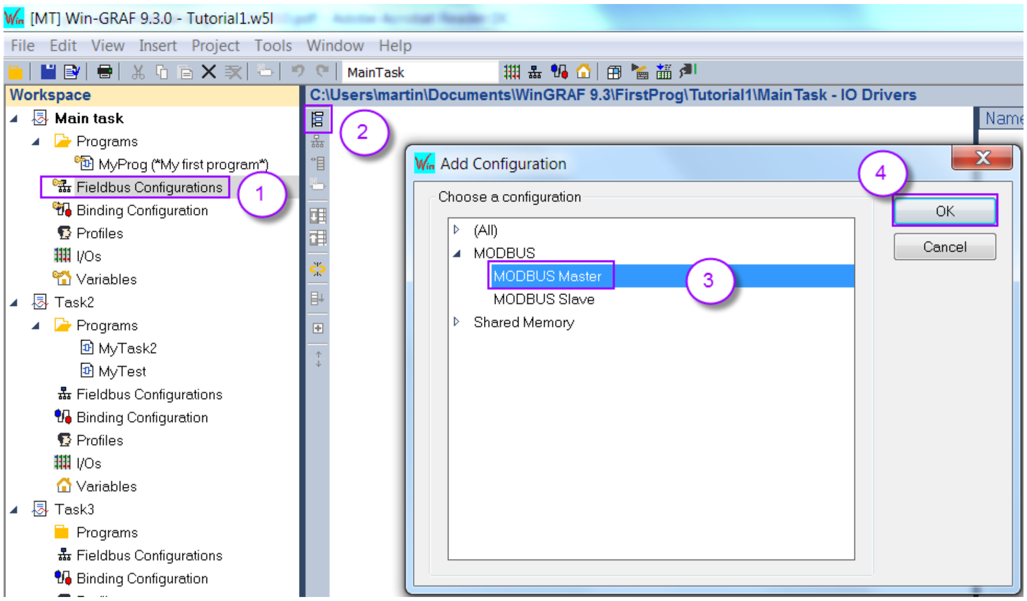
11.1 Modbus RTU/ASCII Master

The following shows how to create a RTU Modbus RTU master which communicates via the serial port COM1 to a network of Modbus slaves.

11.1.1 Configure Communication Interface

Step 1: Select Modbus master as the fieldbus:

1. In the workspace double click the *'Fieldbus Configurations'* item
2. Click the *'Insert Configuration'* button on the left of the *'IO Drivers'* window
3. Select *'MODBUS Master'* from the pop-up window and
4. click *'OK'* to select Modbus Master as a fieldbus.



Step 2: Add a Modbus RTU master to the workbench and configure its communication port:

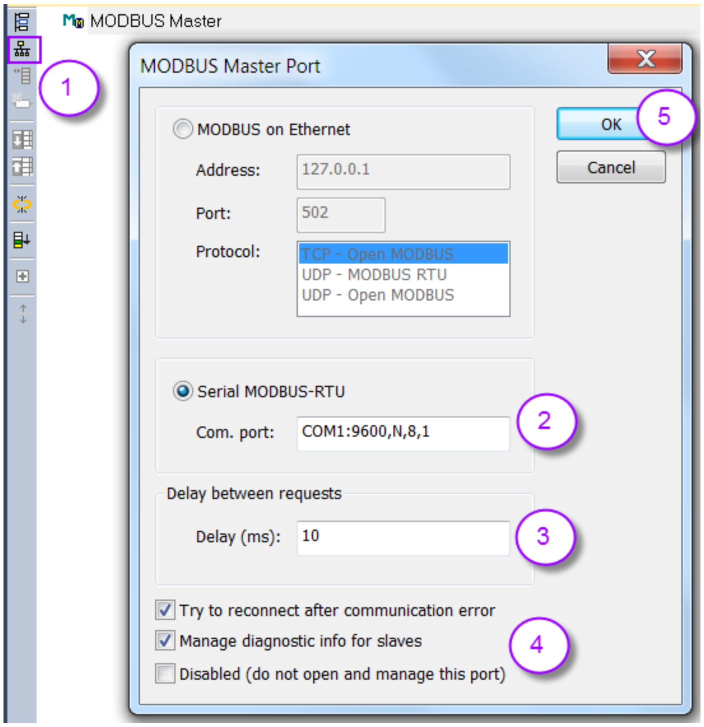
1. Click the *'Insert Master/Port'* button on the left side to open the communication port configuration window

5. Select *'Serial MODBUS-RTU'* and set the serial communication COM port (e.g., *'COM1:9600,N,8,1'*) . For a Modbus ASCII Master the ASCII key word has to be added in front of the COM port setting (e.g. '**ASCII:**COM2:9600,N,8,1'

6. Delay time between a slave responds and next master request (recommended value: 10 ms, it can be 0 to 10000),

7. An option is provided allowing the master to retry opening the port each time the communication fails. Another option at the *'port'* level is to enable the MODBUS stack to record diagnostic information for each slave.

8. and then click *'OK'*.



○ MODBUS on Ethernet

Address: 127.0.0.1

Port: 502

Protocol: TCP - Open MODBUS / UDP - MODBUS RTU / UDP - Open MODBUS

● Serial MODBUS-RTU

Com. port: COM1:9600,N,8,1

Delay between requests

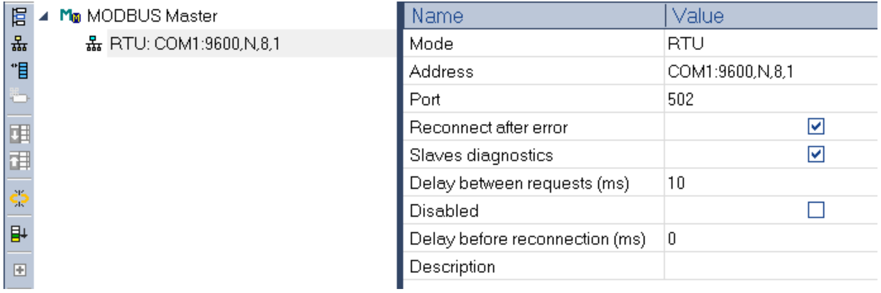
Delay (ms): 10

☑ Try to reconnect after communication error

☑ Manage diagnostic info for slaves

□ Disabled (do not open and manage this port)

The master communication parameters are shown in the property window and can be modified by either double clicking the master or by directly modifying the settings in the property window.



Name	Value
Mode	RTU
Address	COM1:9600,N,8,1
Port	502
Reconnect after error	☑
Slaves diagnostics	☑
Delay between requests (ms)	10
Disabled	□
Delay before reconnection (ms)	0
Description	

Step 3: Add a data block to the master which stores input data received from the slave and output data to be sent to the slave.

1. Click the *'Insert Slave/Data Block'* button on the left side to create a data block.
2. Slave/Unit: Enter the Net-ID of the Slave device. (Example: Remote slave ID is *'1'*).
3. MODBUS Request: Select the Modbus command type (function code) to be used for accessing the slave table (Table 27). Example: <2> Read Input Bits

Type	Function Code	Modbus Request	Description
Read	1	Read coil bits	Read digital output (DO) data

Type	Function Code	Modbus Request	Description
	2	Read input bits	Read digital input (DI) data
	3	Read holding registers	Read analog output (AO) data
	4	Read input registers	Read analog input (AI) data
Write	5	Write single coil bit	Write digital output (DO) data
	6	Write single holding register	Write one analog output (AO) data (16-bit)
	15	Write coil bits	Write multiple digital output (DO) data.
	16	Write Holding Registers	Write multiple analog output (AO) data (16/32 bits)

Table 27: Function code table

9. Configure the data block which holds data from the slave. The data block setup of the master should be an exact representation of the slave table. In other word it should be an image of the slave data block in size and data type. Data read from the slave is being stored to the read section of the data block and data which has been modified in the write section of the data block is being sent to the slave.

- Each data block is identified by a MODBUS slave number, a base address and a number of items/entries (bits or words). The number of items/entries is limited by MODBUS (2000 bits read, 1968 bits forced, 125 words read or 120 words forced).
- **Base address**: Starts from *'1'* by default. It indicates the start address of the Modbus slave table from which the master starts to read. The data read from the start address will be copied to the first position of the master input block data area.
  - ■ Note: If you want to change the *'Base address'*, right-click the *'MODBUS Master'* and then select the *'MODBUS Master Addresses'* to modify the value.
- **Nb items**: The number of slave table entries to read. (Example: the number is set to 16).

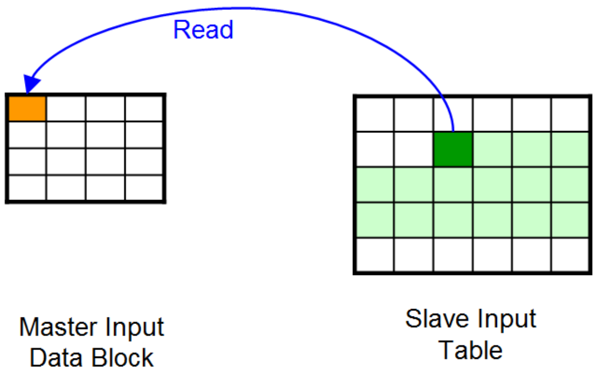
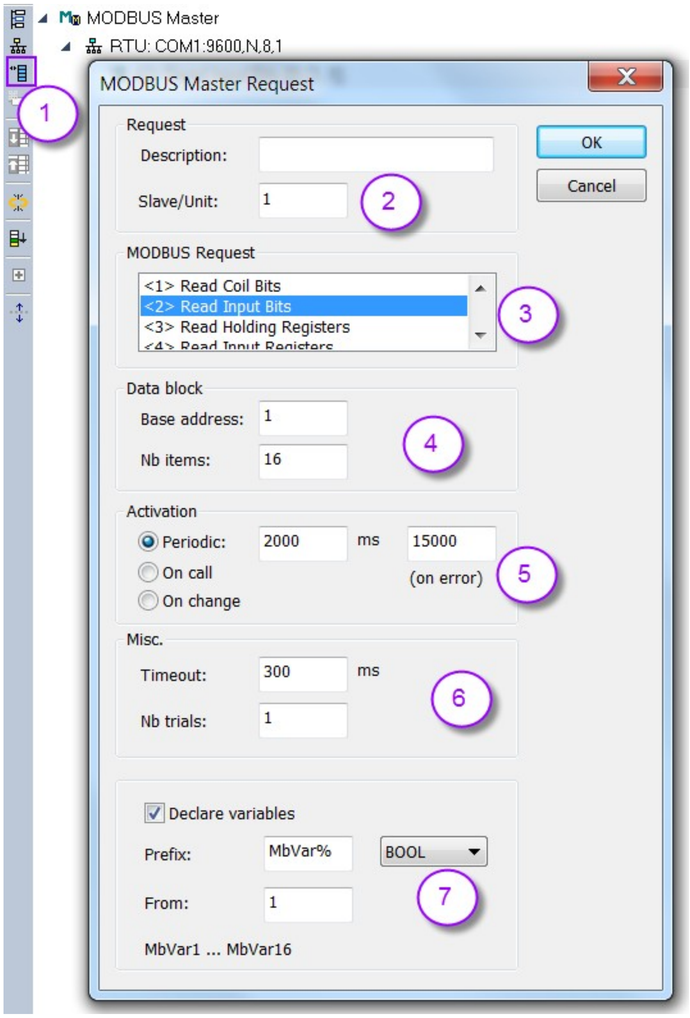


Figure 146: Base address set to 9 and Nb to 16

Example:
Figure 147 creates a input data block which holds 16 input bits and start reading from the slave discrete input table at address 1.

10. **Activation:** Set how the Modbus request is being triggered by the master.
    - **Periodic**: Sending the request periodically. (Example: send once every two seconds.) *'on error'* means the next sending time after an Modbus exception occurred (e.g., 15 seconds).
    - **On call**: The request is activated via a program call by using a variable. If the variable mapped to *'Command (one shot)'* operation turns from false to true a Modbus request is being triggered (Figure 150).
    - **On change**: The request is triggered once a variable mapped to the output area has changed. This option is can only be used for writing and not reading commands.
11. **Timeout**: Set a timeout value. If the slave does not respond to the master request within the timeout period an timeout error will be generated. (The recommended timeout value for the Modbus RTU/ASCII communication is between 200 and 1000 ms.).
    Nb Trials: If the slave fails to respond or the master receives an invalid response, the master will then retry for the configured number of retries before moving on to the next command in the list.
12. **Declare variables**: This option allows the user to declare new variables and automatically map them to the Modbus master data block. Enter a variable name and add the symbol *'%'* at the end of the name and enter the start number for the *'%'* symbol. The workbench replace the *'%'* with a number which increases incrementally for each new variable as indicated at the bottom of the window (Figure 147). The new variables are automatically mapped to the Modbus master data block (Figure 148). Disable this option if variables has already been declared in the variable editor. In this case the variables have to be mapped manually by dragging the variable from the variable editor and dropping it to the mapping area.



MODBUS Master

RTU: COM1:9600,N,8,1

MODBUS Master Request

Request

Description:

Slave/Unit: 1

MODBUS Request

<1> Read Coil Bits

<2> Read Input Bits

<3> Read Holding Registers

<4> Read Input Registers

Data block

Base address: 1

Nb items: 16

Activation

● Periodic: 2000 ms 15000 (on error)

○ On call

○ On change

Misc.

Timeout: 300 ms

Nb trials: 1

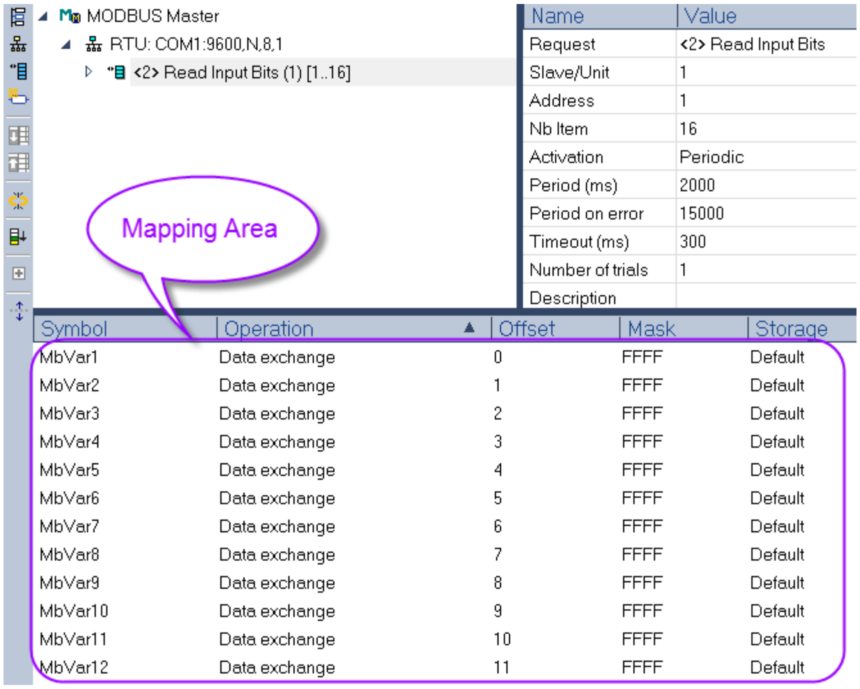
☑ Declare variables

Prefix: MbVar% BOOL

From: 1

MbVar1 ... MbVar16

Figure 147: Modbus Master data block setting



MODBUS Master
RTU: COM1:9600,N,8,1
<2> Read Input Bits (1) [1..16]

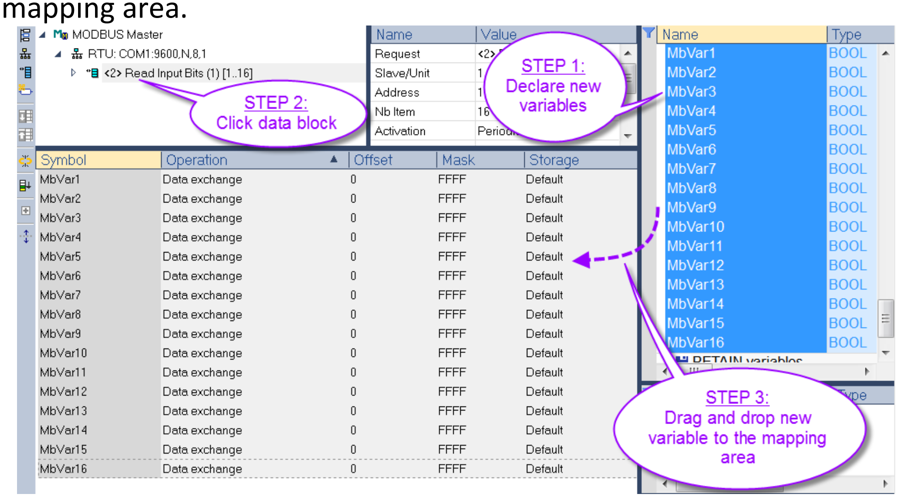
Name	Value
Request	<2> Read Input Bits
Slave/Unit	1
Address	1
Nb Item	16
Activation	Periodic
Period (ms)	2000
Period on error	15000
Timeout (ms)	300
Number of trials	1
Description	

Symbol	Operation	Offset	Mask	Storage
MbVar1	Data exchange	0	FFFF	Default
MbVar2	Data exchange	1	FFFF	Default
MbVar3	Data exchange	2	FFFF	Default
MbVar4	Data exchange	3	FFFF	Default
MbVar5	Data exchange	4	FFFF	Default
MbVar6	Data exchange	5	FFFF	Default
MbVar7	Data exchange	6	FFFF	Default
MbVar8	Data exchange	7	FFFF	Default
MbVar9	Data exchange	8	FFFF	Default
MbVar10	Data exchange	9	FFFF	Default
MbVar11	Data exchange	10	FFFF	Default
MbVar12	Data exchange	11	FFFF	Default

Figure 148: Variable mapped to the Master data block

Step 4: This step needs only to be done if the *'Declare variable'* option (Figure 147) has not been selected. Declare and map variables to the master data block:

1. Declare new variables and drag and drop the variable to the data block mapping area.



MODBUS Master
RTU: COM1:9600,N,8,1
<2> Read Input Bits (1) [1..16]

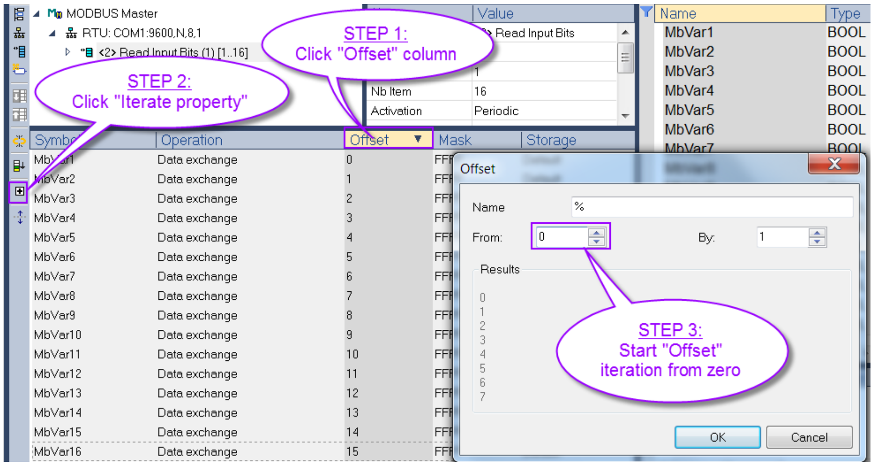
Name	Value
Request	<2>
Slave/Unit	1
Address	1
Nb Item	16
Activation	Period

Symbol	Operation	Offset	Mask	Storage
MbVar1	Data exchange	0	FFFF	Default
MbVar2	Data exchange	0	FFFF	Default
MbVar3	Data exchange	0	FFFF	Default
MbVar4	Data exchange	0	FFFF	Default
MbVar5	Data exchange	0	FFFF	Default
MbVar6	Data exchange	0	FFFF	Default
MbVar7	Data exchange	0	FFFF	Default
MbVar8	Data exchange	0	FFFF	Default
MbVar9	Data exchange	0	FFFF	Default
MbVar10	Data exchange	0	FFFF	Default
MbVar11	Data exchange	0	FFFF	Default
MbVar12	Data exchange	0	FFFF	Default
MbVar13	Data exchange	0	FFFF	Default
MbVar14	Data exchange	0	FFFF	Default
MbVar15	Data exchange	0	FFFF	Default
MbVar16	Data exchange	0	FFFF	Default

Name	Type
MbVar1	BOOL
MbVar2	BOOL
MbVar3	BOOL
MbVar4	BOOL
MbVar5	BOOL
MbVar6	BOOL
MbVar7	BOOL
MbVar8	BOOL
MbVar9	BOOL
MbVar10	BOOL
MbVar11	BOOL
MbVar12	BOOL
MbVar13	BOOL
MbVar14	BOOL
MbVar15	BOOL
MbVar16	BOOL

RETAIN variables

13. **Offset:** Set the data block offset of each variable. Each variable should have a different offset number otherwise memory overlap occurs.



STEP 1:
Click "Offset" column

STEP 2:
Click "Iterate property"

STEP 3:
Start "Offset"
iteration from zero

	Value
	Read Input Bits
	1
Nb Item	16
Activation	Periodic

Name	Type
MbVar1	BOOL
MbVar2	BOOL
MbVar3	BOOL
MbVar4	BOOL
MbVar5	BOOL
MbVar6	BOOL
MbVar7	BOOL

Symbo	Operation	Offset	Mask	Storage
MbVar1	Data exchange	0	FFF	
MbVar2	Data exchange	1	FFF	
MbVar3	Data exchange	2	FFF	
MbVar4	Data exchange	3	FFF	
MbVar5	Data exchange	4	FFF	
MbVar6	Data exchange	5	FFF	
MbVar7	Data exchange	6	FFF	
MbVar8	Data exchange	7	FFF	
MbVar9	Data exchange	8	FFF	
MbVar10	Data exchange	9	FFF	
MbVar11	Data exchange	10	FFF	
MbVar12	Data exchange	11	FFF	
MbVar13	Data exchange	12	FFF	
MbVar14	Data exchange	13	FFF	
MbVar15	Data exchange	14	FFF	
MbVar16	Data exchange	15	FFF	

Offset

Name: %

From: 0

By: 1

Results

0
1
2
3
4
5
6
7

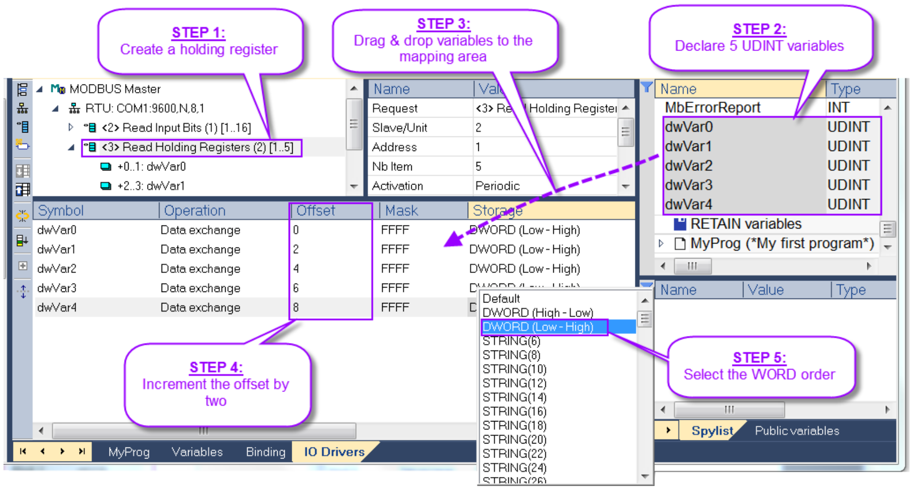
14. Mask:

- The mask in the mapping area only has to be set for input and holding registers and is being ignored for coil and input bits.
- A Modbus register has a memory size of 16 bit (same as INT, UINT, WORD). The mask (hex number) can be used to filter out bits from the register.
- Example:

Register Value	Mask (hex)	Result
65565	0001	1
65565	00FF	255
65565	FFFF	65565

15. Storage:

- The storage column is only relevant for the input and holding register setting.
- For exchanging 32 bit variables (DINT, REAL...), two consecutive Modbus register can be mapped to one variable.
- For exchanging strings multiple register can be mapped to one string



STEP 1: Create a holding register

STEP 3: Drag & drop variables to the mapping area

STEP 2: Declare 5 UDINT variables

STEP 4: Increment the offset by two

STEP 5: Select the WORD order

Name	Val
Request	<3> Re… Holding Register
Slave/Unit	2
Address	1
Nb Item	5
Activation	Periodic

Symbol	Operation	Offset	Mask	Storage
dwVar0	Data exchange	0	FFFF	DWORD (Low - High)
dwVar1	Data exchange	2	FFFF	DWORD (Low - High)
dwVar2	Data exchange	4	FFFF	DWORD (Low - High)
dwVar3	Data exchange	6	FFFF	DWORD (Low - High)
dwVar4	Data exchange	8	FFFF	D

Name	Type
MbErrorReport	INT
dwVar0	UDINT
dwVar1	UDINT
dwVar2	UDINT
dwVar3	UDINT
dwVar4	UDINT

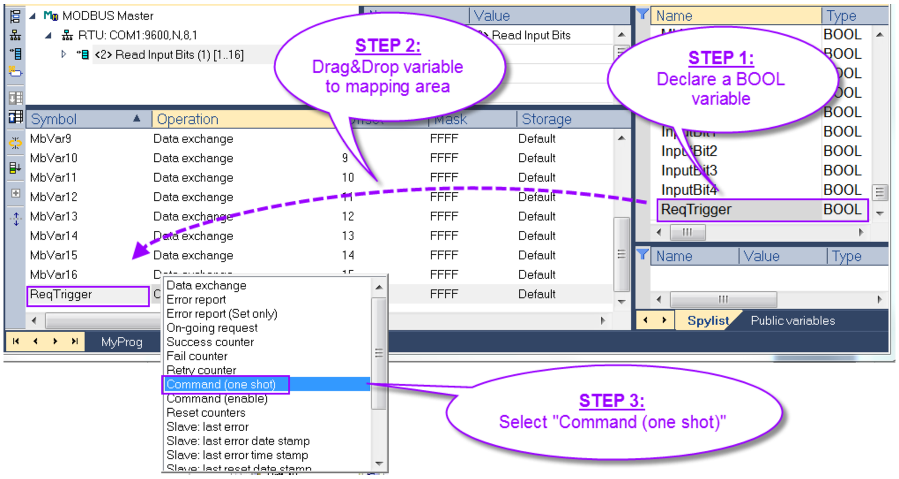
Figure 149: Mapping 32 bit variables to holding register

Step 5: This step has only to be done if the *'On call'* checkbox has been selected (No. 5 in Figure 147). If the master is in the *'On call'* mode the Win-GRAF runtime does not automatically sent a request to the slave. The logic program of the PLC has to trigger a request. Two command types are available to initiate a request:

1. Command (one shot)
   - A request will be sent once the attached flag (Figure 150: *'ReqTrigger'*) changes from FALSE to TRUE. Each time the PLC logic switches the flag to TRUE one Modbus command is being sent to the slave. After the request has been sent the flag will automatically be reset to zero by the runtime.
2. Command (Enable)
   - Once the BOOL variable mapped to the *'Command (Enable)'* operation is set TRUE by the PLC logic the Modbus master will send requests to the slave until the variable is set to FALSE. No commands will be sent if the variable is set to FALSE.

Procedure for implementing the *'On call'* procedure:

1. Declare a BOOL variable
2. Drag and drop the new variable to the mapping area
3. Select the *'Command (one shot)'* by double clicking the *'Operation'* column next to the new variable. The setting in the offset and storage column is only relevant for *'Data exchange'* operation and therefore can be ignored. If the PLC logic sets the *'RegTrigger'* to TRUE one master request will be sent to the slave and the runtime automatically resets the variable back to FALSE once the response has been received.



STEP 2:
Drag&Drop variable to mapping area

STEP 1:
Declare a BOOL variable

STEP 3:
Select "Command (one shot)"

Figure 150: Create a 'On call' variable

Step 6: Map a variable to the diagnostic and status information of the master. Information like communication timeout, invalid data address, invalid command, number of failed request etc. are recorded by the master. In the following a variable will be mapped to the *'Error report'* operation which records the error of the last request. Table 28 shows all the error codes used by the *'Error report'* operation.

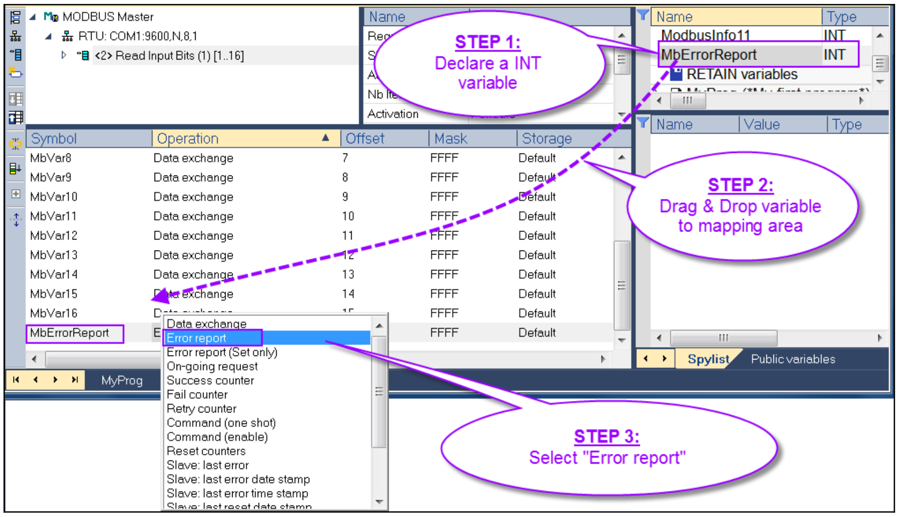
Error Code	Description
0	The communication is OK.
1	MODBUS function not supported.
2	Invalid MODBUS address.
3	Invalid MODBUS value.
4	MODBUS Server failure.
6	Server is busy.
8	Data Parity Error.
10	Invalid gateway path.
11	Gateway target failed.
128	Communication timeout.
129	Bad CRC16.
130	RS-232 communication error.

Table 28: Error code for 'Error report' operation

Procedure for implementing the *'Error report'* procedure (Figure 151):

1. Declare a INT variable (e.g. '*MbErrorReport*').
2. Drag and drop the new variable to the mapping list.
3. Double click the *'Operation'* column next to the variable and select *'Error*

report'. The offset and storage column setting is ignored by this operation. The variable will show an error code when a Modbus request error occurs, and will be reset to zero after the next request was successful.



Symbol	Operation	Offset	Mask	Storage
MbVar8	Data exchange	7	FFFF	Default
MbVar9	Data exchange	8	FFFF	Default
MbVar10	Data exchange	9	FFFF	Default
MbVar11	Data exchange	10	FFFF	Default
MbVar12	Data exchange	11	FFFF	Default
MbVar13	Data exchange	12	FFFF	Default
MbVar14	Data exchange	13	FFFF	Default
MbVar15	Data exchange	14	FFFF	Default
MbVar16	Data exchange	15	FFFF	Default
MbErrorReport	E		FFFF	Default

Name	Type
ModbusInfo11	INT
MbErrorReport	INT

Name	Value	Type

STEP 1:
Declare a INT variable

STEP 2:
Drag & Drop variable to mapping area

STEP 3:
Select "Error report"

Figure 151: Error report setting

12 Variables

All variables used in programs must be first declared in the variable editor. Each variable belongs to a group and is must be identified by a unique name within its group.

Groups
A group is a set of variables. A group either refers to a physical class of variables, or identifies the variables local to a program or user defined function block.

Below are the possible groups:

Group	Description
Global	Internal variables known by all programs
Retain	Non-volatile internal variables known by all programs
Program_{xxx}	All internal variables local to a program (The name of the group is the name of the program)
UDFB_{xxx}	All internal variables local to a User Define Function Block (UDFB) plus its IN and OUT parameters (The name of the group is the name of the program)
%I..	Channels of an input board - variables with same data type linked to a physical input device.
%Q..	Channels of an output board - variables with same data type linked to a physical output device.

Table 29: Groups

Data type and dimension

Each variable must have a valid data type. It can be either a basic data type or a function block. In that case the variable is an instance of the function block.
If the selected data type is STRING, you must specify a maximum length, that cannot exceed 255 characters.

Refer to the list of available data types for more information. Refer to the section describing function blocks for further information about how to use a function instance.

Additionally, you can specify dimension(s) for an internal variable, in order to declare an array. Arrays have at most 3 dimensions. All indexes are 0 based. For instance, in case of single dimension array, the first element is always identified by ArrayName[0]. The total number of items in an array (merging all dimensions) cannot exceed 65535. The possible range of Arrays reaches from ArrayName 0 until ArrayName 65534.

Naming a variable

A variable must be identified by a unique name within its parent group. The variable name cannot be a reserved keyword of the programming languages and cannot have the same name as a function block listed in the function block library. A variable should not have the same name as a program or a user defined function block.

The name of a variable should begin by a letter or an underscore ('_ ') mark, followed by letters, digits or underscore marks. It is not allowed to put two consecutive underscores within a variable name.
Note:
Naming is case insensitive. Two names with different cases are considered as the same.

Attributes of a variable

For each internal variable, you can select the '*Read Only*' option.

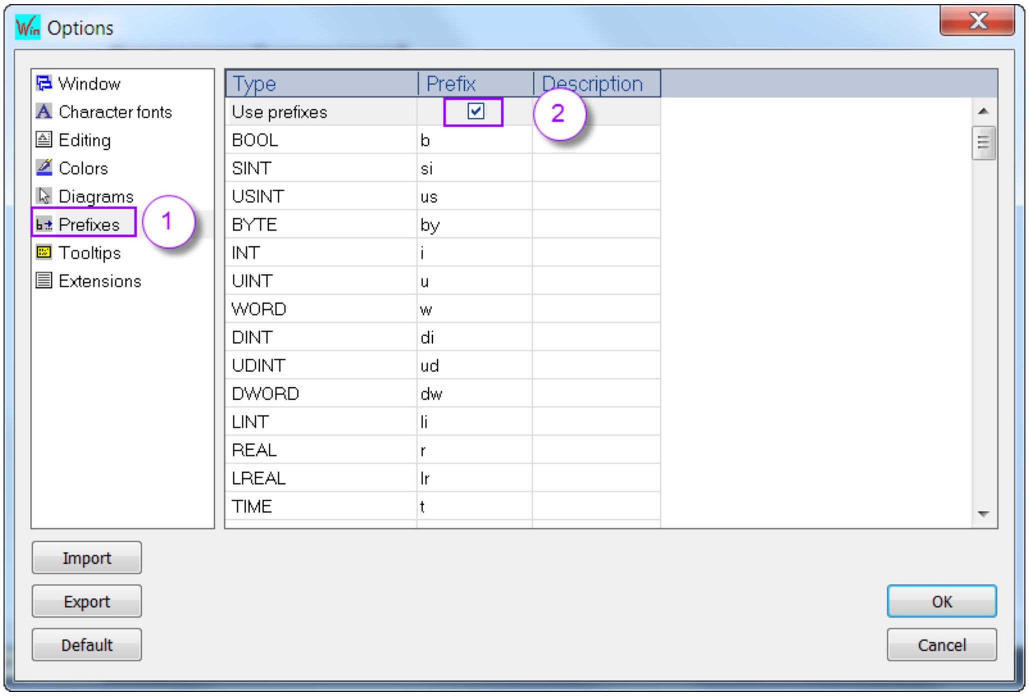
Parameters of User Defined Function Blocks and sub-programs are marked as either IN or OUT.

Parameters of sub-programs and UDFBs

Sub-programs and UDFBs may have input or output parameters. Output parameters cannot be arrays of data structures but only single data. When an array is passed as an input parameter to a UDFB, it is considered as INOUT so the UDFB can read and write in it. The support of complex data types for input parameters may depend on selected compiling options.

12.1 Create Variables

To increase the readability of the PLC program it is suggested to stick to a naming convention for the variable names. The Win-GRAF workbench allows you to set the prefixes for each variable type (e.g. *'b'* for BOOL, *'si'* for SINT, etc.).
Got to *'Tools/Options...'* in the menu and select *'Prefixes'* to set the prefixes for each variable type (Figure 152) and enable the use of prefixes. The enables the workbench during a variable declaration to automatically selects the type after entering the variable name with the prefix.



Type	Prefix	Description
Use prefixes	☑	
BOOL	b	
SINT	si	
USINT	us	
BYTE	by	
INT	i	
UINT	u	
WORD	w	
DINT	di	
UDINT	ud	
DWORD	dw	
LINT	li	
REAL	r	
LREAL	lr	
TIME	t	

Figure 152: Variable type prefix setting

12.1.1 Declare Variable in the Variable Editor

First determine group in which to declare a global, retain or local variable.

Step 1: Open the variable editor by double clicking the *'Variables'* in the workspace

Step 2: Right click the group (Global, Retain, local) and select *'Add Variables'* (Figure 153). A alternative way is to select the group and press the *'Insert'* key on the keyboard. A variable with the name *'NewVar'* of BOOL type is being added to the editor.

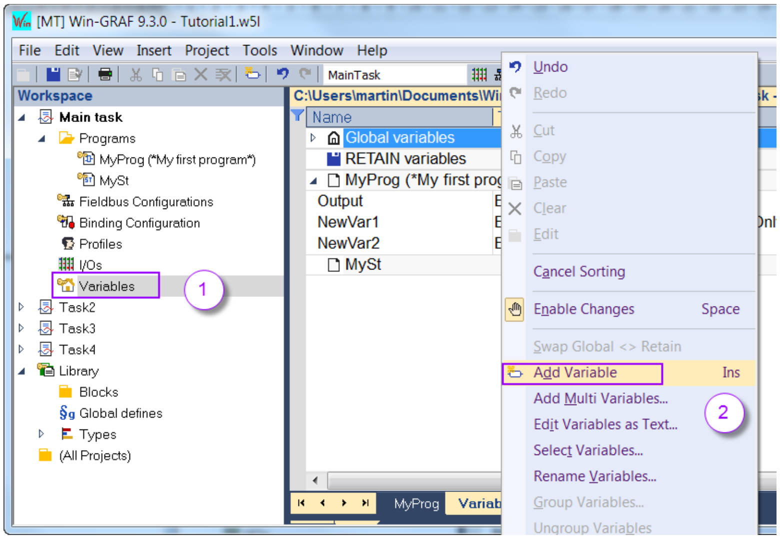
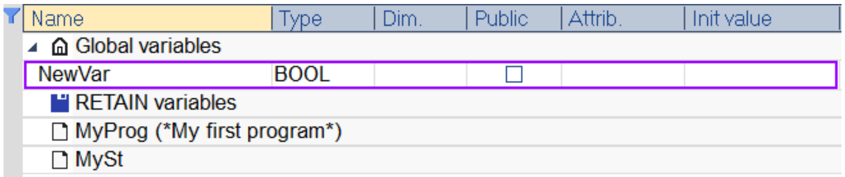


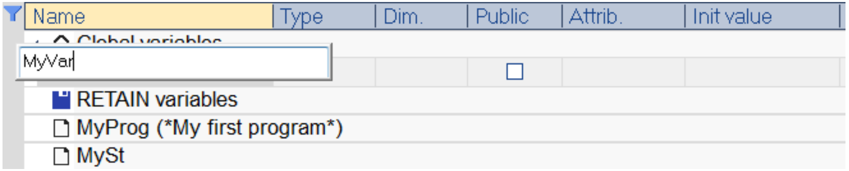
Figure 153: Declare a variable using the variable editor



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
NewVar	BOOL		□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

Step 3: Set variable name:

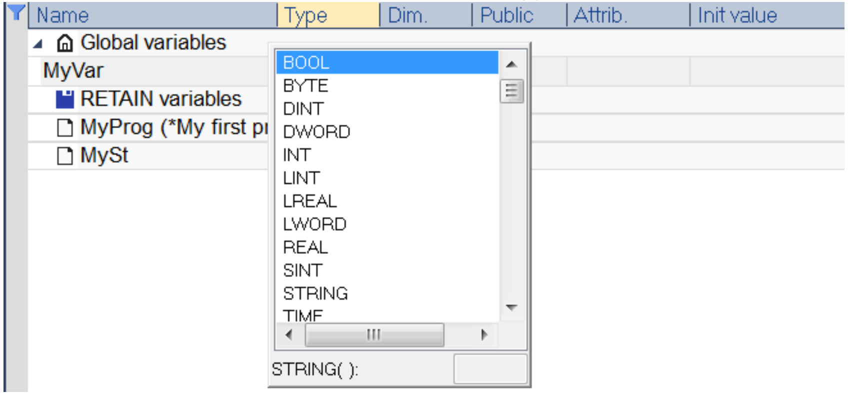
Double click the variable name, enter a new name in the edit box and press enter to assign the new name.



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar			□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

Step 4: Set data type:

Double the *'Type'* field to open a list of available data types and function blocks. Select one of the data types.

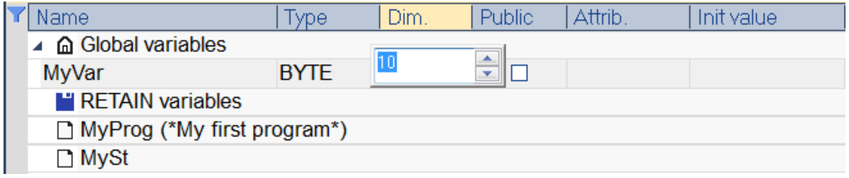


Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar					
RETAIN variables					
MyProg (*My first pı					
MySt					

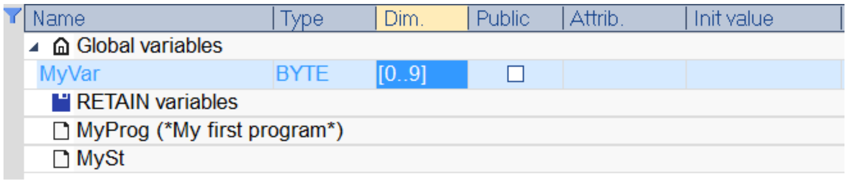
Step 5: Set array dimension:
Ignore this step if you do not want to declare an array.

Single dimensional array

Double click the dimension *'Dim'* field and assign the size of the array.



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE	10	□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

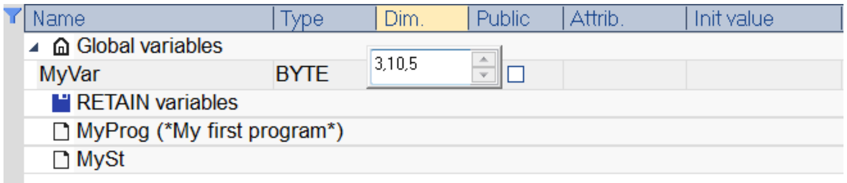


Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE	[0..9]	□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

All indexes are 0 based. Arrays have at most 3 dimensions. For instance, in case of single dimension array, the first element is always identified by MyVar[0]. The total number of items in an array (merging all dimensions) cannot exceed 65535. The possible range of Arrays reaches from ArrayName 0 until ArrayName 65534.

Multi-dimensional array:

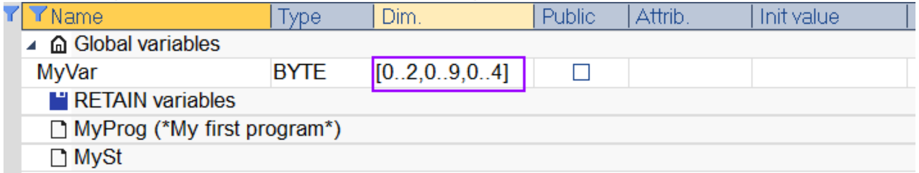
Using the variable editor, you must enter the number of elements for each dimensions separated by commas. Note that arrays have at most three dimension.
For example *'3,10,5'* is a three dimensional array, the first dimension has three elements, the second dimension has 10 elements, and the third dimension has five elements (Figure 154).



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE	3,10,5	□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

Figure 154: Multi-dimensional array declaration

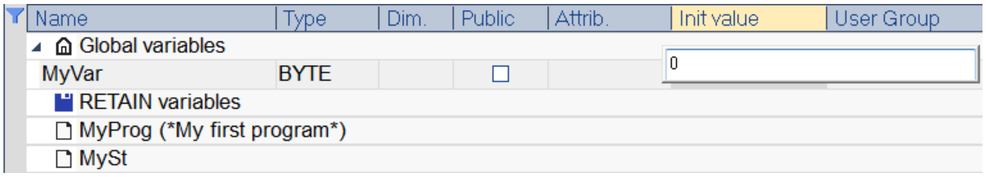
After the dimension has been edited the variable editor shows the following dimension information (Figure 155):



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE	[0..2,0..9,0..4]	□		
RETAIN variables					
MyProg (*My first program*)					
MySt					

Figure 155':Multi-dimensional array information after declaration

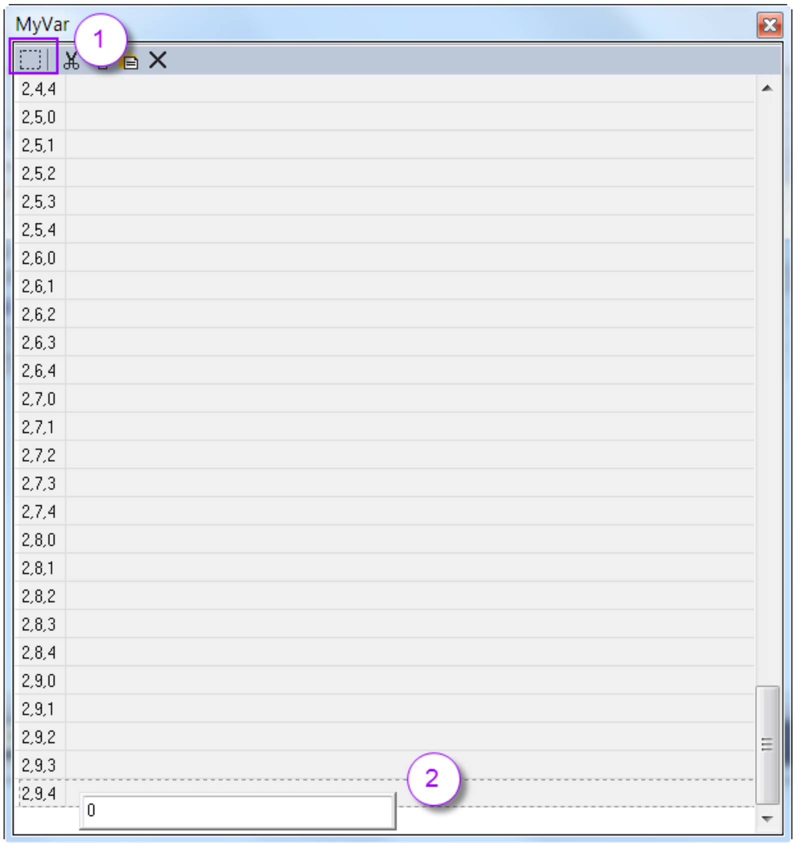
Step 6: Set initial value:
Double click the *'Init value'* field and enter a initial value.



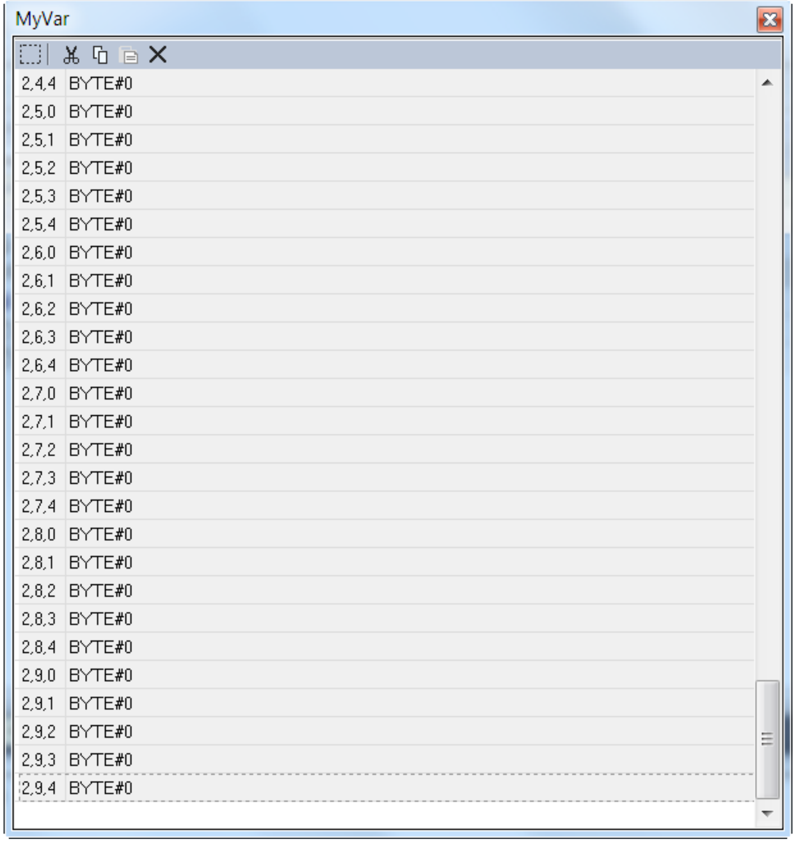
Name	Type	Dim.	Public	Attrib.	Init value	User Group
Global variables						
MyVar	BYTE		□		0	
RETAIN variables						
MyProg (*My first program*)						
MySt						

Initialize an array:

- Double clicking the *'Init value'* will open a window which list all the array element. A value can be assigned to each single element by double clicking the element line.
- Initialization of multi-dimensional arrays is done in the same manner as by one-dimensional arrays; first all elements for the first dimension are initialized (i.e. for example array[0,0], array[0,1], array[0,2] to array[0,n]) and then the procedure is repeated for the other values of the first index.
- To initial all the array element to the same value click the *'Select All'* button on the left top window to select all the elements. Enter a value via the keyboard which will be shown in a popup edit box. Press the *'Enter'* key of the key board to assign the value to all array elements.

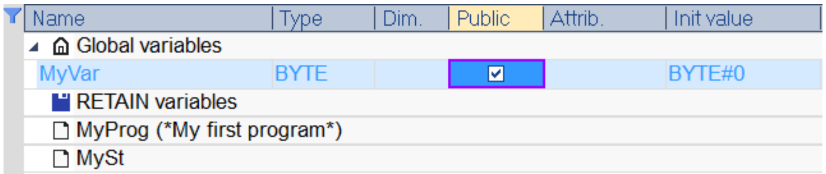


2,4,4	
2,5,0	
2,5,1	
2,5,2	
2,5,3	
2,5,4	
2,6,0	
2,6,1	
2,6,2	
2,6,3	
2,6,4	
2,7,0	
2,7,1	
2,7,2	
2,7,3	
2,7,4	
2,8,0	
2,8,1	
2,8,2	
2,8,3	
2,8,4	
2,9,0	
2,9,1	
2,9,2	
2,9,3	
2,9,4	0



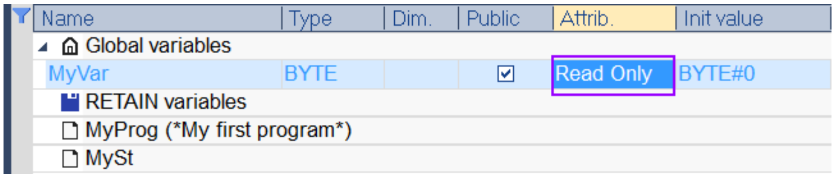
2,4,4	BYTE#0
2,5,0	BYTE#0
2,5,1	BYTE#0
2,5,2	BYTE#0
2,5,3	BYTE#0
2,5,4	BYTE#0
2,6,0	BYTE#0
2,6,1	BYTE#0
2,6,2	BYTE#0
2,6,3	BYTE#0
2,6,4	BYTE#0
2,7,0	BYTE#0
2,7,1	BYTE#0
2,7,2	BYTE#0
2,7,3	BYTE#0
2,7,4	BYTE#0
2,8,0	BYTE#0
2,8,1	BYTE#0
2,8,2	BYTE#0
2,8,3	BYTE#0
2,8,4	BYTE#0
2,9,0	BYTE#0
2,9,1	BYTE#0
2,9,2	BYTE#0
2,9,3	BYTE#0
2,9,4	BYTE#0

Step 7: Public variable: If other tasks needs to access the variable then the checkbox in the *'Public'* field has to be enabled.



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE		☑		BYTE#0
RETAIN variables					
MyProg (*My first program*)					
MySt					

Other task has now read and write access to this variable. The access can be limited to read only if the attribute is set to *'Read Only'* in the *'Attrib.'* column.

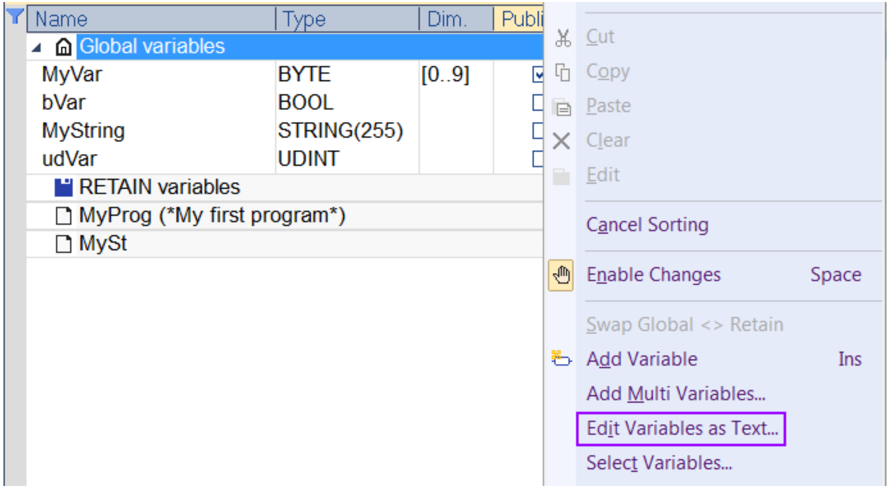


Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
MyVar	BYTE		☑	Read Only	BYTE#0
RETAIN variables					
MyProg (*My first program*)					
MySt					

12.1.2 Declare Variable as Text

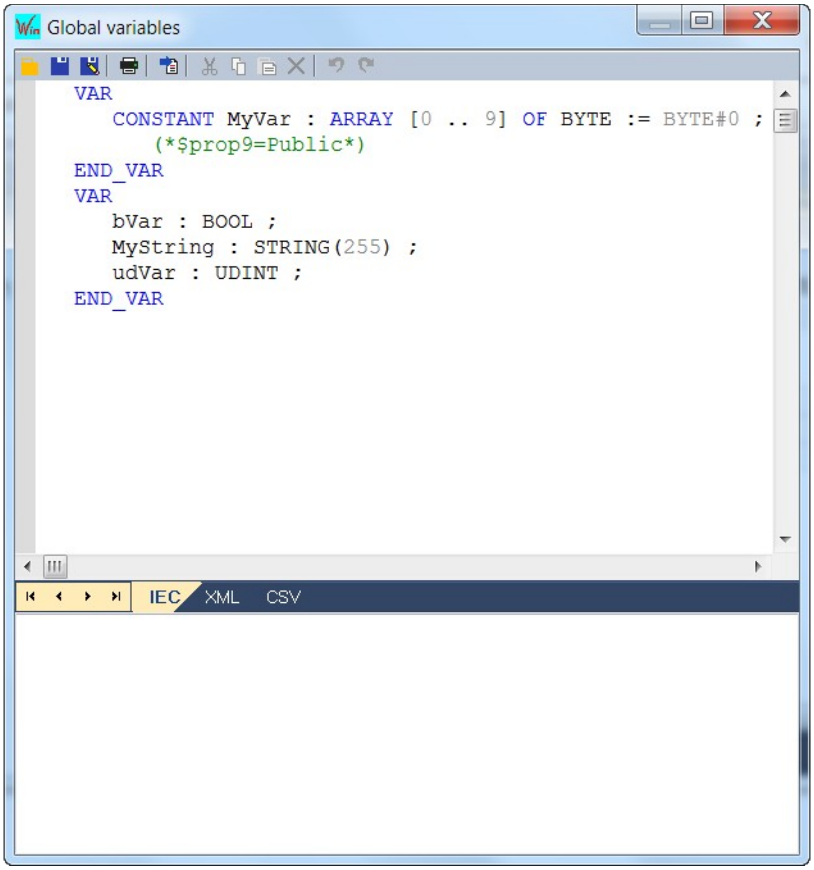
An alternative way for declaring variables is to directly type the variable name with the data type as an text using the IEC61131-3 format.

Step 1: Open the IEC61131-3 variable editor: Right click the variable editor area and select *'Edit Variables as Text...'* from the popup menu.



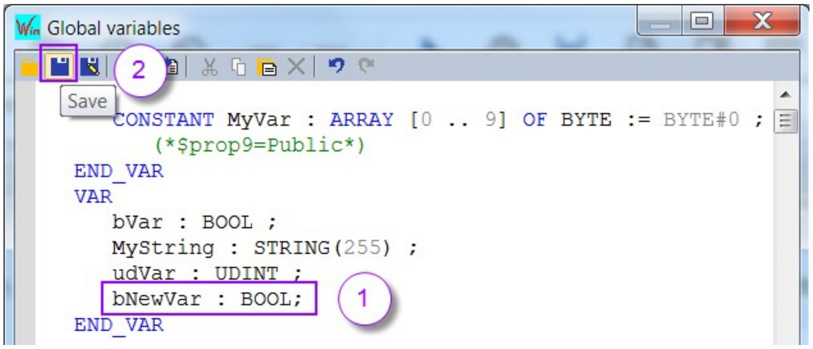
Name	Type	Dim.	Publi
Global variables			
MyVar	BYTE	[0..9]	☑
bVar	BOOL		□
MyString	STRING(255)		□
udVar	UDINT		□
RETAIN variables			
MyProg (*My first program*)			
MySt			

The editor shows all the declared variable of the group:



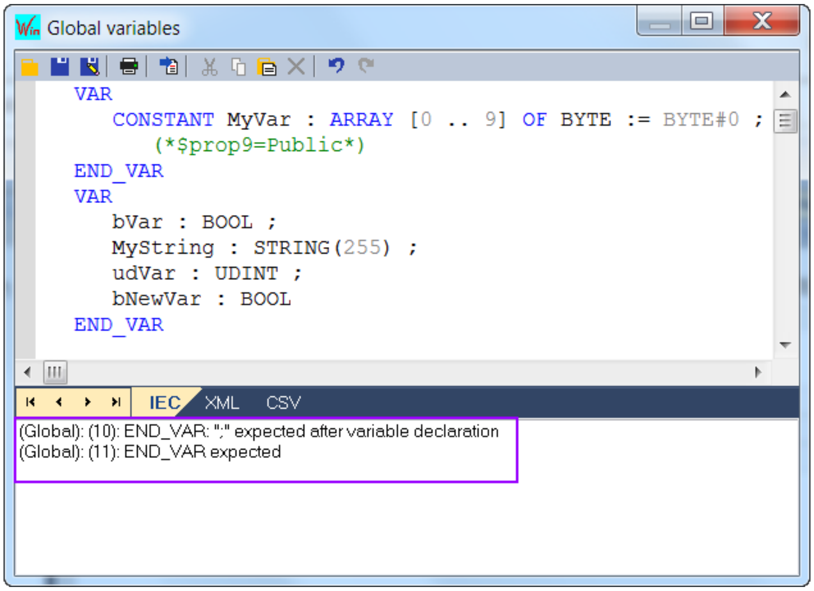
```
VAR
   CONSTANT MyVar : ARRAY [0 .. 9] OF BYTE := BYTE#0 ;
      (*$prop9=Public*)
END_VAR
VAR
   bVar : BOOL ;
   MyString : STRING(255) ;
   udVar : UDINT ;
END_VAR
```

Step 2: Add a new variable declaration and click save to validate the new settings.



```
   CONSTANT MyVar : ARRAY [0 .. 9] OF BYTE := BYTE#0 ;
      (*$prop9=Public*)
END_VAR
VAR
   bVar : BOOL ;
   MyString : STRING(255) ;
   udVar : UDINT ;
   bNewVar : BOOL;
END_VAR
```

After the save button has been clicked the Win-GRAF workbench will check the declaration for any syntax errors. If any error were detected the file will not be saved and the errors will be shown at the bottom of the window. By double clicking the error message the workbench will jump to the variable declaration were the error was detected.



```
VAR
   CONSTANT MyVar : ARRAY [0 .. 9] OF BYTE := BYTE#0 ;
      (*$prop9=Public*)
END_VAR
VAR
   bVar : BOOL ;
   MyString : STRING(255) ;
   udVar : UDINT ;
   bNewVar : BOOL
END_VAR
```

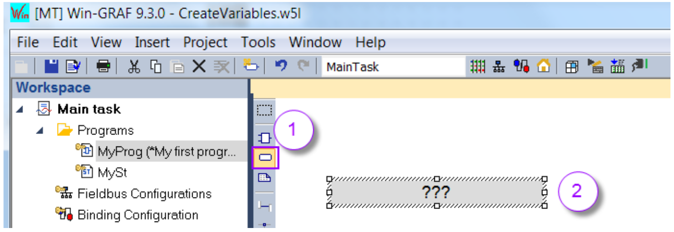
```
(Global): (10): END_VAR: ";" expected after variable declaration
(Global): (11): END_VAR expected
```

12.1.3 Declare Variable from the Program Editor

12.1.3.1 Declare a simple Variable

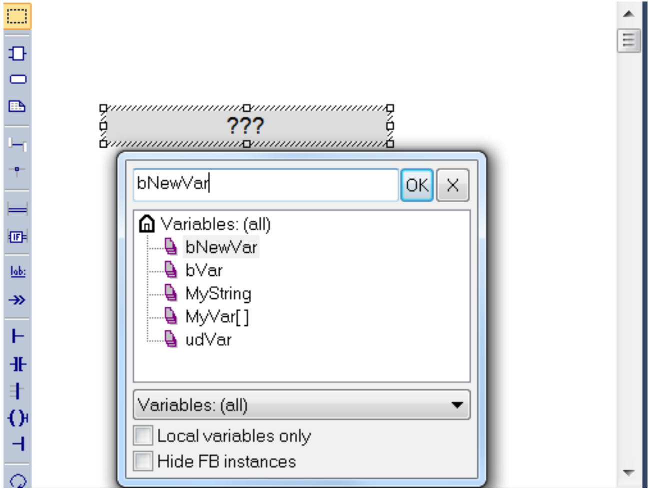
Variables can be directly create via the program editor. The following describes the procedure of adding a variable to the function block diagram (FBD) area. The procedures for the other ICE 61131 programming languages are very similar.

Step 1: Add a variable box:
The variable box represent a variable in FB and Ladder programming. Click the *'Add variable'* button on the left of the program editor and click a location in the editor where to place the variable box.

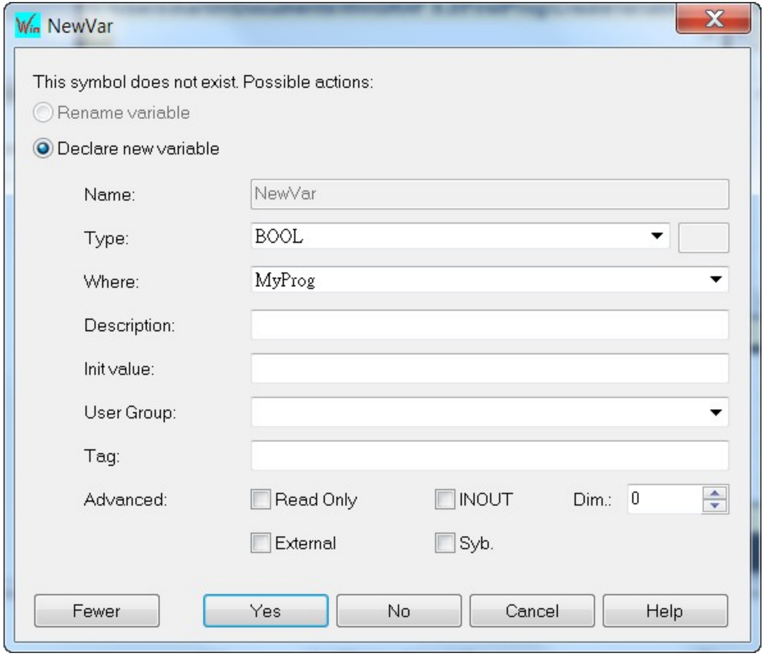

[MT] Win-GRAF 9.3.0 - CreateVariables.w5l
File Edit View Insert Project Tools Window Help
MainTask
Workspace
Main task
Programs
MyProg (*My first progr...
MySt
Fieldbus Configurations
Binding Configuration
1
???
2

Step 2: Set variable name and type:
Double click the variable box. A popup window shows all the already declared variables. You can declare a new variable or select one of the available

variables from the list. Declare a new variable by entering a new name in the top text box of the popup window. The name is prefixed by *'b'* which indicates that a BOOL type should be declared as defined in the prefix table (Figure 152). After clicking *'OK'* the workbench will automatically declare a variable of an BOOL type.



If the variable name (e.g. *'NewVar'*) does not contain a prefixed character (e.g. *'b'*) as defined in the prefix table then a pop up windows appear which enables you to select the data type, group, array dimension, etc.


NewVar

This symbol does not exist. Possible actions:

○ Rename variable

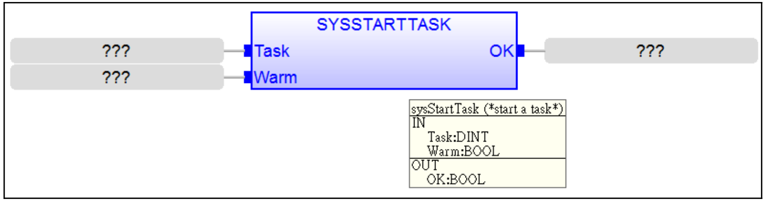
● Declare new variable

Name:	NewVar
Type:	BOOL
Where:	MyProg
Description:	
Init value:	
User Group:	
Tag:	
Advanced:	□ Read Only □ INOUT Dim.: 0
	□ External □ Syb.

12.1.3.2 Declare Variable for Function In-and Output

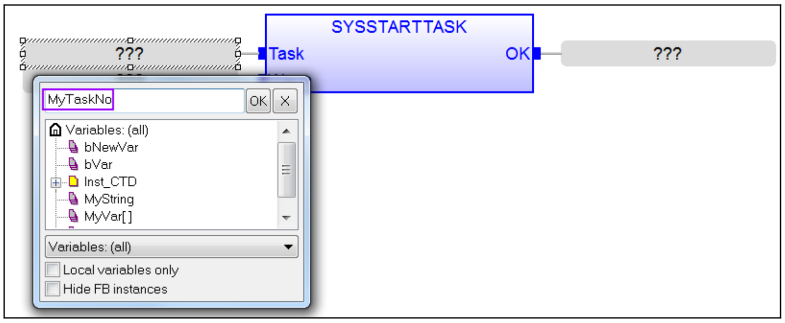
After a function has been added to the programming area variables needs to be assigned to the in and outputs. Existing variables listed in the variable editor can be assigned or new variables can be directly declared in the program editor. The procedure to add a variable to the in- and outputs of a function is as follows:

Step 1: After a function has been added to the function block program editor the in and outputs are marked by *'???'* which indicates that no variables have been assigned yet. Hover the mouse over the function block to get information about the data type of each port. Click the block and press F1 to open the help file for the function.



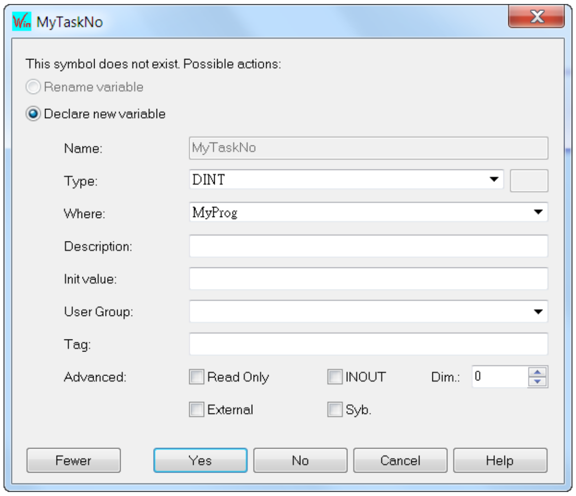
sysStartTask (*start a task*)
IN
Task:DINT
Warm:BOOL
OUT
OK:BOOL

Step 2: Declare variable:
Double click a box which contains three question marks (*'???'*) to open an editor to enter a variable name (e.g. *'MyTasNo'*). Click *'OK'* to confirm the name.



The popup editor also list all the declared variables. If the variable to be assigned to the block port has already been declared previously you just need to select the variable name from the list and click *'OK'*

Step 3: Set variable type:
If the variable has not been declared before a window pops up which allows you to directly set the data type, location, initial value, etc. of the variable. By default the popup editor will set the data type to the type required by the corresponding function block in- or output port.



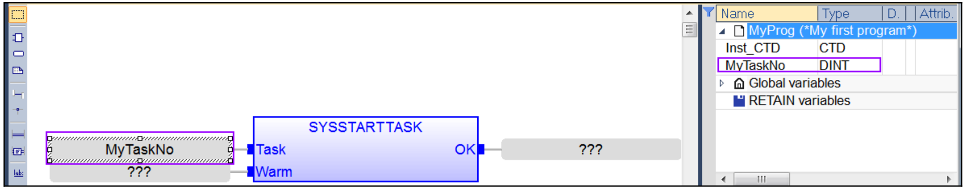
This symbol does not exist. Possible actions:

○ Rename variable

● Declare new variable

Name:	MyTaskNo
Type:	DINT
Where:	MyProg
Description:	
Init value:	
User Group:	
Tag:	
Advanced:	□ Read Only □ INOUT Dim.: 0
	□ External □ Syb.

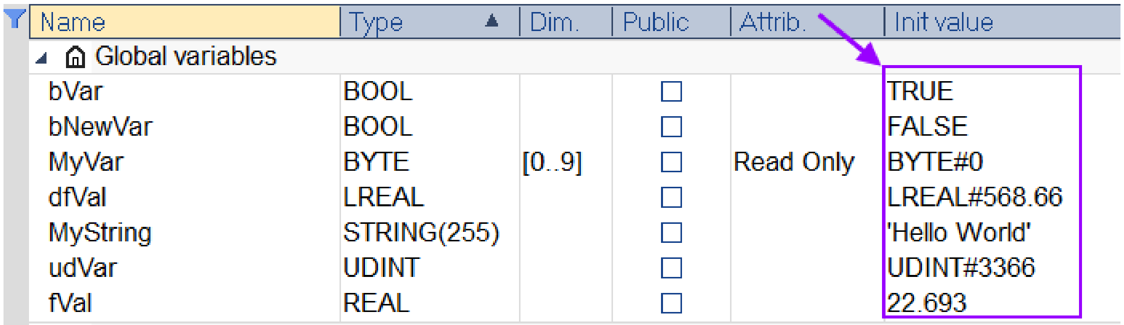
After clicking *'Yes'* the variable is added to variable list in the variable editor and assigned to the function port



Name	Type	D.	Attrib.
MyProg (*My first program*)			
Inst_CTD	CTD		
MyTaskNo	DINT		
Global variables			
RETAIN variables			

12.1.3.3 Assign Variable a Constant Expression

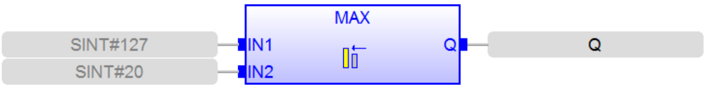
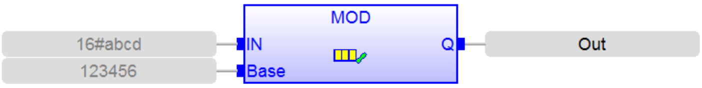
Constant expressions can be used in all five programming languages for assigning a variable with a value (Figure 156). All constant expressions have a well defined data type according to their semantics. If you program an operation between variables and constant expressions having inconsistent data types, it will lead to syntax errors when the program is compiled.

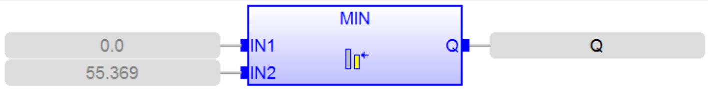
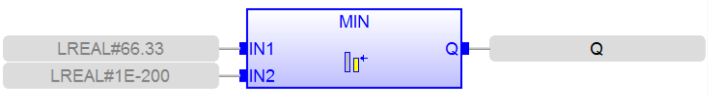
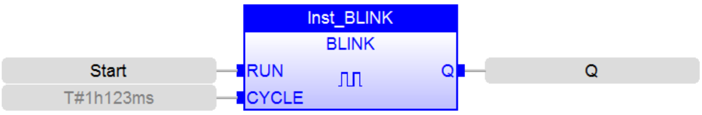


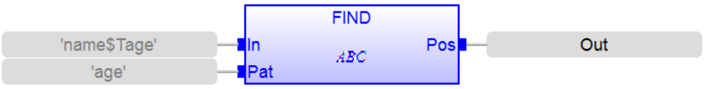
Name	Type ▲	Dim.	Public	Attrib.	Init value
Global variables					
bVar	BOOL		□		TRUE
bNewVar	BOOL		□		FALSE
MyVar	BYTE	[0..9]	□	Read Only	BYTE#0
dfVal	LREAL		□		LREAL#568.66
MyString	STRING(255)		□		'Hello World'
udVar	UDINT		□		UDINT#3366
fVal	REAL		□		22.693

Figure 156: Example of assigning a variable with a value

Below are the syntactic rules for constant expressions according to possible data types:

Type	Prefix	Description
BOOL		Boolean - There are only two possible Boolean constant expressions. They are reserved keywords `TRUE` and `FALSE`.
SINT	SINT#	Small (8 bit) Integer - Small integer constant expressions are valid integer values (between -128 and 127) and must be prefixed with SINT#. - All integer expressions having no prefix are considered as DINT integers. 
USINT / BYTE	USINT#	Unsigned 8 bit Integer - Unsigned small integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with USINT#. - All integer expressions having no prefix are considered as DINT integers.
INT	INT#	16 bit integer - 16 bit integer constant expressions are valid integer values (between -32768 and 32767) and must be prefixed with INT#. - All integer expressions having no prefix are considered as DINT integers.
UINT / WORD	UINT#	Unsigned 16 bit integer - Unsigned 16 bit integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with UINT#. - All integer expressions having no prefix are considered as DINT integers.
DINT		32 bit (default) integer - 32 bit integer constant expressions must be valid numbers between -2147483648 to +2147483647. - DINT is the default size for integers: such constant expressions do not need any prefix. - You can use `2#`, `8#` or `16#` prefixes for specifying a number in respectively binary, octal or hexadecimal basis. 
UDINT / DWORD	UDINT#	Unsigned 32 bit integer - Unsigned 32 bit integer constant expressions are valid integer values (between 0 and 4294967295) and must be prefixed with UDINT#. - All integer expressions having no prefix are considered as DINT integers.
LINT	LINT#	Long (64 bit) integer - Long integer constant expressions are valid integer values and must be prefixed with LINT#. - All integer expressions having no prefix are considered as DINT

Type	Prefix	Description
		integers.
REAL		Single precision floating point value - Real constant expressions must be valid number, and must include a dot ('.'). - If you need to enter a real expression having an integer value, add .0 at the end of the number. - You can use F or E separators for specifying the exponent in case of a scientist representation. - REAL is the default precision for floating points: such expressions do not need any prefix. 
LREAL	LREAL#	Double precision floating point value - Real constant expressions must be valid number, and must include a dot ('.'), and must be prefixed with LREAL#. - If you need to enter a real expression having an integer value, add .0 at the end of the number. - You can use F or E separators for specifying the exponent in case of a scientist representation. 
TIME	TIME# or T#	Time of day - Time constant expressions represent durations that must be less than 24 hours. Expressions must be prefixed by either TIME# or T#. They are expressed as • a number of hours followed by h, • a number of minutes followed by m, • a number of seconds followed by s, and • a number of milliseconds followed by ms. - The order of units (hour, minutes, seconds, milliseconds) must be respected. - You cannot insert blank characters in the time expression. There must be at least one valid unit letter in the expression. - Example: `T#23h59m59s999ms` - maximum TIME value `TIME#0s` - null TIME value `T#1h123ms` - TIME value with some units missing  Inst_BLINK BLINK Start T#1h123ms RUN CYCLE Q Q

<table>
<tr><th>Type</th><th>Prefix</th><th>Description</th></tr>
<tr><td>STRING</td><td></td><td>Character string
- String expressions must be written between single quote marks.
- The length of the string cannot exceed 255 characters.
- You can use the following sequences to represent a special or not printable character within a string:
<table>
<tr><th>Sequence</th><th>Description</th></tr>
<tr><td>$$</td><td>a <i>'$'</i> character</td></tr>
<tr><td>$'</td><td>a single quote</td></tr>
<tr><td>$T</td><td>a tab stop (ASCII code 9)</td></tr>
<tr><td>$R</td><td>a carriage return character (ASCII code 13)</td></tr>
<tr><td>$L</td><td>a line feed character (ASCII code 10)</td></tr>
<tr><td>$N</td><td>carriage return plus line feed characters (ASCII codes 13 and 10)</td></tr>
<tr><td>$P</td><td>a page break character (ASCII code 12)</td></tr>
<tr><td>$xx</td><td>any character (xx is the ASCII code expressed on two hexadecimal digits</td></tr>
</table>
- Example:
<code>'hello'</code> - character string
<code>'name$Tage'</code> - character string with two words separated by a tab
<code>'I$'m here'</code> - character string with a quote inside (I'm here)
<code>'x$00y'</code> - character string with two characters separated by a null character (ASCII code 0)


</td></tr>
</table>

Table 30: Prefixes for constant expressions

Below are some examples of typical errors in constant expressions:

Expression	Error-Description
1a2b	basis prefix (*'16#'*) omitted
1E-200	*'LREAL#'* prefix omitted for a double precision float
T#12	Time unit missing
'I'm here'	quote within a string with *'$'* mark omitted
hello	quotes omitted around a character string

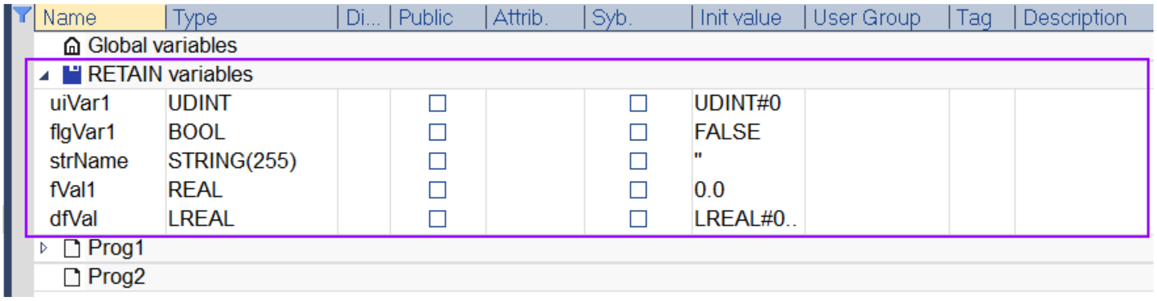
Table 31: Constant expressions syntax errors

12.2 Retain Variables

A retain variable is a PLC variable which:

- is non-volatile and is stored in a normal disk file.
- is known by all programs (when its content is changed, the change is propagated to all equations in which this variable is used)
- normally does not contain real-time critical data.

Retain variables are declared in the same way as a volatile variable. In the variable editor retain variables have to be declared in the *'RETAIN variables'* section (Figure 157). Function blocks instances can not be set as retain variables.



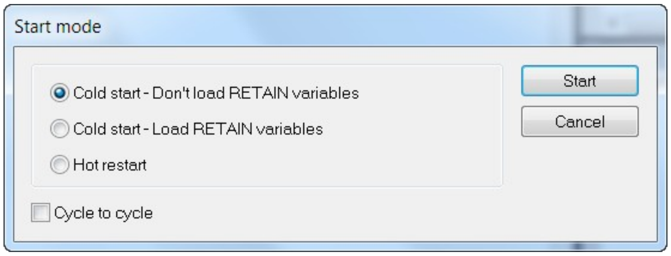
Name	Type	Di...	Public	Attrib.	Syb.	Init value	User Group	Tag	Description
Global variables									
RETAIN variables									
uiVar1	UDINT		☐		☐	UDINT#0			
flgVar1	BOOL		☐		☐	FALSE			
strName	STRING(255)		☐		☐	''			
fVal1	REAL		☐		☐	0.0			
dfVal	LREAL		☐		☐	LREAL#0..			
Prog1									
Prog2									

Figure 157: Retain variable declaration

Retain variables are saved when Win-GRAF runtimes shuts down. The retain variables of each task are stored in a separated file (Main task: '*t5_1.ret*'; Task 2: '*t5_2.ret*'; Task3: '*t5_3.ret*', etc.). The next time the runtime is started the retain variables are initialized with the value stored.

When using the workbench to start the PLC application the user can select whether the retain value are initialized with the default value or with the with the value stored in the file (Figure 158):

- On an application *'Cold start - Don'*t load RETAIN variables', the workbench initializes the retain variables with their default value. Default values are values entered in the *'Init value'* column of the variable editor.
- On an application *'Cold start - Load RETAIN variables'*, the workbench initializes the retain variables with the value stored in the disk file.



Start mode

● Cold start - Don't load RETAIN variables

○ Cold start - Load RETAIN variables

○ Hot restart

☐ Cycle to cycle

Figure 158: Workbench

12.2.1 Programmatically Save/Load Retain Variables

The runtime will automatically store the retain variables to file once the runtime is stopped. The files of the *'.ret'* format are located in the directory of the runtime execution file.

Win-GRAF provides functions which allows the user to programmatically save and load retain variables. Retain variables can be written to the file with the *'F_SAVERETAIN'* function and be loaded by the *'F_LOADRETAIN'* function (Table 32). By using these two functions the user can within the PLC program implement the time interval or a trigger event at which the retain variables are saved or loaded.

Function	Description
F_SAVERETAIN (Path) 	Save retain variable data to a file at the specified path every time the function is being called.
F_LOADERETAIN (Path) 	Load retain variable data from a file at the specified path.

Table 32: Functions for saving and loading retain variables

Note:

- F_SAVERETAIN and F_LOADRETAIN will not automatically create a folder if it does not exist. Table 33 shows the different path names supported by the functions.
- Do not change the retain variable declaration after a retain variable file has been created otherwise a variable mismatch may occur when the PLC application starts running and initializing the retain variable by loading the stored values from the disk. It is strongly suggested to delete the retain file (*'.ret'*) from the disk before changing the retain variable declaration.

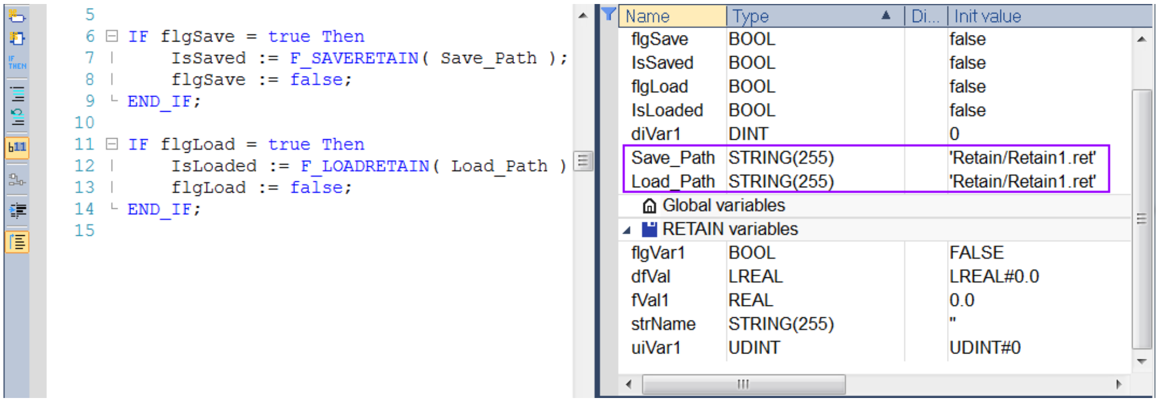
  The following actions will reset retained value(s) to their Init value(s):
  - Changing the type of a retain variable
  - Changing the length of a string retain variable
  - Changing the size of an array variable
  - Changing any element of a structure variable

Path	Description

FileName.ret	File will be in the directory of the runtime execution file
Folder/FileName.ret	File will be in the subdirectory *'Folder'* of the runtime execution file directory
./Folder/FileName.ret	
/Folder/FileName.ret	File will be in the directory *'Local Disk\Folder'*, whereby the *'Local Disk'* is the disk of the runtime execution (e.g. `D:\Folder\FileName.ret`)

Table 33: Path name definition

Figure 159 shows an example of how to programmatically use the F_SAVERETAIN and F_LOADRETAIN functions. Saving/loading data takes some time and slows down the system and therefore it is not suggested to call these function in a short time interval. In order to prevent the retain data to be save in every cycle flags (flgSave, flgLoad) are being used to trigger a save or load action.



```
IF flgSave = true Then
    IsSaved := F_SAVERETAIN( Save_Path );
    flgSave := false;
END_IF;

IF flgLoad = true Then
    IsLoaded := F_LOADRETAIN( Load_Path )
    flgLoad := false;
END_IF;

```

Name	Type	Di...	Init value
flgSave	BOOL		false
IsSaved	BOOL		false
flgLoad	BOOL		false
IsLoaded	BOOL		false
diVar1	DINT		0
Save_Path	STRING(255)		'Retain/Retain1.ret'
Load_Path	STRING(255)		'Retain/Retain1.ret'
Global variables			
RETAIN variables			
flgVar1	BOOL		FALSE
dfVal	LREAL		LREAL#0.0
fVal1	REAL		0.0
strName	STRING(255)		''
uiVar1	UDINT		UDINT#0

Figure 159: Programmatically saving and loading retain variable values

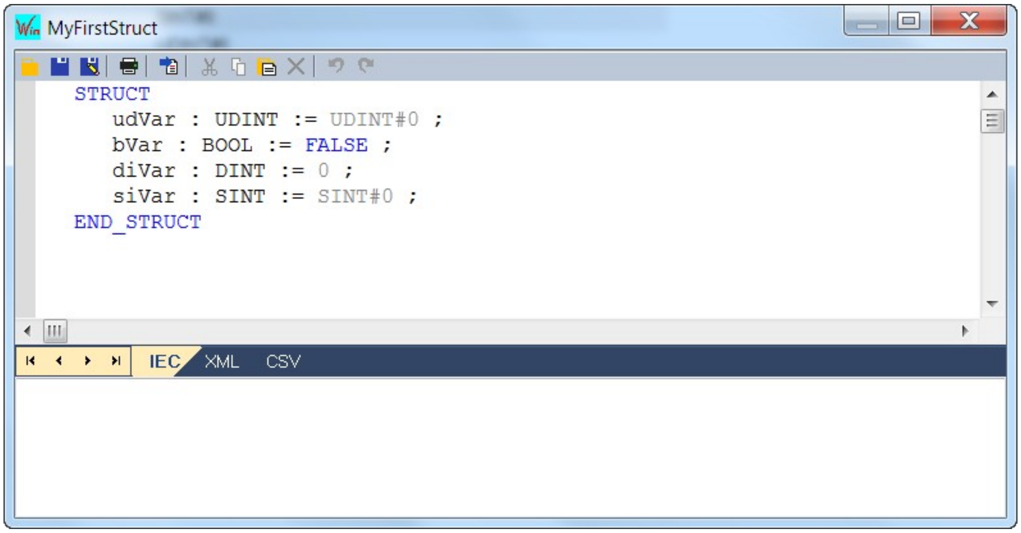
13 Derived Data Type

Derived types are data types specified by manufacturer or by user and can be declared by means of textual structure TYPE…END_TYPE. The names of new types, their data types, possible with their initial values, are given within this textual structure. These derived data types can be further used together with the elementary data types in declarations of variables. The definition of the derived data type is global, i.e. can be used in any PLC program part. The derived data type takes adopts the type features from which it was derived from.

13.1 Structures

A structure is a user defined data type. A structure can be derived from elementary as well as from derived data types. Structures can be used like other data types to declare variables.

According to IC61133 the definition of a new structure data type is done using the keywords `STRUCT` and `END_STRUCT`. Data types of individual members of a structure and their names are stated inside `STRUCT` … `END_STRUCT`. It is possible to initialize structures by stating member values behind the sign '*:=*' (Figure 160). Member of a structure can have an initial value. In that case, corresponding members of all declared variables having this structure type will be initialized with the initial value of the member.
Use the name of the structure instance followed by a dot and member name to access individual structure members: '*instanceName.memberName*'.



```
STRUCT
   udVar : UDINT := UDINT#0 ;
   bVar : BOOL := FALSE ;
   diVar : DINT := 0 ;
   siVar : SINT := SINT#0 ;
END_STRUCT
```

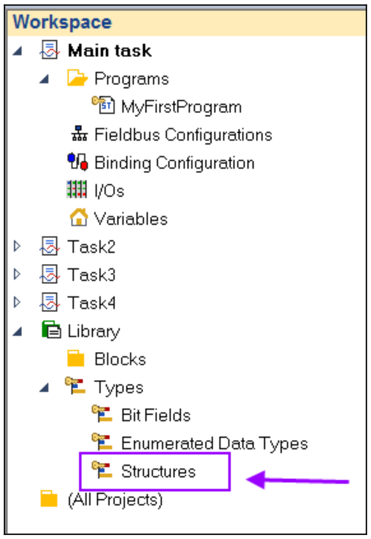
Figure 160: Structure definition

A structure can be created hierarchical which means that an already defined structure can be an member of another structure. A structure must be defined before it can be used as a member of another structure.

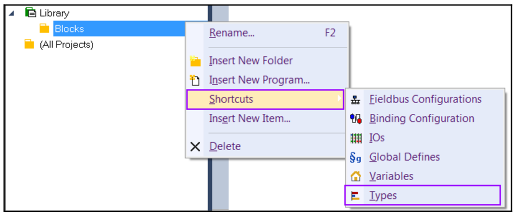
13.1.1 Define a Structure

13.1.1.1 Define a Structure from the Editors

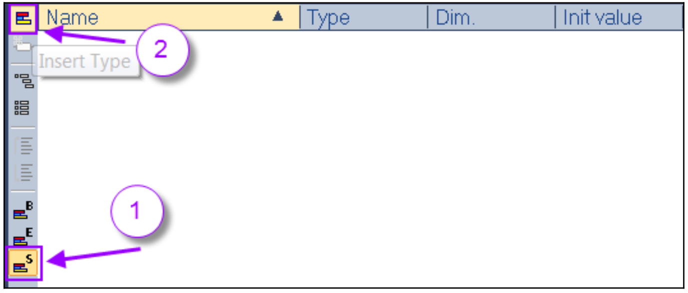
Step 1: Open the structure editor:
Go to the *'Library'* node in the workspace and double click the *'Structures'* item in the *'Types'* tree to open the editor. If the *'Types'* tree is not visible then enable it by right clicking the *'Library'* node and selecting *'Shortcuts\Types'* from the popup menu.



Workspace
- Main task
  - Programs
    - MyFirstProgram
  - Fieldbus Configurations
  - Binding Configuration
  - I/Os
  - Variables
- Task2
- Task3
- Task4
- Library
  - Blocks
  - Types
    - Bit Fields
    - Enumerated Data Types
    - Structures
  - (All Projects)



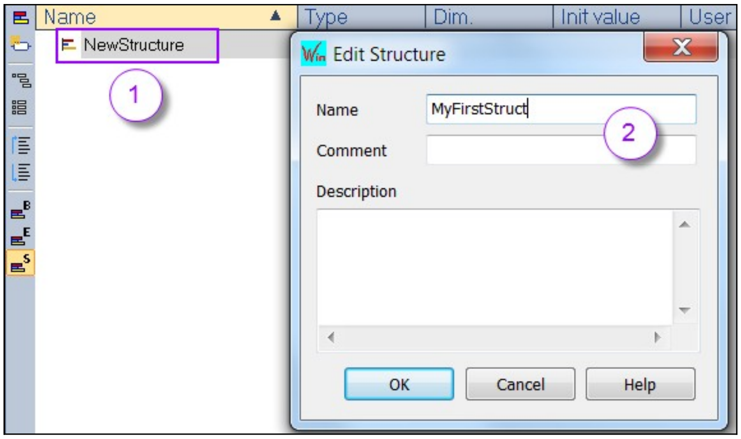
Step 2: Click the *'Insert Type'* button on the top left to add a new structure



Make sure that the icon with the superscripted *'S'* is active which indicates that structure types are being displayed.

Icon	Data Type
S	Structure
E	Enumeration
B	Bitfield

Step 3: Double click the newly added structure *'NewStructure'* to rename it (e.g. MyFirstStruct)



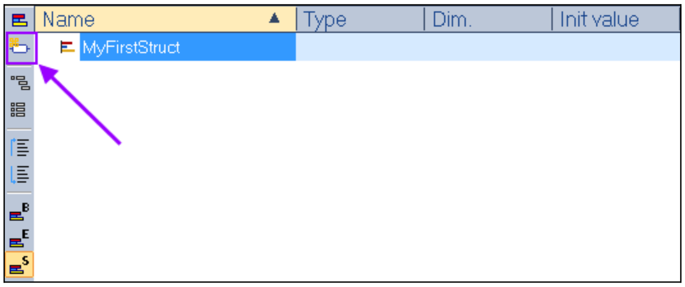
Name	Type	Dim.	Init value	User
NewStructure				

Name: MyFirstStruct

Comment:

Description:

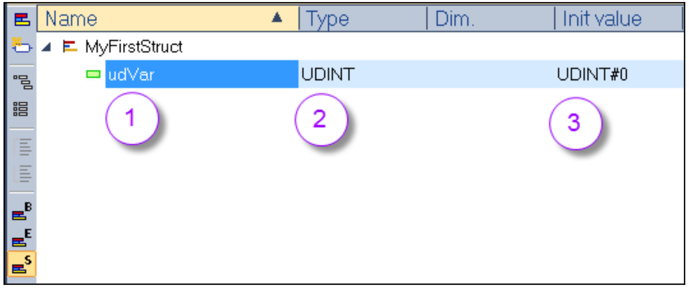
Step 4: Add a variable to the structure:
Click the structure name and press the *'Insert'* key or click *'Insert Variable'* button



Name	Type	Dim.	Init value
MyFirstStruct			

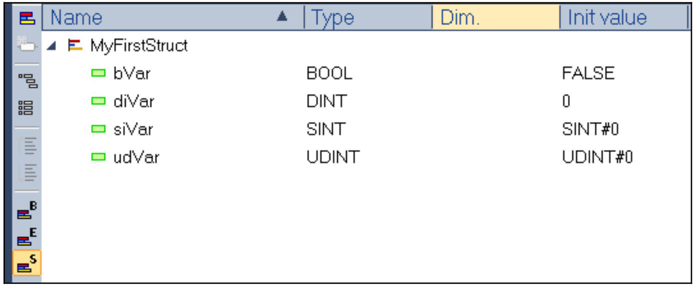
Step 5: Set structure member:

1. Rename the variable by double clicking the variable name.
2. Reset the data type by double clicking the type field.
3. Set initial value by double clicking the *'Init value'* field



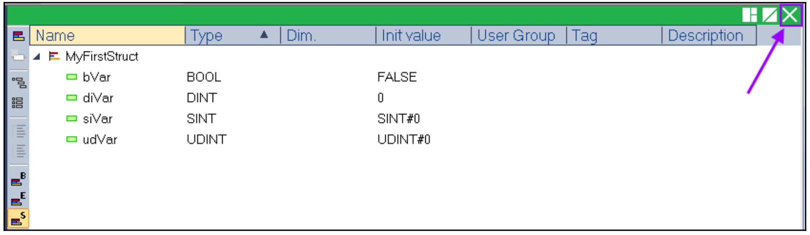
Name	Type	Dim.	Init value
MyFirstStruct			
udVar	UDINT		UDINT#0

Step 6: Repeat step 4 and 5 to add variables to the structure



Name	Type	Dim.	Init value
MyFirstStruct			
bVar	BOOL		FALSE
diVar	DINT		0
siVar	SINT		SINT#0
udVar	UDINT		UDINT#0

Step 7: Close the structure editor.
It is important to close the structure editor before the structure can be used. Closing the editor causes the workbench to store the structure definition and make it available for declaring structure instances.

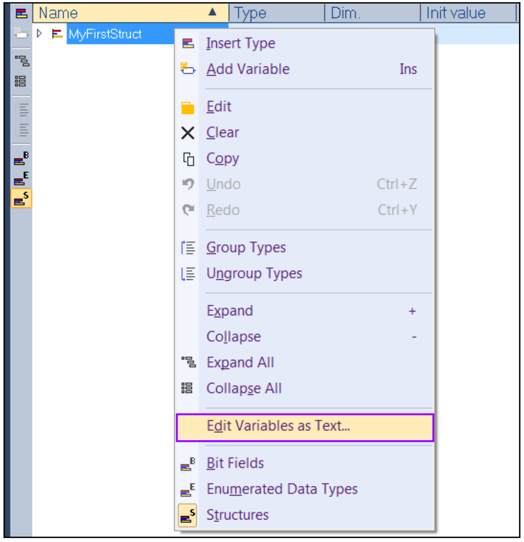


Name	Type	Dim.	Init value	User Group	Tag	Description
MyFirstStruct						
bVar	BOOL		FALSE			
diVar	DINT		0			
siVar	SINT		SINT#0			
udVar	UDINT		UDINT#0			

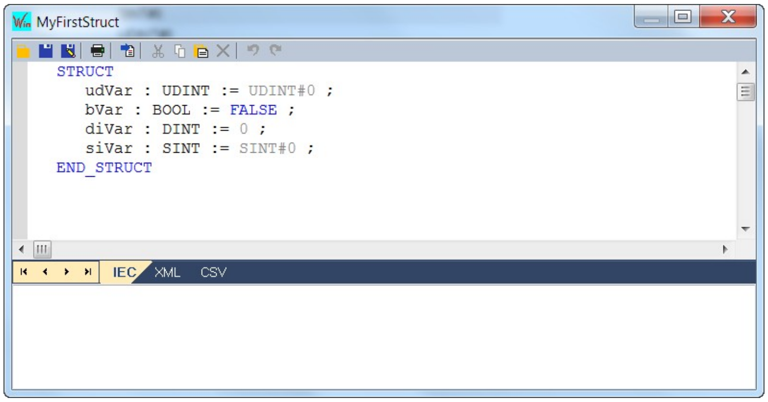
13.1.1.2 Define a Structure as Text

The workbench provides a text editor for directly adding members to the structure without using the edit function provided by the user interface (Figure 161).

Step 1: Add a new structure (follow the steps 1 to 3) described in chapter 13.1.1.1.
Step 2: Open the text editor for the structure by right clicking the structure name and selecting *'Edit Variable as Text...'*



Step 3: Edit member variables and initialization values. Save the structure before closing the window.



```
STRUCT
   udVar : UDINT := UDINT#0 ;
   bVar : BOOL := FALSE ;
   diVar : DINT := 0 ;
   siVar : SINT := SINT#0 ;
END_STRUCT
```

Figure 161: Edit structure members as text

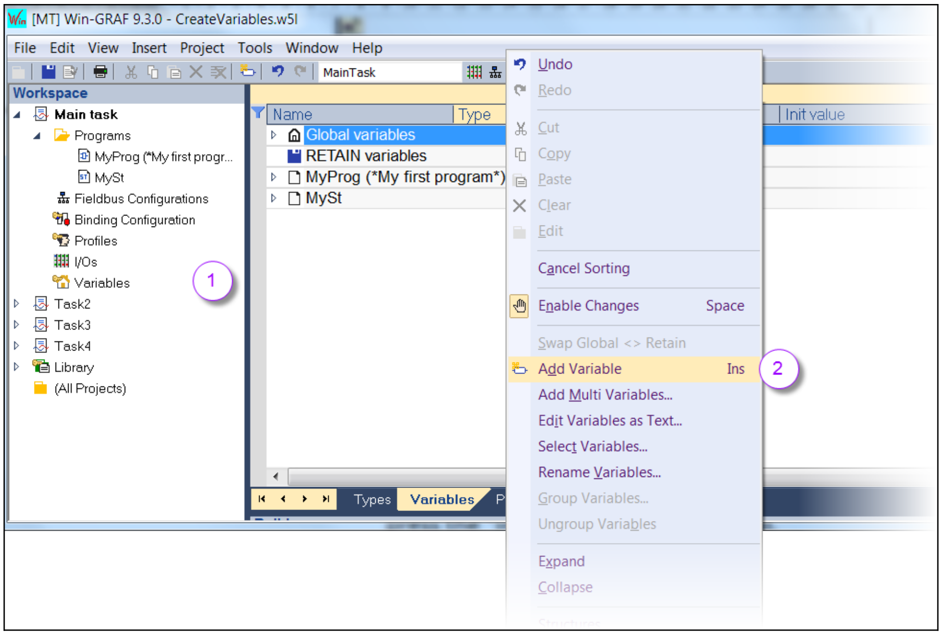
If the workbench encounters a structure syntax error the save process will be aborted and the error type will be listed at the bottom. By double clicking the error message the workbench will jump to the line of the structure where the errors was detected.

13.1.2 Declare Instance of a Structure

The instance of a structure is declared in the same way as a new variable or a instance of a function block.

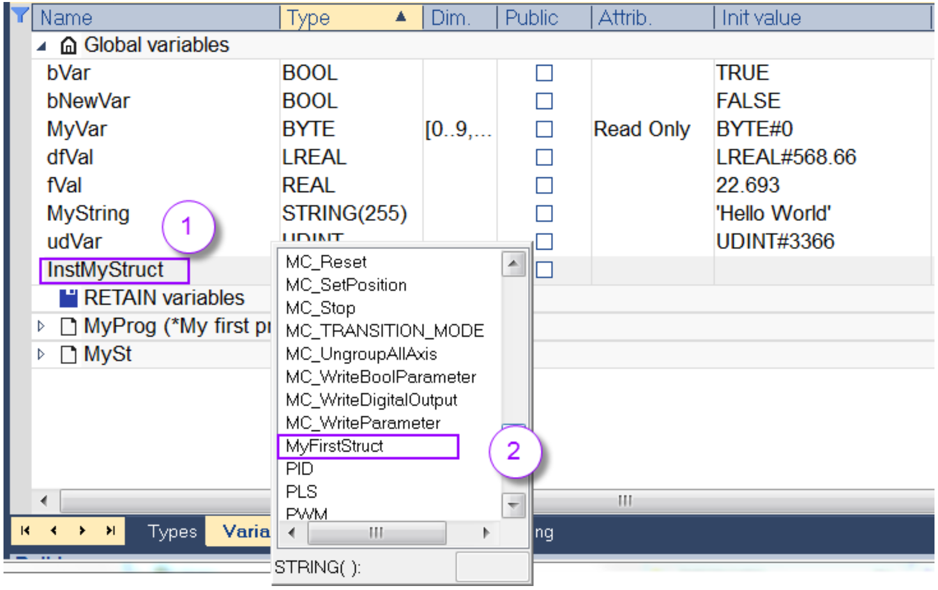
Step 1: Declare a new variable:

1. Open the variable editor by double clicking the *'Variable'* item in the workspace
2. Right click the group to which the instance should be added and select *'Add Variable'*. An alternative way is to click a group and press the *'Insert'* key of the keyboard.



Step 2: Change variable name and data type:

1. Double click the name to rename the structure instance (e.g. InstMyStruct)
2. Double click the *'Type'* field to select the name of the structure type (*'MyFirstStruct'*).



Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
bVar	BOOL		□		TRUE
bNewVar	BOOL		□		FALSE
MyVar	BYTE	[0..9,…	□	Read Only	BYTE#0
dfVal	LREAL		□		LREAL#568.66
fVal	REAL		□		22.693
MyString	STRING(255)		□		'Hello World'
udVar	UDINT		□		UDINT#3366
InstMyStruct			□		
RETAIN variables					
MyProg (*My first pr					
MySt					

Step 4: The structure instance can now be used in the PLC program like any variable. For accessing a member of a structured variable use the following notation:

```
VariableName.MemberName
```

Example:

```
InstMyStruct.bVar
InstMyStruct.diVar
InstMyStruct.siVar
InstMyStruct.udVar
```

13.2 Enums

The enumerated data type also belongs to simple derived data types. It is usually used for naming features or versions instead of using a number code to each version which makes the program easier to read. The initialization value of the enumerated data type is always the value of the first element stated in the enumeration.

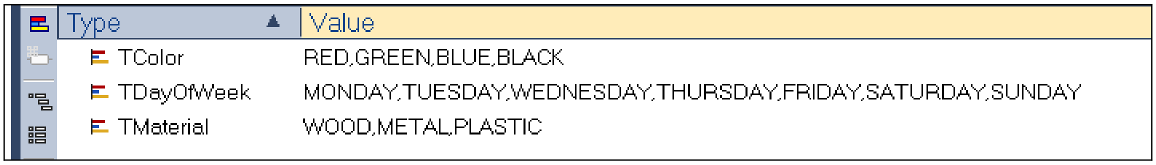
You can define some new data types that are a enumeration of named values. For example:

```
type: LIGHT
values: GREEN, ORANGE, RED
```

Then in programs, you can use one of the enumerated values, prefixed by the type name:

```
Light1 := LIGHT#RED;
```

Variables having enumerated data types can only be used for assignment, comparison, and SEL/MUX functions.



Type ▲	Value
TColor	RED,GREEN,BLUE,BLACK
TDayOfWeek	MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY
TMaterial	WOOD,METAL,PLASTIC

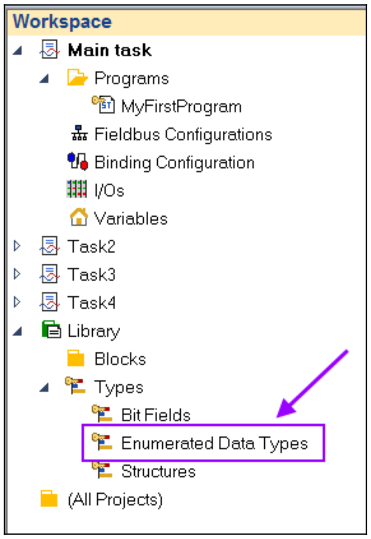
Figure 162: Defined enumerate data types

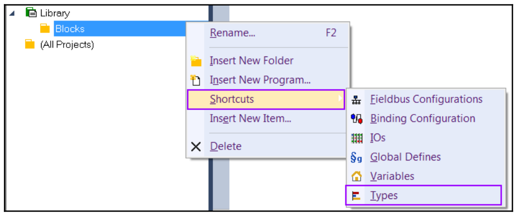
13.2.1 Define a Enumerate Type

An enumerate data type is created as follows:

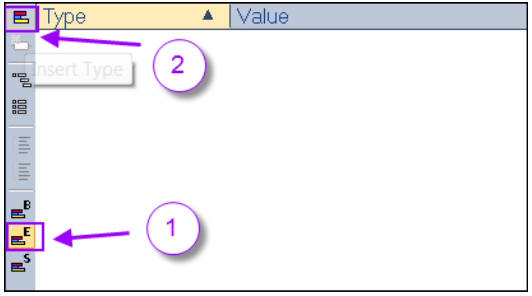
Step 1: Open the enum editor:
Go to the *'Library'* node in the workspace and double click the *'Enumerated*

Data Types' item in the *'Types'* tree to open the editor. If the *'Types'* tree is not visible then enable it by right clicking the *'Library'* node and selecting *'Shortcuts\Types'* from the popup menu.





Step 2: Click the *'Insert Type'* button on the top left or press the *'Insert'* key to add a new enum type.



Make sure that the icon with the superscripted *'E'* is active which indicates that only enum data types can be edited.

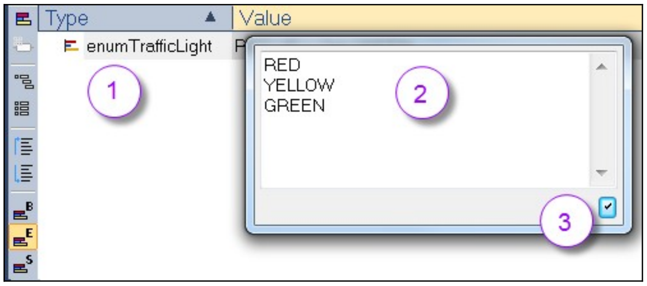
Icon	Data Type
S	Structure
E	Enumeration
B	Bitfield

Step 3: Rename the type and assign type a number of named values:

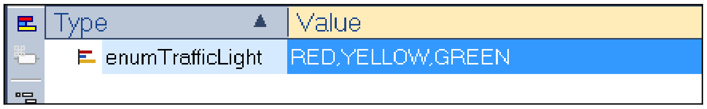
1. Double click the type field to rename it (e.g. enumTrafficLight) and press enter
2. Double click the *'Value'* field to add named values. An editor pops up which allows you to enter the names. Each name must be edited in a new line.
3. Click the button with the check sign to confirm the setting

Note:

- An enum must contain at least two named values.
- An enum should not contain special characters such as #, @, etc.

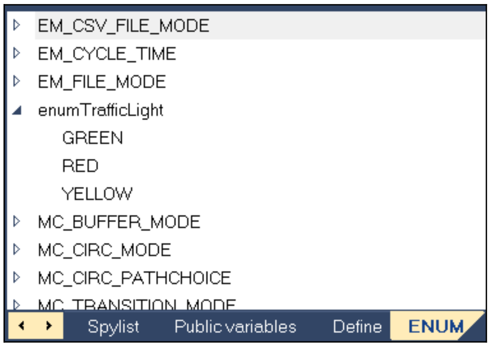


The named values are listed in the *'Value'* field.



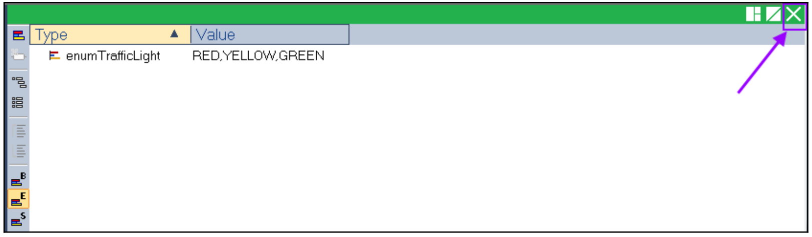
Type	Value
enumTrafficLight	RED,YELLOW,GREEN

The new enum data type is automatically added to the *'Enum'* tab of the information window



- EM_CSV_FILE_MODE
- EM_CYCLE_TIME
- EM_FILE_MODE
- enumTrafficLight
  - GREEN
  - RED
  - YELLOW
- MC_BUFFER_MODE
- MC_CIRC_MODE
- MC_CIRC_PATHCHOICE
- MC_TRANSITION_MODE

Step 4: Close the editor after the enum type has been defined. The new enum type only take effect after the editor has been closed.



Type	Value
enumTrafficLight	RED,YELLOW,GREEN

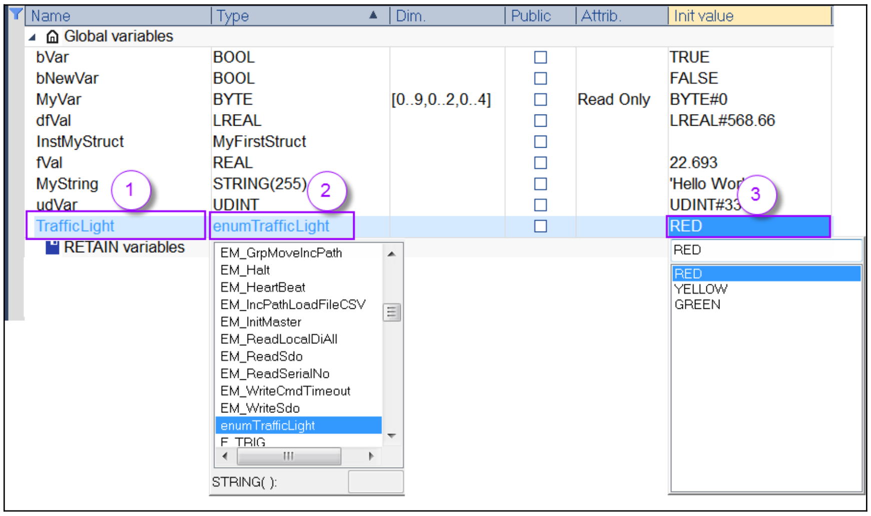
13.2.2 Declare an Enumerate Variable

Enumerate variables are declared in the same way as basic type variables.

Step 1: Add a new variable by opening the variable editor and pressing the *'Insert'* key. Alternatively you can right click the variable editor and select *'Add Variable'* from the popup menu.

Step 2: Configure the variable:

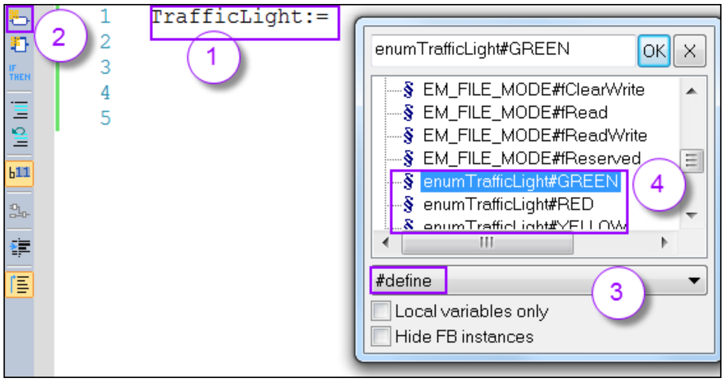
1. Rename the variable by double clicking the name
2. Select the enum data type from the popup list (e.g. *'enumTrafficLigth'*) by double clicking the *'Type'* field
3. Set the initial value by double clicking the *'Init value'* field and selecting one of the named values.

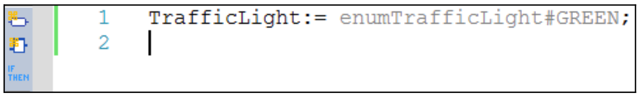


Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
bVar	BOOL		□		TRUE
bNewVar	BOOL		□		FALSE
MyVar	BYTE	[0..9,0..2,0..4]	□	Read Only	BYTE#0
dfVal	LREAL		□		LREAL#568.66
InstMyStruct	MyFirstStruct		□		
fVal	REAL		□		22.693
MyString	STRING(255)		□		'Hello Wor
udVar	UDINT		□		UDINT#33
TrafficLight	enumTrafficLight		□		RED
RETAIN variables					

Step 3: Use of enum variable:

1. Add the new variable name (e.g.*'TrafficLight'*) to the programming editor (e.g. Structured Text) either by typing the name or by drag and drop from the variable editor.
2. Click the *'Insert Variable'* button on the top left to open the variable list
3. Select *'#define'* form the drop box. All the enumerate entries are stored in the *'#define'* category.
4. Scroll to the corresponding enumerate entry and select one entry. Confirm the setting by clicking *'OK'*.

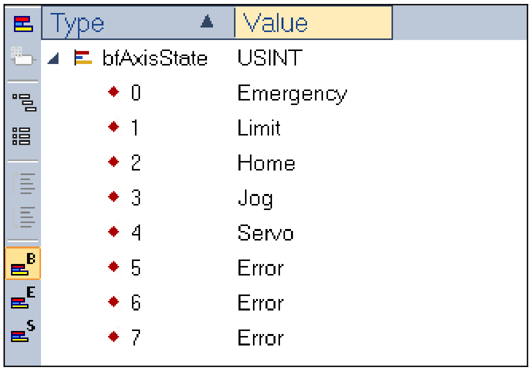




```
TrafficLight:= enumTrafficLight#GREEN;
```

13.3 Bit Field

A bit field is a data structure which consists of a number of adjacent memory locations which have been allocated to hold a sequence of bits, stored so that any single bit or group of bits within the set can be addressed. A bit field is most commonly used to represent integral types of known, fixed bit-width (e.g. SINT, USINT etc.).
The meaning of the individual bits within the field is determined by the programmer. For example a bit can represent a state of a digital input.



Type	Value
bfAxisState	USINT
0	Emergency
1	Limit
2	Home
3	Jog
4	Servo
5	Error
6	Error
7	Error

Figure 163: Bit field definition

You can define new bit field data types derived from integer data types and assign each bit or a group of bit a name (Figure 163). The single bit can be accessed in the program by using the variable name followed by a dot and the bit name:

```
VarName.BitName
```

Example:

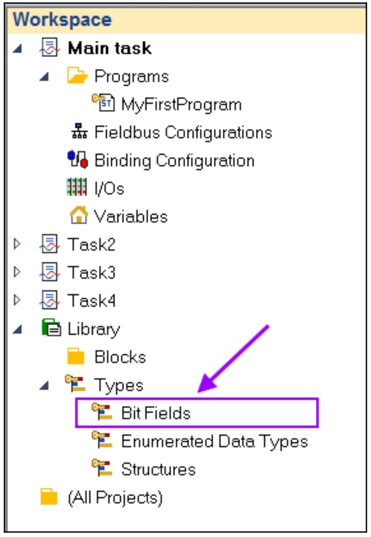
Example refers to Figure 163.

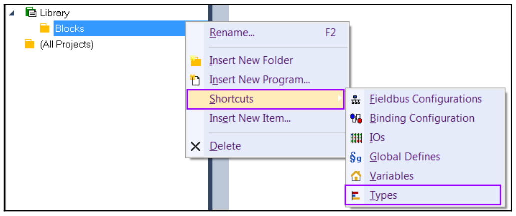
```
//Declare bit field variable
VAR
   AxisStatus : bfAxisState := USINT#0 ;
END_VAR

//Use bit field variable
AxisStatus.Emergency = TRUE;
AxisStatus.Limit = TRUE;
AxisStatus.Home = TRUE;
```

13.3.1 Define a Bit Field Type

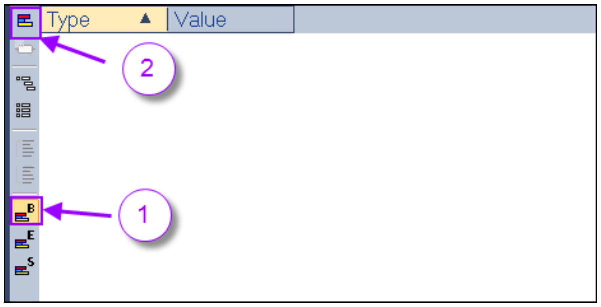
An enumerate data type is created as follows:

Step 1: Open the bit field editor:
Go to the *'Library'* node in the workspace and double click the *'Bit Fields'* item in the *'Types'* tree to open the editor. If the *'Types'* tree is not visible then enable it by right clicking the *'Library'* node and selecting *'Shortcuts\Types'* from the popup menu.





Step 2: Click the *'Insert Type'* button on the top left or press the *'Insert'* key to add a new bit field type. By default a bit field with the name *'BitField1'* of the INT type is being create.

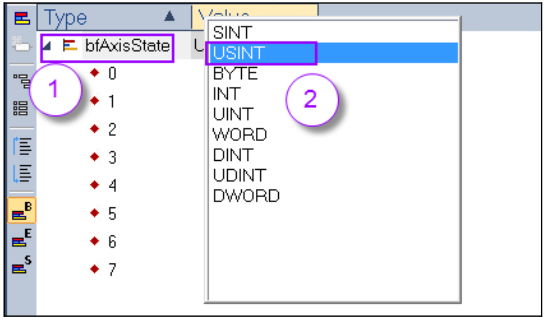


Make sure that the icon with the superscripted *'B'* is active which indicates that only bit field data types can be edited.

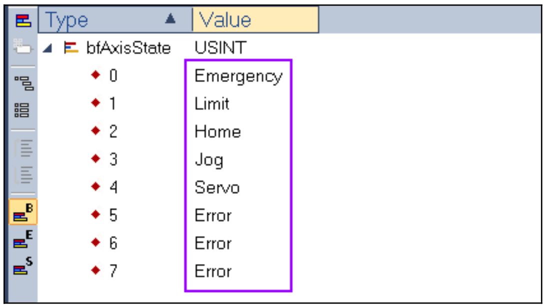
Icon	Data Type
S	Structure
E	Enumeration
B	Bitfield

Step 3: Rename the bit field and assign a integer type:

1. Double click the type field to rename it (e.g. bfAxisState) and press enter
2. Double click the *'Value'* field to select an integral type which represents the bit field length.

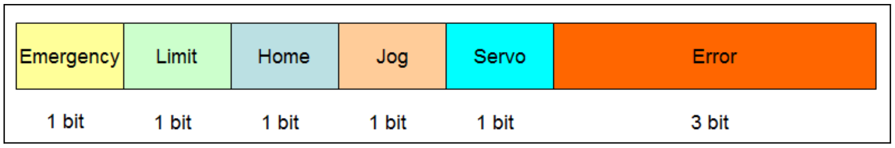


Step 4: Enter a name next to each bit in order to identify the bit in the program. A name can represent a single bit (e.g. Emergency) or a bit field (e.g. *'Error'*).



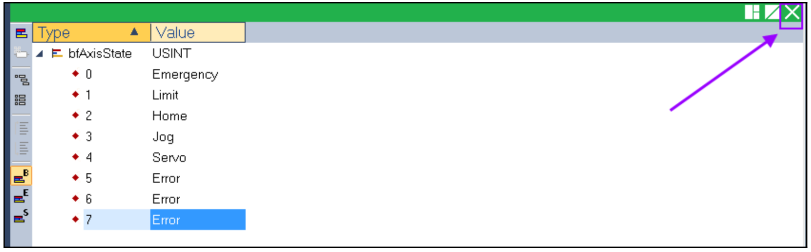
Type	Value
bfAxisState	USINT
0	Emergency
1	Limit
2	Home
3	Jog
4	Servo
5	Error
6	Error
7	Error

The USINT is 8 bit long. The bit field is divided into 6 subfields:



Emergency	Limit	Home	Jog	Servo	Error
1 bit	1 bit	1 bit	1 bit	1 bit	3 bit

Step 5: Close the editor after the bit field type has been defined. The new bit field type can only be used after the bit field editor has been closed.



Type	Value
bfAxisState	USINT
0	Emergency
1	Limit
2	Home
3	Jog
4	Servo
5	Error
6	Error
7	Error

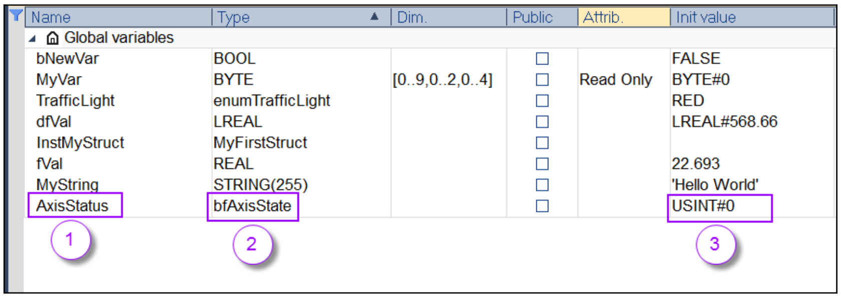
13.3.2 Declare Bit Field Variable

Bit field variables are declared in the same way as basic type variables.

Step 1: Add a new variable by opening the variable editor and pressing the *'Insert'* key. Alternatively you can right click the variable editor and select *'Add Variable'* from the popup menu.

Step 2: Configure the variable:

1. Rename the variable by double clicking the name
2. Select a bit field data type from the popup list (e.g. *'bfAxisState'*) by double clicking the *'Type'* field
3. Set the initial value by double clicking the *'Init value'* field and editing a value.

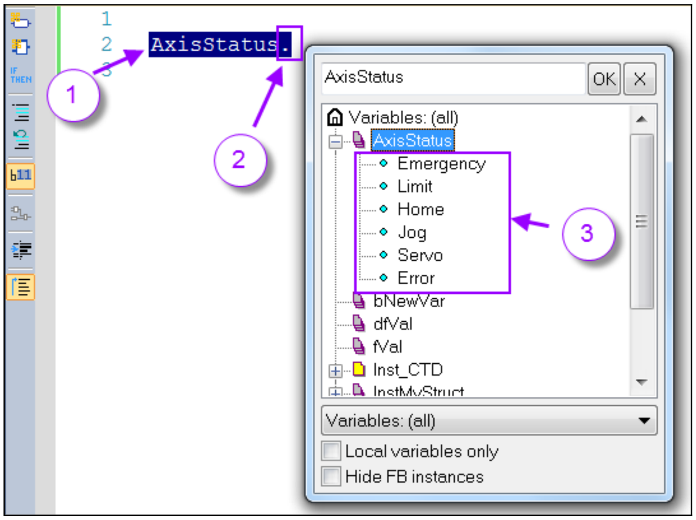


Name	Type	Dim.	Public	Attrib.	Init value
Global variables					
bNewVar	BOOL		□		FALSE
MyVar	BYTE	[0..9,0..2,0..4]	□	Read Only	BYTE#0
TrafficLight	enumTrafficLight		□		RED
dfVal	LREAL		□		LREAL#568.66
InstMyStruct	MyFirstStruct		□		
fVal	REAL		□		22.693
MyString	STRING(255)		□		'Hello World'
AxisStatus	bfAxisState		□		USINT#0

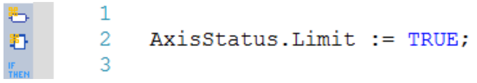
Step 3: Use of bit field variable:

1. Add the new variable name to the programming editor (Structured Text) either by typing the name or by drag and drop from the variable editor.
2. Add a dot after the variable name which causes the variable list to pop up.

3. Select one of the bit name from the variable list and click '*OK*'.



```
AxisStatus.
```



```
AxisStatus.Limit := TRUE;
```

13.4 Function and Function Block

According to the IEC 61131-3 standard, there is a difference between a function and a function block in PLC programming. The main difference involves internal memory. The main difference is that a function block has a permanent internal memory whereas a function memory is only temporary and is being released after the function call so that all internal data are lost. A instance of a function block has to be creates before it can be called whereas a function can be directly called.

Function Block	Function
Memory to store internal data	Temporary memory, no internal data are stored
Function block call requires an function block instance	Function can be directly called
Supports multiple output parameters	Only one output parameter
A function blocks has very often its own, internal machine state and an output to indicate when the function blocks execution is	Functions are synchronous, which means the calling program waits until the function finishes before executing

done. A FB is most likely to be asynchronous.	the next command.

Table 34: Function block and function comparison

A ***function*** can be described as something like an equation or formula that accepts inputs and calculates an output value. Moreover, it always returns the same output value for the same inputs. In contrast, a ***function block*** relies on internal memory. So it is possible to have a different output value with the same inputs because there is another value stored in memory that has an impact on the final output value. Function blocks normally need multiple task cycle to execute. A input triggers the execution of the function block, the output shows the status of execution in each cycle until it has finished. Often, you will have to use the same piece of code in your PLC program multiple times. It could be a function for controlling a valve, a motor etc. With function blocks you can make a function block specific for a motor and use it several times.

Win-GRAF comes with many standard function blocks in the library. The workbench allows you to create your own function blocks (User Defined Function Blocks, UDFB) and it can be programmed in one of the five PLC programming languages.

Example:

By default adding a function block to the programming editor via drag and drop causes the workbench to automatically declare a function block instance in the variable editor and show the instance name in the title bar of the block (Figure 164). In contrary no instance is being generated when a function is added to programming editor.

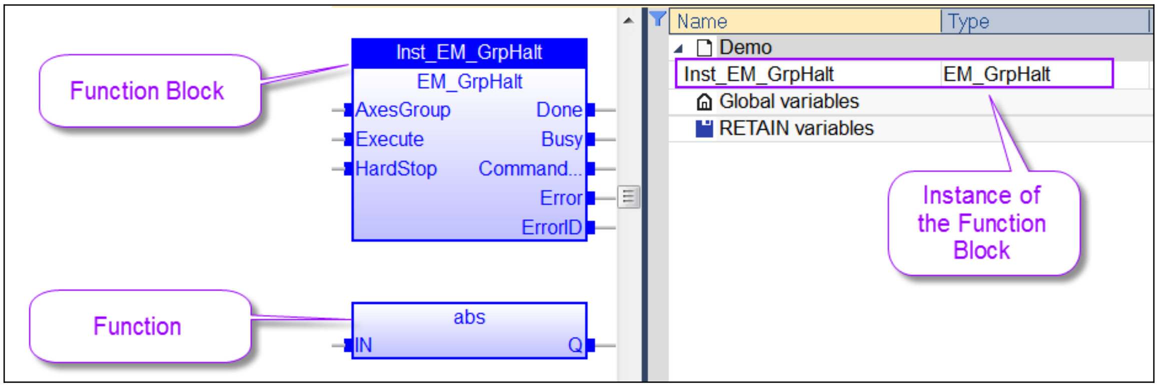
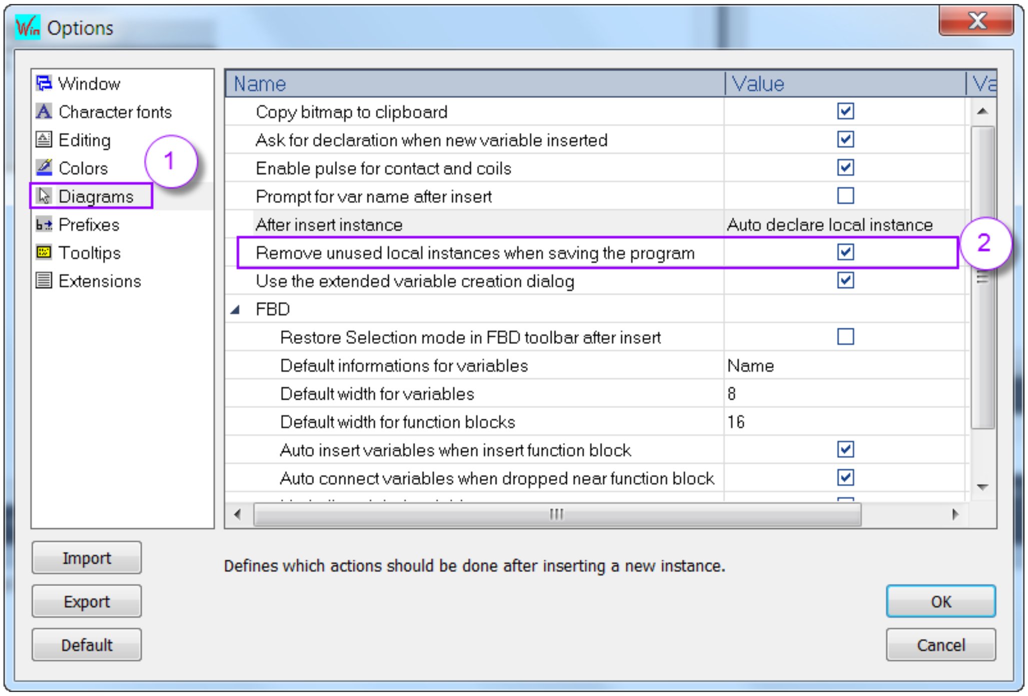


Figure 164: Function block and function

Note:

During PLC logic programming it is important to remember that after deleting a function block from the programming editor to also manually remove its instance from the variable editor. The workbench can be configured to remove any unreferenced variables (Figure 165).



Options

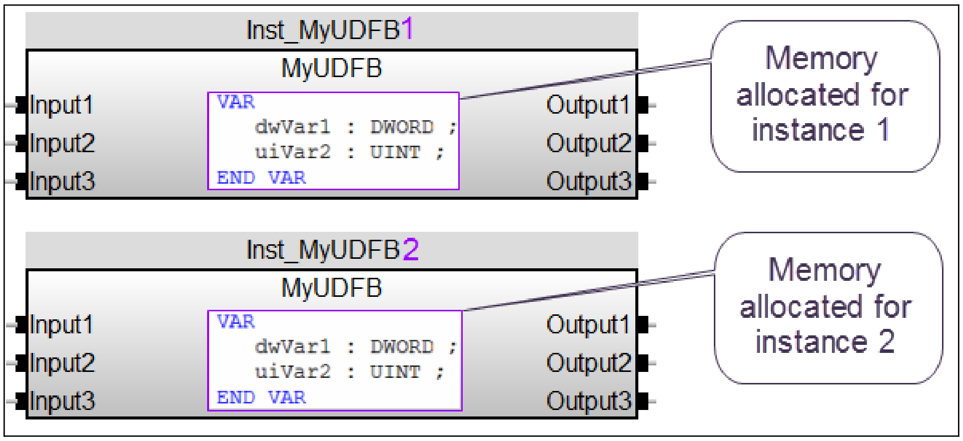
- Window
- Character fonts
- Editing
- Colors
- Diagrams
- Prefixes
- Tooltips
- Extensions

Name	Value
Copy bitmap to clipboard	☑
Ask for declaration when new variable inserted	☑
Enable pulse for contact and coils	☑
Prompt for var name after insert	□
After insert instance	Auto declare local instance
Remove unused local instances when saving the program	☑
Use the extended variable creation dialog	☑
FBD	
Restore Selection mode in FBD toolbar after insert	□
Default informations for variables	Name
Default width for variables	8
Default width for function blocks	16
Auto insert variables when insert function block	☑
Auto connect variables when dropped near function block	☑

Defines which actions should be done after inserting a new instance.

Figure 165: Automatically remove a unreferenced instance from the variable editor

The Win-GRAF workbench handles all private variables declared inside a function block as statically allocated in memory. Each time a function block is initiated, its private variables are duplicated for the declare instance, which means for each function block instance a memory for the private variables will be allocated. The private variables of a function block instance are independent of private variables of other instances and just store internal data of their instance and no data exchange takes place between private variables across different instances. From outside, only input and output parameters of a function block are accessible, that is, the private variables of a function block are hidden to the user of the function block. A privates data for example may represent the state of the function block state machine. The state of the state machine of each instance of a function block differs depending on when the instance has been activated and the device it is representing (e.g. servo motor).



```
VAR
    dwVar1 : DWORD ;
    uiVar2 : UINT ;
END VAR
```

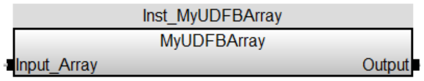
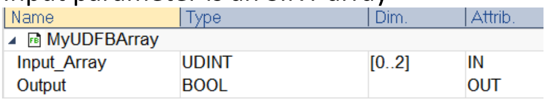
Figure 166: Two instance of the same function block

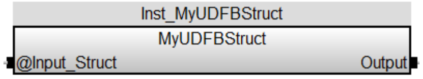
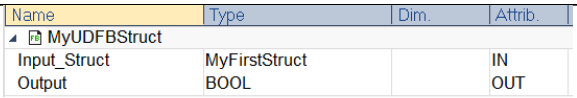
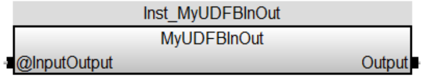
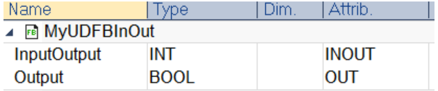
13.4.1 Define Function Block

13.4.1.1 Function Block Input and Output Parameters

Function blocks three type of external variables:

1. Input variables (IN):
   - Input parameters can only be read by the function block and data is passed by the calling program to function block.
   - The input variables are listed on the left side of the function block in a graphical representation.
   - Exception: Complex parameters (array, structures) should always be declared as input parameters (IN). Internally complex parameters can be read and modified by the function block
   - A maximum of 32 input parameters are allowed
2. Output variables (OUT)
   - The function block writes data to the output. Output variables are provided by the function block to the calling program. The calling program can only read this output and can not directly modify its data value.
   - In a graphical representation output parameters are shown on the right hand sight of the function block.
   - Only simple data types can be output parameters (Table 30). If the user needs to create a function block with a complex structure (array or structure) as an output parameter then this structure has to be declared as input parameter. In this case internally the function block has read/write access to the parameter.
   - A maximum of 32 output parameters are allowed
3. Input/output variables (IN_OUT)
   - The function block can read and directly modify the IN_OUT variables
   - In a graphical representation IN_OUT parameters are shown on the left hand sight of the function block. The IN_OUT parameter is indicated by the '@' symbol in front of name.
   - Only simple data types can be declared as input/output parameters.

<table>
<tr><th>Function Block</th><th>Description</th></tr>
<tr><td>
</td><td>Input parameter is an SINT array

<table>
<tr><th>Name</th><th>Type</th><th>Dim.</th><th>Attrib.</th></tr>
<tr><td>MyUDFBArray</td><td></td><td></td><td></td></tr>
<tr><td>Input_Array</td><td>UDINT</td><td>[0..2]</td><td>IN</td></tr>
<tr><td>Output</td><td>BOOL</td><td></td><td>OUT</td></tr>
</table></td></tr>
<tr><td></td><td>Input parameter is a user define structure type</td></tr>
</table>

 Inst_MyUDFBStruct MyUDFBStruct @Input_Struct Output	
	Input parameter is IN_OUT of a simple data type 

Name	Type	Dim.	Attrib.
MyUDFBStruct			
Input_Struct	MyFirstStruct		IN
Output	BOOL		OUT

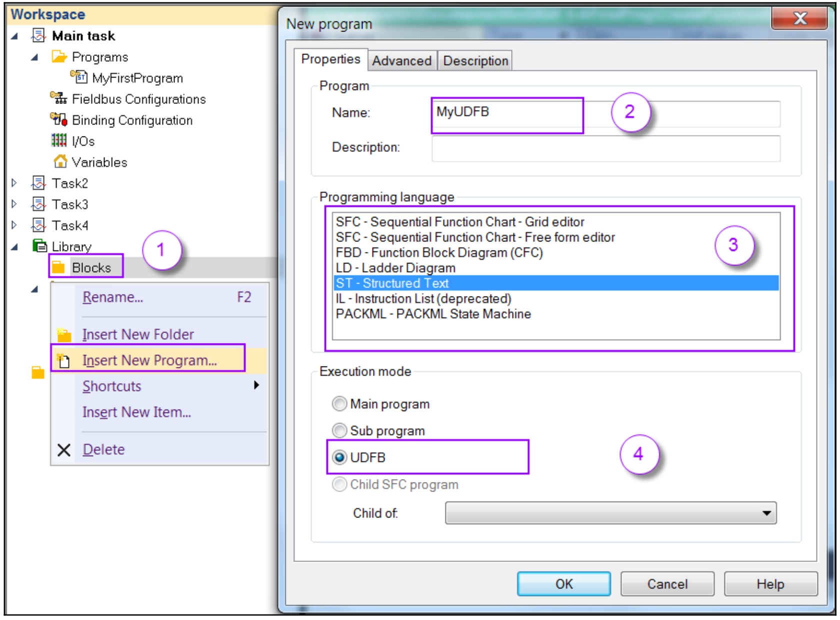
Name	Type	Dim.	Attrib.
MyUDFBInOut			
InputOutput	INT		INOUT
Output	BOOL		OUT

Table 35: Function block with different input parameter type

This section explains how to use the wizard to create a new function block.

Step 1: Add empty function block program editor:

1. Right click the *'Blocks'* in the *'Library'* tree of the workspace and select *'Insert New Program..'*. A dialog for selecting the program type pops up.
2. Enter the name of the function block
3. Enter the programming language to be use for programming the function block body code
4. Select UDFB (User Defined Function Block)
5. Click *'OK'*. A dialog for editing the function block parameters pops up.



Program

Name: MyUDFB

Description:

Programming language

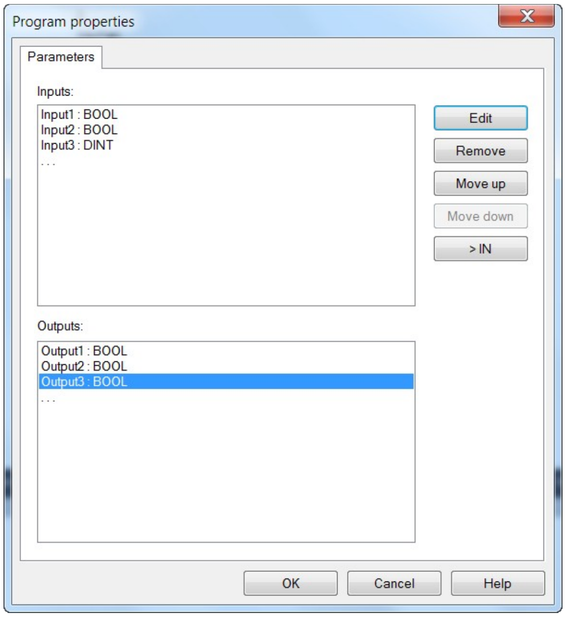
- SFC - Sequential Function Chart - Grid editor
- SFC - Sequential Function Chart - Free form editor
- FBD - Function Block Diagram (CFC)
- LD - Ladder Diagram
- ST - Structured Text
- IL - Instruction List (deprecated)
- PACKML - PACKML State Machine

Execution mode

○ Main program
○ Sub program
● UDFB
○ Child SFC program

Child of:

Step 2: Declare the input and output parameters for the function block. The input and output variables can also be declared in the variable editor of the function block program after it has been created.



Program properties

Parameters

Inputs:

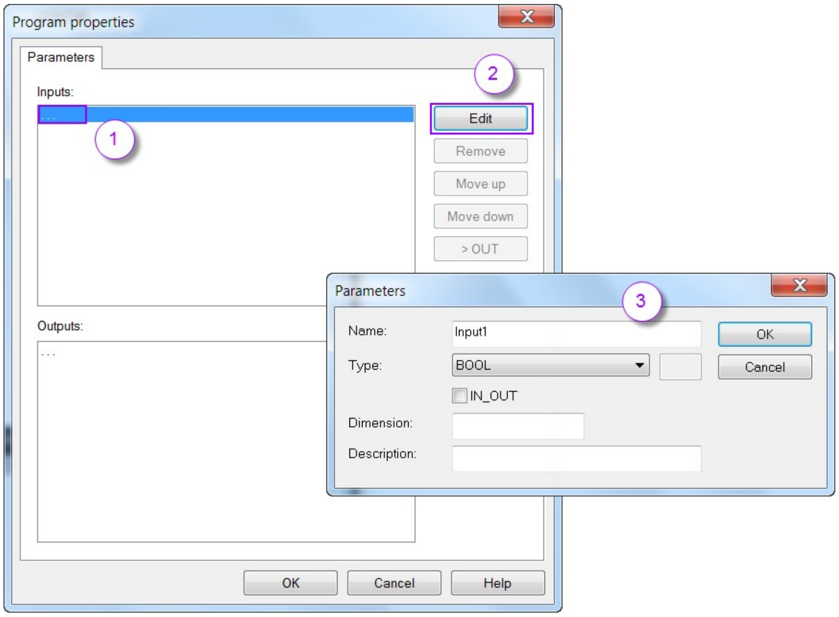
Input1 : BOOL
Input2 : BOOL
Input3 : DINT
...

Outputs:

Output1 : BOOL
Output2 : BOOL
Output3 : BOOL
...

Edit input parameter:

1. Click the *'...'* in the input area
2. Click *'Edit'* or double click the entry *'...'* in the input area
3. Enter the name and data type for the input parameter and click *'OK'*



Program properties

Parameters

Inputs:

...

Outputs:

...

Parameters

Name: Input1

Type: BOOL

□ IN_OUT

Dimension:

Description:

Note:

- If the *'IN_OUT'* check box is not checked, the input can be only read by the function block. If checked, then the block can change the value of the input. An *'IN_OUT'* parameter must be a single data type therefore can neither be an array nor a structure.

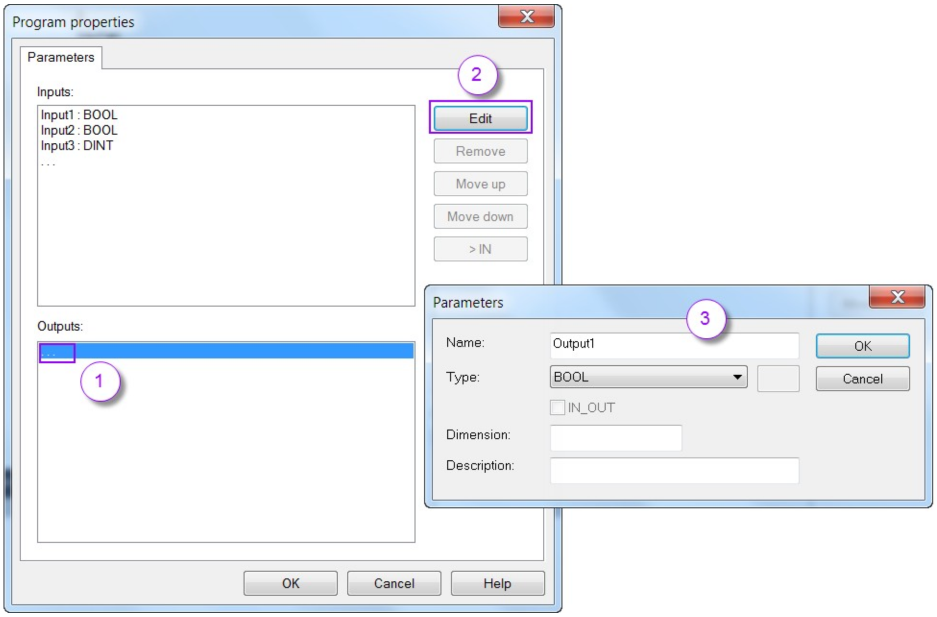
- Parameters being arrays of structures must always be declared as INPUTs. However, they are always considered implicitly as IN_OUT, which means the function block can read data from and write data to the array or structure. Declaring complex parameters for a function block can have some limitations if the *'Complex variables in a separate segment'* option is not enabled in the project settings.

Edit output parameter:

1. Click the *'...'* in the output area

2. Click *'Edit'* or double click the entry *'...'* in the output area

3. Enter the name and data type for the output parameter and click *'OK'*



Inputs:
Input1 : BOOL
Input2 : BOOL
Input3 : DINT
...

Outputs:
...

Name: Output1
Type: BOOL
□ IN_OUT
Dimension:
Description:

Step 3: The input and output parameter list can be modified after they have been edited.

Button	Description
Edit	Click this button to change the parameter name or its data type
Remove	Press this button to remove a the selected parameter from the list
Move up	This button moves a parameter entry in the list one position up
Move down	This button moves a parameter in the list one line down
>IN	Moves am output parameter from the output area to the input area

>OUT	Moves a input parameter from the input area to the output area

All used defined function blocks are automatically added to the *'Library'* category in the *'Blocks'* window (Figure 167). It is necessary to close the programming editor of the function block (Window with the green frame) in order for it to be listed in the *'Library'* category. The used defined function blocks can be used in the same way as the standard function blocks provided by the Workbench by drag and drop to the programming editor.

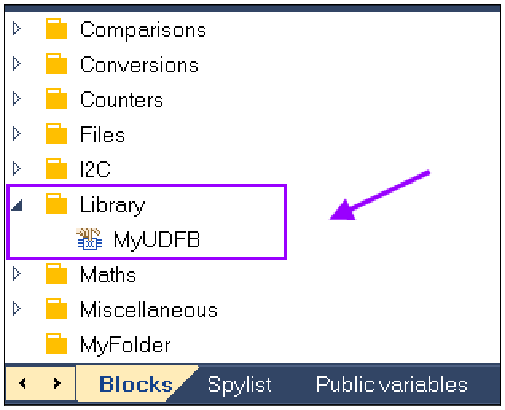


Figure 167: User defined function block

In the library manager of Win-GRAF the user defined function blocks are indicated by a special icon which makes it more easier to distinguish from the standard function blocks (Table 36). In addition the user defined function blocks are shown in the function block diagram as a different color (Table 37).

Function Block Icon	Description
▦	Win-GRAF preinstalled function block
▦	Indicates user defined function block

Table 36: Function block icons

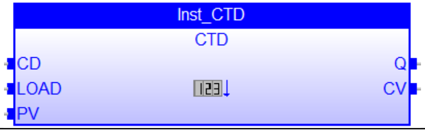
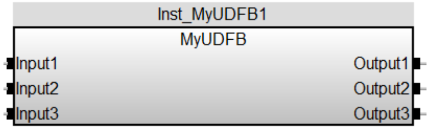
Function Block	Description
	Win-GRAF preinstalled function block
	User defined function block (UDFB)

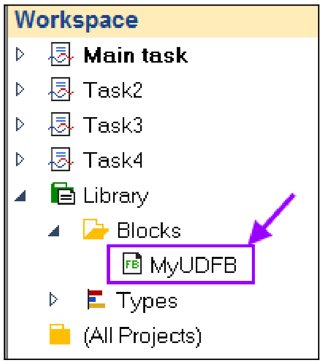
Table 37: Function block

13.4.1.2 Define Function Block Variables

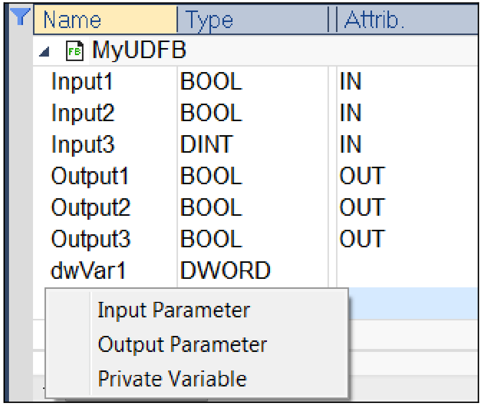
A function block has three types of variables: Input, output and local variables. The previous section describes the declaration of the input and output variables of a function block by using the function block wizard. The variable editor of the workbench allows the user to directly declare new input and output variables or remove existing after the function blocks wizard has been closed. Local variables can only be declared via the variable editor.

The following procedure shows how to add a private, input or output variable to the function block:

Step 1: Open the function block program editor:
Double click the function block name in the workspace.



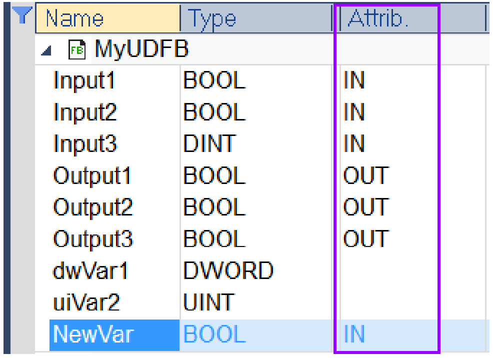
Step 2: Add a new variable:
Click on the variable editor and press the *'Insert'* to add a new variable. A window pops up which allows you to select the type of variable to enter: Input, output or private variable.



Name	Type	Attrib.
MyUDFB		
Input1	BOOL	IN
Input2	BOOL	IN
Input3	DINT	IN
Output1	BOOL	OUT
Output2	BOOL	OUT
Output3	BOOL	OUT
dwVar1	DWORD	

The attribute column *'Attrib.'* indicates the type of variable:

- *'IN'* - input variable
- *'OUT'* - output variable
- Empty attribute indicates a private variable



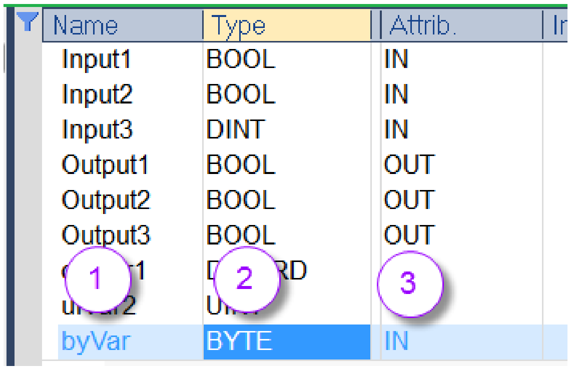
Name	Type	Attrib.
MyUDFB		
Input1	BOOL	IN
Input2	BOOL	IN
Input3	DINT	IN
Output1	BOOL	OUT
Output2	BOOL	OUT
Output3	BOOL	OUT
dwVar1	DWORD	
uiVar2	UINT	
NewVar	BOOL	IN

The basic setting of the attribute field can not be changed via the variable editor. This has to be done by using

1. the class wizard (right click the class name in the workspace tree and select *'Parameters...'* from the popup menu)
2. the variable text editor (right click the variable in the variable editor and select *'Edit Variable as Text...'*)

Step 3: Set variable name and type:

1. Set name: Double click the *'Name field'* and enter a new name
2. Set variable type: Double click the *'Type'* field and select a type from the drop list
3. Set the attribute: Only the *'IN'* attribute can be changed to *'INOUT'*. Double click the *'Attrib.'* field to change the attribute from *'IN'* to *'INOUT'* or vice versa.



Name	Type	Attrib.	Ir
Input1	BOOL	IN	
Input2	BOOL	IN	
Input3	DINT	IN	
Output1	BOOL	OUT	
Output2	BOOL	OUT	
Output3	BOOL	OUT	
dwVar1	DWORD		
uiVar2	UINT		
byVar	BYTE	IN	

The variables can now be used inside the user defined function block in the same way as in a normal PLC program.

13.4.1.3 Implementing Function Block Logic

Programming a function block is very much the same as writing a normal PLC program. The programming can be done in one of the five programming language defined by IEC 61131.

It is important to pay attention to a few point:

- Do not write or change input data if it is of a simple data type. Only the calling program has write access to it.
- Input parameters of complex structure types (e.g. arrays, structure) are regarded by the workbench as an input/output type and therefore the function block body and the calling program has read and write access to it.
- Only the function block body has write access to a output parameter of a simple data type. The calling program has only read access.
- Use the private data to store some information from the current cycle which is needed by the function block for the next cycle (e.g. state of state machine, previous input, etc.)
- Declare a parameter of a simple data type as input/output when both the calling program and the function block body has write access to it.
- A maximum of 32 input and 32 output parameters are supported for a function block

Example:

In the following a function block is being implemented which triggers an output when a rising edge is being detected at its input (Figure 168).

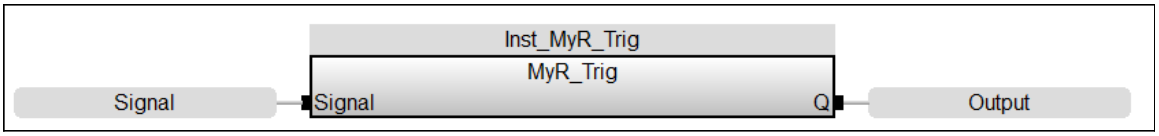
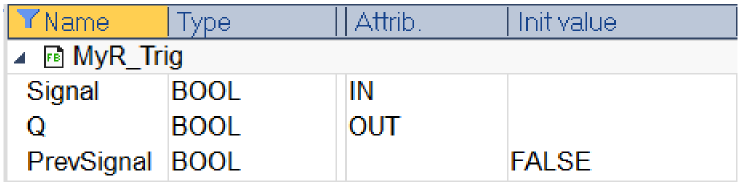


Figure 168: Rising edge function block

Three parameter are being declared for the function block body (Figure 169):

- One input parameter which shows the active level of the signal,
- One output parameter which indicates to the calling program when a rising input signal has been detected
- One private variable which records the signal level of the current cycle.



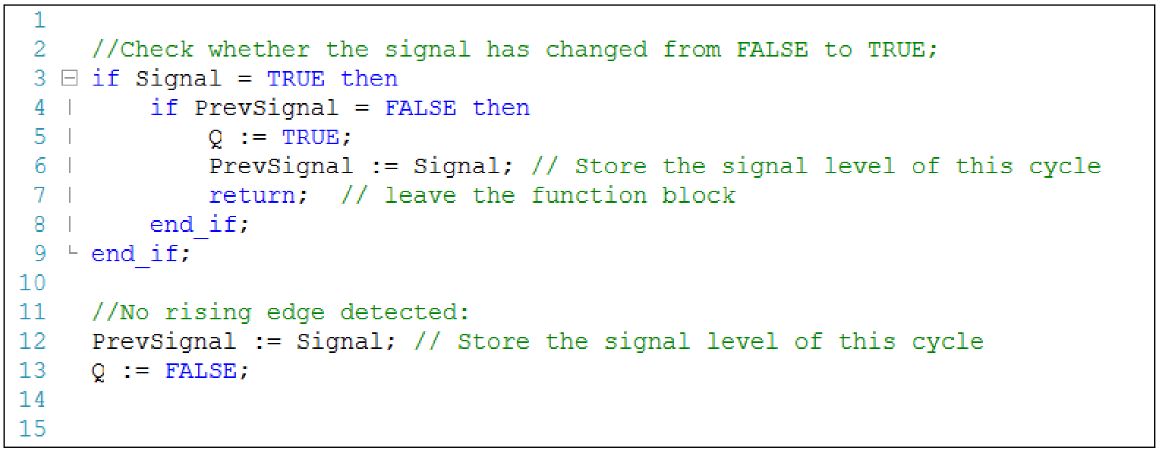
Name	Type	Attrib.	Init value
MyR_Trig			
Signal	BOOL	IN	
Q	BOOL	OUT	
PrevSignal	BOOL		FALSE

Figure 169: Variable declaration

Function body logic description (Figure 170):

- If the signal level of the previous cycle was off (PrevSignal := FALSE) and the input signal of the current signal is active (Signal := TRUE), then a rising edge has been detected and the output variable will be activate (Q := TRUE).

- If the input signal was active in the previous cycle (PrevSignal := TRUE) and is still active in the current cycle (Signal := TRUE) then no rising edge has been detected with the result that the output is to false (Q := FALSE).
- If the input signal for the previous and current cycle is inactive (PrevSignal := FALSE ; Signal := TRUE) then the output will be set to false (Q := FALSE).



```

//Check whether the signal has changed from FALSE to TRUE;
if Signal = TRUE then
    if PrevSignal = FALSE then
        Q := TRUE;
        PrevSignal := Signal; // Store the signal level of this cycle
        return;  // leave the function block
    end_if;
end_if;

//No rising edge detected:
PrevSignal := Signal; // Store the signal level of this cycle
Q := FALSE;


```

Figure 170: Function block body

Creating an instance of the function block

The user defined function block is automatically being added to the library category of the *'Blocks'* tab. By simple dragging and dropping the function block over the program editor an instance of the function block is being created. Next the input and output have to be linked to variables. This can be done by simple dragging a variable from the variable editor to the input or output area of the function block (Figure 171).

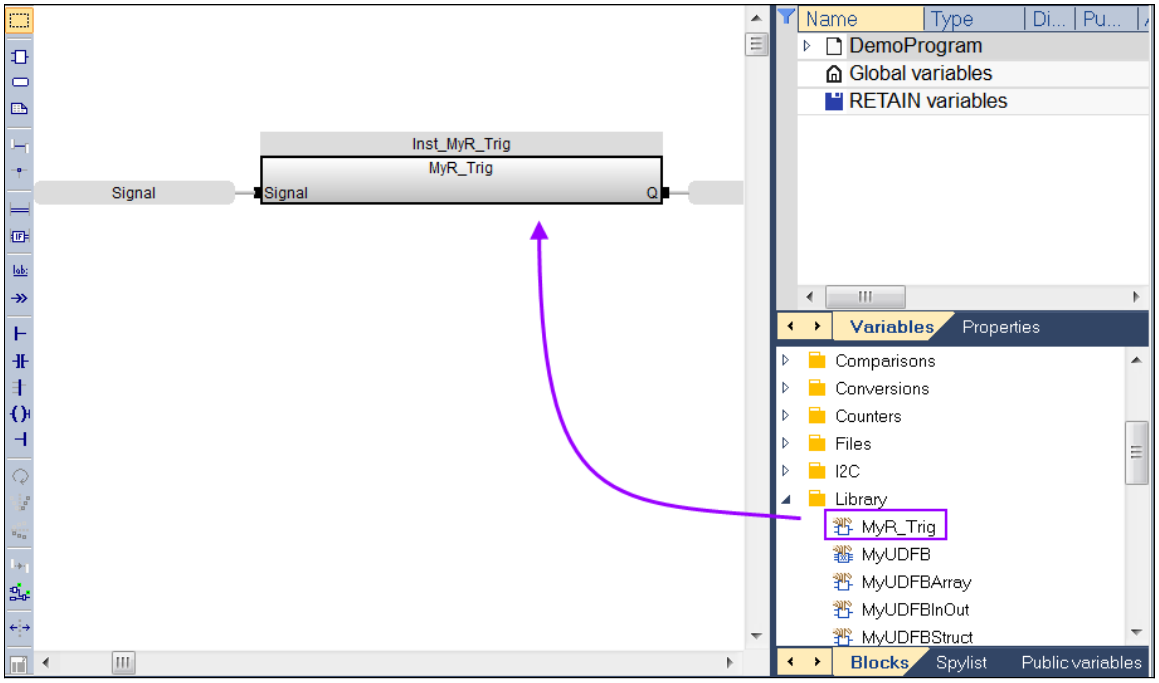


Figure 171: Creating a function block instance

13.4.2 Define Function

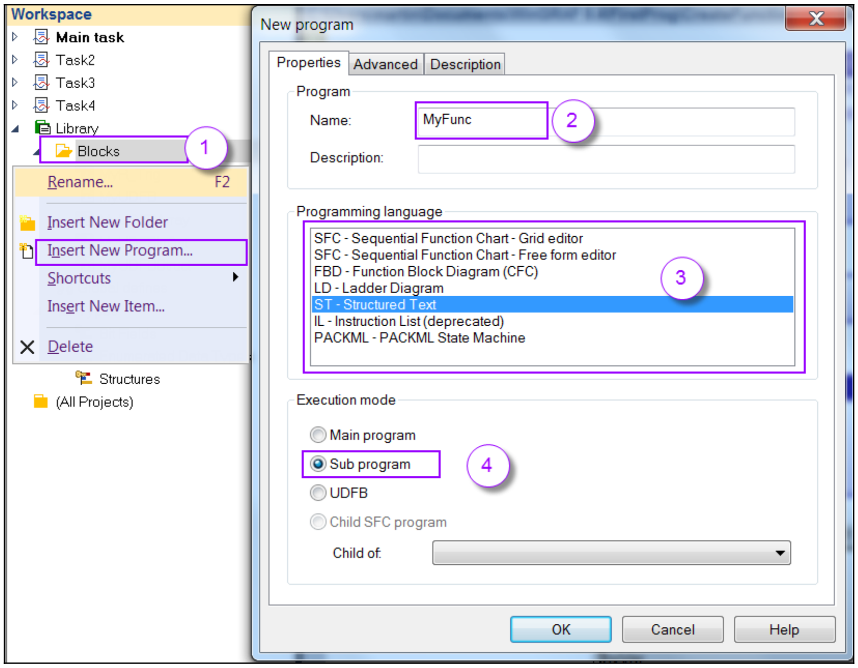
13.4.2.1 Function Input and Output Parameters

A function is defined as a program organization unit which, when called, yields exactly one result and arbitrarily many additional output elements (the function result can be multi-valued, e.g., an array or structure). Functions contain no internal state information, therefore a function execution with the same input parameters shall always yield the same output values .
The preinstalled Win-GRAF functions has one or more input parameters but only one output parameter. The Win-GRAF workbench allows the user to defined functions with multiple in- and outputs.
The procedure of defining a function is very similar to a function block definition. Therefore this section only describes very briefly the procedure. For more information read the description of the function block definition procedure.

The steps explains how to use the wizard to create a new function.

Step 1: Add empty function block program editor:

1. Right click the *'Blocks'* in the *'Library'* tree of the workspace and select *'Insert New Program...'*. A dialog box pops up.
2. Enter the name of the function
3. Select the programming language to be use for programming the function code
4. Select *'Sub program'*
5. Click *'OK'*. A window will pop up which allows you to define the input and output parameters of the function.



New program

Properties | Advanced | Description

Program

Name: MyFunc

Description:

Programming language

- SFC - Sequential Function Chart - Grid editor
- SFC - Sequential Function Chart - Free form editor
- FBD - Function Block Diagram (CFC)
- LD - Ladder Diagram
- ST - Structured Text
- IL - Instruction List (deprecated)
- PACKML - PACKML State Machine

Execution mode

○ Main program

● Sub program

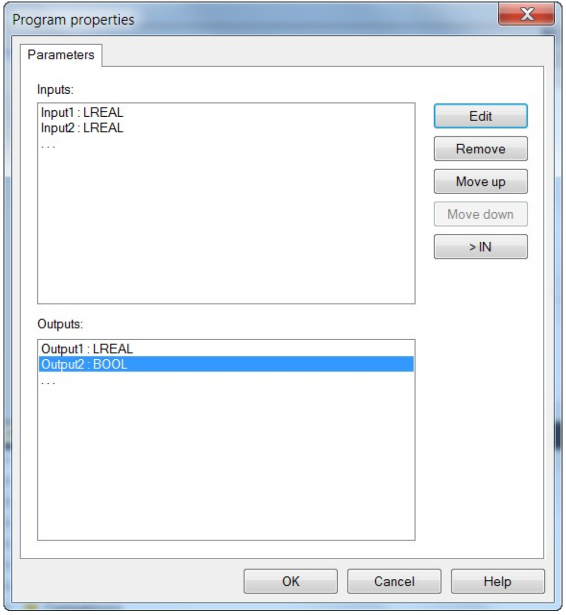
○ UDFB

○ Child SFC program

Child of:

Step 2: Define the input and output parameters for the function in the function parameter editor. The input and output variables can also be defined in the variable editor of the function program after it has been created. Refer to the function block section to get more information about the function parameter editor.

Click *'OK'* after the parameters have been edited.



Program properties

Parameters

Inputs:

Input1 : LREAL
Input2 : LREAL
...

Outputs:

Output1 : LREAL
Output2 : BOOL
...

Step 3: Edit the source code for the function body (see function block description).

The new function will be also automatically added to the *'Library'* node in the workspace and to the category in the *'Blocks'* window after the programming editor for the function body has been closed (Figure 172).

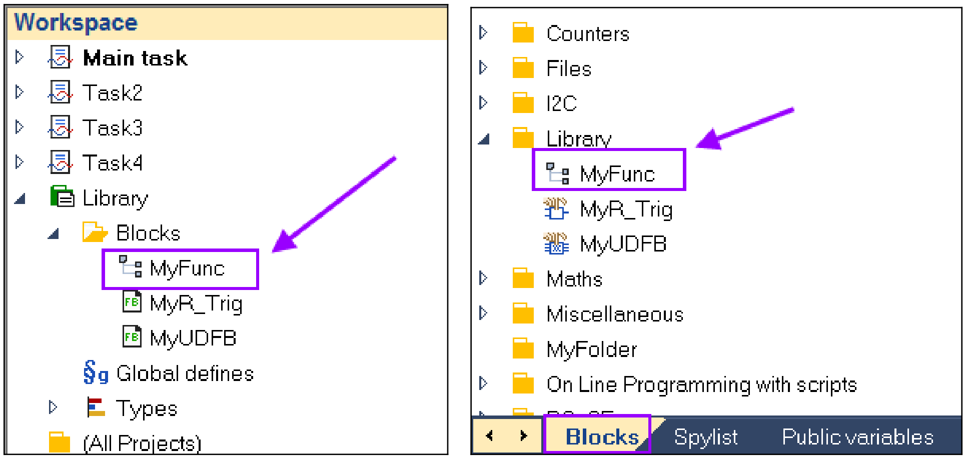


Figure 172: User defined function added to the library directory

The library manager of Win-GRAF assign different icons and diagram colors to the preinstalled and user defined functions (Table 38, Table 39).

Function Block Icon	Description
⊡	Win-GRAF preinstalled function
⊞	User defined function

Table 38: Function icons

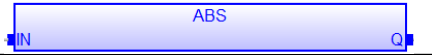
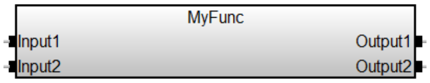
Function Block	Description
	Win-GRAF preinstalled function
	User defined function

Table 39: Preinstalled and user define function diagrams

14 Backup Management

14.1 Save Project Backup to Local PC

It is suggested to regularly make a backup of your project to prevent any accidental loss of data. The workbench allows you to save the project as a zip file to memory (Figure 173). The project zip file can be directly loaded by the workbench. During the loading process the zip file will be automatically unzipped.

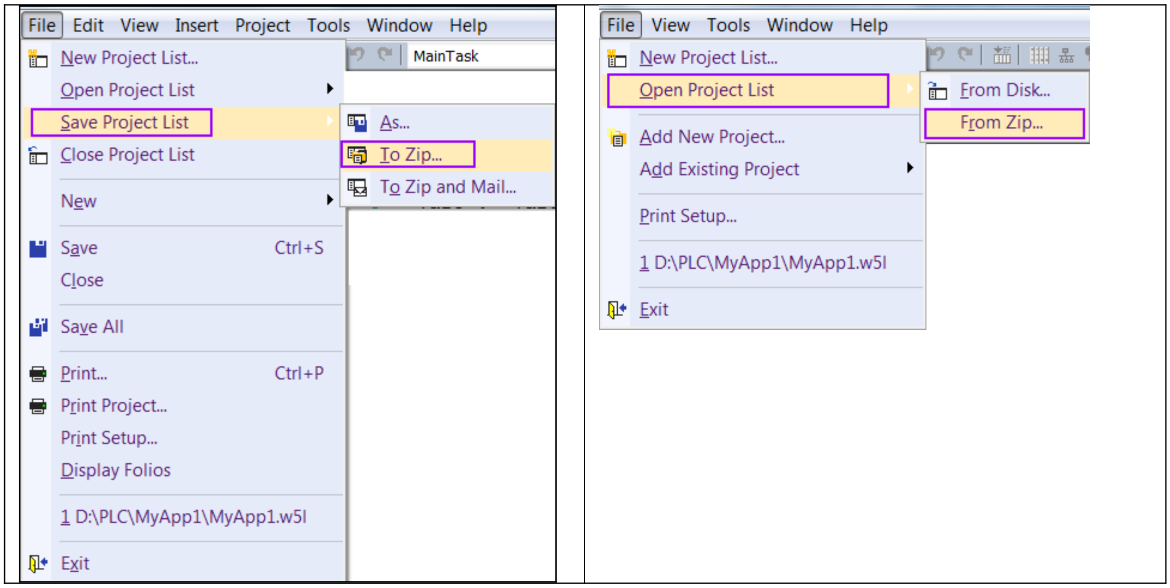


Figure 173: Saving (left) and open (right) a backup file

14.2 Save Project to Runtime Target

The workbench enables you to embed on the target runtime the source code of the project, so that it can be uploaded later. Source code is filtered and zipped in order to reduce backup memory requirements. As sending source to the target may involve a significant download time, it is up to you to explicitly activate the download command.

<u>**Note:**</u>
The project can only be save to the runtime if it has been created in single-tasking environment (chapter: Single-Tasking). Projects created in a multi-tasking environment (chapter: Multi-Tasking) can not be saved to the runtime.

Those commands are available from the contextual popup menu in the workspace window:

- Save Project to Target: zip project source files and send them to the target.
- Open project from Target: upload zipped source file from the target and rebuild the

project.

When saving the project to the target you have to specify the address and communication parameters of the remote runtime system. The following options enable you to send more or less optional information with project source code:

- Symbol table: the symbol table will be required after upload for monitoring variables. If you do not embed the symbol table, you will have to recompile the uploaded application for reading or writing variables.
- Debug information: this file will be required after upload for step by step debugging and use of breakpoints. If you do not embed the symbol table, you will have to recompile the uploaded application for stepping the application.
- Spy lists: these are all files created with the Watch Window, such as lists and recipes.
- Wizard settings: these are current settings of wizards such as the Monitoring Application Builder.
- Project history: this is the list of modifications entered in the project.
- Comment texts: all comments entered for variables, programs. Comments within the programs are always saved.
- Bitmaps and icons: these are all BMP, GIF, JPG or ICO files stored in the project folder and possibly used in monitoring views.
- Referenced OEM library elements: definition of all the library elements (*'C'* functions and blocks, I/Os, profiles) actually used in the project.

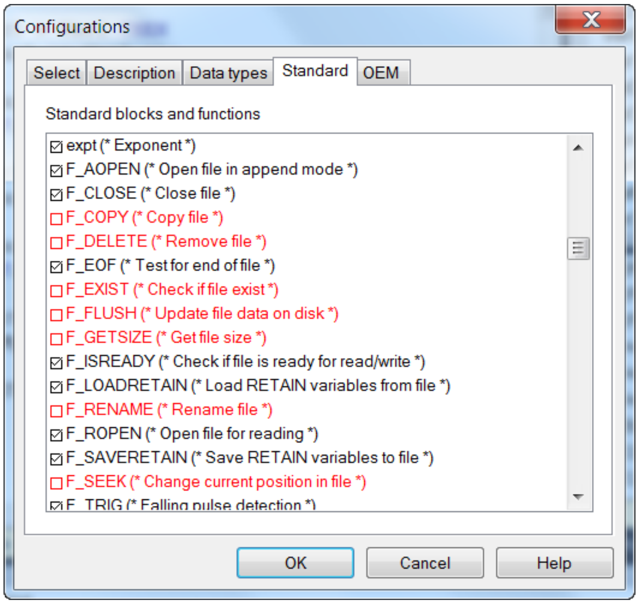
In addition to standard files, you can specify some extra files to be downloaded. In that case, all of them will be located in the loaded project folder after upload, even if they are originally located in other folders.

Removing some options enables you to reduce the size of embedded source code.

15 Target Runtime Configuration

The variable types, functions and function blocks provided by the workbench are not all supported by every Win-GRAF runtime versions. It is therefore suggested to first upload the target runtime configuration before compiling the application, to enable the compiler to check for any unsupported data types. The runtime configuration list all the supported data types and the runtime type and version number. The compiler uses this configuration information for the compiling process to determine whether the source code meets the specifications of the target runtime.

After the configuration has been uploaded the variable types, functions and function blocks not supported by the runtime are marked in red (Figure 174). The compiler will output an error if it encounters an unsupported data types.



Standard blocks and functions

☑ expt (* Exponent *)
☑ F_AOPEN (* Open file in append mode *)
☑ F_CLOSE (* Close file *)
□ F_COPY (* Copy file *)
□ F_DELETE (* Remove file *)
☑ F_EOF (* Test for end of file *)
□ F_EXIST (* Check if file exist *)
□ F_FLUSH (* Update file data on disk *)
□ F_GETSIZE (* Get file size *)
☑ F_ISREADY (* Check if file is ready for read/write *)
☑ F_LOADRETAIN (* Load RETAIN variables from file *)
□ F_RENAME (* Rename file *)
☑ F_ROPEN (* Open file for reading *)
☑ F_SAVERETAIN (* Save RETAIN variables to file *)
□ F_SEEK (* Change current position in file *)
☑ F_TRIG (* Falling pulse detection *)

Figure 174: Target system configuration

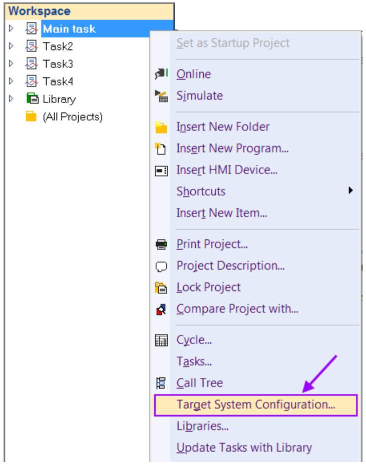
Procedure for uploading the configuration data of the target platform:

Step 1: Make sure the runtime is executing.

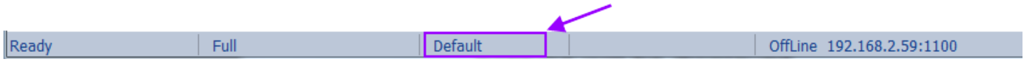
Step 2: Open the configuration dialog

The dialog can be opened in the following two ways:

1. Right click any task in the workspace and select *'Target System Configuration...'* from the popup



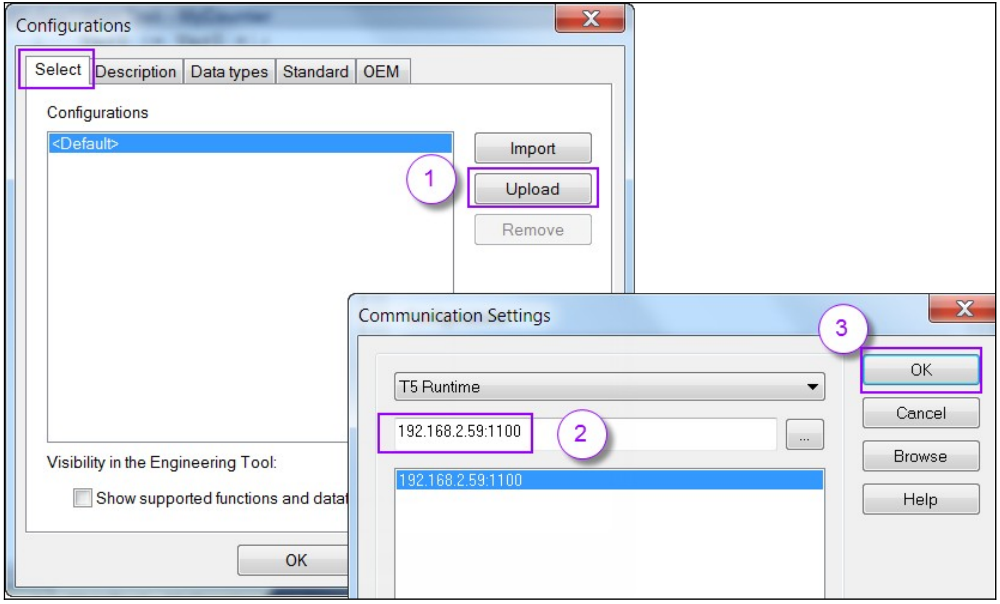
2. Double-click on third section of the status bar entry.



Hint: If no configuration is set, *'Default'* will be displayed in the status bar. If there is no open project, nothing will be displayed.

Step 3: Upload the configuration of the target runtime.

1. Click the *'Upload'* button in the *'Select'* tab.
2. In the communication window enter the IP address and port number of the target runtime
3. Click *'OK'* to start the upload process. It will take a few seconds. Once the data has been uploaded a *'Save As'* dialog pops up



T5 Runtime

192.168.2.59:1100

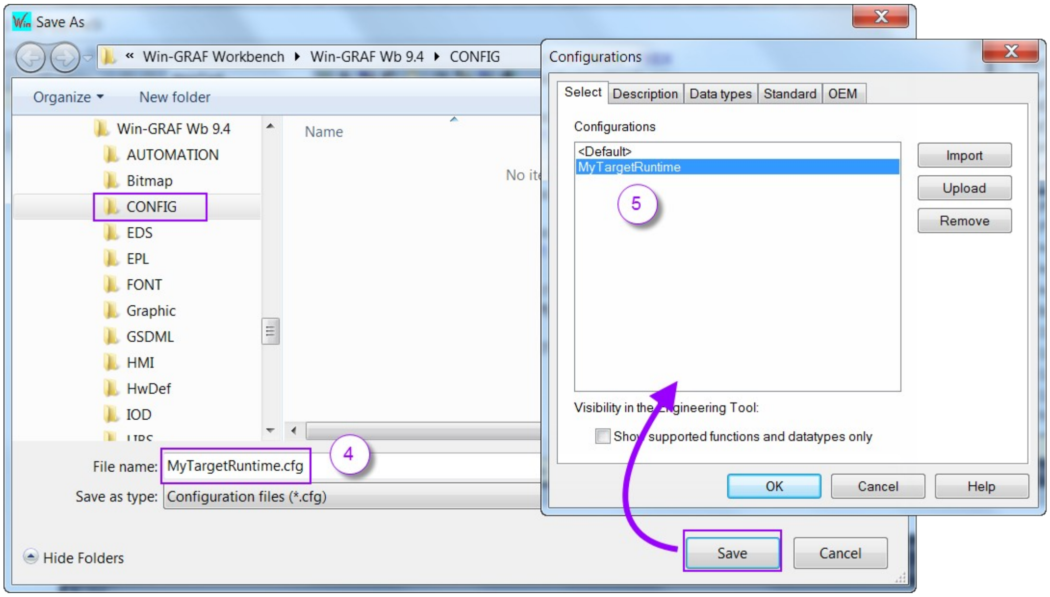
192.168.2.59:1100

4. Save the configuration as a *'.cfg'* format to the following *'CONFIG'* directory of the workbench:

'C:\Users\Public\Documents\Win-GRAF Workbench\Win-GRAF Wb xx.xx\CONFIG'

The user can assign any name for the configuration file. It is best to select a name which is related to the target system for easier identification.

5. After the file has been saved its name will be listed in the configuration list. Configuration files from many different runtime platforms can be added to the list.



File name: MyTargetRuntime.cfg

Save as type: Configuration files (*.cfg)

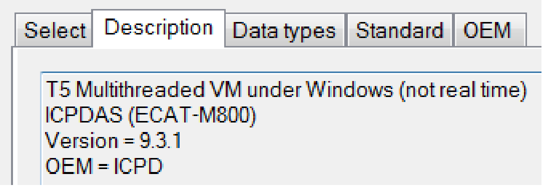
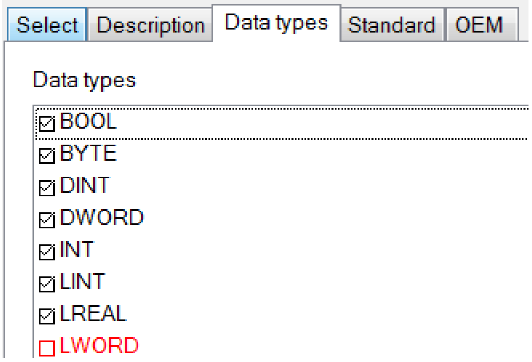
Configurations

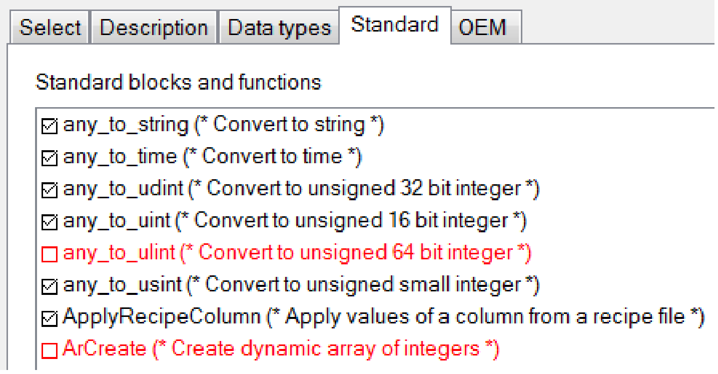
<Default>
MyTargetRuntime

Visibility in the Engineering Tool:

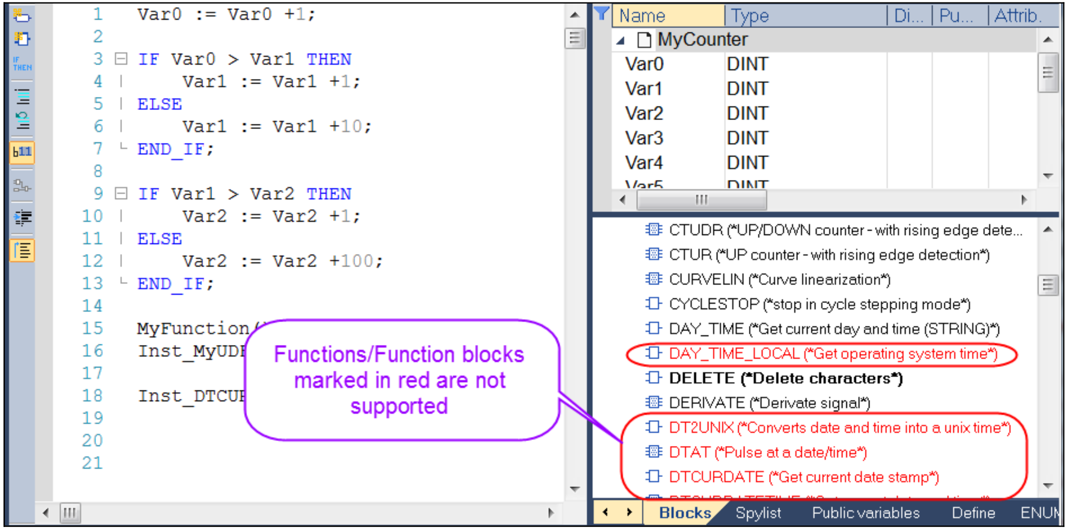
□ Show supported functions and datatypes only

Step 4: Click the different tabs to get more information of the target system. Data types which are not supported are indicated in red.

Tab Name	Description
Select	Win-GRAF preinstalled function
Description	Shows detailed information about the selected Target System Configuration according to the Select tab  T5 Multithreaded VM under Windows (not real time) ICPDAS (ECAT-M800) Version = 9.3.1 OEM = ICPD
Data type	Shows all data types available on the selected target system.  Data types ☑ BOOL ☑ BYTE ☑ DINT ☑ DWORD ☑ INT ☑ LINT ☑ LREAL □ LWORD
Standard	Shows all standard blocks and functions available on the target system.

	 Select · Description · Data types · Standard · OEM Standard blocks and functions ☑ any_to_string (* Convert to string *) ☑ any_to_time (* Convert to time *) ☑ any_to_udint (* Convert to unsigned 32 bit integer *) ☑ any_to_uint (* Convert to unsigned 16 bit integer *) □ any_to_ulint (* Convert to unsigned 64 bit integer *) ☑ any_to_usint (* Convert to unsigned small integer *) ☑ ApplyRecipeColumn (* Apply values of a column from a recipe file *) □ ArCreate (* Create dynamic array of integers *)
OEM	Shows all OEM specific blocks and functions available on the target system.

Step 5: After selecting the target configuration from the *'Configurations'* dialog and clicking *'OK'* the unsupported functions/function blocks are shown in a red color. The compiler will generate an error if it encounters one of the unsupported functions/function blocks or data type.



```
Var0 := Var0 +1;

IF Var0 > Var1 THEN
    Var1 := Var1 +1;
ELSE
    Var1 := Var1 +10;
END_IF;

IF Var1 > Var2 THEN
    Var2 := Var2 +1;
ELSE
    Var2 := Var2 +100;
END_IF;

MyFunction(
Inst_MyUDF

Inst_DTCUF
```

Functions/Function blocks marked in red are not supported

Name	Type	Di...	Pu...	Attrib.
MyCounter				
Var0	DINT			
Var1	DINT			
Var2	DINT			
Var3	DINT			
Var4	DINT			
Var5	DINT			

- CTUDR (*UP/DOWN counter - with rising edge dete...
- CTUR (*UP counter - with rising edge detection*)
- CURVELIN (*Curve linearization*)
- CYCLESTOP (*stop in cycle stepping mode*)
- DAY_TIME (*Get current day and time (STRING)*)
- DAY_TIME_LOCAL (*Get operating system time*)
- **DELETE (*Delete characters*)**
- DERIVATE (*Derivate signal*)
- DT2UNIX (*Converts date and time into a unix time*)
- DTAT (*Pulse at a date/time*)
- DTCURDATE (*Get current date stamp*)

16 Basic Operations

Below are the language features for basic data manipulation:

- Variable assignment
- Bit access

- Parenthesis
- Calling a function
- Calling a function block
- Calling a sub-program
- MOVEBLOCK: Copying/moving array items
- COUNTOF: Number of items in an array
- INC: Increase a variable
- DEC: decrease a variable

Below are the language features for controlling the execution of a program:

- Labels
- Jumps
- RETURN

Below are the structured statements for controlling the execution of a program (Table 40):

Statement	Description
IF	Conditional execution of statements.
WHILE	Repeat statements while a condition is TRUE.
REPEAT	Repeat statements until a condition is TRUE.
FOR	Execute iterations of statements.
CASE	Switch to one of various possible statements.
EXIT	Exit from a loop instruction.
WAIT	Delay program execution
ON	Conditional execution

Table 40: Program execution statements

16.1 Variable assignment

An assignment statement consists of a variable reference on the left-hand side, followed by the assignment operator ':=', followed by the expression to be evaluated. The output variable and the input expression must have the same type.

Operator	:=

The assignment statement can be used to assign

- a simple variable:

```
INT_1 := INT_2;
```

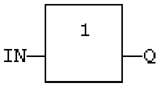
- a whole data structure (multi-element variables)
  `Struct_1 := Struct_2;`
- a return value of a function

Example:

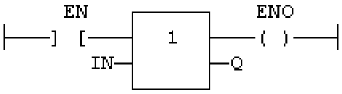
ST Language

```
//Copy IN to variable Q:
Q := IN;

//Assign the result of a complex expression
Q := (IN1 + (IN2 / IN 3)) * IN4;

//Assign a variable with the result of a
//function
result := SIN (angle);

//Assign a variable with an output parameter
// of a function block
time := MyTon.ET;
```

FBD Language In LD FBD languages, the 1 block is available to perform a *'1 gain'* data copy.



LD Language In LD the 1 block is available to perform a *'1 gain'* data copy.
In LD language, the input rung (EN) enables the assignment, and the output rung keeps the state of the input rung.
The copy is executed only if EN is TRUE.
ENO has the same value as EN.



16.2 Access to bits of an integer

You can directly specify a bit within an integer variable in expressions and diagrams, using the following notation:

```
Variable.BitNo
```

- **Variable**: is the name of an integer variable.
- **BitNo**: is the number of the bit in the integer. 0 always represents the less significant bit.

Example:

```
//Variables 'Bool_0', 'Bool_1', 'Bool_2', 'Bool_3' are
//declared as BOOL
//Variable 'USINT_1' is declared as USINT
Bool_0 := USINT_1.0;
Bool_1 := USINT_1.1;
Bool_2 := USINT_1.2;
Bool_3 := USINT_1.3;
```

The variable can have one of the following data types:
- SINT, USINT, BYTE (8 bits from .0 to .7)
- INT, UINT, WORD (16 bits from .0 to .15)
- DINT, UDINT, DWORD (32 bits from .0 to 31)
- LINT (from 0 to 63)

16.3 Parenthesis

The parenthesis operator forces the evaluation order in a complex expression.

Operator	()

Parenthesis are used in ST and IL language for changing the default evaluation order of various operations within a complex expression.
For instance, the default evaluation of *'2 * 3 + 4'* expression in ST language gives a result of 10 as *'*'* operator has highest priority. Changing the expression as *'2 * (3 + 4)'* gives a result of 14. Parenthesis can be nested in a complex expression.

Below is the default evaluation order for ST language operations (first is highest priority):

Precedence	Operator	Description
1	- NOT	Unary operators
2	* /	Multiply/Divide
3	+ -	Add/Subtract
4	< > <= >= = <>	Comparisons
5	& AND	Boolean And
6	OR	Boolean Or
7	XOR	Exclusive OR

Table 41: Operator Precedence

Example:

```
Q := (IN1 + (IN2 / IN 3)) * IN4;
```

16.4 Calling a function

A function calculates a result according to the current value of its inputs. Unlike a function block, a function has no internal data and is not linked to declared instances. A function has only one output: the result of the function. A function can be:

- A standard function (SHL, SIN...).
- A function written in *'C'* language and embedded on the target.

ST Language

To call a function block in ST, you have to enter its name, followed by the input parameters written between parenthesis and separated by comas. The function call may be inserted into any complex expression. A function call can be used as an input parameter of another function call. The following example demonstrates a call to ODD and SEL functions:

```
(*
The following statement converts any odd
integer value into the nearest even integer:
The return value of the ODD function call is
being used as an input parameter for the SEL
function call
*)
iEvenVal := SEL ( ODD( iValue ), iValue,
iValue+1 );
```

FBD Language and LD Language

To call a function block in FBD or LD languages, you just need to insert the function in the diagram and to connect its inputs and output.

IL Language

To call a function block in IL language, you must load its first input parameter before the call, and then use the function name as an instruction, followed by the other input parameters, separated by comas. The result of the function is then the current result. The following Example demonstrates a call to ODD and SEL functions:

```
(* The following statement converts any odd
integer into 0: * )

Op1: LD iValue ODD SEL iValue, 0 ST iResult
```

Table 42: Function call syntax

16.5 Calling a function block

A function block groups an algorithm and a set of private data. It has inputs and outputs. A function block can be:
- A standard function block (RS, TON...).
- A block written in *'C'* language and embedded on the target.
- A User Defined Function Block (UDFB) written in ST, FBD, LD or IL.

To use a function block, you have to declare an instance of the block as a variable, identified by a unique name. Each instance of a function block as its own set of private data and can be called separately. A call to a function block instance processes the block algorithm on the private data of the instance, using the specified input parameters.

ST Language

- To call a function block in ST, you have to specify the name of the instance, followed by the input parameters written between parenthesis and separated by comas.
- To have access to an output parameter, use the name of the instance followed by a dot '**.**' and the name of the wished parameter.
- The following example demonstrates a call to an instance of TON function block (MyTimer is declared as an instance of TON):

```
MyTimer (bTrig, t#2s);
TimerOutput := MyTimer.Q;
ElapsedTime := MyTimer.ET;
```

FBD Language and LD Language

To call a function block in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs. The name of the instance must be specified upon the rectangle of the block.

IL Language

To call a function block in IL language, you must use the **CAL** instruction, and use a declared instance of the function block. The instance name is the operand of the **CAL** instruction, followed by the input parameters written between parenthesis and separated by comas. Alternatively the **CALC**, **CALCN** or **CALNC** conditional instructions can be used:
- `CAL` Calls the function block.

- CALC Calls the function block if the current result is TRUE.
- CALNC Calls the function block if the current result is FALSE.
- CALCN same as CALNC.

The following Example demonstrates a call to an instance of TON function block
(MyTimer is declared as an instance of TON):

```
Op1: CAL MyTimer (bTrig, t#2s )
LD MyTimer.Q
ST TimerOutput
LD MyTimer.ET
ST ElapsedTimer

Op2: LD bCond
CALC MyTimer (bTrig, t#2s ) (* called only if
bCond is TRUE * )

Op3: LD bCond
CALNC MyTimer (bTrig, t#2s ) (* called only
if bCond is FALSE * )
```

Table 43: Function block call syntax

16.6 Calling a sub-program

A sub-program is called by another program. Unlike function blocks, local variables of a sub-program are not instantiated, and thus you do not need to declare instances. A call to a sub-program processes the block algorithm using the specified input parameters. Output parameters can then be accessed.

ST Language

- To call a sub-program in ST, you have to specify its name, followed by the **input parameters** written between parenthesis and separated by comas.
- To have access to an **output parameter**, use the name of the sub-program followed by a dot '**.**' and the name of the wished parameter:

```
(* calls the sub-program *)
MySubProg (i1, i2);
Res1 := MySubProg.Q1;
Res2 := MySubProg.Q2;
```

- Alternatively, if a sub-program has one and only one output parameter, it can be called as a function in ST language:

```
Res := MySubProg (i1, i2);
```

FBD Language and LD Language

To call a sub-program in FBD or LD languages, you just need to insert the block in the diagram and to connect its inputs and outputs.

IL Language

To call a sub-program in IL language, you must use the CAL instruction with the name of the sub-program, followed by the input parameters written between parenthesis and separated by comas. Alternatively the CALC, CALCN or CALNC conditional instructions can be used:

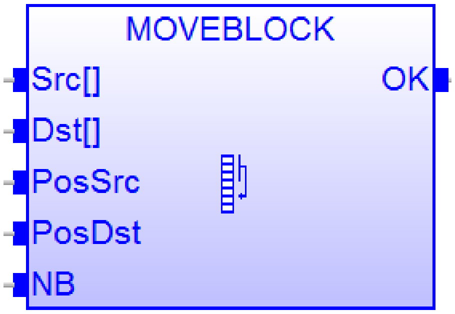
- `CAL` Calls the sub-program.
- `CALC` Calls the sub-program if the current result is TRUE.
- `CALNC` Calls the sub-program if the current result is FALSE.
- `CALCN` same as CALNC.

```
Op1: CAL MySubProg (i1, i2 )
LD MySubProg.Q1
ST Res1
LD MySubProg.Q2
ST Res2
```

Table 44: Sub-program call syntax

16.7 MOVEBLOCK - Move/Copy items of an array

The function copies a number of consecutive items starting at the index of the source array to a position in destination array. Source and destination can be the same array. In that case, the function avoids lost items when source and destination areas overlap. Arrays of string are not supported by this function.



	Para Name	Data Type	Description
Input	Src	ANY	Array containing the source of the copy. Can not be a string.
	Dst	ANY	Array containing the destination of the copy. Can not be a string
	PosSrc	DINT	Index of the first character in SRC
	PosDst	DINT	Index of the destination in DST
	NB	DINT	Number of items to be copied
Output	OK	BOOL	TRUE if successful

In FFLD language, the operation is executed only if the input rung (EN) is TRUE. The function is not available in IL Closed language.

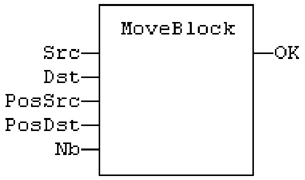
The function copies a number (NB) of consecutive items starting at the `PosSrc` index in `Src` array to `PosDst` position in `Dst` array. `Src` and `Dst` can be the same array. In that case, the function avoids lost items when source and destination areas overlap.

This function checks array bounds and is always safe. The function returns TRUE if successful. It returns FALSE if input positions and number do not fit the bounds of SRC and DST arrays.

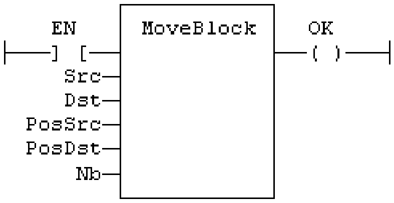
ST Language

```
OK := MOVEBLOCK (SRC, DST, PosSRS, PosDST,
NB);
```

FBD Language



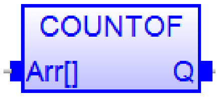
LD Language In LD language, the operation is executed only if the input rung (EN) is TRUE.



IL Language Not supported.

16.8 CountOf - Count Items in an Array

Returns the number of items in an array.



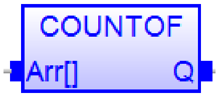
	Para Name	Data Type	Description
Input	Arr	ANY	Declared array
Output	Q	DINT	Total number of items in the array

The input must be an array and can have any data type. This function is particularly useful to avoid writing directly the actual size of an array in a program, and thus keep the program independent from the declaration.

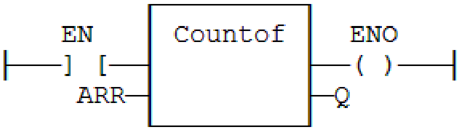
ST Language

```
FOR i := 1 TO CountOf (MyArray) DO
      MyArray[i-1] := 0;
END_FOR;
```

FBD Language



LD Language In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.



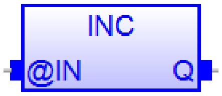
IL Language Not supported.

Example:

Array	Size
Arr1 [0..9]	10
Arr2 [0..4 , 0..9]	50

16.9 INC - Increment Numerical Variable

This function increases a numerical variable by one. For REAL data type the variable is increased by 1.0 and for TIME data type the variable is increased by 1ms. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.



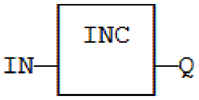
	Para Name	Data Type	Description
Input	@IN	ANY	Numerical variable (will increased after function call). The '@' character indicates that the variable is a in and output variable.
Output	Q	ANY	Increased value.

When the function is called, the variable connected to the **IN input is increased** and copied to output Q. The IN input must be directly connected to a variable, and cannot be a constant or complex expression.
This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

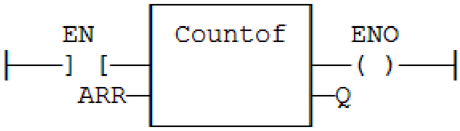
ST Language

```
IN := 1;
Q := INC (IN);
(* now: IN = 2 ; Q = 2 *)

INC (IN); (* simplified call *)
```

FBD Language



LD Language In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.



IL Language Not available.

16.10 DEC - Decrement Numerical Variable

This function decreases a numerical variable by one. For REAL data type the variable is decreased by 1.0 and for TIME data type the variable is decreased by 1ms. All data types are supported except BOOL and STRING: for these types, the output is the copy of IN.

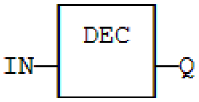
	Para Name	Data Type	Description
Input	@IN	ANY	Numerical variable (will decreased after function call). The '@' character indicates that the variable is a in- and output variable.
Output	Q	ANY	Decreased value.

When the function is called, the variable connected to the **IN input is first decreased** and then copied to output Q. The IN input must be directly connected to a variable, and cannot be a constant or complex expression.
This function is particularly designed for ST language. It allows simplified writing as assigning the result of the function is not mandatory.

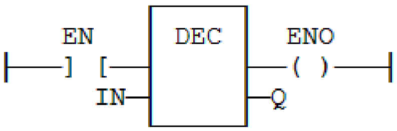
ST Language

```
IN := 2;
Q := DEC (IN);
(* now: IN = 1 ; Q = 1 *)

DEC (IN); (* simplified call *)
```

FBD Language



LD Language In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.



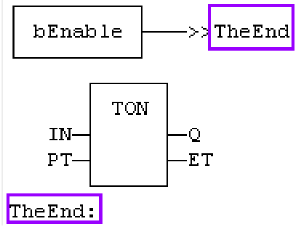
IL Language Not available.

16.11 Labels

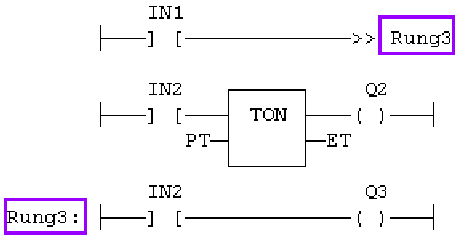
Labels are used as a destination of a jump instruction in FDB, LD or IL language. Labels and jumps cannot be used in structured ST language. A label must be represented by a unique name, followed by a colon (':'). In FBD language, labels can be inserted anywhere in the diagram, and are connected to nothing. In LD language, a label must identify a rung, and is shown on the left side of the rung.

ST Language Not available.

FBD Language In this example the TON block will not be called if `bEnable` is TRUE:



LD Language In this example the second rung will be skipped if IN1 is TRUE: rung.



IL Language

```
(* unused label - just for readability * )
Start: LD IN1

(* Jump to 'TheRest' if IN1 is TRUE * )
JMPC TheRest
```

```
LD IN2 (* these two instructions are not
executed * ) ST Q2 (* if IN1 is
TRUE * )
TheRest: LD IN3 (* label used as the jump
destination * ) ST Q3
```

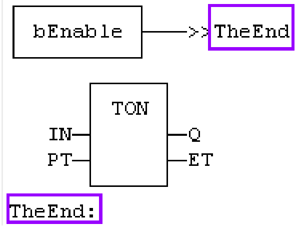
16.12 Jumps

A jump to a label branches the execution of the program after the specified label. Labels and jumps cannot be used in structured ST language. In FBD language, a jump is represented by the >> symbol followed by the label name. The input of the >> symbol must be connected to a valid boolean signal. The jump is performed only if the input is TRUE. In LD language, the >> symbol, followed by the target label name, is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE.

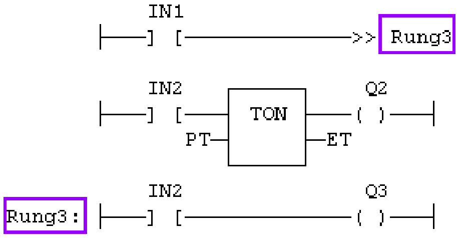
Attention
Backward jumps may lead to infinite loops that block the target cycle.

ST Language Not available.

FBD Language In this example the TON block will not be called if `bEnable` is TRUE:



LD Language In this example the second rung will be skipped if IN1 is TRUE: rung.



IL Language Below is the meaning of possible jump instructions:

- `JMP` Jump always
- `JMPC` Jump if the current result is TRUE
- `JMPNC` Jump if the current result is FALSE

```
Start: LD IN1
   JMPC TheRest (* Jump to 'TheRest' if IN1 is TRUE * )
   LD IN2 (* these three instructions are not executed * )
   ST Q2 (* if IN1 is TRUE * )
   JMP TheEnd (* unconditional jump to 'TheEnd' * )
TheRest: LD IN3
ST Q3
TheEnd:
```

16.13 RETURN - Jump to the End of the POU

The RETURN statement jumps to the end of the POU and therefore provides early exit from a function, function block or program. The program of the POU following the RETURN statements will not be executed.

The RETURN instruction can be used inside a function and function block to exit it when a condition has been met.

- When a RETURN statement is used inside a function, the function output variable has to be set before the RETURN statement is executed otherwise the output value is undefined.
- When a RETURN statement is used inside a function block, the output variables of the function blocks has to be set before the statement is executed otherwise the outputs may contain either the initialization values or the value set in the preceding function block invocation.

When used within an action block of a SFC step, the RETURN statement jumps to the end of the action block.

ST Language

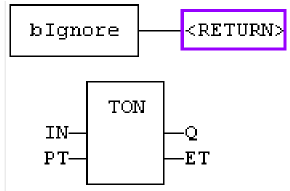
```
IF NOT bEnable THEN
   RETURN;
END_IF;
```

The rest of the program will not be executed if bEnabled is FALSE.

FBD Language

In FBD language, the return statement is represented by the '**<RETURN>**' symbol. The input of the symbol must be connected to a valid boolean signal. The jump is performed only if the input is TRUE.

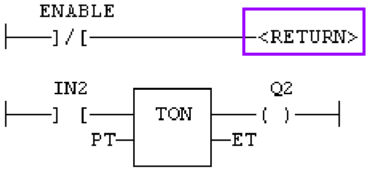
In this example the TON block will not be called if bIgnore is TRUE:



LD Language

In LD language, the '**<RETURN>**' symbol is used as a coil at the end of a rung. The jump is performed only if the rung state is TRUE.

In this example the second rung will be skipped if IN1 is TRUE: rung.



IL Language

Below is the meaning of possible instructions:

- RET Jump to the end always.
- RETC Jump to the end if the current result is TRUE.
- RETNC Jump to the end if the current result is FALSE.
- RETCN Same as RETNC.

```
Start: LD IN1
    RETC  (* Jump to the end if IN1 is TRUE * )
    LD IN2  (* these instructions are not executed * )
    ST Q2  (* if IN1 is TRUE * )
    RET  (* Jump to the end unconditionally * )
    LD IN3  (* these instructions are never executed * )
    ST Q3
```

16.14 IF - Statement

The IF statement is available in ST only. The IF statement specifies that a group of statements following the IF line is to be executed only if the associated Boolean expression evaluates to be true (TRUE). If the condition is false, then either no statement is to be executed, or the statement group following the ELSE keyword (or the ELSIF keyword if its associated Boolean condition is true) is to be executed.

Both ELSIF and ELSE are optional in a IF statement. There can be several ELSIF statements. You can use the ELSIF and ELSE keywords for multiple conditions in the same IF statement. The ELSE statement works as a default option for your IF statement. If all the IF and ELSIF boolean expressions are evaluated to FALSE, the statements after the ELSE keyword will be executed.

Syntax

```
    IF <BOOL expression> THEN
        <statements>
    ELSIF <BOOL expression> THEN
        <statements>
    ELSE
        <statements>
    END_IF;
```

ST Language

```
(* simple condition *)
IF bCond THEN
   Q1 := IN1;
   Q2 := TRUE;
END_IF;

(* binary selection *)
IF bCond THEN
   Q1 := IN1;
   Q2 := TRUE;
ELSE
   Q1 := IN2;
   Q2 := FALSE;
END_IF;

(* enumerated conditions *)
IF bCond1 THEN
   Q1 := IN1;
ELSIF bCond2 THEN
   Q1 := IN2;
ELSIF bCond3 THEN
   Q1 := IN3;
ELSE
   Q1 := IN4;
END_IF;
```

The rest of the program will not be executed if bEnabled is FALSE.

FBD Language	Not available.
LD Language	Not available.
IL Language	Not available.

16.15 WHILE - Statement

The WHILE statement causes a group of statements between the DO and the END_WHILE keyword to be executed repeatedly until the associated Boolean expression is false. If the expression is initially false, then the group of statements is not executed at all.

Syntax

```
    WHILE <BOOL expression> DO
        <statements>
    END_WHILE;
```

Attention

Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in '*While*' loop as the input may not be refreshed before the next cycle.

ST Language

```
iMax := 10;
WHILE iPos < iMax DO
   MyArray[iPos]:= 0;
   iPos +:=1;
END_WHILE;
```

The group of statement will not execute if the condition '*iPos<iMax*' is FALSE.

FBD Language	Not available.
LD Language	Not available.
IL Language	Not available.

16.16 REPEAT - Statement

Similar to the WHILE statement the REPEAT statement causes a group of statements up to the UNTIL keyword to be executed repeatedly (and at least once) until the associated Boolean condition is true.
The difference between the WHILE and REPEAT statement:

- the REPEAT statement will execute at least once independent of the Boolean condition
- the REPEAT statement will execute until the Boolean condition turns true
- the WHILE state the group of statement will only be executed if the Boolean condition is true

Syntax

```
    REPEAT
        <statements>
    UNTIL <BOOL expression> END_REPEAT;
```

<u>Attention</u>
Loop instructions may lead to infinite loops that block the target cycle. Never test the state of an input in this condition as the input will not be refreshed before the next cycle.

ST Language

```
iPos := 0;
REPEAT
   MyArray[iPos] := 0;
   iNbCleared := iNbCleared + 1;
   iPos := iPos + 1;
UNTIL iPos = iMax END_REPEAT;
```

The group of statement will execute until the condition '*iPos=iMax*' turns true.

FBD Language Not available.
LD Language Not available.
IL Language Not available.

16.17 FOR - Statement

The FOR loop is used to execute a group of statements between the DO and the END_FOR keyword with a certain number of repetitions.
The FOR statement increments the control variable `<variable>` up or down from an

initial value <start_value> to a final value <end_value> in increments determined by the step value <step> (this value defaults to 1).

Syntax

```
FOR <variable> := <start_value> TO <end_value> BY <step> DO
    <statements>
END_FOR;
```

- variable = DINT control variable.
- start_value = DINT expression: initial value for index.
- end_value = DINT expression: maximum allowed value for index.
- step = DINT expression: increasing step of index after each iteration (default is 1).

Remarks

- The BY <step> statement can be omitted. The default value for the step is 1.
- The <statements> are executed as long as the counter <variable> is not greater than the <end_value>. This is checked before executing the <statements> so that the <statements> are never executed if <start_value> is greater than <end_value>. When <statements> are executed, <index> is always increased by <step>.

ST Language

```
iArrayDim := 10;

(* resets all items of the array to 0 *)
FOR iPos := 0 TO (iArrayDim - 1) DO
   MyArray[iPos] := 0;
END_FOR;

(* set all items with odd index to 1 *)
FOR iPos := 1 TO 9 BY 2 DO
   MyArray[ipos] := 1;
END_FOR;
```

FBD Language	Not available.
LD Language	Not available.
IL Language	Not available.

16.18 CASE - Statement

The CASE statement executes a block of statements based on a selector value. With the CASE instructions one can combine several conditioned instructions with the same selector variable in one construct.

Syntax

```
CASE <DINT selector> OF
<label>:
    <statements>
<label>:
    <statements>;
<label>:
    <statements>;
<label> , <label>:
    <statements>;
<label> .. <label>:
    <statements>;
ELSE
    <statements>
END_CASE;
```

<u>Remarks</u>

- All enumerated values correspond to the evaluation of the DINT expression and are possible cases in the execution of the statements.
- The statements specified after the ELSE keyword are executed if the expression takes a value that is not enumerated in the switch.
- For each case, you must specify either
  - a value, or
  - a list of possible values separated by comas (*','*) or
  - a range of values specified by a *'min .. max'* interval. You must enter space characters before and after the *'..'* separator.

ST Language This example checks the first prime numbers:

```
CASE iNumber OF
0 :
   Alarm := TRUE;
   AlarmText := '0 gives no result';
1 .. 3, 5 :
   bPrime := TRUE;
4, 6 :
   bPrime := FALSE;
ELSE
   Alarm := TRUE;
   AlarmText := 'I don't know after 6 !';
END_CASE;
```

FBD Language Not available.
LD Language Not available.
IL Language Not available.

16.19 EXIT - Statement

The EXIT statement is used to terminate the current loop before it has completed.

- A FOR Loop is stopped before the loop variable reaches its target value.
- A WHILE Loop is stopped before the condition becomes false.
- A REPEAT Loop is stopped before the condition becomes true.

The execution continues after the END_WHILE, END_REPEAT or END_FOR keyword or the loop where the EXIT is. When the EXIT statement is located within nested loops, it only exits the loop in which the EXIT is located, and control is passed to the next statement of the outer loop. EXIT quits only one loop and cannot be used to exit at the same time several levels of nested loops.

ST Language

```
(*This program searches for the first non null
item of an array:*)
iFound = -1; (* means: not found *)
FOR iPos := 0 TO (iArrayDim - 1) DO
   IF iPos <> 0 THEN
      iFound := iPos;
      EXIT;
   END_IF;
END_FOR;
```

```
For Index:= 0 To 10 Do
    If Item[ Index ] != 0 Then
```

```
        Exit;
    End_If;
End_For;
```

FBD Language	Not available.
LD Language	Not available.
IL Language	Not available.

16.20 WAIT- Statement

The `WAIT` statement checks the attached Boolean expression and does the following:

- If the expression is TRUE, the program continues normally.
- If the expression is FALSE, then the execution of the program is suspended up to the next PLC cycle. The boolean expression will be checked again during next cycles until it becomes TRUE. The execution of other programs is not affected.

The `WAIT_TIME` statement suspends the execution of the program for the specified duration. The execution of other programs is not affected.

These instructions are available in ST language only and has no correspondence in other languages. These instructions cannot be called in a User Defined Function Block (UDFB). The use of WAIT or WAIT_TIME in a UDFB provokes a compile error.

WAIT and WAIT_TIME instructions can be called in a sub-program. However, this may lead to some unsafe situation if the same sub program is called from various programs. Re-entrance is not supported by WAIT and WAIT_TIME instructions. Avoiding this situation is the responsibility of the programmer. The compiler outputs some warning messages if a sub-program containing a WAIT or WAIT_TIME instruction is called from more than one program.

These instructions should not be called from ST parts of SFC programs. This makes no sense as SFC is already a state machine. The use of WAIT or WAIT_TME in SFC or in a sub-program called from SFC provokes a compile error.

These instructions are not available when the code is compiled through a *'C'* compiler. Using *'C'* code generation with a program containing a WAIT or WAIT_TIME instruction provokes an error during post-compiling.

These statement are extensions to the standard and are not IEC61131-3 compliant.

ST Language

```
(* use of WAIT with different kinds of BOOL
expressions *)
WAIT BoolVariable;
WAIT (diLevel > 100) AND NOT bAlarm;

WAIT SubProgCall ();


(* use of WAIT_TIME with different kinds of
TIME expressions *)
WAIT_TIME t#2s;
WAIT_TIME TimeVariable;
```

FBD Language Not available.
LD Language Not available.
IL Language Not available.

16.21 ON - Statement

Statements within the ON structure are executed only when the boolean expression rises from FALSE to TRUE. The ON instruction avoids systematic use of the R_TRIG function block or other *'last state'* flags.

The ON syntax is available in any program, sub-program or UDFB. It is available in both T5 p-code or native code compilation modes.

This statement is an extension to the standard and is not IEC61131-3 compliant.

Syntax

```
    ON <BOOL expression> DO
        <statements>
    END_DO
```

ST Language

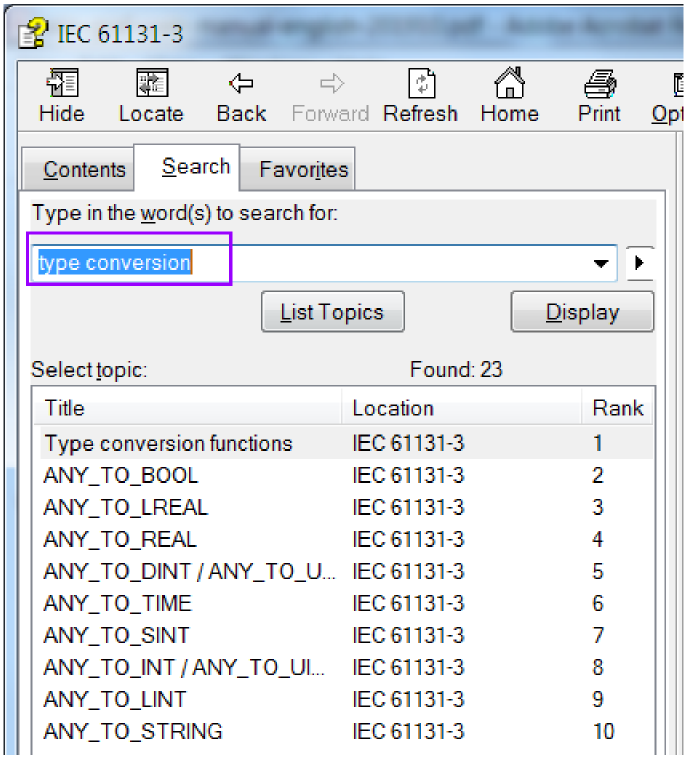
```

(* This example counts the rising edges of
variable bIN *)
```

```
ON bIN DO
   diCount := diCount + 1;
END_DO;
```

FBD Language	Not available.
LD Language	Not available.
IL Language	Not available.

17 Standard Function/Function Blocks Library

Get more information about the standard function/function blocks library by entering the function name in the search box of the workbench (Figure 175).



Type in the word(s) to search for:

type conversion

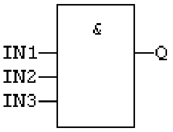
Select topic: Found: 23

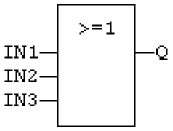
Title	Location	Rank
Type conversion functions	IEC 61131-3	1
ANY_TO_BOOL	IEC 61131-3	2
ANY_TO_LREAL	IEC 61131-3	3
ANY_TO_REAL	IEC 61131-3	4
ANY_TO_DINT / ANY_TO_U...	IEC 61131-3	5
ANY_TO_TIME	IEC 61131-3	6
ANY_TO_SINT	IEC 61131-3	7
ANY_TO_INT / ANY_TO_UI...	IEC 61131-3	8
ANY_TO_LINT	IEC 61131-3	9
ANY_TO_STRING	IEC 61131-3	10

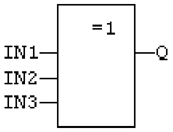
Figure 175: Workbench help search

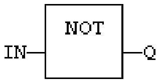
17.1 Boolean Operations

Operator	Block Diagram	Description

AND		Performs a boolean AND operation.
		Performs a logical AND of all inputs.
		
		ST Language
		
		`Q := IN1 AND IN2;`
		`Q := IN1 & IN2 & IN3;`

OR		Performs a logical OR of all inputs.
		
		ST Language
		`Q := IN1 OR IN2;`
		`Q := IN1 OR IN2 OR IN3;`

XOR		Performs an exclusive OR of all inputs.
		
		ST Language
		`Q := IN1 XOR IN2;`
		`Q := IN1 XOR IN2 XOR IN3;`

NOT		Performs a boolean negation of the input.
		
		ST Language
		`Q := NOT IN;`
		`Q := NOT (IN1 OR IN2);`

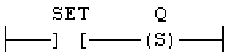
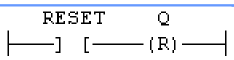
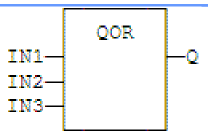
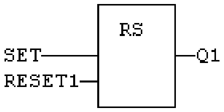
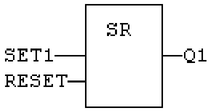
S		Force a boolean output to TRUE Only supported by LD Language.
R		Force a boolean output to FALSE. Only supported by LD Language.
QOR		Count the number of TRUE inputs.

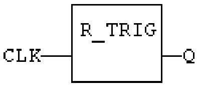
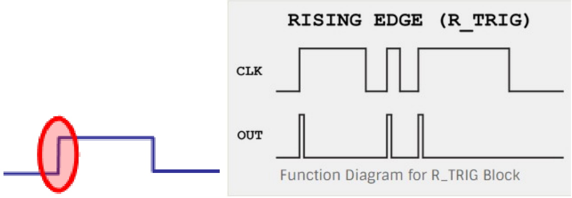
Table 45: Operators for managing booleans

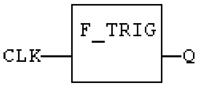
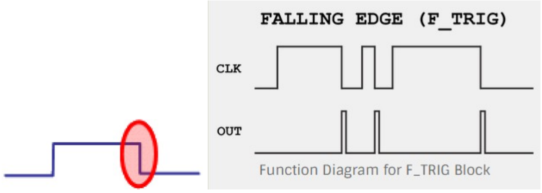
Function	Block Diagram	Description

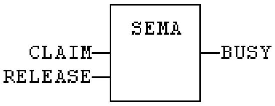
RS		Reset dominant bistable
		
		The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to FALSE (reset dominant).
		
		ST Language
		MyRS is declared as an instance of RS function block:
		`MyRS (SET, RESET1);`
		`Q1 := MyRS.Q1;`

Function	Block Diagram	Description

SR		Set dominant bistable.
		The output is unchanged when both inputs are FALSE. When both inputs are TRUE, the output is forced to TRUE (set dominant).
		
		ST Language
		MySR is declared as an instance of SR function block:
		`MySR (SET1, RESET);`
		`Q1 := MySR.Q1;`

R_TRIG		Rising pulse detection.
		 RISING EDGE (R_TRIG)
		CLK
		OUT
		Function Diagram for R_TRIG Block
		
		ST Language
		MyTrigger is declared as an instance of R_TRIG function block:
		`MyTrigger (CLK);`
		`Q := MyTrigger.Q;`

F_TRIG		Falling pulse detection.
		 FALLING EDGE (F_TRIG)
		CLK
		OUT
		Function Diagram for F_TRIG Block
		
		ST Language
		MyTrigger is declared as an instance of F_TRIG function block:
		`MyTrigger (CLK);`
		`Q := MyTrigger.Q;`

SEMA		Semaphore
		
		ST Language
		MySema is a declared instance of SEMA function block:
		`MySema (CLAIM, RELEASE);`
		`BUSY := MySema.BUSY;`

Function	Block Diagram	Description

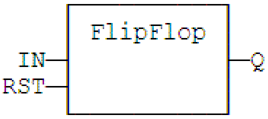
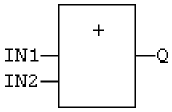
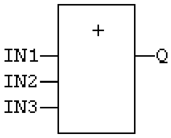
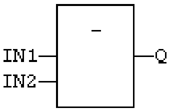
FLIPFLOP		Flipflop^bistable
		- The output is systematically reset to FALSE if RST is TRUE.
		- The output changes on each rising edge of the IN input, if RST is FALSE
		
		<u>ST Language</u>
		MyFlipFlop is declared as an instance of FLIPFLOP function block:
		`MyFlipFlop (IN, RST);`
		`Q := MyFlipFlop.Q;`

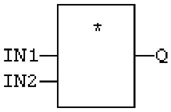
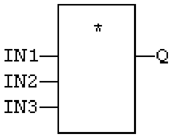
Table 46: Blocks for managing boolean signals

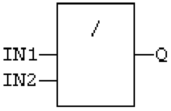
17.2 Arithmetic operations

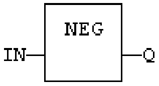
Operator	Block Diagram	Description

+	
	
		Performs an addition of all inputs.
		- All inputs and the output must have the same type.
		- In FBD language, the block may have up to 16 inputs. (Set number of inputs: Double click the function block and enter the number of inputs in the scroll box (see description below))
		- In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung.
		The addition can be used with strings. The result is the concatenation of the input strings.
		
		ST Language:
		`Q := IN1 + IN2;`
		
		`(* MyString is equal to 'Hello' *)`
		`MyString := 'He' + 'll ' + 'o';`

-		Performs a subtraction of inputs - All inputs and the output must have the same type. - In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung. ST Language: `Q := IN1 - IN2;`

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>*</td><td>
</td><td>Performs a multiplication of all inputs.
- All inputs and the output must have the same type.
- In FBD language, the block may have up to 16 inputs. (Set number of inputs: Double click the function block and enter the number of inputs in the scroll box (see description below))
- In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung.

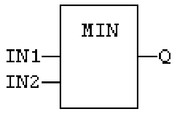
ST Language:
<code>Q := IN1 * IN2;</code></td></tr>
<tr><td>/</td><td></td><td>Performs a division of inputs.
- All inputs and the output must have the same type.
- In LD language, the input rung (EN) enables the operation, and the output rung keeps the same value as the input rung.

ST Language:
<code>Q := IN1 / IN2;</code></td></tr>
<tr><td>-</td><td></td><td>Integer negation (unary operator)
- Performs an integer negation of the input.
- In FBD and LD language, the block NEG can be used.
- In LD language, the operation is executed only if the input rung (EN) is TRUE. The output rung (ENO) keeps the same value as the input rung.
- In ST language, '-' can be followed by a complex boolean expression between parenthesis.

Truth table (examples):
<table><tr><th>IN</th><th>Q</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>-1</td></tr><tr><td>-123</td><td>123</td></tr></table>
ST Language:
<code>Q := -IN;</code>
<code>Q := - (IN1 + IN2);</code></td></tr>
</table>

Table 47: Standard arithmetic operators

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>MIN</td><td></td><td>Get the minimum of two values.
<table><tr><th>In/ Output</th><th>Data type</th></tr></table></td></tr>
</table>

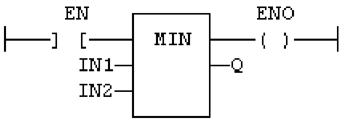
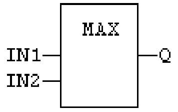
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td></td><td>
</td><td>
<table>
<tr><td>IN1</td><td>ANY</td></tr>
<tr><td>IN2</td><td>ANY</td></tr>
<tr><td>Q</td><td>ANY</td></tr>
</table>

ST Language:

```
Q := MIN (IN1, IN2);
```

LD Language:

- In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung.
- The comparison is executed only if EN is TRUE.
- ENO has the same value as EN.


</td></tr>
<tr><td>MAX</td><td>
</td><td>Get the maximum of two values

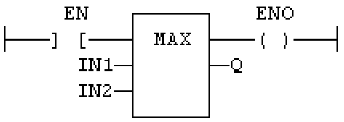
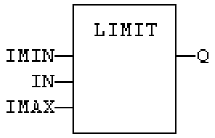
<table>
<tr><th>In/
Output</th><th>Data type</th></tr>
<tr><td>IN1</td><td>ANY</td></tr>
<tr><td>IN2</td><td>ANY</td></tr>
<tr><td>Q</td><td>ANY</td></tr>
</table>

ST Language:

```
Q := MAX (IN1, IN2);
```

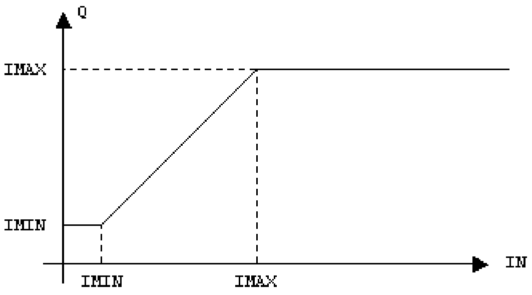
LD Language:

- In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung.
- The comparison is executed only if EN is TRUE.
- ENO has the same value as EN.


</td></tr>
<tr><td>LIMIT</td><td>
</td><td>Bounds an integer to low and high limits

- IMIN if IN < IMIN;
- IMAX if IN > IMAX;
- IN otherwise

<table>
<tr><th>In/
Output</th><th>Data type</th></tr>
<tr><td>IMIN</td><td>DINT</td></tr>
<tr><td>IN</td><td>DINT</td></tr>
<tr><td>IMAX</td><td>DINT</td></tr>
</table>
</td></tr>
</table>

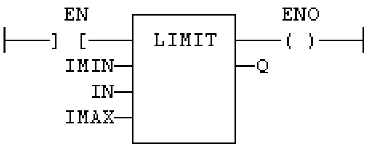
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td></td><td></td><td>
<table><tr><td>Q</td><td>ADINT</td></tr></table>


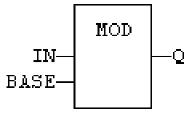
ST Language:

```
Q := LIMIT (IMIN, IN, IMAX);
```

LD Language:

- In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung.
- The comparison is executed only if EN is TRUE.
- ENO has the same value as EN.



</td></tr>
<tr><td>MOD</td><td>


</td><td>
Calculation of modulo.

- The result of the function is -1 if the argument BASE is less than or equal to 0.

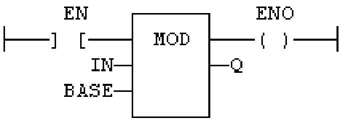
<table>
<tr><th>In/ Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>DINT/REAL/ LREAL</td><td>Input value.</td></tr>
<tr><td>BASE</td><td>DINT/REAL/ LREAL</td><td>Base of the modulo.</td></tr>
<tr><td>Q</td><td>DINT/REAL/ LREAL</td><td>Modulo: rest of the integer division (IN / BASE).</td></tr>
</table>

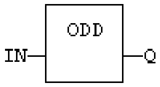
ST Language:

```
Q := MOD (IN, BASE);
```

LD Language:

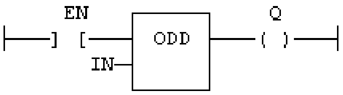
- In LD language, the input rung (EN) enables the operation, and the output rung keeps the state of the input rung.
</td></tr>
</table>

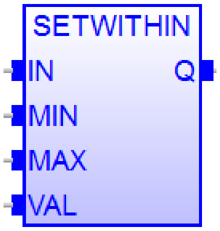
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td></td><td></td><td>
- The comparison is executed only if EN is TRUE.
- ENO has the same value as EN.


</td></tr>
<tr><td>ODD</td><td>


</td><td>
Test if an integer is odd
<table>
<tr><th>In/
Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>DINT</td><td>Input value.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN is odd.
FALSE if IN is even.</td></tr>
</table>
ST Language:

```
Q := ODD (IN);
```

LD Language:
- In LD language, the input rung (EN) enables the operation, and the output rung is the result of the function.
- The function is executed only if EN is TRUE


</td></tr>
<tr><td>SetWithin</td><td>


</td><td>
Force a value when inside an interval
- The output is forced to VAL when the IN value is within the [MIN .. MAX] interval.
- It is set to IN when outside the interval.
<table>
<tr><th>In/
Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN</td><td>ANY</td><td>Input value.</td></tr>
<tr><td>MIN</td><td>ANY</td><td>Low limit of the interval.</td></tr>
<tr><td>MAX</td><td>ANY</td><td>High limit of the interval.</td></tr>
<tr><td>VAL</td><td>ANY</td><td>Value to apply when inside the interval.</td></tr>
<tr><td>Q</td><td>ANY</td><td>Result.</td></tr>
</table>
Truth table:
<table>
<tr><th>IN</th><th>Q</th></tr>
<tr><td>IN &lt; MIN</td><td>IN</td></tr>
<tr><td>IN &gt; MAX</td><td>IN</td></tr>
<tr><td>MIN &lt; IN &lt; MAX</td><td>VAL</td></tr>
</table>
</td></tr>
</table>

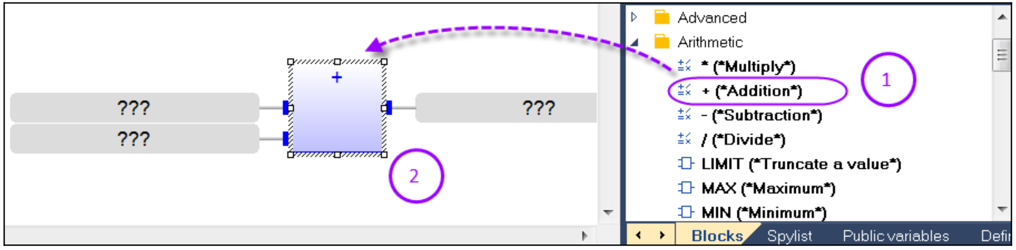
Operator	Block Diagram	Description

Table 48: Standard functions for performing arithmetic operations

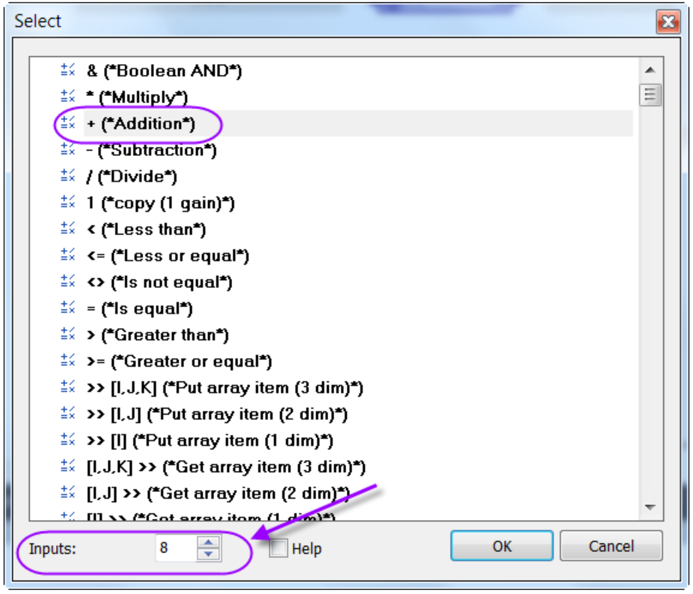
17.2.1 Set Number of Input Parameters

For certain functions in the library the workbench allows user to set number of function input parameter of the block diagram:

Step 1: Drag and drop the block diagram from the *'Block'* library to the program editor

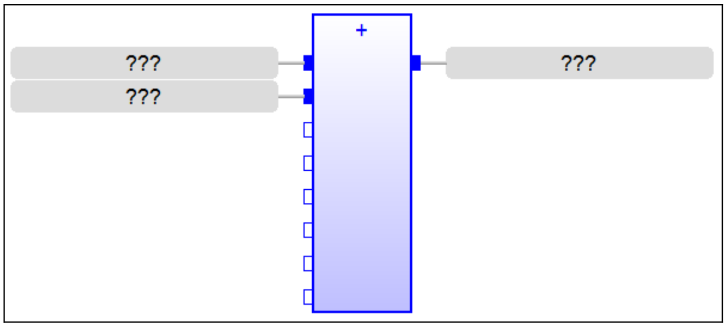


Step 2: Double click the block diagram and set the number of inputs (example: 8 inputs). Confirm the setting by clicking OK.



Inputs: 8

Step 3: The *'+'* block diagram shows 8 inputs now. Assign variable to the new inputs



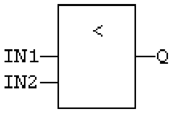
17.3 Comparison Operations

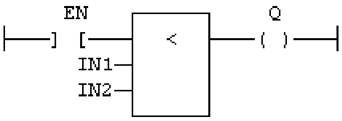
Below are the standard operators and blocks that perform comparisons:

Operator	Meaning
<	less than
>	greater than
<=	less or equal
>=	greater or equal
=	is equal
<>	is not equal
CMP	detailed comparison

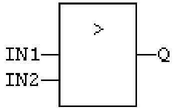
Table 49: Standard comparison operators

17.3.1 Less Than (< LT)

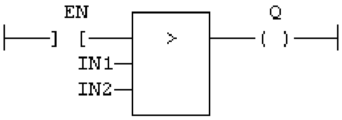
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td><</td><td>
</td><td>Test if first input is <b>less than</b> second input.
<table>
<tr><th>In/ Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1 < IN2.</td></tr>
</table>
<ul>
<li>Both inputs must have the same type.</li>
<li>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, <i>'ABC'</i> is less than <i>'ZX'</i> ; <i>'ABCD'</i> is greater than <i>'ABC'</i>.</li>
</ul>
ST Language:</td></tr>
</table>

Operator	Block Diagram	Description
		`Q := IN1 < IN2;` <u>LD Language:</u> - In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison. - The comparison is executed only if EN is TRUE: 

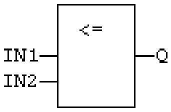
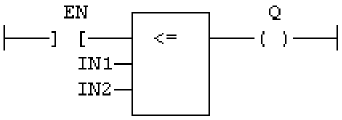
17.3.2 Greater Than (> GT)

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr>
<td>></td>
<td> </td>
<td>Test if first input is <b>greater than</b> second input.
<table>
<tr><th>In/
Output</th><th>Data
type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1 > IN2.</td></tr>
</table>
- Both inputs must have the same type.
- Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, <i>'ABC'</i> is less than <i>'ZX'</i> ; <i>'ABCD'</i> is greater than <i>'ABC'</i>.

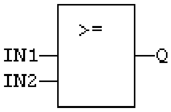
ST Language:
<code>Q := IN1 > IN2;</code>

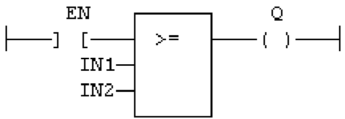
<u>LD Language:</u>
- In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison.
- The comparison is executed only if EN is TRUE:
 </td>
</tr>
</table>

17.3.3 Less Than or Equal (<= LE)

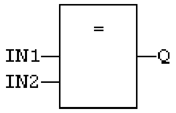
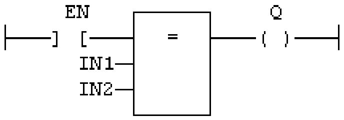
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>&lt;=</td><td>
</td><td>Test if first input is <b>less than</b> or <b>equal</b> to second input.
<table>
<tr><th>In/
Output</th><th>Data
type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1<= IN2.</td></tr>
</table>
<ul>
<li>Both inputs must have the same type.</li>
<li>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, <i>'ABC'</i> is less than <i>'ZX'</i> ; <i>'ABCD'</i> is greater than <i>'ABC'</i>.</li>
</ul>
<u>ST Language:</u>
<pre>Q := IN1 <= IN2;</pre>
<u>LD Language:</u>
<ul>
<li>In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison.</li>
<li>The comparison is executed only if EN is TRUE:</li>
</ul>

</td></tr>
</table>

17.3.4 Greater Than or Equal (<= LE)

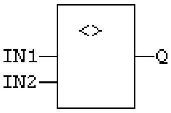
<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>&gt;=</td><td>
</td><td>Test if first input is <b>greater than</b> or <b>equal</b> to second input.
<table>
<tr><th>In/
Output</th><th>Data
type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1<= IN2.</td></tr>
</table>
<ul>
<li>Both inputs must have the same type.</li>
<li>Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, <i>'ABC'</i> is less than <i>'ZX'</i> ; <i>'ABCD'</i> is greater than <i>'ABC'</i>.</li>
</ul>
ST Language:
<pre>Q := IN1 <= IN2;</pre></td></tr>
</table>

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td></td><td></td><td>LD Language:
- In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison.
- The comparison is executed only if EN is TRUE:

 </td></tr>
</table>

17.3.5 Equal (= EQ)

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>=</td><td>
 </td><td>Test if first input is <b>equal to</b> second input.
<table>
<tr><th>In/ Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1= IN2.</td></tr>
</table>
- Both inputs must have the same type.
- Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, <i>'ABC'</i> is less than <i>'ZX'</i> ; <i>'ABCD'</i> is greater than <i>'ABC'</i>.
ST Language:
<code>Q := IN1 = IN2;</code>
LD Language:
- In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison.
- The comparison is executed only if EN is TRUE:

 </td></tr>
</table>

17.3.6 Not Equal (<> NE)

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>&lt;&gt;</td><td>
</td><td>Test if first input is <b>not equal to</b> second input.
<table>
<tr><th>In/ Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>ANY</td><td>First input.</td></tr>
<tr><td>IN2</td><td>ANY</td><td>Second input.</td></tr>
<tr><td>Q</td><td>BOOL</td><td>TRUE if IN1<&gt; IN2.</td></tr>
</table>

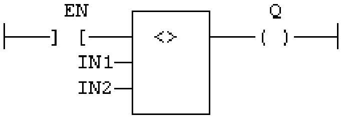
- Both inputs must have the same type.
- Comparisons can be used with strings. In that case, the lexical order is used for comparing the input strings. For instance, *'ABC'* is less than *'ZX'* ; *'ABCD'* is greater than *'ABC'*.

ST Language:

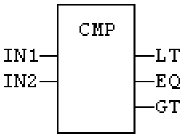
```
Q := IN1 <> IN2;
```

LD Language:

- In LD language, the input rung (EN) enables the operation, and the output rung is the result of the comparison.
- The comparison is executed only if EN is TRUE:


</td></tr>
</table>

17.3.7 Detailed Comparison

<table>
<tr><th>Operator</th><th>Block Diagram</th><th>Description</th></tr>
<tr><td>CMP</td><td>
</td><td>Function Block - Comparison with detailed outputs for integer inputs.
<table>
<tr><th>In/ Output</th><th>Data type</th><th>Description</th></tr>
<tr><td>IN1</td><td>DINT</td><td>First input.</td></tr>
<tr><td>IN2</td><td>DINT</td><td>Second input.</td></tr>
<tr><td>LT</td><td>BOOL</td><td>TRUE if IN1 &lt; IN2.</td></tr>
<tr><td>EQ</td><td>BOOL</td><td>TRUE if IN1= IN2.</td></tr>
<tr><td>GT</td><td>BOOL</td><td>TRUE if IN1&gt; IN2.</td></tr>
</table>

ST Language:

```
MyCMP (IN1, IN2);
```
</td></tr>
</table>

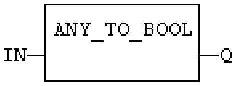
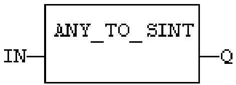
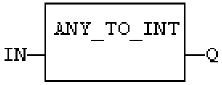
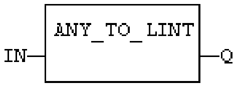
Operator	Block Diagram	Description

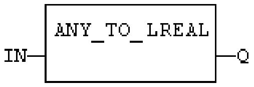
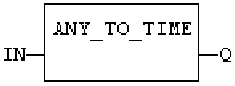
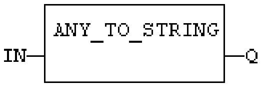
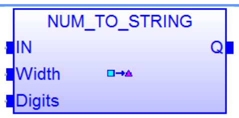
		`bLT := MyCmp.LT;`
		`bEQ := MyCmp.EQ;`
		`bGT := MyCmp.GT;`

17.4 Data Type Conversion Functions

For arithmetic, mathematic and comparison operations all the operands need to be of the same data type. Use the typecasting functions (Table 50) to convert a data type.

In Win-GRAF all data type conversion has to be done explicitly as it does not support implicit conversion of data. That means the compiler will not automatically convert a *'smaller'* data type to a *'larger'* data type (e.g. from INT to DINT; or from BYTE to WORD) and it will generate an error if it encounters an expressions or assignments with a mismatch of data types.

Function	Block Diagram	Description
ANY_TO_BOOL		Converts the input variable into boolean value.
ANY_TO_SINT		Converts the input into a short integer (8 bit) value.
ANY_TO_INT		Converts the input into a 16 bit integer value.
ANY_TO_DINT	ANY_TO_DINT IN Q	Convert to Long Integer (32-bit – Default)
ANY_TO_LINT		Convert to Long Integer (64-bit)
ANY_TO_REAL	ANY_TO_REAL IN Q	Convert to real value (floating point) - For BOOL input data types, the output is 0.0 or 1.0. - For DINT input data type, the output is the same number. - For TIME input data types, the result is the number of milliseconds. - For STRING inputs, the output is the number represented by the string, or 0.0

Function	Block Diagram	Description
		if the string does not represent a valid number.
ANY_TO_LREAL		Converts the input into double precision real value. - For BOOL input data types, the output is 0.0 or 1.0. - For DINT input data type, the output is the same number. - For TIME input data types, the result is the number of milliseconds. - For STRING inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number.
ANY_TO_TIME		Convert the input to Timer value - For BOOL input data types, the output is t#0ms or t#1ms. - For DINT or REAL input data type, the output is the time represented by the input number as a number of milliseconds. - For STRING inputs, the output is the time represented by the string, or t#0ms if the string does not represent a valid time.
ANY_TO_STRING		converts to character string - For BOOL input data types, the output is 1 or 0 for TRUE and FALSE respectively. - For DINT, REAL or TIME input data types, the output is the string representation of the input number. - This is a number of milliseconds for TIME inputs.
NUM_TO_STRING		Convert Number to String. - Can set the decimal digital number after converting - This function converts any numerical value to a string. Unlike the ANY_TO_STRING function, it allows you to specify a wished length and a number of digits after the decimal points. - If WIDTH is 0, the string is formatted with the necessary length. - If WIDTH is greater than 0, the string is completed with heading blank characters in order to match the value of WIDTH. - If WIDTH is lower than 0, the string is

Function	Block Diagram	Description

		completed with trailing blank characters in order to match the absolute value of WIDTH.
		- If DIGITS is lower or equal to 0, then neither decimal part nor point are added.
		- If DIGITS is greater than 0, the corresponding number of decimal digits are added. *'0'* digits are added if necessary
		- If the value is too long for the specified width, then the string is filled with '*' characters.
		
		Examples
		`Q := NUM_TO_STRING(123.4, 8, 2);`
		`(* Q is ' 123.40' *)`
		`Q := NUM_TO_STRING(123.4, -8, 2);`
		`(* Q is '123.40 ' *)`
		`Q :=NUM_TO_STRING(1.333333, 0,2);`
		`(* Q is '1.33' *)`
		`Q := NUM_TO_STRING(1234, 3, 0);`
		`(* Q is '***' *)`

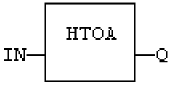
ATOH		Convert hexadecimal string to integer. Converts integer to string using hexadecimal basis.
HTOA		Convert integer to hexadecimal string Converts string to integer using hexadecimal basis.

Table 50: Typecasting function

Example:
In the following floating point calculation the DINT variable is first explicit converted to a REAL type before the calculation is being done:

```
  REAL_Val_1 := ANY_TO_REAL (DINT_Val_1) * 3.5 + 4.8 ;
```

17.5 Bit Operation

The tables below list the standard functions for executing bit operations on 8 bit to 32 bit variables.

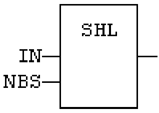
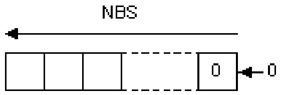
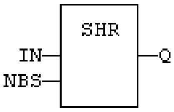
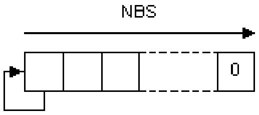
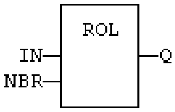
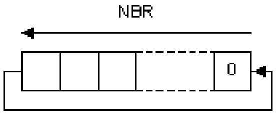
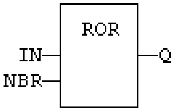
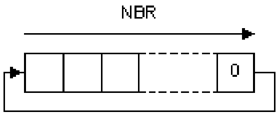
Function	Block Diagram	Description	Diagram
SHL		Shift bits of a operand to the left.	
SHR		Shift bits of a operand to the right.	
ROL		Rotate bits of a operand to the left.	
ROR		Rotate bits of a operand to the right.	

Table 51: Bitshift operators

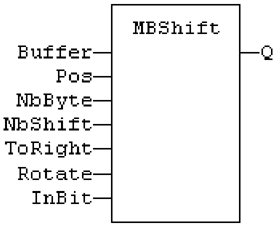
Function	Block Diagram	Description
MBSHIFT	 MBShift Buffer Pos NbByte NbShift ToRight Rotate InBit Q	Multibyte shift / rotate

Table 52: Byte shift operator

Bitmask operators are used for bitwise operations, particularly in a bit field. Using a mask, multiple bits in a byte, word, integer etc. can be set either on, off or inverted from on to off (or vice versa) (Table 53).

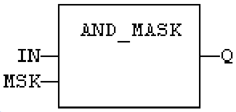
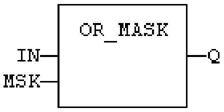
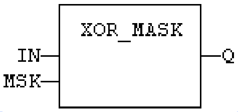
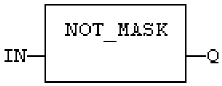
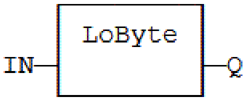
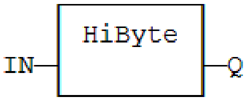
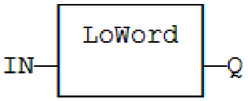
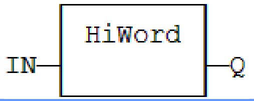
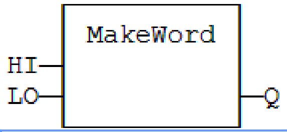
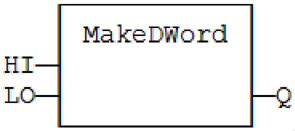
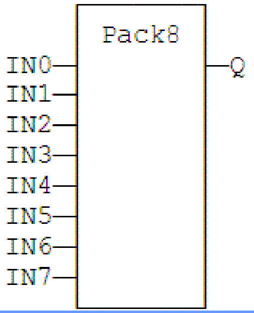
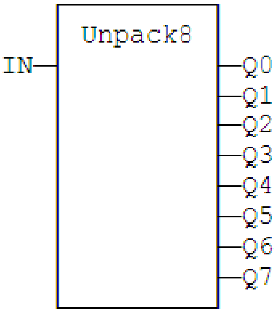
Function	Block Diagram	Description
AND_MASK		Performs a bit to bit AND between two integer values
OR_MASK		Performs a bit to bit OR between two integer values.
XOR_MASK		Performs a bit to bit exclusive OR between two integer values.
NOT_MASK		Performs a bit to bit negation of an integer value.

Table 53: Bitmask operator

Function	Block Diagram	Description
LOBYTE		Get the lowest byte of a word. Get the less significant byte of a word.
HIBYTE		Get the highest byte of a word. Get the most significant byte of a word
LOWORD		Get the lowest word of a double word. Get the less significant word of a double word.
HIWORD		Get the highest word of a double word. Get the most significant word of a double word.
MAKEWORD		Pack bytes to a word. Builds a word as the concatenation of two bytes.
MAKEDWORD		Pack words to a double word. Builds a double word as the concatenation of two words.
PACK8	 Pack8 IN0 IN1 IN2 IN3 IN4 IN5 IN6 IN7 Q	Pack bits in a byte. Builds a byte with bits.

UNPACK8	 Unpack8
	IN
	Q0
	Q1
	Q2
	Q3
	Q4
	Q5
	Q6
	Q7
		Extract bits of a byte.
		Structure Text example:
		`MyUnpack (IN);`
		`Q0 := MyUnpack.Q0;`
		`Q1 := MyUnpack.Q1;`
		`Q2 := MyUnpack.Q2;`
		`Q3 := MyUnpack.Q3;`
		`Q4 := MyUnpack.Q4;`
		`Q5 := MyUnpack.Q5;`
		`Q6 := MyUnpack.Q6;`
		`Q7 := MyUnpack.Q7;`

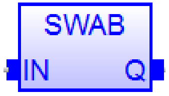
SWAB		Swap the bytes of a integer Supported data types are INT, UINT, WORD, DINT, UDINT and DWORD.

Table 54: Pack/unpack 8, 16 and 32 bit registers

A single bit in a 8 bit to 32 bit integer can be directly turn on or off. It is also possible to directly check whether a certain bit in a particular bit field has been set (Table 55).

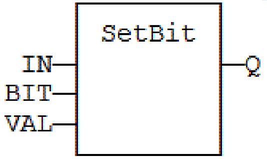
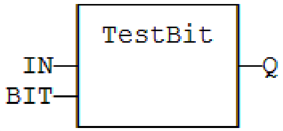
Function	Block Diagram	Description
SETBIT		Set a bit in an integer register.
TESTBIT		Test a bit of an integer register. Indicates whether a bit at a certain position in an integer value is set or not.

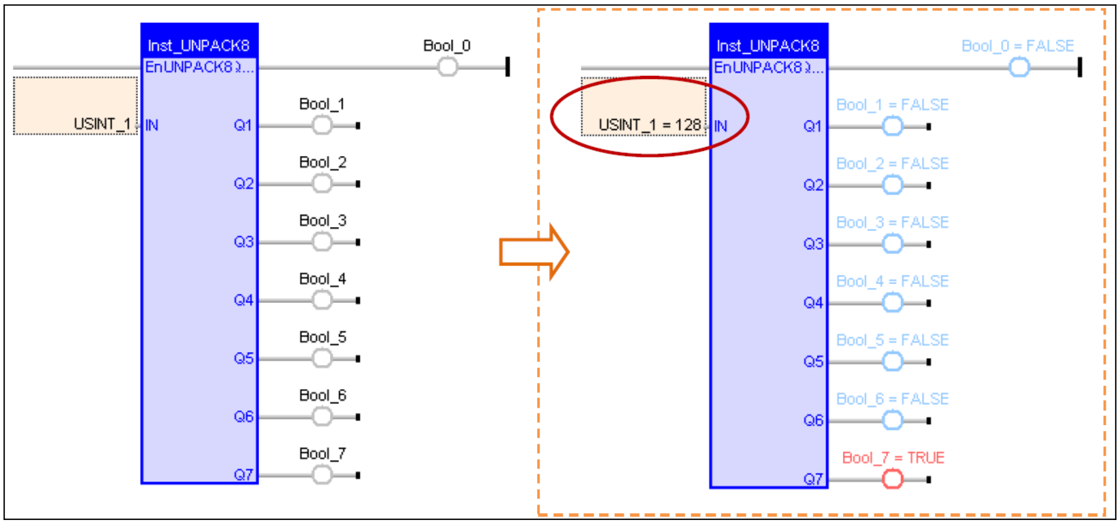
Table 55: Bit access in 8 bit to 32 bit integers

17.5.1 Examples

17.5.1.1 *Extract Bits from a Byte*

This section shows how to extract single bits from a BYTE, USINT and INT data type by using the UNPACK8 function:

- Unpack one BYTE (or USINT, range: 0 to 255) to 8 Booleans by using the *'UNPACK8'* function.

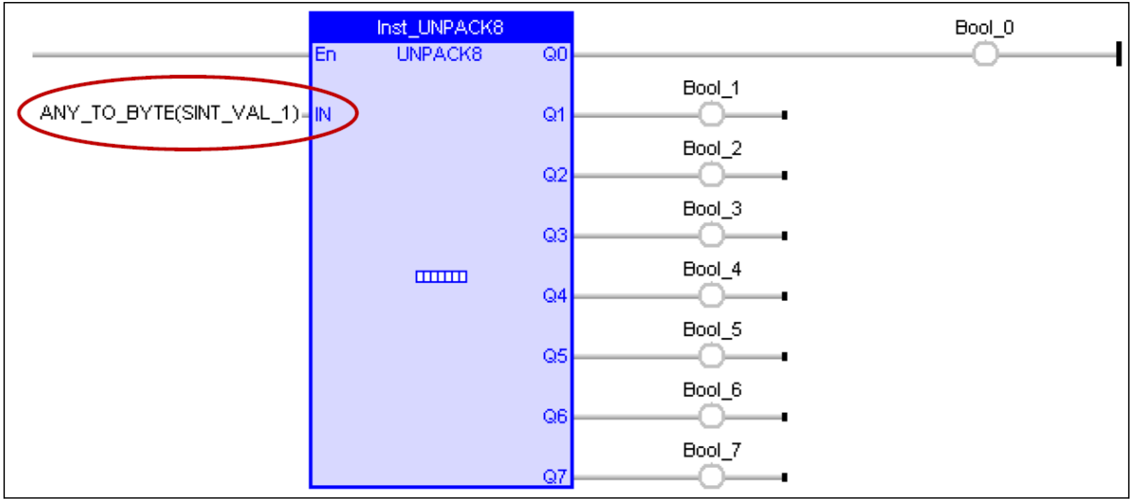


In Structure Text each bit of a integer data type can be individually addressed by adding *'.'* and then the bit number:

```
Bool_0 := USINT_1.0;
Bool_1 := USINT_1.1;
```

```
Bool_2 := USINT_1.2;
Bool_3 := USINT_1.3;
Bool_4 := USINT_1.4;
Bool_5 := USINT_1.5;
Bool_6 := USINT_1.6;
Bool_7 := USINT_1.7;
```

- To unpack one SINT to 8 Booleans, the data type first has to be converted into a BYTE type by calling the *'ANY_TO_BYTE ()'* function:

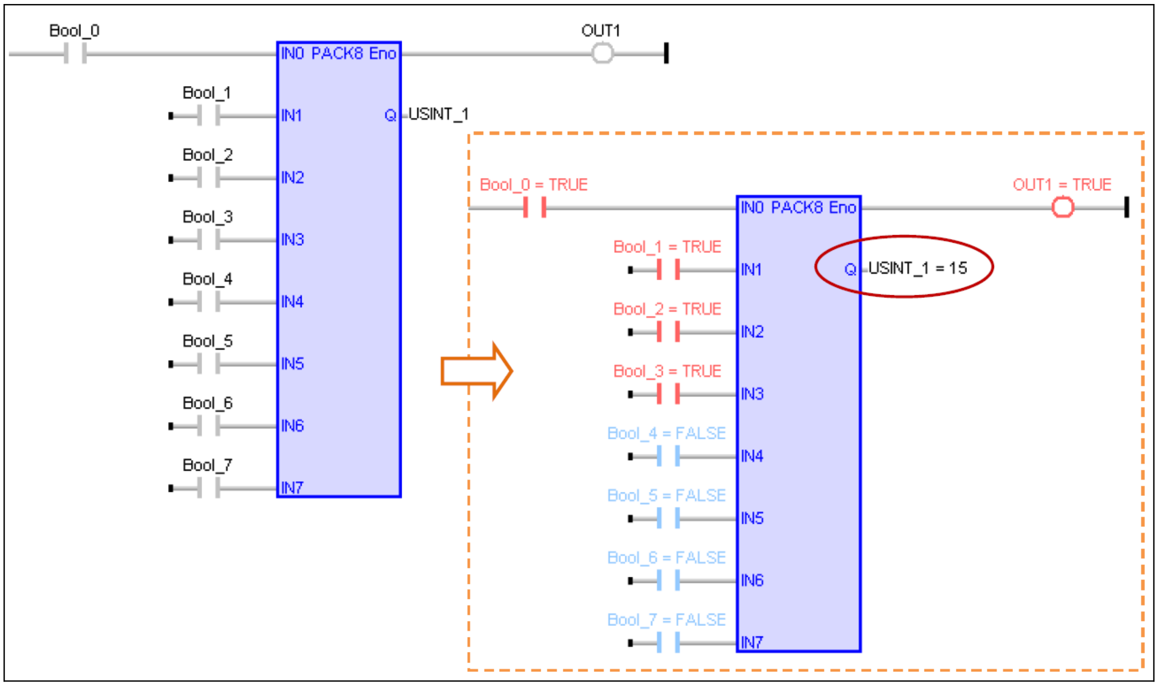


17.5.1.2 Pack Bits in a Byte

The *'PACK8'* function extract single bits from a BYTE, USINT and INT data type:

- Use the *'PACK8'* function to pack 8 Booleans into one BYTE (or USINT, range: 0 to 255):

Ladder:



Structure Text:

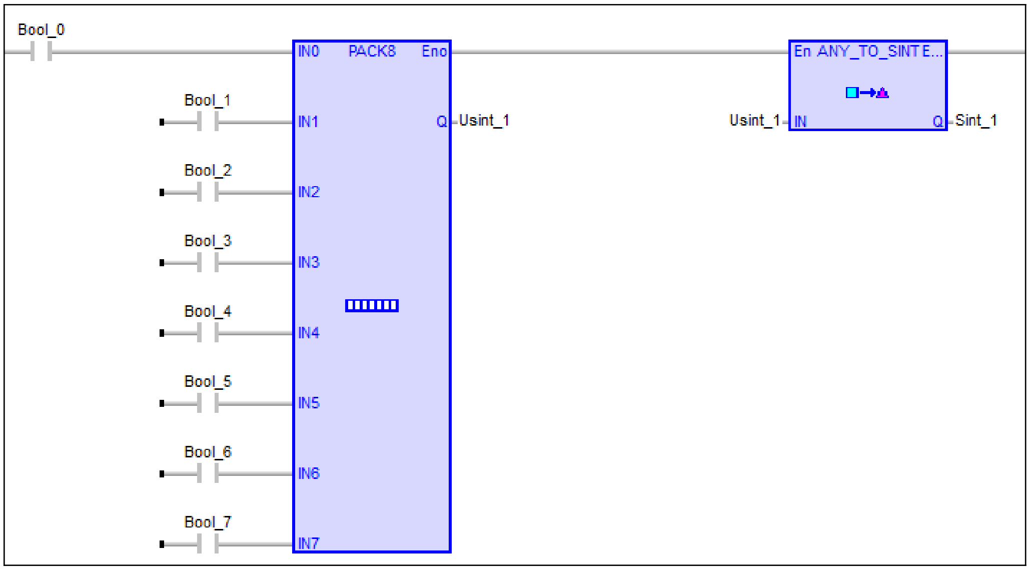
Method 1:

```
USINT_1 := PACK8 (Bool_0, I Bool_1, Bool_2, Bool_3,
Bool_4, Bool_5, Bool_6, Bool_7);
```

Method 2:

```
USINT_1.0 := Bool_0;
USINT_1.1 := Bool_1;
USINT_1.2 := Bool_2;
USINT_1.3 := Bool_3;
USINT_1.4 := Bool_4;
USINT_1.5 := Bool_5;
USINT_1.6 := Bool_6;
USINT_1.7 := Bool_7;
```

- Pack 8 Booleans into one SINT type: The *'PACK8'* function can only pack Booleans into a USINT type, it is therefore necessary to convert the *'PACK8'* output into a SINT type by calling the *'ANY_TO_SINT'* function:


Bool_0
IN0
PACK8
Eno
Bool_1
IN1
Q
Usint_1
Bool_2
IN2
Bool_3
IN3
Bool_4
IN4
Bool_5
IN5
Bool_6
IN6
Bool_7
IN7
En
ANY_TO_SINTE...
Usint_1
IN
Q
Sint_1