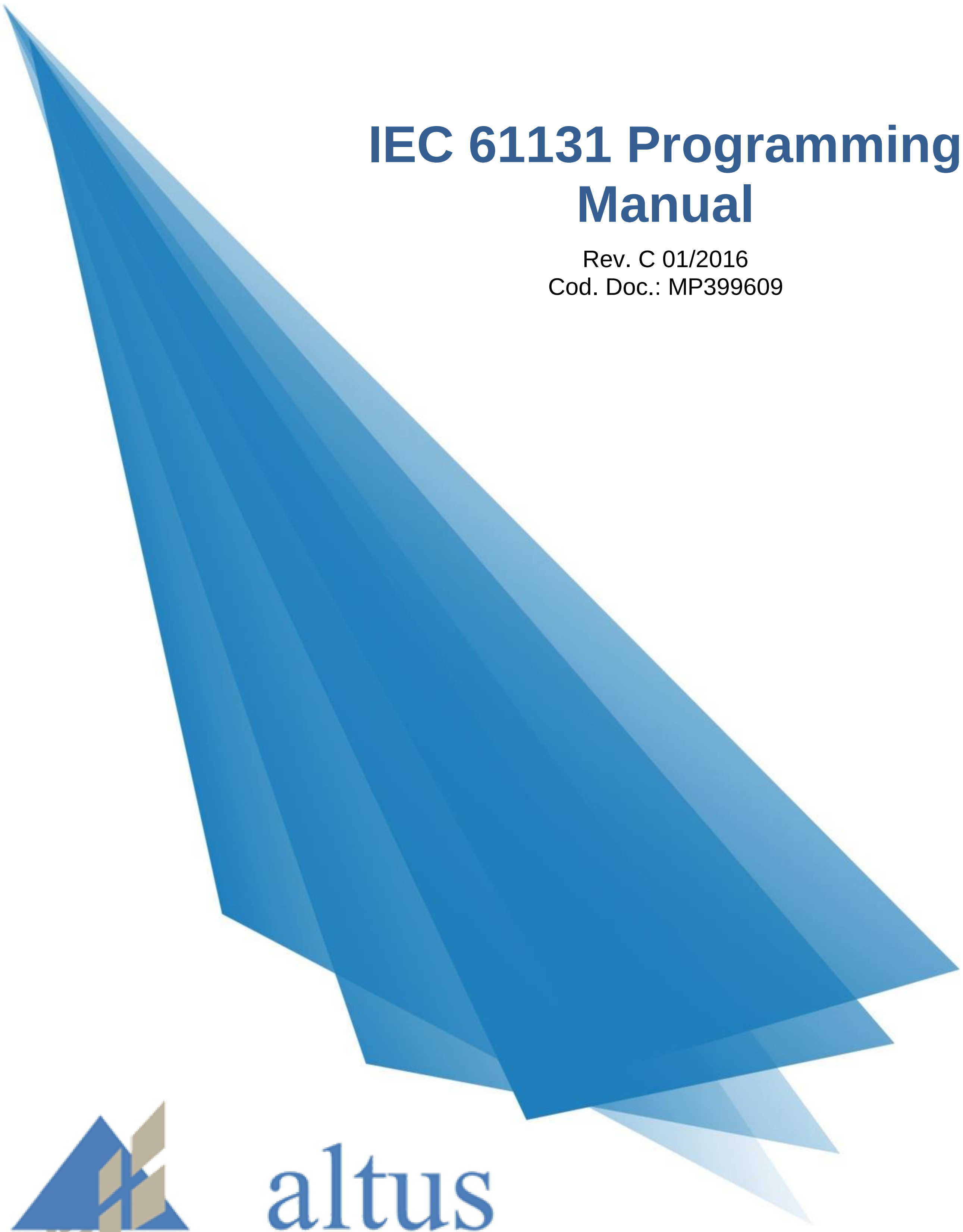
IEC 61131 Programming Manual

Rev. C 01/2016

Cod. Doc.: MP399609



altus

www.altus.com.br

According to current legislation in Brazil, the Consumer Defense Code, we are giving the following information to clients who use our products, regarding personal safety and premises.

The industrial automation equipment, manufactured by Altus, is strong and reliable due to the stringent quality control it is subjected to. However, any electronic industrial control equipment (programmable controllers, numerical commands, etc.) can damage machines or processes controlled by them when there are defective components and/or when a programming or installation error occurs. This can even put human lives at risk.

The user should consider the possible consequences of the defects and should provide additional external installations for safety reasons. This concern is higher when in initial commissioning and testing.

The equipment manufactured by Altus does not directly expose the environment to hazards, since they do not issue any kind of pollutant during their use. However, concerning the disposal of equipment, it is important to point out that built-in electronics may contain materials which are harmful to nature when improperly discarded. Therefore, it is recommended that whenever discarding this type of product, it should be forwarded to recycling plants, which guarantee proper waste management.

It is essential to read and understand the product documentation, such as manuals and technical characteristics before its installation or use.

The examples and figures presented in this document are solely for illustrative purposes. Due to possible upgrades and improvements that the products may present, Altus assumes no responsibility for the use of these examples and figures in real applications. They should only be used to assist user trainings and improve experience with the products and their features.

Altus warrants its equipment as described in General Conditions of Supply, attached to the commercial proposals.

Altus guarantees that their equipment works in accordance with the clear instructions contained in their manuals and/or technical characteristics, not guaranteeing the success of any particular type of application of the equipment.

Altus does not acknowledge any other guarantee, directly or implied, mainly when end customers are dealing with third-party suppliers.

The requests for additional information about the supply, equipment features and/or any other Altus services must be made in writing form. Altus is not responsible for supplying information about its equipment without formal request.

COPYRIGHTS

Summary

1. Introduction

Nexto Series is a powerful and complete Programmable Logic Controller (PLC) with unique and innovative features. Due to its flexibility, smart design, enhanced diagnostics capabilities and modular architecture, Nexto can be used for control systems from medium or high-end applications. Due to its compact size and superior performance, Nexto can also be used for small automation systems with time critical requirements.

MasterTool IEC XE is a complete tool for programming, debugging and performing configuration and simulation of user applications. Based on a concept of being integrated, flexible and easy to use, this software provides six programming languages defined by IEC 61131-3 standard: Structured Text (ST), Sequential Function Chart (SFC), Function Block Diagram (FBD), Ladder Diagram (LD), Instruction List (IL) and Continuous Function Chart (CFC). MasterTool IEC XE allows the use of different languages on the same application, providing to the user a powerful way to organize the application and to reuse codes used in previous applications.

This product offers features for every stage of an automation application, starting from initial graphical architecture topology analyses, passing through a programming environment that supports IEC 61131-3 languages and a realistic simulation tool, where the user can verify application's behavior before running in a real system and ending in a complete diagnostics and status visualization interface.

MasterTool IEC XE also offers two different protection schemes as application security features: IP Protection and Secure PLC Login. IP Protection is targeted to protect user's intellectual property, allowing the user to protect the complete project and files by defining a password to access them. This means that these files will be available (both read and write operations) only after unlocking them with the correct password. Secure PLC Login provides a way to protect the user application from any unauthorized access. By enabling this feature, Nexto CPU will request a user and a password before performing any available command in MasterTool IEC XE and Nexto CPU, as stopping, programming and forcing of outputs in the target CPU.

MasterTool IEC XE allows the use of fieldbus interfaces in an easier way than ever seen before. The user does not need any special software to configure fieldbuses anymore, because MasterTool IEC XE covers this requirement providing a unique tool reducing engineering time and making applications more simple.

In order to increase user's productivity, some important features are also available: Module Printing which is a report generation of every module specific parameters and application general settings, Logic Printing which is a report generation of all application code, Enhanced Project Verification which helps user to check several different conditions during programming, like programming syntax, power supply module current consumption, placement rules for Nexto modules, modules parameters and settings, Real Time Debugging which provides useful way to check the application step-by-step, verify variables values or add and remove breakpoints during Nexto CPU programming.

Documents Related to this Manual

For additional information about MasterTool IEC XE, you can examine other specific documents in addition to this one. These documents are available in its last review on www.altus.com.br.

General Regards on ALTUS Documentation

Each product has a document called Technical Characteristics (CT), where there are the characteristics for the product in question. Additionally, the product may have User Manuals (manual's codes, if applicable, are always mentioned at CTs from the respective modules).

MasterTool IEC XE Support Documentation

It is advisable to consult the following documents as a source of additional information:

Document Code	Description	Language
CE114000	Nexto Series – Features and Configuration	English
CT114000	Série Nexto – Características e Configurações	Portuguese
CS114000	Serie Nexto – Especificaciones y Configuraciones	Spanish
CE114100	CPUs Nexto Series – Features and Configuration	English
CT114100	UCPs Série Nexto – Características e Configurações	Portuguese
CS114100	UCPs Serie Nexto – Especificaciones y Configuraciones	Spanish
CE103705	MasterTool IEC XE – Features and Configuration	English
CT103705	MasterTool IEC XE – Características e Configurações	Portuguese
CS103705	MasterTool IEC XE – Especificaciones y Configuraciones	Spanish
MU214600	Nexto Series User Manual	English
MU214000	Manual de Utilização Série Nexto	Portuguese
MU214300	Manual Del Usuario Serie Nexto	Spanish
MU214605	Nexto Séries CPUs User Manual	English
MU214100	Manual de Utilização UCPs Série Nexto	Portuguese
MU214305	Manual del Usuario UCPs Serie Nexto	Spanish
MU299609	MasterTool IEC XE User Manual	English
MU299048	Manual de Utilização MasterTool IEC XE	Portuguese
MU299800	Manual del Usuario MasterTool IEC XE	Spanish
MU399609	IEC 61131 Programming Manual	English
MU399048	Manual de Programação IEC 61131	Portuguese
MU399800	Manual de Programación IEC 61131	Spanish

Table 1-1. Support Documentation

Visual Inspection

Prior to installation, we recommend performing a careful visual inspection of equipment, by checking if there is damage caused by shipping. Make sure all components of your order are in perfect condition. In case of defects, inform the transportation company and the nearest Altus representative or distributor.

CAUTION:
Before removing modules from the package, it is important to discharge eventual static potentials accrued in the body. For this, touch (with nude hands) in a metallic surface grounded before modules handling. Such procedure ensures that the levels of static electricity supported by the module will not be overcome.

It is important to record the serial number of each item received, as well as software revisions, if any. This information will be necessary if you need to contact Altus Technical Support.

Technical Support

To contact Altus Technical Support in São Leopoldo, RS, call +55 51 3589-9500. To find the existent centers of Altus Technical Support in other locations, see our website (www.altus.com.br) or send an email to altus@altus.com.br.

If the equipment is already installed, please have the following information when requesting assistance:

- Models of equipment used and the configuration of installed system
- Serial number of CPU
- Equipment review and executive software version, listed on the label affixed to the product side

- Information about the operation of CPU, obtained through MasterTool IEC XE programmer, and graphical display from CPU
- Contents of the application program, obtained through MasterTool IEC XE programmer
- Version of the programmer used.

Warning Messages Used in this Manual

In this manual, warning messages will present the following formats and meanings:

DANGER:
Relates potential causes, which if not noted, generate damages to physical integrity and health, property, environment and production loss.

CAUTION:
Relates configuration details, application and installation that shall be followed to avoid condition that could lead to system fail, and its related consequences.

ATTENTION:
Indicate important details to configuration, application or installation to obtain the maximum operation performance from the system.

2. Concepts and Basic Components

Introduction

MasterTool IEC XE is a device-independent PLC software programming. Matching the IEC 61131-3 standard it supports all standard programming languages.

Basic Concepts

Regard the following basic concepts determining programming with MasterTool IEC XE:

- **Object Orientation:** The mind of object orientation is not only reflected by the availability of appropriate programming elements and features but also in the structure and version handling of MasterTool IEC XE and in the project organization.
- **Component-based structure of the programming system:** The functionality available in the user interface (editors, menus etc.) depends on the currently used components defined in a profile. There are system components, which are essential, and optional components.
- **Project Organization is also determined by the mind of object orientation:** A MasterTool IEC XE project contains a PLC program composed of various programming objects and it contains definitions of the "resources" which are needed to run instances of the program (application) on defined target systems (devices, PLCs). So there are two main types of objects in a project:
    - **Programming objects:** Programming objects (POUs) which can be instantiated in the entire project, i.e. for all applications defined in the project, must be managed in the POUs window. These are programs, functions, function blocks, methods, actions, data type definitions etc. The instantiating is done by calling a program POU by an application-assigned task. Programming objects which are managed in the devices window, i.e. which are directly assigned to an application, cannot only be instantiated by another application inserted below.
    - **Resource objects:** These are device objects, applications, task configurations and which are managed in the "device tree" or in the graphic editor (depending on the device type). When inserting objects in the devices tree, the hardware to be controlled must be mapped according to certain rules.
- **Code generation:** By integrated compilers and use of machine code results in short execution times.
- **Data transfer to the controller device:** The data transfer between MasterTool IEC XE and the device is done via a gateway (component) and a runtime system.
- **Standard and professional interface:** Predefined feature sets serve to be able to choose between a "standard" user interface with a reduced selection of features and less complexity and a "professional" environment supporting all features. When the programming system is initially started after the first installation on the system, you will be prompted to choose one of the sets. But also later you still can switch the set and also a user-defined customization of the currently used feature set is possible. For the particular differences between the standard and professional version please see **Features** in the MasterTool IEC XE User Manual - MU299609.

Advanced Functionalities

In the following you can see the advanced functionalities available in the MasterTool IEC XE.

Object Orientation in Programming and in the Project Structure

Extensions for function blocks: Properties, Methods, Inheritance, Method Invocation.

Applications related to devices such as independents programming objects instances.

Special Data Types

- UNION
- LTIME
- References
- Enumerations: types of basic data can be specified
- DI: DINT := DINT#16#FFFFFFFF

Operators and Special Variables

- Scope operators: extended namespaces
- Function pointer: replacing the INSTANCE_OF operator
- Init Method: replacing the INI operator
- Exit Method
- Output variables in functions and method calls
- VAR_TEMP/VAR_STAT/VAR_RETAIN/ VAR_PERSISTENT
- Arbitrary expressions for variables initialization
- Assignment as expression
- Index access with pointers and strings

User Management and Access Rights Concept

- User accounts, user groups, specific rights to groups for access and actions in specific objects.

Characteristics in Editors

- ST Editor: editing resources, break line, autocomplete, monitoring and SET/RESET assignment in line
- FBD, LD, IL Editors reversible and programmable in one combined editor
- IL Editor as Table editor
- FBD, LD, IL Editors: possibility of changing the main output in boxes with multiple outputs
- FBD, LD, IL Editors without automatic update of the box parameters
- FBD, LD, IL Editors: branches and networks inside the networks
- SFC Editor: only one type of step, macros, multiple selections of independent elements, without syntactic checkup during editing and automatic declaration of signal variables

Library Versions

- Several library versions can be used in the same project using namespaces
- Installation in repositories, automatic update and debugging

Additional Functionalities

- Menus, toolbar and keyboard usage
- Inclusion of specific user components
- PC Configuration and task configuration integrated in the device tree
- UNICODE Support
- Comments in line
- Watchdog
- Multiple selection in the project object device
- Online help on the user interface
- Conditional building
- Conditional breakpoints
- Debugging: step to the cursor and return to previous call
- Field bus driver according to IEC 61131-3
- PC and symbol configuration available on the application
- Free allocation of code and data memory

- Each object can be specified as “internal” or “external” (late link on the runway system)
- Pre building notes concerning syntactic errors

Profiles

A MasterTool IEC XE project profile is a set of rules, common characteristics and patterns used in the development of an industrial automation solution. It’s a profile that influences the form of implementation of the application. With the diversity of types of applications supported by Nexto Series Runtime System, to follow a profile is a way to reduce the complexity in programming.

Applications can be created as one of the following profiles:

- Single
- Basic
- Normal
- Expert
- Custom

MasterTool IEC XE software provides a template called *MasterTool Standard Project*, which shall be selected by the user as a model in the project creation. The new application will be developed as a specific profile, also chosen by the user, adopting rules, features and predefined patterns. Each project profile defines.

To ensure compatibility of a project to a certain profile throughout development, we use two approaches:

- MasterTool IEC XE only allows the creation of projects based on a template by selecting at the same time the profile to be used
- In code generation, MasterTool IEC XE checks all the rules defined for the valid profile to the project

For further details, see **Profiles** in the User Manual Nexto Series CPUs – MU214605, chapter Initial Programming.

Project

A project contains the POU objects which make up a PLC program, as well as the definitions of resource objects necessary to run one or several instances of the program (application) on certain target systems (PLCs, devices). POU objects are managed in the POUs view window or at the Devices window. POUs created by the Wizard appear at the Devices window; Device specific resource objects are managed also managed in this window.

A project is saved in a file <project name>.project.

NOTE: The appearance and properties of the user interface are defined and stored in MasterTool IEC XE and not in the project.

Device

In the *Devices* window (“device tree”) the hardware can be mapped on which the application is to run.

Each “device object” represents a specific (target) hardware object. Examples: controller, field bus node, bus coupler, I/O-module, monitor.

Each device is defined by a device description and must be installed on the local system in order to be available for inserting in the *Device* tree (see below). The device description file defines the properties of a device concerning configurability, programmability and possible connections to other devices.

In the *Device* tree are managed objects needed to run an application on the device (controller, CP), including implementation, configuration tasks and tasks. However, specific programming objects (POUs, global variable lists and Library Manager) can - instead of being managed as units instantiable global design in the window of POUs - ONLY be managed in the *Device* tree and in this case are available only in its application or its "secondary applications".

Application

An “application” is a set of objects which are needed for running a particular instance of the PLC program on a certain hardware device (PLC, controller). For this purpose “independent” objects, managed in the POUs view, are instantiated and assigned to a device in the *Devices* view. This meets the mind of object orientated programming. However also purely application-specific POUs can be used.

An application is represented by an Application () object in the Devices tree , insertable below a programmable device node (PLC Logic). Below an application entry the objects defining the applications “resource set” are to be inserted.

The standard application, “Application”, is created along with new projects created from the template *MasterTool Standard Project*. It is added to the device tree below the item *Device* and *PLC Logic*.

An essential part of each application is the Task Configuration that controls the execution of a program (POUs instances or specific POU of the application). Additionally, resources objects can be assigned such as global variables lists, libraries, which - unlike those managed in the window of POUs - can only be used for specific application and its sub items.

When going to log in with an application on a target device (PLC or simulation target), it will be checked which applications currently are on the PLC and whether the application parameters on the PLC are matching those in the project configuration. Appropriate messages will indicate mismatches.

Task Configuration

The Task Configuration () defines one or several tasks for controlling the processing of an application program.

It is an essential resource object for an application and it is inserted automatically when you create a new project from *MasterTool Standard Project*. A task can call an application-specific program POU, which is only available in the device tree below the application, as well as a program which is managed in the POUs window. In the latter case the project-globally available program will be instantiated by the application.

A task configuration can be edited in the Task Editor, the available options being target-specific.

In online mode the Task Editor provides a monitoring view giving information on cycles, cycle times and task status.

Important Notes for Multitasking Systems

On some systems real preemptive multitasking is realized. In this case the following must be regarded.

All tasks share one process map. Reason: An own process map for each task would charge the performance. So however the process map always can only be consistent to one task. Hence the user, when creating a project, explicitly must take care that in case of conflicts the input data will be copied to a save area, the same problems have to be regarded for the outputs. For example, modules of the SysSem.library could be used to solve the synchronization problems.

Also when accessing other global objects (global variables, modules), consistency problems might occur as soon as the size of the objects exceeds the data width of the processor (structures or ARRAYS forming a logical unit). Also here the modules of the SysSem.library might be used to solve the problems.

Communication

For information about **Communication** (Configuration of a PLC, Network topology, Addressing and routing, Structure of addresses and Network variables), see MasterTool IEC XE User Manual – MU299609.

Code Generation and Online Change

Code Generation and Compile Information

Machine code will not be generated until the application project gets downloaded to the target device (PLC, simulation target). At each download the compile information, containing the code and a reference ID of the loaded application, will be stored in the project directory in a file "<project name>.<device name>.<application>.compileinfo". The compile info will be deleted when the *Clean* and *Clean all* command are performed.

Online Change

If the application project currently running on the controller has been changed in the programming system since it has been downloaded last, just the modified objects of the project will be loaded to the controller while the program keeps running there.

ATTENTION:
Online Change modifies the running application program and does not affect a restart process. Make sure that the new application code nevertheless will affect the desired behavior of the system. Depending on the controlled system, damages to machines and parts could result, or even health and life of persons could be endangered.

NOTES:
- When an online change is done, the application-specific initializations (homing etc.) will not be executed because the machine keeps its state. For this reason the new program code might not be able to work as desired.
- Pointer variables keep their values form the last cycle. If there is a pointer on a variable, which has changed its size due to an online change, the value will not be correct any longer. Make sure that pointer variables get re-assigned in each cycle.

Boot Application (Boot Project)

A boot application is the project which will be started automatically when the controller gets started . For this purpose the project must be available on the PLC in a file <project name>.app. This file can be created in offline mode via the *Create boot application* command (*Online* menu).

Each sending was successful, the active application will be automatically stored in the file "<application>.app" in the system folder of the device, thus available as a startup application. The *Create boot application* command also lets you save a file in this application in offline mode.

Sending/Login Project Method without Project Differences

In order to ensure that the user won't have problems on sending and logging same projects in the CPUs running from different stations, it can be performed the following steps after sending a project:

- In the *Additional files* dialog (*Project*, *Project Settings*, *Source Download* menu and *Additional files..* button) select the option *Download information files*.
- Close all dialogs by clicking *OK*
- Run the command *Source download* (*File* menu)
- In the *Select Device* dialog that opens, choose the CPU on which the project was sent
- Close the dialog by clicking *OK*, wait for the download of the project

To login on running CPUs without generating changes on project from different stations, you must open a Project Archive generated from the original project and execute the *Login* command. In the absence thereof may be made of the following steps:

- Run the command *Source upload...* (*File* menu)
- In the *Select Device* dialog that opens, choose the CPU on which the project was sent
- Close the dialog by clicking *OK*, wait for the loading of the project
- In the *Project Archive* dialog, which opens at the end of the loading process, choose the folder to extract and click on *Extract*
- The project will open and *Login* command can be executed in the corresponding CPU

For further information see: **File Menu** and **Online Menu** in the MasterTool IEC XE User Manual - MU299609.

Monitoring

In online mode there are various possibilities to display the current values of the watch expressions of an object on the PLC:

For more information see **Monitoring** in the MasterTool IEC XE User Manual - MU299609.

Debugging

To evaluate programming errors you can use the MasterTool IEC XE debugging functionality in online mode. In this context regard the possibility to check an application in simulation mode, i.e. without the need of connecting to a real hardware target device.

Breakpoints can be set at certain positions to force an execution break. Certain conditions, such as which tasks are concerned and in which cycles the breakpoint should be effective, can be set for each breakpoint. Stepping functions are available to get a program executed in controlled steps. At each break the current values of the variables can be examined. A call stack can be viewed for the currently reached step position.

For further information about this item see: **Breakpoints** in the MasterTool IEC XE User Manual - MU299609.

Supported Programming Languages

All of the programming languages mentioned in the IEC standard IEC 61131 are supported via specially adapted editors.

- FBD/LD/IL editor for Function Block Diagram (FBD), Ladder Logic Diagram (LD) and Instruction List (IL)
- SFC editor for Sequential Function Chart
- ST editor for Structured Text

Additionally MasterTool IEC XE provides an editor for programming in CFC that is not part of the IEC standard: CFC editor for Continuous Function Chart.

Program Organization Units

It is basically used for all objects which are used to create a PLC program.

POUs which are managed in the "POUs" view are not device-specific but they might be instantiated for the use on a device (application). For this purpose program POUs must be called by a task of the respective application.

POUs, which are ONLY managed in the Devices view, i.e. which are inserted in the "device tree" explicitly below an application, can only be instantiated by applications indented below this application (child application). For further information see the descriptions of the **Device Tree** and **Application** in the MasterTool IEC XE User Manual – MU299609.

But POU also is the name of a certain sub-category of these objects in the *Add Object* menu, at this place just comprising programs, function blocks and functions.

So, a Program Organization Unit object in general is a programming unit, an object which is managed either non-device-specifically in the *POUs* window or device-specifically in the *Devices* window and can be viewed and edited in an editor window. A POU object can be a program, function, function block as well as a method, action, DUT (Data Unit Type) or an external file of any format.

Regard the possibility to set certain properties (like e.g. build conditions etc.) for each particular POU object.

The following POU object types can be used per default:

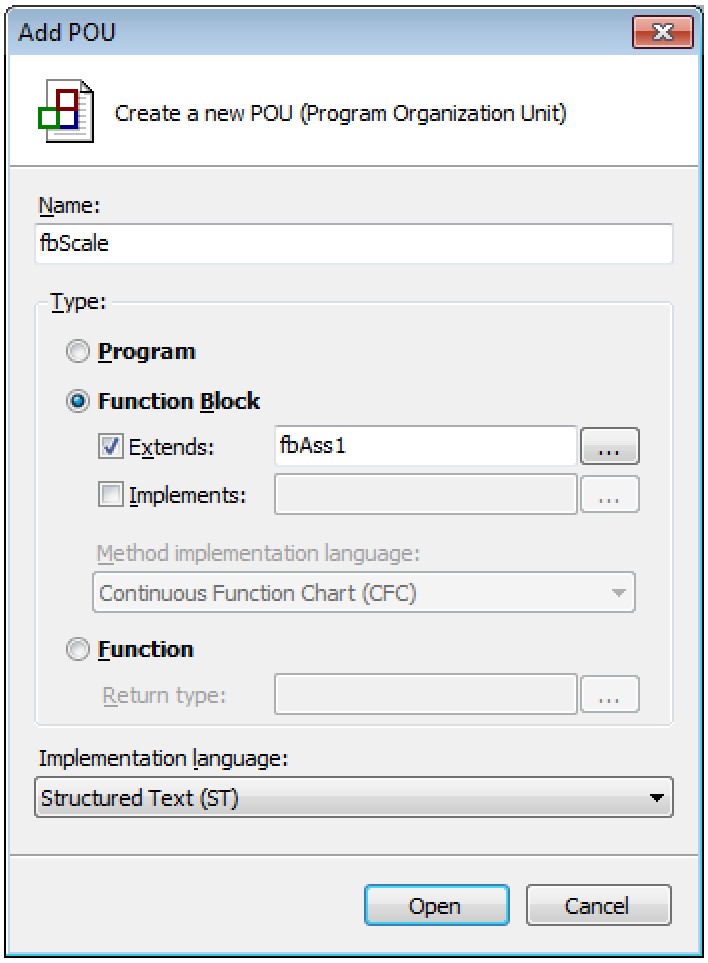
- POU
- Action
- DUT (Data Type Unit)
- External file
- Global Variable List
- Method
- Property
- Program
- Function
- Function Block
- Persistent Vars
- POUs for Implicit Checks

Besides the Program Organization Unit objects there are Device objects used for running the program on the target system (resource, application, task configuration, etc...). Those are managed in the *Devices* view.

POU

A POU - in this context - is a Program Organization Unit of type Program, Function or Function block.

In order to add a POU () select an appropriate entry in the *POUs* or *Devices* tree (for example an application object), use command *Add Object* from the context menu and select POU from the appearing submenu. The *Add POU* dialog will open, where you have to configure the POU concerning name, type and implementation language. In case of a function block optionally EXTENDS and IMPLEMENTS properties can be defined, in case of a function also the return type must be specified. See the respective chapters on **Program**, **Function**, and **Function Block**.


Add POU

Create a new POU (Program Organization Unit)

Name: fbScale

Type:

( ) Program

(●) Function Block

[✓] Extends: fbAss1

[ ] Implements:

Method implementation language: Continuous Function Chart (CFC)

( ) Function

Return type:

Implementation language: Structured Text (ST)

Figure 2-1. Add POU Dialog

Depending on the type the POU can be supplemented by methods, properties, actions, transitions. For this also use the *Add Object* command.

The hierarchical order of processing the POU instances of an application depends on a device specific configuration (call stack).

Each POU consists of a declaration part and an implementation part. The body is written in one of the available programming languages, e.g. IL, ST, SFC, FBD, LD or CFC.

The MasterTool IEC XE supports all POUs described by IEC 61131-3. To use these POUs in the project, you must include the standard.library library. The projects created from the template *MasterTool Standard Project* already have this library loaded.

NOTE: In some examples of this manual the code is declared sequentially, but to use it must be separated, the top part of the programming language editor is used for declarations and the bottom part should be used for the implementations.

Calling POUs

POUs can call other POUs. Recursions however are not allowed.

When a POU assigned to an application calls another POU just by its name (without any namespace added), regard the following order of browsing the project for the POU to be called:

1. Current application

2. Library Manager of current application

3. POUs view

4. Library Manager in POUs view

If a POU with the name specified in the call is available in a library of the applications library manager as well as an object in the POUs view, there is no syntax for explicitly calling the POU in the POUs view just by using its name. In this case you would have to move the concerned library from the applications library manager to the POUs view library manager. Then you could call the POU from the POUs view just by its name (and if needed that from the library by preceding the library namespace).

For further information see also the item: **POUs for Implicit Checks.**

Program

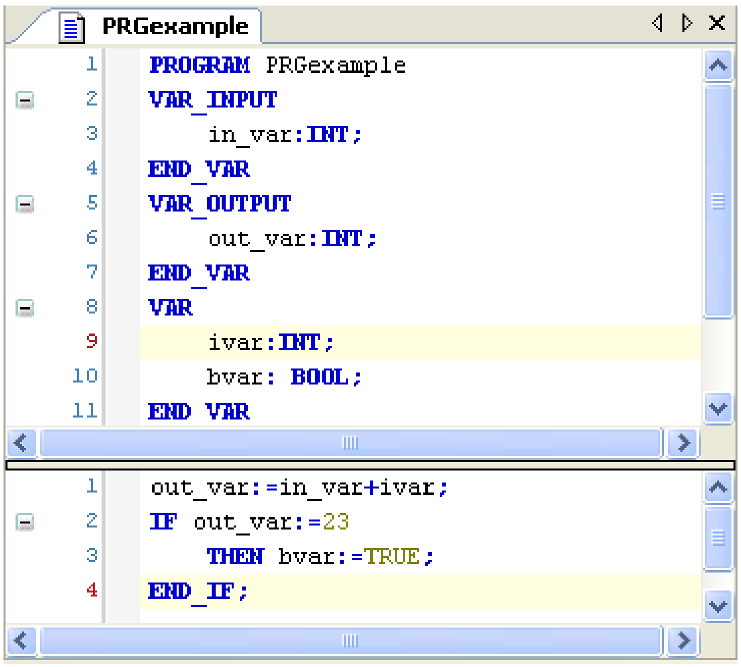
A program is a POU which returns one or several values during operation. All values are retained from the last time the program was run until the next.

A program POU can be added to the project via the *Add Object* command. To assign the program to an existing application, select the application entry in the Devices view and use the command from the context menu. Otherwise it will be added to the *POUs* view. In the *Add POU* dialog choose type *Program*, enter a name for the program and set the desired implementation language. After confirming the settings with button *Open* the editor window for the new program will open and you can start editing the program.

The syntax for declaring a program:

```
PROGRAM <PROGRAM NAME>
```

This is followed by the variable declarations of input, output and program variables, optionally also access variables.



```
PROGRAM PRGexample
VAR_INPUT
    in_var:INT;
END_VAR
VAR_OUTPUT
    out_var:INT;
END_VAR
VAR
    ivar:INT;
    bvar: BOOL;
END_VAR
```

```
out_var:=in_var+ivar;
IF out_var:=23
    THEN bvar:=TRUE;
END_IF;
```

Figure 2-2. Program

Program Calls

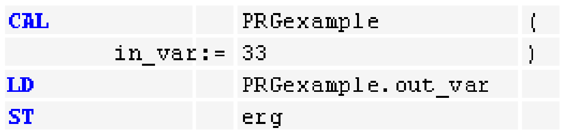
A program can be called by another POU. But: A program call in a function is not allowed. There are also no instances of programs.

If a POU has called a program and if thereby the values of the program have been changed, these changes will be retained until the program gets called again, even if it will be called from within another POU. Regard that this is different from calling a function block, where only the values in the given instance of the function block are changed and so the changes only are of effect when the same instance will be called again.

If you want to set input and/or output parameters in the course of a program call, you can do this in text language editors (e.g. ST) by assigning values to the parameters after the program name in parentheses. For input parameters this assignment takes place using ":=" just as with the initialization of variables at the declaration position, for output parameters "=>" is to be used. See for an example below.

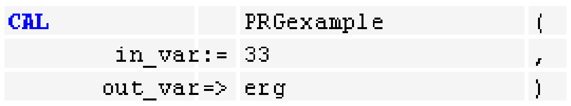
If the program is inserted via *Input Assistant* with option *Insert With Arguments* in the implementation window of a text language editor, it will be displayed automatically according to this syntax with all parameters. However you not necessarily must assign these parameters.

Examples for program calls in IL:



```
CAL                PRGexample          (
         in_var:=  33                  )
LD                 PRGexample.out_var
ST                 erg
```

Assigning the parameters (*Input Assistant With Arguments*, see above):



```
CAL                PRGexample          (
         in_var:=  33                  ,
        out_var=>  erg                 )
```

Example in ST:

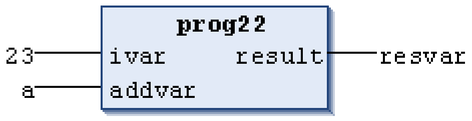
```
PRGEXAMPLE();
ERG := PRGEXAMPLE.OUT_VAR;
```

Assigning parameters (*Input Assistant With Arguments*):

```
PRGEXAMPLE (in_var:=33, out_var=>erg );
```

NOTE: Brackets are obligatory.

Example in FBD:



Function

A function is a POU, which yields exactly one data element (which can consist of several elements, such as fields or structures) when it is processed, and whose call in textual languages can occur as an operator in expressions.

A function POU can be added to the project via command *Add Object* and *Add POU*. To assign the function to an existing application, select the application entry in the *Devices* view and use the command from the context menu. Otherwise it will be added to the *POUs* view. In the *Add POU* dialog choose type *Function*, enter a name (<function name>) and a return data type (<data type>) for the new function and choose the desired implementation language. After confirming the settings with button *Open* the editor window for the new function will open and you can start editing.

Declaration:

A function declaration begins with the keyword FUNCTION. A name and a data type must be defined.

Syntax:

```
FUNCTION < FUNCTION NAME> : <DATA TYPE>
```

This is followed by the variable declarations of input and function variables.

A result must be assigned to a function. That is that the function name is used as an output variable.

NOTE: If a local variable is declared as RETAIN in a function, this is without any effect. The variable will not be written to the Retain area.

Example of a function in ST:

```
FUNCTION FCT : INT
VAR_INPUT
IVAR1:INT;
IVAR2:INT;
IVAR3:INT;
END_VAR
FCT:=IVAR1+IVAR2*IVAR3;
```

This function takes three input variables and returns the product of the second two added to the first one.

Function Call

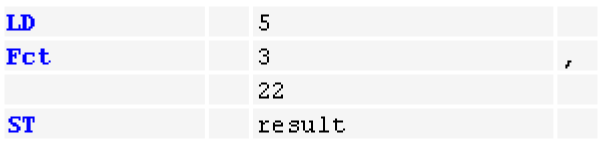
The call of a function in ST can appear as an operand in expressions.

In IL a function call only can be positioned within actions of a step or within a transition.

Functions (in contrast to a program or function block) contain no internal state information, that is, invocation of a function with the same arguments (input parameters) always will yield the same values (output). For that reason functions may not contain global variables and addresses.

Examples of function calls:

In IL:

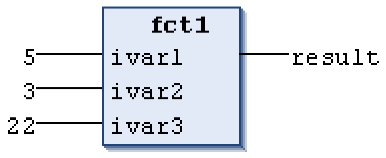


```
LD      5
Fct     3       ,
        22
ST      result
```

In ST:

```
RESULT := FCT1(5, 3, 22);
```

In FBD:



In function calls it is no longer possible to mix explicit parameter assignment with implicit ones. This allows changing the order of parameter input assignments.

Example:

```
FUN(FORMAL1 := ACTUAL1, ACTUAL2); // -> Error Message
FUN(FORMAL2 := ACTUAL2, FORMAL1 := ACTUAL1); // Same Semantics As The:
FUN(FORMAL1 := ACTUAL1, FORMAL2 := ACTUAL2);
```

According to the IEC 61131-3 standard, functions can have additional outputs. Those must be assigned in the call of the function, for example in ST according to syntax:

```
OUT1 => <OUTPUT VARIABLE 1> | OUT2 => < OUTPUT VARIABLE 2> | ... FURTHER
OUTPUT VARIABLES
```

Example:

Function fun is defined with two input variables in1 and in2. The return value of fun will be written to the locally declared output variables (VAR_OUTPUT) loc1 and loc2.

```
FUN(IN1 := 1, IN2 := 2, OUT1 => LOC1, OUT2 => LOC2);
```

Function Block

A function block is a POU which provides one or more values during the processing of a PLC program. As opposed to a function, the values of the output variables and the necessary internal variables shall persist from one execution of the function block to the next. So invocation of a function block with the same arguments (input parameters) need not always yield the same output values.

In addition to the functionality described by standard IEC 61131-3 object oriented programming will be supported and function blocks can be defined as extensions of other functions. This means that "inheritance" can be used when programming with function blocks.

A function block always is called via an instance, which is a reproduction (copy) of the function block.

A function block can be added to the project via command *Add Object / POU*. To assign the function block to an existing application, select the application entry in the *Devices* view and use the command from the context menu. Otherwise it will be added to the *POUs* view.

In the *Add object* dialog choose type *Function Block*, enter a function block name (<identifier>) and choose the desired implementation language.

Additionally the following options may be set:

- *Extends*: Enter here the name of another function block available in the project, which should be the base for the current one.
- *Implements*: Not supported

After having confirmed the settings with button *Open* the editor window for the new function block will open and you can start editing.

Declaration:

Syntax:

```
FUNCTION_BLOCK <FUNCTION BLOCK NAME> | EXTENDS <FUNCTION BLOCK NAME>
```

This is followed by the declaration of the variables.

NOTE: FUNCTION BLOCK is no longer a valid alternative keyword.

Example:

FBexample shown in the following picture has two input variables and two output variables out1 and out2. Out1 is the sum of the two inputs, out2 is the result of a comparison for equality.

Example of a function block in ST:

```
FUNCTION_BLOCK FBEXAMPLE
VAR_INPUT
INP1:INT;
INP2:INT;
END_VAR
VAR_OUTPUT
OUT1:INT;
OUT2:BOOL;
END_VAR
OUT1:=INP1+INP2;
OUT2:= INP1=INP2;
```

Function Block Instance

Function blocks are always called through an instance which is a reproduction (copy) of a function block.

Each instance has its own identifier (instance name), and a data structure containing its inputs, outputs, and internal variables.

Instances like variables are declared locally or globally, whereby the name of the function block is indicated as the data type of an identifier.

Syntax for declaring a function block instance:

```
<IDENTIFIER>:<FUNCTION BLOCK NAME>;
```

Example:

Declaration (e.g. in the declaration part of a program) of instance INSTANCE of function block FUB:

```
INSTANCE: FUB;
```

The declaration parts of function blocks and programs can contain instance declarations. But: Instance declarations are not permitted in functions.

Calling a Function Block

Function blocks are always called through a function block instance. Thus a function block instance must be declared locally or globally.

The desired function block variable can be accessed using the following syntax:

Syntax:

```
<INSTANCE NAME>.<VARIABLE NAME>
```

Regard the following:

- Only the input and output variables of a function block can be accessed from outside of a function block instance, not its internal variables
- Access to a function block instance is limited to the POU in which it was declared unless it was declared globally
- At calling the instance the desired values can be assigned to the function block parameters. See below
- The InOutVariables (VAR_IN_OUT) of a function block are passed as pointers
- In SFC function block calls can only take place in steps
- The instance name of a function block instance can be used as an input parameter for a function or another function block
- All values of a function block are retained until the next processing of the function block. Therefore function block calls do not always return the same output values, even if done with identic arguments

NOTE: If there at least one of the function block variables is a remanent variable, the total instance is stored in the retain data area.

Examples for accessing function block variables.

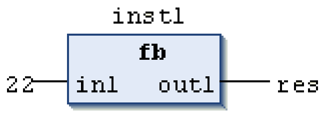
Assume: function block “fb” has an input variable “in1” of the type INT. See here the call of this variable from within program prog.

Declaration and implementation in ST:

```
PROGRAM PROG
VAR
INST1:FB;
```

```
RES:INT;
END_VAR
INST1.IN1:=22; (* fb is called and input variable in1 gets assigned value
22 *)
INST1(); (* fb is called, this is needed for the following access on the
output variable *)
RES:=FBINST.OUTL; (* output variable of fb is read *)
```

Example in FBD:



Assigning Parameters at Call

In the text languages IL and ST you can set input and/or output parameters immediately when calling the function block. The values can be assigned to the parameters in parentheses after the instance name of the function block. For input parameters this assignment takes place using ":=" just as with the initialization of variables at the declaration position, for output parameters "=>" is to be used.

In this example a timer function block (instance CMD_TMR) is called with assignments for the parameters IN and PT. Then the result variable Q is assigned to the variable A. The result variable is addressed with the name of the function block instance, a following point, and the name of the variable:

```
CMD_TMR(IN := %IX5.0, PT := 300);
A:=CMD_TMR.Q
```

If the instance is inserted via *Input Assistant* (<F2>) with option *Insert With Arguments* in the implementation window of a ST or IL POU, it will be displayed automatically according to the syntax showed in the following example with all of its parameters. But you not necessarily must assign these parameters. For the above mentioned example the call would be displayed as follows:

Example, insert via *Input Assistant* with arguments:

```
CMD_TMR(in:=, pt:=, q=>)
->
CMD_TMR(in:=bvar, pt:=t#200ms, q=>bres);
```

Extension of a Function Block

Supporting object orientated programming a function block can be derived from another function block. This means a function block can extend another, thus automatically getting the properties of the basing function block in addition to its own.

The extension is done by using the keyword EXTENDS in the declaration of a function block. You can choose the "extends" option already during adding a function block to the project via the *Add Object* dialog.

Syntax:

```
FUNCTION_BLOCK <function block name> EXTENDS <function block name>
```

This is followed by the declaration of the variables.

Example of definition of function block fbA:

```
FUNCTION_BLOCK fbA
VAR_INPUT
x:int;
...
```

Definition of function block fbB:

```
FUNCTION_BLOCK fbB EXTENDS fbA
VAR_INPUT
ivar:int;
...
```

In the above example:

- FbB does contain all data and methods which are defined by fbA. An instance of fbB can now be used in any context were a function block of type fbA is expected
- FbB is allowed to override the methods defined in fbA. That means: fbB can declare a method with the same name and the same inputs and output as declared by A
- FbB is not allowed to use function block variables with the same name as used in fbA. In this case the compiler will prompt an error
- FbA variables and methods can be accessed directly within an fbB-Scope by using the keyword SUPER (SUPER^.<method>)

NOTE: Multiple inheritances are not allowed, i.e. it’s not possible to extend a Function Block more than once.

Method Invocation

Object oriented programming with function blocks is - besides of the possibility of extension via EXTENDS. This requires dynamically resolved method invocations, also called "virtual function calls".

See Method for further information on **Method**.

Virtual function calls need some more time than normal function calls and are used when:

- A call is performed via a pointer to a function block (pfub^.method)
- A method calls another method of the same function block

Virtual function calls make possible that the same call in a program source code will invoke different methods during runtime.

According to the IEC 61131-3 standard, methods like normal functions can have additional outputs. Those must be assigned in the method call according to syntax:

```
<METHOD>(IN1:=<VALUE> |, FURTHER INPUT ASSIGNMENTS, OUT1 => <OUTPUT
VARIABLE 1> | OUT2 => < OUTPUT VARIABLE 2> | ... FURTHER OUTPUT
ASSIGNMENTS)
```

This effects that the output of the method is written to the locally declared output variables as given within the call.

Example:

Assume that function blocks fub1 and fub2 EXTEND function block fubbase. Method method1 is contained.

Possible use of the method calls:

```
VAR_INPUT
B : BOOL;
END_VAR
VAR
PINST : POINTER TO FUBBASE;
INSTBASE : FUBBASE;
INST1 : FUB1;
INST2 : FUB2;
END_VAR
IF B THEN
PINST := ADR(INSTBASE);
ELSE
PINST := ADR(INST1);
```

```
END_IF
PINST^.METHOD1(); (* If B is true, FUBBASE.METHOD1 is called, else
FUB1.METHOD1 is called *)
```

Now assume that fubbase of the upper example contains two methods method1 and method2. fub1 overrides method2 but not method1. method1 is called like shown in the upper example:

```
PINST^.METHOD1(); (*If B is true FUBBASE.METHOD1 is called, else
FUB1.METHOD1 is called*)
```

Call via pointers - method1 implementation:

```
METHOD METHOD1 : BOOL
VAR_INPUT
END_VAR
METHOD1 := METHOD2();
```

For calling fubbase.method1 via pointers the following is true:

- If THIS is of type fubbase, fubbase.method2 will be called
- If THIS is of type fub1, fub1.method2 will be called

Data Type Unit

Along with the standard data types the user can define own data types. Structures, enumeration types and references can be created as Data Type Units (DUTs) in a DUT editor.

A DUT ( ) object can be added to the project via the *Add Object* command. To assign it to an existing application, first select the application entry in the *Devices* view. Otherwise it will be added to the POUs view. In the *Add DUT* dialog enter a name for the new data type unit and choose the desired type *Structure*, *Enumeration*, *Alias* or *Union*.

In case of type *Structure* you might utilize the principle of inheritance, thus supporting object oriented programming. Optionally you can specify that the DUT should extend another DUT, which is already defined within the project. This means that the definitions of the extended DUT will be automatically valid within the current one. For this purpose activate option Extends: and enter the name of the other DUT.

After confirming the settings with button *Open* the editor window for the new DUT will open and you can start editing.

Syntax for declaration of a DUT:

```
TYPE <IDENTIFIER> : <DUT COMPONENTS DECLARATION>
END_TYPE
```

The DUT components declaration depends on the type of DUT, e.g. a structure or an enumeration.

Example:

See in the following two DUTS, defining structures struct1 and struct2; struct2 extends struct1, which means that you can use struct2.a in your implementation to access variable a.

```
TYPE struct1 :
STRUCT
A:INT;
B:BOOL;
END_STRUCT
END_TYPE

TYPE struct2 EXTENDS struct1 :
STRUCT
C:DWORD;
D:STRING;
END_STRUCT
END_TYPE
```

Method

NOTE: This functionality is only available if supported by the currently used feature set.

Supporting object oriented programming, *Methods* ( ) can be used to describe a sequence of instructions. Like a function a method is not an independent POU, but must be assigned to a function block. It can be regarded as a function which contains an instance of the respective function block.

Inserting Methods

To assign a method to a function block, select the appropriate function block entry in the POUs or Device tree and in the context menu use *Add Object / Method*. In the *Add Method* dialog enter a name, the desired return type and the implementation language. For choosing the return data type you can use the button ... to get the *Input Assistant* dialog. After having confirmed the settings via *Open* the method editor view will be opened.

Declaration

Syntax:

```
METHOD <METHOD NAME> : <DATA TYPE>
VAR_INPUT
X: INT;
END_VAR
```

Method Call

Method calls are also named virtual function calls. Please see: **Method Invocation**.

NOTES:
- All data of a method is temporary and only valid during the execution of the method (stack variables).
- In the body of a method access to the function block instance variables is allowed.
- THIS Pointer: You can use identifier THIS to point directly to the implicit function block instance, which is available automatically. Notice that a locally declared variable might hide a function block variable. So in case of the example shown above in the syntax description THIS^.x would not refer to the methods input x, but to the function block variable x VAR_IN_OUT or VAR_TEMP-variables of the function block cannot be accessed in a method.
-Methods, like functions, can have additional outputs. Those must be assigned during method invocation.

Special Methods for a Function Block

- Init Method: A method named FB_Init always is declared implicitly, but also can be declared explicitly. It contains initialization code for the function block as declared in the declaration part of the function block. See: **FB_Init**
- Reinit Method: If a method named FB_Reinit is declared for a function block instance, it will be called after the instance has been copied and will reinitialize the new instance module. See: **FB_Reinit**
- Exit Method: If an exit method named FB_Exit is desired (for example for deallocating any, it must be declared explicitly. There is no implicit declaration. The exit method will be called for each instance of the function block before a new download, a reset or during online change for all moved or deleted instances. See: **FB_Exit**
- Properties: Set and Get methods; see: **Property**

Method Call also (Application is Stopped)

In the device description file it can be defined that a certain method should always be called task-cyclically by a certain function block instance (of a library module). If this method has the following input parameters, it will be processed also when the active application currently is not running.

```
VAR_INPUT
pTaskInfo : POINTER TO DWORD;
pApplicationInfo: POINTER TO _IMPLICIT_APPLICATION_INFO;
END_VAR
```

The programmer now can check the application status via pApplicationInfo, and define what should happen.

Example:

```
IF PAPPLICATIONINFO^.STATE=RUNNING THEN <INSTRUCTIONS> END_IF
```

Property

NOTE: This functionality is only available if supported by the currently used feature set.

A *Property* ( ) is an object type, which can be inserted below a program or function block via command *Add object / Property* from the context menu. In the *Add Property* dialog the name, return type and desired implementation language have to be specified.

A property contains two special methods which will be inserted automatically in the objects tree below the property object:

- The Set method is called when the property is written, that is the name of the property is used as input
- The Get method is called when the property is read, that is the name of the property is used as output

Example: Function block FB1 uses a local variable MILLI. This variable is determined by the properties Get and Set:

Code on property Get:

```
SECONDS := MILLI / 1000;
```

Code on property Set:

```
MILLI := SECONDS * 1000;
```

You can write the property of the function block (Set method) for example by fbinst.seconds := 22; (fbinst is the instance of FB1) or you can read the property of the function block (Get method) for example by testvar := fbinst.seconds;.

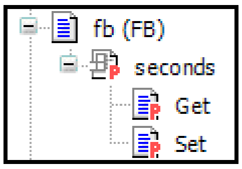


Figure 2-3. Property "Seconds" Added to Function Block Fb

A property can have additional local variables but no additional inputs and - in contrast to a function or method - no additional outputs.

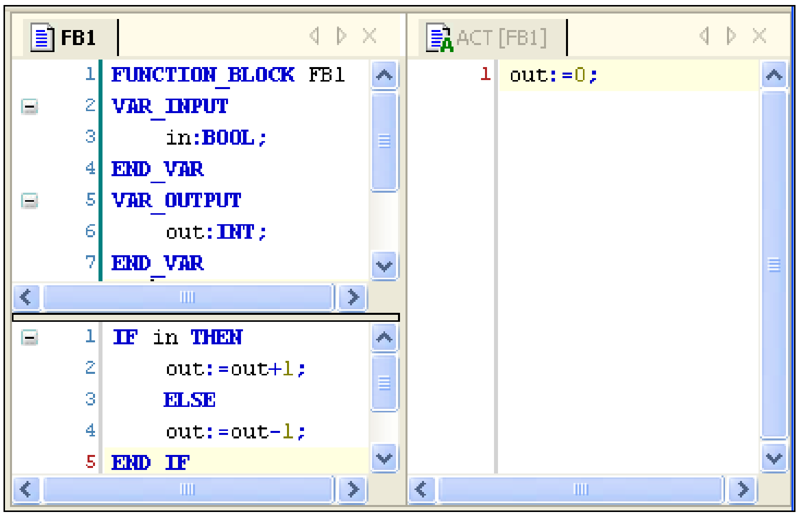
Monitoring a Property

A property may be monitored in online mode either with help of *Online Monitoring* or with help of a *Watch List*. The condition precedent to monitoring a property is the addition of the pragma {attribute 'monitoring':='variable'} on top of its definition.

Action

Actions ( ) can be defined and assigned to function blocks and programs using the command *Add Object*. An action is an additional implementation which can be created in a different language than the "basic" implementation is. Each action is given a name.

An action works with the data of the function block or program which it belongs to. It uses the input/output variables and local variables defined there and does not contain own declarations.



```
FUNCTION_BLOCK FB1
VAR_INPUT
    in:BOOL;
END_VAR
VAR_OUTPUT
    out:INT;
END_VAR
```

```
IF in THEN
    out:=out+1;
    ELSE
    out:=out-1;
END IF
```

```
out:=0;
```

Figure 2-4. Example of an Action of a Function Block

In this example each call of the function block FB1 will increase or decrease the output variable Out, depending on the value of the input variable “in”. Calling action Reset of the function block will set the output variable Out to zero. The same variable Out is written in both cases.

An action can be added via command *Add Object / Action* when the respective program or function block object is selected in the *Devices* or *POUs* tree. In the *Add Action* dialog define the action name and desired implementation language.

Calling an Action

An action is called with:

`<PROGRAM_NAME>.<ACTION_NAME > OU <INSTANCE_NAME>.<ACTION_NAME>.`

Regard the notation in FBD (see example below).

If it is required to call the action within its own block, i.e. in the program or function block it belongs to, it is sufficient to just use the action name.

Examples for the call of the above described action from another POU:

Declaration for all examples:

```
PROGRAM MAINPRG
VAR
    INST : COUNTER;
END_VAR
```

Call of action Reset in another POU, which is programmed in IL:

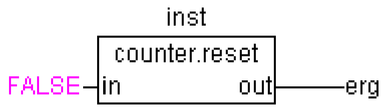
```
CAL INST.RESET(IN := FALSE)
LD INST.OUT
ST ERG
```

Call of action Reset in another POU, which is programmed in ST:

```
INST.RESET(IN := FALSE);
```

```
ERG := INST.OUT;
```

Call of action Reset in another POU, which is programmed in FBD:



NOTE: The IEC standard does not recognize actions other than actions of the sequential function chart (SFC). In there, actions are an essential part, containing the instructions to be processed at the particular steps of the chart.

External Function, Function Block, Method

For an external function, function block or method no code will be generated by the programming system.

Perform the following steps to create an external POU:

- Add the desired POU object in the POUs view of your project like any internal object and define the respective input and output variables

NOTE: Local variables must be defined in external function blocks, but may not be defined in external functions or methods. Also notice that VAR_STAT variables cannot be used in the runtime system.

In the runtime system an equivalent function, function block or method must be implemented. At a program download for each external POU the equivalent will be searched in the runtime system and - if found - will be linked.

Global Variable List - GVL

A () Global Variables List (GVL) is used to declare global variables. If a GVL is placed in the POUs view, the variables will be available all over the project, if a GVL is assigned to a certain application, the variables will be valid within this application.

A GVL can be added via command *Add Object* and *Add Global Variable List.* To assign it to an existing application, choose the command from the context menu while the application is selected in the device tree. Otherwise the new GVL object will be added to the *POUs* view.

The GVL editor is used to edit a Global Variable List.

If the target system supports the network functionality, the variables contained in a GVL can be defined to be available as network variables, i.e. for a broadcast data exchange with other devices in the network. For this purpose appropriate network properties must be configured for the GVL.

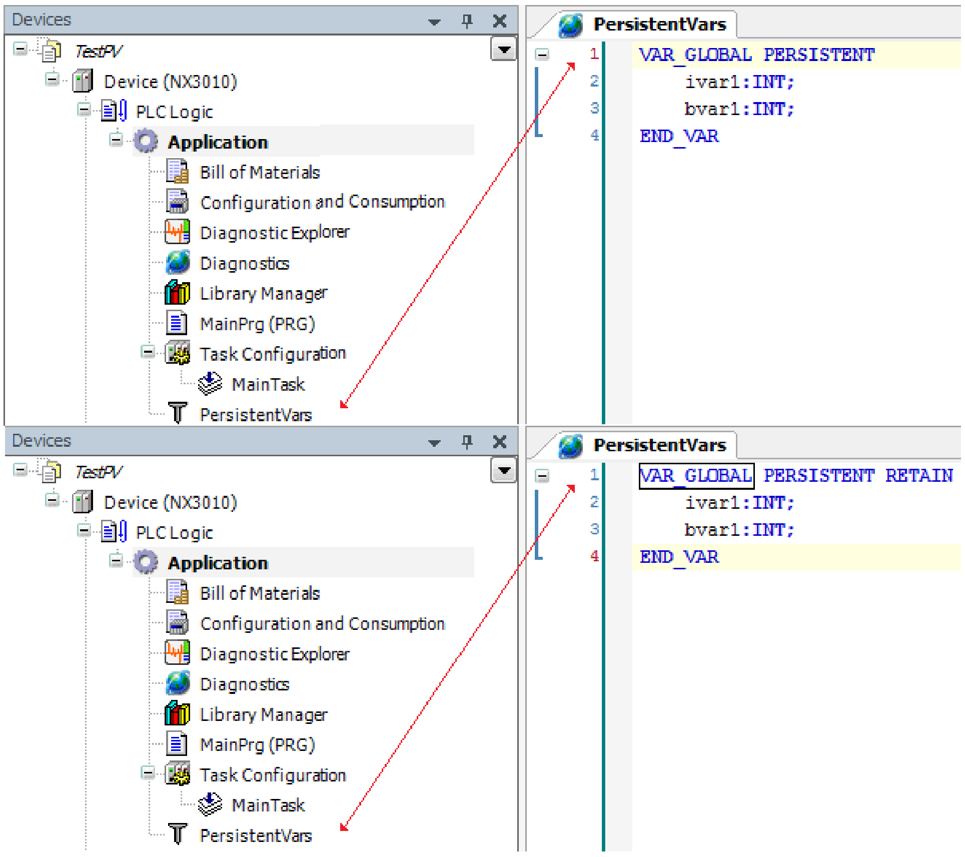
Regard that variables declared in GVLs always get initialized before local variables of POUs.

Persistent Variables

This object is a global variables list, which however only contains persistent variables of an application. Thus it must be assigned to an application and for this purpose be inserted in the device tree via the *Add Object* dialog below this application.

Persistent variables only get re-initialized at a *Reset Origin*. For further information please see the **Remanent Variables**.

A persistent variables list is edited in the GVL Editor, whereby VAR_GLOBAL PERSISTENT RETAIN already is preset in the first line.



```
VAR_GLOBAL PERSISTENT
    ivar1:INT;
    bvar1:INT;
END_VAR
```

```
VAR_GLOBAL PERSISTENT RETAIN
    ivar1:INT;
    bvar1:INT;
END_VAR
```

Figure 2-5. Persistent Variable List

External File

Any external file can be added to the POUs view of a project via the *Add Object* command. In the *Add object* dialog choose object type *External File* ( ).

Press button ... for getting the dialog for browsing for a file, the path of which will be entered to the field below *File path*. In the field below *Name* automatically the name of the chosen file will be entered without extension. You can edit this field to define another name for the file under which it should be handled within the project.

Select one of the following options.

- *Remember the link*: The file will be available in the project only if it is available in the defined link path
- *Remember the link and embed into project*: A copy of the file will be stored internally in the project but also the link to the external file will be remembered. If the external file is linked to the project, you can additionally select one of the options
  - Reload the file automatically: The file will be updated within the project as soon as it has been changed externally
  - Prompt whether to reload the file: A dialog will pop up as soon as the file has been changed externally. You can decide whether the file should be updated also within the project
  - Do nothing: The file will remain unchanged within the project, even when it is changed externally
- *Embed into project*: Just a copy of the file will be stored in the project. There will be no further connection to the external file

On this dialog have the button *Display file properties...*, it opens the standard dialog for the properties of a file, which also appears when you select the file object in the POUs window and use command

Properties. The dialog i.e. contains a tab *External file* where the properties which have been set in the *Add Object* dialog, can be viewed and modified.

POUs for Implicit Checks

Below an application you might add special POUs, which must be available there, if the implicitly provided check functionality for array and range boundaries, divisions by zero and pointers during runtime should be used.

For this purpose the *Add Object* menu in category *POUs for Implicit Checks* () presents the following functions:

- CheckBounds
- CheckDivDInt
- CheckDivLInt
- CheckDivReal
- CheckDivLReal
- CheckRangeSigned
- CheckRangeUnsigned
- CheckPointer

After having inserted a check POU, it will be opened in the editor corresponding to the implementation language selected. A default implementation that might be adapted to your requirements is available in the ST editor.

After having inserted a certain check POU, the option will not be available in the dialog any longer, thus avoiding a double insertion. If all types of check POUs have already been added below the application, the *Add Object* dialog at all will not provide the *POUs for Implicit Checks* category any longer.

NOTE: Do not modify the declaration part of an implicit check function, in order to maintain the check functionality.

Library Management

Libraries can provide functions and function blocks as well as data types, global variables and even visualizations which can be used in the project just like the other POUs and variables which are defined directly within the project.

The default extension for a library file is .library.

The management of the libraries in a project is done in the *Library Manager* (), the preceding installation on the system in the *Library Repository* dialog.

NOTE: The *Library Repository* dialog is only available if predefined feature sets chosen by the user are *Professional* or the option *Enable repository dialog* is enabled. For further information about features, see **Features** in the MasterTool IEC XE User Manual - MU299609.

The project functions for local and global search and replace also work for included libraries.

See general information on:

- Installation and Including in project
- Referenced Libraries
- Library Versions
- Unique access to library modules or variables
- Creating libraries, Encoding, Documentation MasterTool IEC XE
- Internal and external library modules

Installing and Including on Project

- Libraries can be managed on the local system in one or various repositories (folders, locations). A library cannot be included in a project before it has got installed in a repository on the local system. The installation is to be done in the *Library Repository* dialog
- As a precondition for installation, a library must have got assigned a title, a version info and a company name in its *Project Information*. Optionally a category can be defined which might serve later for sorting in the *Library Manager*
- If no category assignment is defined in the *Project Information*, the library automatically will belong to category *Miscellaneous*. Further categories might be defined in one or several xml-files *.libcat.xml which can be loaded in the *Project Information* dialog in order to select one of the categories. Please see also below, **Creating Libraries, Encoding, Documentation**
- The *Library Manager* is used to include libraries in a project. In a *MasterTool Standard Project* a *Library Manager* object by default is automatically assigned to the default device. However a *Library Manager* object can also be added explicitly in the *Devices* or *POUs* view window. This is to be done like for other objects with the Add Object dialog. Libraries referenced in other libraries by default are also displayed in the *Library Manager*, however also hidden libraries are possible, see below, **Referenced Libraries**
- If the .library* file is available (and not only its compiled version *.compiled-library), the POUs of the related library may be opened by a double click on their referencing within the library manager
- If a library module is called by an application, all libraries and repositories will be searched in that order which is defined in the *Library Repository* dialog. See below for unique accessing

Referenced Libraries

- Libraries can include further libraries (referenced libraries), whereby the nesting can have any desired depth. When you add a library including other libraries in the library manager, the referenced libraries will be added automatically too
- When creating a library project which includes other libraries, in the *Properties* of each referenced library it can be defined, how it should behave later when getting inserted in a project with its father library
- Its pure visibility in the *Library Manager*, indented below the father library, can be deactivated. Thus hidden libraries can be available in a project
- If you are just creating a pure container library, that is a library projects which is not defining own modules, but only is referencing other libraries, the access on the modules of these libraries can be simplified. A container library is created because you want to include a complete set of libraries in a project at the same time by just including the container library. In this case it might be desired to simplify the access on the modules of these libraries by making them top-level libraries, which allows leaving out the namespace of the container library in the access path. This can be reached by activating option *Publish...*. However this option should really only be activated when creating a container library and should be handled with great caution

Library Versions

- Multiple versions of the same library can be installed on the system
- Multiple versions of the same library can be included in a project. It is regulated like described in the following, which version of a library an application will use:
  - If multiple versions are available on the same level within the same manager, it will depend on the current library properties, which version will be accessed (a defined one or always the newest)
  - If multiple versions of the same library are available on different levels within the same manager (which is the case for referenced libraries), unique access to the library modules or variables is reached by adding the appropriate namespace (see the following paragraph)

Unique Access to Library Modules or Variables

- Basically, if there are several modules or variables with the same name within a project, the access on a module component must be unique, otherwise compile errors will be detected. This applies for local project modules or variables as well as those which are available in included libraries and in libraries referenced by the included ones. The uniqueness is reached in such cases by prefixing the module name by the appropriate library namespace
- The default namespace of a library is defined in the library properties. If it is not defined explicitly, it will equate with the library name. However, when creating a library project, also a different standard namespace can be specified in the *Properties* dialog. Later, when the library has got included in a project, the namespace always can be modified locally by the user, again in the *Properties* dialog

Examples: Assume that for the following examples the namespace of library Lib1 is specified in the library properties to be Lib1. See in the right column the use of the namespaces for unique access on variable var 1 which is defined in the modules module1 and POU1.

	Variable var1 is available at the following locations	**Access on var1 via using the appropriate namespace path**
1	In library Lib1 in the global Library Manager in the POUs window.	Lib1.module1.var1
2	In library Lib1 in the Library Manager below Application App1 of Device Dev1 in the Devices window.	Dev1.App1.Lib1.module1.var1
3	In library Lib1 which is included in library F_Lib in the global Library Manager in the POUs window.	By default (option Publish... in the library Properties is deactivated): F_Lib.Lib1.module1.var1 If option Publish... was activated, module1 would be treated like a component of a top-level library. Thus accessing would be possible by Lib1.module1.var1 or module1.var1. In the current example this however would cause compiler errors because the access path is not unique, see (1) and (4).
4	In object module1 which is defined in the POUs window.	module1.var1
5	In object POU1which is defined in the POUs window.	POUxy.var1

Table 2-1. Unique Access

Creating Libraries, Encoding, Documentation

A MasterTool IEC XE project can be saved as library <project name>.library) and optionally can be installed at the same time in the System Library Repository. Only the objects managed in the *POUs* window will be regarded for a library project. If you pointedly are going to create a library project, it is recommended to choose template *Empty library* in the *New Project* dialog. Notice further on the following:

- In the *Project Information* a title and a version and the company must be specified. If the Default namespace should be another than the library name, this can be defined immediately here. Additionally it is recommended to specify a category, because these will later serve for sorting the entries in the *Library Repository* and *Library Manager* dialogs. If the library should belong to another category than the default (Miscellaneous), an appropriate category description must be loaded, either from a XML-file *.libcat.xml or from another library, which already contains the information of such a description file. If necessary, a new category description file must be issued or an existing one must be modified. The information of the selected categories and the basic category description file will be transferred to the local library project and later, when installing the library - to the *Library Repository*. So the categories will be known in the repository. If afterwards another library again brings a description file with the same ID but different content, then the information of the new file will be valid in the repository

- If the library includes further libraries, you should consider, how those referenced libraries should behave later, when the other library will be included in a project. This concerns version handling, namespace, visibility and access properties, which can be configured in the properties dialog of the particular, referenced library. If the library later, when it gets included in a project, always should reference another, device-specific library, a placeholder can be used when configuring the reference. If the library modules should be protected against viewing and accessing, a library project can be saved in encoded in encoded format (<project name>.compiled-library)
- Data structures of a library can be marked as lib .internal. These non-public objects carry the attribute 'hide' and therefore do not appear within the library manager, the *List Components* functionality or the *Input Assistant*
- In order to provide the user with information on a library module in an easy way, an appropriate comment can be added to the declaration of a module parameter. This comment will be displayed later, when the library is included in a project, on the *Documentation* tab of the *Library Manager*.
- The following commands are available by default in the *File* menu for saving a library project:
    - *Save Project As...*
    - *Save Project As Compiled Library*
    - *Save Project And Install Into Library Repository*

3. Menu Commands

This manual treat only menu commands that make the library management, to see the commands from other menus see **Menu Commands** in the MasterTool IEC XE User Manual - MU299609.

Library Manager

This category provides the *Library Manager* editor commands for the management of the libraries used in the project.

Library Manager Commands

Provides the commands listed below. They are part of the *Libraries* menu when the *Library Manager* is active.

Available commands:

- Add Library...
- Properties...
- Try to Reload the Library

For general information on the library management in MasterTool IEC XE please see **Library Manager**.

Add Library

This command is part of the *Libraries* menu and the *Library Manager* editor window.

Activate the command if you want to include libraries via the currently opened *Library Manager* into the project. Only libraries which are already installed on your system can be added. Multiple versions of a library can be included in the project at the same time, within one or multiple library managers.

The command opens the *Add Library* dialog with its tabs *Library* and *Placeholder* (*Placeholder* feature is only available with the option *Enable repository dialog* enabled, for further information see the item **Features** in the MasterTool IEC XE User Manual - MU299609).

Sub dialog Library

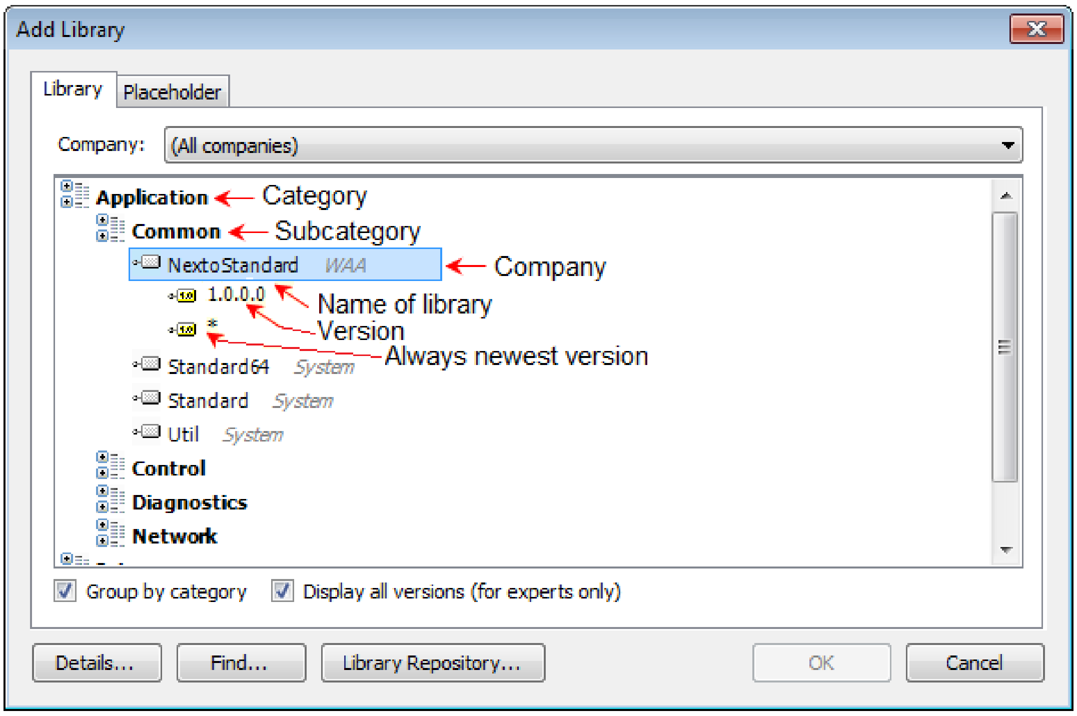


Figure 3-1. Add Library Dialog (All Versions)

Here all libraries currently installed on your system will be listed. You can filter the display by setting a certain providing *Company* from the selection list. *All companies* will list all available libraries.

If option *Group by Category* is activated, the libraries of the currently set company will be listed according to the available categories, otherwise alphabetically. If the grouping by categories is activated, the categories appear as nodes and for the currently selected category the libraries (or further categories) will be displayed indented below.

Choose the desired library. If option *Display all versions (for experts only)* is activated, all installed version(s) of the libraries appear(s) indented below the currently selected library entry.

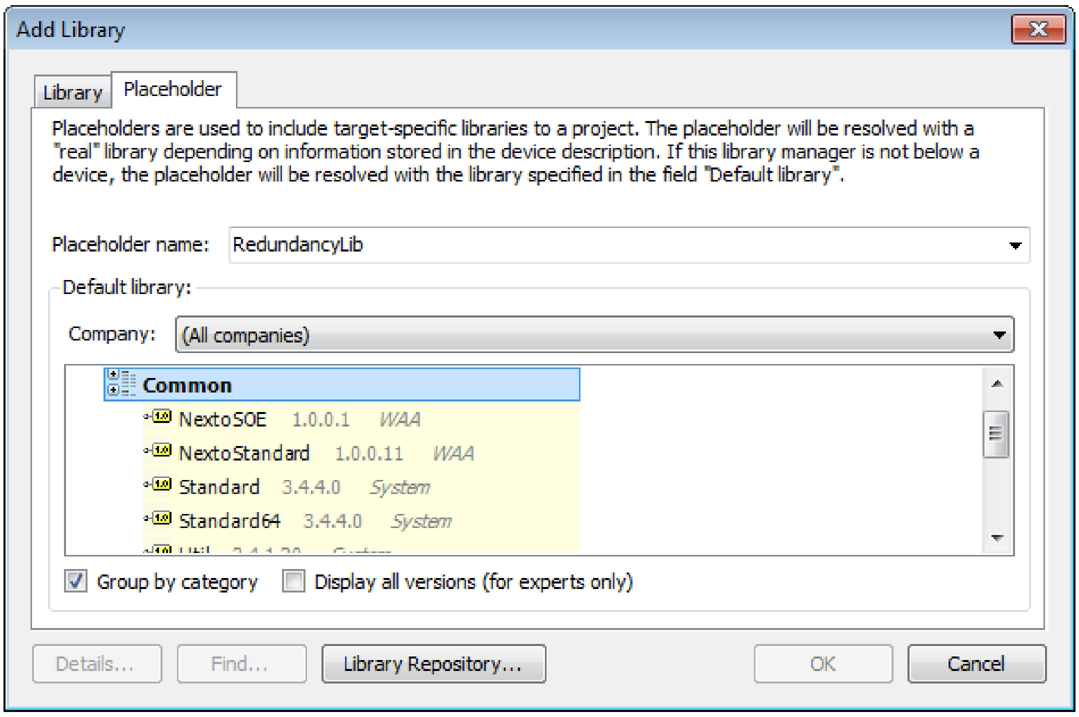
Additionally to the explicit version identifiers each a "*" is available, which means latest version. In this case you can choose among the versions. By default however this option is deactivated and only the latest version will be displayed. In this case a multiselection of libraries is possible: Keep the <SHIFT> key pressed while selecting the desired libraries.

After having confirmed with *OK*, the selected libraries will be added to the list in the *Library Manager* window.

If you want to include a library which is not yet installed on the local system, you might use the *Library Repository* button to get to the *Library Repository* dialog for doing the required installation.

NOTE: The *Library Repository* dialog is only available if predefined feature sets chosen by the user are *Professional* or the option *Enable repository dialog* is enabled. For further information about features, see **Features** in the MasterTool IEC XE User Manual - MU299609.

Sub dialog Placeholder



Placeholders are used to include target-specific libraries to a project. The placeholder will be resolved with a "real" library depending on information stored in the device description. If this library manager is not below a device, the placeholder will be resolved with the library specified in the field "Default library".

Placeholder name: RedundancyLib

Default library:

Company: (All companies)

Common		
NextoSOE	1.0.0.1	WAA
NextoStandard	1.0.0.11	WAA
Standard	3.4.4.0	System
Standard64	3.4.4.0	System

[✓] Group by category [ ] Display all versions (for experts only)

Figure 3-2. Add Library, Placeholder

NOTE: *Placeholder* is only available if predefined feature sets chosen by the user are *Professional* or the option *Enable repository dialog* is enabled. For further information about features, see **Features** in the MasterTool IEC XE User Manual - MU299609.

The tab *Placeholder* is dedicated to the following two use cases:

- Creation of a target independent project
- Creation of a library project <library_xy>, which references another library that is meant to be target-specific, that is device-specific

Placeholder Within Project

If a project shall be designed as compatible for multiple interchangeable target devices, the target specific libraries have to be included within the library manager of the project via placeholders.

As soon as the target device gets specified, the placeholders will be cast according to the related device description. Even if no device description is actually available, the placeholders allow the project to pass a syntactical check.

To include a library into the library manager via a placeholder, the library has to be selected within the bottom part *Default library* of sub dialog *Placeholder*. Thereby it is possible to constrict the catalogue of proposed libraries according to the providing company.

In addition the placeholder name has to be inserted in the associated edit field. To ensure correct insertion of the name you might use the selection list offering all placeholder names currently defined in device descriptions.

Placeholder Within Library Project

If the library project is based on further libraries that are target-specific, that is device-specific, these libraries have to be included in the library project via placeholders.

This means that instead of specifying one particular library for being included, a placeholder will be inserted, which will be replaced later, when <library_xy> is used in another project for a certain device, by the name of a device-specifically defined library. This name must be specified in the respective device description file, <library_xy> which assigns the placeholder name to a real library name.

If the library manager for any reason currently is not assigned to a device, the placeholder will be replaced by the default library specified here in the dialog. (This for example allows compilation of the currently edited library project without detected errors even if no suitable device description is available at that moment.)

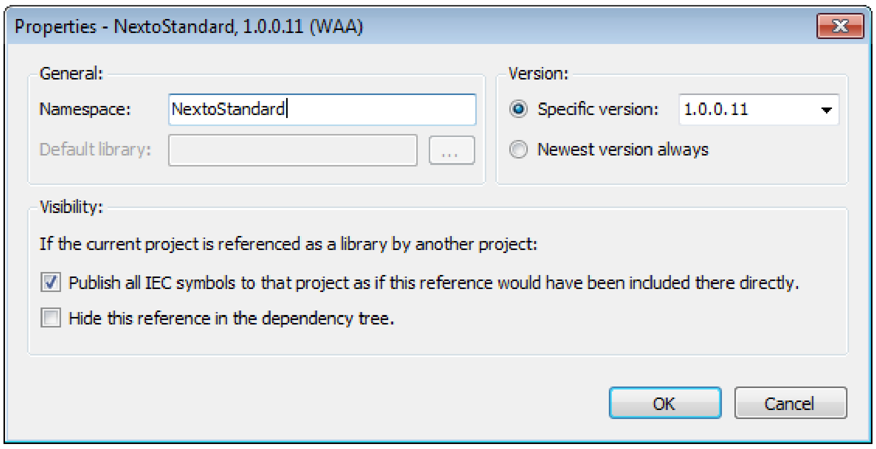
In the *Placeholder Name* field enter any string as a name for the placeholder. Further on choose a default library from the currently installed libraries. This is to be done like described above for adding a library in the *Library* sub dialog and like there option *Display all versions (for experts only)* might be activated to get displayed all currently installed versions of a library.

After closing the dialog with *OK*, the placeholder library will be entered in the library manager tree. When you open the *Properties* dialog for the library placeholder, you get information on the currently set default library.

Properties

This command is available in the *Library Manager*editor window.

It opens the *Properties* dialog for the library which is currently selected in the *Library Manager* window and allows the following settings concerning namespace, version handling, availability and visibility of library references.


Properties - NextoStandard, 1.0.0.11 (WAA)

General:
Namespace: NextoStandard
Default library:

Version:
[✓ Specific version: 1.0.0.11]
Newest version always

Visibility:
If the current project is referenced as a library by another project:
[✓] Publish all IEC symbols to that project as if this reference would have been included there directly.
[ ] Hide this reference in the dependency tree.

Figure 3-3. Properties Dialog for Library

- **Namespace:** The current namespace of the library is displayed. By default the namespace of a library primarily is identic with the library name, except another string has been defined explicitly in the *Project Information* when creating the library project. You can edit the namespace anytime here in the properties dialog. For further information on the namespace of libraries see the item editor window, see the **Library Manager Editor** in the MasterTool IEC XE User Manual – MU299609
- **Default library:** (only available if option *Enable repository dialog* is enabled, see **Features** in the MasterTool IEC XE User Manual - MU299609) If a library placeholder is currently selected in the *Library Manager*, this field will contain the name of the library which should replace the placeholder, if no device-specific library is available. Please see the correspondent item for information on library placeholder
- **Version:** (only available if option *Enable repository dialog* is enabled, see **Features** in the MasterTool IEC XE User Manual - MU299609) Here you can configure, which version of the library should be used in the project:

- **Specific version:** Exactly the version entered here (you can select from the list) will be used
- **Newest version always:** Always the newest version found in the library repository will be used, that is the modules actually used might change because a newer version of the library is available

- **Visibility:** (only available if option *Enable repository dialog* is enabled, see **Features** in the MasterTool IEC XE User Manual - MU299609) These settings are of interest as soon as the library gets included that is referenced by another library. Per default they are deactivated
  - **Publish all IEC symbols to that project as if this reference would have been included there directly:** As long as this option is deactivated, the components of the current library - if referenced by another one - can be accessed uniquely by using the appropriate namespace path (composed of the namespaces of the "father" library and its own namespace in addition to the modules and variable identifier)
  - **Hide this reference in the dependency tree:** If this option gets activated, the current library will not be displayed later, when its father library is included in a project. This allows to include hidden libraries, however needs careful use, because in case of library error messages you might have problems to find the causing library

NOTE: The option *Publish all IEC symbols to that project as if this reference would have been included there directly* should only be activated, if you want to use "container libraries", not containing own modules, but just including other libraries for the purpose of "packaging" them. This packaging for example allows to include multiple libraries in a project at once just by including the "container library". In this case however it might be desired to get the particular libraries on top-level of the projects' Library Manager, because then the modules can be accessed directly, that is the namespace of the container library can be left out.

Try to Reload the Library

This command is part of the *Libraries* menu and the *Library Manager* editor window.

If a library included in a project is for any reason not available at the defined path when opening the project in the programming system, an appropriate message will be generated. After having checked the error and correctly made available the library again, use the *Try to Reload Library* command when the library entry is selected in the *Library Manager*. Thus the library can be reloaded without the necessity of leaving the project.

4. Programming Reference

Declaration

The variables of a project are to be declared manually in the declaration editor or via the *Auto Declare* dialog. See the further related help pages referring for example on the various categories of variables (local, global, input, output etc.), the initialization, the use of pragmas, init method etc.

Variables Declaration

The declaration of a variable can be done in the declaration part of a POU or via the *Auto Declare* dialog, as well as in a DUT or GVL editor.

The sort (in the declaration dialog it is named scope) of the variable(s) to be declared is specified by the keywords embracing the declaration of one or several variables. The common variable declaration for example is embraced by VAR and END_VAR.

```
VAR_INPUT
VAR_OUTPUT
VAR_IN_OUT
VAR_GLOBAL
VAR_TEMP
VAR_STAT
VAR_EXTERNAL
VAR_CONFIG
```

The variable type keywords may be supplemented by attribute keywords, like for example RETAIN (VAR_INPUT RETAIN).

The declaration of a variable must match the following rules:

Syntax**:**

```
<IDENTIFIER> {AT <ADDRESS>}:<TYPE> {:=<INITIALIZATION>};
```

The parts in braces ({}) are optional.

The identifier is the name of a variable. The items listed in the following in each case must be regarded when defining an identifier, but please also consider to follow some recommendations which are given in **Recommendations on the Naming of the Identifiers**.

- It must not contain spaces or special characters
- It is not case-sensitive, which means that for example VAR1, Var1 and var1 are all the same variable
- The underscore character is recognized in identifiers (for example, A_BCD and AB_CD are considered two different identifiers), but an identifier must not have more than one underscore character in a row
- The length of the identifier as well as the meaningful part of it, are unlimited
- The rules listed in the following text box concerning multiple uses must be regarded

Multiple Use of Identifiers (Namespaces)

An identifier must not be used duplicate locally and must not be identical to any keyword.

Globally an identifier can be used multiple times, thus a local variable can have the same name as a global one. Within a POU in this case the local variable will have priority.

A variable defined in a Global Variables List can have the same name as a variable defined in another Global Variables List. In this context notice the following IEC 61131-3 extending features:

- Global scope operator: An instance path starting with "." opens a global scope. So, if there is a local variable, for example, ivar, with the same name as a global variable, .ivar refers to the global variable
- The name of a global variables list can be used as a namespace for the included variables. So variables can be declared with the same name in different global variable lists and can be accessed specifically by preceding the variable name with the list name. Example:

```
GLOBLIST1.IVAR := GLOBLIST2.IVAR; (*IVAR from GLOBLIST2 is copied to IVAR
in GLOBLIST1 *)
```

- Variables defined in a Global Variables List of an included library can be accessed according to syntax <Library Namespace>.<Name of Global Variables List>.<Variable>. See below for namespaces of libraries. Example:

```
GLOBLIST1.IVAR := LIB1.GLOBLIST1.IVAR (*IVAR from GLOBLIST1 in library
LIB1 is copied to IVAR in GLOBLIST1 *)
```

For a library also a namespace is defined, when it gets included via the *Library Manager*. So you can access a library module or variable by <Library Namespace>.<Modulename | Variablename>. Notice that in case of nested libraries the namespaces of all libraries concerned have to been stated successively. Example: If Lib1 was referenced by Lib0, the module fun being part of Lib1 is accessed by Lib0.Lib1.fun:

```
IVAR := LIB0.LIB1.FUN(4, 5); (* return value of FUN is copied to variable
IVAR in project *)
```

As far as the checkbox *Publish all IEC Symbols to that project as if this reference would have been included there directly* has been activated within the properties of the referenced library Lib, the module fun may also be accessed directly via Lib0.fun.

AT <Address>

The variable can directly be linked to a definite address using the keyword AT.

In function blocks you can also specify variables with incomplete address statements. In order that such a variable can be used in a local instance, there must be an entry for it in the variable configuration.

Type

Valid data type, optionally extended by a ":=<initialization>".

Optionally pragma instructions can be added in the declaration part of an object, in order to affect the code generation for various purposes.

NOTE: Pay attention to the possibility of an automatic declaration. For faster input of the declarations, use the shortcut mode.

Recommendations on the Naming of the Identifiers

Identifiers are defined at the declaration of variables (variable names), user-defined data types and at the creation of POUs (functions, function blocks, programs). In addition to the items to be regarded anyway when defining an identifier you might consider to follow some recommendations in order to make the naming as unique as possible.

Variable Names

The naming of variables in applications and libraries as far as possible should follow the Hungarian notation.

For each variable a meaningful, short description should be found, the base name.

The first letter of each word of a base name should be a capital letter, the others should be small ones (Example: FileSize).

Before the base name, corresponding to the data type of the variable, prefix(es) are added in small letters.

See in the following table some information and the recommended prefixes on the particular data types:

Data type	Lower limit	Upper limit	Information content	Prefix	Comment
BIT	0	1	1 Bit	b	
BOOL	FALSE	TRUE	1 Bit	x	
BYTE			8 Bits	by	Bit string, not for arithm. operations
WORD			16 Bits	w	Bit string, not for arithm. operations
DWORD			32 Bits	dw	Bit string, not for arithm. operations
LWORD			64 Bits	lw	Not for arithm. operations
SINT	-128	127	8 Bits	si	
USINT	0	255	8 Bits	usi	
INT	-32.768	32.767	16 Bits	i	
UINT	0	65.535	16 Bits	ui	
DINT	-2.147.483.648	2.147.483.647	32 Bits	di	
UDINT	0	4.294.967.295	32 Bits	udi	
LINT	-2^{63}	$2^{63} - 1$	64 Bits	li	
ULINT	0	$2^{64} - 1$	64 Bits	uli	
REAL			32 Bits	r	
LREAL			64 Bits	lr	
STRING				s	
TIME				tim	
TIME_OF_DAY				tod	
DATE_AND_TIME				dt	
DATE				date	
ENUM			16 Bits	e	
POINTER				p	
ARRAY				a	

Table 4-1. Data Types

Note:

x: Pointedly for Boolean variables x is chosen as prefix, in order to differentiate from BYTE and also in order to accommodate the perception of an IEC-programmer (see addressing "%IX0.0").

Examples:

```
SubIndex: BYTE;
sFileName: STRING;
udiCounter: UDINT;
```

In nested declarations the prefixes are attached to each other in the order of the declarations:

Example:

```
pabyTelegramData: POINTER TO ARRAY [0..7] OF BYTE;
```

Function block instances and variables of user-defined data types as a prefix get a shortcut for the FB- and data type name (for example: sdo).

Example:

```
cansdoReceivedTelegram: CAN_SDOTelegram;
```

```
TYPE CAN_SDOTelegram : (* Prefix: sdo *)
STRUCT
wIndex:WORD;
bySubIndex:BYTE;
byLen:BYTE;
aby: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE
```

Local constants (c) start with prefix “c” and an attached underscore, followed by the type prefix and the variable name.

Example:

```
VAR CONSTANT
c_uiSyncID: UINT := 16#80;
END_VAR
```

For global variables (g) and global constants (gc) an additional prefix + underscore are attached to the library prefix:

Examples:

```
VAR_GLOBAL
CAN_g_iTest: INT;
END_VAR
VAR_GLOBAL CONSTANT
CAN_gc_dwExample: DWORD;
END_VAR
```

Variable Names in MasterTool IEC XE Libraries

Basically see above **Variable Names**, with the following exception: global variables and constants do not need a library prefix, because using the namespace takes the function of a prefix.

Example:

```
g_iTest: INT; (* Declaration *)
CAN.g_iTest (* Implementation: call in an application program *)
```

User Defined Data Types (DUT)

Structure: The name of each structure data type consists of a library prefix (Example: CAN), an underscore and a preferably short expressive description (for example: SDOTelegram) of the structure. The associated prefix for used variables of this structure should follow directly after the colon.

Example:

```
TYPE COM_SDOTelegram : (* Prefix: sdo *)
STRUCT
wIndex:WORD;
bySubIndex:BYTE;
byLen:BYTE;
abyData: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE
```

Enumerations start with the library prefix (Example: CAL), followed by an underscore and the identifier in capital letters.

ENUMs should be defined with correct INT values.

Example:

```
TYPE CAL_Day :(
CAL_MONDAY,
CAL_TUESDAY,
CAL_WEDNESDAY,
CAL_THIRSDAY,
CAL_FRIDAY,
CAL_SATURDAY,
CAL_SUNDAY);
```

Declaration:

```
eToday: CAL_Day;
```

User Defined Data Types (DUTs) in MasterTool IEC XE Libraries

DUT names in MasterTool IEC XE libraries do not get a library prefix, because using the namespace takes the function of a prefix. Also enumeration components are defined without library prefixes.

Example (in library with namespace CAL):

```
TYPE Day :(
MONDAY,
TUESDAY,
WEDNESDAY,
THIRSDAY,
FRIDAY,
SATURDAY,
SUNDAY);
```

Declaration:

```
eToday: CAL.Day;
```

Use in application:

```
IF eToday = CAL.Day.MONDAY THEN
```

Functions, Function blocks, Programs (POU), Actions

The names of functions, function blocks and programs consist of the library prefix (Example: CAN), an underscore and an expressive short name (e.g.: SENDTELEGRAM) of the POU. Like with variables always the first letter of a word of the POU name should be a capital letter, the others should be small letters. It is recommended to compose the name of the POU of a verb and a substantive.

```
FUNCTION_BLOCK Com_SendTelegram (* Prefix: comst *)
```

In the declaration part a short description of the POU should be provided as a comment. Further on all inputs and outputs should be provided with comments. In case of function blocks the associated prefix for set-up instances should follow directly after the name.

Actions get no prefix; just actions which should be called only internally, that is by the POU itself, start with prv_.

POUs in MasterTool IEC XE Libraries

POU names in MasterTool IEC XE libraries do not get a library prefix, because using the namespace takes the function of a prefix. For creating method names the same rules apply as for actions. Possible inputs of a method should get English comments. Also a short description of a method should be added in its declaration.

Visualization Names

Currently just avoid to name visualization like another object in the project, because this would cause problems in case of visualization changes.

Variables Initialization

The default-initialization value is 0 for all declarations, but user defined initialization values can be added in the declaration of each variable and data type.

The user defined initialization is brought about by the assignment operator ":=" and can be any valid ST expression. Thus constant values as well as other variables or functions can be used to define the init value. The programmer just has to make sure that a variable used for the initialization of another variable is already initialized itself.

Examples for valid variable initializations:

```
VAR
   var1:INT := 12;              (* Integer variable with initial value of
   12 *)
   x : INT := 13 + 8;           (* Init value defined by an expression
   with con constants *)
   y : INT := x + fun(4);       (* Init value defined by an expression
   containing a function call; be aware of the order in this case! *)
   z : POINTER TO INT := ADR(y); (* Not described by the IEC61131-3: Init
   value defined by an address function; Be careful in this case: the
   pointer will not be initialized during online change! *)
END_VAR
```

See: **ARRAYS**, **Structures**, **Subrange Types** and **Arbitrary Expressions For Variable Initialization**.

NOTE: Variables of global variables lists are always initialized before local variables of a POU.

Arbitrary Expressions For Variable Initialization

A variable can be initialized with any valid ST expression. It is possible to access other variables out of the same scope, and it is possible to call functions. However the programmer has to assure that a variable used for initialization of another variable is already initialized.

Examples for valid variable initializations:

```
VAR
x : INT := 13 + 8;
y : INT := x + fun(4);
z : POINTER TO INT := ADR(y); (* Be careful: the pointer will not be
initialized during online change! *)
END_VAR
```

Declaration Editor

The declaration editor is a text or tabular editor used for the declaration of variables. Usually it is provided in combination with the language editors.

For detailed information see the **Declaration Editor** chapter in the MasterTool IEC XE User Manual – MU299609.

Autodeclaration Dialog

It can be defined in the *Options* dialog, category *Text Editor*, that an *Auto Declare* dialog should open as soon as a not yet declared string is entered in the implementation part of an editor and the <ENTER> key is pressed.

This dialog will support the declaration of the variable. The dialog also can be opened explicitly by command *Auto Declare*, which by default is available in the *Edit* menu, or by <SHIFT>+<F2>. If an already declared variable is selected before, its declaration can be edited in the dialog.

Shortcut Mode

The declaration editor as well as other text editors where declarations are done, supports the shortcut mode.

This mode is activated when you end a line of declaration with <CTRL>+<ENTER> and allows to use shortcuts instead of completely typing the declaration.

The following shortcuts are supported:

- All identifiers up to the last identifier of a line will become declaration variable identifiers
- The type of declaration is determined by the last identifier of the line. In this context, the following will apply:

Identifier	Result
B or BOOL	Results in BOOL
I or INT	Results in INT
R or REAL	Results in REAL
S or string	Results in STRING

Table 4-2. Declaration Type

- If no type has been established through these rules, automatically BOOL is the type and the last identifier will not be used as a type
- Every constant, depending on the type of declaration, will turn into an initialization or a string
- An address (as in %MD12) is extended by the AT attribute
- A text after a semicolon (;) becomes a comment
- All other characters in the line are ignored

Example:

Shortcut	Declaration
A	A: BOOL;
A B I 2	A, B: INT := 2;
ST S 2; string A	ST:STRING(2); (* String A *)
X %MD12 R 5 Real number	X AT %MD12: REAL := 5.0;(* Real number *)
B !	B: BOOL;

Table 4-3. Shortcut Examples

AT Declaration

In order to link a project variable directly with a definite address you can enter this address in the declaration of the variable. Regard that the assignment of a variable to an address also can be done in the mapping dialog of a device in the PLC configuration (device tree).

Syntax**:**

```
<Identifier> AT <Address> : <Data type>;
```

Keyword AT must be followed by a valid address. See the **Address** help page for further information and consider possible overlaps in case of byte addressing mode.

This declaration allows to assign a meaningful name to an address. Any changes concerning an incoming or outgoing signal may only be done in a single place (for example in the declaration).

Notice the following when choosing a variable to be assigned to an address:

- Variables requiring an input cannot be accessed by writing. The compiler intercepts this with error

- AT declarations only can be used with local or global variables, not however with input and output variables of POUs
- AT declarations must not be used in combination with VAR RETAIN or VAR PERSISTENT
- If AT declarations are used with structure or function block members, all instances will access the same memory location, which corresponds to static variables in classic programming (languages like e.g. C)
- The memory layout of structures is determined by the target too

Examples:

```
counter_heat7 AT %QX0.0: BOOL;
lightcabinetimpulse AT %IX7.2: BOOL;
download AT %MX2.2: BOOL;
```

NOTE: If Boolean variables are assigned to a Byte, Word or DWORD address, they occupy one byte with TRUE or FALSE, not just the first bit after the offset.

Keywords

Keywords are to be written in uppercase letters in all editors.

The following strings are reserved as keywords, i.e. they cannot be used as identifiers for variables or POUs:

ABS, ACOS, ACTION (only used in the Export Format), ADD, ADR, AND, ARRAY, ASIN, AT, ATAN, BITADR, BOOL, BY, BYTE, CAL, CALC, CALCN, CASE, CONSTANT, COS, DATE, DINT, DIV, DO, DT, DWORD, ELSE, ELSIF, END_ACTION (only used in the Export Format), END_CASE, END_FOR, END_FUNCTION (only used in the Export Format), END_FUNCTION_BLOCK (only used in the Export Format), END_IF, END_PROGRAM (only used in the Export Format), END_REPEAT, END_STRUCT, END_TYPE, END_VAR, END_WHILE, EQ, EXIT, EXP, EXPT, FALSE, FOR, FUNCTION, FUNCTION_BLOCK, GE, GT, IF, INDEXOF, INT, JMP, JMPC, JMPCN, LD, LDN, LE, LINT, LN, LOG, LREAL, LT, LTIME, LWORD, MAX, MIN, MOD, MOVE, MUL, MUX, NE, NOT, OF, OR, PARAMS, PERSISTENT, POINTER, PROGRAM, R, REFERENCE, READ_ONLY, READ_WRITE, REAL, REPEAT, RET, RETAIN, RETC, RETCN, RETURN, ROL, ROR, S, SEL, SHL, SHR, SIN, SINT, SIZEOF, SUPER, SQRT, ST, STN, STRING, STRUCT, SUPER, SUB, TAN, THEN, TIME, TO, TOD, TRUE, TRUNC, TYPE, UDINT, UINT, ULINT, UNTIL, USINT, VAR, VAR_ACCESS, (only used very specifically, depending on the hardware), VAR_CONFIG, VAR_EXTERNAL, VAR_GLOBAL, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT, VAR_STAT, VAR_TEMP, WHILE, WORD, WSTRING and XOR.

Additionally all conversion operators as listed in the *Input Assistant* are handled as keywords.

Local Variables VAR

Between the keywords VAR and END_VAR all local variables of a POU are declared. These have no external connection; in other words, they cannot be written from the outside.

Regard the possibility of adding an attribute to VAR.

Example:

```
VAR
iLoc1:INT; (* 1. Local variable *)
END_VAR
```

Input Variables - VAR_INPUT

Between the key words VAR_INPUT and END_VAR all variables are declared that serve as input variables for a POU. That means that at the call position, the value of the variables can be given along with a call.

Regard the possibility of adding an attribute to VAR_INPUT.

```
VAR_INPUT
iIn1:INT (* 1. Input variable *)
END_VAR
```

Output Variables - VAR_OUTPUT

Between the key words VAR_OUTPUT and END_VAR all variables are declared that serve as output variables of a POU. That means that these values are carried back to the POU making the call. There they can be answered and used further.

Regard the possibility of adding an attribute to VAR_OUTPUT.

Example:

```
VAR_OUTPUT
iOut1:INT; (* 1. Output variable *)
END_VAR
```

Output Variables in Functions and Methods

According to IEC 61131-3 draft 2, functions (and methods) can have additional outputs. Those must be assigned in the call of the function like this:

```
fun(iIn1 := 1, iIn2 := 2, iOut1 => iLoc1, iOut2 => iLoc2);
```

The return value of the function fun will be written to the e.g. locally declared variables loc1 and loc2.

Input and Output Variables - VAR_IN_OUT

Between the key words VAR_IN_OUT and END_VAR all variables are declared that serve as input and output variables for a POU.

NOTE: With variables of this type the value of the transferred variable is changed (transferred as a pointer, call-by-reference). That means that the input value for such variables cannot be a constant. For this reason, even the VAR_IN_OUT variables of a function block cannot be read or written directly from outside via <function block instance><in/output variable>.

Example:

```
VAR_IN_OUT
iInOut1:INT; (* 1. Input and output variable *)
END_VAR
```

Global Variables - VAR_GLOBAL

Normal variables, constants, external or remanent variables that are known throughout the project can be declared as global variables.

NOTES:
A variable defined locally in a POU with the same name as a global variable will have priority within the POU. Global variables always will be initialized before local variables of POUs.

The variables have to be declared locally between the keywords VAR_GLOBAL and END_VAR. Regard the possibility of adding an attribute to VAR_GLOBAL.

A variable is recognized as a global variable by a preceding dot, for example .iGlobVar1.

For detailed information on multiple uses of variable names, the global scope operator "." and name spaces see: **Global Scope Operator**.

Use global variables lists (GVLs) to structure and handle global variables within a project. A GVL can be added via the *Add Object* command.

Temporary Variables - VAR_TEMP

This feature is an extension to the IEC 61131-3 standard.

VAR_TEMP declarations are only possible within programs and function blocks. Those variables are also only accessible within the body of the program or function block. VAR_TEMP declared variables are (re)initialized every time the POU is called.

The variables have to be declared locally between the keywords VAR_TEMP and END_VAR.

Static Variables - VAR-STAT

This feature is an extension to the IEC 61131-3 standard.

Static variables can be used in function blocks, methods and functions. They have to be declared locally between the keywords VAR_STAT and END_VAR and will be initialized at the first call of the respective POU.

Static variables are only accessible within the scope in which they are declared (like a static variable in C), but like a global variable does not lose its value after the POU is left. For example in a function they might be used as a counter for the number of function calls.

Regard the possibility of adding an attribute to VAR_STAT.

External Variables – VAR_EXTERNAL

These are global variables which are imported into the POU.

They have to be declared locally between the keywords VAR_EXTERNAL and END_VAR and in the global variable list. The declaration must exactly match the global declaration. If the global variable does not exist, an error message will appear.

NOTE: In MasterTool IEC XE it is not necessary to define variables as external. This keyword is provided in order to maintain compatibility to IEC 61131-3.

Example:

```
VAR_EXTERNAL
iVarExt1:INT; (* First external variable *)
END_VAR
```

Attribute Keywords for Variable Types

The following attribute keywords can be added to the declaration of the variables type in order to specify the scope:

- RETAIN: See **Remanent Variables** of type RETAIN
- PERSISTENT: See **Remanent Variables** of type PERSISTENT
- CONSTANT: See **Constants**

Remanent Variables

Remanent variables can retain their value throughout the usual program run period.

The declaration determines the degree of resistance of a remanent variable in the case of resets, downloads or a reboot of the PLC. In applications mainly the combination of both remanent flags will be required.

See in the following:

- **Retain Variables**

- **Persistent Variables**

Retain Variables

Variables declared as retains will be kept PLC-dependently but typically in a separate memory area. They get the keyword RETAIN in their declaration in a POU and in a global variable list.

Example:

```
VAR RETAIN
iRem1 : INT; (* 1. Retain variable *)
END_VAR
```

Retain variables are identified by the keyword RETAIN. These variables maintain their value even after an uncontrolled shutdown of the controller as well as after a normal switch off and on of the controller and at the online command *Reset warm*.

When the program is run again, the stored values will be processed further.

All other variables are newly initialized, either with their initialized values or with the standard initializations.

Contrary to persistent variables, retain variables are reinitialized at a new download of the program.

Retain variables however are re-initialized at *Reset origin* and - in contrast to persistent variables - at *Reset cold* and an application download.

The *Retain* property can be combined with the *Persistent* property.

NOTES:
- If a local variable in a program is declared as VAR RETAIN, then exactly that variable will be saved in the retain area (like a global retain variable).
- If a local variable in a function block is declared as VAR RETAIN, then the complete instance of the function block will be saved in the retain area (all data of the POU), whereby only the declared retain variable will be handled as a retain.
- If a local variable in a function is declared as VAR RETAIN, then this will be without any effect. The variable will not be saved in the retain area. If a local variable is declared as PERSISTENT in a function, then this will be without any effect also.

Persistent Variables

Persistent variables are identified by keyword PERSISTENT (VAR_GLOBAL PERSISTENT). They get only re-initialized at a *Reset origin*. In contrast to Retain variables they maintain their values after a download. An application example for persistent variables would be a counter for operating hours, which should continue counting even after a power fail or download. See below the synoptic table.

Persistent variables ONLY can be declared in a special global variables list of object type *Persistent Variables*, which is assigned to an application. There might be only ONE such list per application.

NOTE: A declaration with VAR_GLOBAL PERSISTENT effects the same as a declaration with VAR_GLOBAL PERSISTENT RETAIN or VAR_GLOBAL RETAIN PERSISTENT.

Like retain variables the persistent variables get stored in a separate memory area.

Example:

```
VAR GLOBAL PERSISTENT RETAIN
iVarPers1 : DINT; (* 1. Persistent+Retain Variable App1 *)
bVarPers : BOOL; (* 2. Persistent+Retain Variable App1 *)
END_VAR
```

NOTE: Currently only global persistent variables are possible, except that there is some mechanism that allows the PLC using this type of operation.

The target system must provide a separate memory area for the persistent variables list of each application.

At each re-load of the application the persistent variables list on the PLC will be checked against that of the project. The list on the PLC inter alia is identified by the application name. In case of inconsistencies the user will be prompted to re-initialize all persistent variables of the application. Inconsistency might result from renaming or removing or other modifications of the existing declarations in the list.

NOTE: So carefully consider any modifications in the declaration part of the persistent variables list and the effect of the resultantly requested re-initialization.

New declarations only can be added at the end of the list. At a download those will be detected as "new" and will not demand a re-initialization of the complete list.

Legend of the behavior: Value gets maintained (x) and Value gets initialized (-).

Situation	**Behavior**		
After online command	VAR	VAR RETAIN	VAR PERSISTENT VAR RETAIN PERSISTENT VAR PERSISTENT RETAIN
Reset warm	-	x	x
Reset cold	-	-	x
Reset origin	-	-	-
Download <application>	-	-	x
Online Change <application>	x	x	x
Reboot PLC	-	x	x

Table 4-4. Behavior of Remanent Variables

Constants

Constants are identified by the key word CONSTANT. They can be declared locally or globally.

Syntax:

```
VAR CONSTANT
<Identifier>:<Type> := <Initialization>;
END_VAR
```

Example:

```
VAR CONSTANT
c_iCon1:INT:=12; (* 1. Constant*)
END_VAR
```

See the **Operands** description for a listing of possible constants. See there also regarding the possibility of using typed constants.

Typed Literals

Basically, in using IEC constants, the smallest possible data type will be used. If another data type must be used, this can be achieved with the help of typed literals without the necessity of explicitly declaring the constants. For this, the constant will be provided with a prefix which determines the type.

Syntax:

```
<Type>#<Literal>
<Type> specifies the desired data type; possible entries are all simple
data types. The type must be written in uppercase letters.
```

`<Literal>` specifies the constant. The data entered must fit within the data type specified under `<Type>`.

Example:

```
iVar1:=DINT#34;
```

If the constant cannot be converted to the target type without data loss, an error message is issued:

Typed literals can be used wherever normal constants can be used.

Constants in Online Mode

The constants are indicated by a 🟢 symbol preceding the value in the value column of the *Declaration* or *Watch* window in online mode. In this case they cannot be accessed by e.g. forcing or writing.

Variables Configuration – VAR_CONFIG

The variables configuration can be used to map function block variables on the process image, that is on the device I/Os, without the need of specifying the definite address already in the declaration of the function block variable. The assignment of the definite address in this case is done centrally for all function block instances in a global VAR_CONFIG list.

For this purpose you can assign incomplete addresses to the function block variables declared between the key words VAR and END_VAR. These addresses are to be identified with an asterisk.

Syntax:

```
<Identifier> AT %<I|Q>* : <Data type>;
```

Example of the use of not completely defined addresses:

```
FUNCTION_BLOCK locio
VAR
xLocIn AT %I*: BOOL := TRUE;
xLocOut AT %Q*: BOOL;
END_VAR
```

Here two local I/O-variables are defined, a local-In (%I*) and a local-Out (%Q*).

Then the final definition of the addresses is to be done in the variables configuration in a global variables list.

For this purpose add a *Global Variable List* object (GVL) to the *Devices* window by the *Add Object* command. There enter the declarations of the instance variables with the definite addresses between the key words VAR_CONFIG and END_VAR.

The instance variables must be specified by the complete instance path through which the individual POUs and instance names are separated from one another by periods. The declaration must contain an address whose class (input/output) corresponds to that of the incompletely specified address (%I*, %Q*) in the function block. Also the data type must agree with the declaration in the function block.

Syntax:

```
<Instance variable path> AT %<I|Q><Location> : <Data type>;
```

Configuration variables, whose instance path is invalid because the instance does not exist, are denoted as errors. On the other hand, an error is also reported if no definite address configuration exists for an instance variable assigned to an incomplete address.

Example for a variable configuration:

Assume that the following definition for function block "locio" - see the example above - is given in a program.

```
PROGRAM MainPrg
VAR
```

```
locioVar1: locio;
locioVar2: locio;
END_VAR
```

Then a corrected variable configuration would look this way:

```
VAR_CONFIG
MainPrg.locioVar1.xLocIn AT %IX1.0 : BOOL;
MainPrg.locioVar1.xLocOut AT %QX0.0 : BOOL;
MainPrg.locioVar2.xLocIn AT %IX1.0 : BOOL;
MainPrg.locioVar2.xLocOut AT %QX0.3 : BOOL;
END_VAR
```

NOTE: Changes on directly mapped I/Os are immediately shown in the process image, whereas changes on variables mapped via VAR_CONFIG are not shown before the end of the responsible task.

Declaration and Initialization of User Defined Data Types

Besides the standard data types also user defined types might be used.

For information on declaration and initialization see the pages on **Data Types** and **User Defined Data Types**.

FB_Init and FB_Reinit Methods

FB_Init

The method FB_init replaces the INI operator. It is a special method for a function block which can be declared explicitly but also and in any case is available implicitly. Thus in any case it can be accessed for each function block.

The init method contains initialization code for the function block as declared in the declaration part of the function block. If the init method is declared explicitly, the implicit initialization code will be inserted into this method. The programmer can add further initialization code.

NOTE: When execution reaches the user defined initialization code, the function block is already fully initialized via the implicit initialization code.

The Init method is called after download all for each declared instance.

ATTENTION:
In online change the previous values will overwrite the initialization values.

On the call sequence in case of inheritance please see: **FB_Exit**.

Regard also the possibility of defining a function block instance method to be called automatically after initialization via FB_init: attribute call_after_init.

Interface of the Init Method

```
METHOD fb_init : BOOL
VAR_INPUT
bInitRetains : BOOL; // If TRUE, the retain variables are initialized
(warm start / cold start)
bInCopyCode : BOOL; // If TRUE, the instance afterwards gets moved into
the copy code (online change)
END_VAR
```

The return value is not interpreted.

NOTE: Notice also the possible use of an "exit" method and the resulting execution order: See: **FB_Exit**.

User Defined Input

In an init method additional inputs can be declared. Those must be assigned at the declaration of a function block instance.

Example for an init method for a function block “serialdevice”:

```
METHOD fb_init : BOOL
VAR_INPUT
bInitRetains : BOOL; (*Initialization of retains *)
bInCopyCode : BOOL; (*Instance moved into copy code *)
nCOMnum : INT; (*Additional input: number of the COM interface to listen
at *)
END_VAR
```

Example for declaration of function block “serialdevice”:

```
COM1 : serialdevice(nCOMnum:=1);
COM0 : serialdevice(nCOMnum:=0);
```

FB_Reinit

If a method named “FB_reinit” is declared for a function block instance, this method will be called when the instance gets copied (which for example is the case at an online change after a modification of the function block declaration). The method will cause a reinitialization of the new instance module that has been created by the copy code. A reinitialization might be desired because the data of the original instance will be written to the new instance after the copying, but you might want to get the original init values. The FB_reinit method has no inputs. Regard that in contrast to FB_init the method must be declared explicitly. If a reinitialization of the basic function block implementation is desired, FB_reinit must be explicitly called for that POU.

The FB_reinit method has no inputs.

On the call sequence in case of inheritance please see: **FB_Exit**.

FB_Exit

A special method for a function block is a method named FB_Exit. It must be declared explicitly, there is no implicit declaration. The exit method - if present - is called for all declared instances of the function block before a new download, at a reset or during online change for all moved or deleted instances.

There is only one mandatory parameter:

```
METHOD fb_exit : BOOL
VAR_INPUT
bInCopyCode : BOOL; // If TRUE, the exit method is called for exiting an
instance that is copied afterwards (online change).
END_VAR
```

Regard also the FB_init method and the following execution order.

- Exit method: exit old instance: old_inst.fb_exit (bInCopyCode:= TRUE)
- Init method: init new instance: new_inst.fb_init (bInitRetains:= FALSE, bInCopyCode := TRUE)
- Copy function block values (copy code): copy_fub (&old_inst, &new_inst);

Besides this, in case of inheritance the following call sequence is true (whereby for the POUs used for example in this listing the following is assumed: SubFB EXTENDS MainFB and SubSubFB EXTENDS SubFB):

```
fbSubSubFb.FB_Exit(...);
fbSubFb.FB_Exit(...);
```

```
fbMainFb.FB_Exit(...);
fbMainFb.FB_Init(...);
fbSubFb.FB_Init(...);
fbSubSubFb.FB_Init(...);
```

For FB_reinit:

```
fbMainFb.FB_reinit(...);
fbSubFb.FB_reinit(...);
fbSubSubFb.FB_Init(...);
```

Pragma Instructions

A pragma instruction is used to affect the properties of one or several variables concerning the compilation and precompilation process. This means that a pragma influences the code generation. For example it might determine whether a variable will be initialized, monitored, added to a parameter list, made invisible in the library manager or should be added to the symbol configuration. Message outputs during the build process can be forced and also conditional pragmas can be used, which define how the variable should be treated depending on certain conditions. Those pragmas also can be entered as "defines" in the compile properties of a particular object.

A pragma can be used in a separate line, or in with supplementary text in an implementation or declaration editor line. Within the FBD/LD/IL editor use the command *Insert label* and replace the default text *Label*: in the arising text field by the pragma. In case you want to set a label as well as a pragma, insert the pragma first and the label afterwards.

The pragma instruction is enclosed in curly brackets, upper- and lower-case are ignored:

```
{ <Instruction text> }
```

The opening bracket can immediately come after a variable name. Opening and closing bracket must be in the same line.

Depending on the type and contents of a pragma the pragma either operates on the line in which it is located or on all subsequent lines until it is ended by an appropriate pragma, or until the same pragma is executed with different parameters, or until the end of the file is reached. A "file" in this context is a declaration part, implementation part, global variable list or type declaration.

If the compiler cannot meaningfully interpret the instruction text, the entire pragma is handled as a comment and read over. A warning will be issued in this case.

See the following pragma types:

- Message pragmas
- Attribute pragmas
- Additional pragmas

Message Pragma

Message pragmas can be used to force the output of messages in the *Messages* window during the compilation (build) of the project.

The pragma instruction can be inserted in an existing line or in a separate line in the text editor of a POU. Message pragmas positioned within currently not defined sections of the implementation code will not be regarded when the project is compiled. On this see the example provided with the description of the “defined (identifier)” Conditional Pragma.

There are four types of message pragmas:

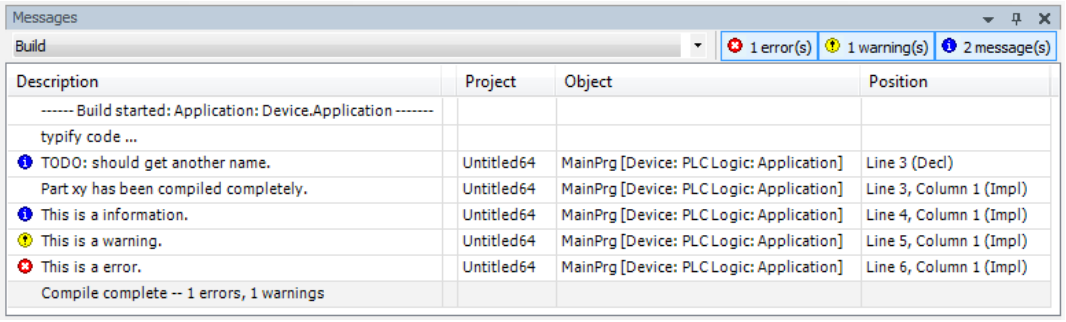
Pragma	Message type
{text 'string'}	Text: The specified textstring will be displayed.
{info 'string'}	ℹ Information: The specified textstring will be displayed.
{warning digit 'string'}	Warning ⚠: The specified textstring will be displayed Unlike a "data type-global" Obsolete Pragma this warning is explicitly defined for the local position.
{error 'string'}	Error ✖: The specified textstring will be displayed.

Table 4-5. Message Pragma Types

NOTE: In case of messages of types Information, Warning and Error the source position of the message - that is where the pragma is placed in a POU - can be reached via the Next Message command. This is not possible for the *Text* type.

Example of declaration and implementation in ST editor:

```
VAR
ivar : INT; {info 'TODO: should get another name.'}
bvar : BOOL;
arrTest : ARRAY [0..10] OF INT;
i:INT;
END_VAR
arrTest[i] := arrTest[i]+1;
ivar:=ivar+1;
{text 'Part xy has been compiled completely.'}
{info 'This is a information.'}
{warning 'This is a warning.'}
{error 'This is a error.'}
```



Messages

Build | ✖ 1 error(s) | ⚠ 1 warning(s) | ℹ 2 message(s)

Description	Project	Object	Position
------ Build started: Application: Device.Application -------			
typify code ...			
ℹ TODO: should get another name.	Untitled64	MainPrg [Device: PLC Logic: Application]	Line 3 (Decl)
Part xy has been compiled completely.	Untitled64	MainPrg [Device: PLC Logic: Application]	Line 3, Column 1 (Impl)
ℹ This is a information.	Untitled64	MainPrg [Device: PLC Logic: Application]	Line 4, Column 1 (Impl)
⚠ This is a warning.	Untitled64	MainPrg [Device: PLC Logic: Application]	Line 5, Column 1 (Impl)
✖ This is a error.	Untitled64	MainPrg [Device: PLC Logic: Application]	Line 6, Column 1 (Impl)
Compile complete -- 1 errors, 1 warnings			

Figure 4-1. Output in Messages Window

Attribute Pragmas

Attribute pragmas might be assigned to a signature in order to influence the compilation and pre-compilation, i.e. the code generation.

There are user defined pragmas which might be used in combination with conditional pragmas and there are also the following predefined standard attribute pragmas.

- Attribute 'displaymode'
- Attribute 'global_init_slot'
- Attribute 'hide'
- Attribute 'hide_all_locals'
- Attribute 'Init_Namespace'
- Attribute 'init_on_onlchange'
- Attribute 'instance-path'

- Attribute 'linkalways'
- Attribute 'Monitoring'
- Attribute 'no_check'
- Attribute 'noinit'
- Attribute 'obsolete'
- Attribute 'pack_mode'
- Attribute 'qualified_only'
- Attribute 'reflection'
- Attribute 'symbol'

User Defined Attributes

It is possible to assign arbitrary user- or application-defined attribute pragmas to POUs, type declarations or variables. This attribute can be queried before compilation by means of conditional pragmas.

Syntax:

```
{attribute 'attribute'}
```

This pragma instruction is valid for the current or - if placed in a separate line - for the subsequent line. See for the use of **Conditional Pragmas**.

An user defined attribute can be assigned to the following objects:

- POUs and Actions
- Variables
- Types

Example for POUs and Actions:

Attribute 'displaymode' to the variable dwVar1:

```
VAR
{attribute 'displaymode':='hex'}
dwVar1: DWORD;
END_VAR
```

Example for variables:

Attribute 'DoCount', defined in **Conditional Compilation Operators**, for the variable ivar is added:

```
PROGRAM MAINPRG
VAR
{attribute 'DoCount'}
ivar:INT;
bvar:BOOL;
END_VAR
```

Displaymode Attribute

With the help of the pragma {attribute 'displaymode'} the display mode of a single variable can be defined. This setting will overwrite the global setting for the display mode of all monitoring variables, done via the commands of the submenu *Display Mode* (available in the *Online* menu).

The pragma must be positioned in the line above the line containing the variables declaration.

Syntax:

```
{attribute ‘display mode’:=’<display mode>’}
```

For display in binary format:

```
{attribute'display mode':='bin'}
{attribute'display mode':='binary'}
```

For display in decimal format:

```
{attribute'display mode':='dec'}
{attribute'display mode':='decimal'}
```

For display in hexadecimal format:

```
{attribute'display mode':='hex'}
{attribute'display mode':='hexadecimal'}
```

Example:

```
VAR
{attribute 'display mode':='hex'}
dwVar1: DWORD;
END_VAR
```

Global_init_slot Attribute

The pragma {attribute 'global_init_slot'} can only be applied for signatures. Per default the order of initializing the variables of a global variable list is arbitrary. However, in some cases prescribing an order is necessary, e.g. if variables of one global list are referring to variables of another list. In this case you may make use of the pragma to arrange the order for global initialization.

Syntax:

```
{attribute 'global_init_slot’ := '<Value>'}
```

The template <Value> has to be replaced by an integer value describing the initialization order of the signature. The default value is 50000. A lower value provokes an earlier initialization. In case of signatures carrying the same value for the attribute 'global_init_slot' the sequence of their initialization rests undefined.

Example:

Assume the project including two global variable lists GVL_1 and GVL_2.

Figure 4-2. Global Variable Lists

The variables B and C are members of global variable list GVL_1, their initial values depend on the variable A.

```
VAR_GLOBAL
B : INT:=A+1;
C : INT:=A-1;
END_VAR
```

The global variable A is member of the global variable list GVL_2.

```
{attribute 'global_init_slot' := '300'}
VAR_GLOBAL
A : INT:=1000;
END_VAR
```

Setting the attribute 'global_init_slot' of GVL_2 to 300 (lowest value in order of initializing), ensures that the expression "A + 1" is defined during initialization of B and C, so you can use these variables normally in any POU of the project. If the pragma is removed from GVL_2 and variables of GVL_1 are used in any POU it will show an error during project building.

Hide Attribute

With the help of the pragma {attribute 'hide'} you may prevent variables or even whole signatures from being displayed within the *List Components* functionality, the *Input Assistant* or the declaration part in online-modus. Only the variable subsequent to the pragma will be hidden.

Syntax:

```
{attribute 'hide’}
```

To hide all local variables of a signature use this attribute.

Example:

The function block myPOU is implemented making use of the attribute:

```
FUNCTION_BLOCK myPOU
VAR_INPUT
a:INT;
{attribute 'hide'}
a_invisible: BOOL;
a_visible: BOOL;
END_VAR
VAR_OUTPUT
b:INT;
END_VAR
VAR
END_VAR
```

In the main program two instances of function block myPOU are defined:

```
PROGRAM mainprg
VAR
POU1, POU2: myPOU;
END_VAR
```

When assigning e.g. an input value to POU1, the *List Components* function working on typing POU1 in the implementation part of MainPrg will display the variables a, a_visible and a, but not the hidden variable a_invisible.

Hide_all_locals Attribute

With the help of the pragma {attribute 'hide_all_locals'} you may prevent all local variables of a signature from being displayed within the *List Components* functionality or *Input Assistant*. This attribute is identical to assigning the attribute hide to each of the local variables.

Syntax:

```
{attribute 'hide_all_locals’}
```

Example:

The function block myPOU is implemented making use of the attribute.

```
{attribute 'hide_all_locals'}
FUNCTION_BLOCK myPOU
VAR_INPUT
a:INT;
END_VAR
VAR_OUTPUT
b:BOOL;
END_VAR
VAR
c,d:INT;
END_VAR
```

In the main program two instances of function block myPOU are defined:

```
PROGRAM MainPrg
VAR
POU1, POU2: myPOU;
END_VAR
```

When assigning e.g. an input value to POU1, the *List Components* function working on typing POU1 in the implementation part of MainPrg will display the variables A and B, but none of the hidden local variables C or D.

Init_Namespace Attribute

A variable of type STRING or WSTRING, which is provided with the pragma {attribute 'init_namespace'}, will be initialized with the current namespace, this is the path of the instance of the function block this variable is contained. Applying this pragma presumes the use of the additional attribute 'noinit' for the string variable and the attribute 'reflection' for the corresponding function block.

Syntax:

```
{attribute 'init_namespace’}
```

Example:

The function block POU is provided with all necessary attributes.

```
{attribute 'reflection'}
FUNCTION_BLOCK POU
VAR_OUTPUT
{attribute 'no_init'}
{attribute 'instance-path'}
myStr: STRING;
END_VAR
```

Within the main program MainPrg an instance fb of the function block POU is defined.

```
PROGRAM MAINPRG
VAR
fb:POU;
newString: STRING;
END_VAR
newString:=fb.myStr;
```

The variable myStr will be initialized with the current namespace, e.g. Device.Application.MainPrg.fb. This value will be assigned to newString within the main program.

Init_on_onlchange Attribute

The pragma {attribute 'init_on_onlchange'} attached to a variable will cause this variable to get initialized with each online change.

Syntax:

```
{attribute 'init_on_onlchange'}
```

Instance-path Attribute

The pragma {attribute 'instance-path'} may be added to a local string variable that, in consequence, will be initialized with the device tree path of the POU this string variable belongs to. This may be a useful feature for error messages. Applying this pragma presumes the use of the attribute 'reflection' for the corresponding POU and the additional attribute 'noinit' for the string variable.

Syntax:

```
{attribute 'instance-path’}
```

Example:

Assume the following function block being equipped with the attribute 'reflection':

```
{attribute 'reflection'}
FUNCTION_BLOCK POU
VAR
{attribute 'instance-path'}
{attribute 'noinit'}
str: STRING;
END_VAR
```

An instance of the function block is called in the program MainPrg:

```
PROGRAM MAINPRG
VAR
myPOU: POU;
myString: STRING;
END_VAR
myPOU();
myString:=myPOU.str;
```

The instance myPOU having been initialized the string variable STR will be assigned the path of the instance myPOU of POU, e.g. Device.Application.MAINPRG.myPOU. This path will be assigned to variable myString within the main program.

NOTE: The length of a string variable may be arbitrarily defined (even 255), however be aware that the string will be cut (from its back end) if it gets assigned to a variable of a too short data type.

Linkalways Attribute

The pragma {attribute 'linkalways'} allows to mark POUs for the compiler in a way so that they are always included into the compile information. As a result, objects with this option will be always compiled and downloaded to the PLC. This option only affects POUs and GVLs that are located below an application or in libraries which are inserted below an application. The compiler option *Link always* does the same.

Syntax:

```
{attribute 'linkalways’}
```

When using the symbol configuration editor the marked POUs are used as basis for the selectable variables for the symbol configuration.

Example:

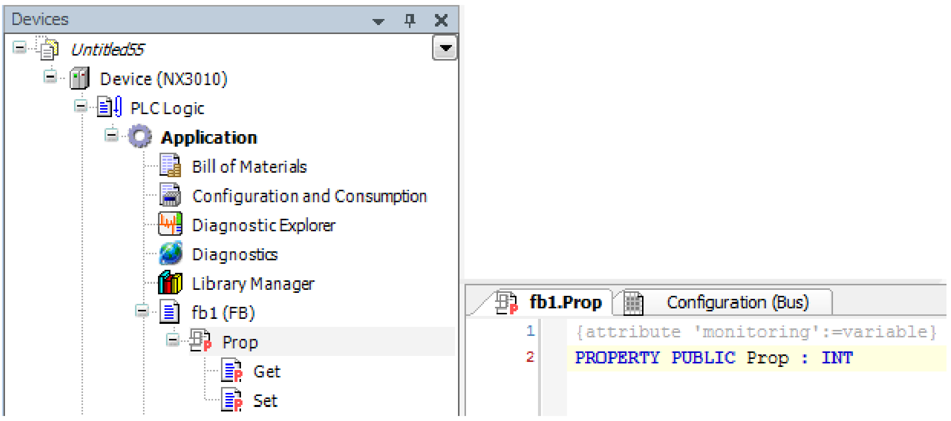
The global variable list GVLMoreSymbols is implemented making use of the attribute.

```
{attribute 'linkalways'}
VAR_GLOBAL
g_iVar1: INT;
g_iVar2: INT;
END_VAR
```

Monitoring Attribute

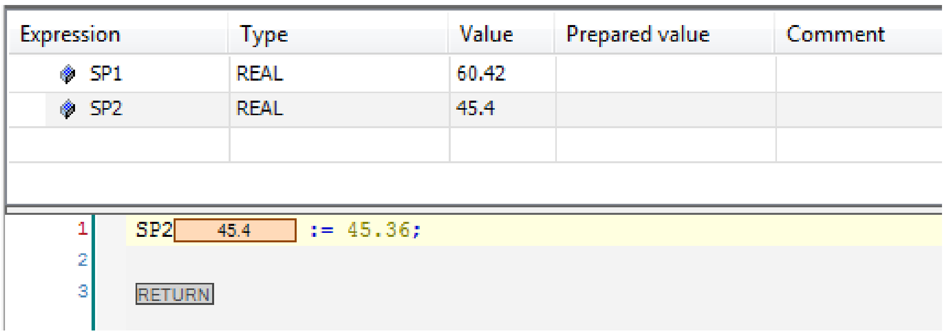
A property may be monitored in online mode either with help of *Monitoring* window.

The monitoring can be activated by adding the monitoring attribute pragma in the line above the property definition. Then in online view of the POU using the property and in a watch list the name, type and value of the variables of the property will be displayed. Therein you may also enter prepared values to force variables belonging to the property.



```
{attribute 'monitoring':=variable}
PROPERTY PUBLIC Prop : INT
```

Figure 4-3. Example of Property Prepared for Variable Monitoring



Expression	Type	Value	Prepared value	Comment
SP1	REAL	60.42		
SP2	REAL	45.4		

```
SP2 45.4 := 45.36;

RETURN
```

Figure 4-4. Example of Monitoring View

There are two different ways to get monitored the current value of the property variables. For the particular use case consider carefully which attribute is suitable to actually get the desired value. This will depend on whether operations on the variables are implemented within the property.

Pragma {attribute 'monitoring':=variable}

Pragma {attribute 'monitoring':='call'}

In the first case ({attribute 'monitoring':='variable'}'), an implicit variable is created for the property, which will get the current property value always when the application calls the Set or Get method. The value of this implicit variable will be monitored.

In the second case ({attribute 'monitoring':='call'}', the attribute can only be used for properties returning simple data types or pointers, not for structured types.

The value to be monitored is retrieved directly by calling the property, i.e. the monitoring service of the runtime system calls the Get method. Regard that if any operations on the variables are implemented within the property, the value might still change.

No_check Attribute

The pragma {attribute 'no_check'} is added to a POU in purpose to suppress the call of any **POUs for Implicit Checks**. As check functions may influence the performance, it is reasonable to apply this attribute to POUs that are frequently called or already approved.

Syntax:

```
{attribute 'no_check’}
```

Example:

Using the POU for implicit checks CheckRangeSigned, for example, added to the project, run the code below with the pragma {attribute 'no_check'}, the function won’t be checked in this case, allowing the variable "x" to accept any value.

```
{attribute 'no_check'}
PROGRAM MainPrg
VAR
   x: DINT (100..200);
   y: DINT;
END_VAR
x := y;
```

No_init Attribute

Variables provided with the pragma {attribute 'no_init'} won't be initialized implicitly. The pragma belongs to the variable declared subsequently.

Syntax:

```
{attribute 'no_init'}
```

or

```
{attribute'no-init'}
{attribute 'noinit'}
```

Example:

```
PROGRAM MainPrg
VAR
A : INT;
{attribute 'no_init'}
B : INT;
END_VAR
```

If a reset is performed on the associated application the integer variable A will be again initialized implicitly with 0, whereas variable B maintains the value it is currently assigned to.

Obsolete Attribute

An pragma {attribute 'obsolete'} can be added to a data type definition in order to cause a user-defined warning during a build run, if the respective data type (structure, function block etc.) is used within the project. Thus for example you might announce that the data type should not be used any longer.

Unlike a locally used Message Pragma this warning is defined within the definition and thus globally for all instances of the data type.

This pragma instruction always is valid for the current line or - if placed in a separate line - for the subsequent line.

Syntax:

```
{attribute 'obsolete' := 'User defined text'}
```

Example:

The obsolete pragma is inserted in the definition of function block fb1.

```
{attribute 'obsolete' := 'Data type fb1 not valid'}
FUNCTION_BLOCK fb1
VAR_INPUT
i:INT;
END_VAR
...
```

If fb1 is used as a data type in a declaration, e.g. fbinst: fb1, the following warning will be dumped when the project is built: Data type fb1 not valid.

Pack_mode Attribute

The pragma {attribute 'pack_mode'} defines the mode a data structure is packed while being allocated. The attribute has to be set on top of a data structure and will influence the packing of the whole structure.

Syntax:

```
{attribute 'pack_mode’ := '<Value>'}
```

The template <Value> included in single quotes has to be replaced by one of the following values available:

Value	Description
0	Aligned, i.e. there will be no memory gaps.
1	1-byte- aligned (identical to aligned).
2	2-byte- aligned, i.e. the maximum size of a memory gap is 1 byte.
4	4-byte- aligned, i.e. the maximum size of a memory gap is 3 bytes.
8	8-byte-aligned, i.e. the maximum size of a memory gap is 7 bytes.

Table 4-6. Pack_mode Attribute

Example:

```
{attribute'pack_mode':= '1'}
TYPE myStruct:
STRUCT
Enable: BOOL;
Counter: INT;
MaxSize: BOOL;
MaxSizeReached: BOOL;
END_STRUCT
END_TYPE
```

A variable of data type myStruct will be instantiated aligned: If the address of its component Enable is 0x0100 for example, then the component Counter will follow on address 0x0101, MaxSize on 0x0103 and MaxSizeReached on 0x0104. With 'pack_mode'=2 Counter would be found on 0x0102, MaxSize on 0x0104 and MaxSizeReached on 0x0105.

NOTE: The attribute may also be applied to POUs. Though, you have to be careful with its application due to eventually existing internal pointers of the POU.

Qualified_only Attribute

After assigning the pragma {attribute 'qualified_only'} on top of a global variable list, its variables can only be accessed by using the global variable name, e.g. gvl.g_var. This works even for variables of enumeration type and may be useful to avoid name mismatch with local variables.

Syntax:

```
{attribute 'qualified_only’}
```

Example:

Assume the following global variable list GVL with attribute 'qualified_only':

```
{attribute 'qualified_only'}
VAR_GLOBAL
iVar:INT;
END_VAR
```

Within a POU MainPrg, for example, this global variable has to be called with the prefix GVL:

```
GVL.iVar:=5;
```

The following incomplete call of the variable will cause instead an error:

```
iVar:=5;
```

Reflection Attribute

The pragma {attribute 'reflection'} is attached to signatures. Due to performance reasons it is an obligatory attribute for POUs carrying the instance-path attribute.

Syntax:

```
{attribute 'reflection'}
```

Example:

See the example of the instance-path attribute.

Symbol Attribute

The pragma {attribute 'symbol'} defines which variables are to be handled in the *Symbol configuration*, which means that they will be exported as symbols into a symbol list, exported to a XML-file (<Project Name>.Device.Application.xml) in the project directory as well as to a file not visible for the user and available on the target system for external access, e.g. by an OPC-Server. Variables provided with that attribute will be downloaded to the PLC even if they have not been configured or are not visible within the symbol configuration editor.

Regard that the *Symbol configuration* must be available as an object below the respective application in the *Devices* tree.

Syntax:

```
{attribute 'symbol' := 'none' | 'read' | 'write' | 'readwrite'}
```

Regard that access is only allowed on symbols coming from programs or global variable lists. For accessing a symbol the symbol name must be specified completely.

The pragma definition can be assigned to particular variables or collectively to all variables declared in a program:

- To be valid for a single variable the pragma has to be placed in the line before the variables declaration
- To be valid for all variables contained in the declaration part of a program the pragma has to be placed in the first line of the declaration editor. Anyway also in this case the settings for particular variables might be modified by an explicit adding of a pragma

The possible access on a symbol is defined by the pragma parameter 'none', 'read', 'write' or 'readwrite'. If no parameter is defined, the default 'readwrite' will be valid.

Example:

With the following configuration the variables A and B will be exported with read and write access. Variable D will be exported with read access.

```
{attribute 'symbol' := 'readwrite'}
PROGRAM MAINPRG
VAR
A : INT;
B : INT;
{attribute 'symbol' := 'none'}
C : INT;
{attribute 'symbol' := 'read'}
D : INT;
END_VAR
```

Conditional Pragmas

The ExST (Extended ST) language supports several conditional pragma instructions which affect the code generation in the pre-compile and compile process.

The implementation code which will be regarded for compilation for example might depends on:

- whether a certain data type or variable is declared
- whether a type or variable has got a certain attribute
- whether a variable has a certain data type
- whether a certain POU or task is available and is part of the call tree

NOTE: It is not possible for a POU or GVL declared in the POUs tree to use a {define...} declared in an application.

Pragma	Description
{define identifier string}	During preprocessing all subsequent instances of the identifier will be replaced with the given sequence of tokens, if the token string is not empty (which is allowed and well-defined). The identifier remains defined and in scope until the end of the object or until it is undefined in an {undefine} directive. See Conditional compilation below.
{undefine identifier}	The identifier's preprocessor definition (by {define}, see above) will be removed, the identifier hence is "undefined". If the specified identifier is not currently defined, this pragma will be ignored.

{IF expr}
...
{ELSIF expr}
...
{ELSE}
...
{END_IF}
	These are pragmas for conditional compilation. The specified expressions **exprs** are required to be constant at compile time; they are evaluated in the order in which they appear until one of the expressions evaluates to a nonzero value. The text associated with the successful directive is preprocessed and compiled normally; the others are ignored. The order of the sections is determinate; however, the **elsif** and **else** sections are optional, and **elsif** sections may appear arbitrarily often.Within the constant expr several "conditional compilation operators" can be used, which are described below.

Table 4-7. Conditional Pragmas

Conditional Compilation Operators

Within the constant expression "expr" of a conditional compilation pragma ({if} or {elsif}), several operators can be used. These operators itself may not be undefined or redefined via {undefine} or {define}, respectively. Regard that these expressions as well as the definition done by {define} can also be used in the *Compiler defines* in the *Properties* dialog of an object.

The following operators are supported:

SYNTAX	Description

Defined (Identifier)	When applied to an identifier, its value is TRUE if that identifier has been defined with a {define} instruction and not later undefined using {undefine}; otherwise its value is FALSE.
	Example: Precondition: there are two POUs. The variable "pdef1" is defined by a {define} on POU_1, but not in POU_2. Both have the code below:
	{IF DEFINED (PDEF1)}
	(* This code is processed in POU_1 *)
	{INFO 'PDEF1 DEFINED'}
	IVAR := IVAR + SINT#1;
	{ELSE}
	(*This code is processed in POU_2*)
	{INFO 'PDEF1 NOT DEFINED'}
	IVAR := IVAR - SINT#1;
	{END_IF}
	Here additionally an example of a message pragma is included: Only information string "pdef1 defined" will be displayed in the message

SYNTAX	Description
	window when the POU_1 was called, because pdef1 actually is defined. Info message "pdef1 not defined" will be displayed in case pdef1 is not defined, POU_2.
Defined (variable: name of variable)	When applied to a variable, its value is TRUE if this particular variable is declared within the current scope. Otherwise is FALSE. Both have the code below: Example: Precondition: There are two POUs, POU_1 and POU_2. Variable g_bTest is declared in POU_2, but not in POU_1. {IF defined (variable:g_bTest)} (*The following code is only processed in POU_2 *) g_bTest := x > 300; {END_IF}

Defined (type: identifier)	When applied to a type identifier, its value is TRUE if a type with that particular name is declared. Otherwise is FALSE.
	Example: Precondition: At first declare the data type of DUT in the project, does not state a second time, see the difference in the boolean variable "bDutDefined."
	{IF defined (type:DUT)}
	(*The following code line only will be processed if have the data type declared*)
	bDutDefined := TRUE;
	{END_IF}

Defined (pou: POU name)	When applied to a pou-name, its value is TRUE if a POU or an action with that particular name is defined. Otherwise is FALSE.
	Example:
	Precondition: In a scenario the POU CheckBounds was added to the project and not in another. In a POU there is the code below:
	{IF defined (pou:CheckBounds)}
	(*The following line of code will only be processed in the scenario where there is a POU CheckBounds *)
	arrTest[CheckBounds(0,i,10)] := arrTest[CheckBounds(0,i,10)] + 1;
	{ELSE}
	(*The following line of code will only be processed in the scenario where there is not a POU CheckBounds *)
	arrTest[i] := arrTest[i]+1;
	{END_IF}

Hasattribute (pou: POU name, 'attribute')	TRUE if this particular attribute is specified in the first line of the POUs declaration part. Example: Precondition: There are two function blocks fun1 and fun2, but in fun1 additionally has got an attribute 'vision'. Definition of fun1: {attribute 'vision'} FUNCTION fun1 : INT VAR_INPUT i : INT; END_VAR VAR END_VAR Definition of fun2: FUNCTION fun2 : INT VAR_INPUT i : INT; END_VAR VAR END_VAR In a POU exists the following code: {IF hasattribute (pou: fun1, 'vision')} (* The following code line will be processed *) ergvar := fun1(ivar); {END_IF} {IF hasattribute (pou: fun2, 'vision')}

SYNTAX	Description

	(* The following code line not will be processed *)
	ergvar := fun2(ivar);
	{END_IF}

Hasattribute (variable: name of variable, 'attribute')	When applied to a variable, its value is TRUE if this particular attribute is specified via the {attribute} instruction in a line before the variable's declaration.
	Example:
	Precondition: There are two POUs, POU_1 and POU_2. Variable “g_globalInt” is used in POU_1 and POU_2, but in POU_1 additionally
	has got an attribute 'DoCount' :
	Declaration of g_globalInt in POU_1:
	VAR_GLOBAL
	{attribute 'DoCount'}
	g_globalInt : INT;
	g_multiType : STRING;
	END_VAR
	Declaration of g_globalInt in POU_2 :
	VAR_GLOBAL
	g_globalInt : INT;
	g_multiType : STRING;
	END_VAR
	{IF hasattribute (variable: g_globalInt, 'DoCount')}
	(*The following code line will only be processed in POU_1, because there variable g_globalInt has got the attribute 'DoCount' *)
	g_globalInt := g_globalInt + 1;
	{END_IF}

Hastype (variable: variable, type)	When applied to a variable, its value is TRUE if this particular variable has the specified type-spec. Otherwise is FALSE ANY ANY_DERIVED ANY_ELEMENTARY ANY_MAGNITUDE ANY_BIT ANY_STRING ANY_DATE ANY_NUM ANY_REAL ANY_INT LREAL REAL LINT DINT INT SINT ULINT UDINT UINT USINT TIME LWORD DWORD WORD BYTE BOOL STRING WSTRING DATE_AND_TIME DATE TIME_OF_DAY Example: Precondition: In a POU with the code below, at first time the variable

SYNTAX	Description

	g_multitype is declared as LREAL and at second time is declared as STRING.
	{IF (hastype (variable: g_multitype, LREAL))}
	(*The following line of code will only be processed when the variable is declared as LREAL *)
	g_multitype := (0.9 + g_multitype) * 1.1;
	{ELSIF (hastype (variable: g_multitype, STRING))}
	(*The following line of code will only be processed when the variable is declared as STRING *)
	g_multitype := ‘this is a multi-talented’;
	{END_IF}

Hasvalue (define-ident, string)	If the define (define-ident) is defined and it has the specified value (char-string), then its value is TRUE, otherwise is FALSE
	Example:
	Precondition: Variable "test" is used in a POU, it gets value "1" at first and value "2" a second time.
	{IF has value(test,'1')}
	(*The following code line will be processed when variable test has value "1" *)
	x := x + 1;
	{ELSIF has value(test,'2')}
	(*The following code line will be processed when variable test has value "2" *)
	x := x + 2;
	{END_IF}

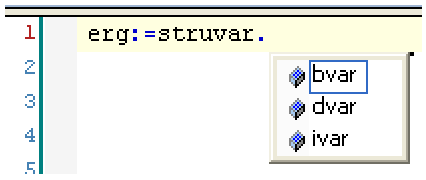
NOT operator	Inverts the value of the operator. Example: Precondition: In a scenario the CheckBounds function was added to the project and not in another, there is the POU MainPrg in both cases. In a POU there is the following code: {IF defined (pou: MainPrg) AND NOT (defined (pou: CheckBounds))} (* The following line of code will only be processed in the scenario where there is not a CheckBounds function *) bAndNotTest := TRUE; {END_IF}
AND operator	Is TRUE if both operators are TRUE. Example: Precondition: In a scenario the CheckBounds function was added to the project and not in another, the POU MainPrg there is in both cases. In a POU there is the following code: {IF defined (pou: MAINPRG) AND (defined (pou: CheckBounds))} (* The following line of code will only be processed in the first scenario, because only there MainPrg and CheckBounds are defined *) bAndTest := TRUE; {END_IF}
OR operator	Is TRUE if one of the operators are TRUE. Example: Precondition: In a scenario the CheckBounds function was added to the project and not in another, the POU MainPrg there is in both cases. In a POU there is the following code: {IF defined (pou: MAINPRG) OR (defined (pou: CheckBounds))} (* The following line of code is processed in two scenarios, because both contain at least one of the POUs *) bOrTest := TRUE; {END_IF}
(Operator)	Braces "()" the operator.

Table 4-8. Syntax of “Defined”

List Components Functionality

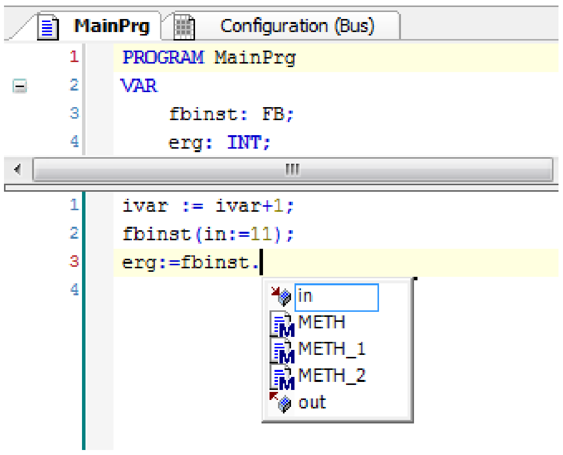
The text input is supported noticing standard IEC 61131-3. The feature *List Components* (*Tools, Options, SmartCoding* menu) helps to insert a correct identifier:

- If you - at any place, where a global identifier can be inserted - insert a dot (".") instead of the identifier, a selection box will appear, listing all currently available global variables. You can choose one of these elements and press <ENTER> to insert it behind the dot. You can also insert the element by a double click on the list entry
- If you enter a function block instance or a structure variable followed by a dot, then a selection box listing all input and output variables of the corresponding function block resp. listing the structure components will appear, where you can choose the desired element and enter it by pressing <ENTER> or by a double click. Examples:



```
erg:=struvar.
```

Figure 4-5. List Components (Structure)



```
PROGRAM MainPrg
VAR
    fbinst: FB;
    erg: INT;
```

```
ivar := ivar+1;
fbinst(in:=11);
erg:=fbinst.
```

Figure 4-6. List Components (Function Block)

- If you enter any string and press <CTRL>+<SPACE> a selection box will appear listing all POUs and global variables available in the project. The first list entry, which is starting with the given string, will be selected and can be entered to the program by pressing the <ENTER> key

I/O Mapping

This sub-dialog of the devices is called *Bus I/O Mapping*, it used to setup a I/O mapping of the PLC. This means that the project variables, used by applications, are assigned to inputs and outputs of the device.

General

All I/O mapping can be set for the current device.

An address can also be assigned to a variable using AT declaration. In this case consider the following points:

- AT declarations can be used with only the local or global variables, however, not with inputs and outputs variable of the POUs.
- For AT declarations is not possible to create forces for variables.

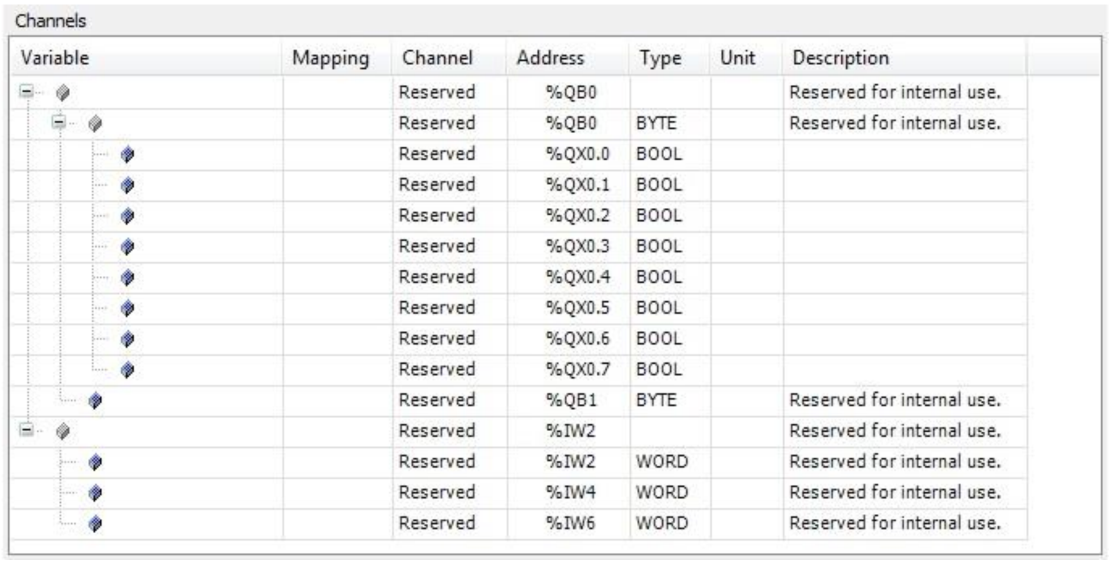
- If AT declarations are used to structure or function block, all instances will access the same memory location, corresponding to static variables in programming languages, such as C.

For further information, see **AT Declaration**.

The *Bus I/O Mapping* dialog contains on its bottom part the following commands:

- *Reset mapping*: This button resets the settings for the mapping default defined by the device description file.
- *Always update variables*: If this option is enabled, all variables are updated in each cycle of the task, no matter if they are used, or if they are mapped to an input or output.

Channels



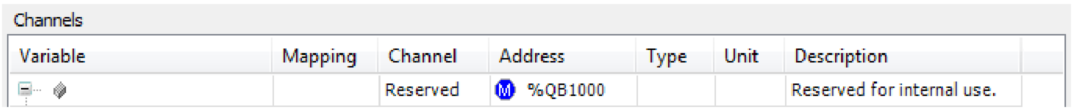
Channels

Variable	Mapping	Channel	Address	Type	Unit	Description
		Reserved	%QB0			Reserved for internal use.
		Reserved	%QB0	BYTE		Reserved for internal use.
		Reserved	%QX0.0	BOOL		
		Reserved	%QX0.1	BOOL		
		Reserved	%QX0.2	BOOL		
		Reserved	%QX0.3	BOOL		
		Reserved	%QX0.4	BOOL		
		Reserved	%QX0.5	BOOL		
		Reserved	%QX0.6	BOOL		
		Reserved	%QX0.7	BOOL		
		Reserved	%QB1	BYTE		Reserved for internal use.
		Reserved	%IW2			Reserved for internal use.
		Reserved	%IW2	WORD		Reserved for internal use.
		Reserved	%IW4	WORD		Reserved for internal use.
		Reserved	%IW6	WORD		Reserved for internal use.

Figure 4-7. Channels Dialog

- *Variable*: a variable can be mapped to the channel through the *Input Assistant*, or created a new one by editing the field
- *Mapping*: Indicates whether the variable is new ( ), will be declared internally as a global variable, or will be a mapping to an existing ( ) in this case the address will appear scratched and should not be used directly
- *Channel*: Channel name input or output of the device
- *Address*: Address of the channel, for example: "% IW0"
- *Type*: Data type of the input or output channel, for example: "BOOL"
- *Unit*: The unit of the parameter value, for example "MS" for milliseconds
- *Description*: Text description for the parameter
- *Current value*: Current value of the parameter, displayed in the online mode

You can modify and correct an input or output *Address* field. To do this select the column address and press <SPACE> to open the edit field. Now, modify or unchanged the value and close the edit field by pressing <ENTER>. The address field will be marked by the symbol Ⓜ.



Channels

Variable	Mapping	Channel	Address	Type	Unit	Description
		Reserved	Ⓜ %QB1000			Reserved for internal use.

Figure 4-8. Example of Modified Address Manually

You can only change the whole input or output address, not its sub elements. Thus, if an input or output is represented in the mapping table with any subtree, only the highest address field can be edited.

If you want to remove the fixing of the value, reopen the *Address* field edition, delete the value and close by <ENTER>. The address and the following addresses will be set back to the values they had before the manual modification and the symbol Ⓜ will be removed.

NOTE: For projects created from the *MasterTool Standard Project*, the devices channels will be modified by MasterTool IEC XE in order to maintain a better distribution and organization of channels. Therefore, they will appear with the symbol Ⓜ.

Data Types

You can use standard data types, user-defined data types or instances of function blocks when programming. Each identifier is assigned to a data type which dictates how much memory space will be reserved and what type of values it stores.

Standard Data Types

All data types described by standard IEC 61131-3 are supported by MasterTool IEC XE. See here the following:

- BOOL / BIT
- Integer Data Types
- REAL / LREAL
- STRING / WSTRING
- Time Data Types

Notice that there are also some norm-extending data types and that you can also define types on your own (user defined data types).

BOOL

BOOL type variables may be given the values TRUE (1) and FALSE (0). 8 bits of memory space will be reserved.

See also: **BOOL Constants**.

BIT

As the BOOL type variables, the BIT type variables may have values TRUE and FALSE. Unlike the BOOL type, the BIT type occupies only one bit of memory space. However, for this allocation can happen correctly this data type can only be declared in Function Blocks or Structures (defined on DUT type objects).

Integer Data Types

See below a list of all available integer data types. Each of the different number types covers a different range of values. The following range limitations apply to the integer data types:

Type	Lower limit	Upper limit	Memory space
BYTE	0	255	8 Bits
WORD	0	65535	16 Bits
DWORD	0	4294967295	32 Bits
LWORD	0	2^{64}-1	64 Bits
SINT	-128	127	8 Bits
USINT	0	255	8 Bits
INT	-32768	32767	16 Bits
UINT	0	65535	16 Bits

DINT	-2147483648	2147483647	32 Bits
UDINT	0	4294967295	32 Bits
LINT	-2^{63}	$2^{63}-1$	64 Bits
ULINT	0	$2^{64}-1$	64 Bits

Table 4-9. Integer Data Types

As a result when larger types are converted to smaller types, information may be lost.

See also: **Number Constants**.

REAL/LREAL

REAL and LREAL are so-called floating-point types. They are required to represent rational numbers. 32 bits of memory space is reserved for REAL and 64 bits for LREAL.

Value range for REAL: 1.175494351e-38 to 3.402823466e+38.

Value range for LREAL: 2.2250738585072014e-308 to 1.7976931348623158e+308.

NOTES:
- The support of data type LREAL depends on the target device. Please see in the corresponding documentation whether the 64 bit type LREAL gets converted to REAL during compilation (possibly with a loss of information) or persists.
- If a REAL or LREAL is converted to SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT and the value of the real number is out of the value range of that integer, the result will be undefined and will depend on the target system. Even an exception is possible in this case! In order to get target-independent code, handle any range exceedance by the application. If the real/lreal number is within the integer value range, the conversion will work on all systems in the same way.

See also: **REAL/LREAL Constants**.

STRING

A STRING type variable can contain any string of characters. The size entry in the declaration determines how much memory space should be reserved for the variable. It refers to the number of characters in the string and can be placed in parentheses or square brackets. If no size specification is given, the default size of 80 characters will be used.

The string length basically is not limited in MasterTool IEC XE, but string functions only can process strings of 1 - 255 characters! If a variable is initialized with a string too long for the variables datatype, the string will be correspondingly cut from right to left.

Example of a String Declaration with 35 characters:

```
str:STRING(35):='This is a string';
```

See also: **WSTRING** and **STRING Constants**.

Time Data Types

The data types TIME, TIME_OF_DAY (abb. TOD), DATE and DATE_AND_TIME (abb. DT) are handled internally like DWORD.

Time is given in milliseconds in TIME and TOD, time in TOD begins at 12:00 A.M.

Time is given in seconds in DATE and DT beginning with January 1, 1970 at 12:00 A.M.

Notice in this context also:

- **LTIME** (available as a 32-Bit time)
- **TIME Constants**
- **DATE Constants**

- **DATE_AND_TIME Constants**
- **TIME_OF_DAY Constants**

Extensions to the IEC 1131-3 Standard

Norm- Extended Data Types

In addition to the data types according to IEC1131-3 there are also some norm-extending data types available implicitly in MasterTool IEC XE:

- UNION
- LTIME
- WSTRING
- Pointers
- References

UNION

As an extension to the IEC 61131-3 standard it is possible to declare unions in user-defined types.

In a union all components have the same offset: they all occupy the same storage location. Thus, assuming a union definition as shown in the following example, an assignment to name “.a” also would manipulate name “.b”.

Example:

```
TYPE name: UNION
a : LREAL;
b : LINT;
END_UNION
END_TYPE
```

LTIME

As extension to the IEC 61131-3 LTIME is supported as time base for high resolution timers. LTIME is of size 64 Bit and resolution nanoseconds.

Syntax:

```
LTIME#<Time declaration>
```

The time declaration can include the time units as used with the TIME constant and additionally, microseconds (us) and nanoseconds (ns).

Example:

```
LTIME1 := LTIME#1000d15h23m12s34ms2us44ns
```

Compare to **TIME Constants** (size 32 Bit and resolution milliseconds).

WSTRING

This string data type is an extension to the IEC 61131-3 standard.

It differs from the standard STRING type (ASCII) by getting interpreted in Unicode format.

Example:

```
wstr:WSTRING:="This is a WString";
```

See also: STRING and **STRING Constants** (Operands).

Pointers

As an extension to the IEC 61131-3 standard it is possible to use pointers.

Pointers save the addresses of variables, programs, function blocks, methods and functions while an application program is running. A pointer can point to any of those objects and to any data type, even to user-defined data types. Notice the possibility of using an implicit pointer check function.

Syntax:

```
<Identifier>: POINTER TO <Data type | Function block | Program | Method |
Function>;
```

Dereferencing a pointer means to obtain the value currently stored at the address to which it is pointing A pointer can be dereferenced by adding the content operator “^” after the pointer identifier; see for example “pt^” in the example below.

The Address Operator ADR can be used to assign the address of a variable to a pointer.

Example:

```
VAR
pt:POINTER TO INT; (* Declaration of pointer pt *)
var_int1:INT := 5; (*Declaration of variables var_int1 and var_int2 *)
var_int2:INT;
END_VAR
pt := ADR(var_int1); (* Address of varint1 is assigned to pointer pt *)
var_int2:= pt^; (* Value 5 of var_int1 gets assigned to var_int2 via
dereferencing of pointer pt; *)
```

Function Pointers

Function pointers are supported, replacing the INDEXOF operator. These pointers can be passed to external libraries, but there is no possibility to call a function pointer within an application in the programming system. The runtime function for registration of callback functions (system library function) expects the function pointer, and, depending on the callback for which the registration was requested, then the respective function will be called implicitly by the runtime system (for example at STOP). In order to enable such a system call (runtime system) the respective property (category *Build*) must be set for the function object.

The ADR operator can be used on function names, program names, function block names and method names. Since functions can move after online change, the result is not the address of the function, but the address of a pointer to the function. This address is valid as long as the function exists on the target.

See also: **INDEXOF**.

Index Access to Pointers

As extension to the IEC 61131-3 standard, index access “[]” to variables of type POINTER, STRING and WSTRING is allowed.

- Pint[i] will return the base data type
- Index access on pointers is arithmetic: If the index access is used on a variable of type pointer, the offset will be calculated by: pint[i] = (pint + i * SIZEOF(base type))^. The index access also performs an implicit dereferencation on the pointer. The result type is the base type of the pointer. Note that pint[7] != (pint + 7)^!
- If the index access is used on a variable of type STRING, the result is the character at offset index-expr. The result is of type BYTE. str[i] will return the i-th character of the string as a SINT (ASCII)
- If the index access is used on a variable of type WSTRING the result is the character at offset index-expr. The result is of type WORD. wstr[i] will return the i-th character of the string as INT (Unicode)

NOTE: There is also the possibility of using **References**, which in contrast to a pointer directly affect a value.

CheckPointer Function

For checking pointer access during runtime you might use the implicitly available check function *CheckPointer* being called before each access on the address of a pointer. Therefore add the object *POUs for implicit checks* to the application using the *Add Object* dialog. Mark the checkbox related to the type *CheckPointer*, choose an implementation language and confirm your settings with *Open*, whereon the check function will be opened in the editor corresponding to the implementation language selected. Independently of that choice the declaration part is preset and may not be modified except for adding further local variables. However, in contrast to other check functions, there is no default implementation of *CheckPointer* available, the implementation is left to the user.

Function *CheckPointer* should check whether the address the pointer refers to is within the valid memory range. In addition it should be taken care of that the alignment of the referenced memory area fits to the data type of the variable the pointer points to. If both conditions are fulfilled, *CheckPointer* should return the unchanged input pointer. A proper handling of detected error cases is left to the user.

Template:

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckPointer : POINTER TO BYTE
VAR_INPUT
ptToTest : POINTER TO BYTE;
iSize : DINT;
iGran : DINT;
bWrite: BOOL;
END_VAR
```

Implementation part:

```
// No standard way of implementation. Fill your own code here.
CheckPointer := ptToTest;
```

When called the function gets the following input parameters:

- ptToTest: Target address of the pointer
- iSize: Size of referenced variable; the data type of iSize must be integer-compatible and must cover the maximum potential data size stored at the pointer address
- iGran: Granularity of the access, that is the largest not-structured datatype used in the referenced variable; the data type of iGran must be integer-compatible
- bWrite: type of access (TRUE=write access, FALSE= read access); the data type of bWrite must be BOOL
- Return value: Address which is used for dereferencing the pointer, thus at best the one that has been passed on as the first input parameter (ptToTest)

References

This data type is an extension to the IEC 61131-3 standard.

A REFERENCE is an alias for an object. The alias can be written or read via identifiers. The difference to a pointer is that the value pointed to is directly affected and that the assignment of reference and value is fix. The address of the reference must be set via a separate assignment operation. Whether a reference points to a valid value (that is unequal 0) can be checked with the help of a special operator, see below.

A reference is declared according to the following syntax:

Syntax:

```
<Identifier> : REFERENCE TO <DATA TYPE>
```

Example declaration:

```
ref_int : REFERENCE TO INT;
a : INT;
b : INT;
```

The ref_int is now available for being used as an alias for variables of type INT.

Example of use:

```
ref_int REF= a; (*ref_int does now point to a *)
ref_int := 12; (* a does now have the value 12 *)
b := ref_int * 2; (* b does now have the value 24 *)
ref_int REF= b; (*ref_int does now point to *)
ref_int := a / 2; (* b 6 *)
ref_int REF= 0; (*Explicit initialization of the reference *)
```

NOTES:
- It is not possible to declare references like REFERENCE TO REFERENCE, ARRAY OF REFERENCE or POINTER TO REFERENCE.
- The references will be initializes (with 0).

Check For Valid References

Operator “__ISVALIDREF” can be used to check whether a reference points to a valid value, that is a value unequal 0.

Syntax:

```
<Boolean variable> := __ISVALIDREF(<Identifier declared with type
REFERENCE TO < Data Type>);
```

<Boolean variable> will be TRUE, if the reference points to a valid value, FALSE if not.

Example:

Declaration:

```
ivar : INT;
ref_int : REFERENCE TO INT;
ref_int0: REFERENCE TO INT;
testref: BOOL := FALSE;
```

Implementation:

```
ivar := ivar +1;
ref_int REF= hugo;
ref_int0 REF= 0;
testref := __ISVALIDREF(ref_int); (* Will be TRUE, because ref_int points
to ivar, which is unequal 0 *)
testref0 := __ISVALIDREF(ref_int0); (*Will be FALSE, because ref_int is
set to 0 *)
```

User Defined Data Types

Additionally to the standard data types the user can define special data types within a project.

These definitions are possible via creating DUT (Data unit type) objects in the POUs window and within the declaration part of a POU.

Please notice the given recommendations on the naming of objects in order to make it as unique as possible.

See the following user defined data types:

- ARRAYS
- Structures
- Enumerations
- Subrange Types

- References
- Pointers

ARRAYS

One-, two-, and three-dimensional fields (ARRAYS) are supported as elementary data types. Arrays can be defined both in the declaration part of a POU and in the global variable lists. Notice the possibility of using implicit boundary checks.

Syntax:

```
<Name>:ARRAY [<ll1>..<ul1>,<ll2>..<ul2>] OF <Type>
```

Ll1, ll2, ll3 identify the lower limit of the field range; ul1, ul2 and ul3 identify the upper limit. The range values must be integers.

Example:

```
Card_game: ARRAY [1..13, 1..4] OF INT;
```

ARRAYS Initialization

ATTENTION:
Squared brackets must be put around the initialization part.

Example for complete initialization of an array:

```
arr1 : ARRAY [1..5] OF INT := [1,2,3,4,5];
arr2 : ARRAY [1..2,3..4] OF INT := [1,3(7)]; (* Short for 1,7,7,7 *)
arr3 : ARRAY [1..2,2..3,3..4] OF INT := [2(0),4(4),2,3];
 (*Short for 0,0,4,4,4,4,2,3 *)
```

Example of the initialization of an array of a structure:

Structure definition:

```
TYPE STRUCT1
STRUCT
p1:int;
p2:int;
p3:dword;
END_STRUCT
END_TYPE
```

ARRAY initialization:

```
ARRAY[1..3] OF STRUCT1:= [(p1:=1,p2:=10,p3:=4723),(p1:=2,p2:=0,p3:=299),
(p1:=14,p2:=5,p3:=112)];
```

Example of the partial initialization of an ARRAY:

```
arr1 : ARRAY [1..10] OF INT := [1,2];
```

Elements to which no value is pre-assigned, are initialized with the default initial value of the basic type. In the example above, the elements anarray[6] to anarray[10] are therefore initialized with 0.

Accessing ARRAY Components

ARRAY components are accessed in a two-dimensional ARRAY using the following syntax:

```
<Name>[Index 1, Index 2]
```

Example:

```
Card_game [9,2]
```

Check Functions

In order to a proper access to ARRAY elements during runtime the function *CheckBounds* must be available to the application. Therefore add the object *POUs for implicit checks* to the application using the *Add Object* dialog. Mark the checkbox related to the type *CheckBounds*, choose an implementation language and confirm your settings with *Open*, whereon the function *CheckBound* will be opened in the editor corresponding to the implementation language selected. Independently of that choice the declaration part is preset and may not be modified except for adding further local variables. A proposal default implementation of the function that might be modified is given in the ST Editor.

This check function has to treat boundary violations by an appropriate method (for example by setting a detected error flag or changing the index). The function will be called implicitly as soon as a variable of type array is assigned.

ATTENTION: In order to maintain the check functionality, do not change the declaration part of an implicit check function.

Example for the use of function *CheckBounds*:

The default implementation of the check function is the following.

Declaration part:

```
// Implicitly generated code : DO NOT EDIT
FUNCTION CheckBounds : DINT
VAR_INPUT
index, lower, upper: DINT;
END_VAR
```

Implementation part:

```
// Implicitly generated code: Only an Implementation suggestion.
IF index < lower THEN
CheckBounds := lower;
ELSIF index > upper THEN
CheckBounds := upper;
ELSE
CheckBounds := index;
END_IF
```

When called the function gets the following input parameters:

- Index: Field element index
- Lower: The lower limit of the field range
- Upper: The upper limit of the field range

As long as the index is within the range, the return value is the index itself.

Otherwise -in correspondence to the range violation- either the upper or the lower limit of the filed range will be returned.

In the program beyond the upper limit of the ARRAY A is exceeded:

```
PROGRAM MAINPRG
VAR
a: ARRAY[0..7] OF BOOL;
b: INT:=10;
END_VAR
a[b]:=TRUE;
```

In this case the implicit call to the *CheckBounds* function preceding the assignment effects that the value of the index is changed from 10 (ten) into the upper limit 7 (seven). Therefore the value TRUE will be assigned to the element a[7] of the ARRAY. This is how attempted access outside the field range can be corrected via the function *CheckBounds*.

Structures

Structures are created as *DUT* (Data Type Unit) objects via the *Add Object* dialog.

They begin with the keywords TYPE and STRUCT and end with END_STRUCT and END_TYPE.

ATTENTION: TYPE in structure declarations must be followed by a “:”.

The syntax for structure declarations is as follows:

```
TYPE <Structure name>:
STRUCT
<Declaration of variables 1>
...
<Declaration of variables n>
END_STRUCT
END_TYPE
```

<Structure name> is a type that is recognized throughout the project and can be used like a standard data type.

Interlocking structures are allowed. The only restriction is that variables may not be assigned to addresses (the AT declaration is not allowed.).

Example for a structure definition named Polygonline:

```
TYPE Polygonline:
STRUCT
Start:ARRAY [1..2] OF INT;
Point1:ARRAY [1..2] OF INT;
Point2:ARRAY [1..2] OF INT;
Point3:ARRAY [1..2] OF INT;
Point4:ARRAY [1..2] OF INT;
End:ARRAY [1..2] OF INT;
END_STRUCT
END_TYPE
```

Initialization of Structures

Example for the initialization of a structure:

```
Poly_1:polygonline := ( Start:=[3,3], Point1 =[5,2], Point2:=[7,3],
Point3:=[8,5], Point4:=[5,7], End := [3,5]);
```

Initializations with variables are not possible. See an example of the initialization of an array of a structure under ARRAYS.

Access on Structure Components

You can gain access to structure components using the following syntax:

```
<Structure name>.<Component name>
```

So for the above mentioned example of the structure 'polygonline' you can access the component 'start' by Poly_1.Start.

Enumerations

An enumeration is an user-defined data type that is made up of a number of string constants. These constants are referred to as enumeration values.

Enumeration values are recognized globally in all areas of the project even if they were declared within a POU.

An enumeration is created as a *DUT* object via the *Add Object* dialog.

ATTENTION:
A local enumeration declaration is no longer possible except within TYPE.

Syntax:

```
TYPE <Identifier>:(<Enum_0> ,<Enum_1>, ...,<Enum_n>)|<Base data type>;
END_TYPE
```

A variable of type <Identifier> can take on one of the enumeration values <Enum_..> and will be initialized with the first one. These values are compatible with whole numbers which means that you can perform operations with them just as you would do with integer variables. You can assign a number x to the variable. If the enumeration values are not initialized with specific values within the declaration, counting will begin with 0. When initializing, make sure that the initial values are increasing within the row of components. The validity of the number will be checked at the time it is run.

Example:

Definition of two enumerations:

```
TYPE TRAFFIC_SIGNAL: (red, yellow, green:=10); (* The initial value for
each of the colors is red 0, yellow 1, green 10 *)
END_TYPE
```

Declaration:

```
TRAFFIC_SIGNAL1 : TRAFFIC_SIGNAL;
```

Use of enumeration TRAFFIC_SIGNAL in a POU:

```
TRAFFIC_SIGNAL1:=0; (* The value of the traffic signal is red *)
```

Extensions to the IEC 61131-3 Standard

The type name of enumerations can be used (as a scope operator) to disambiguate the access to an enumeration constant.

So it is possible to use the same constant in different enumerations.

Example:

Definition of two enumerations:

```
TYPE COLORS_1: (red, blue);
END_TYPE
TYPE COLORS_2: (green, blue, yellow);
END_TYPE
```

Use of enumeration value blue in a POU:

Declaration:

```
colorvar1 : COLORS_1;
colorvar2 : COLORS_2;
```

Implementation:

```
(* Possible: *)
colorvar1 := colors_1.blue;
colorvar2 := colors_2.blue;
(* Not Possible: *)
colorvar1 := blue;
colorvar2 := blue;
```

The base data type of the enumeration - which by default is INT - can be explicitly specified.

Example:

The base data type for enumeration BigEnum should be DINT:

```
TYPE BigEnum : (yellow, blue, green:=16#8000) DINT;
END_TYPE
```

Subrange Types

A subrange type is a user defined type whose range of values is only a subset of that of the basic data type. Notice the possibility of using implicit range boundary checks.

The declaration can be done in a DUT object but also a variable can be directly declared with a subrange type:

Syntax for the declaration as a DUT object:

```
TYPE <Name> : <Inttype> (<ug>..<og>) END_TYPE;
```

Syntax	Description
<Name>	Must be a valid IEC identifier.
<Inttype>	Is one of the data types SINT, USINT, INT, UINT, DINT, UDINT, BYTE, WORD, DWORD (LINT, ULINT, LWORD).
<ug>	Is a constant which must be compatible with the basic type and which sets the lower boundary of the range types. The lower boundary itself is included in this range.
<og>	Is a constant that must be compatible with the basic type, and sets the upper boundary of the range types. The upper boundary itself is included in this basic type.

Table 4-10. Subrange Types

Examples:

```
TYPE
   SubInt : INT (-4095..4095);
END_TYPE
```

Direct declaration of a variable with a subrange type:

```
VAR
   i : INT (-4095..4095);
   ui : UINT (0..10000);
END_VAR
```

If a value is assigned to a subrange type (in the declaration or in the implementation) but does not match this range (for example i:=5000 in the upper shown declaration example), an error message will be issued.

Check Functions

In order to check the range limits during runtime the functions CheckRangeSigned or CheckRangeUnsigned must be available to the application. Therefore add the object *POUs for implicit checks* to the application using the *Add Object* dialog. Mark the checkbox related to the type *CheckRangeSigned* or *CheckRangeUnsigned*, choose an implementation language and confirm your settings with *Open*, whereon the selected function will be opened in the editor corresponding to the implementation language selected. Independently of that choice the declaration part is preset and may not be modified except for adding further local variables. A proposal default implementation of the function that might be modified is given in the ST editor.

The purpose of this check function is the proper treatment of violations of the subrange (for example by setting a detected error flag or changing the value). The function will be called implicitly as soon as a variable of subrange type is assigned.

ATTENTION:
In order to maintain the check functionality do not change the declaration part of an implicit check function.

Example:

The assignment of a variable belonging to a signed subrange type entail an implicit call to CheckRangeSigned. The default implementation of that function trimming a value to the permissible range is provided as follows:

Declaration part:

```
// Implicitly generated code: DO NOT EDIT
FUNCTION CheckRangeSigned : DINT
VAR_INPUT
value, lower, upper: DINT;
END_VAR
```

Implementation part:

```
// Implicitly generated code: Only an Implementation suggestion
IF (value < lower) THEN
CheckRangeSigned := lower;
ELSIF(value > upper) THEN
CheckRangeSigned := upper;
ELSE
CheckRangeSigned := value;
END_IF
```

When called the function gets the following input parameters:

- Value: The value to be assigned to the range type
- Lower: The lower boundary of the range
- Upper: The upper boundary of the range.

As long as the assigned value is within the range, the output of the function is the value itself. Otherwise -in correspondence to the range violation- either the upper or the lower boundary of the range will be returned.

The assignment i:=10*y will now be replaced implicitly by:

```
i := CheckRangeSigned(10*y, -4095, 4095);
```

If y has the value 1000 for example, the variable i will not be assigned to 10*1000=10000 as provided by the original implementation, but to the upper boundary of the range, that is 4095.

The same applies to function CheckRangeUnsigned.

NOTE: If neither of the functions CheckRangeSigned or CheckRangeUnsigned is present, no type checking of subrange types occurs during runtime. In this case variable i could get any value between –32768 and 32767 at any time.

ATENTION:
The use of the functions CheckRangeSigned and CheckRangeUnsigned may result in an endless loop, for example if a subrange type is used as increment of a FOR loop that does not match the subrange.

Example of an endless loop:

```
VAR
   ui : UINT (0..10000);
END_VAR
```

```
FOR ui:=0 TO 10000 DO
...
END_FOR
```

The FOR loop will never be left as the check function prevents the variable “ui” to be assigned to values greater than10000.

Operators

IEC Operators and Norm-Extending Functions

MasterTool IEC XE supports all IEC operators. In contrast to the standard functions these operators are recognized implicitly throughout the project.

Besides the IEC operators also the following operators are supported which are not prescribed by the standard: ANDN, ORN, XORN, INDEXOF and SIZEOF (see **Arithmetic Operators**), ADR, BITADR and content operator (see **Address Operators**), some Scope Operators.

Operators are used like functions in a POU.

NOTE: At operations with floating point variables the result depends on the currently used target system.

See the following categories of operators:

- Assignment Operators: :=, MOVE
- Arithmetic Operators
- Bitstring Operators
- Bit-Shift Operators
- Selection Operators
- Comparison Operators
- Address Operators
- Calling Operators
- Type Conversion Functions
- Numeric Functions
- IEC extending Operators
- IEC extending Scope Operators

Arithmetic Operators

The following operators, prescribed by the IEC1131-3 standard, are available:

- ADD
- MUL
- SUB
- DIV
- MOD
- MOVE

There are also two norm-extending operators:

- SIZEOF
- INDEXOF

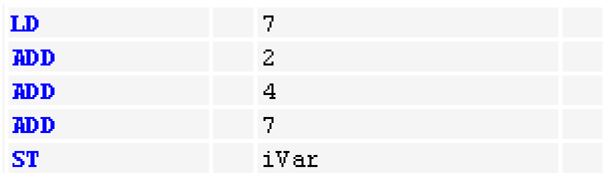
ADD

IEC Operator: Addition of variables.

Allowed types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

Two TIME variables can also be added together resulting in another time (for example, t#45s + t#50s = t#1m35s).

Example in IL:

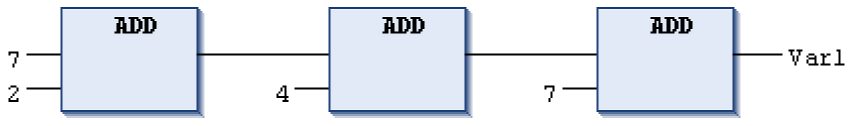


```
LD      7
ADD     2
ADD     4
ADD     7
ST      iVar
```

Example in ST:

```
var1 := 7+2+4+7;
```

Example in FBD:

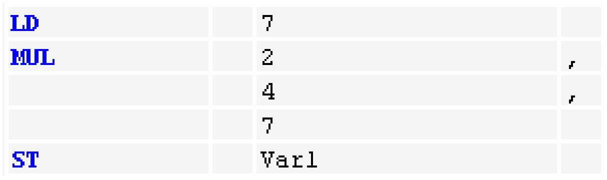


MUL

IEC Operator: Multiplication of variables.

Allowed types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

Example in IL:

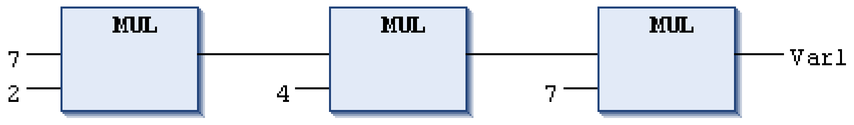


```
LD      7
MUL     2       ,
        4       ,
        7
ST      Var1
```

Example in ST:

```
var1 := 7*2*4*7;
```

Example in FBD:



SUB

IEC Operator: Subtraction of one variable from another one.

Allowed types s: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

A TIME variable may also be subtracted from another TIME variable resulting in third TIME type variable. Note that negative TIME values are undefined.

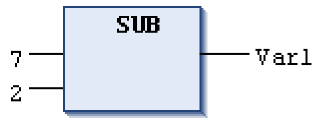
Example in IL:

LD	7	
SUB	2	
ST	Var1	

Example in ST:

```
var1 := 7-2;
```

Example in FBD:



DIV

IEC Operator: Division of one variable by another one.

Allowed types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT, REAL and LREAL.

Example in IL:

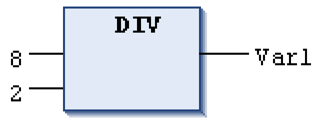
LD	8	
DIV	2	
ST	Var1	

Result in Var1 is 4.

Example in ST:

```
var1 := 8/2;
```

Example in FBD:



NOTE: Please notice, that different target systems may behave differently concerning a division by zero.

NOTE: Observe que a utilização do operador DIV com tipos de dados inteiros retorna apenas o quociente da divisão. Caso se queira retornar o resto da divisão o operador a ser utilizado é o MOD descrito a seguir.

Check Functions

In order to check the value of the divisor, for example in order to avoid a division by 0 you may make use of the provided check functions CheckDivInt, CheckDivLint, CheckDivReal and CheckDivLReal. After they have been included in the application each division occurring in the related code will provoke a preceding call to these functions. To include them in the application use the *Add Object* dialog. Therein choose the object *POUs for implicit checks*, mark the checkbox of a corresponding check function, select an implementation language and confirm your choice with *Open*. The selected function will be opened in the editor corresponding to the choice of the implementation function. Independently of this choice the declaration part of the functions is preset and must not be changed except for adding local variables. A default implementation of the functions that might be modified is available in ST.

See the following example for an implementation of the function CheckDivReal:

Declaration part:

```
// Implicitly generated code: DO NOT EDIT
FUNCTION CheckDivReal : REAL
VAR_INPUT
divisor:REAL;
END_VAR
```

Implementation part:

```
// Implicitly generated code: Only an suggestion for implementation.
IF divisor = 0 THEN
CheckDivReal:=1;
ELSE
CheckDivReal:=divisor;
END_IF;
```

The operator DIV uses the output of function CheckDivReal as divisor. In the following example a division by 0 is prohibited as the implicitly with 0 initialized value of the divisor D is changed to 1 (by CheckDivReal) prior to the execution of the division. Therefore the result of the division is 799.

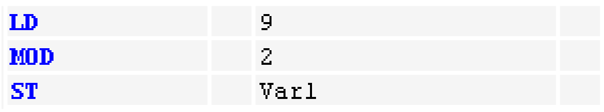
```
PROGRAM MAINPRG
VAR
erg:REAL;
v1:REAL:=799;
d:REAL;
END_VAR
erg:= v1 / d;
```

MOD

IEC Operator: Modulo Division of one variable by another one.

Allowed types: BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, UDINT, LINT, ULINT. The result of this function will be the remainder of the division. This result will be a whole number.

Example in IL:



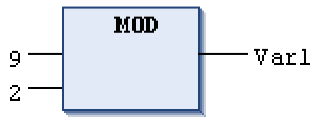
LD		9	
MOD		2	
ST		Var1	

Result in Var1 is 1.

Example in ST:

```
var1 := 9 MOD 2;
```

Example in FBD:



MOVE

IEC Operator: Assignment of a variable to another variable of an appropriate type.

As MOVE is available as a box in the graphic editors FBD, LD, CFC, there the (unlocking) EN/EN0 functionality can also be applied on a variable assignment.

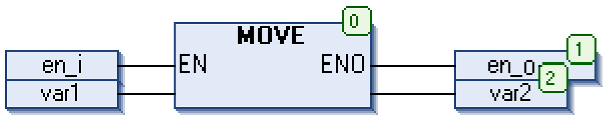


Figure 4-9. Example in CFC in Conjunction with the EN/EN0 Function

Only if en_i is TRUE, var1 will be assigned to var2.

Example in IL:

```
LD      var1
MOVE
ST      var2
```

Result is var2 gets value of var.

You get the same result with:

```
LD      var1
ST      var2
```

Example in ST:

```
ivar2 := MOVE(ivar1); (* The same result with: ivar2 := ivar1; *)
```

SIZEOF

This arithmetic operator is not prescribed by the standard IEC 61131-3.

It can be used to determine the number of bytes required by the given variable x.

The SIZEOF operator returns an unsigned value. The type of the return value will be adapted to the found size of variable x.

Return value of SIZEOF(x)	Data type of the constant implicitly used for the found size
0 <= size of x < 256	USINT
256 <= size of x < 65536	UINT
65536 <= size of x < 4294967296	UDINT
4294967296 <= size of x	ULINT

Table 4-11. SIZEOF Operator

Example in ST:

```
VAR
arr1:ARRAY[0..4] OF INT;
Var1:INT;
end_var
var1 := SIZEOF(arr1); (* d.h.: var1:=USINT#10; *)
```

Example in IL:

```
LD      arr1
SIZEOF
ST      Var1
```

Result is 10.

INDEXOF

This arithmetic operator is not prescribed by the standard IEC 61131-3.

Perform this function to find the internal index for a POU.

Example in ST:

```
var1 := INDEXOF(POU2);
```

Bitstring Operators

The following bitstring operators are available, matching the IEC1131-3 standard:

AND, OR, XOR, NOT.

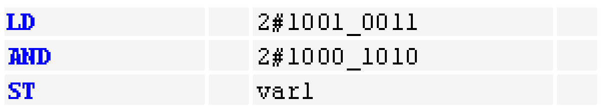
Bitstring operators compare the corresponding bits of two or several operands.

AND

IEC Bitstring Operator: Bitwise AND of bit operands. If the input bits each are 1, then the resulting bit will be “1”, otherwise “0”.

Allowed types: BOOL, BYTE, WORD, DWORD, LWORD.

Example in IL:



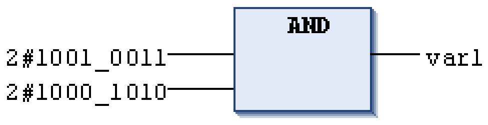
```
LD      2#1001_0011
AND     2#1000_1010
ST      var1
```

Result in Var1 is 2#1000_0010.

Example in ST:

```
VAR
Var1:BYTE;
END_VAR
var1 := 2#1001_0011 AND 2#1000_1010;
```

Example in FBD:

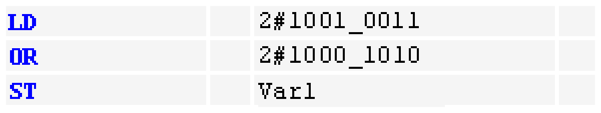


OR

IEC Bitstring Operator: Bitwise OR of bit operands. If at least one of the input bits is 1, the resulting bit will be “1”, otherwise “0”.

Allowed types: BOOL, BYTE, WORD or DWORD, LWORD.

Example in IL:



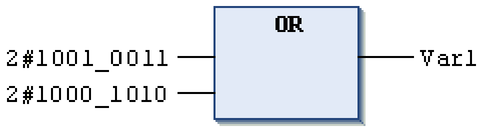
```
LD      2#1001_0011
OR      2#1000_1010
ST      Var1
```

Result in var1 is 2#1001_1011 (BYTE type).

Example in ST:

```
Var1 := 2#1001_0011 OR 2#1000_1010;
```

Example in FBD:



XOR

IEC Bitstring Operator: Bitwise XOR operation of bit operands. If only one of the input bits is 1, then the resulting bit will be "1"; if both or none are "1", the resulting bit will be "0".

Allowed types: BOOL, BYTE, WORD, DWORD, LWORD.

NOTE: Notice the behavior of the XOR function in extended form, that means if there are more than 2 inputs. The inputs will be checked in pairs and the particular results will then be compared again in pairs (this complies with the standard, but may not be expected by the user).

Example in IL:

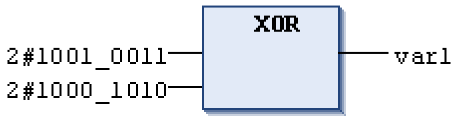
LD		2#1001_0011	
XOR		2#1000_1010	
ST		var1	

Result is 2#0001_1001 (BYTE type).

Example in ST:

```
Var1 := 2#1001_0011 XOR 2#1000_1010;
```

Example in FBD:



NOT

IEC Bitstring Operator IEC: Bitwise NOT operation of a bit operand. The resulting bit will be "1", if the corresponding input bit is "0" and vice versa.

Allowed types: BOOL, BYTE, WORD, DWORD, LWORD.

Example in IL:

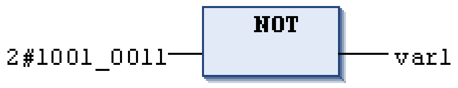
LD		2#1001_0011	
NOT			
ST		var1	

Result in Var1 is 2#0110_1100 (BYTE type).

Example in ST:

```
Var1 := NOT 2#1001_0011;
```

Example in FBD:



Bit-Shift Operators

The following bit-shift operators, matching the IEC1131-e standard, are available:

SHL, SHR, ROL and ROR.

SHL

IEC Operator: Bitwise left-shift of an operand.

```
erg:= SHL (in, n)
```

in: Operand to be shifted to the left.

n: Number of bits, by which in gets shifted to the left.

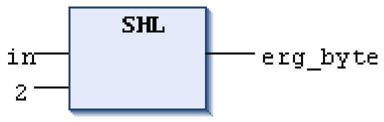
If n exceeds the data type width, BYTE, WORD, DWORD and LWORD operands will be filled with zeros, operands of signed data types, like for example INT, will get an arithmetic shift, that means they will be filled with the value of the topmost bit.

NOTES:
- Please note, that the amount of bits, which is noticed for the arithmetic operation, is pretended by the data type of the input variable ! If the input variable is a constant the smallest possible data type is noticed. The data type of the output variable has no effect at all on the arithmetic operation.
- See in the following example in hexadecimal notation that you get different results for erg_byte and erg_word depending on the data type of the input variable (BYTE or WORD), although the values of the input variables in_byte and in_word are the same.

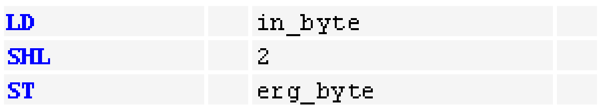
Example in ST:

```
PROGRAM shl_st
VAR
   in_byte : BYTE:=16#45;
   in_word : WORD:=16#45;
   erg_byte : BYTE;
   erg_word : WORD;
   n: BYTE :=2;
END_VAR
erg_byte:=SHL(in_byte,n); (* Result is 16#14 *)
erg_word:=SHL(in_word,n); (* Result is 16#0114 *)
```

Example in FBD:



Example in IL:



```
LD      in_byte
SHL     2
ST      erg_byte
```

SHR

IEC Operator: Bitwise right-shift of an operand.

```
erg:= SHR (in, n)
```

in: Operand to be shifted to the right.

n: Number of bits, by which in gets shifted to the right.

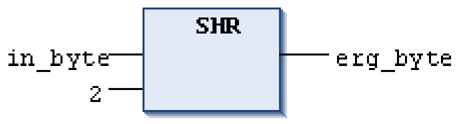
If n exceeds the data type width, operands of type BYTE, WORD, DWORD and LWORD will be filled with zeros, operands of signed data types, like for example INT, will get an arithmetic shift, that means they will be filled with the value of the topmost bit.

NOTES:
- Please note, that the amount of bits, which is noticed for the arithmetic operation, is pretended by the data type of the input variable ! If the input variable is a constant the smallest possible data type is noticed. The data type of the output variable has no effect at all on the arithmetic operation.
- See the following example in hexadecimal notation to notice the results of the arithmetic operation depending on the type of the input variable (BYTE or WORD).

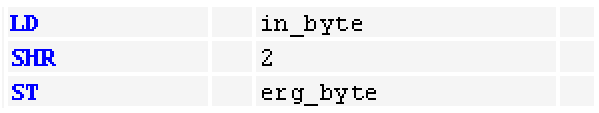
Example in ST:

```
PROGRAM shr_st
VAR
   in_byte : BYTE:=16#45;
   in_word : WORD:=16#45;
   erg_byte : BYTE;
   erg_word : WORD;
   n: BYTE :=2;
END_VAR
erg_byte:=SHR(in_byte,n); (* Result is 11 *)
erg_word:=SHR(in_word,n); (*Result is 0011 *)
```

Example in FBD:



Example in IL:



LD	in_byte	
SHR	2	
ST	erg_byte	

ROL

IEC Operator: Bitwise rotation of an operand to the left.

```
erg:= ROL (in, n)
```

Allowed data types: BYTE, WORD, DWORD, LWORD.

In will be shifted one bit position to the left n times while the bit that is furthest to the left will be reinserted from the right.

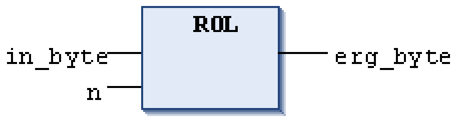
NOTES:
- Please note, that the amount of bits, which is noticed for the arithmetic operation, is pretended by the data type of the input variable ! If the input variable is a constant the smallest possible data type is noticed. The data type of the output variable has no effect at all on the arithmetic operation.
- See in the following example in hexadecimal notation that you get different results for "erg_byte" and "erg_word" depending on the data type of the input variable (BYTE or WORD), although the values of the input variables "in_byte" and "in_word" are the same.

Example in ST:

```
PROGRAM rol_st
VAR
   in_byte : BYTE:=16#45;
   in_word : WORD:=16#45;
   erg_byte : BYTE;
   erg_word : WORD;
   n: BYTE :=2;
END_VAR
erg_byte:=ROL(in_byte,n); (*Result is 16#15 *)
```

```
erg_word:=ROL(in_word,n); (* Result is 16#0114 *)
```

Example in FBD:



Example in IL:

LD	in_byte
ROL	n
ST	erg_byte

ROR

IEC Operator: Bitwise rotation of an operand to the right.

```
erg = ROR (in, n)
```

Allowed data types: BYTE, WORD, DWORD, LWORD.

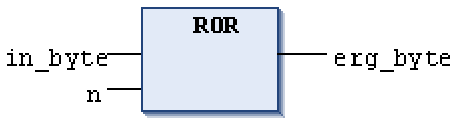
In will be shifted one bit position to the right n times while the bit that is furthest to the left will be reinserted from the left.

NOTES:
- Please note, that the amount of bits, which is noticed for the arithmetic operation, is pretended by the data type of the input variable ! If the input variable is a constant the smallest possible data type is noticed. The data type of the output variable has no effect at all on the arithmetic operation.
- See in the following example in hexadecimal notation that you get different results for "erg_byte" and "erg_word" depending on the data type of the input variable (BYTE or WORD), although the values of the input variables "in_byte" and "in_word" are the same.

Example in ST:

```
PROGRAM ror_st
VAR
    in_byte : BYTE:=16#45;
    in_word : WORD:=16#45;
    erg_byte : BYTE;
    erg_word : WORD;
    n: BYTE :=2;
END_VAR
erg_byte:=ROR(in_byte,n); (* Result is 16#51 *)
erg_word:=ROR(in_word,n); (* Result is 16#4011 *)
```

Example in FBD:



Example in IL:

LD	in_byte
ROR	n
ST	erg_byte

Selection Operators

All selection operations can also be performed with variables. For purposes of clarity we will limit our examples to the following which use constants as operators: SEL, MAX, MIN, LIMIT, MUX.

SEL

IEC Selection Operator: Binary Selection. G determines whether IN0 or IN1 is assigned to OUT.

```
OUT := SEL(G, IN0, IN1)
OUT := IN0; (*If G=FALSE*)
OUT := IN1; (*If G=TRUE*)
```

Allowed data types:

IN0, IN1 and OUT: any type;

G: BOOL;

Example in IL:

LD	TRUE	
SEL	3	,
	4	
ST	Var1	

Result is 4.

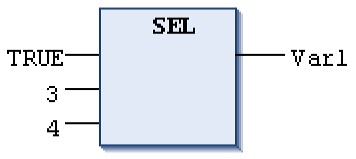
LD	FALSE	
SEL	3	,
	4	
ST	Var1	

Result is 3.

Example in ST:

```
Var1:=SEL(TRUE,3,4); (* Result is 4 *)
```

Example in FBD:



NOTE: Note that an expression occurring ahead of IN1 or IN2 will not be processed if IN0 is TRUE.

MAX

IEC Selection Operator: Maximum function. Returns the greater of the two values.

```
OUT := MAX(IN0, IN1);
```

IN0, IN1 e OUT and OUT can be any type of variable.

Example in IL:

LD	90	
MAX	30	
MAX	40	
MAX	77	
ST	Var1	

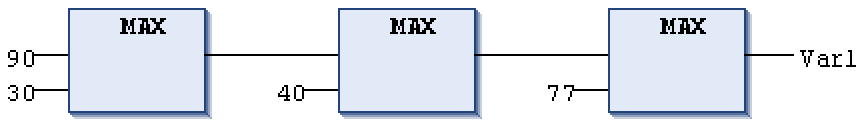
Result is 90.

Example in ST:

```
Var1:=MAX(30,40); (*Result is 40 *)
```

```
Var1:=MAX(40,MAX(90,30)); (*Result is 90 *)
```

Example in FBD:



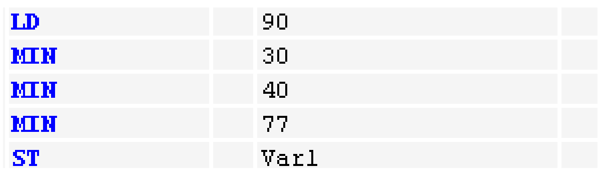
MIN

IEC Selection Operator: Minimum function. Returns the lesser of the two values.

```
OUT := MIN(IN0, IN1)
```

IN0, IN1 and OUT can be any type of variable.

Example in IL:



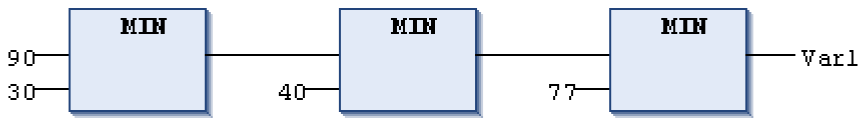
```
LD    90
MIN   30
MIN   40
MIN   77
ST    Var1
```

Result is 30.

Example in ST:

```
Var1:=MIN(90,30); (*Result is 30 *);
Var1:=MIN(MIN(90,30),40); (*Result is 30 *);
```

Example in FBD:



LIMIT

IEC Selection Operator: limits.

```
OUT := LIMIT(Min, IN, Max)
```

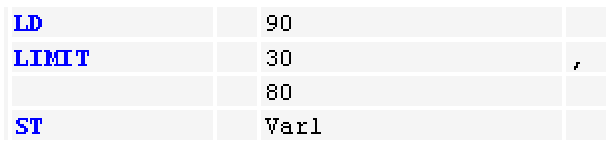
Which can be written as:

```
OUT := MIN (MAX (IN, Min), Max);
```

Max is the upper and Min the lower limit for the result. Should the value IN exceed the upper limit Max, LIMIT will return Max. Should IN fall below Min, the result will be Min.

IN and OUT can be any type of variable.

Example in IL:



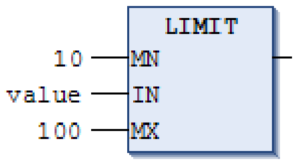
```
LD     90
LIMIT  30    ,
       80
ST     Var1
```

Result is 80.

Example in ST:

```
Var1:=LIMIT(30,90,80); (*Result is 80 *);
```

Example in FBD:



MUX

IEC Selection Operator: multiplexer operator.

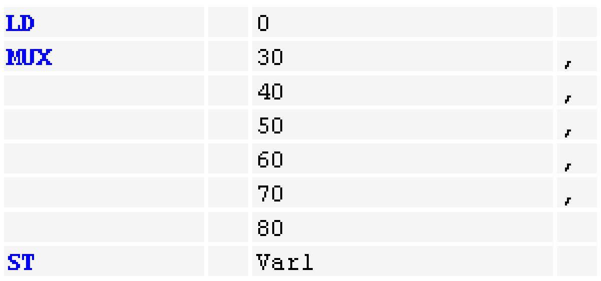
```
OUT := MUX(K, IN0,...,INn)
```

Which can be written as:

```
OUT := INK;
```

IN0, ..., INn and OUT can be any type of variable. K must be BYTE, WORD, DWORD, LWORD, SINT, USINT, INT, UINT, DINT, LINT, ULINT or UDINT. MUX selects the Kth value from among a group of values.

Example in IL:



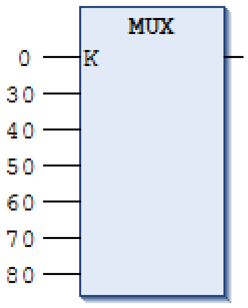
LD	0	
MUX	30	,
	40	,
	50	,
	60	,
	70	,
	80	
ST	Var1	

Result is 30.

Example in ST:

```
Var1:=MUX(0,30,40,50,60,70,80); (* Result is 30 *);
```

Example in FBD:



Result is 30.

NOTE: Note that an expression occurring ahead of an input other than INK will not be processed to save run time. Only in simulation mode all expressions will be executed.

Comparison Operators

The following operators, matching the IEC1131-3 standard are available:

GT, LT, LE, GE, EQ, NE

These are Boolean operators, each comparing two inputs (first and second operand).

GT

IEC Comparison Operator: Greater than.

A Boolean operator which returns the value TRUE when the value of the first operand is greater than that of the second. The operands can be of any basic data type.

Example in IL:

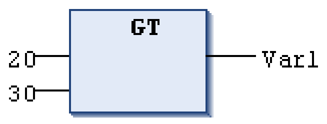
```
LD      20
GT      30
ST      Var1
```

Result is FALSE.

Example in ST:

```
VAR1 := 20 > 30 > 40 > 50 > 60 > 70;
```

Example in FBD:



LT

IEC Comparison Operator: Less than.

A Boolean operator that returns the value TRUE when the value of the first operand is less than that of the second. The operands can be of any basic data type.

Example in IL:

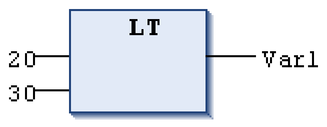
```
LD      20
LT      30
ST      Var1
```

Result is TRUE.

Example in ST:

```
VAR1 := 20 < 30;
```

Example in FBD:



LE

IEC Comparison Operator: Less than or equal to.

A Boolean operator that returns the value TRUE when the value of the first operand is less than or equal to that of the second. The operands can be of any basic data type.

Example in IL:

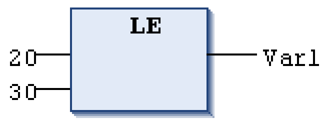
```
LD      20
LE      30
ST      Var1
```

Result is TRUE.

Example in ST:

```
VAR1 := 20 <= 30;
```

Example in FBD:



GE

IEC Comparison Operator: Greater than or equal to.

A Boolean operator that returns the value TRUE when the value of the first operand is greater than or equal to that of the second. The operands can be of any basic data type.

Example in IL:

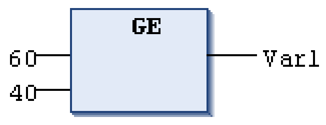
LD	60	
GE	40	
ST	Var1	

Result is TRUE.

Example in ST:

```
VAR1 := 60 >= 40;
```

Example in FBD:



EQ

IEC Comparison Operator: Equal to.

A Boolean operator that returns the value TRUE when the operands are equal. The operands can be of any basic data type.

Example in IL:

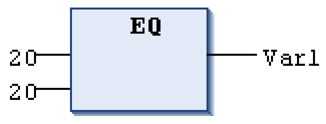
LD	40	
EQ	40	
ST	Var1	

Result is TRUE.

Example in ST:

```
VAR1 := 40 = 40;
```

Example in FBD:



NE

IEC Comparison Operator: Not equal to.

A Boolean operator that returns that value TRUE when the operands are not equal. The operands can be of any basic data type.

Example in IL:

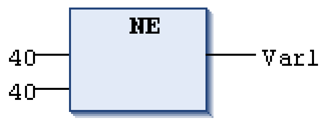
LD	40	
NE	40	
ST	Var1	

Result is FALSE.

Example in ST:

```
VAR1 := 40 <> 40;
```

Example in FBD:



Address Operators

ADR and BITADR and the content operator "^" are norm-extending address operators available in MasterTool IEC XE.

ADR

This address operator is not prescribed by the standard IEC 61131-3.

ADR returns the address of its argument in a DWORD. This address can be sent to manufacturing functions to be treated as a pointer or it can be assigned to a pointer within the project.

NOTE: The ADR-Operator can be used with function names, program names, function block names and method names, thus replacing the INDEXOF operator.

See in this context 'Function pointers'. Notice anyway that function pointers can be passed to external libraries, but there is no possibility to call a function pointer within MasterTool IEC XE. In order to enable a system call (runtime system) the respective property (category *Build*) must be set for the function object. See **Function Pointers**.

Example in ST:

```
dwVar:=ADR(bVAR);
```

Example in IL:

LD	bVar	
ADR		
ST	dwVar	

NOTE: After an Online Change there might be changes concerning the data on some addresses. Notice this in this case of using pointers on addresses.

BITADR

This address operator is not prescribed by the standard IEC 61131-3.

BITADR returns the bit offset within the segment in a DWORD. Notice that the offset value depends on whether the option byte addressing in the target settings is activated or not.

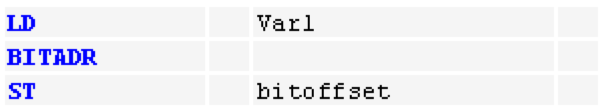
```
VAR
var1 AT %IX2.3:BOOL;
```

```
bitoffset: DWORD;
END_VAR
```

Example in ST:

```
bitoffset:=BITADR(var1); (*Result: 16#80000013 *)
```

Example in IL:



```
LD      Var1
BITADR
ST      bitoffset
```

NOTE: After an Online Change there might be changes concerning the data on some addresses. Please notice this in this case of using pointers on addresses.

Content Operator

This address operator is not prescribed by the standard IEC 61131-3.IEC. A pointer can be dereferenced by adding the content operator “^” after the pointer identifier.

Example in ST:

```
pt:POINTER TO INT;
var_int1:INT;
var_int2:INT;
pt := ADR(var_int1);
var_int2:=pt^;
```

NOTE: After an Online Change there might be changes concerning the data on some addresses. Please notice this in this case of using pointers on addresses.

Calling Operator

CAL

IEC Operator for calling a function block or a program.

Use CAL in IL to call up a function block instance. The variables that will serve as the input variables are placed in parentheses right after the name of the function block instance.

Example:

Calling up the instance “Inst” from a function block where input variables Par1 and Par2 are 0 and TRUE respectively.

```
CAL INST(PAR1 := 0, PAR2 := TRUE)
```

Type Conversion Functions

It is forbidden to implicitly convert from a "larger" type to a "smaller" type (for example from INT to BYTE or from DINT to WORD. One can basically convert from any elementary type to any other elementary type.

Syntax:

```
<elem.Typ1>_TO_<elem.Typ2>
```

Please notice that at ...TO_STRING conversions the string is generated left-justified. If it is defined to short, it will be cut from the right side.

The following type conversions are supported:

- BOOL_TO Conversions
- TO_BOOL Conversions
- Conversion between integral number types

- REAL_TO-/ LREAL_TO Conversions
- TIME_TO/TIME_OF_DAY Conversions
- DATE_TO/DT_TO Conversions
- STRING_TO Conversions
- TRUNC (conversion to DINT)
- TRUNC_INT
- ANY_NUM_TO_<numeric datatype>
- ANY_TO_<any datatype>Conversions BOOL_TO

BOOL_TO Conversions

IEC Operator: Converting from type BOOL to any other type.

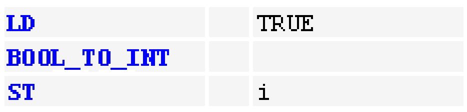
Syntax for a BOOL_TO conversion operator:

```
BOOL_TO_<Data type>
```

For number types the result is “1”, when the operand is TRUE, and “0”, when the operand is FALSE.

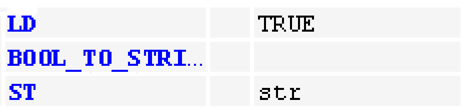
For the STRING type the result is ‚TRUE‚ when the operand is TRUE or FALSE when the operand is FALSE.

Examples in IL:



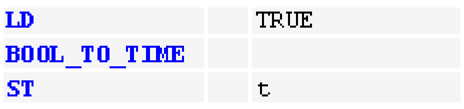
```
LD            TRUE
BOOL_TO_INT
ST            i
```

Result is 1.



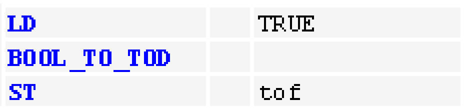
```
LD            TRUE
BOOL_TO_STRI...
ST            str
```

Result is TRUE.



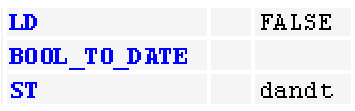
```
LD            TRUE
BOOL_TO_TIME
ST            t
```

Result is T#1ms.



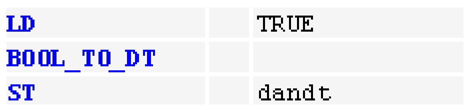
```
LD            TRUE
BOOL_TO_TOD
ST            tof
```

Result is TOD#00:00:00.001.



```
LD            FALSE
BOOL_TO_DATE
ST            dandt
```

Result is D#1970-01-01.



```
LD            TRUE
BOOL_TO_DT
ST            dandt
```

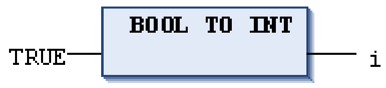
Result is DT#1970-01-01-00:00:01.

Examples in ST:

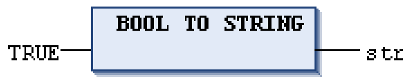
```
i:=BOOL_TO_INT(TRUE);       (* Result is 1 *)
```

```
str:=BOOL_TO_STRING(TRUE); (* Result is "TRUE" *)
t:=BOOL_TO_TIME(TRUE);     (* Result is T#1ms *)
tof:=BOOL_TO_TOD(TRUE);    (* Result is TOD#00:00:00.001 *)
dat:=BOOL_TO_DATE(FALSE);  (* Result is D#1970 *)
dandt:=BOOL_TO_DT(TRUE);   (* Result is DT#1970-01-01-00:00:01 *)
```

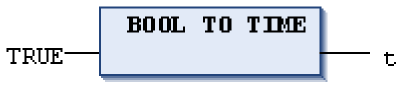
Examples in FBD:



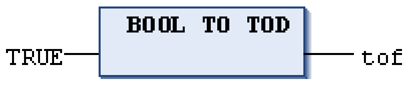
Result is 1.



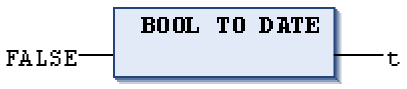
Result is TRUE.



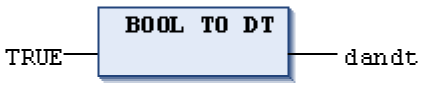
Result is T#1ms.



Result is TOD#00:00:00.001.



Result is D#1970-01-01.



Result is DT#1970-01-01-00:00:01.

TO_BOOL Conversions

IEC Operator: Converting from another variable type to BOOL

Syntax for a TO_BOOL conversion operator:

```
<Data type>_TO_BOOL
```

The result is TRUE when the operand is not equal to 0. The result is FALSE when the operand is equal to 0.

The result is TRUE for STRING type variables when the operand is "TRUE", otherwise the result is FALSE.

Examples in IL:

LD	213
BYTE_TO_BOOL	
ST	b

Result is TRUE.

LD	0
INT_TO_BOOL	
ST	b

Result is FALSE.

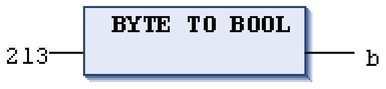
LD	T#5ms
TIME_TO_BOOL	
ST	b

Result is TRUE.

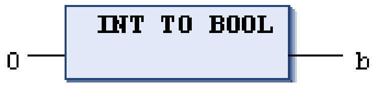
LD	'TRUE'
STRING_TO_BOOL	
ST	b

Result is TRUE.

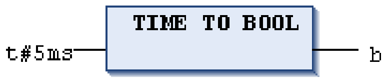
Examples in FBD:



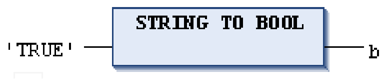
Result is TRUE.



Result is FALSE.



Result is TRUE.



Result is TRUE.

Examples in ST:

```
b := BYTE_TO_BOOL(2#11010101);    (* Result is TRUE *)
b := INT_TO_BOOL(0);              (* Result is FALSE *)
b := TIME_TO_BOOL(T#5ms);         (* Result is TRUE *)
b := STRING_TO_BOOL(TRUE);        (* Result is TRUE *)
```

Conversion between Integral Data Types

Conversion from an integral number type to another number type.

Syntax for the conversion operator:

```
<INT data type>_TO_<INT data type>
```

When you perform a type conversion from a larger to a smaller type, you risk losing some information. If the number you are converting exceeds the range limit, the first bytes for the number will be ignored.

Example in ST:

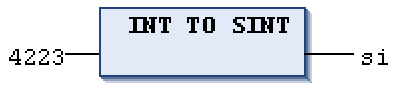
```
si := INT_TO_SINT(4223); (* Result is 127 *)
```

If you save the integer 4223 (16#107f represented in hexadecimal) as a SINT variable, it will appear as 127 (16#7f represented in hexadecimal).

Example in IL:

```
LD              4223
INT_TO_SINT
ST              si
```

Example in FBD:



REAL_TO / LREAL_TO Conversions

IEC Operator: Converting from the variable type REAL or LREAL to a different type.

The value will be rounded up or down to the nearest whole number and converted into the new variable type. Exceptions to this are the variable types STRING, BOOL, REAL and LREAL.

> NOTE: If a REAL or LREAL is converted to SINT, USINT, INT, UINT, DINT, UDINT, LINT or ULINT and the value of the real number is out of the value range of that integer, the result will be undefined and will depend on the target system. Even an exception is possible in this case! In order to get target-independent code, handle any range exceedance by the application. If the REAL/LREAL number is within the integer value range, the conversion will work on all systems in the same way.

Notice at a conversion to type STRING that the total number of digits is limited to 16. If the REAL/LREAL-number has more digits, then the sixteenth will be rounded. If the length of the STRING is defined to short, it will be cut beginning from the right end.

When you perform a type conversion from a larger to a smaller type, you risk losing some information.

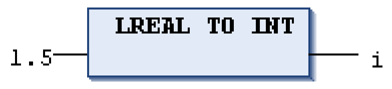
Example in ST:

```
i := REAL_TO_INT(1.5); (* Result is 2 *)
j := REAL_TO_INT(1.4); (* Result is 1 *)
i := REAL_TO_INT(-1.5); (* Result is -2 *)
j := REAL_TO_INT(-1.4); (* Result is -1 *)
```

Example in IL:

```
LD              2.7
REAL_TO_INT
ST              i
```

Example in FBD:



TIME_TO/TIME_OF_DAY Conversions

IEC Operator: from the variable type TIME or TIME_OF_DAY to a different type.

Syntax for the conversion operator:

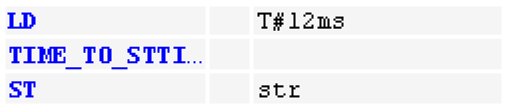
```
<TIME data type>_TO_<Data type>
```

The time will be stored internally in a DWORD in milliseconds (beginning with 12:00 A.M. for the TIME_OF_DAY variable). This value will then be converted.

When you perform a type conversion from a larger to a smaller type, you risk losing some information.

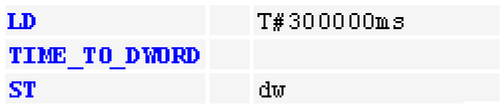
For the STRING type variable, the result is a time constant.

Examples in IL:



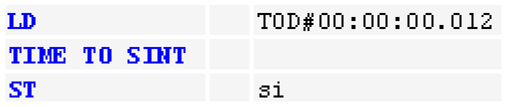
```
LD              T#12ms
TIME_TO_STTI...
ST              str
```

Result is T#12ms.



```
LD              T#300000ms
TIME_TO_DWORD
ST              dw
```

Result is 300000.



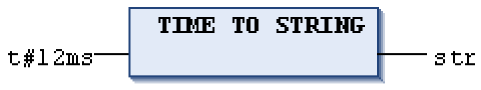
```
LD              TOD#00:00:00.012
TIME TO SINT
ST              si
```

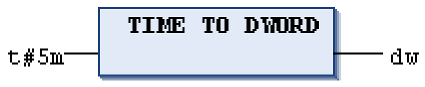
Result is 12.

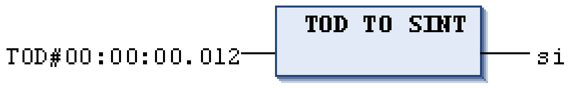
Examples in ST:

```
str :=TIME_TO_STRING(T#12ms);           (* Result is T#12ms *)
dw:=TIME_TO_DWORD(T#5m);                (* Result is 300000 *)
si:=TOD_TO_SINT(TOD#00:00:00.012);      (* Result is 12 *)
```

Examples in FBD:







DATE_TO/DT_TO Conversions

IEC Operator: Converting from the variable type DATE or DATE_AND_TIME to a different type.

Syntax for the conversion operator:

```
<DATE data type>_TO_<Data type>
```

The date will be stored internally in a DWORD in seconds since Jan. 1, 1970. This value will then be converted.

When you perform a type conversion from a larger to a smaller type, you risk losing some information.

For STRING type variables, the result is the date constant.

Examples in IL:

LD	D#1970-01-01
DATE_TO_BOOL	
ST	b

Result is FALSE.

LD	D#1970-01-01
DATE_TO_INT	
ST	i

Result is 29952.

LD	D#1970-01-15-05:05:.
DATE_TO_BYTE	
ST	byt

Result is 129.

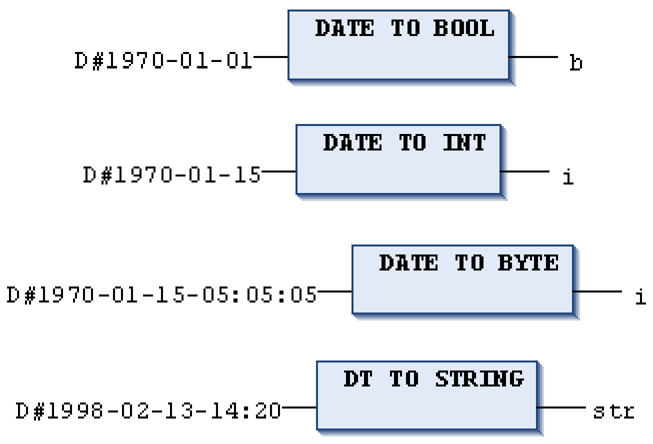
LD	D#1998-02-13-14:20
DATE_TO_STRI...	
ST	str

Result is DT#1998-02-13-14:20.

Examples in ST:

```
b :=DATE_TO_BOOL(D#1970-01-01);              (* Result is FALSE *)
i :=DATE_TO_INT(D#1970-01-15);               (* Result is 29952 *)
byt :=DT_TO_BYTE(DT#1970-01-15-05:05:05);    (* Result is 129 *)
str:=DT_TO_STRING(DT#1998-02-13-14:20);      (* Result is DT#1998-02-13-
14:20 *)
```

Examples in FBD:



STRING_TO Conversions

IEC Operator: Converting from the variable type STRING to a different type.

The conversion works according to the standard C compilation mechanism: STRING to INT and then INT to BYTE. The higher byte will be cut, thus only values of 0-255 result.

This way it is possible on most processors to generate optimal code because the conversion can be performed by a single machine instruction.

Syntax for the conversion operator:

```
STRING_TO_<Data type>
```

The operand from the STRING type variable must contain a value that is valid in the target variable type, otherwise the result will be “0”.

```
LD              'TRUE'
STRING_TO_BOOL
ST              b
```

Result is TRUE.

```
LD              'abc34'
STRING_TO_WORD
ST              w
```

Result is 0.

```
LD              't#117ms'
STRING_TO_TIME
ST              t
```

Result is T#117ms.

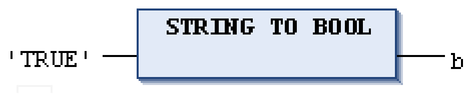
```
LD              '500'
STRING_TO_BYTE
ST              bv
```

Result is 244.

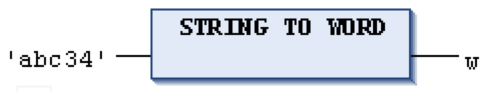
Examples in ST:

```
b :=STRING_TO_BOOL(TRUE);          (* Result is TRUE *)
w :=STRING_TO_WORD(abc34);         (* Result is 0 *)
t :=STRING_TO_TIME(T#127ms);       (* Result is T#127ms *)
bv :=STRING:TO_BYTE(500);          (* Result is 244 *)
```

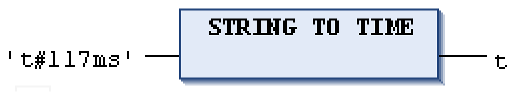
Examples in FBD:



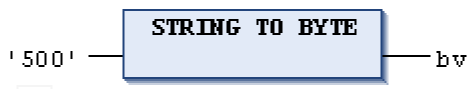
Result is TRUE.



Result is 0.



Result is T#127ms.



Result is 244.

TRUNC

IEC Operator: Converting from REAL to DINT. The whole number portion of the value will be used.

NOTE: The TRUNC operator converts from REAL to INT. For this reason when importing a previous version project TRUNC automatically will be replaced by TRUNC_INT.

When you perform a type conversion from a larger to a smaller type, you risk losing some information.

Example in IL:

```
LD          1.9
TRUNC
ST          diVar
```

Examples in ST:

```
diVar:=TRUNC(1.9);    (* Result is 1 *)
diVar:=TRUNC(-1.4);   (* Result is -1 *)
```

TRUNC_INT

IEC Operator: Converting from REAL to DINT. The whole number portion of the value will be used.

NOTE: TRUNC_INT corresponds to traditional TRUNC operator.

When you perform a type conversion from a larger to a smaller type, you risk losing some information.

Example in IL:

```
LD          1.9
TRUNC_INT
ST          iVar
```

Examples in ST:

```
iVar:=TRUNC_INT(1.9);      (* Result is 1 *)
iVar:=TRUNC_INT(-1.4);     (* Result is -1 *)
```

ANY...TO Conversions

An IEC Operator from any data type, and more specifically from any numeric data type to another data type. A reasonable choice of data types is assumed.

Syntax:

```
ANY_NUM_TO_<Numeric data type>
ANY_TO_<Any data type>
```

Example:

Conversion from a variable of data type REAL to INT:

```
re : REAL := 1.234;
i : INT := ANY_TO_INT(re)
```

Numeric Functions

The following numeric IEC operators are available:

ABS, SQRT, LN, LOG, EXP, SIN, COS, TAN, ASIN, ACOS, ATAN and EXPT.

ABS

IEC Operator: Returns the absolute value of a number. ABS(-2) returns 2.

In- and output can be of any numeric basic data type.

Example in IL:

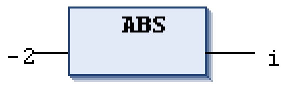
```
LD      -2
ABS
ST      i
```

Result in "i" is 2.

Example in ST:

`i:=ABS(-2);`

Example in FBD:



SQRT

IEC Operator: Returns the square root of a number.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

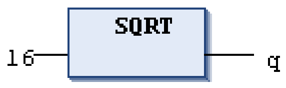
```
LD      16
SQRT
ST      q
```

Result in "q" is 4.

Example in ST:

`q:=SQRT(16);`

Example in FBD:



LN

IEC Operator: Returns the natural logarithm of a number.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

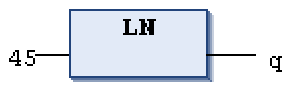
```
LD      45
LN
ST      q
```

Result in "q" is 3,80666.

Example in ST:

`q:=LN(45);`

Example in FBD:



LOG

IEC Operator: Returns the logarithm of a number in base 10.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

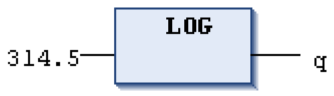
LD	314.5	
LOG		
ST	q	

Result in "q" is 2,49762.

Example in ST:

```
q:=LOG(314.5);
```

Example in FBD:



EXP

IEC Operator: Returns the exponential function.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

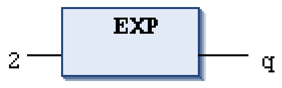
LD	2	
EXP		
ST	q	

Result in "q" is 7,389056.

Example in ST:

```
q:=EXP(2);
```

Example in FBD:

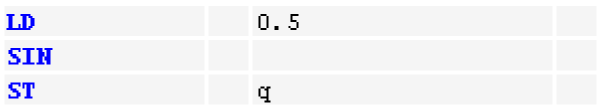


SIN

IEC Operator: Returns the sine of a number.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:



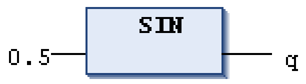
LD	0.5	
SIN		
ST	q	

Result in "q" is 0,479426.

Example in ST:

```
q:=SIN(0.5);
```

Example in FBD:

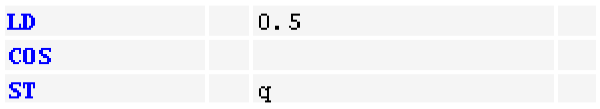


COS

IEC Operator: Returns the cosine of number. The value is calculated in arch minutes.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:



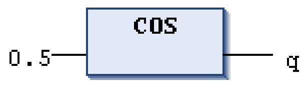
LD	0.5	
COS		
ST	q	

Result in "q" is 0.877583.

Example in ST:

```
q:=COS(0.5);
```

Example in FBD:

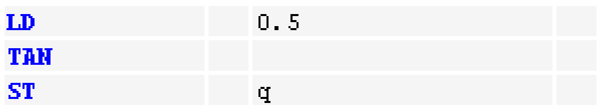


TAN

IEC Operator: Returns the tangent of a number. The value is calculated in arch minutes.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:



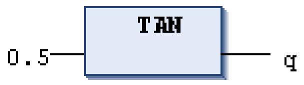
LD	0.5	
TAN		
ST	q	

Result in “q” is 0,546302.

Example in ST:

```
q:=TAN(0.5);
```

Example in FBD:



ASIN

IEC Operator: Returns the arc sine (inverse function of sine) of a number.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

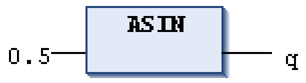
LD	0.5	
ASIN		
ST	q	

Result in “q” is 0,523599.

Example in ST:

```
q:=ASIN(0.5);
```

Example in FBD:



ACOS

IEC Operator: Returns the arc cosine (inverse function of cosine) of a number. The value is calculated in arch minutes.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

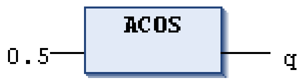
LD	0.5	
ACOS		
ST	q	

Result in "q" is 1,0472.

Example in ST:

```
q:=ACOS(0.5);
```

Example in FBD:



ATAN

IEC Operator: Returns the arc tangent (inverse function of tangent) of a number. The value is calculated in arch minutes.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

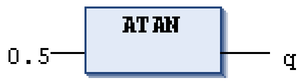
LD	0.5	
ATAN		
ST	q	

Result in "q" is 0,463648.

Example in ST:

```
q:=ATAN(0.5);
```

Example in FBD:



EXPT

IEC Operator: Exponentiation of a variable with another variable.

OUT = IN1IN2.

The input variable can be of any numeric basic data type, the output variable must be type REAL or LREAL.

Example in IL:

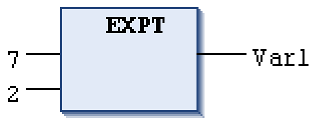
LD	7	
EXPT	2	
ST	Var1	

Result is 49.

Example in ST:

```
var1 := EXPT(7,2);
```

Example in FBD:



IEC Extending Operators

Additionally to IEC operators, the MasterTool IEC XE supports the operator "__ISVALIDREF."

__ISVALIDREF

This operator is not required by IEC 61131-3. It can be used to verify if the reference points to a valid value. To obtain information about its usage and examples, see **Check For Valid References**.

Norm-Extending Scope Operators

In extension to the IEC operators there are several possibilities to disambiguate the access to variables or modules if the variables or module name is used multiple times within the scope of a project. To define the respective namespace the following scope operators can be used:

- "." (global scope operator)
- "<global variable list name>"
- "<library name>"
- "<enumeration name>"

Global Scope Operator

Scope Operator: Extension to the IEC 61131-3 standard.

An instance path starting with "." opens a global scope (namespace). So, if there is a local variable with the same name "<var name>" as a global variable ".<var name>" will refer to the global variable.

Global Variable List Name

Scope Operator: Extension to the IEC 61131-3 standard.

The name of a global variable list can be used as a namespace for the variables enclosed in this list. Thus it is possible to declare variables with identical names in different global variable lists and by preceding the variable name by "<globalvariable list name>" it is possible to access the desired one.

Example:

The global variables lists globlist1 and globlist2 each contain a variable named varx. In the following line varx out of globlist2 is copied to varx in globlist1:

```
globlist1.varx := globlist2.varx; (* In this code line *)
```

If a variable name declared in more than one global variable lists is referenced without the global variable list name as a preceding operator, an error message will be output.

Library Namespace

Scope Operator: Extension to the IEC 61131-3 standard.

The library namespace can be used to explicitly access the library components. Example: If a library which is included in a project, contains a module "fun" and there is also a POU "fun" defined locally in the project, then add the "namespace" of the library can be added to the module-name in order to make the access unique. The syntax is: "<namespace>.<module name>", for example "lib1.fun".

By default the namespace of a library is identic with the library name, however you can define another one either in the *Project Information* when creating a library project, or later in the *Properties* dialog for a library.

Example:

There is a function fun1 in library lib. There is also a function fun1 declared in the project. By default the namespace of library lib is named lib:

```
res1 := fun(in := 12); // call of the project function fun.
res2 := lib.fun(in := 12); // call of the library function fun.
```

Enumeration Name

Scope Operator: Extension to the IEC 61131-3 standard.

The type name of enumerations can be used to disambiguate the access to an enumeration constant. In this case <enumeration name>. precedes the constant name. So it is possible to use the same constant in different enumerations.

Example:

The constant Blue is a component of enumeration "Colors" as well as of enumeration Feelings.

```
color := Colors.Blue; // Access to enum value Blue in type Colors.
feeling := Feelings.Blue; // Access to enum value Blue in type Feelings.
```

Operands

The following can be used as an operand:

- Constant (BOOL, TIME, DATE, TIME_OF_DAY, DATE_AND_TIME, number, REAL/LREAL, STRING, Typed Literals)
- Variable
- Addresses
- Functions

Constants

BOOL Constants

BOOL constants are the logical values TRUE and FALSE.

See also: **BOOL** (**Standard Data Types**).

TIME Constants

TIME constants are generally used to operate the standard timer modules. Besides the time constant TIME, which is of size 32 Bit and matching the IEC 61131-3 standard, LTIME is supported as an extension to the standard as time base for high resolution timers. LTIME is of size 64 Bit and resolution nanoseconds.

Syntax for TIME constant:

```
t#<Time declaration>
```

Instead of "t#" also the following can be used: "T#", "time", "TIME".

The time declaration can include the following time units. These must be used in the following sequence, but it is not required to use all of them:

- “d”: days
- “h”: hours
- “m”: minutes
- “s”: seconds
- “ms”: milliseconds

Examples of correct TIME constants in a ST assignment:

```
TIME1 := T#14ms;
TIME1 := T#100S12ms; (*The highest component may be allowed to exceed its
limit *)
TIME1 := t#12h34m15s;
```

The following would be incorrect:

```
TIME1 := t#5m68s;     (* Limit exceeded in a lower component *)
TIME1 := 15ms;        (* T# is missing *)
TIME1 := t#4ms13d;    (* Incorrect order of entries *)
```

Syntax for LTIME constant:

```
LTIME#<Time declaration>
```

The time declaration can include the time units as used with the TIME constant (see above) and additionally:

- “us” : microseconds
- “ns” : nanoseconds

Examples of correct LTIME constants in a ST assignment:

```
LTIME1 := LTIME#1000d15h23m12s34ms2us44ns
LTIME1 := LTIME#3445343m3424732874823ns
```

See also: **Time Data Types**.

DATE Constants

These constants can be used to enter dates.

Syntax**:**

```
d#<Date declaration>
```

Instead of “d#” also the following can be used: “D#”, “date#”, “DATE#”.

The date declaration is to be entered in format <year-month-day>.

DATE values are internally handled as DWORD values, containing the time span in seconds since 01.01.1970, 00:00 clock.

Examples:

```
DATE#1996-05-06
d#1972-03-29
```

See also: **Time Data Types**.

TIME_OF_DAY Constants

Use this type of constant to store times of the day.

```
Syntax:
tod#<Time declaration>
```

Instead of “tod#” also the following can be used: “TOD#”, “time_of_day”, “TIME_OF_DAY”.

The time declaration is to be entered in format <hour:minute:second>.

Seconds can be entered as real numbers, that is also fractions of a second can be specified.

TIME_OF_DAY values are internally handled as DWORD values, containing the time span in milliseconds since 00:00 clock.

Examples:

```
TIME_OF_DAY#15:36:30.123
tod#00:00:00
```

See also: **Time Data Types**.

DATE_AND_TIME Constants

DATE constants and TIME_OF_DAY constants can also be combined to form so-called DATE_AND_TIME constants.

Syntax:

```
dt#< date and time declaration >
```

Instead of “dt#” also the following can be used: “DT#”, “date_and_time”, “DATE_AND_TIME”.

The date and time declaration is to be entered in format <year-month-day-hour:minute:second>.

Seconds can be entered as real numbers, that is also fractions of a second can be specified.

DATE_AND_TIME values are internally handled as DWORD values, containing the time span in seconds since 01.01.1970, 00:00 clock.

Examples:

```
DATE_AND_TIME#1996-05-06-15:36:30
dt#1972-03-29-00:00:00
```

See also: **Time Data Types**.

Number Constants

Number values can appear as binary numbers, octal numbers, decimal numbers and hexadecimal numbers.

If an integer value is not a decimal number, the base must be followed by the number sign (#) in front of the integer constant.

The values for the numbers 10-15 in hexadecimal numbers will be represented by the letters A-F.

You may include the underscore character within the number.

Examples:

```
14              (* decimal number *)
2#1001_0011     (* dual number *)
8#67            (* octal number *)
16#A            (* hexadecimal number *)
```

These number values can be of type BYTE, WORD, DWORD, SINT, USINT, INT, UINT, DINT, UDINT, REAL or LREAL.

Implicit conversions from larger to smaller variable types are not permitted. This means that a DINT variable cannot simply be used as an INT variable. You must use the type conversion.

REAL/LREAL Constants

REAL and LREAL constants can be given as decimal fractions and represented exponentially. Use the standard American format with the decimal point to do this.

Example:

```
7.4             (* instead of 7,4 *)
1.64e+009       (* instead of 1,64e+009 *)
```

STRING Constants

A string is a sequence of characters.

STRING constants are preceded and followed by single quotation marks. You may also enter blank spaces and special characters (UMLAUTS for instance). They will be treated just like all other characters.

Notice the following possibilities of using the dollar sign “$” in string constants:

Sign	Result
$< two hex numbers >	Hexadecimal representation of the eight bit character code
$$	Dollar sign
$'	Single quotation mark
$L or $l	Line feed
$N or $n	New line
$P or $p	Page feed
$R or $r	Line break
$T or $t	Tab

Table 4-12. Possibilities of Use for $ Signal

Examples:

```
' Abby and Craig '
':-)'
'costs ($$)'
```

Typed Literals

Basically in using IEC constants the smallest possible data type will be used. If another data type must be used, this can be achieved with the help of typed literals without the necessity of explicitly declaring the constants.

For this purpose the constant will be provided with a prefix which determines the type.

Syntax:

```
<TYPE>#<Literal>
```

<TYPE> specifies the desired data type; possible entries are: BOOL, SINT, USINT, BYTE, INT, UINT, WORD, DINT, UDINT, DWORD, REAL, LREAL. The type must be written in uppercase letters.

<Literal> specifies the constant. The data entered must fit within the data type specified under <TYPE>.

Example:

```
var1:=DINT#34;
```

If the constant cannot be converted to the target type without data loss, an error message will be issued.

Typed literals can be used wherever normal constants can be used.

Variables

Variables can be declared either locally in the declaration part of a POU or in a Global Variable List.

See **Variables Declaration** for information on the declaration of a variable, including the rules concerning the variable identifier and multiple use.

Variables can be used anywhere the declared type allows for them.

You can access available variables through the *Input Assistant*.

Accessing Variables

Syntax:

Use the following syntax for accessing two-dimensional array:

```
<ARRAY NAME>[INDEX1, INDEX2]
```

Structure variables:

```
<STRUCTURE NAME>.<VARIABLE NAME>
```

Block and program variables:

```
<FUNCTION BLOCK NAME>.<VARIABLE NAME>
```

Addressing Bits

In integer variables individual bits can be accessed. For this purpose the index of the bit to be addressed is appended to the variable, separated by a dot. The bit-index can be given by any constant. Indexing is 0-based.

Syntax:

```
<Variable name>.<Bit index>
```

Example:

```
a : INT;
b : BOOL;
```

```
...
a.2 := b;
```

The third bit of the variable A will be set to the value of the variable B.

If the index is greater than the bit width of the variable, the following error message will be issued: “Index '<n>' outside the valid range for variable '<var>'!”

Bit addressing is possible with the following variable types: SINT, INT, DINT, USINT, UINT, UDINT, BYTE, WORD, DWORD.

If the variable type does not allow bit accessing, the following error message will be issued: “Invalid data type '<type>' for direct indexing”.

A bit access must not be assigned to a VAR_IN_OUT variable.

Bitaccess Via a Global Constant

If you have declared a global constant defining the bit-index, you can use this constant for a bitaccess.

Examples for a bit access via a global constant on a variable and on a structure variable:

Declaration in Global Variables List

Variable Enable defines which bit should be accessed:

```
VAR_GLOBAL CONSTANT
enable:int:=2;
END_VAR
```

Example 1, Bit access on an integer variable:

Declaration in POU:

```
VAR
xxx:int;
END_VAR
```

Bitaccess:

```
xxx.enable:=true;(* The third bit in variable xxx will be set TRUE *)
```

Example 2, Bit access on an integer structure component:

Declaration of structure stru1:

```
TYPE stru1 :
STRUCT
bvar:BOOL;
rvar:REAL;
wvar:WORD;
{bitaccess enable 42 'Start drive'}
END_STRUCT
END_TYPE
```

Declaration in POU:

```
VAR
x:stru1;
END_VAR
```

Bit access:

```
x.wvar.enable:=true;
```

This will set TRUE the 42. bit in variable x. Since bvar has 8 bits and rvar has 32 bits, the bitaccess will be done on the second bit of variable wvar, which as a result will get value 4.

Address

NOTE: Online change might change the contents on addresses. Please notice this when using pointers on addresses.

Memory Location

You can use any supported size to access the memory location.

For example, the address %MD48 would address bytes numbers 192, 193, 194, and 195 in the memory location area (48 * 4 = 192). The number of the first byte is 0. The table below shows the corresponding memory location dependent on the size (X: bit, B: byte, W: word, D: dword) for IEC addressing.

Examples:

	Address			
%MX	96.0 - 96.7	96.8 - 192.15	97.0 - 97.7	97.8 - 97.15
%MB	192	193	194	195
%MW	96		97	
%MD		48		

Table 4-13. Examples of Memory Positions

You can access words, bytes and even bits in the same way: the address %MX96.0 allows you to access the first bit in the 96th word (Bits are generally saved wordwise).

See **Address** for further information on addressing.

NOTE: Online Change might change the contents on addresses. Please notice this when using pointers on addresses.

Address

When specifying an address, the memory location and size are indicated by special character sequences.

Syntax:

Address with bit:

```
%<Memory area prefix><Prefix size><number.number>
```

Address without bit:

```
%<Memory area prefix><Prefix size><number.number>
```

Type	Description
I	Input (physical inputs via input driver, "sensors ")
Q	Output (physical outputs via output driver, "actors ")
M	Memory location

Table 4-14. Supported Memory Area Prefixes

Type	Description
X	Single bit
B	Byte (8 bits)
W	Word (16 bits)
D	Double Word (32 bits)

Table 4-15. Supported Size Prefixes

Examples:

Example	Description
%QX7.5	Output bit 7.5
%IW215	Input word 215
%QB7	Output byte 7
%MD48	Double word in memory position 48 in the memory location.
ivar AT %IW0 : WORD;	Example of a variable declaration including an address assignment.

Table 4-16. Addressing Examples

For assigning a valid address within an application, first of all you must know the appropriate position within the process image, that is the memory area to be used: Input (I), Output (Q) or Memory (M) area. Further on specify the desired size: bit, byte, word, dword (see above: X, B, W, D).

A decisive role plays the current device configuration and settings (hardware structure, device description, I/O settings). Especially consider the differences in address interpretation between devices using "byte addressing mode" or those using word oriented IEC addressing mode.

So depending on the size and addressing mode different memory cells might be addressed by the same address definition.

See the table below for a comparison of byte addressing and word oriented IEC addressing for bits, bytes, words and dwords. After all it visualizes the overlapping memory areas in case of byte addressing mode.

Concerning the notation regard that for bit addresses the IEC addressing mode is always word oriented, that means that the place before the dot corresponds to the number of the word, the place behind names the number of the bit.

Obs.: n = number of the byte.

DWords/Words				**Bytes**	**X (bits)**					
Byte addressing		**Word oriented IEC addressing**			**Byte addressing**			**Word oriented IEC addressing**		
D0	W0	D1	W0	B0	X0.7	...	X0.0	X0.7	...	X0.0
D1	W1			B1	X1.7	...	X1.0	X10.15	...	X0.8
	W2		W1	B2	...			X1.7	...	X1.0
	W3			B3				X1.15	...	X1.8
	W4	D1	W2	B4						
	...			B5						
			W3	B6						
				B7						
		D2		B8						
		...		...						
		...		...						
		...		...						
D(n-3)		D(n/4)	...							
	W(n-1)		W(n/2)							
				Bn	Xn.7	...	Xn.0	X(n/2).15	...	X(n/2).8

Table 4-17. Comparison of Byte and Word Oriented Addressing for the Address Sizes D,W, B and X

Overlap of memory ranges in case of byte addressing mode, example:

D0 contains B0 - B3, W0 contains B0 and B1, W1 contains B1 and B2, W2 contains B2 and B3. In order to get around the overlap W1, D1, D2, D3 must not be used for addressing.

NOTES:
- Boolean values will be allocated bytewise, if no explicit single-bit address is specified. Example: A change in the value of varbool1 AT %QW0 affects the range from QX0.0 to QX0.7.
- Online Change might change the contents on addresses. Please notice this when using pointers on addresses.

Functions

In ST a function call can also appear as an operand.

Example:

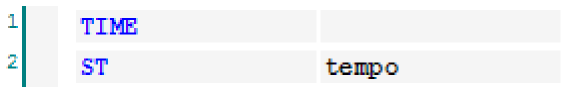
```
Result := Fct(7) + 3;
```

TIME()-Function

This function returns the time (based on milliseconds) which has been passed since the system was started.

The data type is TIME.

Example in IL:



```
TIME
ST      tempo
```

Example in ST:

```
time:=TIME();
```

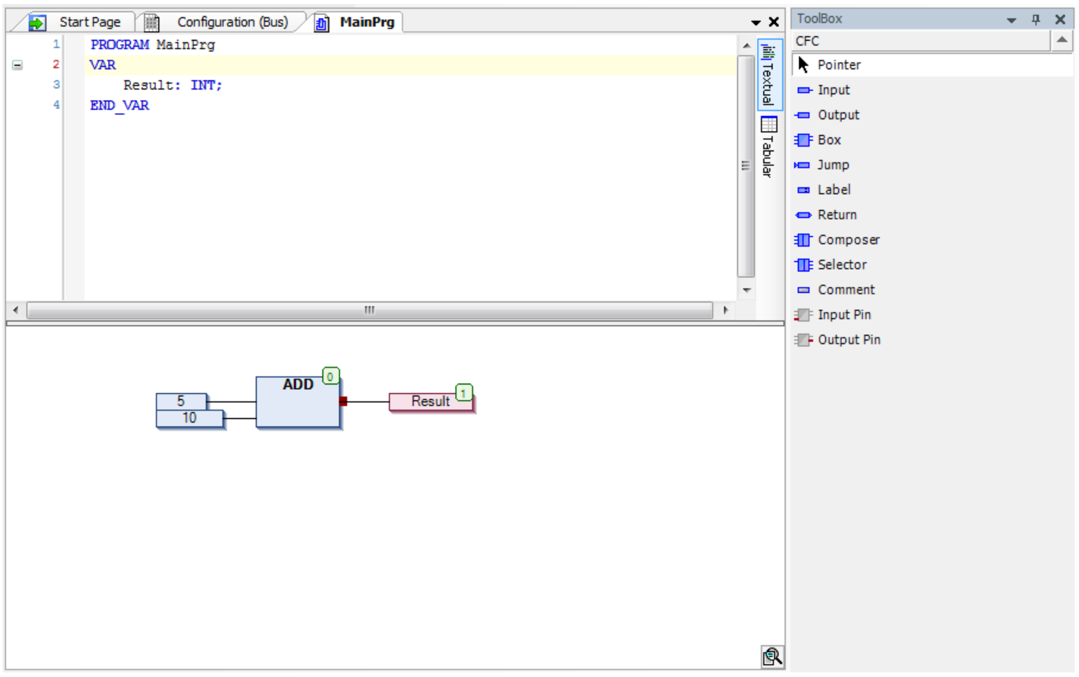
5. Programming Languages Editors

CFC Editor

The CFC Editor is available for programming objects in the programming language Continuous Function Chart (CFC), which is an extension to the IEC 61131-3 programming languages. You choose the language when adding a new *POU* object via the *Add Object* command in your project.

The CFC Editor is a graphical editor.

The editor will be available in the lower part of the window which opens when opening a CFC POU object and which also includes the *Declaration Editor* in its upper part.



```
PROGRAM MainPrg
VAR
    Result: INT;
END_VAR
```

Figure 5-1. CFC Editor

The CFC editor in contrast to the network editors allows free positioning of the elements, which for example allows direct insertion of feedback paths. The sequence of processing is determined by a list which contains all currently inserted elements and can be modified.

The following elements are available in a *Toolbox* for inserting: box (operators, functions, function blocks and programs), input, output, comment, label, jump, composer, selector.

The input and output lines of the elements can be connected by dragging a connection with the mouse. The course of the connecting line will be created automatically and noticing the shortest possible way. The connecting lines are automatically adjusted as soon as the elements are moved. See also: **Insert and Organize Elements**.

You may change the dimension of the editor window by zooming: Use the button in the lower right corner of the window and choose between the listed zoom factors. Alternatively you may select the entry ... to open a dialog where you can enter any arbitrary factor by typing.

The commands for working in the CFC editor can be called from the context menu or from the *CFC* menu which is available as soon as the CFC editor is active.

Continuous Function Chart Language - CFC

The Continuous Function Chart in extension to the IEC 61131-3 standard is a graphical programming language basing on the Function Block Diagram language. However in contrast to that no networks are used but free positioning of graphic elements, which for example allows feedback loops.

For creating CFC programming objects in MasterTool IEC XE see: **CFC Editor**.

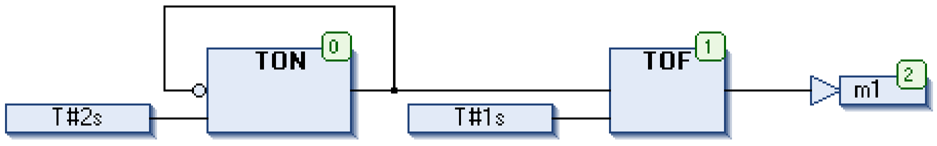


Figure 5-2. Example of a CFC Network

Cursor Positions in CFC

A possible cursor position in a CFC program is indicated by default by a grey shadow when moving with the cursor over the elements.

As soon as you click on one of these shadowed areas, still before you are leaving the mouse-button, the area will change color to red. As soon as you leave the pressed mouse-button, this will become the current cursor position, the respective element or text being selected and displayed red-colored.

There are three categories of cursor positions. See, on Figure 5-3, Figure 5-4 and Figure 5-5, the possible positions indicated by a grey shaded area as shown in the following pictures:

- If the cursor is positioned on a text, this will be displayed blue-shaded and can be edited. The ... button is available for opening the *Input Assistant*. Primarily after having inserted an element "???" is displayed and must be replaced by a valid identifier

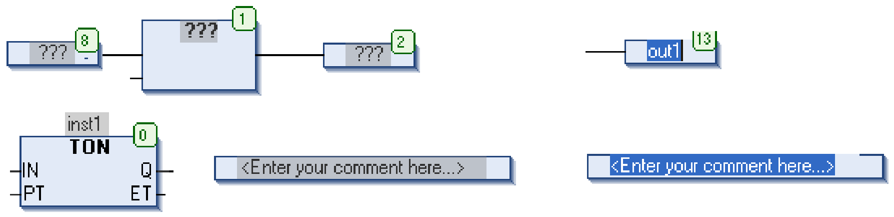


Figure 5-3. Possible Cursor Positions and Two Examples of Selected Texts

- If the cursor is positioned on the body of an element (box, input, output, jump, label, return, comment), this will be displayed red-colored and can be moved by moving the mouse

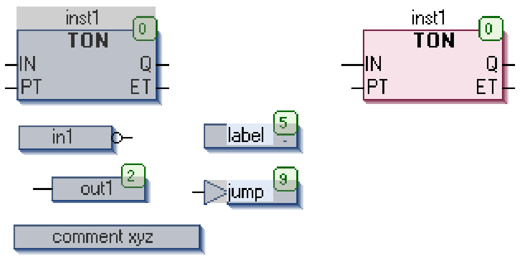


Figure 5-4. Possible Cursor Positions and Example of a Selected Body

- If the cursor is positioned on an input or output connection of an element, this will get displayed red-colored and can be negated or set/reset

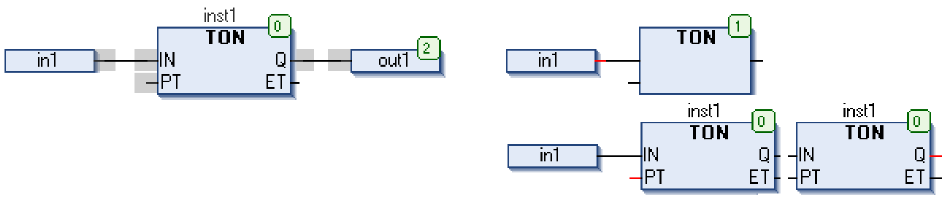


Figure 5-5. Possible Cursor Positions and Examples of selected Output and Input Positions

CFC Elements / Toolbox

The graphical elements available for programming in the CFC editor window are provided by a toolbox. The toolbox can be opened in a view window by command *Toolbox* which by default is in the *View* menu.

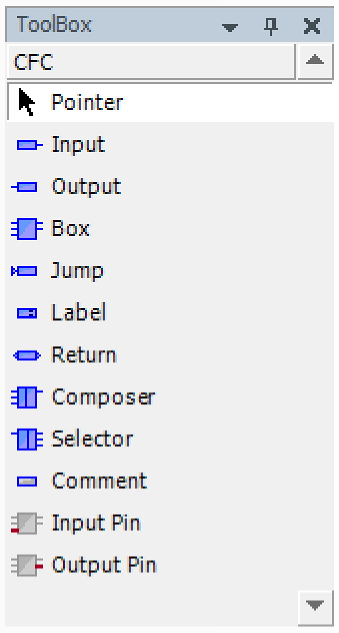
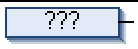
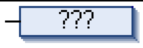
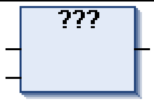


Figure 5-6. CFC Toolbox, Default

The desired element can be selected in the *Toolbox* and inserted in the editor window via drag&drop.

Besides the programming elements there is an entry Pointer , by default at the top of the toolbox list. As long as this entry is selected, the cursor has the shape of an arrow and you can select elements in the editor window for positioning and editing them.

The elements:

Symbol	Pin	Representation	Meaning
	Input		The text offered by "???" can be selected and replaced by a variable or constant. The input assistance serves to select a valid identifier.
	Output		The text offered by "???" can be selected and replaced by a variable or constant. The input assistance serves to select a valid identifier.
	Box		A box can be used to represent operators, functions, function blocks and programs. The text offered "???" can be selected and replaced by an operator, function, function block or program name. The input assistance serves to select one of the available objects. In case a function block is inserted, another "???" will be displayed above the box and have to be replaced by the name of the function block instance. If an existing box is

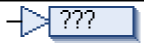
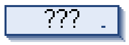
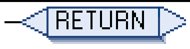
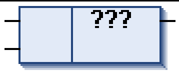
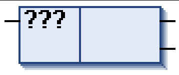
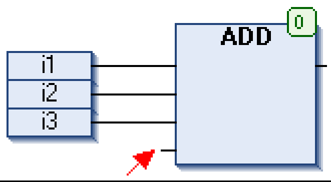
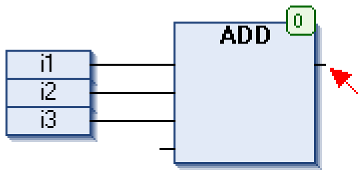
			replaced by another one (by modifying the entered name) and the new one has a different minimum or maximum number of input or output pins, the pins will be adapted correspondingly. If pins are to be removed, the lowest one will be removed first.
	Jump		The jump element is used to indicate at which position the execution of the program should continue. This position is defined by a label (see below). So replace the text offered by "???" by the label name.
	Label		A label marks the position to which the program can jump (see above Jump).
	Return		In Online mode a RETURN element is automatically inserted in the first column and after the last element in the editor. In stepping it is automatically jumped to before execution leaves the POU.
	Composer		A composer is used to handle an input of a box which is of type of a structure. The composer will display the structure components and thus make them accessible in the CFC for the programmer. For this purpose name the composer like the concerned structure (by replacing "???" by the name) and connect it to the box instead of using an "input" element.
	Selector		A selector in contrast to the composer is used to handle an output of a box which is of type of a structure. The selector will display the structure components and thus make them accessible in the CFC for the programmer. For this purpose name the selector like the concerned structure (by replacing "???" by the name) and connect it to the box instead of using an "output" element.
	Comment		Use this element to add any comments to the chart. Select the placeholder text and replace it with any desired text. You obtain a new line within the comment with <Ctrl> + <Enter>.
	Input Pin		Depending on the box type an additional input might be added. For this purpose select the box element in the CFC network and draw the Input Pin element on the box.
	Output Pin		Depending on the box type an additional output might be added. For this purpose select the box element in the CFC network and draw the Output Pin element on the box.

Table 5-1. Elements in the Editor Window

Example of the *Composer* element:

A CFC program cfc_prog handles an instance of function block fubblo1, which has an input variable struvar of type of a structure. By using the *Composer* element the structure components can be accessed:

Structure stru1 definition:

```
TYPE stru1 :
STRUCT
    ivar:INT;
    strvar:STRING:='hallo';
END_STRUCT
END_TYPE
```

Function block fublo1, declaration AND implementation:

```
FUNCTION_BLOCK fublo1
VAR_INPUT
struvar:STRU1;
END_VAR
VAR_OUTPUT
fbout_i:INT;
   fbout_str:STRING;
END_VAR
VAR
   fbvar:STRING:='world';
END_VAR
fbout_i:=struvar.ivar+2;
fbout_str:=CONCAT (struvar.strvar,fbvar);
```

Programa cfc_prog, declaration and implementation:

```
PROGRAM cfc_prog
VAR
intvar: INT;
stringvar: STRING;
fbinst: fublo1;
erg1: INT;
erg2: STRING;
END_VAR
```

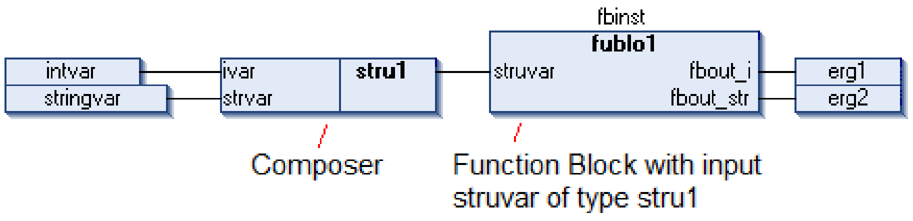


Figure 5-7. Composer Example

A CFC program cfc_prog handles an instance of function block fubblo2, which has an output variable fbout of type of a structure stru1. By using the *Selector* element the structure components can be accessed.

Structure stru1 definition:

```
TYPE stru1 :
STRUCT
ivar:INT;
strvar:STRING:='hallo';
END_STRUCT
END_TYPE
```

Function block fublo1, declaration and implementation:

```
FUNCTION_BLOCK fublo2
VAR_INPUT CONSTANT
fbin1:INT;
fbin2:DWORD:=24354333;
fbin3:STRING:='hallo';
END_VAR
VAR_INPUT
fbin : INT;
END_VAR
VAR_OUTPUT
fbout : stru1;
fbout2:DWORD;
```

```
END_VAR
VAR
fbvar:INT;
fbvar2:STRING;
END_VAR
```

Program cfc_prog, declaration and implementation:

```
VAR
intvar: INT;
stringvar: STRING;
fbinst: fublo1;
erg1: INT;
erg2: STRING;
fbinst2: fublo2;
END_VAR
```

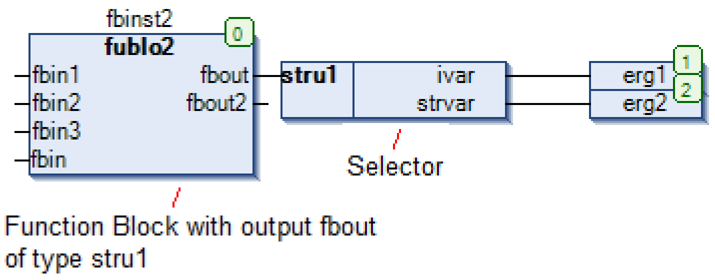


Figure 5-8. Selector Example

Insert and Organize Elements

The elements available for programming in the CFC Editor are provided in a *Toolbox* which by default is available in a window as soon as the CFC editor is opened.

The *CFC Editor Options* define general settings for the working within the editor.

Inserting

To insert an element select it in the *Toolbox* by a mouse-click, keep the mouse-button pressed and draw the element to the desired position in the editor window. During drawing the cursor will be displayed as an arrow plus an rectangle and a plus-sign. When you leave the mouse-button the element will be inserted.

Selecting

To select an inserted element for further actions like editing or rearranging notice the possible cursor positions for element bodies, in- and outputs and text By a mouse-click on a elements body the element gets selected and will be displayed by default red-shaded now. By additionally keeping pressed the <SHIFT> key you can click on and thereby select further elements. You also can press the left mouse-button and draw a dotted rectangle around all elements which should be selected. As soon as you leave the button the selection will be indicated. By command *Select all*, which by default is available in the context menu, all elements are selected at once.

By using the arrow keys the selection mark can be shifted to the next possible cursor position. The sequence depends on the execution order or the elements, which is indicated by element numbers, see below.

When an input pin is selected and <CTRL>+<LEFT ARROW> are pressed, the corresponding output will be selected. When an output pin is selected and <CTRL>+<LEFT ARROW> are pressed, the corresponding output(s) will be selected.

Replacing Boxes

To replace an existing box element, replace the currently inserted identifier by that of the desired new element. Notice that the number of input and output pins will be adapted if necessary due to the definition of the POUs and thus some existing assignments might be removed.

Moving

To move an element, select the element by a mouse-click on the element body (see possible cursor positions) and drag it, while keeping the mouse-button pressed, to the desired position. Then leave the mouse-button to place the element. You also can use the *Cut* and *Paste* commands for this purpose.

Connecting

The connections between the inputs and outputs of elements can be drawn with the mouse. The shortest possible connection will be created taking into account the other elements and connections. If the course of connection lines is painted light-grey-colored, this might indicate that elements are positioned covering each other.

Copying

To copy an element, select it and use the *Copy* and *Paste* commands.

Editing

After inserting an element, by default the text part is represented by “???”. To replace this by the desired text (POU name, label name, instance name, comment etc.) select the text by a mouse-click to get an edit field. Also button will be available then to open the *Input Assistant*.

Deleting

A selected element can be deleted by command *Delete*, which is available in the context menu, or by the <DEL>.

Execution Order, Element Numbers

The sequence in which the elements in a CFC network are executed in online mode is indicated by numbers in the upper right corner of the box, output, jump, return and label elements. The processing starts at the element with the lowest number, which is "0". The execution order can be modified by commands which are by default available in submenu *Execution Order* in the *CFC* menu.

When adding an element, the number will automatically be given according to the topological sequence (from left to right and from above to below). The new element receives the number of its topological successor if the sequence has already been changed, and all higher numbers are increased by one.

Notice that the number of an element remains constant when it is moved.

Consider that the sequence influences the result and must be changed in certain cases.

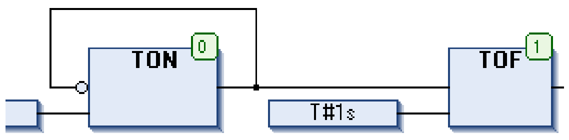


Figure 5-9. Example of Element Numbers

Changing Size of the Working Sheet

In order to get more space around an existing CFC chart in the editor window, the working area (working sheet) size can be changed. This might be done by selecting and dragging all elements with the mouse or by the *Cut* and *Paste* commands (see above, **Moving**).

Alternatively a special dimensions settings dialog can be used, which might save time in the case of big charts. See **Edit Working Sheet** in the MasterTool IEC XE User Manual– MU299609 for a description.

CFC Editor in Online Mode

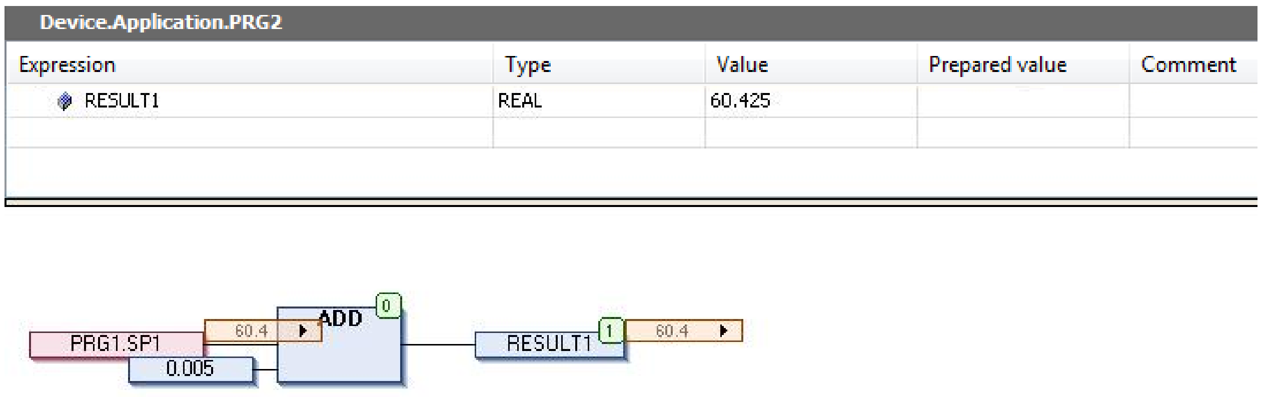
In online mode the CFC editor provides views for monitoring and for writing and forcing the variables and expressions on the controller. *Debug* functionality (breakpoints, stepping etc.) is available.

For information on how to open objects in online mode see **User Interface in Online Mode** in the MasterTool IEC XE User Manual– MU299609.

Notice that the editor window of an CFC object also includes the *Declaration Editor* in the upper part. For information on the declaration editor in online mode see **Declaration Editor in Online Mode** in the MasterTool IEC XE User Manual– MU299609.

Monitoring

The actual values are displayed in small monitoring windows behind each variable (inline monitoring).


Device.Application.PRG2

Expression	Type	Value	Prepared value	Comment
RESULT1	REAL	60.425		

Figure 5-10.Online view of a Program Object (MAINPRG)

Notice that for the online view of a function block POU: In the implementation part no values will be viewed in the monitoring windows but “<Value of the expression>” and the inline monitoring fields in the implementation part will show three question marks each.

Breakpoint Positions in CFC Editor

Basically the breakpoint positions are those positions in a POU at which values of variables can change or at which the program flow branches out resp. another POU is called Figure 5-11.

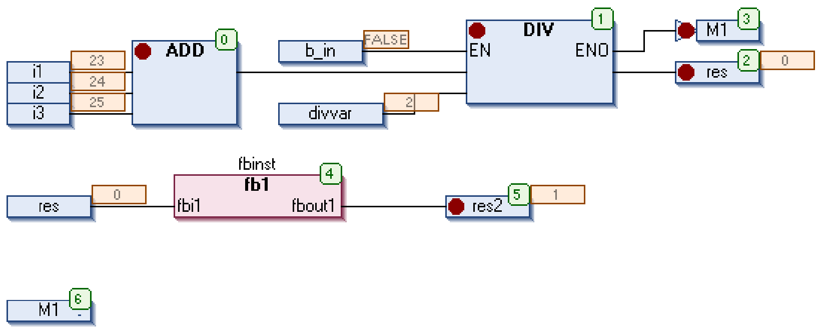


Figure 5-11. Breakpoint Positions in CFC Editor

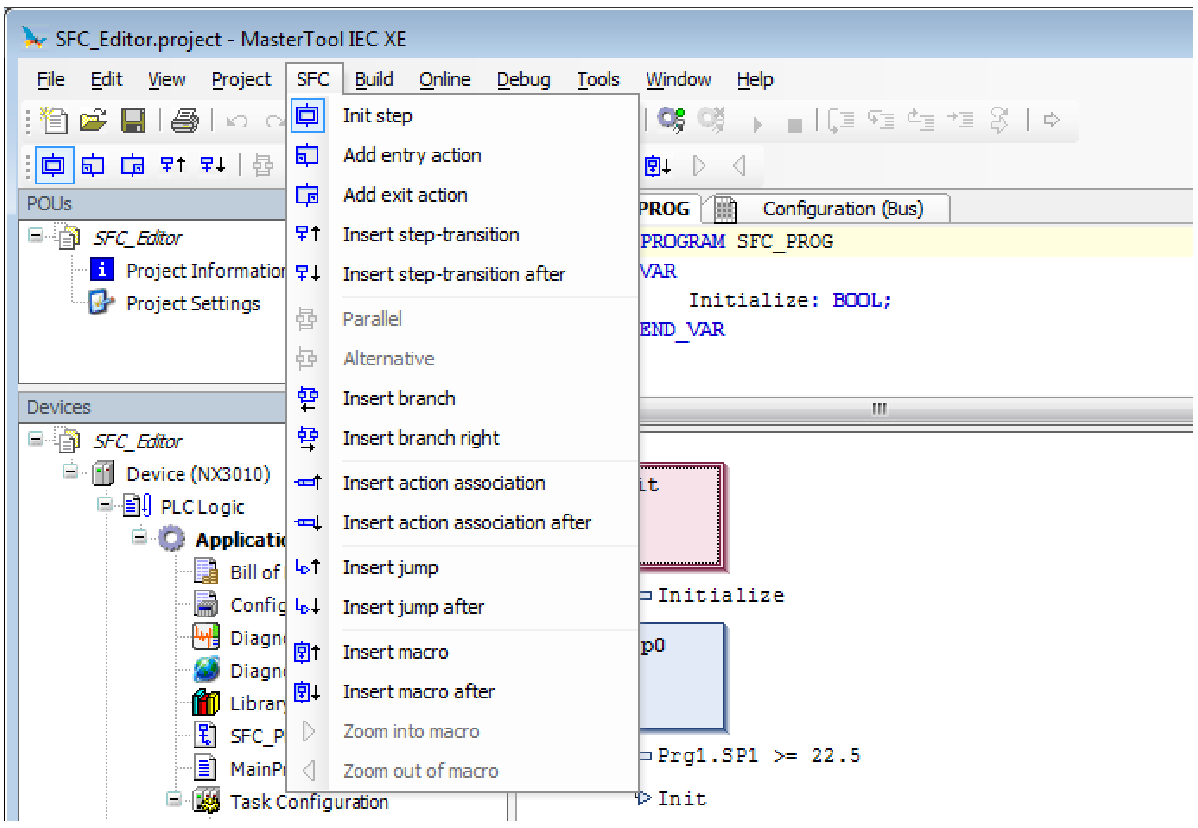
NOTE: Notice for breakpoints in methods: A breakpoint will be set automatically in all methods which might be called. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing the method.

SFC Editor

The SFC editor is available for programming objects in the IEC 61131-3 programming language Sequential Function Chart (SFC). The language is to be chosen when adding a new POU object to the project via the *Add Object* command. The SFC editor is a graphical editor.

General settings concerning behavior and display are done in the *SFC Editor Options* dialog.

The SFC editor is available in the lower part of the window which opens when you edit a SFC POU object. In its upper part this window contains the *Declaration Editor*.



```
PROGRAM SFC_PROG
VAR
    Initialize: BOOL;
END_VAR
```

Figure 5-12. SFC Editor

The elements used in a SFC diagram by default are available in the *SFC* menu, which by default is available as soon as the SFC editor is active. They are to be arranged in a sequence/parallel sequences of steps which are connected by transitions. See also: **Working in SFC Editor**.

The properties of steps can be edited in a separate *Properties* box. Inter alia there the minimum and maximum time of activity can be defined for each step.

Implicit variables can be accessed for controlling the processing of a SFC (for example step status, timeout analysis, reset etc.).

The commands for working in the SFC Editor can be called from the context menu or from the *SFC* menu which by default is available as soon as the SFC Editor is active.

In the SFC editor:

- Editing is made more comfortable by the fact that each particular element can be selected and arranged individually. During editing the syntax of the SFC not necessarily must be matched. syntax errors will not be checked until a *Generate Code* command
- There is only one step type, combining the two types (IEC steps and non-IEC steps), which are used in previous versions. Actions always must be provided as POUs. The actions always are assigned via the step element properties
- Macros can be used for structuring purposes

SFC - Sequential Function Chart

The Sequential Function Chart (SFC) is a graphically oriented language which allows to describe the chronological order of particular actions within a program. These actions are available as separate programming objects, written in any available programming language. In a SFC they get assigned to step elements and the sequence of processing is controlled by transition elements. For a detailed description on how the steps will be processed in online mode see **Sequence of Processing in SFC**.

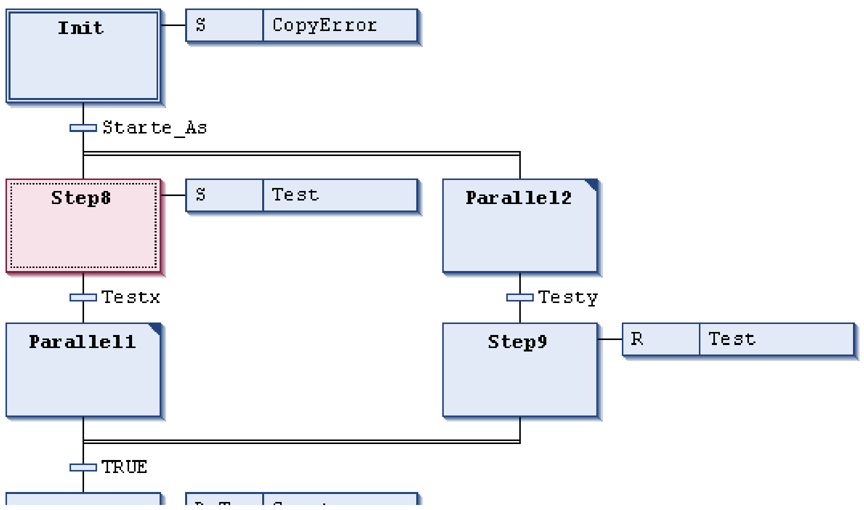


Figure 5-13. Example for a Sequence of Steps in a SFC Module

Cursor Positions in SFC

A possible cursor position in a SFC diagram in the SFC Editor is indicated per default by a grey shadow when moving with the cursor over the elements.

There are two categories of cursor positions: Texts and element bodies. See the possible positions indicated by a grey shaded area as shown in the following pictures:

Texts

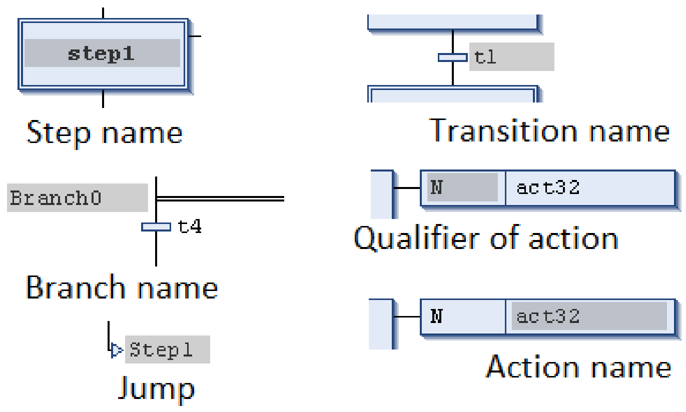


Figure 5-14. Possible Cursor Positions, Texts

When you click on a text cursor position, the string will get editable:

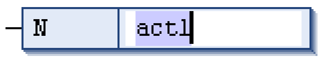


Figure 5-15. Select Action Name for Editing

Element Bodies

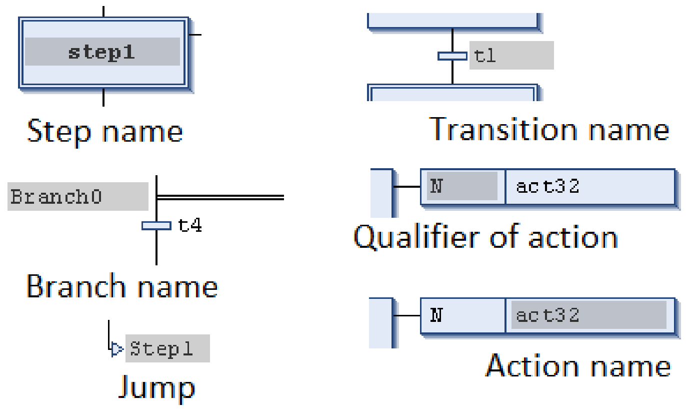


Figure 5-16. Possible Cursor Positions, Element Bodies

When you click on a shadowed area, the element will get selected. It gets a dotted frame and is displayed red-shaded (for multiple selection see **Working in SFC Editor**).

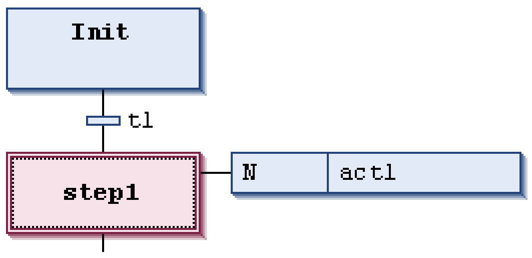


Figure 5-17. Selected Step Element

Working in SFC Editor

By default a new SFC POU contains an init step and a subsequent transition. For how to add further elements, how to arrange and edit the elements see the following information:

For possible cursor positions, see the above item.

Navigating: Jumping to the next / previous element in the chart is possible by using the arrow keys.

Insert Elements

The particular SFC elements can be inserted via the respective commands which by default are available in the *SFC* menu. See **Cursor Positions in SFC** for details. A double-click on an already inserted step, transition or action element, which does not yet reference a project object, will open a dialog for assigning one.

Select Elements

An element and a text field might be selected by a mouse-click on a possible cursor position. The selection might also always be given to an adjacent element by using the arrow keys. The element will change color to red. For examples see **Cursor Positions in SFC**.

Steps and transitions can be selected and thus also moved (cut, copy, paste) or deleted separately. Multiple selection is possible by the following:

- Keep the <SHIFT> key pressed and subsequently click on the particular elements to be selected
- Press the left mouse-key and draw a rectangle (dotted line) around the elements to be selected
- Use command Select All (*Edit* menu)

Editor Texts

By a mouse-click on a text-cursor-position at once the edit field opens, where you can edit the text. If a text area has been selected via the arrow keys, the edit field must be opened explicitly by using the <SPACE> bar.

Edit Associated Actions

A double click on an step (entry, active or exit) or transition action association opens the associated action in the corresponding editor. For example perform a double click on the transition element or on the triangle indicating an exit action in a step element.

Cut, Copy, Paste Elements

Select the element(s) and use command *Cut*, *Copy* and *Paste* (*Edit* menu) or the corresponding keys.

Notice the following behavior:

- When you paste one or several cut or copied element(s), the content of the clipboard will be inserted before the currently selected position. If nothing is currently selected, the element(s) will be appended at the end of the currently loaded chart
- If you paste a branch while the currently selected element is also a branch, the pasted branch elements will be inserted left to the existing ones
- If you paste an action (list) at a currently selected step, the actions will be added at the beginning of the action list of the step resp. an action list for the step will be created
- Incompatible elements when cutting/copying: If an associated action (list) and additionally an element, which is not the step to which the action (list) belongs, are selected, a message box will appear: “The current selection contains incompatible elements. No data will be filed to the clipboard”. The selection will not be stored and you cannot paste or copy it somewhere else
- Incompatible elements when pasting: If you try to paste an action (list) while the currently selected element is not a step or another association, an error box will appear: “The current clipboard content cannot be pasted at the current selection”. If you try to paste an element like a

step, branch or transition when currently an associated action (list) is selected the same message box will appear

Delete Elements

Select the element(s) and use command *Delete* or the <DEL> key. Notice the following:

- Deleting a step also deletes the associated action list
- Deleting the init step automatically sets the following step to be the initial one, that is option Initial step will be activated in the properties of this step
- Deleting the horizontal line preceding a branched area will delete all branches
- Deleting all particular elements of a branch will delete the branch

SFC Element Properties

The properties of a SFC element can be viewed and edited in the *Element properties* window. This window can be opened via command *Element Properties* (*View* menu).

It depends on the currently selected element which properties are displayed. The properties are grouped and the particular group sections can be opened or closed by using the plus/minus signs.

Notice that in the *View* tab of the *SFC Editor* options you can configure whether the particular types of properties should be displayed next to an element in the SFC chart.

Common:

Property	Description
Name	Element name, by default <element><running number> , for example step name Step0, Step1, branch name branch0 etc.
Comment	Element comment, text string, for example "Reset the counter". Line breaks can be inserted via <Ctrl>+<Enter>.
Symbol	For each SFC element implicitly a flag is created, named like the element. Here you can specify whether this flag variable should be exported to the symbol configuration and how the symbol then should be accessible in the PLC. Perform a double-click on the value field, resp. select the value field and use the space-key in order to open a selection list from which you can choose one of the following access options. None: The symbol will be exported to the symbol configuration, but it won't not be accessible in the PLC: Read: The symbol will be exported to the symbol configuration and it will be readable in the PLC. Write: The symbol will be exported to the symbol configuration and it will be written in the PLC. Read/Write: Combination of Read and Write. By default nothing is entered here, which means, that the symbol not at all is exported to the symbol configuration

Table 5-2. Common Properties Description

Specific:

Property	Description
Initial Step	This option always is activated in the properties of the current initial step (init step). By default it is activated for the first step in a SFC and deactivated for other steps. Notice that if you activate this option for another step, you must deactivate it in the previous init step in order to avoid a compiler error.
Times	Notice the possibility to detect timeouts in steps by the SFCError flag.
Minimal active	Minimum length of time the processing of this step should take; permissible values: time according to IEC-syntax (for example t#8s) or TIME variable; default: t#0s.
Maximal active	Maximum length of time the processing of this step should take; permissible values: time according to IEC-syntax (for example t#8s) or TIME variable; default: t#0s.

Actions	Define here the actions to be performed when the step is active. Notice the description of the sequence of processing for details.
Step entry	This action will be executed after the step has got active.
Active step	This action will be executed when the step is active and possible entry actions have been already processed.
Step exit	This action will be executed in the subsequent cycle after a step has been deactivated.

Table 5-3. Specific Properties Description

NOTE: Notice the possibility of getting information on step/action status and timeouts via the appropriate implicit variables and SFC flags.

SFC Elements / Toolbox

The graphic elements usable for programming in the SFC editor window currently can be inserted by using the insert commands (*SFC* menu).

See also: **Working in SFC Editor**.

The following elements are available and described in the following:

- Step
- Transition
- Action
- Branch (Alternative)
- Branch (Parallel)
- Jump
- Macro

Step-Transition

Symbol:

A step is represented by a box containing the step name and being connected to the preceding and subsequent transitions by a line.

The step name can be edited inline.

The box frame of the init step is double-lined.

Notice that each step - by command *Init step* or by activating the respective step property - can be transformed to an init step , that is to that step, which will be executed first when the IL-POU is called.

Each step is defined by the step properties.

The actions to be performed when the step is active (processed) are to be associated (see below, **Action**).

Steps and transitions are basically inserted in combination via command *Insert step-transition (after)*.

NOTES:
- There is only one type of steps (the IEC conforming steps).
- Step names must be unique in the scope of the parent POU. Notice this especially when using actions programmed in SFC.

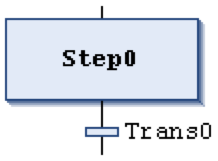


Figure 5-18. Step and Subsequent Transition

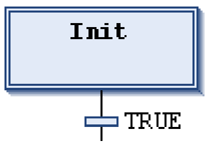


Figure 5-19. Initial Step and Subsequent Transition

Transition

A transition is represented by a small rectangle box connected to the preceding and subsequent steps by a line. It provides the condition on which the following step will get active (as soon as the condition is TRUE).

The transition name and the transition condition is displayed right to the box.

By default automatically a transition “trans<n>” is inserted, whereby n is a running number.

This default name can be selected and modified:

- A valid name is either the name of a transition object () available in the POUs tree (this allows multiple use of transitions; see for example “t1” in the left column)
- A valid conditional expression

NOTE: Regard that transitions which consist of a transition or a property object are indicated by a small triangle in the upper right corner of the rectangle.

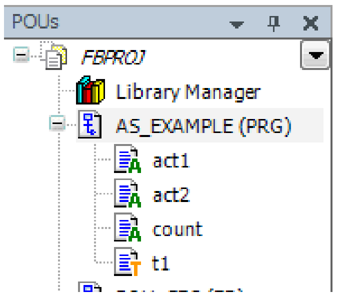


Figure 5-20. Transition in POUs Tree

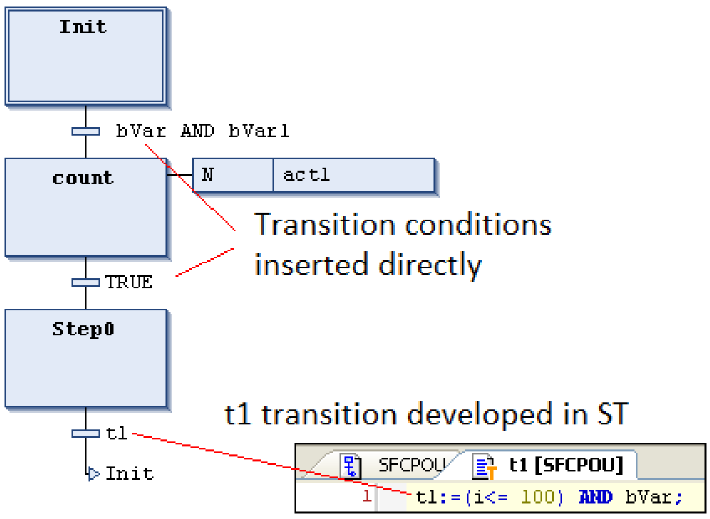


Figure 5-21. Transition Examples

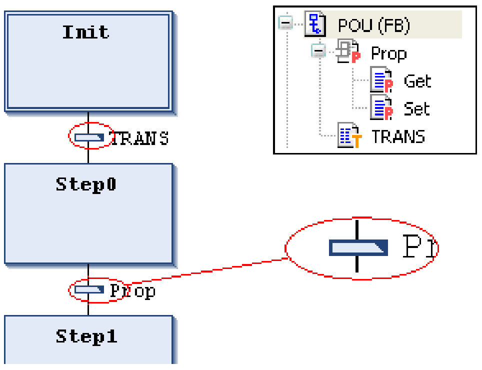


Figure 5-22. Transition or Property Object Indicated by a Triangle

A transition condition must have the value TRUE or FALSE. Thus it can consist of either a Boolean variable, a Boolean address, a Boolean constant, or a series of instructions having a Boolean result. But a transition may not contain programs, function blocks or assignments.

A transition condition is handled like a method call. It be entered according to the syntax:

```
<Transition name>:=<Transition condition>; (* for example "trans1:= a=100"
*)
```

Or:

```
<Transition condition>; (* for example "a=100" *)
```

See example above (t1).

In online mode the subsequent step can only get active if the preceding transition has become TRUE.

Action

Symbol:

An action can contain a series of instructions written in one of the valid programming languages. It is assigned to a step and in online mode it will be processed according to the defined sequence of processing.

Each action to be used in SFC steps must be available as a valid POU within the SFC POU and the project ().

The *Add Object* command is available for adding an action POU below a SFC POU.

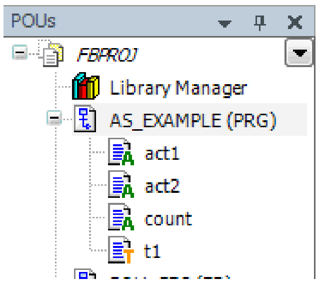
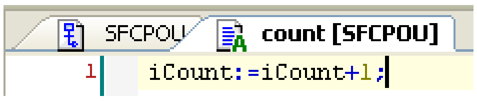


Figure 5-23. Actions in POUs Tree

NOTE: Step names must be unique in the scope of the parent POU. An action may not contain a step having the same name like the step to which it is assigned to.



```
iCount:=iCount+1;
```

Figure 5-24. Example of an Action Written in ST

There are the following types of actions:

- IEC Actions
- IEC-extending step actions

IEC Conforming Step Action (IEC Action)

This is an action according to standard IEC1131-3 which will be processed according to its qualifier when the step has got active and a second time when it has got deactivated. In case of assigning multiple actions to a step (action list) the actions will be executed from up to down.

Different qualifiers can be used for IEC step actions in contrast to a normal step action.

A further difference to the normal step actions is that each IEC step action is provided with a control flag, which allows to make sure that - even if the action is called also by another step - the action will get executed always only once at a time. This is not guaranteed with the normal step actions.

An IEC step action is represented by a bipartite box, connected to the right of a step via a connection line. In the left part it shows the action qualifier, in the right part the action name. Both can be edited inline.

IEC step actions get associated to a step via the *Insert action association (after)* command. One or multiple actions can be associated to a step. The position of the new action depends on the current cursor position and the command. The actions must be available in the project and be inserted with a unique action name (for example MainPrg.a1).

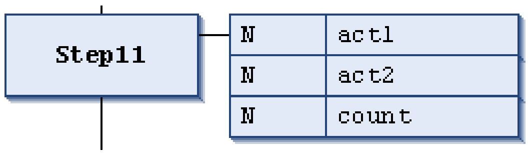


Figure 5-25. IEC Conforming Step Action List Associated to a Step

Each action box in the first column shows the qualifier and in the second the action name.

IEC Extending Step Actions

These are actions extending the IEC standard: The actions must be available as objects below the SFC object. The action names must be unique.

Step Entry Action

This type of step action will be processed as soon as the step has become active and before the step active action.

The action is associated to a step via an entry in the *Step Entry* field of the step properties. It is represented by an "E" in the lower left corner of the respective step box.

Step Active Action

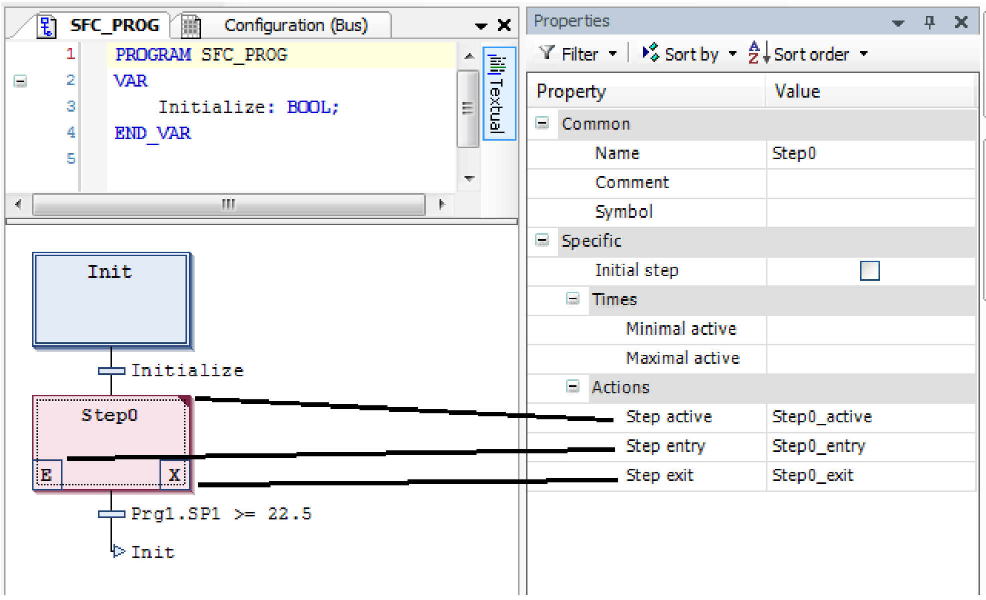
This type of step action will be processed when the step has become active and after a possible step entry action of this step has been processed. However in contrast to an IEC step action it is not executed once more when it gets deactivated and it cannot get assigned qualifiers.

The action is associated to a step via an entry in the *Step active* field of the step properties. It is represented by a small triangle in the upper right corner of the respective step box.

Step Exit Action

An exit action will be executed once when the step has got deactivated. Notice however that this execution will not be done in the same, but at the beginning of the subsequent cycle.

The action is associated to a step via an entry in the *Step exit* field of the step properties. It is represented by an X in the lower right corner of the respective step box.



```
PROGRAM SFC_PROG
VAR
    Initialize: BOOL;
END_VAR
```

Property	Value
Common	
Name	Step0
Comment	
Symbol	
Specific	
Initial step	☐
Times	
Minimal active	
Maximal active	
Actions	
Step active	Step0_active
Step entry	Step0_entry
Step exit	Step0_exit

Figure 5-26. IEC Extending Step Actions

The *Step active*, *Step Entry* and *Step Exit* actions are defined in the step properties.

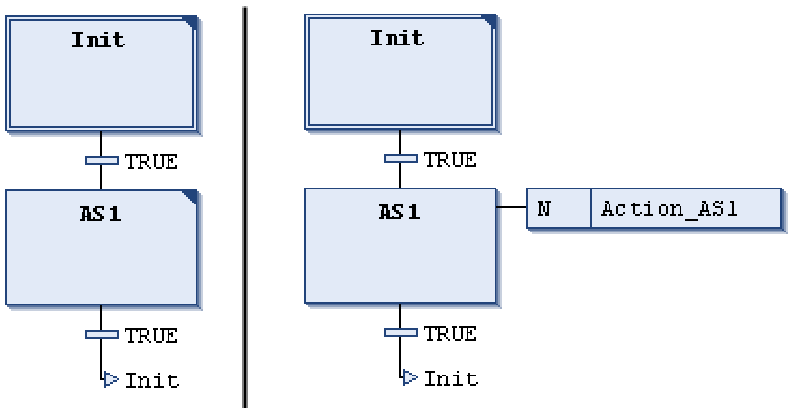


Figure 5-27. Step Active Action

Branches

A sequential function chart can diverge, that is the processing line can be branched into two or several further lines (branches). Parallel branches will be processed parallel (both at a time), in case of alternative branches only one will be processed depending on the preceding transition condition. Each branching within a chart is preceded by a horizontal double (parallel) or simple (alternative) line and also terminated by such a line or by a jump.

Parallel Branch

Symbol:

A parallel branch must begin and end with a step. Parallel branches can contain alternative branches or other parallel branches.

The horizontal lines before and after the branched area are double-lines.

Processing in online mode: If the preceding transition (t2 in Figure 5-28) is TRUE, the first steps of all parallel branches will become active. The particular branches will be processed parallel to one another before the subsequent transition (t3 in Figure 5-28) will be noticed.

A parallel branch is inserted via command *Insert branch (right)* when a step is currently selected.

Notice that parallel and alternative branches can be transformed to each other by the commands *Parallel* and *Alternative*. This might be useful during programming.

Automatically a branch label is added at the horizontal line preceding the branching which is named Branch<n>, whereby n is a running number starting with 0 (zero). This label can be specified when defining a jump target.

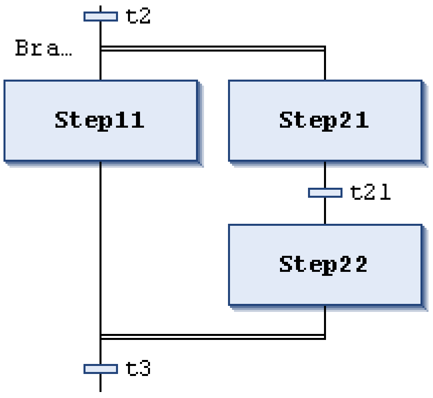


Figure 5-28. Parallel Branch

Alternative Branch

Symbol:

An alternative branch must begin and end with a transition. Alternative branches can contain parallel branches and other alternative branches.

The horizontal lines before and after the branched area are simple lines.

If the step which precedes the alternative beginning line is active, then the first transition of each alternative branch will be evaluated from left to right. The first transition from the left whose transition condition has value TRUE, will be opened and the following steps will be activated.

Alternative branches are inserted via command *Insert branch (right)* when a transition is currently selected.

The horizontal lines before and after the branched area are simple lines.

Notice that parallel and alternative branches can be transformed to each other by the commands *Parallel* and *Alternative*. This might be useful during programming.

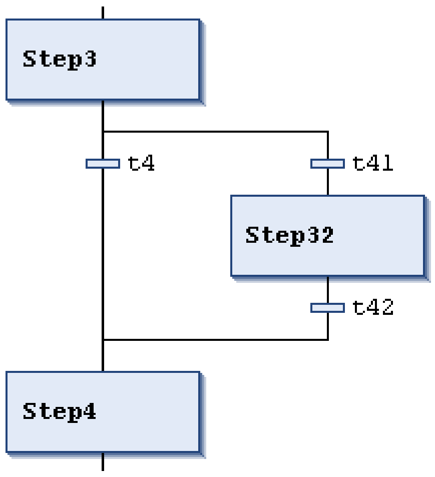


Figure 5-29. Alternative Branch

Jump

Symbol:

A jump is represented by a vertical connection line plus a horizontal arrow and the name of the jump target.

A jump defines the next step to be processed as soon as the preceding transition is TRUE. Jumps might be needed because the processing lines must not cross or lead upward.

Besides the default jump at the end of the chart a jump may only be used at the end of a branch. It gets inserted via command *Insert jump (after)* when the last transition of the branch is selected.

The target of the jump is specified by the associated text string which can be edited inline. It can be a step name or the label of a parallel branch.

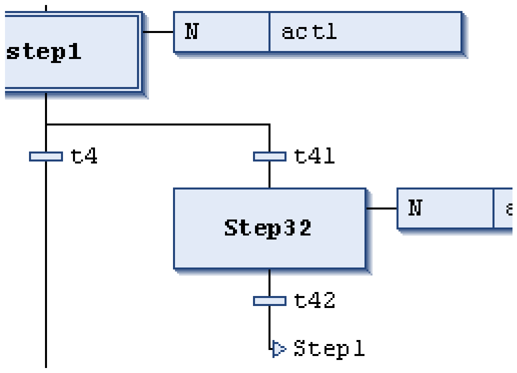


Figure 5-30. Jump

Macro

Symbol:

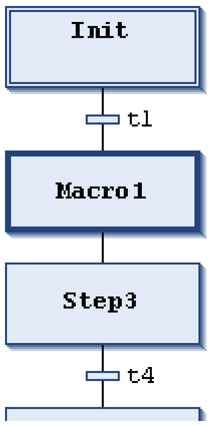


Figure 5-31. Main SFC Editor View

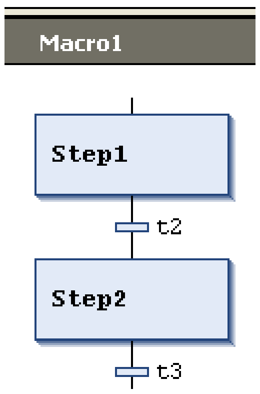


Figure 5-32. Macro Editor View for Macro1

A macro is represented by a bold-framed box containing the macro name.

It includes a part of the SFC chart, which thus is not directly visible in the main editor view.

The process flow is not influenced by using macros, it is just a way to hide some parts of the program, for example in order to simplify the display.

A macro box is inserted by command *Insert macro (after)*. The macro name can be edited inline.

To open the macro editor, perform a double-click on the macro box or use command *Zoom in* to macro. You can edit here just as in the main editor view and enter the desired section of the SFC chart. To get out use *Zoom out* of macro.

The title line of the macro editor always shows the path of the macro within the current SFC.

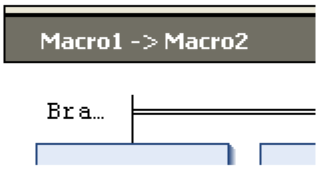


Figure 5-33. Title Line of the Macro Editor

Qualifier

In order to configure in which way the actions should be associated to the IEC steps, some qualifiers are available, which are to be inserted in the qualifier field of an action element.

These qualifiers are handled by the SFCActionControl function block of the IecSfc.library, which automatically is included in a project by the features of SFC.

The available qualifiers:

Qualifier	Name	Description
N	Non-stored	The action is active as long as the step is active.
R	Reset	The action gets deactivated.
S	Set	The action will be started when the step becomes active and will be continued after the step is deactivated, until the action gets reset.
L	Time Limited	The action will be started when the step becomes active and it will continue until the step goes inactive or a set time has passed.
D	Time Delayed	A delay timer will be started when the step becomes active. If the step is still active after the time delay, the action will start and continue until it gets deactivated.
P	Pulse	The action will be started when the step becomes active/deactive and will be executed once.
SD	Pulse stored and time delayed	The action will be started after the set time delay and it will continue until it gets reset.
DS	Delayed and stored	If the step is still active after the specified time delay, the action will start and it will continue until it gets reset.
SL	Stored and time limited	The action will be started when the step becomes active and it will continue for the specified time or until a reset.

Table 5-4. Qualifiers

The qualifiers L, D, SD, DS and SL need a time value in the TIME constant format.

NOTE: When an IEC action has been deactivated it will be executed once more. This means that each action at least is executed twice.

Implicit Variables - SFC Flags

Each SFC step and IEC action provides implicitly generated variables for watching the status of steps and IEC actions during runtime. Also variables can be defined for watching and controlling the execution of a SFC (timeouts, reset, tip mode). These variables also might be generated implicitly by the SFC object.

Basically for each step and each IEC action an implicit variable is generated. A structure instance, named like the element, for example step1 for a step with step name step1. Notice the possibility, to define in the element properties, whether for this flag a symbol definition should be exported to the symbol configuration and how this symbol should be accessible in the PLC.

The data types for those implicit variables are defined in library IecSFC.library. This library will automatically be included in the project as soon as an SFC object is added.

Step and Action Status and Step Time

Basically for each step and each IEC-action an implicit structure variable of type SFCStepType and SFCActionType is created. The structure components (flags) describe the status of a step/action or the currently processed time of an active step.

The syntax for the implicitly done variable declaration is:

```
<Step name>: SFCStepType;
```

E:

```
_<Action name>: SFCActionType;
```

NOTE: Implicit variables for actions always are preceded by an underscore.

The following Boolean flags for step or action states are available:

- <step name>.x: shows the current activation status
- <step name>._x: shows the activation status for the next cycle

If <step name>.x = TRUE, the step will be executed in the current cycle.

If <step name>._x = TRUE and <step name>.x = FALSE, the step will be executed in the following cycle, that is <step name>._x gets copied to <step name>.x at the beginning of a cycle.

The following Boolean flags for step or action states are available:

- _<action name>.x is TRUE, if the action is executed
- _< action name >._x is TRUE, if the action is active

Symbol Generation

In the element properties of a step or an action you can define, whether for the step or action name flag a symbol definition should be added to a possibly created and downloaded symbol application. For this purpose make an entry for the desired access right in column *Symbol* of the element properties view.

NOTE: The flags described above might be used to force a certain status value for a step, that is for setting a step active, but be aware that this will effect uncontrolled states within the SFC.

Time Via TIME Variables

The flag “t” gives the current time span which has passed since the step had got active. This is onlyfor steps, no matter whether there is a minimum time configured in the step attributes or not (see also below: SFCError).

For steps:

```
<stepname>.t (<stepname>._t not usable for external purposes)
```

For actions: the implicit time variables are not used.

Control of SFC Executions (Timeouts, Reset, Tip Mode)

Some implicitly available variables, also named SFC flags, (see table below) can be used to control the operation of an SFC, for example for indicating time overflows or enabling tip mode for switching transitions.

In order to be able to access these flags and to get them work, they must be declared and activated.

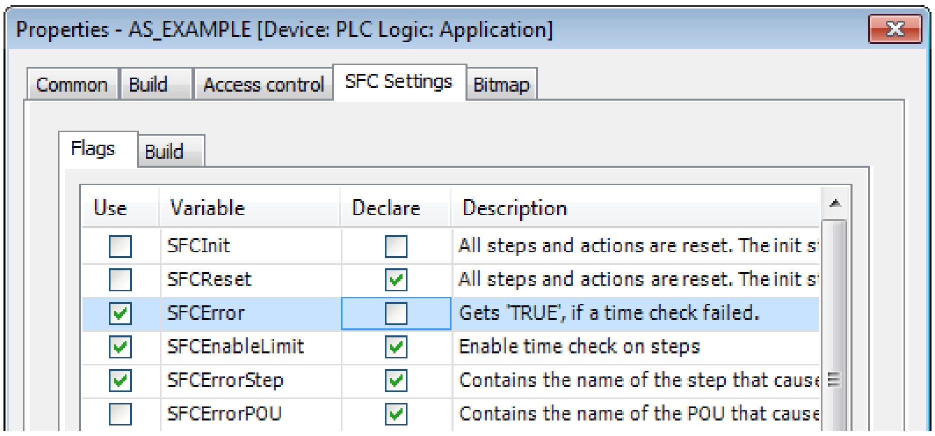
This is to be done in the *SFC Settings* dialog which is a sub dialog of the object *Properties* dialog. Manual declaration, as it was needed in previous versions, is only necessary to enable write access from another POU (see below, **Accessing Flags**). In this case however regard the following: If you declare the flag globally, you must deactivate the *Declare* option in the *SFC Settings* dialog in order not to get a implicitly declared local flag, which then would be used instead of the global one. Keep in mind, that the SFC Settings for a SFC POU initially are determined by the definitions currently set in the *SFC Options* dialog.

Notice that a declaration of a flag variable solely done via the *SFC Settings* dialog will only be visible in the online view of the SFC POU.

There is a SFC POU named sfc1 containing a step s2 which has time limits defined in its step attributes. See the attributes displayed below in Figure 5-34.

If for any reason the step s2 stays active for a longer time than allowed by its time properties (time overflow), a SFC flag will be set which might be read by the application.

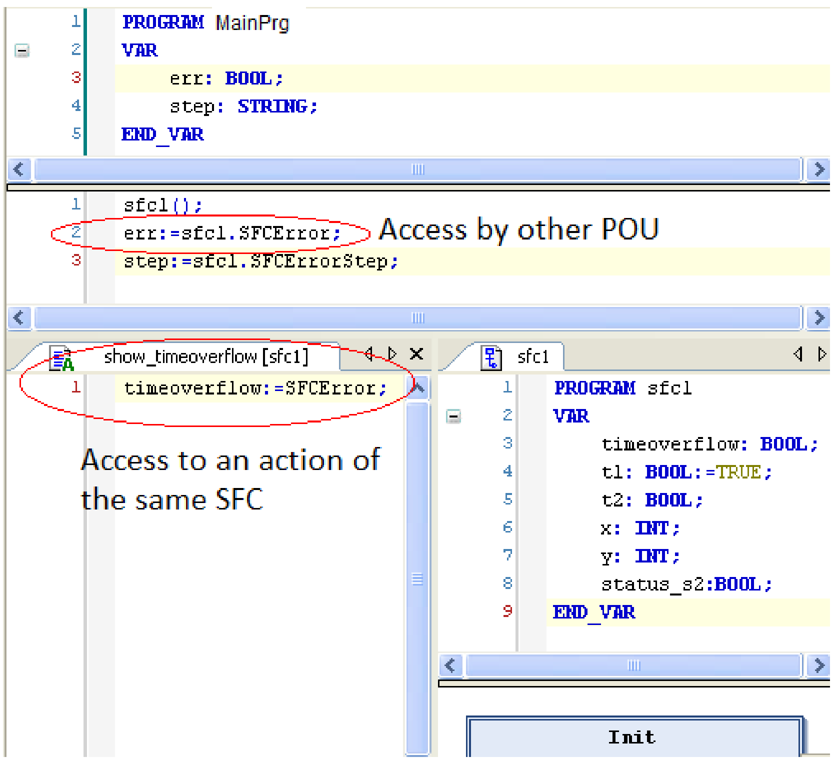
In order to allow this access, declare the flag in the *SFC Settings* dialog. For this purpose select sfc1 in the Devices/POUs window and choose command *Properties* from the context menu. Open sub dialog *SFC Settings* and there, on tab *Flags*, set a checkmark each in column *Declare* and *Use* for flag *SFCError*. Solely declaring would make the variable visible in the online view of the sfc1 declaration part, but it would be without function.



Use	Variable	Declare	Description
☐	SFCInit	☐	All steps and actions are reset. The init s
☐	SFCReset	✓	All steps and actions are reset. The init s
✓	SFCError	☐	Gets 'TRUE', if a time check failed.
✓	SFCEnableLimit	✓	Enable time check on steps
✓	SFCErrorStep	✓	Contains the name of the step that caus
☐	SFCErrorPOU	✓	Contains the name of the POU that cause

Figure 5-34. SFC Settings

Now you can read SFCError within the SFC, for example in an action, via SFCError, or from another POU via sfc1.SFCError.



```
PROGRAM MainPrg
VAR
    err: BOOL;
    step: STRING;
END_VAR
```

```
sfc1();
err:=sfc1.SFCError;
step:=sfc1.SFCErrorStep;
```

Access by other POU

show_timeoverflow [sfc1]

```
timeoverflow:=SFCError;
```

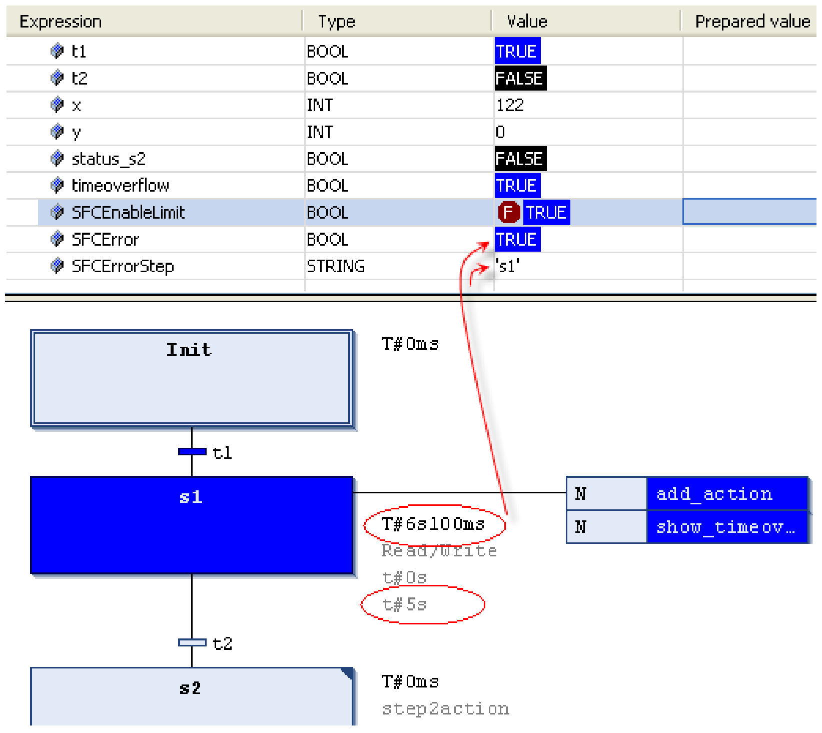
Access to an action of the same SFC

sfc1

```
PROGRAM sfc1
VAR
    timeoverflow: BOOL;
    t1: BOOL:=TRUE;
    t2: BOOL;
    x: INT;
    y: INT;
    status_s2:BOOL;
END_VAR
```

Figure 5-35. Accessing SFCError

SFCError will get TRUE as soon as a timeout within sfc2 occurs.



Expression	Type	Value	Prepared value
t1	BOOL	TRUE	
t2	BOOL	FALSE	
x	INT	122	
y	INT	0	
status_s2	BOOL	FALSE	
timeoverflow	BOOL	TRUE	
SFCEnableLimit	BOOL	F TRUE	
SFCError	BOOL	TRUE	
SFCErrorStep	STRING	's1'	

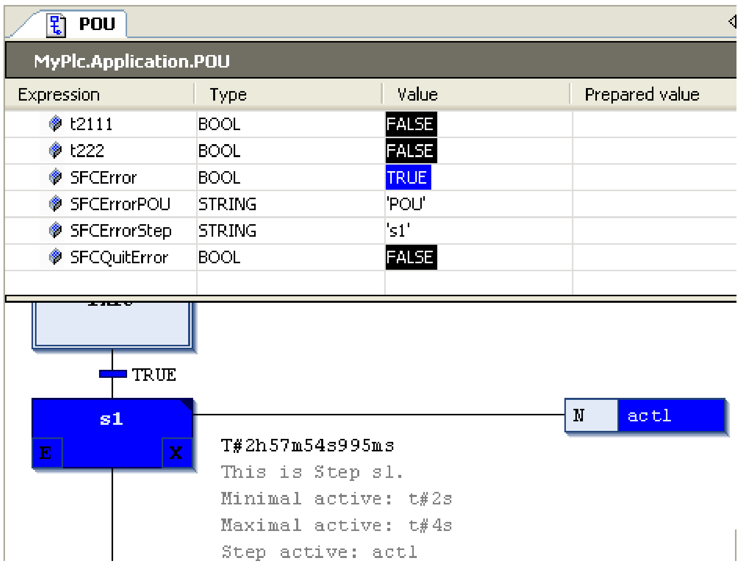
Figure 5-36. Online view of SFC sfc1

The following implicit variables (flags) can be used. For this purpose they must be declared and activated in the *SFC Settings*:

Variable	Description
SFCInit: BOOL;	If this variable gets TRUE, the sequential function chart will be set back to the Init step. All steps and actions and other SFC flags will be reset (initialization). The Init step will remain active, but not be executed as long as the variable is TRUE. SFCInit must be set back to FALSE in order to get back to normal processing.
SFCReset: BOOL;	This variable behaves similarly to SFCInit. Unlike the latter however, further processing takes place after the initialization of the Init step. Thus in this case for example a reset to FALSE of the SFCReset flag could be done in the Init step.
SFCError: BOOL;	As soon as any timeout occurs at one of the steps in the SFC, this variable will get TRUE. Precondition: SFCEnableLimit must be TRUE. Notice that any further timeout cannot be registered before a reset of SFCError. SFCError must be defined, if you want to use the other time-controlling flags (SFCErrorStep, SFCErrorPOU, SFCQuitError).
SFCEnableLimit: BOOL;	This variable can be used for the explicit activation (TRUE) and deactivation (FALS) of the time control in steps via SFCError. This means, that If this variable is declared and activated (SFC Settings) then it must be set TRUE in order to get SFCError working. Otherwise any timeouts of the steps will not be registered. The usage might be reasonable during start-ups or at manual operation. If the variable is not defined, SFCError will work automatically. Of course as a precondition SFCError must be defined!
SFCErrorStep: STRING;	This variable stores the name of a step at which a timeout was registered by SFCError. timeout. Precondition: SFCError must be defined!
SFCErrorPOU: STRING;	This variable stores the name of the SFC POU in which a timeout has occurred. Precondition: SFCError must be defined!
SFCQuitError: BOOL;	As long as this variable is TRUE, the execution of the SFC diagram is stopped and variable SFCError will be reset. As soon as the variable has been reset to FALSE, all current time states in the active steps will be reset. Precondition: SFCError must be defined!
SFCPause: BOOL;	As long as this variable is TRUE, the execution of the SFC diagram is stopped.
SFCTrans: BOOL;	This variable gets TRUE, as soon as a transition is actuated.
SFCCurrentStep: STRING;	This variable stores the name of the currently active step, independently of the time monitoring. In case of simultaneous sequences the name of the outer right step will be registered.
SFCTip, SFCTipMode: BOOL;	This variables allow using the inching mode within the current chart. When this mode has been switched on by SFCTipMode=TRUE, you can only skip to the next step by setting SFCTip=TRUE (rising edge). As long as SFCTipMode is set to FALSE, it is possible to skip by the transitions.

Table 5-5. Implicit Variables

A timeout has been detected in step s1 in SFC object POU by flag SFCError.



Expression	Type	Value	Prepared value
t2111	BOOL	FALSE	
t222	BOOL	FALSE	
SFCError	BOOL	TRUE	
SFCErrorPOU	STRING	'POU'	
SFCErrorStep	STRING	's1'	
SFCQuitError	BOOL	FALSE	

```
T#2h57m54s995ms
This is Step s1.
Minimal active: t#2s
Maximal active: t#4s
Step active: actl
```

Figure 5-37. Example of Some SFC Error Flags in Online Mode of the Editor

Accessing Flags

For enabling access on the flags for the control of SFC execution (timeouts, reset, tip mode), the flag variables must be declared and activated as described above (Control of SFC execution).

Syntax for accessing:

- From an action or transition within the SFC POU: <step name>.<flag> and _<action name>.<flag>. Examples: status:=step1._x; checkerror:=SFCerror
- From another POU:<SFC POU>.<step name>.<flag> and <SFC POU>_<action name >.<flag>. Examples: status:=SFC_prog.step1._x; checkerror:=SFC_prog.SFCerror;

In case of write access from another POU the implicit variable additionally must be declared explicitly as a VAR_INPUT variable of the SFC POU or globally e.g. in a GVL.

Example:

Local declaration:

```
PROGRAM SFC_prog
VAR_INPUT
SFCinit:BOOL;
END_VAR
```

Or global declaration in a GVL:

```
VAR_GLOBAL
SFCinit:BOOL;
END_VAR
```

Accessing the flag in MAINPRG:

```
PROGRAM MAINPRG
VAR
setinit: BOOL;
END_VAR
SFC_prog.SFCinit:=setinit; // write access in the SFCinit on SFC_prog.
```

Sequence of Processing in SFC

In online mode the particular action types will be processed according a defined sequence (Table 5-6).

First note the following use of terms:

- Active step: A step, whose step action is being executed, is called active. In online mode active steps are filled with blue color
- Initial step: In the first cycle after a SFC POU has been called, the initial step automatically gets active and the associated step action is executed
- IEC actions are executed at least twice (the first time when they have got active, the second time - in the following cycle - when they have been deactivated)
- Alternative Branches: If the step preceding the horizontal start line of alternative branches is active, then the first transition of each particular branch will be evaluated from left to right. The first transition from the left whose transition condition has value TRUE will be searched and the respective branch will be executed, that is the subsequent step within this branch will get active
- Parallel Branches: If the double-line at the beginning line of parallel branches is active and the preceding transition condition has the value TRUE, then in all parallel branches each the first step will get active. The branches now will be processed parallel to one another. The step subsequent to the double-line at the end of the branching will become active when all previous steps are active and the transition condition after the double-line has the value TRUE

Processing order of elements in a sequence:

Item	Description
Reset	All action control flags of the IEC actions get re-set (not however the flags of IEC actions that are called within actions!).
Step exit actions	All steps are checked in the order which they assume in the sequence chart (top to bottom and left to right) to determine whether the requirement for execution of the step exit action is provided, and - if that is the case - this will be executed. An exit action will be executed, if the step is going to get deactivated, that is if its entry and step actions - if existing - have been executed during the last cycle, and if the transition for the following step is TRUE.
Step entry actions	All steps are tested in the order which they assume in the sequence to determine whether the requirement for execution of the step entry action is provided and - if that is the case - this will be executed. An entry action will be executed, if the step-preceding transition condition is TRUE and thus the step has been activated.
Timeout check, Step Active Actions	For all steps, the following is done in the order which they assume in the sequence.
IEC Actions	IEC actions that are used in the sequence are executed in alphabetical order. This is done in two passes through the list of actions. In the first pass, all the IEC actions that are deactivated in the current cycle are executed. In the second pass, all the IEC actions that are active in the current cycle are executed.
Transition check, Activating next steps	Transitions are evaluated: If the step in the current cycle was active and the following transition returns TRUE (and if applicable the minimum active time has already elapsed), then the following step is activated.

Table 5-6. Processing Order of Elements in a Sequence

NOTES:
- It can come about that an action is carried out several times in one cycle because it is associated with multiple sequences. For example, an SFC could have two IEC actions A and B, which are both implemented in SFC, and which both call IEC action C; then in IEC actions A and B can both be active in the same cycle and furthermore in both actions IEC action C can be active; then C would be called twice). If the same IEC action is used simultaneously in different levels of an SFC, this could lead to undesired effects due to the processing sequence described above. For this reason, an error message is issued in this case.
- Notice the possibility of using implicit variables for controlling the status of steps and actions resp. the execution of the chart.

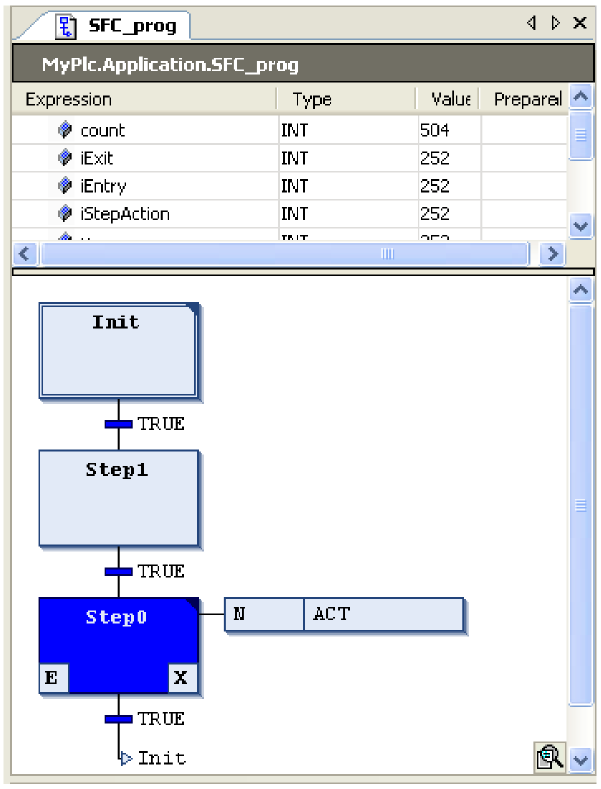
SFC Editor in Online Mode

In online mode the SFC Editor provides views for monitoring and for writing and forcing the variables and expressions on the controller. See below.

- For information on how to open objects in online mode, see **User Interface in Online Mode** in the MasterTool IEC XE User Manual – MU299609
- Notice that the editor window of an SFC object also includes the *Declaration Editor* in the upper part. For general information on the Declaration Editor in online mode, see **Declaration Editor in Online Mode** in the MasterTool IEC XE User Manual – MU299609. In case of having declared implicit variables (SFC flags) via the *SFC Settings* dialog, those will be added here, but will not be viewed in the offline mode of the *Declaration Editor*
- Also please notice the sequence of processing of the elements of a sequential function chart.
- See the object properties and the SFC Editor options and SFC defaults for settings concerning compilation resp. online display of the SFC elements and their attributes
- Consider the possible use of flags for watching and controlling the processing of an SFC

Monitoring

Active steps are displayed filled blue-colored. The display of step attributes depends on the currently set SFC editor options.



Expression	Type	Valu	Prepara
count	INT	504	
iExit	INT	252	
iEntry	INT	252	
iStepAction	INT	252	

5-38. Online View of a Program Object SFC_prog

Structured Text (ST) / Extended Structured Text (ExST)

Structured Text is a textual high-level programming language, similar to PASCAL or C. The program code is composed of expressions and instructions. In contrast to IL (Instruction List), numerous constructions can be used for programming loops, thus allowing the development of complex algorithms.

Example:

```
IF value < 7 THEN
WHILE value < 8 DO
value:=value+1;
END_WHILE;
END_IF;
```

Extended Structured Text (ExST) is a MasterTool IEC XE-specific extension to the IEC 61131-3 standard for Structured Text (ST). Examples: Assignment as expression, Set-/Reset-Operators.

Expressions

An expression is a construction which after its evaluation returns a value. This value is used in instructions.

Expressions are composed of operators, operands and/or assignments. An operand can be a constant, a variable, a function call or another expression.

Examples:

Expression	Definition
33	Constant
ivar	Variable
fct(a,b,c)	Function call
a AND b	Expression
(x*y) / z	Expression
real_var2 := int_var;	Assignment

Table 5-7. Expressions

Valuation of Expressions

The evaluation of expression takes place by means of processing the operators according to certain binding rules. The operator with the strongest binding is processed first, then the operator with the next strongest binding, etc., until all operators have been processed.

Operators with equal binding strength are processed from left to right.

Below you find a table of the ST operators in the order of their binding strength:

Operation	Symbol
Put in parentheses	(expression)
Function call	Function name (parameter list)
Exponentiation	EXPT
Negate	-
Building of complements	NOT
Multiply	*
Divide	/
Modulo	MOD
Add	+
Subtract	-
Compare	<,>,<=,>=
Equal to **Not equal to**	= <>
Boolean AND	AND
Boolean XOR	XOR
Boolean OR	OR

Table 5-8. ST Operators

Assignment as Expression

As extension to the IEC 61131-3 standard (ExST), MasterTool IEC XE allows assignments to be used as an expression.

Examples:

Expressions	Comment
int_var1 := int_var2 := int_var3 + 9;	Assignment of the result of an expression to int_var2 and int_var1
real_var1 := real_var2 := int_var;	Correct assignments, real_var1 and real_var2 will get the value of int_var
int_var := real_var1 := int_var;	This will lead to an error message because of type mismatch real-int

**IF b := (i = 1) THEN
i := i + 1;
END_IF**
	Wrong assignments: use of assignment operator to compare two expressions.

Table 5-9. Assignments Used As Expressions

Instructions

Instructions tell what to do with the given expressions. The following instructions can be used in ST.

Instruction Type	Example
Assignment	A:=B; CV := CV + 1; C:=SIN(X);

Calling a function block and use of FB output	CMD_TMR(IN := %IX5, PT := 300);
	A:=CMD_TMR.Q

RETURN	RETURN;

IF	D:=B*B;
	IF D<0.0 THEN
	C:=A;
	ELSIF D=0.0 THEN
	C:=B;
	ELSE
	C:=D;
	END_IF;

CASE	CASE INT1 OF
	um: BOOL1 := TRUE;
	2: BOOL2 := TRUE;
	ELSE
	 BOOL1 := FALSE;
	 BOOL2 := FALSE;
	END_CASE;

FOR	J:=101;
	FOR I:=1 TO 100 BY 2 DO
	IF ARR[I] = 70 THEN
	J:=I;
	EXIT;
	END_IF;
	END_FOR;

WHILE	J:=1;
	WHILE J<= 100 AND ARR[J] <> 70 DO
	J:=J+2;
	END_WHILE;

REPEAT	J:=-1;
	REPEAT
	J:=J+2;
	UNTIL J= 101 OR ARR[J] = 70
	END_REPEAT;

EXIT	EXIT;
CONTINUE	CONTINUE;

JMP	label: i:=i+1;
	JMP label;

Empty instruction	;

Table 5-10. Instructions

Assignment Operator

On the left side of an assignment there is an operand (variable, address) to which he value of the expression on the right side is assigned by the assignment operator “:=”.

See also the **MOVE** operator which does the same.

Example:

```
Var1 := Var2 * 10;
```

After completion of this line Var1 has the tenfold value of Var2.

Extended Features

Further assignment operators, which are not part of the 61131-3 standard (ExST):

Set Operator S=: The value will be "set", that is if once set to TRUE will remain TRUE.

Example:

```
a S= b;
```

Operand “a” gets the value of “b”; if once set to TRUE it will remain true, even if “b” gets FALSE again.

Reset Operator R=: The value will be reset, that is if once set to FALSE, it will remain FALSE.

Example:

```
a R= b;
```

Operand “a” gets set to FALSE when B = TRUE.

NOTE:
Notice the behavior in case of a multiple assignment: All Set and Reset assignments refer to the last member of the assignment. Example: A S= b R= fun1(par1,par2).
In this case B gets the reset output value of fun1, But “a” does not get the set value of “b”, but gets the set output value of fun1.

Notice that an assignment can be used as an expression.

Calling Function Blocks in ST

A function block (FB) is called in Structured Text according to the following syntax:

Syntax:

```
<name of instance>(FB input variable:=<value or address>|, <further FB
input variable:=<value or address>|... further FB input variables);
```

Example:

In the following example a timer function block (TON) is called with assignments for the parameters IN and PT.

Then result variable Q is assigned to variable A. The timer FB is instantiated by “TMR:TON;”

The result variable, as in IL, is addressed according to syntax <FB instance name>.< FB variable>:

```
TMR(IN := %IX5, PT := 300);
A:=TMR.Q
```

RETURN Instruction

The RETURN instruction can be used to leave a POU, for example depending on a condition.

Syntax:

```
RETURN;
```

Example:

```
IF b=TRUE THEN
RETURN;
END_IF;
a:=a+1;
```

If “b” is TRUE, instruction a:=a+1; will not be executed, the POU will be left immediately.

IF Instruction

With the IF instruction you can check a condition and, depending upon this condition, execute instructions.

Syntax:

```
IF <Boolean Expression 1> THEN
<IF Instructions>
{ELSIF <Boolean expression 2> THEN
<ELSIF instructions 1>
...
ELSIF <Boolean expression 2> THEN
<ELSIF instructions n-1>
ELSE
<ELSE instructions>}
END_IF;
```

The part in braces ({}) is optional.

If the <Boolean_expression1> returns TRUE, then only the <IF_Instructions> are executed and none of the other instructions.

Otherwise the Boolean expressions, beginning with <Boolean_expression 2> are evaluated one after the other until one of the expressions returns TRUE. Then only those instructions after this Boolean expression and before the next ELSE or ELSIF are evaluated.

If none of the Boolean expressions produce TRUE, then only the <ELSE_instructions> are evaluated.

Example:

```
IF temp<17 THEN
heating_on := TRUE;
ELSE
heating_on := FALSE;
END_IF;
```

Here the HEATING_ON variable is turned on when the temperature sinks below 17 degrees. Otherwise it remains off (FALSE).

CASE Instruction

With the CASE instructions one can combine several conditioned instructions with the same condition variable in one construct.

Syntax:

```
CASE <Var1> OF
<Value 1>:<Instruction 1>
<Value 2>:<Instruction 2>
<Value 3, Value 4, Value 5>:<Instruction 3>
<Value 6 .. Value 10>: <Instruction 4>
...
<Value n>:<Instruction n>
ELSE
<ELSE Instruction>
END_CASE;
```

A CASE instruction is processed according to the following model:

- If the variable in <Var1> has the value <Value 1>, then the instruction < Instruction 1> will be executed
- If <Var 1> has none of the indicated values, then the < ELSE instruction > will be executed
- If the same instruction is to be executed for several values of the variables, then one can write these values one after the other separated by commas and thus condition the common execution
- If the same instruction is to be executed for a value range of a variable, one can write the initial value and the end value separated by two dots. So you can condition the common condition

Example:

```
CASE INT1 OF
1, 5: BOOL1 := TRUE;
BOOL3 := FALSE;
2: BOOL2 := FALSE;
BOOL3 := TRUE;
10..20: BOOL1 := TRUE;
BOOL3:= TRUE;
ELSE
BOOL1 := NOT BOOL1;
BOOL2 := BOOL1 OR BOOL2;
END_CASE;
```

FOR Loop

With the FOR loop one can program repeated processes.

Syntax:

```
FOR <INT_Var> := <INIT_VALUE> TO <END_VALUE> {BY <STEP SIZE>} DO
<Instructions>
END_FOR;
```

The part in braces ({}) is optional.

The <INSTRUCTIONS> are executed as long as the counter <INT_Var> is not greater than the <END_VALUE>. This is checked before executing the < INSTRUCTIONS > so that the <instructions> are never executed if <INIT_VALUE> is greater than <END_VALUE>.

When <INSTRUCTIONS> are executed, <INT_Var> is increased by <STEP SIZE>. The step size can have any integer value. If it is missing, then it is set to 1. The loop must also end since <INT_Var> only becomes greater.

Example:

```
FOR Counter:=1 TO 5 BY 1 DO
Var1:=Var1*2;
END_FOR;
Erg:=Var1;
```

Let us assume that the default setting for Var1 is “1”. Then it will have the value “32” after the FOR loop.

NOTE: If <VALOR_FINAL> is equal to the limit value of counter <INT_VAR> for example if Counter - used in the example shown above - is of type SINT and if < VALOR_FINAL > is 127, then you will get an endless loop. So, < VALOR_FINAL > must not be equal to the limit value of the counter.

The CONTINUE instruction can be used within a FOR loop.

WHILE Loop

The WHILE loop can be used like the FOR loop with the difference that the break-off condition can be any Boolean expression. This means you indicate a condition which, when it is fulfilled, the loop will be executed.

Syntax:

```
WHILE <Boolean expression> DO
   <Instructions>
END_WHILE;
```

The <INSTRUCTIONS> are repeatedly executed as long as the <BOOLEAN EXPRESSION> returns TRUE. If the <BOOLEAN EXPRESSION>is already FALSE at the first evaluation, then the <INSTRUCTIONS> are never executed. If <BOOLEAN EXPRESSION> never assumes the value FALSE, then the <INSTRUCTIONS> are repeated endlessly which causes a relative time delay.

NOTE: The programmer must make sure that no endless loop is caused. He does this by changing the condition in the instruction part of the loop, for example, by counting up or down one counter.

Example:

```
WHILE Counter<>0 DO
   Var1 := Var1*2;
   Counter := Counter-1;
END_WHILE
```

The WHILE and REPEAT loops are, in a certain sense, more powerful than the FOR loop since one doesn't need to know the number of cycles before executing the loop. In some cases one will, therefore, only be able to work with these two loop types. If, however, the number of the loop cycles is clear, then a FOR loop is preferable since it allows no endless loops.

The CONTINUE instruction can be used within a WHILE loop.

REPEAT Loop

The REPEAT loop is different from the WHILE loop because the break-off condition is checked only after the loop has been executed. This means that the loop will run through at least once, regardless of the wording of the break-off condition.

Syntax:

```
REPEAT
<Instructions>
UNTIL <Boolean expression>
END_REPEAT;
```

<INSTRUCTIONS> are carried out until the < BOOLEAN EXPRESSION > returns TRUE.

If < BOOLEAN EXPRESSION > is produced already at the first TRUE evaluation, then <INSTRUCTIONS> are executed only once. If < BOOLEAN EXPRESSION > never assumes the value TRUE, then the < INSTRUCTIONS > are repeated endlessly which causes a relative time delay.

NOTE: The programmer must make sure that no endless loop is caused. He does this by changing the condition in the instruction part of the loop, for example by counting up or down one counter.

Example:

```
REPEAT
Var1 := Var1*2;
Counter := Counter-1;
UNTIL
Counter=0
```

```
END_REPEAT:
```

The CONTINUE instruction can be used within a REPEAT loop.

CONTINUE Instruction

As an extension to the IEC 61131-3 standard (ExST) the CONTINUE instruction is supported within FOR, WHILE and REPEAT-loops.

CONTINUE makes the execution proceed with the next loop-cycle.

Example:

```
FOR Counter:=1 TO 5 BY 1 DO
INT1:= INT1/2;
IF INT1=0 THEN
CONTINUE; (* To avoid division by zero *)
END_IF
Var1:=Var1/INT1; (* Only executed, if INT1 is not "0" *)
END_FOR;
Erg:=Var1;
```

EXIT Instruction

If the EXIT instruction appears in a FOR, WHILE, or REPEAT loop, then the innermost loop is ended, regardless of the break-off condition.

JMP Instruction

The JMP instruction can be used for an unconditional jump to a code line marked by a jump label.

Syntax:

```
<LABEL>:
JMP <LABEL>;
```

<LABEL> is an arbitrary, but unambiguous identifier that is placed at the beginning of a program line. The instruction JMP has to be followed by the indication of the jump destination that has to equal a predefined label. When arriving the JMP instruction a flyback to the program line that is provided with the indicated label will be effected.

NOTE: The programmer has to avoid the creation of endless loops, for example by subjecting the jump to an IF condition.

Example:

```
i:=0;
label1: i:=i+1;
(*Instructions*)
IF (i<10) THEN
JMP label1;
END_IF
```

As long as the variable “i” being initialized with 0 has a value less than 10, the conditional jump instruction of the example above will effect a repeated flyback to the program line provided with label label1 and therefore it will effect a repeated processing of the instructions comprised between the label and the JMP instruction. Since these instructions include also the increment of the variable “i”, we can be sure that the jump condition will be FALSE (at the 9th check) and program flow will be proceeded.

This functionality may also be achieved by using a WHILE or REPEAT loop in the example. Generally the use of jump instructions can and should be avoided, because they reduce the readability of the code.

Comments in ST

There are two possibilities to write comments in a Structured Text object.

- Start the comment with “(*” and close it with “*)”.This allows comments which run over several lines. Example:

```
(* This is a comment. *)
```

- Single line comments as an extension to the IEC 61131-3 standard: “//” denotes the start of a comment that ends with the end of the line. Example:

```
// This is a comment.
```

The comments can be placed everywhere within the declaration or implementation part of the ST-Editor.

Nested comments: Comments can be placed within other comments.

Example:

```
(*
a:=inst.out; (*to be checked *)
b:=b+1;
*)
```

In this example the comment that begins with the first bracket is not closed by the bracket following checked, but only by the last bracket.

ST Editor

The ST-Editor is used to create programming objects in the IEC programming language Structured Text (ST) resp. Extended Structured Text which provides some extensions to the IEC 61131-3 standard.

The ST-Editor is a text editor and thus the corresponding text editor settings in the *Options* dialog can be used to configure behavior, appearance and menus. There you can define the default settings for highlight coloring, line numbers, tabs, indenting and many more.

Notice that block selection is possible by pressing <ALT> while selecting the desired text area with the mouse.

The editor will be available in the lower part of a window which also includes the *Declaration Editor* in the upper part.

Notice that in case of syntactic errors during editing the corresponding messages will be displayed in the *Precompile Messages* window. An update of this window is done each time you re-set the focus to the editor window (for example put cursor in another window and then back to the editor window).

ST Editor in Online Mode

In online mode the Structured Text Editor (ST-Editor) provides views for monitoring and for writing and forcing the variables and expressions on the controller. Debugging functionality (breakpoints, stepping etc.) is available. See below.

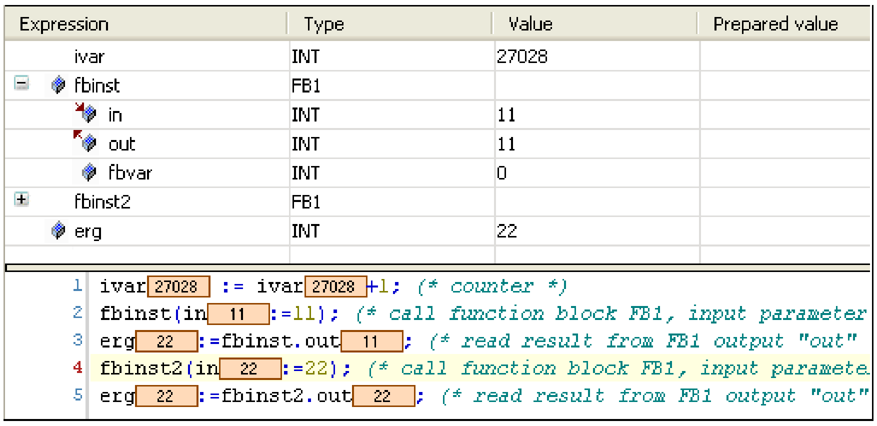
For information on how to open objects in online mode see **User Interface in Online Mode** in the MasterTool IEC XE User Manual– MU299609.

For information on how to enter prepared values for variables in online mode see the item **Forcing of Variables**.

Notice that the Editor window of an ST object also includes the Declaration Editor in the upper part. For information on the Declaration Editor in online mode see **Declaration Editor in Online Mode** in the MasterTool IEC XE User Manual– MU299609.

Monitoring

If the inline monitoring is not explicitly de-activated in the *Options* dialog, small monitoring windows will be displayed behind each variable showing the actual value (inline monitoring).

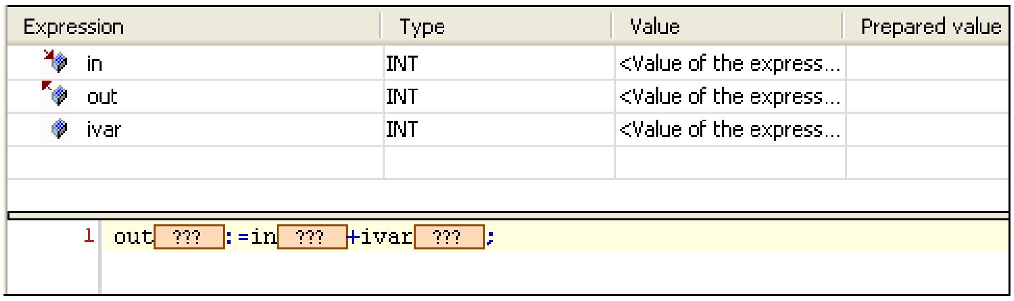


Expression	Type	Value	Prepared value
ivar	INT	27028	
fbinst	FB1		
in	INT	11	
out	INT	11	
fbvar	INT	0	
fbinst2	FB1		
erg	INT	22	

```
ivar 27028 := ivar 27028 +1; (* counter *)
fbinst(in 11 :=11); (* call function block FB1, input parameter
erg 22 :=fbinst.out 11 ; (* read result from FB1 output "out"
fbinst2(in 22 :=22); (* call function block FB1, input paramete
erg 22 :=fbinst2.out 22 ; (* read result from FB1 output "out"
```

Figure 5-39. Online view of a Program Object (MainPrg)

Online view of a function block POU: No values will be viewed. Instead the term <*Value of the expression*> will be displayed in column *Value* and the inline monitoring fields in the implementation part will show three question marks each.



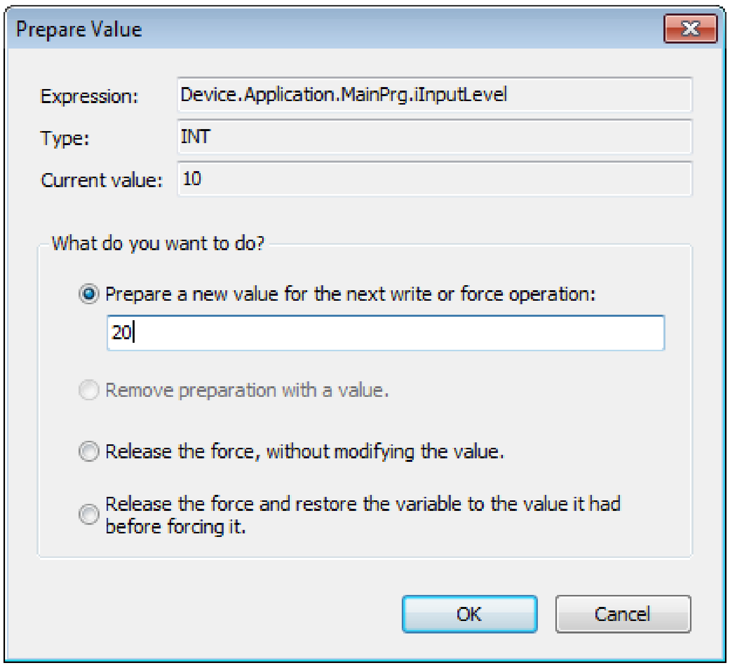
Expression	Type	Value	Prepared value
in	INT	<Value of the express...	
out	INT	<Value of the express...	
ivar	INT	<Value of the express...	

```
out ??? :=in ??? +ivar ??? ;
```

Figure 5-40. Online view of a Function Block (FB1)

Forcing of Variables

In addition to the possibility to enter a prepared value for a variable within the declaration of any editor the ST-Editor provides to click on the monitoring box of a variable within the implementation part (in online mode), whereon you may enter the prepared value in the rising dialogue (Figure 5-41).



Prepare Value

Expression: Device.Application.MainPrg.iInputLevel

Type: INT

Current value: 10

What do you want to do?

- [✓ Prepare a new value for the next write or force operation:] 20
- Remove preparation with a value.
- Release the force, without modifying the value.
- Release the force and restore the variable to the value it had before forcing it.

Figure 5-41. Prepare Value Dialog

You find the name of the variable completed by its path within the *Device* tree (*Expression*), its type and current value. By activating the corresponding item you may choose:

- *Prepare a new value for the next write or force operation:*
- *Remove a preparation with a value.*
- *Release the force, without modifying the value.*
- *Release the force and restore the variable to the value it had before forcing it.*

The selected action will be carried out on executing the menu command *Force Values* (menu *Online*) or pressing <F7>.

Breakpoint Positions in ST Editor

The user can set a breakpoint basically at those positions in a POU at which values of variables can change or at which the program flow branches out resp. another POU is called. In the following descriptions “{BP}” indicates a possible breakpoint position:

- Assignment: At the beginning of the line. Assignments as expressions define no further breakpoint positions within a line

FOR loop: before the initialization of the counter, before the test of the counter and before a statement.

```
{BP} FOR i := 12 TO {BP} x {BP} BY 1 DO
{BP} [statement 1]
...
{BP} [statement -2]
END_FOR
```

- WHILE loop: before the test of the condition and before a statement

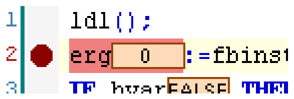
```
{BP} WHILE i < 12 DO
{BP} [statement 1]
...
{BP} [statement -1]
END_WHILE
```

- REPEAT loop: before the test of the condition

```
REPEAT
{BP} [statement 1]
...
{BP} [statement n-1]
{BP} UNTIL i >= 12
END_REPEAT
```

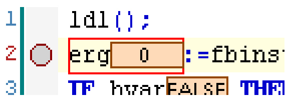
- Call of a program or a function block: At the beginning of the line
- At the end of a POU. When stepping through, this position also will be reached after a RETURN instruction

Breakpoint Display in ST



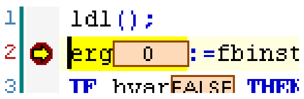
```
ldl();
erg 0 :=fbinst
IF bvarFALSE THEN
```

Figure 5-42. Breakpoint in Online Mode



```
ldl();
erg 0 :=fbins
IF bvarFALSE THEN
```

Figure 5-43. Breakpoint Disabled



```
ldl();
erg 0 :=fbinst
IF bvarFALSE THEN
```

Figure 5-44. Program Stop at Breakpoint

NOTE: The following must be noticed for breakpoints in methods: A breakpoint will be set automatically in all methods which might be called. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing the method.

FBD/LD/IL Editor

The FBD/LD/IL editor provides commands for working in the combined editor for Function Block Diagram (FBD), Ladder Logic Diagram (LD) and Instruction List (IL).

Function Block Diagram - FBD

The Function Block Diagram is a graphically oriented programming language. It works with a list of networks whereby each network contains a graphical structure of boxes and connection lines which represents either a logical or arithmetic expression, the call of a function block, a jump, or a return instruction.

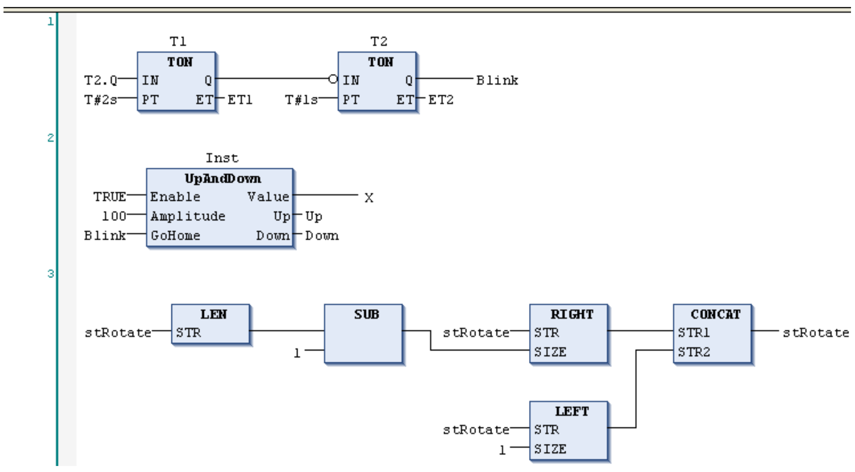


Figure 5-45. Function Block Diagram Network

Ladder Diagram - LD

The Ladder Diagram is a graphics oriented programming language which approaches the structure of an electric circuit.

The Ladder Diagram is suitable for constructing logical switches, on the other hand one can also create networks as in FBD. Therefore the LD is very useful for controlling the call of other POUs.

The Ladder Diagram consists of a series of networks, each being limited by vertical current lines (power rail) on the left and on the right. A network contains a circuit diagram made up of contacts, coils, optionally additional POUs (boxes) and connecting lines. On the left side there is a series of contacts passing from left to right the condition “ON” or “OFF” which corresponds to the Boolean values TRUE and FALSE. To each contact a Boolean variable is assigned.

In case of contacts, if this variable is TRUE, the condition is transmitted from left to right along the line connector. On the other hand if this variable is FALSE, the condition spread to the right will always be OFF.

In case of negated contacts, if this variable is TRUE, the condition spread to the right will always be OFF. However in the case where the variable is FALSE, the condition is transmitted from left to right along the line connector.

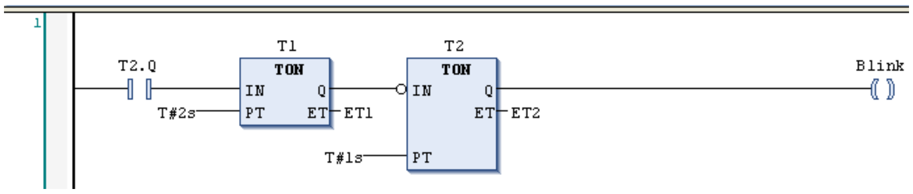


Figure 5-46. LD Network

Instruction List - IL

The Instruction List is similar to Assembly language programming, in accordance with IEC 61131-3.

This language supports programming based on an accumulator. All IEC 61131-3 operators are supported as well as multiple inputs / multiple outputs, negations, comments, set / reset of outputs and unconditional /conditional jumps.

Each instruction is primarily based on the loading of values into the accumulator by using the LD operator. After that the operation is executed with the first parameter taken out of the accumulator. The result of the operation again is available in the accumulator, from where the user should store it with the ST instruction.

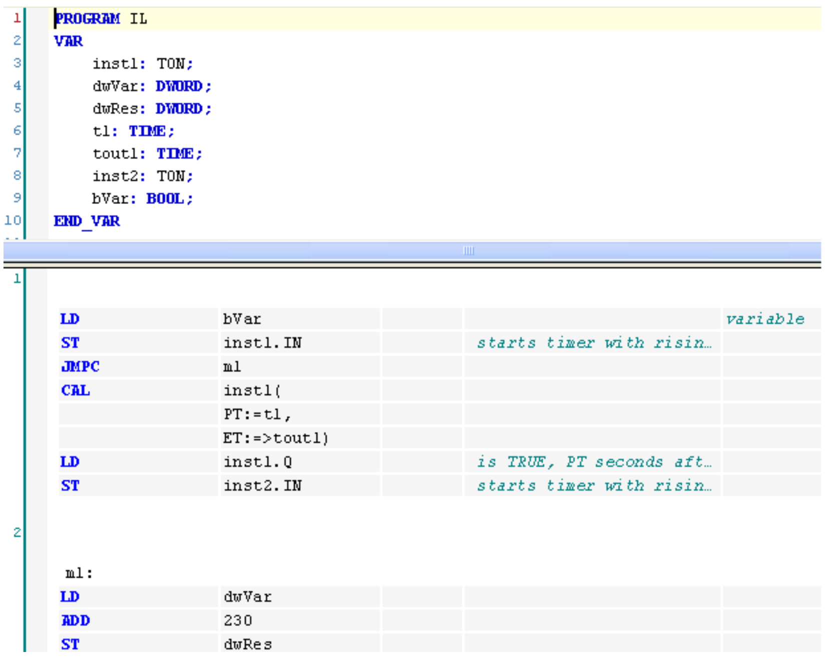
In order to program conditional executions or loops IL supports both comparing operators like EQ, GT, LT, GE, LE, NE and jumps. The latter can be unconditional (JMP) or conditional (JMPC / JMPCN). For conditional jumps the accumulator's value is checked on TRUE or FALSE.

An instruction list (IL) consists of a series of instructions. Each instruction begins in a new line and contains an operator and, depending on the type of operation, one or more operands separated by commas. The operator might be extended by a modifier.

In a line before an instruction there can be an identification mark (label) followed by a colon (:), for example “ml:” in the example shown below. A label can be the target of a jump instruction, for example “JMPC next” in the example shown below.

A comment must be placed as last element of a line.

Empty lines can be inserted between instructions.



```
PROGRAM IL
VAR
    inst1: TON;
    dwVar: DWORD;
    dwRes: DWORD;
    t1: TIME;
    tout1: TIME;
    inst2: TON;
    bVar: BOOL;
END_VAR
```

```
1
 LD      bVar                                        variable
 ST      inst1.IN     starts timer with risin…
 JMPC    ml
 CAL     inst1(
         PT:=t1,
         ET:=>tout1)
 LD      inst1.Q      is TRUE, PT seconds aft…
 ST      inst2.IN     starts timer with risin…

2
 ml:
 LD      dwVar
 ADD     230
 ST      dwRes
```

Figure 5-47. IL Program Example in IL Table editor

The IL Editor is a table editor integrated in the FBD/LD/IL.

Modifiers and Operators in IL

The following modifiers can be used in Instruction List:

Modifier	Context	Description
C	With JMP, CAL, RET	The instruction only will be executed if the result of the preceding expression is TRUE.
N	With JMPC, CALC, RETC	The instruction will only be executed if the result of the preceding expression is FALSE.
N	In any other case	Negation of the operand (not of the accumulator).

Table 5-11. Modifiers

The following Table 5-12 shows which operators can be used in combination with the specified modifiers.

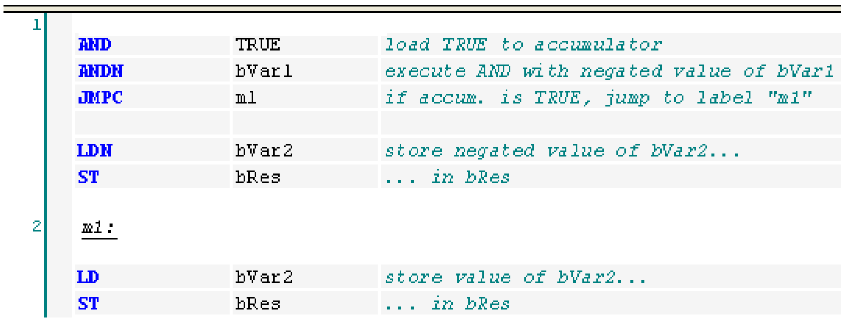
The accumulator always stores the current value, resulting from the preceding operation.

Operator	Modifiers	Meaning	Example
LD	N	Loads the (negated) value of the operand into the accumulator.	LD iVar
ST	N	Stores the (negated) content of the accumulator into the operand variable.	ST iErg
S		Sets the operand (type BOOL) to TRUE when the content of the accumulator is TRUE.	S bVar1
R		Sets the operand (type BOOL) to FALSE when the content of the accumulator is TRUE.	R bVar1
AND	N,(	Bitwise AND of the accumulator and the (negated) operand.	AND bVar2
OR	N,(	Bitwise OR of the accumulator and the (negated) operand.	OR xVar
XOR	N,(	Bitwise exclusive OR of the accumulator and the negated) operand.	XOR N,(bVar1,bVar2)
NOT		Bitwise negation of the accumulator's content.	
ADD	(	Addition of accumulator and operand. Result is copied to the accumulator.	ADD (iVar1,iVar2)
SUB	(	Subtraction of accumulator and operand. Result is copied to the accumulator.	SUB iVar2
MUL	(	Multiplication of accumulator and operand. Result is copied to the accumulator.	MUL iVar2
DIV	(	Division of accumulator and operand. Result is copied to the accumulator.	DIV 44
GT	(	Check if accumulator is greater than operand (>). Result (BOOL) is copied into the accumulator.	GT 23
GE	(	Check if accumulator is greater than or equal to the operand (>=). Result (BOOL) is copied into the accumulator.	GE iVar2
EQ	(	Check if accumulator is equal to the operand (=). Result (BOOL) is copied into the accumulator.	EQ iVar2
NE	(	Check if accumulator is not equal to the operand (<>). Result (BOOL) is copied into the accumulator.	NE iVar1
LE	(	Check if accumulator is less than or equal to the operand (<=). Result (BOOL) is copied into the accumulator.	LE 5
LT	(	Check if accumulator is less than operand (<). Result (BOOL) is copied into the accumulator.	LT cVar1
JMP	CN	Unconditional (conditional) jump to the label.	JMPN next
CAL	CN	Call (Conditional) of a PROGRAM or FUNCTION_BLOCK (if accumulator is TRUE).	CAL prog1
RET		Early return of the POU and jump back to the calling POU.	RET
RET	C	Conditional - if accumulator is TRUE…Early return of the POU and jump back to the calling POU.	RETC
RET	CN	Conditional - if accumulator is FALSE… Early return of the POU and jump back to the calling POU.	RETCN
)		Evaluate deferred Operation.	

Table 5-12. Operators and Modifiers

See **Operators**.

See also **Working in the IL Editor View** for how to use and handle multiple operands, complex operands, function / method / function block / program / action calls and jumps.



```
AND     TRUE    load TRUE to accumulator
ANDN    bVar1   execute AND with negated value of bVar1
JMPC    m1      if accum. is TRUE, jump to label "m1"

LDN     bVar2   store negated value of bVar2...
ST      bRes    ... in bRes

m1:

LD      bVar2   store value of bVar2...
ST      bRes    ... in bRes
```

Figure 5-48. IL Program with Operators

Working in the FBD e LD Editor View

Networks are the basic entities in FBD and LD programming. Each network contains a structure that displays a logical or an arithmetical expression, a POU (function, program, function block call, etc.), a jump, a return instruction.

When creating a new object, the editor window automatically contains one empty network.

Notice the general editor settings in the *Options* dialog, tab *FBD, LD and IL editor*.

The cursor being placed on the name of a variable or box parameter will prompt a tooltip showing the respective type and in case of function block instances the initialization value. For IEC operators SEL, LIMIT, MUX a short description on the inputs will appear. If defined, also the address and the symbol comment will be shown as well as the operand comment (in quotation marks in a second line).

Inserting and arranging elements:

- Elements also can be directly dragged with the mouse from the toolbox to the editor window or from one position within the editor to another ("Drag&Drop"). For this purpose select the element by a mouse-click, then keep the mouse-button pressed and drag the element into the respective network in the editor view. As soon as you have reached the network, all possible insert positions for the respective type of element will be indicated by grey position markers. When you place the mouse-cursor on one of these positions, the position marker will change to green and you can release the mouse-button in order to place the element at that position
- The *Cut*, *Copy*, *Paste* and *Delete* commands, by default available in the Edit menu, can be used to arrange elements. Copying an element is also possible by drag and drop: Select the element within a network by a mouse-click and while keeping the mouse button pressed, drag it to the target position. As soon as that is reached (green position marker), a plus-symbol will be added to the cursor symbol. Release the mouse-button to insert the element
- For all possible cursor positions see: **Cursor Positions in FBD, LD and IL**

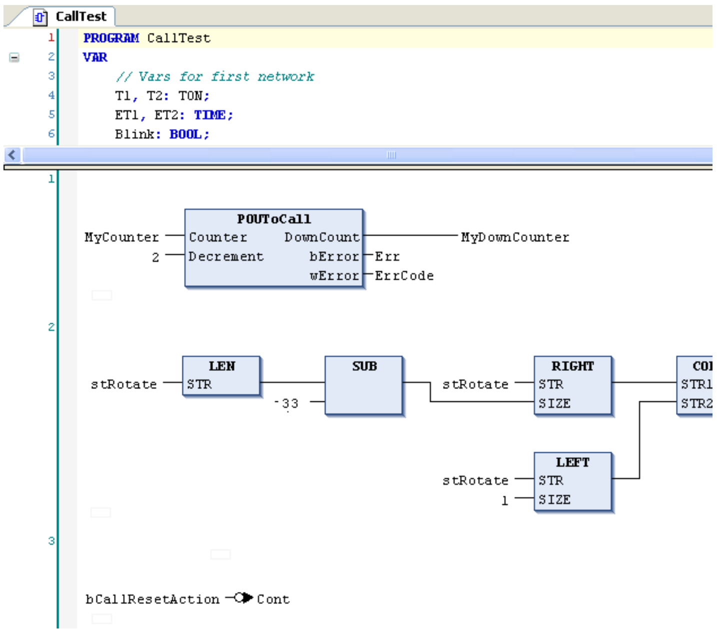
Navigating:

- The arrow keys might be used to jump to the next/previous cursor position; also possible between networks
- The <TAB> key might be used to jump to the next/previous cursor position within a network
- <CTRL>+<HOME> scrolls to the begin of the document and marks the first network
- <CTRL>+<END> scrolls to the end of the document and marks the last network
- <PAGEUP> scrolls one screen up and marks the topmost rectangle
- <PAGEDOWN> scrolls one screen down and marks the topmost rectangle

Selecting:

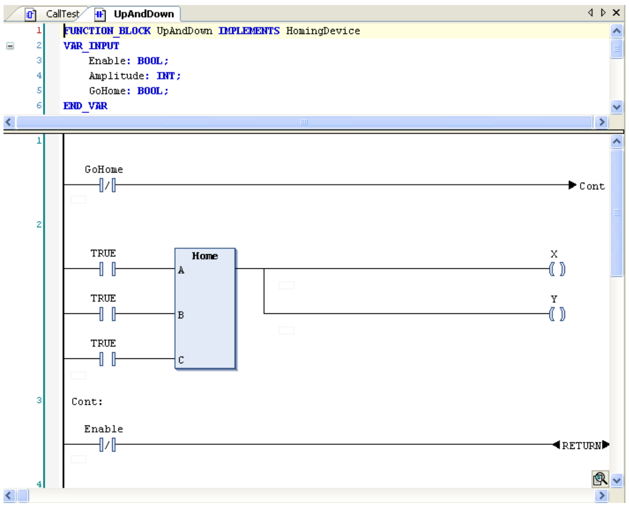
- An element, also network, can be selected via taking the respective cursor position by a mouse-click or using the arrow or tabulator keys

- Multiselection of non-adjacent elements resp. networks is possible by keeping the while selecting the desired elements one after the other
- In the LD editor multiselection of adjacent elements or networks can be done by keeping pressed the <SHIFT> key while selecting two contacts determining start and end of the desired network section. If you want to cut (copy) and paste a section of a network, it will be sufficient to keep the <CTRL> key pressed while selecting just two contacts defining the borders of this section. Then the elements between will be noticed automatically



```
PROGRAM CallTest
VAR
    // Vars for first network
    T1, T2: TON;
    ET1, ET2: TIME;
    Blink: BOOL;
```

Figure 5-49. FBD Editor Window



```
FUNCTION_BLOCK UpAndDown IMPLEMENTS HomingDevice
VAR_INPUT
    Enable: BOOL;
    Amplitude: INT;
    GoHome: BOOL;
END_VAR
```

Figure 5-50. LD Editor Window

For information on the languages see:

- **Function Block Diagram - FBD**
- **Ladder Diagram - LD**

Working in the IL Editor View

The IL (Instruction List) editor is a table editor in contrast to the pure text editor used in MasterTool IEC. The network structure of FBD or LD programs is also represented in an IL program. Basically one network is sufficient in an IL program, but considering switching between FBD, LD and IL you also might consciously use networks for structuring an IL program.

Notice the general editor settings in the *Options* dialog, tab *FBD, LD and IL editor*.

Tooltip containing information on variables or box parameters: Please see **Working in the FBD e LD Editor View**.

Inserting and arranging elements:

The commands for working in the editor by default are available in the *FBD/LD/IL* menu, the most important always also in the context menu.

Programming units, that is elements, are inserted each at the current cursor position via the *Insert* command, by default available in the *FBD/LD/IL* menu.

The *Cut*, *Copy*, *Paste* and *Delete* commands, by default available in the *Edit* menu, can be used to arrange elements.

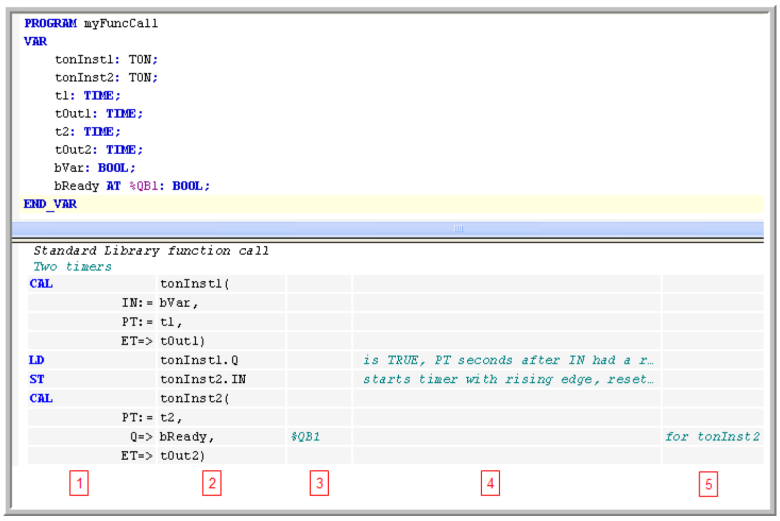
See below to the tabular editor is structured and how you can navigate through it using complex operands, calls and jumps.

Structure of the do IL Tabular Editor

Each program line is written in a table row, structured in fields by the following table columns:

Column	Contains	Description
1	Operator	This field contains the IL operator (LD, ST, CAL, AND, OR etc.) or a function name. In case of a function block call here also the respective parameters are specified, in this case in the Prefix field ":=" or "=>" must be entered.
2	Operand	This field contains exactly one operand or a jump label. If more than one operand is needed (multiple/extensible operators "AND A, B, C" or function calls with several parameters), those must be written into the following lines where the operator field is to be left empty. In this case add a parameter-separating comma. In case of a function block, program or action call the appropriate opening and/or closing brackets must be added.
3	Address	This field contains the address of the operand as defined in the declaration part. The field cannot be edited and can be switched on or off via option 'Show symbol address'.
4	Symbol comment	This field contains the comment as defined for the operand in the declaration part. The field cannot be edited and can be switched on or off via option 'Show symbol comment'.
5	Operand comment	This field contains the comment for the current line. It is editable and can be switched on or off via option 'Show operand comment'.

Table 5-13. Structure of the do IL Tabular Editor



```
PROGRAM myFuncCall
VAR
    tonInst1: TON;
    tonInst2: TON;
    t1: TIME;
    tOut1: TIME;
    t2: TIME;
    tOut2: TIME;
    bVar: BOOL;
    bReady AT %QB1: BOOL;
END_VAR
```

```
 Standard Library function call
 Two timers
CAL          tonInst1(
        IN:= bVar,
        PT:= t1,
        ET=> tOut1)
LD           tonInst1.Q                is TRUE, PT seconds after IN had a r…
ST           tonInst2.IN               starts timer with rising edge, reset…
CAL          tonInst2(
        PT:= t2,
         Q=> bReady,    %QB1                                                   for tonInst2
        ET=> tOut2)
```

Figure 5-51. IL Tabular Editor

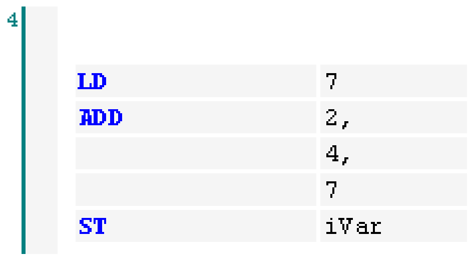
Navigating:

- <↑> and <↓> keys: Moving to the field above/ below
- <TAB>: Moving within a line to the field to the right
- <SHIFT> + <TAB>: Moving within in a line to the field to the left
- <SPACE>: Open the currently selected field for editing. Alternatively perform a further mouse-click on the field. If applicable the input assistant will be available via the button …. A currently opened edit field can be closed by <ENTER> confirming the current entries, or by <ESC>, cancelling the made entries
- <CTRL> + <ENTER>: Enter a new line below the current one
- <DELETE>: Remove the current line, that is where you have currently selected any field
- *Cut*, *Copy*, *Paste*: To copy one or several lines select at least one field of the line(s) and use the copy command. To cut a line, use the *Cut* command. *Paste* will insert the previously copied/cut lines before the line where currently a field is selected. If no field is selected they will be inserted at the end of the network
- <CTRL> + <HOME>: scrolls to the begin of the document and marks the first network
- <CTRL> + <END>: scrolls to the end of the document and marks the last network
- <PAGEUP>: scrolls one screen up and marks the topmost rectangle
- <PAGEDOWN>: scrolls one screen down and marks the topmost rectangle

Multiple Operands (Extensible Operators)

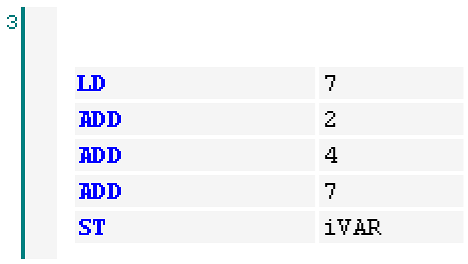
If the same operator is used with multiple operands, two ways of programming are possible:

- The operands are entered in subsequent lines, separated by commas, example:



```
4
LD     7
ADD    2,
       4,
       7
ST     iVar
```

- The instruction is repeated in subsequent lines, example:

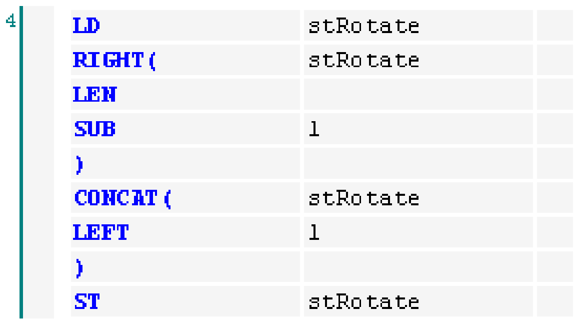


LD	7
ADD	2
ADD	4
ADD	7
ST	iVAR

Complex Operands

If a complex operand is to be used, enter an opening bracket before, then use the following lines for the particular operand components and below those, in a separate line enter the closing bracket.

Example: rotating a string by 1 character at each cycle.



LD	stRotate	
RIGHT(	stRotate	
LEN		
SUB	1	
)		
CONCAT(	stRotate	
LEFT	1	
)		
ST	stRotate	

Function Calls

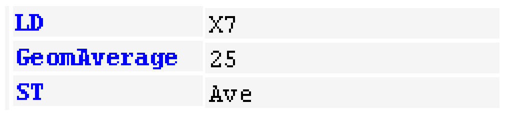
Enter the function name in the operator field. The first input parameter is to be given as an operand in a preceding LD operation. If there are further parameters, the next one must be given in the same line as the function name. The further ones can also be added in this line, separated by commas, or in subsequent lines.

The function return value will be stored in the accumulator, but notice the following restriction concerning the IEC standard: A function call with multiple return values is not possible, only one return value can be used for a succeeding operation.

Example: Function GeomAverage, which has three input parameters, is called. The first parameter is given by X7 in a preceding operation, the second one, 25 is given behind the function name. The third one is given by variable tvar, either in the same line or in the subsequent one. The return value is assigned to variable Ave.

Example for function call GEOMAVERAGE in ST:

```
Ave := GeomAverage(X7, 25, tvar);
Example for function call GEOMAVERAGE in IL:
```



LD	X7
GeomAverage	25
ST	Ave

Function Block Calls, Program Calls

Use the CAL- or CALC operator. Enter the function block instance name resp. the program name in the operand field (second column) followed by the opening bracket. Enter the input parameters each in one of the following lines:

First column: operator (parameter name) and:

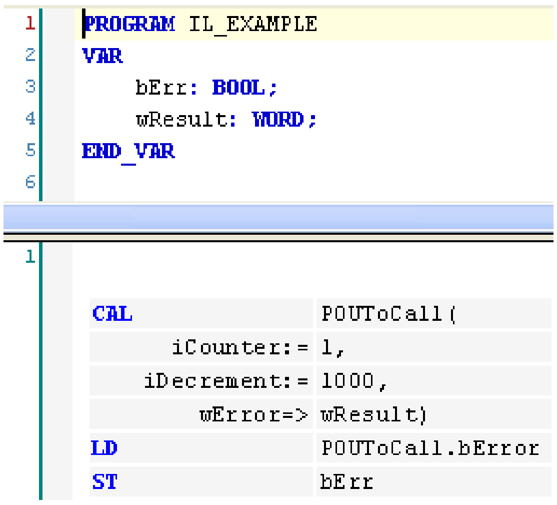
- “:=” for input parameters

- “=>” for output parameters

Second column: operand (actual parameter) and:

- “,” if further parameters follow
- “)” after the last parameter
- “()” in case of parameter-less calls

Example: Call of POUToCAll with two input and two output parameters.



```
PROGRAM IL_EXAMPLE
VAR
    bErr: BOOL;
    wResult: WORD;
END_VAR
```

```
CAL                POUToCall(
        iCounter:= 1,
      iDecrement:= 1000,
          wError=> wResult)
LD                 POUToCall.bError
ST                 bErr
```

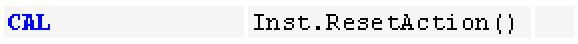
It is not necessary to use all parameters of a function block or program.

NOTE: As a restriction to the IEC standard complex expressions cannot be used, those must be assigned to the input of the function block or program before the call instruction.

Action Call

To be done like a function block or program call. The action name is to be appended to the instance name or program name.

Example of calling the action ResetAction in IL:

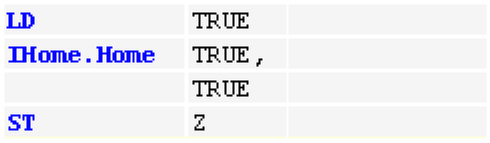


```
CAL                Inst.ResetAction()
```

Method Call

To be done like a function call. The instance name with appended method name is to be entered in the first column (operator).

Example of calling the method Home in IL:

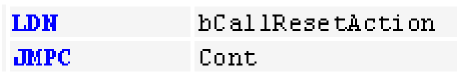


```
LD                 TRUE
IHome.Home         TRUE,
                   TRUE
ST                 Z
```

Jump

A jump is programmed by JMP in the first column (operator) and a label name in the second column (operand). The label is to be defined in the target network in the label field. Remark that the statement list preceding the unconditional jump has to end with one of the following commands: ST, STN, S, R, CAL, RET or another JMP. This is not the case for a conditional jump being programmed by JMPC in the first column (operator) instead of JMP. The execution of the jump depends on the value loaded.

Example: Conditional jump instruction; in case bCallRestAction is TRUE, the program should jump to the network labeled with Cont:



```
LDN     bCallResetAction
JMPC    Cont
```

Cursor Positions in FBD, LD and IL

IL Editor

This is a text editor, structured in form of a table. Each table cell is a possible cursor position. See also: **Working in the IL Editor View**.

FBD and LD Editors

These are graphic editors, see below examples to showing the possible cursor positions: text, input, output, box, contact, coil, line between elements, sub-network, network. Actions like Cut, Copy, Paste, Delete and other editor-specific commands refer to the current cursor position resp. selected element. See: **Working in the FBD e LD Editor View**.

Basically in FBD a dotted rectangle around the respective element indicates the current position of the cursor, additionally texts and boxes get blue and red-shadowed.

In LD coils and contacts get red-colored as soon as the cursor is positioned on.

The cursor position determines which elements are available in the context menu for getting inserted.

Possible cursor positions:

Every text field: in the left picture the possible cursor positions are marked by a red-frame, the right picture shows a box with the cursor being placed in the “AND” field. Notice the possibility to enter addresses instead of variables names if configured appropriately in the *Options* dialog, tab *FBD and LD Editor*.

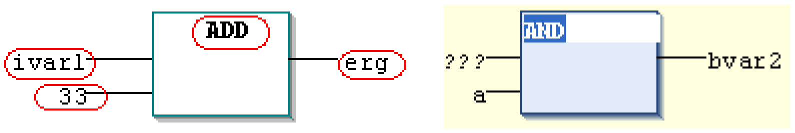


Figure 5-52. Text Fields

- Every input:

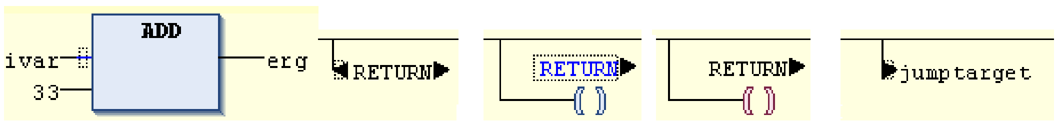


Figure 5-53. Inputs

- Every operator, function, or function block:

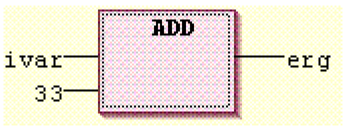


Figure 5-54. Operator, Function or Function Block

- Outputs, if an assignment or a jump comes afterward:

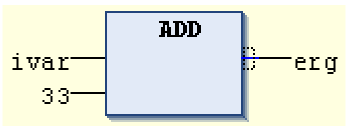


Figure 5-55. Output

- Just before the lined cross above an assignment, before a jump or a return instruction:

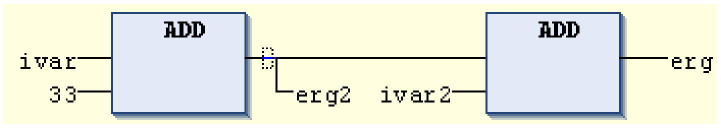


Figure 5-56. Before the Crossed Line

- The right-most cursor position in the network or anywhere else in the network besides the other cursor positions. This will select the whole network:

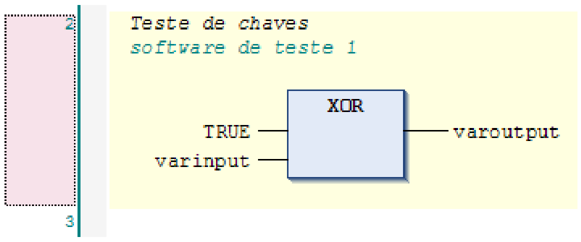


Figure 5-57. Cursor Position (Right in the Network)

- The lined cross directly in front of an assignment:

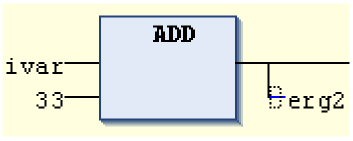


Figure 5-58. Front of Assignment

- Every contact:

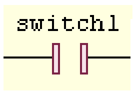


Figure 5-59. Contact

- Every coil:

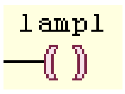


Figure 5-60. Coil

- The connecting line between the contacts and the coils.

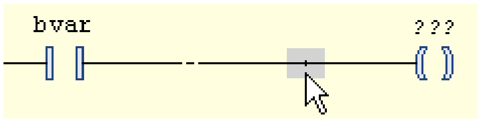


Figure 5-61. Connecting Line Position

- Branch/ sub network within a network:

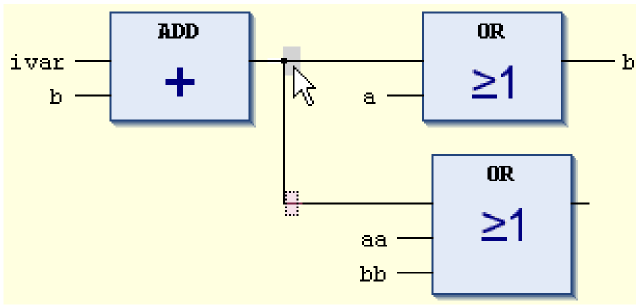


Figure 5-62. Branch or Sub network

FBD/LD/IL Menu

When the cursor is placed in the FBD/LD/IL editor window, the FBD/LD/IL menu by default is available in the menu bar, providing the commands for programming in the currently set editor view.

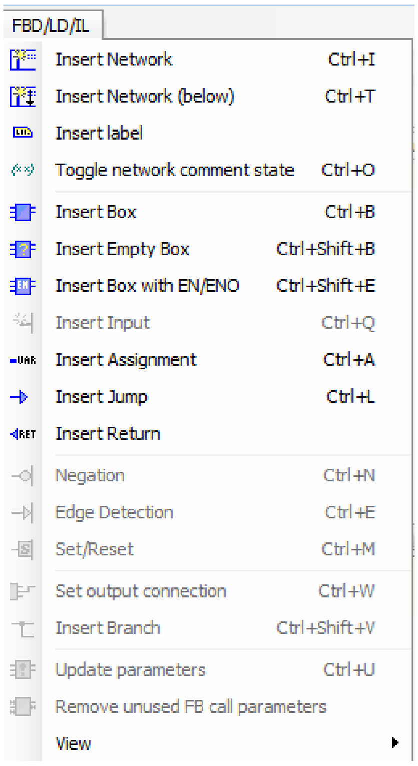

FBD/LD/IL
Insert Network Ctrl+I
Insert Network (below) Ctrl+T
Insert label
Toggle network comment state Ctrl+O
Insert Box Ctrl+B
Insert Empty Box Ctrl+Shift+B
Insert Box with EN/ENO Ctrl+Shift+E
Insert Input Ctrl+Q
Insert Assignment Ctrl+A
Insert Jump Ctrl+L
Insert Return
Negation Ctrl+N
Edge Detection Ctrl+E
Set/Reset Ctrl+M
Set output connection Ctrl+W
Insert Branch Ctrl+Shift+V
Update parameters Ctrl+U
Remove unused FB call parameters
View

Figure 5-63. FBD/LD/IL Menu in FBD Editor View

For a description of the commands see: **Editor FBD/LD/IL Commands** in the MasterTool IEC XE User Manual - MU299609.

Elements

FBD/LD/IL Toolbox

The FBD/LD/IL editor provides a toolbox which offers the programming elements for being inserted in the editor window by drag&drop. The toolbox by default can be opened via command *Toolbox* in the *View* menu.

It depends on the currently active editor view which elements are available for inserting (see the respective description of the *Insert* commands). The elements are sorted in categories: *General* (general elements like Network, Assignment etc..), *Boolean operators*, *Math operators*, *Other operators* (SEL, MUX, LIMIT and MOVE), *Function blocks* (R_TRIG, F_TRIG, RS, SR, TON, TOF, CTD, CTU), *Ladder elements* and *POUs* (user-defined).

The POUs category lists all POUs, which have been defined by the user below the same application as the FBD/LD/IL object which is currently opened in the editor. If a POU has got assigned a bitmap in its properties, then this will be displayed before the POU name, otherwise the standard icon for indicating the POU type. The list will be updated automatically when POUs are added or removed from the application.

The category folders can be unfolded by a mouse-click on the button showing the respective category name. See in the Figure 5-64: Category *General* currently is unfolded, the others are folded. The picture shows an example for inserting an Assignment element by drag&drop from the toolbox. Currently only section *General* in the toolbox is unfolded.

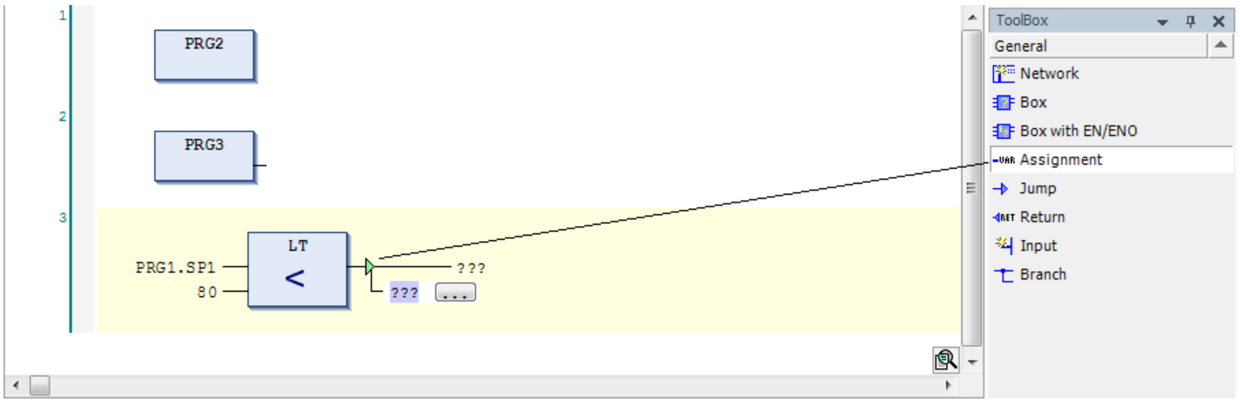


Figure 5-64. Inserting from Toolbox

To insert an element in the editor, select it in the toolbox by a mouse-click and by drag&drop bring it to the editor window. The possible insert positions will be indicated by position markers, which appear as long as the element is drawn - keeping the mouse button pressed - across the editor window. Always the nearest possible position will light up green. When leaving the mouse button the element will be inserted at the currently green position.

If you draw a box element on an existing box element, the new one will replace the old one; if inputs and outputs already have been assigned, those primarily will remain as defined.

Network

A network is the basic entity of a FBD or LD program. In the FBD/LD editor the networks are arranged in a vertical list. Each network is designated on the left side by a serial network number and has a structure consisting of either a logical or an arithmetic expression, a program, function or function block call, and a jump or a return instruction.

The IL editor, due to the common editor base with the FBD and LD editors, also uses the network element. If an object initially was programmed in FBD or LD and then is converted to IL, the networks will be still present in the IL program. Vice versa, if you start programming an object in IL, you at least need 1 network element which might contain all instructions, but you also can use further networks to structure the program, for example if considering a conversion to FBD or LD.

A network optionally can get assigned a title, a comment and a label:

The availability of the title and the comment fields can be switched on and off in the *Options* dialog, tab *FBD, LD, IL Editor*. If the option is activated, you can open an edit field for the title by a mouse-click in the network directly below the upper border. For entering a comment correspondingly open an edit field directly below the title field. The comment might be multi-lined. Linebreaks can be inserted via <ENTER>, the input of the comment text is terminated by <CTRL>+<ENTER>.

To add a label, which then can be addressed by a jump, use the command *Insert label*. If a label is defined, it will be displayed below the title and comment field resp. - if those are not available - directly below the upper border of the network.

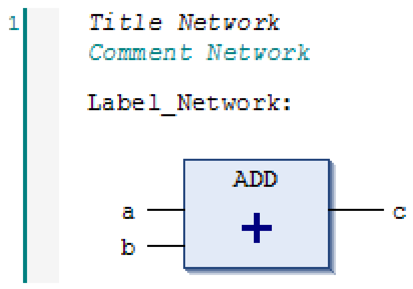


Figure 5-65. Positions of Title, Comment and Label in a Network

A network can be set in comment state, which effects that the network is not processed but displayed and handled like a comment.

On a currently selected network the default commands for *Copy*, *Cut*, *Insert* and *Delete* can be applied.

NOTE: The right mouse click being executed over a title, comment or label will select this entry only instead of the whole network. So the execution of the default commands will have no influence on the network itself.

To insert a network, use command *Insert Network* or drag it from the toolbox. A network with all belonging elements can also be copied or moved by drag&drop within the editor.

Notice the possibility to create sub networks by inserting branches.

RET Network

In online mode automatically an additional, empty network will be displayed below the existing networks. Instead of a network number it is identified by RET. It represents the position at which the execution will return to the calling POU and provides a possible halt position.

Assignment in FBD/LD/IL

Depending on the selected position in FBD or LD an assignment will be inserted directly in front of the selected input, directly after the selected output or at the end of the network. In an LD network an assignment will be displayed as a coil.

Alternatively drag the assignment element from the toolbox or copy or move it by drag&drop within the editor view.

After insertion the text string “???” can be replaced by the name of the variable that is to be assigned. For this via the button you can use the *Input Assistant* (...).

In IL an assignment is programmed via LD and ST instructions. See in this context: **Modifiers and Operators in IL**.

Jump

Depending on the selected position in FBD or LD a jump will be inserted directly in front of the selected input, directly after the selected output or at the end of the network. Alternatively drag the jump element from the *Toolbox* or copy or move it by drag&drop within the editor.

After insertion the automatically entered “???” can be replaced by the label to which the jump should be assigned.

In IL a jump is inserted via an JMP instruction.

Label

Each FBD / LD or IL network below the network comment field has a text input field for defining a label (see command *Insert label*). The label is an optional identifier for the network and can be addressed when defining a jump. It can consist of any sequence of characters.

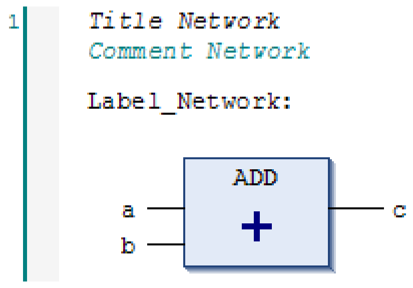


Figure 5-66. Position of the Label in a Network

See the *Options* dialog, tab *FBD, LD, IL Editor* for defining the display of comment and title.

Boxes in FBD/LD/IL

A box, insertable in a FBD, LD or IL network, is a complex element and can represent additional functions like e.g. Timers, Counters, arithmetic operations or also programs, IEC functions and IEC function blocks.

A box can have any desired inputs and outputs and can be provided by a system library or be programmed by the user. At least one input and one output however must be assigned to Boolean values.

If provided with the respective module and if option *Show box* icon is activated, an icon will be displayed within the box.

Use in FBD, LD

A box can be positioned in the left part of a LD network (like a contact) resp. in a FBD network by using command *Insert Box*, *Insert Empty Box*. Alternatively it can be inserted from the *Toolbox* or copied or moved within the editor via drag&drop. Please see **Insert Box** in the MasterTool IEC XE User Manual - MU299609.

Use in IL

In an IL program a CAL instruction with parameters will be inserted in order to represent a box element.

An update of the box parameters (inputs, outputs) - in case the box interface has been changed - in the current implementation can be done without having to re-insert the box by the *Update Parameters* command.

RETURN Instruction in FBD/LD/IL

With a RETURN instruction the FBD, LD or IL POU can be left.

In a FBD or LD network it can be placed in parallel or at the end of the previous elements. If the input of a RETURN is TRUE, the processing of the POU immediately will be aborted.

For inserting use command *Insert Return*. Alternatively drag the element from the *Toolbox* or copy or move it from another position within the editor.

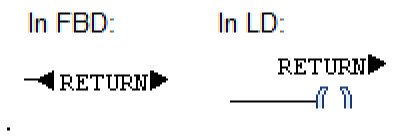


Figure 5-67. RETURN Element

In IL the RET instruction is used.

Branch / Hanging Coil in FBD/LD/IL

FBD, LD

In a FBD/ LD network a branch and a hanging coil splits up the processing line as from the current cursor position. The processing line will continue in two "sub networks" which will be executed one after each other from up to down. Each sub network can get a further branch, such allowing multiple branching within a network.

Each sub network gets an own "marker" (an upstanding rectangle symbol) which can be selected (cursor position 11) in order to perform actions on this arm of the branch.

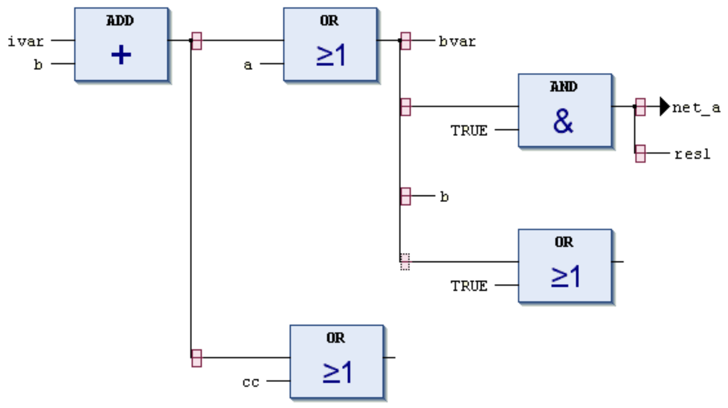


Figure 5-68. Branch and Markers

In FBD a branch gets inserted via command *Insert Branch*. Alternatively drag the element from the *Toolbox*. To verify the possible insert positions, see **Insert Branch** in the MasterTool IEC XE User Manual - MU299609.

NOTE: *Cut* and *Paste* is not possible for sub networks.

See the example shown in the Figure 5-69: A branch has been inserted at the SUB box output. This created two sub networks, each selectable by its subnet marker. After that an ADD box was added in each sub network.

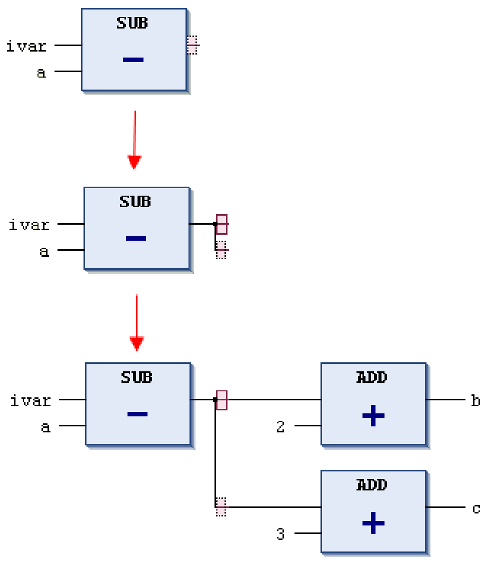


Figure 5-69. Example in FBD, Inserting a Branch

To delete a sub network, first remove all elements of the sub network, that is all elements which are positioned to the right of the sub network's marker, then select the marker and use the standard *Delete* command (<DEL>). See in the Figure 5-70: The 3-input-OR element must be deleted before you can select and delete the marker of the lower sub network.

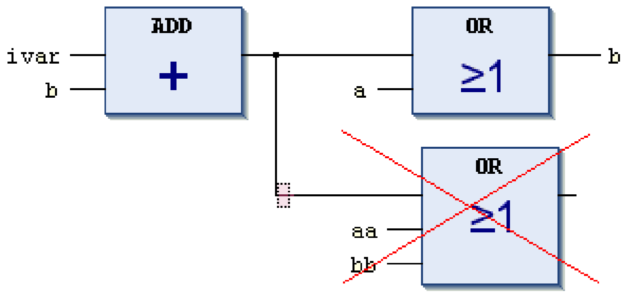


Figure 5-70. Delete Branch and Sub network

Execution in online mode:

The particular branches will be executed from left to right and then from up to down.

IL (Instruction List)

In IL a "branch" resp. "hanging coil" is represented by an appropriate order of instructions. See in this context: **Modifiers and Operators in IL**.

Contact

This is a LD element.

Each network in LD in its left part contains one or several contacts. Contacts are represented by two parallel lines:

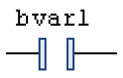


Figure 5-71. Contact

Contacts pass on condition ON (TRUE) or OFF (FALSE) from left to right.

A Boolean variable is assigned to each contact. If this variable is TRUE, the condition is passed from left to right and finally to a coil in the right part of the network, otherwise the right connection receives the value FALSE.

Multiple contacts can be connected in series as well as in parallel. In case of two parallel contacts only one of them must transmit the value TRUE in order to get the parallel branch transmit the value TRUE. In case of contacts connected in series all contacts must transmit the condition TRUE in order to get the last contact transmit the TRUE condition.

So the contact arrangement corresponds to either an electric parallel or a series circuit.

A contact can also be negated, recognizable by the slash in the contact symbol:

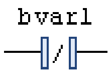


Figure 5-72. Negated Contact

A negated contact passes on the incoming condition (TRUE or FALSE) only if the assigned Boolean variable is FALSE. Notice that the Toolbox directly provides negated contact elements.

A contact can be inserted in a LD network via one of the commands *Insert Contact* or *Insert Contact (right), Insert Contact Parallel (above)* or *Insert Contact Parallel (below)* which by default are part of the *FBD/LD/IL* menu. Alternatively the element can be inserted via drag&drop from the *Toolbox* or from another position within the editor.

FBD, IL

If you are currently working in FBD or IL view, the command will not be available, but contacts and coils inserted in a LD network will be represented by corresponding FBD elements and IL instructions.

Coil

This is a LD element.

On the right side of a LD network there can be any number of so-called coils which are represented by parentheses:

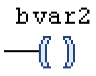


Figure 5-73. Coil

They can only be arranged in parallel. A coil transmits the value of the connections from left to right and copies it to an appropriate Boolean variable. At the entry line the value ON (TRUE) or the value OFF (FALSE) can be present. Contacts and coils can also be negated, recognizable by the slash in the coil symbol:

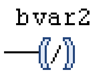


Figure 5-74. Negated Coil

In this case the negated value of the incoming signal will be copied to the appropriate boolean variable, thus: a negated contact only will connect through if the appropriate boolean value is FALSE.

A coil can be inserted in a network via command *Insert Assignment* which per default is part of the *FBD/LD/IL* menu. Alternatively the element can be inserted via drag&drop from the *Toolbox* (Ladder elements) or from another position within the editor. See also: **Set/Reset Coils**.

FBD, IL

If you are currently working in FBD or IL view, the command will not be available, but contacts and coils inserted in a LD network will be represented by corresponding FBD elements resp. IL instructions.

Set/Reset in FBD/LD/IL

FBD and LD

A boolean output in FBD or correspondingly a LD coil can be Set or Reset. To change between the set states use the respective command Set/Reset from the context menu when the output is selected. The output coil will be marked by a S or a R.

Set: If value TRUE arrives at a set output resp. coil, this output/coil will get TRUE and keep TRUE. This value cannot be overwritten at this position as long as the application is running.

Reset: If value TRUE arrives at a reset output resp. coil, this output/coil will get FALSE and keep FALSE. This value cannot be overwritten at this position as long as the application is running.

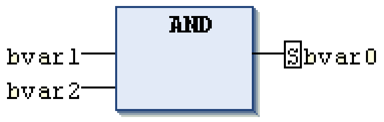


Figure 5-75. Set Output in FBD

See also: **Set/Reset Coils**.

IL

In IL, an instruction list the S and R operators are used to set or reset an operand.

Set/Reset Coils

Coils in the coil symbol: S can also be defined as set or reset coils. One can recognize a set coil by the (S). A set coil will never overwrite the value TRUE in the appropriate boolean variable. That is, the variable once set to TRUE remains TRUE.

One can recognize a reset coil by the R in the coil symbol: (R). A reset coil will never overwrite the value FALSE in the appropriate boolean variable: the variable once set to FALSE will remain FALSE.

In the LD editor Set coils and Reset coils can directly be inserted via drag&drop from the *Toolbox*, category *Ladder elements.*

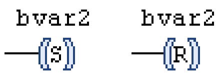


Figure 5-76. Set and Reset Coils

FBD/LD/IL Editors in Online Mode

In online mode the FBD/LD/IL Editor provides views for monitoring and for writing and forcing the variables and expressions on the controller. Debugging functionality (breakpoints, stepping etc.) is available. See below.

For information on how to open objects in online mode see **User Interface in Online Mode** in the MasterTool IEC XE User Manual - MU299609.

Notice that the editor window of an FBD, LD or IL object also includes the *Declaration Editor* in the upper part. See also in this context: **Declaration Editor in Online Mode** in the MasterTool IEC XE User Manual - MU299609.

Monitoring

If the inline monitoring is not explicitly deactivated in the *Options* dialog, it will be supplemented in FBD or LD editor by small monitoring windows behind each variable resp. by an additional monitoring column showing the actual values (inline monitoring). This is even the case for not assigned function block inputs and outputs.

The inline monitoring window of a variable shows a little red triangle in the upper left corner, if the variable currently is forced, a blue one in the lower left corner, if the variable currently is prepared for writing or forcing. On Figure 5-77 an example for a variable which is currently forced and prepared for releasing the force:

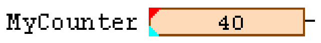
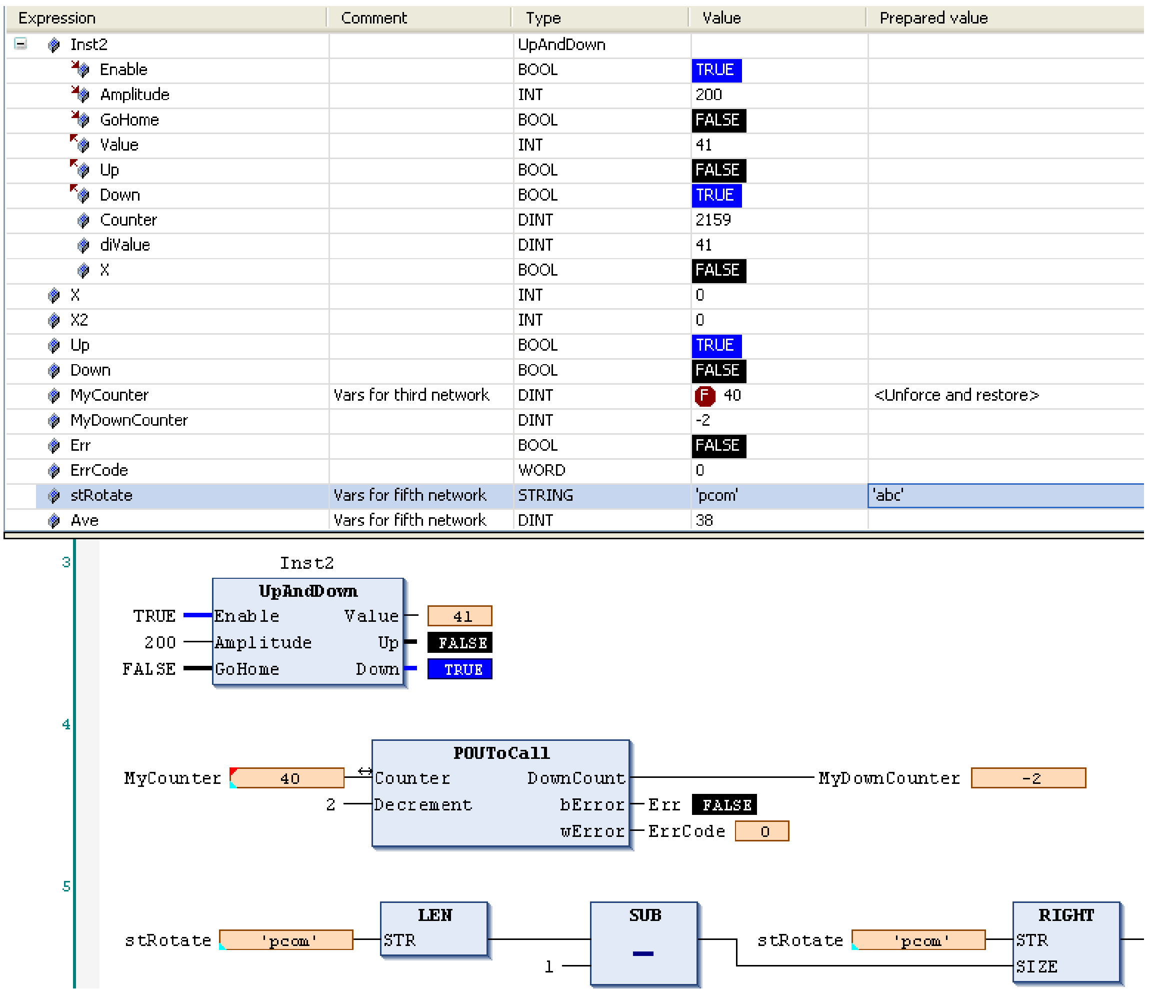
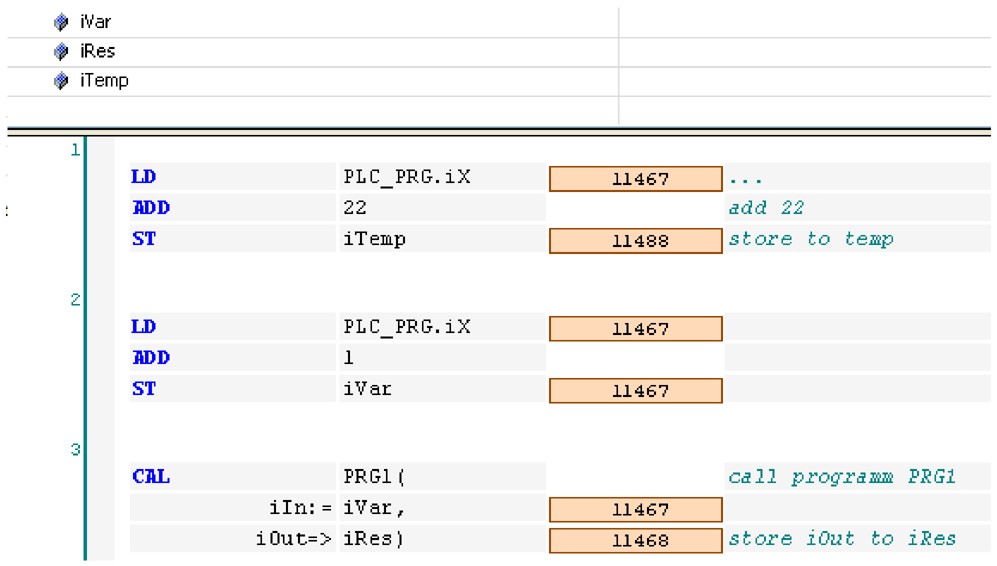


Figure 5-77. Variable Which is Forced and Prepared for Releasing the Force



Expression	Comment	Type	Value	Prepared value
Inst2		UpAndDown		
Enable		BOOL	TRUE	
Amplitude		INT	200	
GoHome		BOOL	FALSE	
Value		INT	41	
Up		BOOL	FALSE	
Down		BOOL	TRUE	
Counter		DINT	2159	
diValue		DINT	41	
X		BOOL	FALSE	
X		INT	0	
X2		INT	0	
Up		BOOL	TRUE	
Down		BOOL	FALSE	
MyCounter	Vars for third network	DINT	F 40	<Unforce and restore>
MyDownCounter		DINT	-2	
Err		BOOL	FALSE	
ErrCode		WORD	0	
stRotate	Vars for fifth network	STRING	'pcom'	'abc'
Ave	Vars for fifth network	DINT	38	

Figure 5-78. Online View of a FBD Program

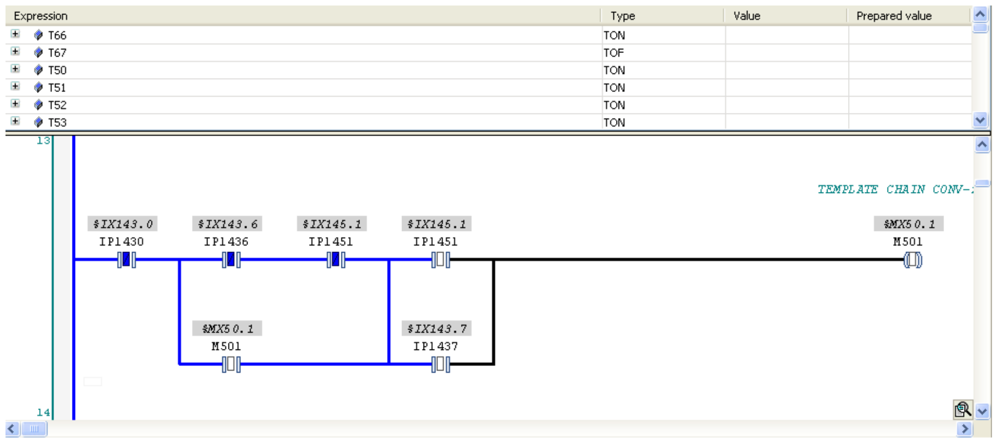


```
iVar
iRes
iTemp

1
LD      PLC_PRG.iX     11467   ...
ADD     22                     add 22
ST      iTemp          11488   store to temp

2
LD      PLC_PRG.iX     11467
ADD     1
ST      iVar           11467

3
CAL     PRG1(                  call programm PRG1
  iIn:= iVar,          11467
 iOut=> iRes)          11468   store iOut to iRes
```

Figure 5-79. Online View of an IL Program

In online view ladder networks have animated connections: Connections with value TRUE are displayed in bold blue, connections with value FALSE in bold black, whereas connections with no known value or with an analog value are displayed in standard outline (black and not fat).

NOTE: The values of the connections are calculated from the monitoring values. It is no real power flow.



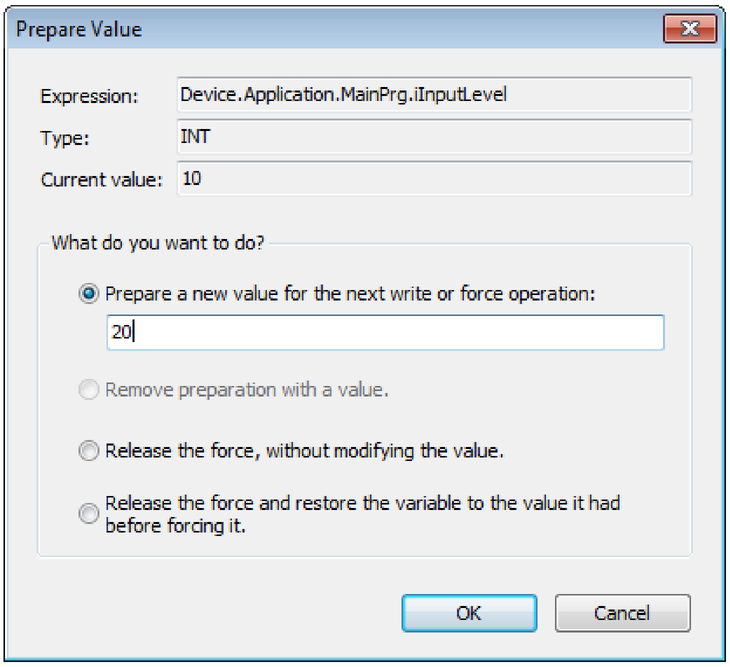
Expression	Type	Value	Prepared value
T66	TON		
T67	TOF		
T50	TON		
T51	TON		
T52	TON		
T53	TON		

Figure 5-80. Online View of a LD Program

Notice for the online view of a POU of type function block: In the implementation part no values will be viewed in the monitoring windows but <*Value of the expression*> and the inline monitoring fields in the implementation part will show three question marks each.

Forcing/Writing of variables

In online mode you can prepare a value for forcing or writing a variable either in the declaration editor or within the implementation part. In the implementation part at a mouse click on the variable the following dialog will open:



Prepare Value

Expression: Device.Application.MainPrg.iInputLevel

Type: INT

Current value: 10

What do you want to do?

[● Prepare a new value for the next write or force operation:] 20

○ Remove preparation with a value.

○ Release the force, without modifying the value.

○ Release the force and restore the variable to the value it had before forcing it.

Figure 5-81. Dialog - Prepare Value

You find the name of the variable completed by its path within the *Device* tree (*Expression*), its *Type* and *Current value*. By activating the corresponding item you may choose whether you want to prepare:

- *Prepare a new value for the next write or force operation:*

- *Remove preparation with a value.*
- *Release the force, without modifying the value.*
- *Release the force and restore the variable to the value it had before forcing it.*

The selected action will be carried out on executing the menu command *Force Values* (per default in the *Online* menu) or pressing <F7>.

Breakpoint and Halt Positions

Possible positions which can be defined for a breakpoint (halt position) for debugging purposes, basically are those positions at which values of variables can change (statements), at which the program flow branches out, or at which another POU is called. That is the following positions:

- On the network at a whole which effects that the breakpoint will be applied to the first possible position within the network
- On a box if this contains a statement; so not possible on operator boxes like for example ADD, DIV. Regard however the note inserted below
- On an assignment
- At the end of a POU at the point of return to the caller; in online mode automatically an empty network will be displayed for this purpose, which instead of a network number is identified by RET

NOTE: Currently you cannot set a breakpoint directly on the first box of a network. If however a breakpoint is set on the complete network, the halt position will automatically be applied to the first box.

You might have a look at the selection list within the breakpoint dialog for all currently possible positions.

A network containing any active breakpoint position is marked with the breakpoint symbol (red filled circle) right to the network number and a red-shaded rectangle background for the first possible breakpoint position within the network. Deactivated breakpoint positions are indicated by a non-filled red circle resp. a surrounding non-filled red rectangle (Figure 5-83).

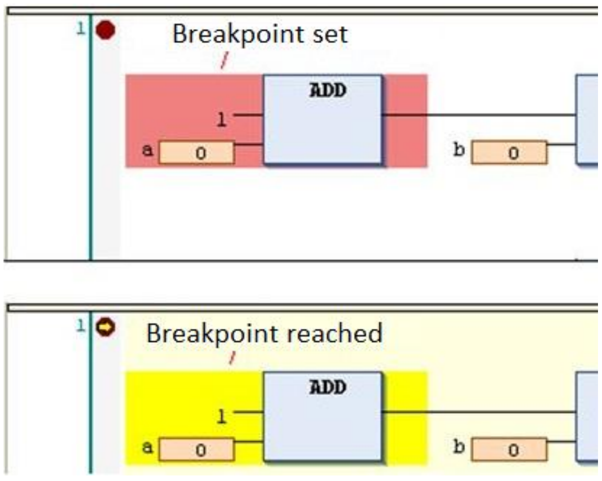


Figure 5-82. Breakpoint (Set and Reached)

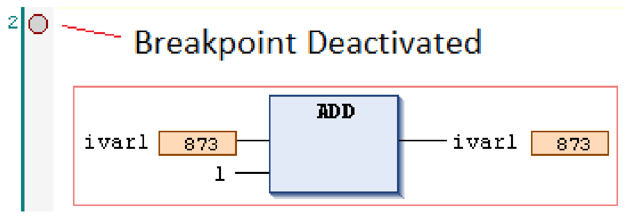


Figure 5-83. Breakpoint Deactivated

As soon as a breakpoint position is reached during stepping or program processing, a yellow arrow will be displayed in the breakpoint symbol and the red shaded area will change to yellow.

The currently reached halt position is indicated by a yellow shadow and the subsequent, not yet reached one by a red shadow:

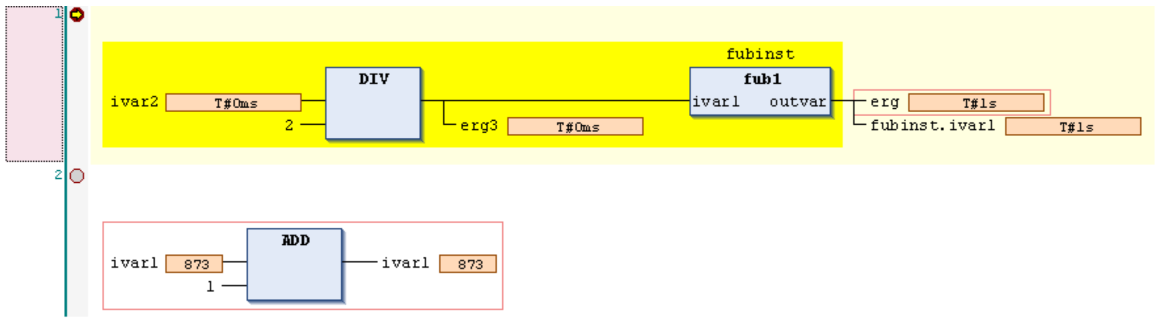
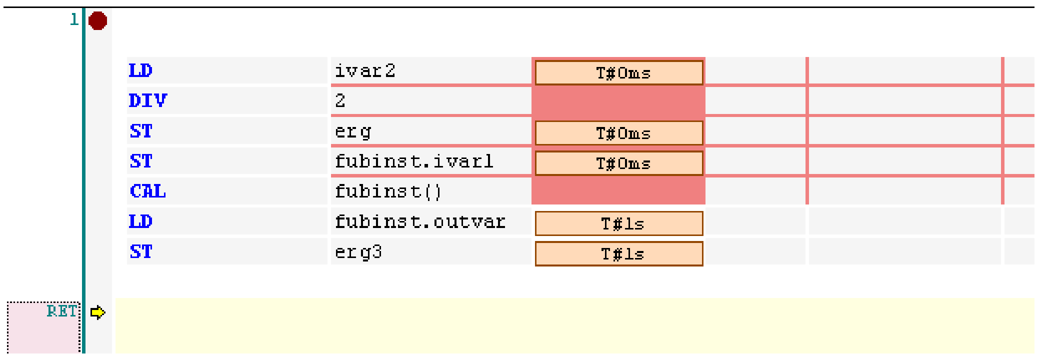


Figure 5-84. Halt Positions Shown in FBD



```
LD    ivar2            T#0ms
DIV   2
ST    erg              T#0ms
ST    fubinst.ivar1    T#0ms
CAL   fubinst()
LD    fubinst.outvar   T#1s
ST    erg3             T#1s
```

Figure 5-85. Halt Positions Shown in IL

NOTE: Notice for breakpoints in methods: A breakpoint will be set automatically in all methods which might be called. If a method is called via a pointer on a function block, breakpoints will be set in the method of the function block and in all derivative function blocks which are subscribing the method.

6. Libraries

The library standard.library is installed by default. It contains all functions and function blocks which are required according to the IEC61131-3 standard as default POUs for an IEC programming system.

Several further libraries are needed for the various functionalities of the programming system, like for example visualization, profiling etc. Those are "implicitly" used libraries, by default will be included automatically in a project and the user has not to deal with them explicitly.

See also in the MasterTool IEC XE User Manual – MU299609, **Library Installation** and **Library Manager Editor**.

The Standard.library Library

The library standard.library is by default provided with the MasterTool IEC XE programming system.

It contains all functions and function blocks which are required matching IEC 61131-3 as standard POUs for an IEC programming system. The difference between a standard function and an operator is that the operator is implicitly recognized by the programming system, while the standard POUs must be tied to the project (standard.library).

String Functions

LEN

Provided by standard.library.

Function of type STRING, returns the length of a string.

Input: STR : STRING, string to be analyzed.

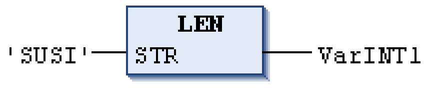
Return value: INT, length of string (number of characters).

Example in IL:

```
LD      'SUSI'
LEN
ST      VarINT1
```

The result is “4”.

Example in FBD:



Example in ST:

```
VarINT1 := LEN ('SUSI');
```

LEFT

Provided by standard.library.

Function of type STRING. Returns the left, initial string for a given string.

Inputs:

STR : STRING; string to be analyzed.

SIZE : INT; length of left initial string (number of characters).

Return value: STRING; initial string.

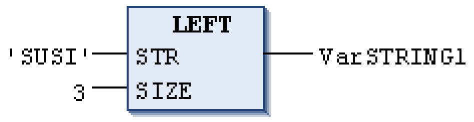
LEFT (STR, SIZE) means: Return the first SIZE characters from the right in the string STR.

Example in IL:

LD	'SUSI'
LEFT	3
ST	VarSTRING1

The result is “SUS”.

Example in FBD:



Example in ST:

```
VarSTRING1 := LEFT ('SUSI',3);
```

RIGHT

Provided by standard.library.

Function of type STRING. Returns the right, initial string for a given string.

Inputs:

STR: STRING; string to be analyzed.

SIZE: INT; number of characters to be counted from the right in string STR.

Return value:

STRING; initial right string.

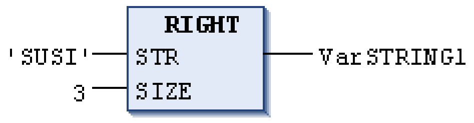
RIGHT (STR, SIZE) means: Return the first SIZE character from the right in the string STR.

Example in IL:

LD	'SUSI'
RIGHT	3
ST	VarSTRING1

The result is “USI”.

Example in FBD:



Example in ST:

```
VarSTRING1 := RIGHT ('SUSI',3);
```

MID

Provided by standard.library.

Function of type STRING, returns a partial string from within a string.

Inputs:

STR: STRING; string to be analyzed.

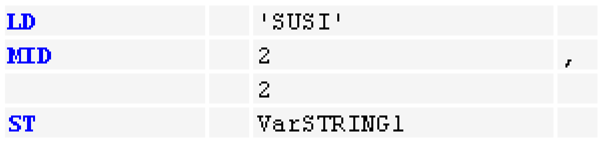
LEN: INT; length of the partial string (number of characters).

POS : INT; start position for the partial string, number of characters counted from the left of STR.

Return value: STRING, partial string.

MID (STR, LEN, POS) means: Retrieve LEN characters from the STR string beginning with the character at position POS.

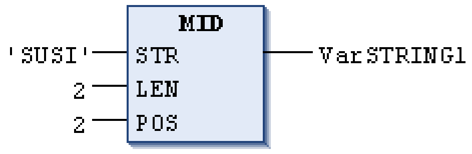
Example in IL:



LD		'SUSI'	
MID		2	,
		2	
ST		VarSTRING1	

The result is “US”.

Example in FBD:



Example in ST:

```
VarSTRING1 := MID ('SUSI',2,2);
```

CONCAT

Provided by standard.library.

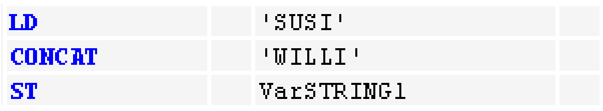
Function of type STRING, doing a concatenation (combination) of two strings.

Inputs: STR1, STR2: STRING; strings to be concatenated.

Return value: STRING, concatenated string.

CONCAT(STR1,STR2) means to connect STR1 and STR2 to a single string STR1STR2.

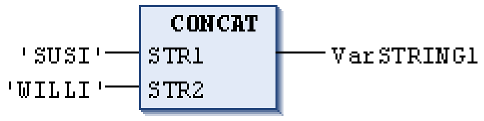
Example in IL:



LD		'SUSI'	
CONCAT		'WILLI'	
ST		VarSTRING1	

The result is “SUSIWILLI”.

Example in FBD:



Example in ST:

```
VarSTRING1 := CONCAT ('SUSI','WILLI');
```

INSERT

Provided by standard.library.

Function of type STRING, inserts a string into another string at a defined point.

Inputs:

STR1: STRING; string into which STR2 has to be inserted.

STR2 : STRING; string which has to be inserted into STR1.

POS : INT; Position in STR1 where STR2 has to be inserted, number of characters counted from left.

Return value: STRING, resulting string.

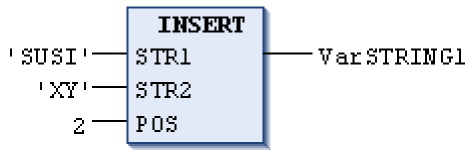
INSERT(STR1, STR2, POS) means: Insert STR2 into STR1 after position POS.

Example in IL:

LD		'SUSI'	
INSERT		'XY'	,
		2	
ST		VarSTRING1	

The result is “SUXYSI”.

Example in FBD:



Example in ST:

```
VarSTRING1 := INSERT ('SUSI','XY',2);
```

DELETE

Provided by standard.library.

Function of type STRING, removes a partial string from a larger string at a defined position.

Inputs:

STR: STRING; string from which a part should be deleted.

LEN: INT; length of the partial string to be deleted (number of characters).

POS: INT; position in STR where the deletion of LEN characters should start, counted from left.

Return value: STRING, string remaining after deletion.

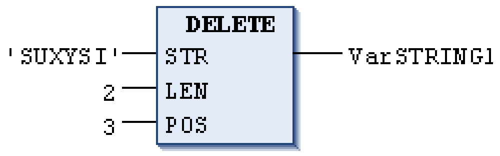
DELETE(STR, L, P) means: Delete L characters from STR, beginning with the character in the POS position.

Example in IL:

LD		'SUXYSI'	
DELETE		2	,
		3	
ST		VarSTRING1	

The result is “SUSI”.

Example in FBD:



Example in ST:

```
Var1 := DELETE ('SUXYSI',2,3);
```

REPLACE

Provided by standard.library.

Function of type STRING, replaces a partial string from a larger string with another string.

Inputs:

STR1: STRING, string of which a part should be replaced by string STR2.

STR2: STRING, string which should replace a part of STR1.

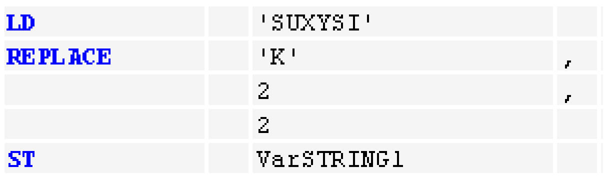
L: INT, length of partial string in STR1 which should be replaced.

P : INT, position where STR2 should be inserted instead of the existing L characters.

Return value: STRING, resulting string.

REPLACE(STR1, STR2, L, P) means: Replace L characters from STR1 by STR2, beginning with the character in the P position.

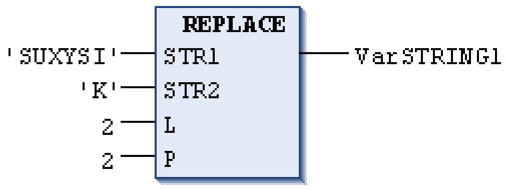
Example in IL:



LD	'SUXYSI'	
REPLACE	'K'	,
	2	,
	2	
ST	VarSTRING1	

Result is “SKYSI”.

Example in FBD:



Example in ST:

```
VarSTRING1 := REPLACE ('SUXYSI','K',2,2);
```

FIND

Provided by standard.library.

Function of type INT, searches for the position of a partial string within a string.

Inputs:

STR1: STRING, string within which STR2 should be searched.

STR2: STRING, string whose position should be searched in STR1.

Return value: INT, start position of STR2 in STR1. "0" if STR2 is not found in STR1.

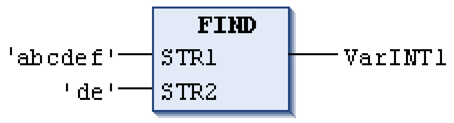
FIND(STR1, STR2) means: Find the position of the first character where STR2 appears in STR1 for the first time. If STR2 is not found in STR1, then OUT:=0.

Example in IL:

```
LD      'abcdef'
FIND    'de'
ST      VarSTRING1
```

The result is “4”.

Example in FBD:



Example in ST:

```
arINT1 := FIND ('abcdef','de');
```

Bistable Function Blocks

SR

Provided by standard.library.

Function block, making bistable function blocks dominant.

Inputs:

SET1 : BOOL;

RESET : BOOL;

Outputs:

Q1 : BOOL;

Q1 = SR (SET1, RESET):

Q1 = (NOT RESET AND Q1) OR SET1

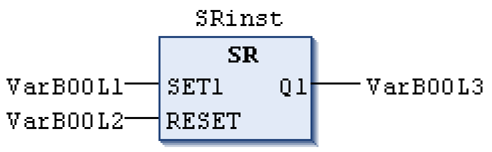
Declaration example:

```
SRInst : SR ;
```

Example in IL:

```
CAL              SRinst       (
         SET1:=  VarBool1     ,
        RESET:=  VarBool2     )
LD               SRinst.Q1
ST               VarBool3
```

Example in FBD:



Example in ST:

```
SRInst(SET1:= VarBOOL1 , RESET:=VarBOOL2 );
VarBOOL3 := SRInst.Q1;
```

RS

Provided by standard.library.

Function block, resetting bistable function blocks.

Inputs:

SET: BOOL;

RESET1: BOOL;

Outputs:

Q1: BOOL;

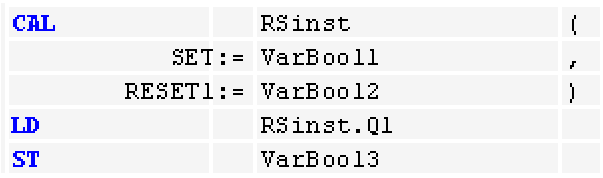
Q1 = RS (SET, RESET1):

Q1 = NOT RESET1 AND (Q1 OR SET)

Declaration example:

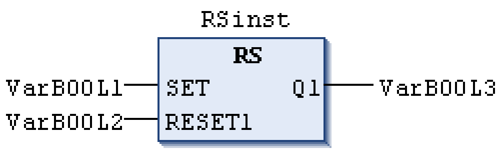
```
RSInst : RS ;
```

Example in IL:



```
CAL                RSinst        (
            SET:=  VarBool1      ,
         RESET1:=  VarBool2      )
LD                 RSinst.Q1
ST                 VarBool3
```

Example in FBD:



Example in ST:

```
RSInst(SET:= VarBOOL1 , RESET1:=VarBOOL2 );
VarBOOL3 := RSInst.Q1;
```

Trigger

R_TRIG

Provided by standard.library.

Function block detecting a rising edge.

Inputs:

CLK: BOOL; incoming Boolean signal to be checked for rising edge.

Outputs:

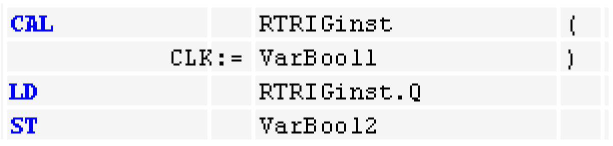
Q: BOOL; becomes TRUE if a rising edge occurs at CLK.

The output Q and an internal Boolean help variable M will remain FALSE as long as the input variable CLK is FALSE. As soon as CLK returns TRUE, Q will first return TRUE, then M will be set to TRUE. This means each time the function is called up, Q first will be set TRUE, then return FALSE, followed by a rising edge in CLK.

Declaration example:

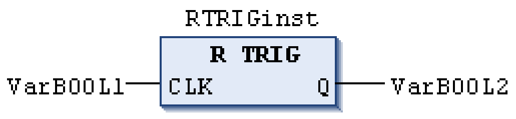
```
RTRIGInst : R_TRIG;
```

Example in IL:



CAL		RTRIGinst	(
	CLK:=	VarBool1	)
LD		RTRIGinst.Q	
ST		VarBool2	

Example in FBD:



Example in ST:

```
RTRIGInst(CLK:= VarBOOL1);
VarBOOL2 := RTRIGInst.Q;
```

F_TRIG

Provided by standard.library.

Function block detecting a falling edge.

Inputs: CLK: BOOL; incoming Boolean signal to be checked for rising edge.

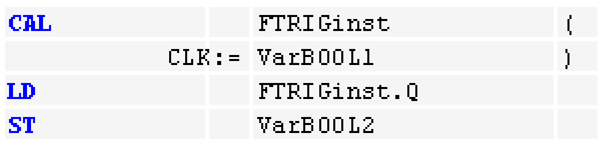
Outputs: Q: BOOL; becomes TRUE if a falling edge occurs at CLK.

The output Q and an internal Boolean help variable M will remain FALSE as long as the input variable CLK returns TRUE. As soon as CLK returns FALSE, Q will first return TRUE, then M will be set to TRUE. This means each time the function is called up, Q first will be set TRUE, then return FALSE, followed by a falling edge in CLK.

Declaration example:

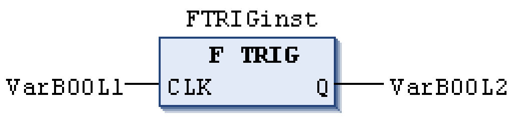
```
FTRIGInst : F_TRIG ;
```

Example in IL:



CAL		FTRIGinst	(
	CLK:=	VarBOOL1	)
LD		FTRIGinst.Q	
ST		VarBOOL2	

Example in FBD:



Example in ST:

```
FTRIGInst(CLK:= VarBOOL1);
VarBOOL2 := FTRIGInst.Q;
```

Counter

CTU

Provided by standard.library.

Function block working as an incrementer.

Inputs:

CU: BOOL; a rising edge at this input starts the incrementing of CV.

RESET: BOOL; If TRUE, CV will be reset to 0.

PV: WORD; upper limit for the incrementing of CV.

NOTE: Datatype WORD, which is used for PV in MasterTool IEC XE, does not match the IEC standard, which for PV defines datatype INT.

Outputs:

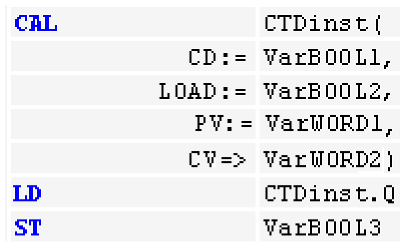
Q: BOOL; gets TRUE as soon as CV has reached the limit given by PV.

CV: WORD; gets counted up until it reaches PV.

Declaration example:

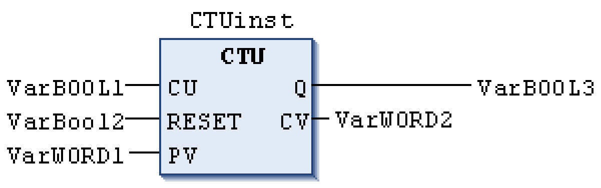
```
CTUInst: CTU;
```

Example in IL:



```
CAL          CTDinst(
        CD:= VarBOOL1,
      LOAD:= VarBOOL2,
        PV:= VarWORD1,
        CV=> VarWORD2)
LD           CTDinst.Q
ST           VarBOOL3
```

Example in FBD:



Example in ST:

```
CTUInst(CU := VarBOOL1, RESET := VarBOOL2 , PV := VarWORD1);
VarBOOL3 := CTUInst.Q;
VarWORD2 := CTUInst.CV;
```

CTD

Provided by standard.library.

Function block working as a decrementer.

Inputs:

CD : BOOL; a rising edge at this input starts the decrementing of CV.

LOAD : BOOL; If TRUE, CV will be reset to the upper limit given by PV.

PV : WORD; upper limit, that is start value for decrementing of CV.

NOTE: Datatype WORD, which is used for PV in MasterTool IEC XE, does not match the IEC standard, which for PV defines datatype INT.

Outputs:

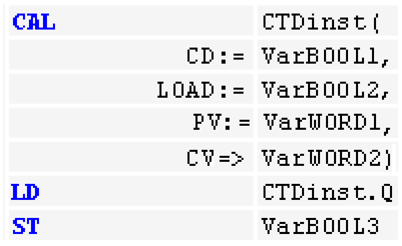
Q: BOOL; gets TRUE as soon as CV is 0.

CV: WORD; gets decremented by 1, starting with PV until 0 is reached.

Declaration example:

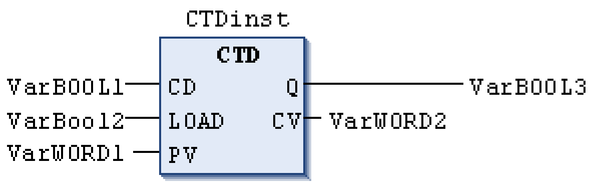
```
CTDInst : CTD ;
```

Example in IL:



```
CAL          CTDinst(
        CD:= VarBOOL1,
      LOAD:= VarBOOL2,
        PV:= VarWORD1,
        CV=> VarWORD2)
LD           CTDinst.Q
ST           VarBOOL3
```

Example in FBD:



Example in ST:

```
CTDInst(CD := VarBOOL1, LOAD := VarBOOL2 , PV := VarWORD1);
VarBOOL3 := CTDInst.Q;
VarWORD2 := CTDInst.CV;
```

CTUD

Provided by standard.library.

Function block working as incrementer and decrementer.

Inputs:

CU: BOOL; if a rising edge occurs at CU, incrementing of CV will be started.

CD: BOOL; if a rising edge occurs at CU, decrementing of CV will be started.

RESET: BOOL; If TRUE, CV will be set to 0.

LOAD : BOOL; if TRUE, CV will be set to PV.

PV : WORD; upper limit for incrementing or decrementing CV.

NOTE: Datatype WORD, which is used for PV in MasterTool IEC XE, does not match the IEC standard, which for PV defines datatype INT.

Outputs:

QU: BOOL; returns TRUE when CV has been incremented to >= PV.

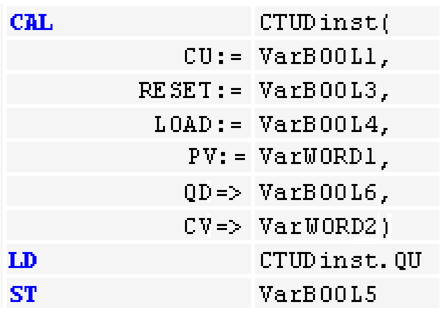
QD: BOOL; returns TRUE when CV has been decremented to 0.

CV: WORD; gets incremented or decremented.

Declaration example:

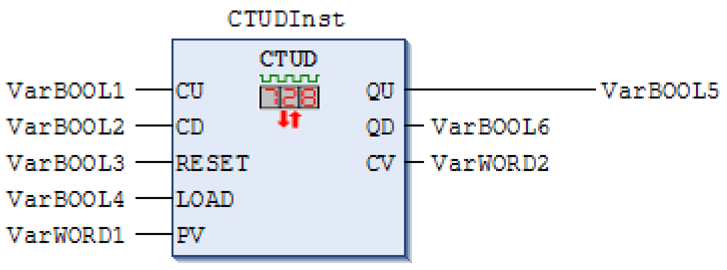
```
CTUDInst : CUTD;
```

Example in IL:



```
CAL        CTUDinst(
     CU:=  VarBOOL1,
  RESET:=  VarBOOL3,
   LOAD:=  VarBOOL4,
     PV:=  VarWORD1,
     QD=>  VarBOOL6,
     CV=>  VarWORD2)
LD         CTUDinst.QU
ST         VarBOOL5
```

Example in FBD:



Example in ST:

```
CTUDInst(CU := VarBOOL1, CD := VarBOOL2, RESET := VarBOOL3, LOAD :=
VarBOOL4 , PV := VarWORD1);
VarBOOL5 := CTUDInst.QU;
VarBOOL6 := CTUDInst.QD;
VarINT1 := CTUDInst.CV;
```

Timer

TP

Provided by standard.library.

Timer function block, working as a trigger. A timer is counted up until a given limit is reached. During counting up a "pulse" variable is TRUE, otherwise it is FALSE.

Inputs:

IN: BOOL; At a rising edge counting up the time in ET will be started;

PT: TIME; Upper limit of the time.

Outputs:

Q: BOOL; TRUE as long as the time is being counted up in ET (pulse).

ET: TIME; Current state of the time.

TP(IN, PT, Q, ET):

If IN is FALSE, Q will be FALSE and ET will be 0.

As soon as IN becomes TRUE, the time will begin to be counted in milliseconds in ET until its value is equal to PT. It will then remain constant.

Q is TRUE as from IN has got TRUE and ET is less than or equal to PT. Otherwise it is FALSE.

Q returns a signal for the time period given in PT.

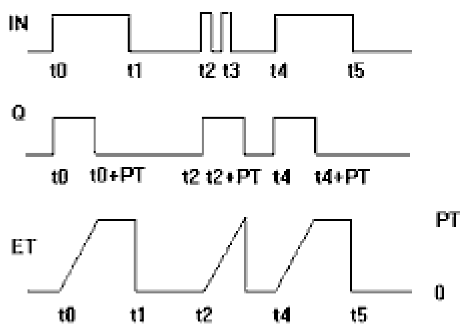
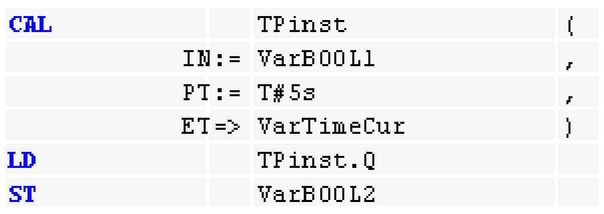


Figure 6-1. Graphic Display of the TP Time Sequence

Declaration example:

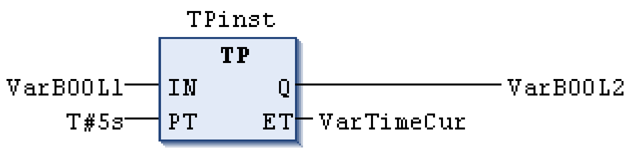
```
TPInst : TP;
```

Example in IL:



```
CAL                TPinst      (
             IN:=  VarBOOL1    ,
             PT:=  T#5s        ,
             ET=>  VarTimeCur  )
LD                 TPinst.Q
ST                 VarBOOL2
```

Example in FBD:



Example in ST:

```
TPInst(IN := VarBOOL1, PT:= T#5s);
VarBOOL2 :=TPInst.Q;
```

TON

Provided by standard.library.

Timer function block, implements a turn-on delay. When the input gets TRUE, first a certain time will run through until also the output gets TRUE.

Inputs:

IN: BOOL; Rising edge starts counting up ET.

PT: TIME; Upper limit for counting up ET (delay time).

Outputs:

Q: BOOL; Gets a rising edge as soon as ET has reached the upper limit PV (delay time is over).

ET: current state of delay time.

TON(IN, PT, Q, ET):

If IN is FALSE, Q will be FALSE and ET will be 0.

As soon as IN becomes TRUE, the time will begin to be counted in milliseconds in ET until its value is equal to PT. It will then remain constant.

Q is TRUE when IN is TRUE and ET is equal to PT. Otherwise it is FALSE.

Thus, Q has a rising edge when the time indicated in PT in milliseconds has run out.

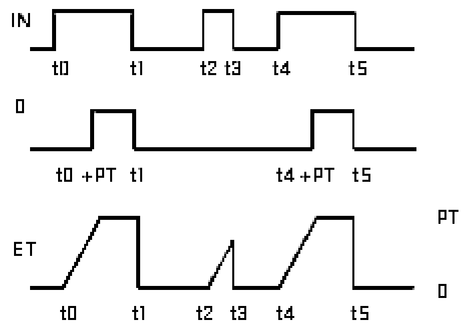
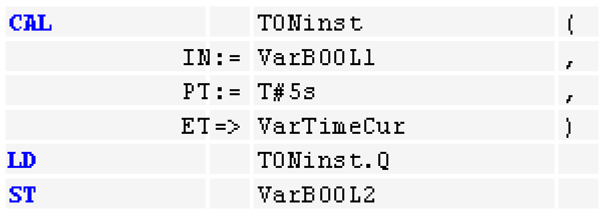


Figure 6-2. Graphic Display of TON Behavior Over Time

Declaration example:

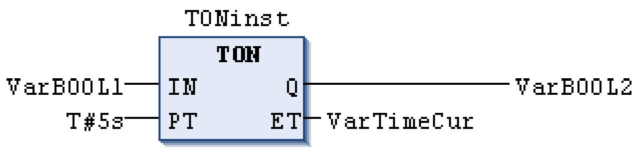
```
TONInst : TON;
```

Example in IL:



```
CAL          TONinst      (
        IN:= VarBOOL1     ,
        PT:= T#5s         ,
        ET=> VarTimeCur   )
LD           TONinst.Q
ST           VarBOOL2
```

Example in FBD:



Example in ST:

```
TONInst(IN := VarBOOL1, PT:= T#5s);
```

TOF

Provided by standard.library.

Timer function block, implements a turn-off delay. When the input changes from TRUE to FALSE (falling edge), first a certain time will run through until also the output gets FALSE.

Inputs:

IN: BOOL; Falling edge starts counting up ET.

PT: TIME; Upper limit for counting up ET (delay time).

Outputs:

Q: BOOL; Gets a falling edge as soon as ET has reached the upper limit PV (delay time is over).

ET: current state of delay time.

TOF(IN, PT, Q, ET):

If IN is TRUE, the outputs will be TRU respectively 0.

As soon as IN becomes FALSE, in ET the time will begin to be counted in milliseconds in ET until its value is equal to PT. It will then remain constant.

Q is FALSE when IN is FALSE und ET equal PT. Otherwise it is TRUE.

Thus, Q has a falling edge when the time indicated in PT in milliseconds has run out.

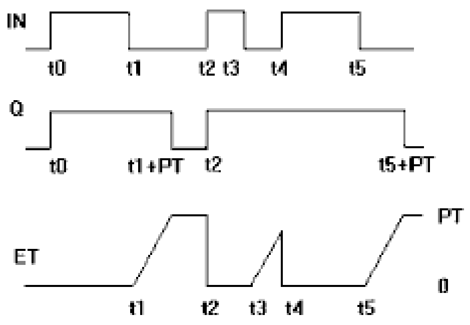
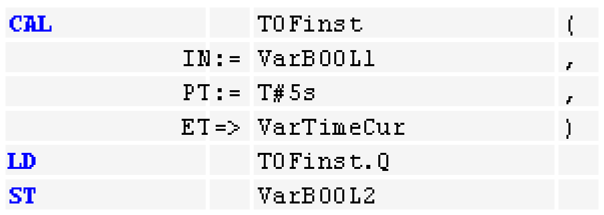


Figure 6-3. Graphic Display of TOF Behavior Over Time

Declaration example:

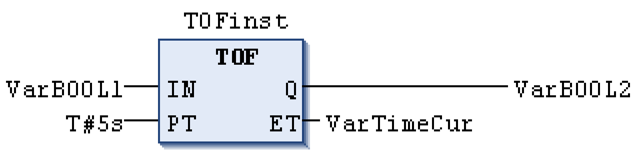
```
TOFInst : TOF;
```

Example in IL:



```
CAL              TOFinst       (
          IN:=   VarBOOL1      ,
          PT:=   T#5s          ,
          ET=>   VarTimeCur    )
LD               TOFinst.Q
ST               VarBOOL2
```

Example in FBD:



Example in ST:

```
TOFInst(IN := VarBOOL1, PT:= T#5s);
VarBOOL2 :=TOFInst.Q;
```

RTC

Provided by standard.library.

Timer function block RunTime Clock, returns, starting at a given time, the current date and time.

Inputs:

EN: BOOL; At a rising edge starts counting up the time in CDT.

PDT: DATE_AND_TIME; Date and time from which the counting up should be started.

Outputs:

Q: BOOL; Is TRUE as long as CDT is counting up.

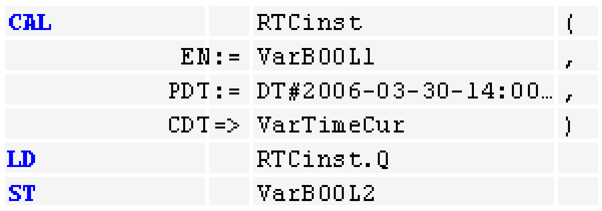
CDT: DATE_AND_TIME; Current state of counted date and time.

VarBOOL2:=RTC(EN, PDT, Q, CDT):

When EN is FALSE, the output variables Q und CDT are FALSE respectively DT#1970-01-01-00:00:00.

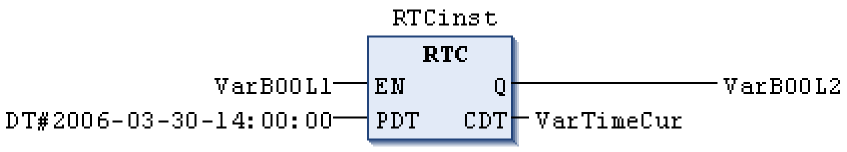
As soon as EN becomes TRUE (rising edge), the time given by PDT is set, is counted up in seconds and returned in CDT as long as EN is TRUE. As soon as EN is reset to FALSE, CDT is reset to the initial value DT#1970-01-01-00:00:00.

Example in IL:



```
CAL           RTCinst              (
         EN:= VarBOOL1             ,
        PDT:= DT#2006-03-30-14:00… ,
        CDT=> VarTimeCur           )
LD            RTCinst.Q
ST            VarBOOL2
```

Example in FBD:



Example in ST:

```
RTC(EN:=VarBOOL1, PDT:=DT#2006-03-30-14:00:00, Q=>VarBOOL2,
CDT=>VarTimeCur);
```

The UTIL.library Library

This library contains an additional collection of various blocks which can be used for BCD conversion, bit/byte functions, mathematical auxiliary functions, as controller, signal generators, function manipulators and for analogue value processing.

BCD Conversion

Provided by util.library.

A byte in the BCD format contains integers between 0 and 99. Four bits are used for each decimal place. The ten decimal place is stored in the bits 4-7. Thus the BCD format is similar to the hexadecimal presentation, with the simple difference that only values between 0 and 99 can be stored in a BCD byte, whereas a hexadecimal byte reaches from 0 to FF.

Example:

The integer “51” should be converted to BCD format. Five in binary is “0101”, one in binary is “0001”, which makes the BCD byte “01010001”, which corresponds to the value $51=81.

BCD_TO_INT

Provided by util.library.

This function converts a byte in BCD format into an INT value.

The input value of the function is type BYTE and the output is type INT.

Where a byte should be converted which is not in the BCD format the output is “-1”.

Examples in ST:

```
i:=BCD_TO_INT(73); (* Result is 49 *)
k:=BCD_TO_INT(151); (* Result is 97 *)
l:=BCD_TO_INT(15); (* Output -1, because it is not in BCD format *)
```

INT_TO_BCD

Provided by util.library.

This function converts an INTEGER value into a byte in BCD format.

The input value of the function is type INT, the output is type BYTE.

The number “255” will be outputted where an INTEGER value should be converted which cannot be converted into a BCD byte.

Examples in ST:

```
i:=INT_TO_BCD(49); (* Result is 73 *)
k:=BCD_TO_INT(97); (* Result is 151 *)
l:=BCD_TO_INT(100); (* Error! Output: 255 *)
```

BIT/BYTE Functions

Extract

Provided by util.library.

Inputs to this function are a DWORD X, as well as a BYTE N. The output is a BOOL value, which contains the content of the Nth bit of the input X, whereby the function begins to count from the zero bit.

Examples in ST:

```
FLAG:=EXTRACT(X:=81, N:=4); (* Result : TRUE, because 81 is binary
1010001, so the 4th bit is 1 *)
FLAG:=EXTRACT(X:=33, N:=0); (* Result : TRUE, because 33 is binary 100001,
so the bit '0' is 1 *)
```

Pack

Provided by util.library.

This function is capable of delivering back eight input bits B0, B1, ..., B7 from type BOOL as a BYTE.

The function block UNPACK is closely related to this function. See **Unpack** for details and examples.

Putbit

Provided by util.library.

The input to this function consists of a DWORD X, a BYTE N and a Boolean value B.

PUTBIT sets the Nth bit from X on the value B, whereby it starts counting from the zero bit.

Example in ST:

```
A:=38; (* Binary 100110 *)
B:=PUTBIT(A,4,TRUE); (* Result: 54 = 2#110110 *)
C:=PUTBIT(A,1,FALSE); (* Result: 36 = 2#100100 *)
```

Unpack

Provided by util.library.

UNPACK converts the input B from type BYTE into 8 output variables B0,...,B7 of the type BOOL, and this is the opposite to PACK.

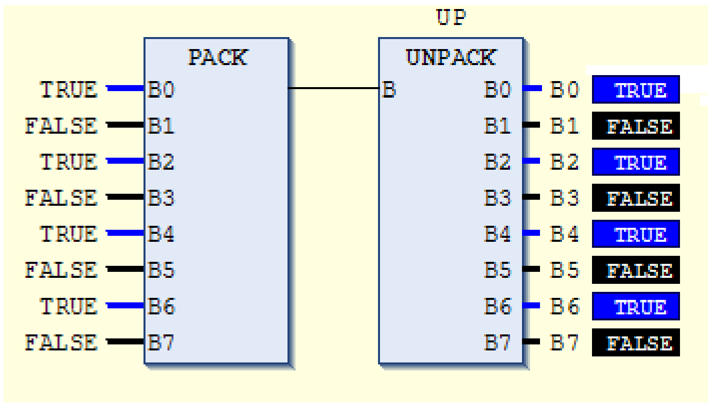


Figure 6-4. Example in FBD, Output

Mathematical Auxiliary Function

Derivative

Provided by util.library.

This function block approximately determines the local derivation. The function value is delivered as a REAL variable by using IN.

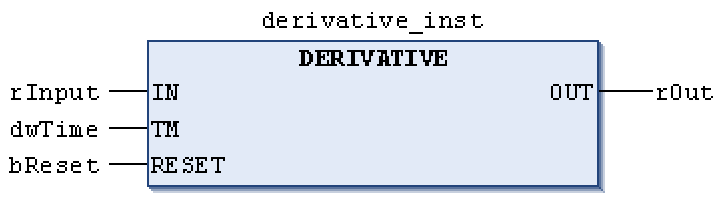
TM contains the time which has passed in msec in a DWORD.

By an input TRUE of RESET the function block can be restarted.

The output OUT is of the type REAL.

In order to obtain the best possible result, DERIVATIVE approximates using the last four values, in order to hold errors which are produced by inaccuracies in the input parameters as low as possible.

Example in FBD:



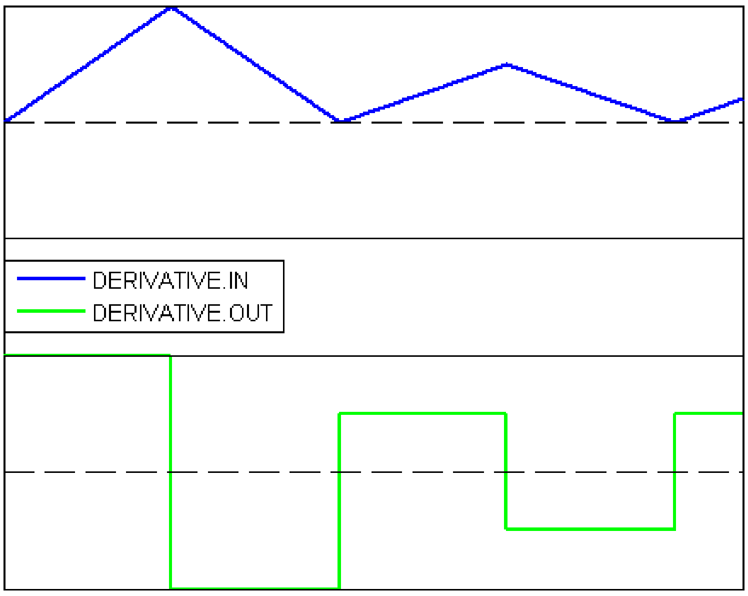


Figure 6-5. FB DERIVATIVE Behavior

Integral

Provided by util.library.

This function block approximately determines the integral of the function.

Analogous to DERIVATIVE the function value is delivered as a REAL variable by using IN.

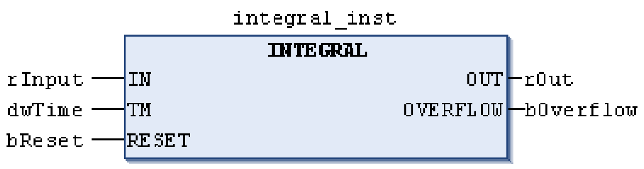
TM contains the time which has passed in msec in a DWORD.

By TRUE in input RESET the function block can be restarted.

Output OUT is of type REAL.

The integral is approximated by two step functions. The average of these is delivered as the approximated integral.

Example in FBD:



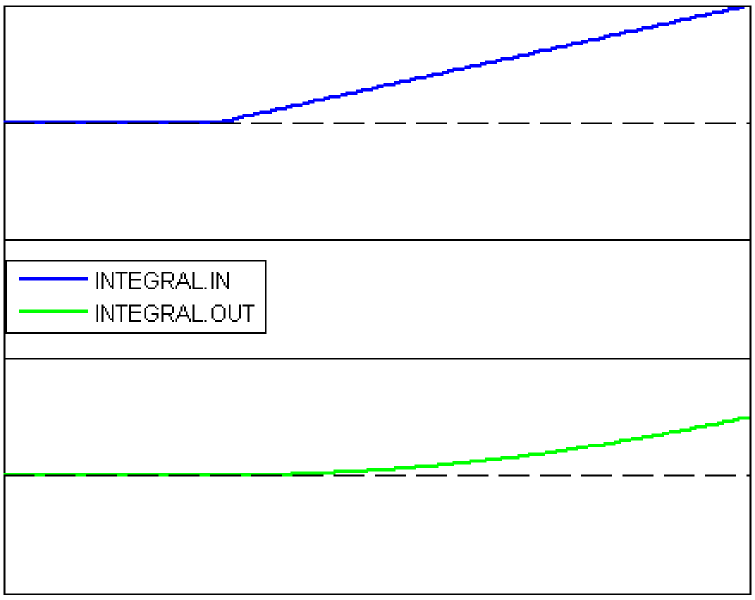
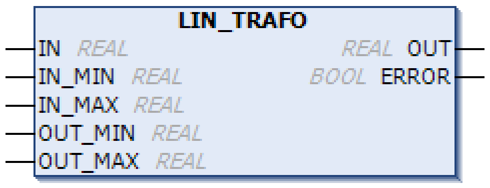


Figure 6-6. FB INTEGRAL Behavior

Lin_TRAFO

This function block (util.library) transforms a REAL-value, which lies in a value range defined by a lower and upper limit value, to a REAL-value which lies correspondingly in another range also defined by a lower and upper limit. The following equation is basis of the conversion: “(IN - IN_MIN) : (IN_MAX - IN) = (OUT - OUT_MIN) : (OUT_MAX - OUT)”.

Example in FBD:



Input Variables

Variable	Data type	Description
IN	REAL	Input value
IN_MIN	REAL	Lower limit of input value range
IN_MAX	REAL	Upper limit of input value range
OUT_MIN	REAL	Lower limit of output value range
OUT_MAX	REAL	Upper limit of output value range

Table 6-1. Input Variables (LIN_TRAFO)

Output Variables

Variable	Data type	Description
OUT	REAL	Output value
ERROR	BOOL	Error occurred: TRUE, if IN_MIN = IN_MAX, or if IN is outside of the specified input value range.

Table 6-2. Output Variables (LIN_TRAFO)

Example:

A temperature sensor delivers information in a 0 to 10 V scale. When this sensor is connected to a NX6000 analog input channel, the minimum value of the scale is 0 and the maximum is 30,000. These values can be edited in the NX6000 Input Parameters tab, where 0 to 30,000 is the standard scale. If the user wishes to convert these values to a Celsius degrees temperature scale, it’s only necessary to change the minimum and maximum value. E.g. in a hypothetical sensor where 0 V is 0 °C and 10 V is 100 °C, the module setup could be 0 to minimum value and 100 to maximum value. In this scenario, the INT type converted variable would have a 1 °C resolution.

It is possible to convert the read sensor INT value to REAL type without accuracy loss. First an INT to REAL conversion takes place using the INT_TO_REAL function. The result of this conversion must then be passed to LIN_TRAFO (through the IN input). The input limites are defined through the block inputs IN_MIN=0 and IN_MAX=30000. The output value range (°C) is defined by OUT_MIN=0 e OUT_MAX=100. The converted output will be in Celcius degrees (OUT output) without accuracy loss since the analog input.

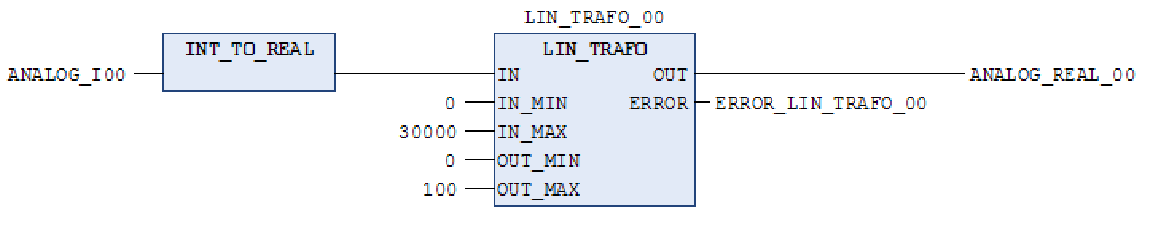


Figure 6-7. FBD Example of LIN_TRAFO block

Figure 6-7 presents an analog voltage input conversion example to a REAL type in a 0 a 100 °C scale.

For instance, a 4.67 Vdc input would result in 14,000, and consequently a 46.67 °C in the REAL type output variable.

STATISTICS-INT

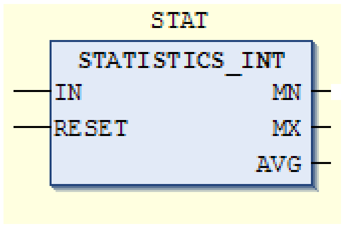
Provided by util.library.

This function block calculates some standard statistical values.

The input IN is of the type INT. All values are initialized anew when the boolean input RESET is TRUE.

The output MN contains the minimum, MX of the maximum value from IN. AVG describes the average, that is the expected value of IN. All three outputs are of the type INT.

Example of STATISTICS-INT in FBD:



STATISTICS_REAL

Provided by util.library.

This function block corresponds to STATISTICS_INT, except that the input IN is of the type REAL like the outputs MN, MX, AVG.

VARIANCE

Provided by util.library.

VARIANCE calculates the variance of the entered values.

The input IN is of the type REAL, RESET is of the type BOOL and the output OUT is again of the type REAL.

This block calculates the variance of the inputted values. VARIANCE can be reset with RESET=TRUE.

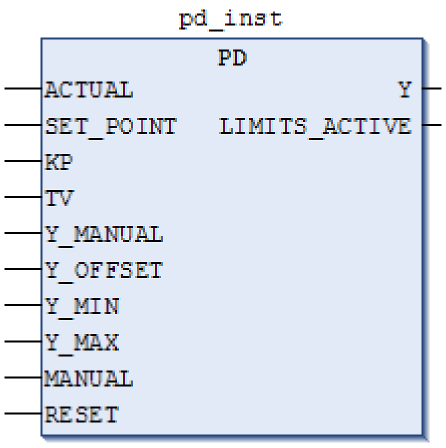
The standard deviation can easily be calculated as the square root of the VARIANCE.

Controllers

PD

The library util.library provides the PD controller function block.

Example of PD in FBD:



Inputs of the Function Block

Variable	Datatype	Description
ACTUAL	REAL	Current value of the controlled variable
SET_POINT	REAL	Desired value, command variable
KP	REAL	Proportionality coefficient, unity gain of the P-part
TV	REAL	Derivative action time, unity gain of the D-part in seconds, for example "0.5" for 500 msec
Y_MANUAL	REAL	Defines output value Y in case of MANUAL = TRUE
Y_OFFSET	REAL	Offset for the manipulated variable Y

Y_MIN, Y_MAX	REAL	Lower/upper limit for the manipulated variable Y. If Y exceeds these limits, output LIMITS_ACTIVE will be set to TRUE and Y will be kept within the prescribed range. This control will only work if Y_MIN<Y_MAX.
MANUAL	BOOL	If TRUE, manual operation will be active, that is the manipulated value will be defined by Y_MANUAL.
RESET	BOOL	TRUE resets the controller. During reinitialization Y = Y_OFFSET

Table 6-3. Input Variables (PD)

Outputs of the Function Block

Variable	Datatype	Description
Y	REAL	Manipulated value, calculated by the function block
LIMITS_ACTIVE	BOOL	TRUE indicates that Y has exceeded the given limits (Y_MIN, Y_MAX).

Table 6-4. Output Variables (PD)

Y_OFFSET, Y_MIN und Y_MAX are used for the transformation of the manipulated variable within a prescribed range.

MANUAL can be used to switch on and off manual operation. RESET serves to reset the controller.

In normal operation (MANUAL, RESET and LIMITS_ACTIVE = FALSE) the controller calculates the controller error (“e”) as difference SET_POINT – ACTUAL, generates the derivation with respect to time de/ dt and stores these values internally.

The output, that is the manipulated variable Y, is calculated as follows: Y = KP × (D + TV dD/dt) + Y_OFFSET onde D=SET_POINT-ACTUAL.

So besides the P-part also the current change of the controller error (D-part) influences the manipulated variable.

Additionally Y is limited on a range prescribed by Y_MIN and Y_MAX. If Y exceeds these limits, LIMITS_ACTIVE will get TRUE. If no limitation of the manipulated variable is desired, Y_MIN and Y_MAX have to be set to 0.

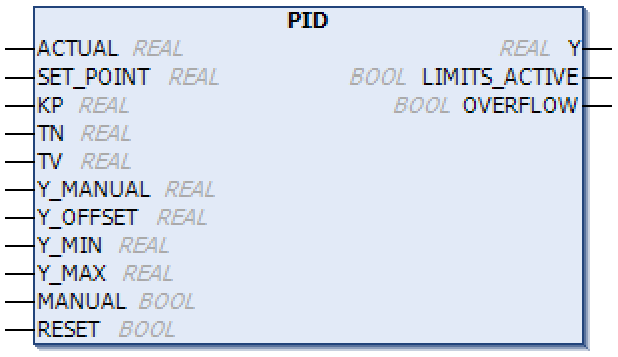
As long as MANUAL=TRUE, Y_MANUAL will be written to Y.

A P-controller can be easily created by setting TV=0.

PID

The library util.library provides the following PID controller function block.

Example of PID in FBD:



Unlike the PD controller, this function block contains a further REAL input TN for the readjusting time in sec (for example "0.5" for 500 msec).

Inputs of the Function Block

Variable	Datatype	Description
ACTUAL	REAL	Current value of the controlled variable
SET_POINT	REAL	Desired value (command variable)
KP	REAL	Proportionality coefficient, unity gain of the P-part
TN	REAL	Reset time, reciprocal unity gain of the I-part; given in seconds, for example "0.5" for 500 msec
TV	REAL	Derivative action time, unity gain of the D-part in seconds, for example "0.5" for 500 msec
Y_MANUAL	REAL	Defines output value Y in case of MANUAL = TRUE
Y_OFFSET	REAL	Offset for the manipulated variable Y
Y_MIN, Y_MAX	REAL	Lower/upper limit for the manipulated variable Y. If Y exceeds these limits, output LIMITS_ACTIVE will be set to TRUE and Y will be kept within the prescribed range. This control will only work if Y_MIN<Y_MAX.
MANUAL	BOOL	If TRUE, manual operation will be active, that is the manipulated value will be defined by Y_MANUAL
RESET	BOOL	TRUE resets the controller. During reinitialization, Y = Y_OFFSET

Table 6-5. Input Variables (PID)

Outputs of the Function Block

Variable	Datatype	Description
Y	REAL	Manipulated value, calculated by the function block
LIMITS_ACTIVE	BOOL	TRUE indicates that Y has exceeded the given limits (Y_MIN, Y_MAX)
OVERFLOW	BOOL	TRUE indicates an overflow

Table 6-6. Output Variables (PID)

Y_OFFSET, Y_MIN und Y_MAX serve for transformation of the manipulated variable within a prescribed range.

MANUAL can be used to switch to manual operation; RESET can be used to re-initialize the controller.

In normal operation (MANUAL = RESET = LIMITS_ACTIVE = FALSE) the controller calculates the controller error ("e") as difference from SET_POINT – ACTUAL, generates the derivation with respect to time de/dt and stores these values internally.

The output, that is the manipulated variable Y unlike the PD controller contains an additional integral part and is calculated as follows: Y = KP × (D + 1/TN edt + TV dD/dt) + Y_OFFSET.

So besides the P-part also the current change of the controller error and the history of the controller error (I-part) influence the manipulated variable.

The PID controller can be easily converted to a PI-controller by setting TV=0.

Because of the additional integral part, an overflow can come about by incorrect parameterization of the controller, if the integral of the error D becomes too great. Therefore for the sake of safety a Boolean output called OVERFLOW is present, which in this case would have the value TRUE. This only will happen if the control system is instable due to incorrect parameterization. At the same time, the controller will be suspended and will only be activated again by re-initialization.

NOTE: As long as the limitation for the manipulated variable (Y_MIN and Y_MAX) is active, the integral part will be adapted, like if the history of the input values had automatically effected the limited output value. If this behavior is not wanted, the following workaround is possible: Switch off the limitation at the PID controller (Y_MIN>=Y_MAX) and instead apply the LIMIT operator (IEC standard) on output value Y.

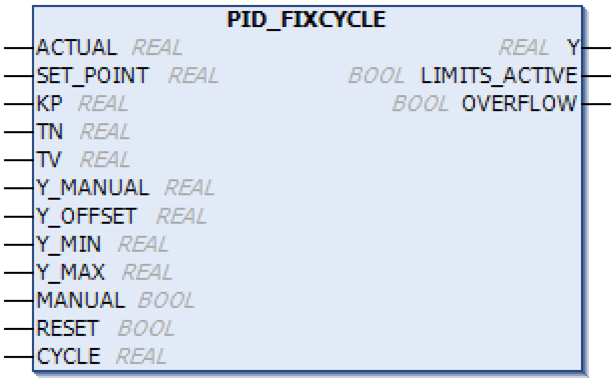
NOTE: Altus recommends using the PID function block available in the library NextoPID and described herein. The PID function block of the library NextoPID has advanced parameters that allow a better fit of the control. The two libraries cannot be used simultaneously.

PID_FIXCYCLE

Provided by util.library.

This function block corresponds to a PID controller with the exception that the cycle time is measured automatically by an internal function, but is defined by input cycle (in seconds).

Example of PID_FIXCYCLE in FBD:



Signal Generators

BLINK

Provided by util.library.

The function block BLINK generates a pulsating signal. The input consists of ENABLE of the type BOOL, as well as TIMELOW and TIMEHIGH of the type TIME. The output OUT is of the type BOOL.

If ENABLE is set to TRUE, BLINK begins to set the output for the time period TIMEHIGH to TRUE and afterwards to set it for the time period TIMELOW to FALSE.

When ENABLE is reset to FALSE, output OUT will not be changed, that is no further pulse will be generated. If you explicitly also want to get OUT FALSE when ENABLE is reset to FALSE, you might use "OUT AND ENABLE" (that is adding an AND box with parameter ENABLE) at the output.

Example of BLINK in CFC:

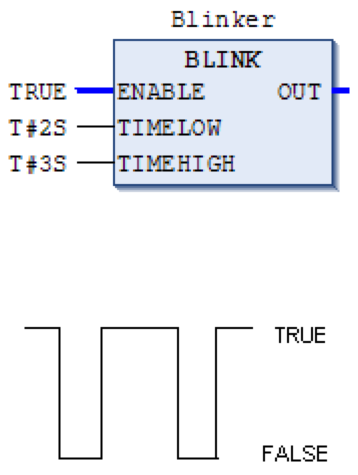


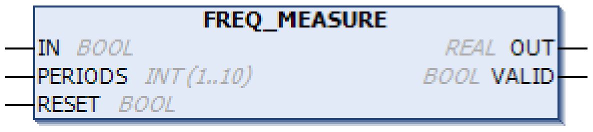
Figure 6-8. Output Results of Block BLINK

FREQ_MEASURE

Provided by util.library.

This function block measures the (average) frequency (Hz) of a boolean input signal. You can specify over how many periods it should be averaged. A period is the time between two rising edges of the input signal.

Example of FREQ_MEASURE in FBD:



Input Variables

Variable	Data Type	Description
IN	BOOL	Input signal
PERIODS	INT	Number of periods, that is the time intervals between the rising edges, over which the average frequency of the input signal should be calculated: 1 to 10.
RESET	BOOL	Reset of all parameters to 0.

Table 6-7. Input Variables (FREQ_MEASURE)

Output Variables

Variable	Data Type	Description
OUT	REAL	Resulting frequency in [Hz]
VALID	BOOL	FALSE until the first measure has been finished, or if the period > 3*OUT (indicating something wrong with the inputs).

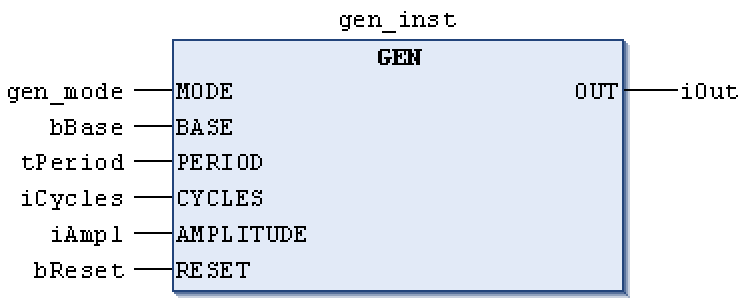
Table 6-8. Output Variables (FREQ_MEASURE)

GEN

Provided by util.library.

The function generator generates typical periodic functions.

Example of GEN in CFC:



MODE describes the function which should be generated, whereby the enumeration values TRIANGLE and TRIANGLE_POS deliver two triangular functions, SAWTOOTH_RISE an ascending, SAWTOOTH_FALL a descending sawtooth, RECTANGLE a rectangular signal and SINE and COSINE the sine and cosine:

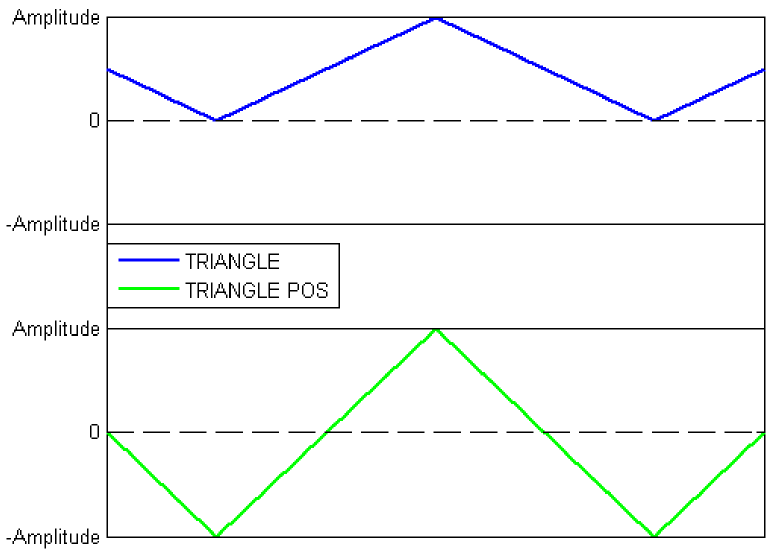


Figure 6-9. Triangles

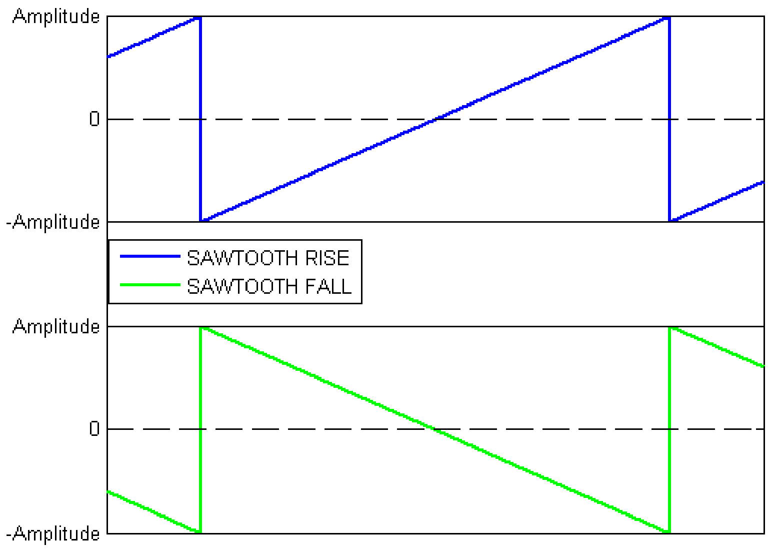


Figure 6-10. Sawtooth Fall

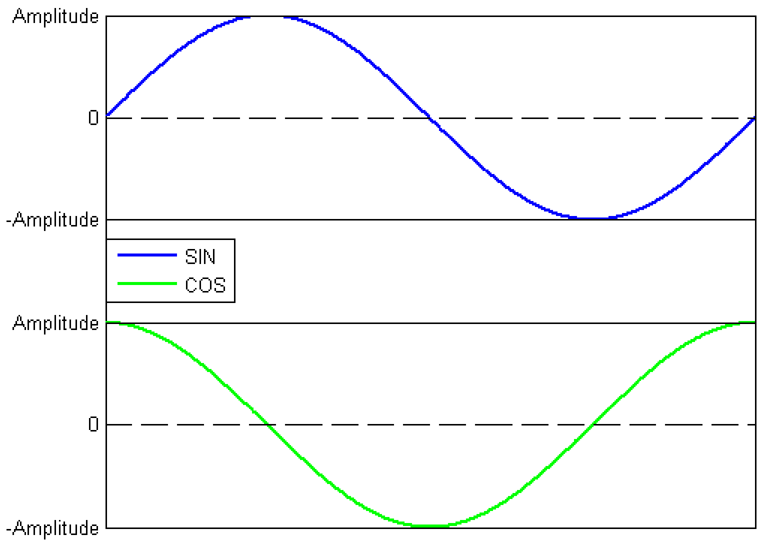


Figure 6-11. Sine/Cosine

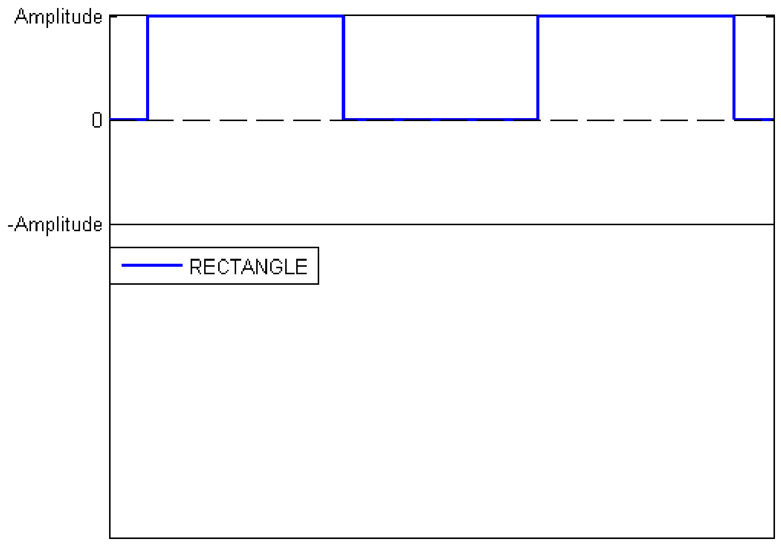


Figure 6-12. Rectangle

BASE defines whether the cycle period is really related to a defined time (BASE=TRUE) or whether it is related to a particular number of cycles, which means the number of calls of function block (BASE=FALSE).

PERIOD or CYCLES defines the corresponding cycle period.

AMPLITUDE defines, in a trivial way, the amplitude of the function to be generated.

The function generator is again set to 0 as soon as RESET=TRUE.

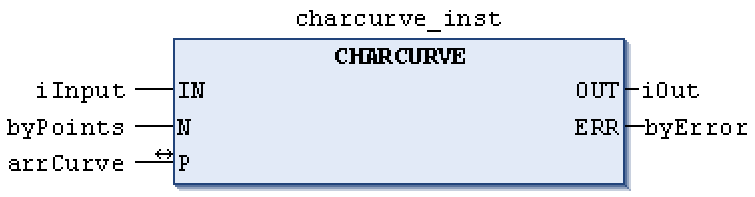
Function Manipulators

CHARCURVE

Provided by util.library.

This function block serves to represent values, piece by piece, on a linear function.

Example of CHARCURVE in FBD:



IN of type INT is fed with the value to be manipulated.

N of type BYTE designates the number of points which defines the presentation function.

P of type ARRAY P[0..10] OF POINT, which is a structure based on two INT values (X and Y), determines this characteristic curve.

OUT of type INT contains the manipulated value.

ERR of type BYTE indicates an error.

The points P[0]..P[N-1] in the ARRAY must be sorted according to their X values, otherwise ERR receives the value 1. If the input IN is not between P[0].X and P[N-1].X, ERR=2 and OUT contains the corresponding limiting value P[0]. Y or P[N-1].Y. If N lies outside of the allowed values which are between 2 and 11, then ERR=4.

Example in ST:

First of all ARRAY P must be defined in the header:

```
VAR
...
CHARACTERISTIC_LINE:CHARCURVE;
KL:ARRAY[0..10] OF POINT:=(X:=0,Y:=0),(X:=250,Y:=50),
(X:=500,Y:=150),(X:=750,Y:=400),7((X:=1000,Y:=1000));
COUNTER:INT;
...
END_VAR
```

Then we supply CHARCURVE with for example a constantly increasing value:

```
COUNTER:=COUNTER+10;
CHARACTERISTIC_LINE(IN:=COUNTER,N:=5,P:=KL);
```

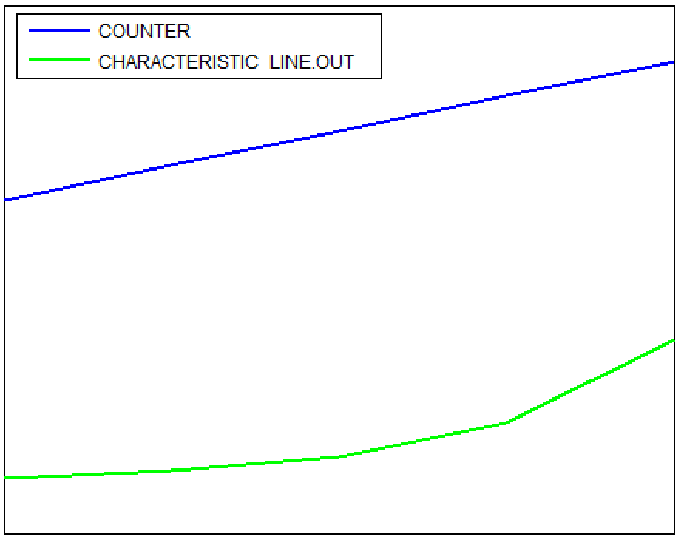


Figure 6-13. Illustration of the Resulting Curves

RAMP_INT

Provided by util.library.

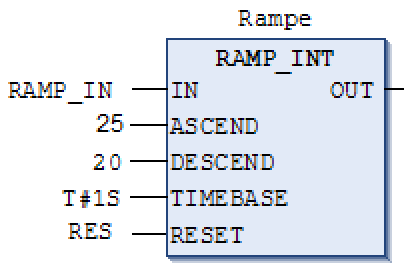
RAMP_INT serves to limit the ascendance or descendance of the function being fed.

The input consists on the one hand out of three INT values: IN, the function input, and ASCEND and DESCEND, the maximum increase or decrease for a given time interval, which is defined by TIMEBASE of the type TIME. Setting RESET to TRUE causes RAMP_INT to be initialized anew.

The output OUT of the type INT contains the ascend and descend limited function value.

When TIMEBASE is set to t#0s, ASCEND and DESCEND are not related to the time interval, but remain the same.

Example of RAMP_INT in CFC:



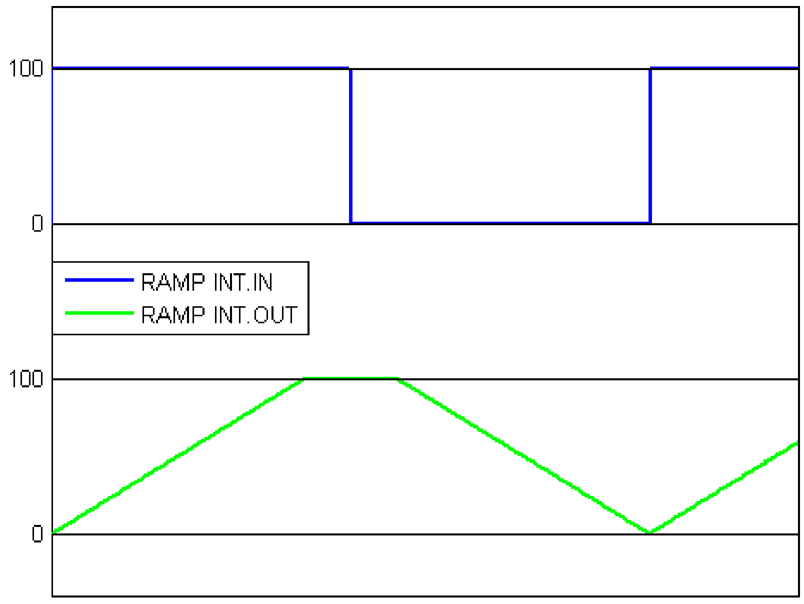


Figure 6-14. RAMP_INT Behavior

RAMP_REAL

Provided by util.library.

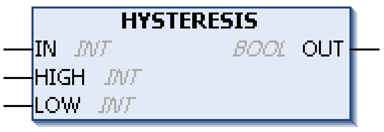
RAMP_REAL functions in the same way as RAMP_INT, with the simple difference that the inputs IN, ASCEND, DESCEND and the output OUT are of the type REAL.

Analogue Value Processing

HYSTERESIS

Provided by util.library.

Example of HYSTERESIS in FBD:



The input to this function block consists of three INT values IN, HIGH and LOW. The output OUT is of type BOOL.

If IN goes below the limiting value LOW, OUT becomes TRUE. If IN exceeds the upper limit HIGH, FALSE is output.

If IN falls below limit LOW, OUT will get TRUE. Not before in the further run IN exceeds the upper limit, FALSE will be output again, until IN once more falls below LOW and thus OUT again gets TRUE.

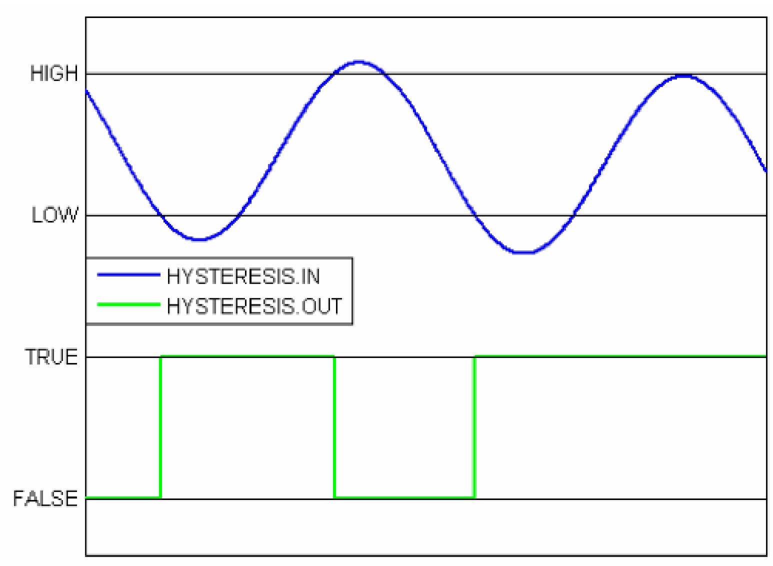


Figure 6-15. Graphical Comparison of Hysteresis.IN and Hysteresis.OUT

LIMITALARM

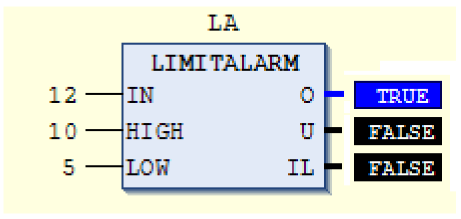
Provided by util.library.

This function block specifies, whether the input value is within a set range and which limits it has violated if it has done so.

The input values IN, HIGH and LOW are each of the type INT, while the outputs O, U and IL are of the type BOOL.

If the upper limit HIGH is exceeded by IN, O becomes TRUE, and when IN is below LOW, U becomes TRUE. IL is TRUE if IN lies between LOW and HIGH.

Example of LIMITALARM in FBD:



NextoPID.library Library

PID

The PID function block is used to control a real process. The block is present in the library NextoPID, which must be added to project.

From a process variable (PV) and setpoint (SP) the functional block calculates the manipulated variable (MV) for the controlled process. This is calculated periodically considering the proportional, integral and derivative factors programmed. This is a control algorithm PID where the proportional gain is the gain of the controller, applied to both the error as to the portions of the integral and derivative of the controller.

The functional block can be represented by the basic diagram of Figure 6-16.

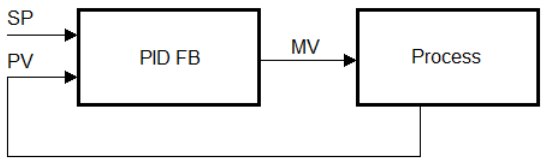


Figure 6-16. Diagram Basic PID

The Figure 6-17 shows the block diagram of a detailed PID loop, as the Nexto CPU execution.

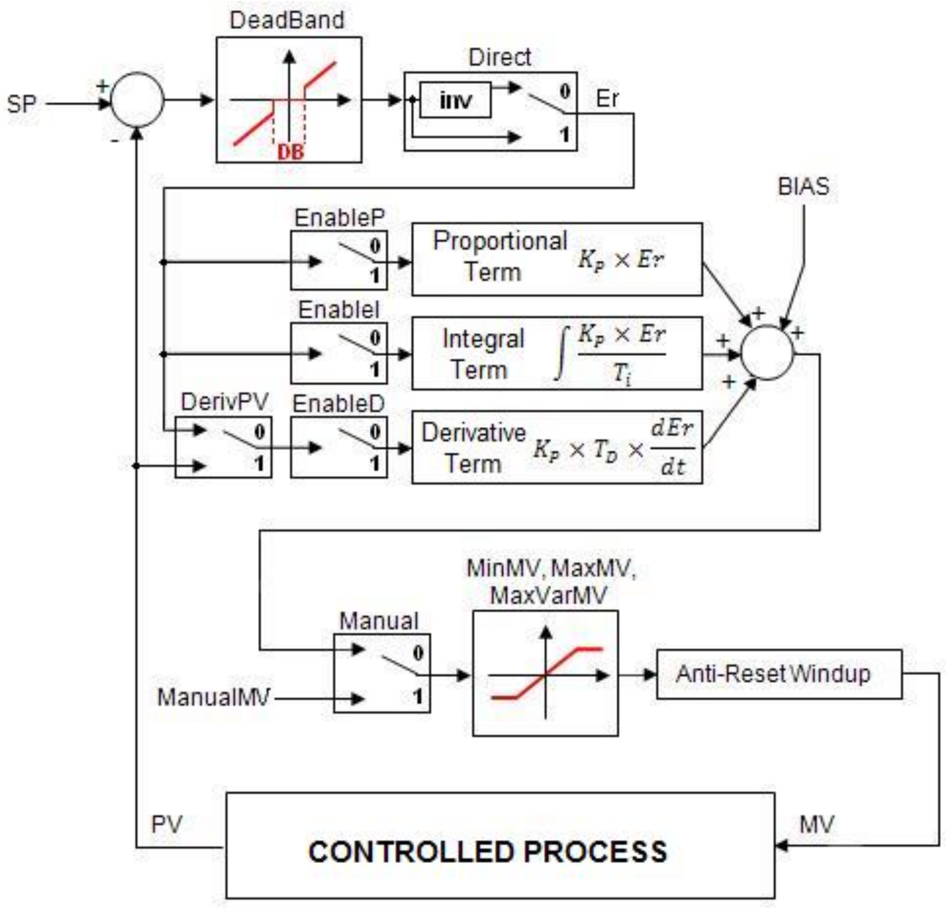


Figure 6-17. Diagram Full PID

Figure 6-18 shows a block diagram of an example of a PID FB controlling a real process. They are also showed auxiliare functions that help and the user must put in the application ("MV to AO Conversion" and "AI to PV Conversion").

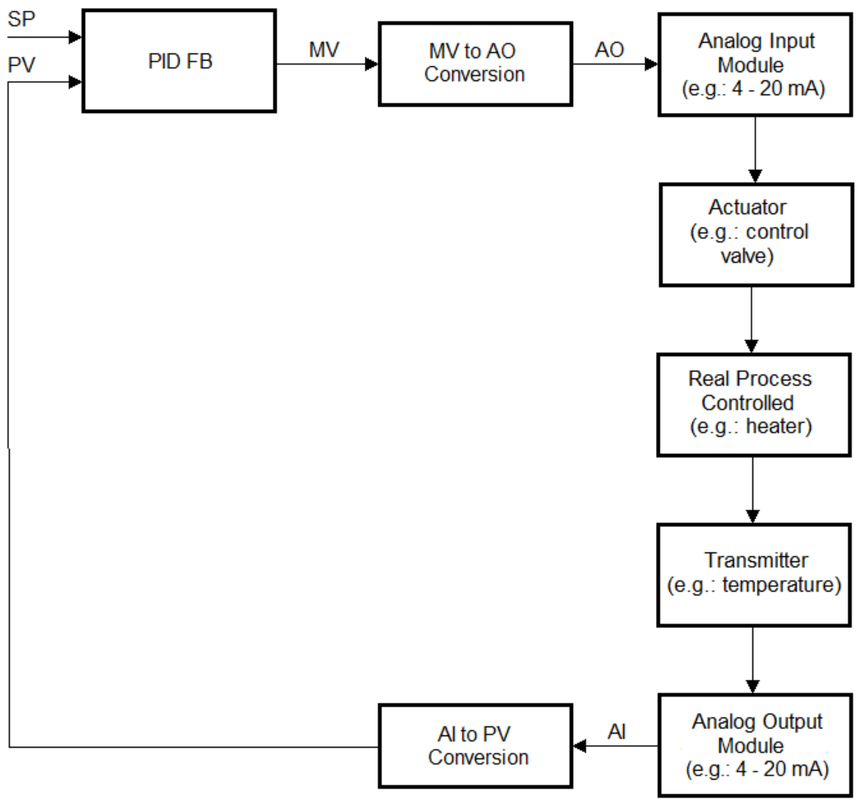

SP
PV
PID FB
MV
MV to AO Conversion
AO
Analog Input Module (e.g.: 4 - 20 mA)
Actuator (e.g.: control valve)
Real Process Controlled (e.g.: heater)
Transmitter (e.g.: temperature)
Analog Output Module (e.g.: 4 - 20 mA)
AI
AI to PV Conversion

Figure 6-18. PID FB Controlling a Real Process

The diagram shows only the main parameters of the PID block. The analog output (AO) is the variable written in analog output module. The analog input (AI) is the variable read from an analog input module.

The AI and AO variables are usually INT type. For example, some input and output analog modules typically operate in the range between 0 and 30000, where 0 corresponds to 4 mA and 30000 corresponds to 20 mA.

In the other hand, the MV and PV parameters of the PID are REAL type. The units and the operating range can be set in the most appropriate. The following examples for PID parameters conversion can be used.

Consider the following examples:

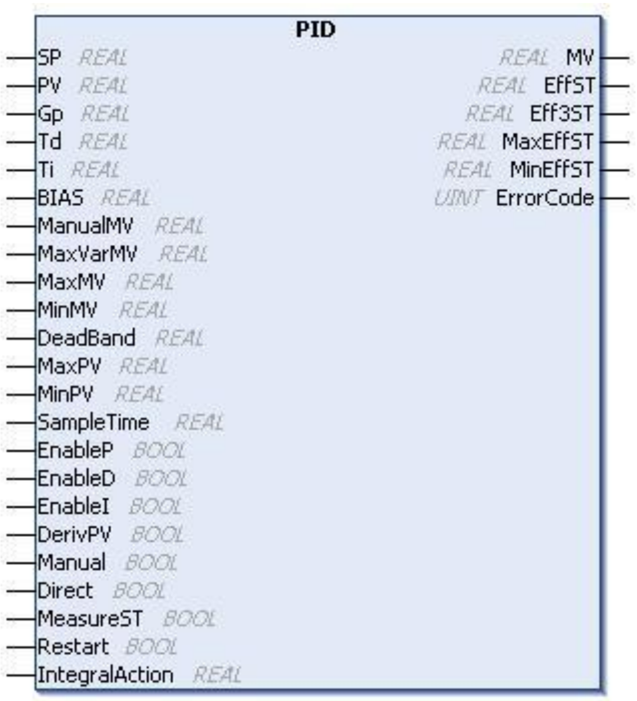
- MV with the same operating range of AO and is not prepared value (eg: 0 .. 30000)
- MV percentage (eg: 0% = control valve fully closed, 100% = control valve fully open)

In each example, the need to observe conversion to implement:

- AO: = REAL_TO_INT (MV);
- AO: = REAL_TO_INT (MV * 30000/100);

NOTE: there are some PID parameters (described below) which impose maximum and minimum values for MV. They are called MaxMV and MinMV respectively. Substituting MV in expressions on the previous examples by MaxMV and MinMV must generate values in the AO operation range (eg: 0 .. 30000). It is necessary to check this to avoid overflow problems.

Example of PID block in FBD:



Input parameters	Type	Description
SP	REAL	Setpoint. Unit and range must be the same as the PV, because the two variables can be compared.
PV	REAL	Process variable. Unit and range must be the same as the SP, because the two variables can be compared.
Gp	REAL	Proportional gain used to calculate the proportional action of the PID block.
Td	REAL	Derivative Time, in seconds, used to calculate the derivative action of the PID block.
Ti	REAL	Integral, Time, in seconds, used to calculate the integral action of the PID block.
BIAS	REAL	Compensation added to the manipulated variable.
ManualMV	REAL	Value attributed to the manipulated variable, when using the manual mode.
MaxVarMV	REAL	Maximum variation in the manipulated variable between the current and previous cycle. If zero or negative, the PID block has no limit of MV variation.
MaxMV	REAL	Maximum value of the manipulated variable. If the calculated value is greater than the configured value, the MV is equal to MaxMV.
MinMV	REAL	Minimum value of the manipulated variable. If the calculated value is less than the configured value, the MV is equal to MinMV.
DeadBand	REAL	Dead time. Minimum amount of error that will cause the correction of MV in automatic mode, i.e., small errors (less than deadband) will not cause changes in the manipulated variable.
MaxPV	REAL	Maximum value of the process variable. If PV value is larger than the configured value PID calculation will stop and will generate an error code output.
MinPV	REAL	Minimum value of the process variable. If PV value is smaller than the configured value PID calculation will stop and will generate an error code output.
SampleTime	REAL	Sampling time. Defines the calling period of PID block, in seconds, ranging from 0.001 s to 1000 s. This parameter is disregarded if the MeasureST is true.
EnableP	BOOL	When true, enables the proportional action of the PID block. If false, the proportional action is set to zero.
EnableD	BOOL	When true, enables the derivative action of the PID block. If false,

		the derivative action is set to zero.
EnableI	BOOL	When true, enables the integral action of the PID block. If false, the integral action is set to zero.
DerivPV	BOOL	When true, the derivative action is calculated in the process variable, being different from zero only when PV is changed. If false, the derivative action is calculated in error, being dependent on the variables SP and PV.
Manual	BOOL	When true, enables the manual mode. If false, enables the automatic mode. The control mode of the PID block affects the way the MV and the action integral is calculated.
Direct	BOOL	When true, you select the direct control, so that MV is considered in the response to be included in the PV. If false, it selects the reverse control. In this case MV is not considered in the response to be included in the PV.
MeasureST	BOOL	When true, the sampling time is measured. If false, the sampling time is entered by the user in the variable SampleTime.
Restart	BOOL	When true, resets PID block, initializing all variables. It can also be used to clear the integral and derivative action, and error codes in the block output.
IntegralAction	REAL	Stores the integral action, which is deleted in error state.

Table 6-9. Input Parameters

Output parameters	**Type**	**Description**
MV	REAL	Manipulated variable.
EffST	REAL	Effective time of sampling, in seconds, used for calculating the derivative action and rate limit of MV.
Eff3ST	REAL	Effective time sampling of the last three cycles, in seconds, used for calculating the derivative action.
MaxEffST	REAL	Maximum value of the effective time of sampling, in seconds, since startup of the PID block.
MinEffST	REAL	Minimum value of the effective time of sampling, in seconds, since startup of the PID block.
ErrorCode	UINT	Error code displayed by the PID block. To remove it, just solve the problem and restart the block via the variable. Restart. In the following the error description: 0: without error 1: MaxMV < MinMV 2: MaxPV < MinPV 3: PV > MaxPV 4: PV < MinPV 5: Ti < 0,001 s (with integral action enabled) 6: Td < 0 s (with derivative action enabled) 7: Gp ≤ 0 8: MaxVarMV < 0 9: DeadBand < 0 10: SampleTime < 0,001 s or SampleTime > 1000 s (with MeasureST = false)

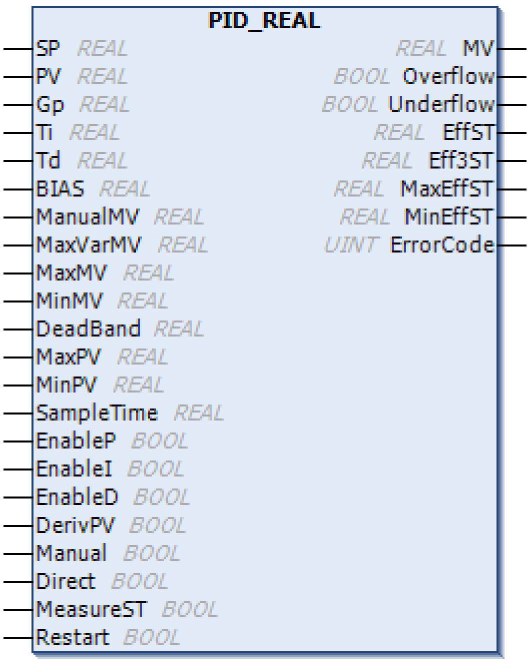
Table 6-10. Output Parameters

PID_REAL

The function block PID_REAL implements an algorithm similar to this library's PID block; However, this block tests input values PV, SP and MV to make sure they are within the specified bounds. If they are out of bounds, the algorithm keeps on executing, but a specific *ErrorCode* output is generated. Also, when MV is saturated, either *Overflow* or *Underflow* outputs will turn TRUE.

This function block doesn't have the VAR_IN_OUT IntegralAction variable.

PID_REAL example in FBD:



Output parameters	Type	Description
SP	REAL	Setpoint. Unit and range must be the same as the PV, because the two variables can be compared.
PV	REAL	Process variable. Unit and range must be the same as the SP, because the two variables can be compared.
Gp	REAL	Proportional gain used to calculate the proportional action of the PID block.
Ti	REAL	Derivative Time, in seconds, used to calculate the derivative action of the PID block.
Td	REAL	Integral, Time, in seconds, used to calculate the integral action of the PID block.
BIAS	REAL	Compensation added to the manipulated variable.
ManualMV	REAL	Value attributed to the manipulated variable, when using the manual mode.
MaxVarMV	REAL	Maximum variation in the manipulated variable between the current and previous cycle. If zero or negative, the PID block has no limit of MV variation.
MaxMV	REAL	Maximum value of the manipulated variable. If the calculated value is greater than the configured value, the MV is equal to MaxMV.
MinMV	REAL	Minimum value of the manipulated variable. If the calculated value is less than the configured value, the MV is equal to MinMV.
DeadBand	REAL	Dead time. Minimum amount of error that will cause the correction of MV in automatic mode, i.e., small errors (less than deadband) will not cause changes in the manipulated variable.
MaxPV	REAL	Maximum value of the process variable. If PV value is larger than the configured value PID calculation will stop and will generate an error code output.
MinPV	REAL	Minimum value of the process variable. If PV value is smaller than the configured value PID calculation will stop and will generate an error code output.
SampleTime	REAL	Sampling time. Defines the calling period of PID block, in seconds, ranging from 0.001 s to 1000 s. This parameter is disregarded if the MeasureST is true.
EnableP	BOOL	When true, enables the proportional action of the PID block. If false, the proportional action is set to zero.
EnableD	BOOL	When true, enables the derivative action of the PID block. If false, the derivative action is set to zero.

EnableI	BOOL	When true, enables the integral action of the PID block. If false, the integral action is set to zero.
DerivPV	BOOL	When true, the derivative action is calculated in the process variable, being different from zero only when PV is changed. If false, the derivative action is calculated in error, being dependent on the variables SP and PV.
Manual	BOOL	When true, enables the manual mode. If false, enables the automatic mode. The control mode of the PID block affects the way the MV and the action integral is calculated.
Direct	BOOL	When true, you select the direct control, so that MV is considered in the response to be included in the PV. If false, it selects the reverse control. In this case MV is not considered in the response to be included in the PV.
MeasureST	BOOL	When true, the sampling time is measured. If false, the sampling time is entered by the user in the variable SampleTime.
Restart	BOOL	When true, resets PID block, initializing all variables. It can also be used to clear the integral and derivative action, and error codes in the block output.

Table 6-11. Input Parameters

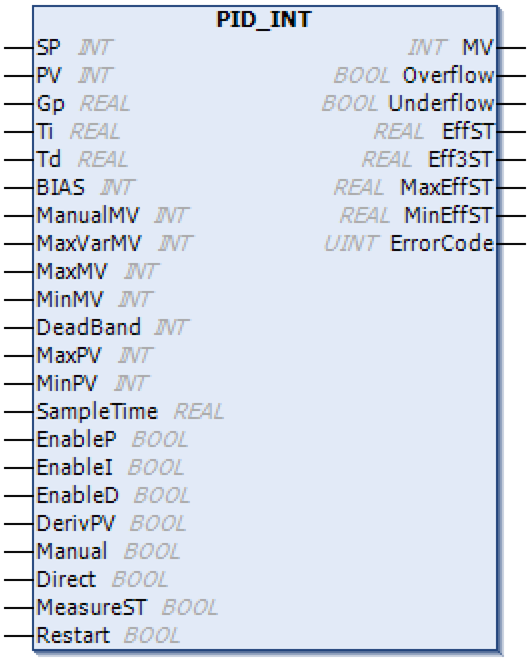
Output parameters	Type	Description
MV	REAL	Manipulated variable.
Overflow	BOOL	If the algorithm calculates an MV value larger than MaxMV, then MV = MaxMV and this output will be TRUE.
Undeflow	BOOL	If the algorithm calculates an MV value smaller than MinMV, then MV = MinMV and this output will be TRUE.
EffST	REAL	Effective time of sampling, in seconds, used for calculating the derivative action and rate limit of MV.
Eff3ST	REAL	Effective time sampling of the last three cycles, in seconds, used for calculating the derivative action.
MaxEffST	REAL	Maximum value of the effective time of sampling, in seconds, since startup of the PID block.
MinEffST	REAL	Minimum value of the effective time of sampling, in seconds, since startup of the PID block.
ErrorCode	UINT	Error code displayed by the PID block. To remove it, just solve the problem and restart the block via the variable. Restart. In the following the error description: 0: without error 1: MaxMV < MinMV 2: MaxPV < MinPV 3: PV > MaxPV 4: PV < MinPV 5: Ti < 0,001 s (with integral action enabled) 6: Td < 0 s (with derivative action enabled) 7: Gp ≤ 0 8: MaxVarMV < 0 9: DeadBand < 0 10: SampleTime < 0,001 s or SampleTime > 1000 s (with MeasureST = false)

Table 6-12. Output Parameters

PID_INT

The function block PID_INT is functionally identical to PID_REAL, but the input variables SP, PV, BIAS, ManualMV, MaxVarMV, MaxMV, MinMV, DeadBand, MaxPV and MinPV and the output variable MV are INT type. This difference is relevant because it allows the direct connection of analog inputs and outputs directly in the inputs and outputs of the block, without any type conversion required.

Example of PID_INT in FBD:



Output parameters	Type	Description
SP	INT	Setpoint. Unit and range must be the same as the PV, because the two variables can be compared.
PV	INT	Process variable. Unit and range must be the same as the SP, because the two variables can be compared.
Gp	INT	Proportional gain used to calculate the proportional action of the PID block.
Ti	REAL	Derivative Time, in seconds, used to calculate the derivative action of the PID block.
Td	REAL	Integral, Time, in seconds, used to calculate the integral action of the PID block.
BIAS	INT	Compensation added to the manipulated variable.
ManualMV	INT	Value attributed to the manipulated variable, when using the manual mode.
MaxVarMV	INT	Maximum variation in the manipulated variable between the current and previous cycle. If zero or negative, the PID block has no limit of MV variation.
MaxMV	INT	Maximum value of the manipulated variable. If the calculated value is greater than the configured value, the MV is equal to MaxMV.
MinMV	INT	Minimum value of the manipulated variable. If the calculated value is less than the configured value, the MV is equal to MinMV.
DeadBand	INT	Dead time. Minimum amount of error that will cause the correction of MV in automatic mode, i.e., small errors (less than deadband) will not cause changes in the manipulated variable.
MaxPV	INT	Maximum value of the process variable. If PV value is larger than the configured value PID calculation will stop and will generate an error code output.
MinPV	INT	Minimum value of the process variable. If PV value is smaller than the configured value PID calculation will stop and will generate an error code output.
SampleTime	REAL	Sampling time. Defines the calling period of PID block, in seconds, ranging from 0.001 s to 1000 s. This parameter is disregarded if the MeasureST is true.
EnableP	BOOL	When true, enables the proportional action of the PID block. If false, the proportional action is set to zero.
EnableD	BOOL	When true, enables the derivative action of the PID block. If false, the derivative action is set to zero.

EnableI	BOOL	When true, enables the integral action of the PID block. If false, the integral action is set to zero.
DerivPV	BOOL	When true, the derivative action is calculated in the process variable, being different from zero only when PV is changed. If false, the derivative action is calculated in error, being dependent on the variables SP and PV.
Manual	BOOL	When true, enables the manual mode. If false, enables the automatic mode. The control mode of the PID block affects the way the MV and the action integral is calculated.
Direct	BOOL	When true, you select the direct control, so that MV is considered in the response to be included in the PV. If false, it selects the reverse control. In this case MV is not considered in the response to be included in the PV.
MeasureST	BOOL	When true, the sampling time is measured. If false, the sampling time is entered by the user in the variable SampleTime.
Restart	BOOL	When true, resets PID block, initializing all variables. It can also be used to clear the integral and derivative action, and error codes in the block output.

Table 6-13. Input Parameters

Output parameters	**Type**	**Description**
MV	INT	Manipulated variable.
Overflow	BOOL	If the algorithm calculates an MV value larger than MaxMV, then MV = MaxMV and this output will be TRUE.
Undeflow	BOOL	If the algorithm calculates an MV value smaller than MinMV, then MV = MinMV and this output will be TRUE.
EffST	REAL	Effective time of sampling, in seconds, used for calculating the derivative action and rate limit of MV.
Eff3ST	REAL	Effective time sampling of the last three cycles, in seconds, used for calculating the derivative action.
MaxEffST	REAL	Maximum value of the effective time of sampling, in seconds, since startup of the PID block.
MinEffST	REAL	Minimum value of the effective time of sampling, in seconds, since startup of the PID block.
ErrorCode	UINT	Error code displayed by the PID block. To remove it, just solve the problem and restart the block via the variable. Restart. In the following the error description: 0: without error 1: MaxMV < MinMV 2: MaxPV < MinPV 3: PV > MaxPV 4: PV < MinPV 5: Ti < 0,001 s (with integral action enabled) 6: Td < 0 s (with derivative action enabled) 7: Gp ≤ 0 8: MaxVarMV < 0 9: DeadBand < 0 10: SampleTime < 0,001 s or SampleTime > 1000 s (with MeasureST = false)

Table 6-14. Output Parameters

Application Notes

Selection of Sample Time

The efficiency of the digital controller is directly related to the sample interval used. As this interval decreases, the result of the digital controller is close to the result of an analog controller. It is advised to use a sample time of the order of one tenth of the time constant of the system, that is: TA = T / 10, where TA is the sample time used and T is the time constant of the system.

Example: The time constant of a first order system can be obtained from its response graph of the manipulated variable (MV) to one step in the setpoint SP with open loop control (PID disabled or mode manual), according the Figure 6-19.

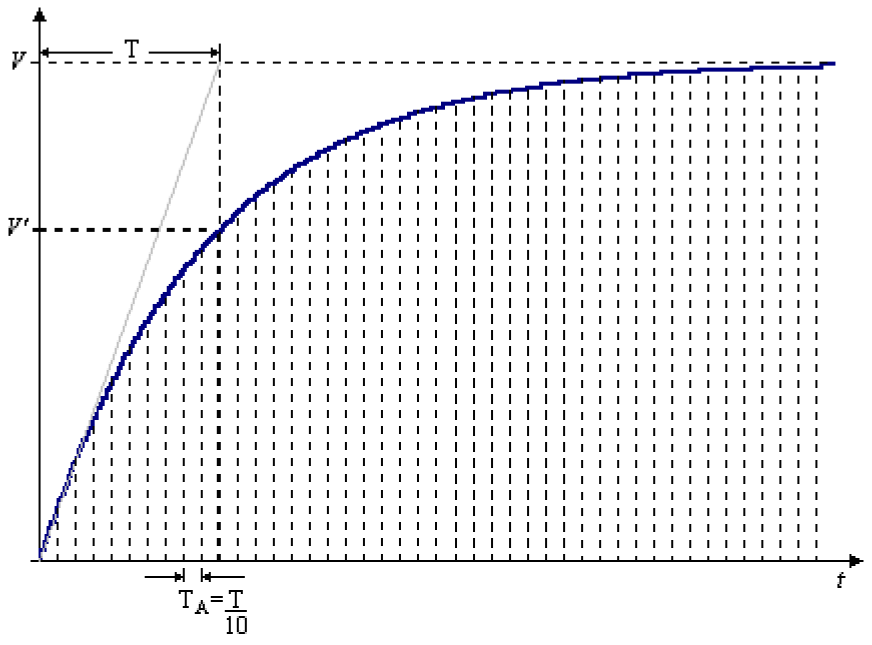


Figure 6-19. Obtaining the Time Constant

Figure 6-19 shows the obtaining of the time constant of the system using two different modes. The most common is taken as the system time constant the time required for the system to reach 63.212% of the final value. Another way is to plot the first derivative of the step response curve, the time constant is the one where this line crosses the final value of the system response.

Once the time constant is defined, you only need to define the sample interval as about one tenth of this value.

It is important to remember that on Nexto series the input and output update occurs on the same order of time of one cycle of the PLC. Whenever the PLC cycle time is longer than the sample time it is advised to use the functions REFRESH_INPUT and REFRESH_OUTPUT.

Feedforward/Bias

Through the memory operand used to feedforward/bias is possible to inject some system variable in the controller output and/or apply a displacement on it.

The objective of the feedforward is to measure the main disturbs of the process and calculate the necessary change in the manipulated variable to compensate it before it cause changes on the process variable.

Can be mentioned as example, a system where the variable to be controlled is the temperature of a hot mixture. At some stage of the process is necessary to pour cold water in this mixture. Without feedforward, would be necessary to wait until the cool water changes the state of the mixture and then the controller generates the corrective action. Using feedforward, a value associated to the temperature of cool water would be injected on the controller output, making it takes corrective action before the cool water starts changing the state of the hot mixture, improving the controller response.

The bias is used whenever it is desired to apply some displacement on the controller output.

Cascade Control

Probably the cascade control is one of the most advanced control techniques used in practice. It is composed by at least two control loops. The Figure 5-20 shows a cascade controller with two loops.

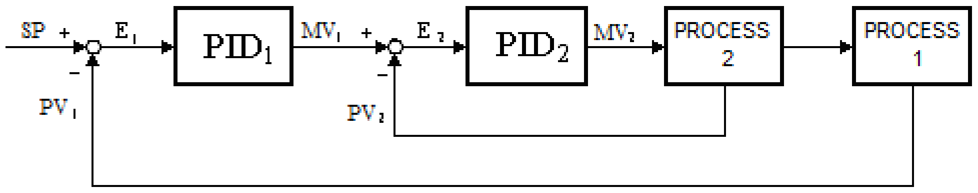


Figure 6-20. Cascade Control with Two Loops

The external loop is called master controller and internal loop is the slave controller. The master controller has its set point fixed and its output provides the set point of the slave controller (MV 1). The manipulated variable of the slave controller (MV 2) operate on the process 2 which will operate on the process 1, closing the loop of the master controller.

This type of controller is applied, for example, on the temperature control by the steam injection. Besides the temperature variation, that must be controlled, the system is subject to pressure variations on the steam line. It is therefore desirable a slave flow controller acting as in function of pressure variations and a master controller to manipulate the slave reference then controlling the process temperature. This example can be represented graphically according the Figure 6-21.

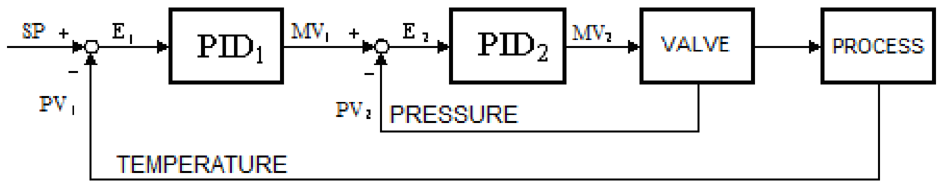


Figure 6-21. Example of Cascade Control Applied

If only a temperature controller were used acting directly on the steam valve, there would not be as to compensate eventual pressure variations on the steam line.

There are three main advantages to use cascade controllers:

- Any disturb that affects the slave controller is detected and compensated by this one before affecting the variable manipulated by the master controller
- Increased controllability of the system. In case of temperature control by steam injection, the system response is improved by the flow controller increasing the controllability of the main loop
- Non-linearities of an internal loop are manipulated into this loop and not perceived by the external loop. In the previous example, the pressure variations are compensated by the slave controller and the master controller notices only a linear relationship between the valve and the temperature

Important Considerations

To use cascade controllers the following precautions should be taken:

- As the slave controller’s setpoint is manipulated as the master controllers output, sudden variation can occur on the slave controller error. If the slave controllers are with the derivative action acting in function of the error, derivative actions will be generated with big values. Therefore it is advisable to use the slave controllers with derivative action in function of the measured variable
- Slave controller must be fast enough to eliminate the disturbs of its loop before they affect the loop of the master controller

Suggestions for Adjustments of the PID Controller

Two methods for the determination of the constants of the PID controller are following presented. The first method consists in determining the constants in function of the oscillation period and of the critical gain, while the second one determines the controller constants in function of the time constant (T), of the dead time (Tm) and of the statistics gain of the system (K). For further details we recommend reading the referenced literature.

WARNING: Altus Sistemas de Automação S.A. is not responsible for any damage caused by configuration errors of the constants of the controller or parameterization. It is recommended that suitably qualified person to execute this task.

Determination of Controller Constants Through the Period and the Critical Gain

This method generates a damped response which damping rate is equal to one quarter. That is, after tuning a loop through this method, it is expected a response as shown in Figure 6-22.

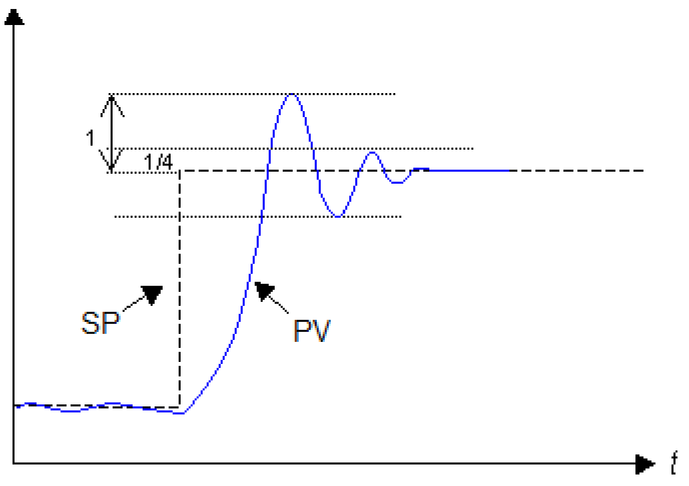


Figure 6-22. Damped Response

The critical gain is defined as the gain of a proportional controller that generates an oscillation with constant amplitude on the closed loop system, while the critical period is the period of oscillation. The critical gain is a measure of system controllability, that is, it is easier as controlling the system as higher is the critical gain. The critical period of oscillation is a measure of the response speed of the closed loop system, that is, the period of oscillation is longer as the system as will be slower. During this chapter the critical gain is named as GPc and a period critical as Tc.

It is important to remember that gains slightly less than GPc generate oscillations which period decreases along the time, while the gains higher than GPc generate oscillations which amplitude increases along the time. In case of gains is higher than GPc it is necessary to be careful and do not make the system critically unstable.

The process to determine GPc and Tc consists in closing the loop with the controller in automatic mode and disabling the integral and derivative actions. The steps are the following:

- Remove the integral and derivative actions through the respective input parameters
- Increase the proportional gain with small increments. After each increment insert a small system disturb on the system using a small step on the set point (SP)
- To verify the behavior of the system (PV), the oscillation amplitude must increase as the gain increases. The critical gain (GPc) will be which generates oscillations with constant amplitude (or almost constant) according the Figure 6-23
- Measure the period of these oscillations (Tc)

To determine the controller constants just apply the values of Tc and GPc on the formulas of the Table 6-15

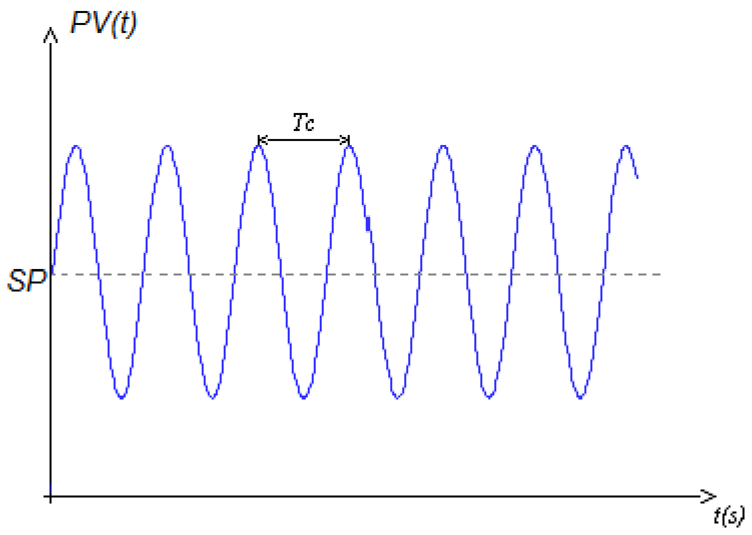


Figure 6-23. Oscillations with Constant Amplitude

Controller Type	Constant
Proportional (P)	GP = 0,5.GPc

Proportional and Integral (PI)	GP = 0,45.GPc
	Ti = Tc/1,2

Proportional, Integral and Derivative (PID)	Gp = 0,75.GPc
	Ti = Tc/1,6
	Td = Tc / 10

Table 6-15. Values of GPc e Tc

Determination of Controller Constants Through the Process Constants

This method is applied well to linear processes, of first order (similar to a RC circuit) and with dead time. In practice, many industrial processes fit into this model.

The method requires, initially, determining the following process characteristics using opened loop:

- K: Static gain of process. It is defined as the ratio between a PV variation and a MV variation, that is, K = ΔPV/ΔMV
- Tm: Dead time, defined as the time between the beginning of a variation on output MV (t0) and the start of the system reaction
- T: Time constant of the system, defined as the time which the process variable takes to reach 63.212% of its final value

Furthermore, the method requires two additional parameters which are not characteristics of the process itself, and should be entered by the user:

- Tr: Response time desired after the loop tuning. This is an interesting feature, because through this parameter the user can enter a condition of performance of the controlled loop
- dt: Sample time in seconds, that is, the period to call the functional block and update the PV input PV and MV output. The constant dt symbolizes an additional dead time, which must be added to Tm. In practice, dt/2 is added to the Tm value, as this is the average dead time inserted

The response time Tr can be compared with a time constant of the closed loop, as shown in Figure 6-24.

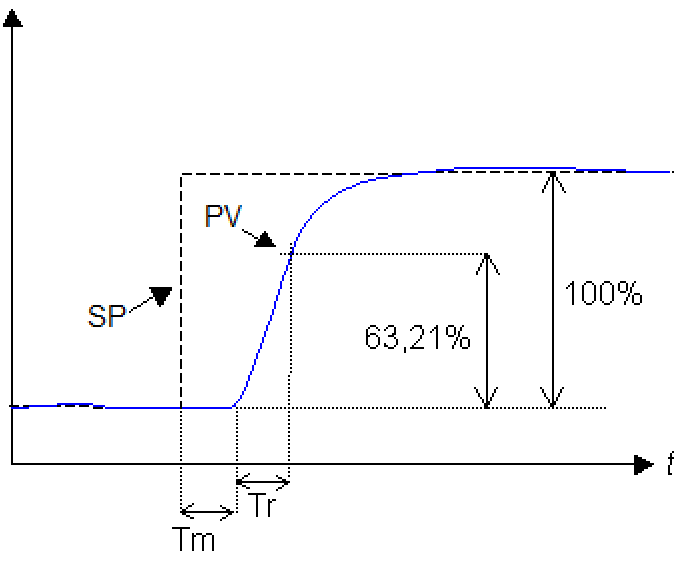


Figure 6-24. Tr Compared with a Time Constant

The parameter Tr, on Figure 5-24, shows the response time desired. It is the measured time between the start of the system response (after the dead time Tm), and when PV reaches 63.21% of its total excursion. Through Tr the user can specify a performance requirement for the controlled loop. It should be careful to not specify response times lower than one tenth of the constant time of the system, because otherwise the system can become unstable. Than Lower as is the value of Tr as greater is the necessary gain.

Afterwards, it is described how to determine, through an opened loop test, the other parameters (K, Tm, and T), that characterize the process. A simple way to determine these process constants is putting the PID function block in manual mode, generate a small step on MV and plot the PV response time. For slow process it can be done manually, but for rapid processes it is advisable the use of an oscilloscope or other device that monitors the PV variation. The step on MV should be large enough to cause a noticeable variation on PV.

The Figure 6-25 represents a step on the MV output, applied at time t0, and the response of a linear system of first order with dead time.

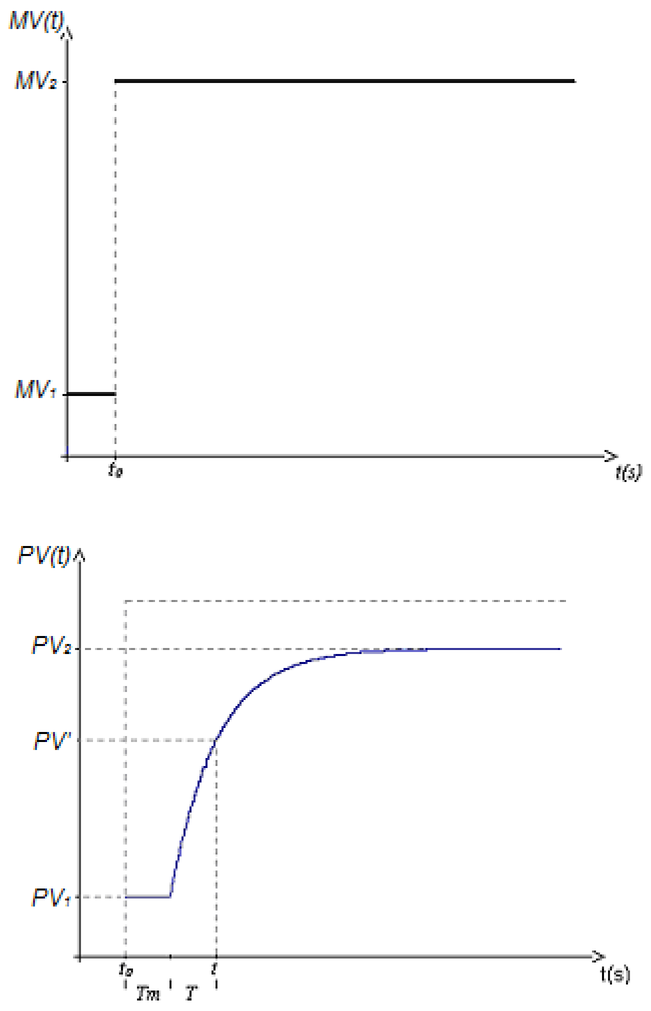


Figure 6-25. Step in MV and System Response to Step

Through Figure 5-25, all required constants for the determination of the controller parameters can be obtained. The static gain of the process is obtained by varying the ratio between the process variable and the variation rate of manipulated variable, that is:

$$K = \frac{PV_2 - PV_1}{MV_2 - MV_1}$$

The dead time, Tm, is the time between the application instant of the step on MV (t0) and the beginning of the system response.

The system time constant, T, is the time between the beginning of the system reaction and 63.212% of the final value of PV, that is:

$$0.63212 = \frac{PV' - PV_1}{PV_2 - PV_1}$$

From the system constants, K, Tm and T, controller parameters can be obtained using the formulas of the Table 6-16.

Controller Type	Constants

Proporcional, Integral e Derivativo (PID)	$GP = \frac{T}{K * (Tr + Tm + dt/2)}$
	Ti = T
	Td = Tm/2 + dt/4

Table 6-16. Controller Parameters

Gains X Scales

It is important to remember that the proportional gain only perform its action correctly when both the input and output of the system are using the same scales. For example, a proportional controller with unitary gain and input (PV) using the range 0-1000 only be really unit if the output range (MV) is also 0-1000.

In many cases the input and output scales are different. There may be mentioned as an example a system where the input analog card is 4-20 mA, where 4 mA corresponds to the value 0 and 20 mA corresponds to the value 30000. And the analog output card is 0 V to 10 V, where 0 V corresponds to the value 0, and 10 V corresponds to the value 1000. In this cases, the scales adjustment can be made by the proportional gain instead of a normalization of the input or output values.

One strategy that can be adopted is, initially, determine the gain in terms of percentage (independent on the scales), without worrying about the type of input and output analog modules used. After that, with this gain determined, the scale corrections must be executed, before introducing the proportional gain on the PID function block.

The strategy consists in determining the proportional gain of the system using the percentage range (0% to 100%) of both the process variable (PV) and the manipulated variable (MV), regardless of the absolute values of both PV and MV.

It will lead to the determination of a proportional gain called GP%. This gain GP% cannot be used directly in the PID function block. Before it is necessary to fix the scales, considering the absolute values of these variables.

WARNING:
In the previous section, **Suggestions for Adjustments of the PID Controller** are suggested methods of adjustment in which the scale corrections are implicit to the method and it should not be considered. In the next section, **Application Example**, the scale corrections are also unnecessary, because it is used one of the methods discussed in the section **Suggestions for Adjustments of the PID Controller**.

The scale corrections are described by the following example.

Consider an air conditioning system where the analog input module is reading a PTC resistor (positive thermal coefficient) and the analog output module generates a voltage of 0 to 10V to act on the valve responsible for circulating the water that cools the air blown.

The input module works with a range from 0 to 30000, but the useful range is 6634 to 8706 with the following meaning:

- EA0 = 6634 = 0% = 884.6 Ω (corresponding to the minimum temperature which can be measured)
- AI1 = 8706 = 100% = 1160.9 Ω (corresponding to the maximum temperature that can be measured)

The output module uses the same range from 0 to 30000 with no restrictions and with the following meaning:

- SA0 = 0 = 0% = 0 V (corresponding to the minimum water flow through the valve)

- SA1 = 30000 = 100% = 10 V (corresponding to the maximum water flow through the valve)

Assuming that the gain GP% was previously determined, the gain GP can be calculated using the following equation:

GP = GP% * R

Where:

$$R = \frac{SA1 - SA0}{EA1 - EA0}$$

For the previous example:

$$R = \frac{3000 - 0}{8706 - 6634} = 14,478$$

This R ratio is a constant that, when multiplied by the proportional gain of the controller, compensates the differences between the input and output ranges without the necessity of a direct normalization.

Application Example

This section will show a practical example of using the PID function block, covering various stages of process design and its control system.

Process Description

The example process has as objective the supply of heated water, at controlled temperature, for a consumer. The heating will be done using a gas burner being controlled from the flow gas variation through a valve.

The Figure 6-26 illustrates this process.

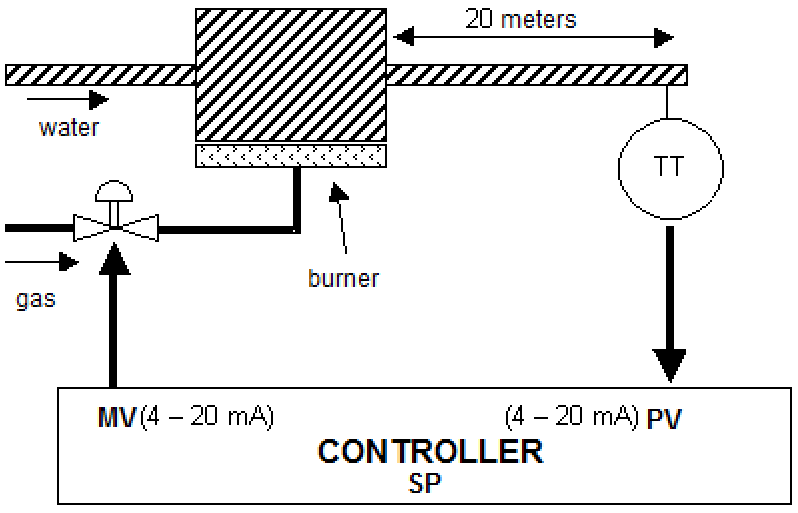


Figure 6-26. Sample Temperature Control

It is observed that the temperature transmitter (TT) is located near the consumer, which is 20 meters from the point of heating the water. Processes like this are good examples of how dead times can be introduced. This is because the heated water at the point of heat takes some time to cover the distance from the measuring point to the consumer. Dead times were previously discussed (Figure 6-24).

Some hypotheses were assumed on this process modeling:

- It is assumed that the water reaches the point on the heating burner is fixed temperature of 30 ºC
- It is assumed that water flow is constant

- It is assumed that the gas pressure is constant
- Some characteristics of this process and the elements used are defined following
- The heated water must have its temperature set between 50 ºC and 80 ºC
- The temperature transmitter TT have output from 4 to 20 mA, and it behaves linearly, such that 4 mA correspond to 30 ºC and 20 mA corresponds to 130 ºC
- It is assumed that to increase the water temperature in 10 ºC, it is necessary to inject 1 m^3/h of gas. This behavior is linear
- The gas valve closes with 4 mA, injecting 0 m^3/h of gas. On the other hand, with 20 mA, it injects 8 m^3/h of gas

Description of Analog Modules

As can be seen in the Figure 6-26, it requires an analog output of 4-20 mA, and an analog input of 4-20 mA, such as interfaces between the controller and process.

Internally to the controller, these ranges of 4-20 mA correspond to variables (PV and MV). These numerical range values can vary depending on the input and output analog modules selected. In this example, it is assumed the following:

PV analog input (0 to 30000):

PV = 0 ---> 4 mA ---> 30 ºC

PV = 30000 ---> 20 mA ---> 130 ºC

MV analog output (0 to 10000):

MV = 0 ---> 4 mA = 0 ---> 0 m^3/h

MV = 10000 ---> 20 mA ---> 8 m^3/h

Setpoint

The variable SP must be used to program the desired temperature between 50 ºC and 80 ºC.

As this variable must be compared with PV, it must have the same numeric range of PV, that is:

SP = 0 ---> 30 ºC

SP = 30000 ---> 130 ºC

Or to restrict the range between 50 ºC and 80 ºC:

SP = 6000 ---> 50 ºC

SP = 15000 ---> 80 ºC

General Block Diagram and Limit Values

The Figure 6-27 shows an overall of the system block diagram (controller + process), where into the controller is shown the PID function block. Note that SP, PV and MV are variables of the controller.

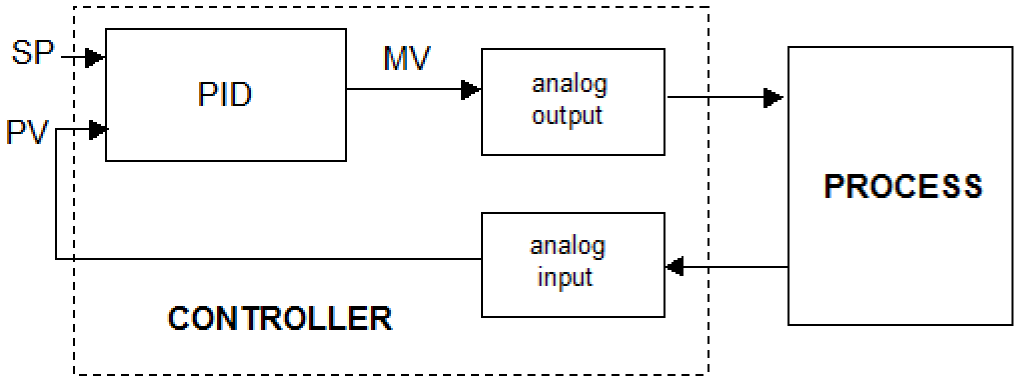


Figure 6-27. Block Diagram of the PID Function Block

SP:

minimum = 6000 (50 ºC)

maximum = 15000 (80 ºC)

PV:

minimum = 0 (30 °C)

maximum = 30000 (130 ºC)

MV:

minimum = 0 (0 m³/h)

maximum = 7500 ---> (6 m³/h)

It is observed that in the case of MV, although the valve is able to inject 8 m³/h, it is desirable to limit this flow in 6 m³/h.

Process Parameters

The Figure 6-28 shows the result of a test on the opened loop process. To perform this test, it was used the variables MV and PV directly, with its internal units.

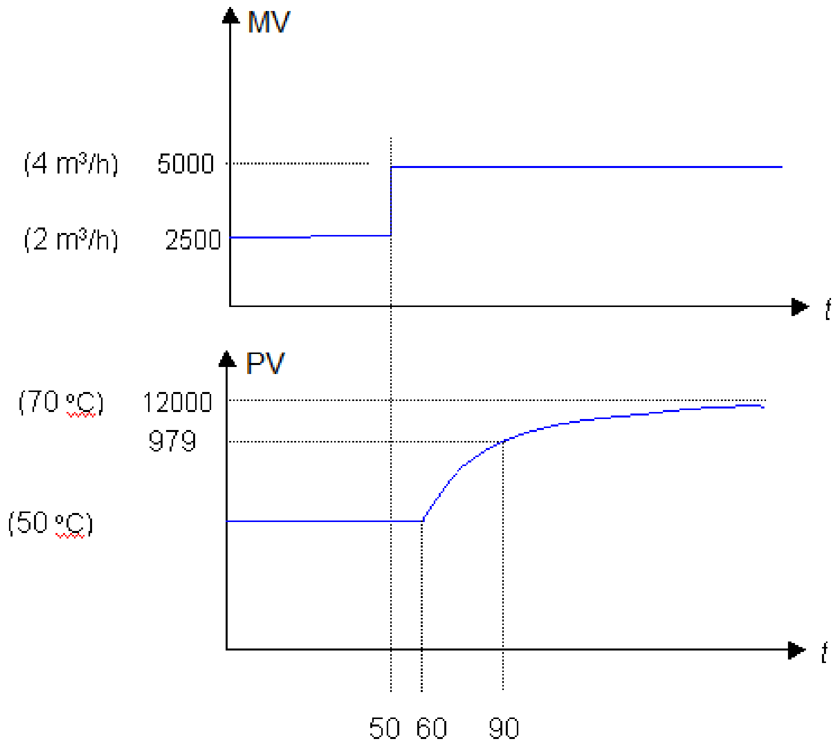


Figure 6-28. Open Loop Test

From Figure 6-28 can determine the three basic parameters, as explained above in the section Application Notes.

Tm = 10 seconds (dead time, as the step was applied at t = 50 s and started responding at t = 60 s).

T = 30 seconds (time constant, as the response started at t = 60 s, and reached 63.21% of excursion at t = 90 s):

9792 = 6000 + (12000 - 6000) * 0,6321.

K = 2.4 (static gain of the process)

$$2.4 = \frac{12000 - 6000}{5000 - 2500}$$

Tuning Controller

As the test opened loop was used, it will use the second tuning method described in the Application Notes.

To use this method, besides the process parameters determined in the previous section (Tm, T and K), it is also necessary that the user enter other two parameters:

- Tr: Or response time desired. In this example, it will be arbitrated in 10 seconds (one-third of the constant time using opened loop)
- dt: Or cycle time of the PID function block. As mentioned above, this time must be less than 10 times the constant time using opened loop, or even less. Therefore, the value must be less than 3 seconds. It was selected dt = 1 second

Now, it is possible to apply the equations of the method:

GP = T / (K * (Tr + Tm + Dt/2)) = 30 / (2.4 * (10 + 10 + 1/2) = 0,609

Ti = T = 30 s/rep

Td = Tm/2 + Dt/4 = 10/2 + 1/2 = 5.25 s

LibRecipeHandler

This library enables the manipulation of Recipes by the PLC.

ATTENTION:
A system library is used for this library to work correctly. This library is added to the project when the Recipe Manager object is added. If Recipe Manager is not added, a compilation error message will be presented if LibRecipeHandler is used.

WriteRecipe

This function allows the writting of values from a recipe inside a Recipe Definition object to application variables running in the PLC. The input parameters of this function are described at Table 6-17.

Input Parameters	Type	Description
sRecipeDefinitionName	STRING	STRING with the name of the Recipe Definition object where the Recipe to be written is. This field is case sensitive, take care to match names perfectly.
sRecipeName	STRING	STRING with the name of the Recpe to be written to and defined within the Recipe Definition object. This field is case sensitive, take care to match names perfectly.

Table 6-17. Input Parameters

When the funtion is executed successfully, it will return 0. When there's an error, it will return an error code. The function's return is a DWORD type, but it can be declared as an enumerable named RECIPE_RETURN_VALUES. The possible return codes are presented at Table 6-18.

Error Code	Value	Description
ERROR_OK	16#0	Function executed correctly.
ERROR_RECIPE_NOT_FOUND	16#4003	The recipe name within the Recipe Definition object is either missing or wrong. It also indicates an error in the STRING input parameter. For instance, the STRING cannot be null and must be no larger than 60 characters, and cannot have the '|', '.' or '/' characters.
ERROR_RECIPE_DEFINITON_NOT_FOUND	16#4004	The Recipe Definition object name is either missing or wrong. It also indicates an error in the STRING input parameter. For instance, the STRING cannot be null and must be no larger than 60 characters, and cannot have the '|', '.' or '/' characters.
ERROR_NO_RECIPE_MANAGER_SET	16#4006	There's no Recipe Manager object in the project where the function was added. This error happens when function parameters are consistent, but failed subsequent function verifications.

Table 6-18. Function's Error Codes

Before executing the write command, the function will consist the input parameters. If there's any inconsistency, it will indicate an error in the Recipe or Recipe Definition. If the parameters' STRINGs are valid, the function will attempt to write. If the Recipe or Recipe Definition are not loaded in the PLC, an error may happen.

ST usage example with a RECIPE_RETURN_VALUES declaration for the function's return:

```
VAR
   sName : STRING := 'Recipe001';
   sDef : STRING := 'Recipes';
   mRET :RECIPE_RETURN_VALUES;
   boolStartProcess : BOOL;
END_VAR
mRET := WriteRecipe(SDef,SName);
IF mRET <> RECIPE_RETURN_VALUES.ERROR_OK THEN
   boolStartProcess:= FALSE;
ELSE
   boolStartProcess := TRUE;
END_IF;
```

7. Glossary

Active CPU	In a redundant system, the active CPU performs the control of the system by reading the values of points of entry, running the program application and triggering the values of outputs.
Algorithm	Finite sequence of well-defined instructions to solve problems.
Application Program	Is the program loaded into a PLC, which determines the operation of a machine or process.
Assembly Language	Microprocessor programming language, also known as machine language.
Bit	Basic information unit, it may be at 1 or 0 logic level.
Breakpoint	Breakpoint in application for debugging.
Broadcast	Simultaneous dissemination of information to all nodes connected to a communications network.
Bus	Set of interconnected I/O modules to a CPU or to a head of network field.
Byte	Information unit composed by eight bits.
CAN	Communication protocol widely used in automotive networks.
Communication network	Set of equipment (nodes) interconnected by communication channels.
Context Menu	Dynamic Menu with content according to the current context.
CPU	Central Processing Unit. It controls the data flow, interprets and executes the program instructions as well as monitors the system devices.
Cycle	Complete application program implementation of a programmable controller.
Default	Preset value for a variable, used in case there is no definition.
Diagnostics	Procedure used to detect and isolate faults. It is also the set of data used for this determination, which serves for the analysis and remediation.
Download	Load configuration or program in the PLC.
Frame	A unit of information transmitted on the network.
Gateway	Equipment for connecting two communication networks with different protocols.
Hardware	Physical equipment used in data processing which normally run programs (software).
IEC 61131	Generic standard for operation and use of PLCs. Former IEC1131.
Interface	Adapts electric and/or logically the transfer of signals between two devices.
Input/output	Also called I/O. I/O devices on a system. In the case of PLCs typically correspond to modules digital or analogue output or input monitoring or driving the controlled device.
I/O	See Input/Output.
I/O Module	Module belonging to subsystem of inputs and outputs.
Kbytes	Memory size unit. Represents 1024 bytes.
Login	Action to establish a communication channel with the PLC.
Master	Equipment connected to a communications network where originate command requests to other network equipment.
Menu	Set of options available and displayed by a program in the video and that can be selected by the user to activate or to perform a particular task.
Module (hardware)	Basic element of a system with very specific functionality. It’s normally connected to the system by connectors and may be easily replaced.
Node	Any station of a network with communication skills using an established protocol.
Operands	Elements on which the instructions work. Can represent constants, variables, or a set of variables.
PLC	Acronym for programmable controller. See Programmable Controller.
POU	Program Organization Unit, is a subdivision of the application program that can be written in any of the available languages.
Programmable Controller	Also known as PLC. Equipment controlling a system under the command of an application program. It is composed of a CPU, a power supply and I/O modules.
Programming Language	A set of rules and conventions used for the elaboration of a program.
Protocol	Procedural rules and conventional formats which, upon control signals, allow the establishment of a data transmission and error recovery between equipment.
RAM	Random access memory. Is the memory where all addresses can be accessed directly in a random way and with the same speed. Is volatile, i.e., its content is lost when the equipment is de-energized, unless if you have a battery for retaining values.
Reset	Command to reboot the PLC.
RUN	Command to put PLC in execution mode.
Set	Action to assign the logical high level state to a boolean variable.
Slave	Equipment connected to a communications network that transmits data only if it is requested by other equipment called master.

Software	Computer programs, procedures and rules related to the operation of a data processing system.
STOP	Command to freeze the PLC in its current state.
Sub network	Segment of a communication network that connects a group of devices (nodes) with the goal of isolating the local data traffic or using different protocols or physical media.
Supervisory Station	Equipment connected to a PLC network with the goal of monitoring and controlling the process variables.
Tag	Name associated with an operand or the logic that allows a brief identification of your content.
Timeout	Maximum preset time to a communication to take place. When exceeded, then retry procedures are started or diagnostics are activated.
Token	Is a tag that indicates who is the master of the bus at the time.
Tooltip	Text box with a help or where you can enter with the help.
Tree	Data structure for hardware configuration.
Upload	Reading of the program or configuration from the PLC.
Watchdog	Electronic circuit that checks the equipment operation integrity.
Word	Information unit composed by 16 bits.
XML	Extensible Markup Language: is a standard for generating markup languages.
Zoom	In the context of keyboard function window, is used for the exchange of screens.