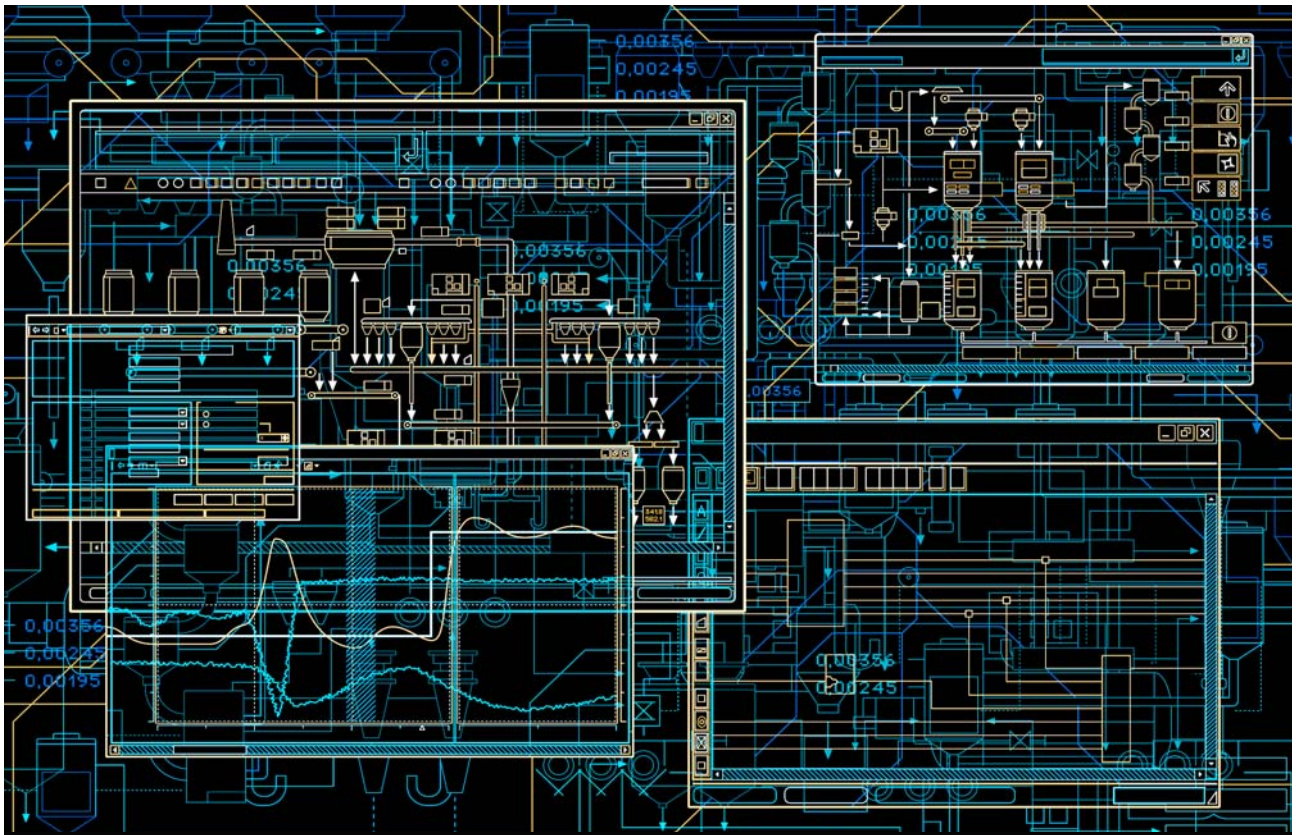


AC 800M

Library Object Style Guide

System Version 6.0

Power and productivity
for a better world™

AC 800M

Library Object Style Guide

System Version 6.0

NOTICE

This document contains information about one or more ABB products and may include a description of or a reference to one or more standards that may be generally relevant to the ABB products. The presence of any such description of a standard or reference to a standard is not a representation that all of the ABB products referenced in this document support all of the features of the described or referenced standard. In order to determine the specific features supported by a particular ABB product, the reader should consult the product specifications for the particular ABB product.

ABB may have one or more patents or pending patent applications protecting the intellectual property in the ABB products described in this document.

The information in this document is subject to change without notice and should not be construed as a commitment by ABB. ABB assumes no responsibility for any errors that may appear in this document.

In no event shall ABB be liable for direct, indirect, special, incidental or consequential damages of any nature or kind arising from the use of this document, nor shall ABB be liable for incidental or consequential damages arising from use of any software or hardware described in this document.

This document and parts thereof must not be reproduced or copied without written permission from ABB, and the contents thereof must not be imparted to a third party nor used for any unauthorized purpose.

The software or hardware described in this document is furnished under a license and may be used, copied, or disclosed only in accordance with the terms of such license. This product meets the requirements specified in EMC Directive 2004/108/EC and in Low Voltage Directive 2006/95/EC.

TRADEMARKS

All rights to copyrights, registered trademarks, and trademarks reside with their respective owners.

Release: August 2014
Document number: 3BSE042835-600

TABLE OF CONTENTS

About This User Manual

General 9
Document Conventions 10
Warning, Caution, Information, and Tip Icons 10
Terminology 11
Related Documentation 11

Section 1 - Libraries

Introduction 13
- Purpose 13

Library Categories 14
- Standard Libraries 14
- User Defined Libraries 15
- Object Libraries 15

Library Dependencies 15
- Connected Libraries 15
- Split Libraries 16
- Support Libraries 16

Naming Convention 17

Section 2 - Types

Naming 19
- Object Types 19
- Compound Words and Abbreviations 21
- Suffixes 22
- Name Space 24

Section 3 - Parameter Interface

Naming Convention 25
Compound Words and Abbreviations 25
Parameter Properties 26
Data Type 27
Attributes 27
Parameter for Control Modules and Diagrams 28
Parameter for Function Blocks 30
FD Port 30
Initial value 31
Description 31
Parameters with Special Purposes 33
Name 33
Description 34
InteractionPar 36
ParError 37
Connections 37
Parameters for Alarm Handling 39
Monitoring Continuous Execution 39

Section 4 - Engineering Interface

General 42
Template Design 43
Control Module Design 43
Graphical Layers 43
Grid and Coordinate System 46
Layers and Interaction Windows 47
Icons 48
Function Block Design 51
Parameter Names 51
Function Block Example 52
Diagram Design 57

Data Flow Order 58
Execution order 60
Reverse and Display value attribute on data types in diagrams 61
Reverse attribute on data types in Diagram types 61
Variable and Parameter 62
Interaction Windows in Online Mode 63
Introduction 63
Interaction Windows 65
Information Windows 65
When to use Interaction Windows 65
Window Appearance 66
Design 67
Interaction principles 69

Section 5 - Operator Interface

Introduction 71
Operator Workplace Interaction 72
Faceplates 72
Display Elements 74
National Language Support (NLS) 75
Interface between Control Builder and Operator Workplace 75
Interaction principles 76
SIL considerations 77
Access Level 77
Support for Confirm Operation Dialog 78
Graphical Indication of ParError 83
The Operator Workplace Graphics 83

Section 6 - Program Code Issues

Program Code 85
Descriptions 86
Variables and Project Constants 87
Object Sub-Structures 88

Protection and Scope 88
Re-use of Code 89
Control Module Types 91
Function Block Types 93
Diagram Types 93
Data Types 93
Templates 94
Task Considerations 94
Parameter Dependency on Tasks and Controllers 95
Calls to Asynchronous Functions 95
Special functions 96
Handling of Input and Output Values 96
Error Handling 97
Alarm and Event Handling 97
Program Stop Complication 101
Power Failure Behavior 102
State algorithms and bumpless parameters changes 102
Range Check 103
Conditional Range Check 105
Overflow handling 107
SIL Mark Restrictions 107

Appendix A - Names and Abbreviations

Suggested Names 109
Recommended Names and Abbreviations 109
Standard Library Parameters 113

INDEX

About This User Manual

General

This manual is primarily a style guide for building object types and data types in application libraries. It describes a concept for enhancing consistency and readability in a library.

The manual is organized in this manner:

- Section 1, Libraries describes briefly the purpose of this style guide and the two main library categories user-defined libraries and support libraries.
- Section 2, Types teaches you the basic design of a given object type, either as a control module type or as a Function Block type.
- Section 3, Parameter Interface discuss a common standard for naming libraries, object types, data types, parameters etc. It also describes the different parameter attributes.
- Section 4, Engineering Interface describes the interface towards the application engineer. It shows the requirement on how Control Modules, Function Blocks, and Diagram Types are designed for usage in the graphical editor environment.
- Section 5, Operator Interface refers to an operator interacting with the system via a graphical interface in online mode. It introduces you to the concept of InteractionPar, a parameter of a type specific structured data type which holds components that can be interacted from a graphic interface.
- Section 6, Program Code Issues provide ideas for efficient programming. It highlights among other things protection attributes, re-use of code for facilitate maintenance, alarm and event handling etc.
- Appendix A, Names and Abbreviations list recommended names and abbreviations for types and parameters.

Document Conventions

Microsoft Windows conventions are normally used for the standard presentation of material when entering text, key sequences, prompts, messages, menu items, screen elements, etc.

Warning, Caution, Information, and Tip Icons

This publication includes **Warning**, **Caution**, and **Information** where appropriate to point out safety related or other important information. It also includes **Tip** to point out useful hints to the reader. The corresponding symbols should be interpreted as follows:

Electrical warning icon indicates the presence of a hazard which could result in *electrical shock.*

Warning icon indicates the presence of a hazard which could result in *personal injury.*

Caution icon indicates important information or warning related to the concept discussed in the text. It might indicate the presence of a hazard which could result in *corruption of software or damage to equipment/property.*

Information icon alerts the reader to pertinent facts and conditions.

Tip icon indicates advice on, for example, how to design your project or how to use a certain function

Although **Warning** hazards are related to personal injury, and **Caution** hazards are associated with equipment or property damage, it should be understood that operation of damaged equipment could, under certain operational conditions, result in degraded process performance leading to personal injury or death. Therefore, **fully comply** with all **Warning** and **Caution** notices.

Terminology

A complete and comprehensive list of Terms is included in the Industrial^{IT} Extended Automation System 800xA, Engineering Concepts instruction (3BDS100972*). The listing included in Engineering Concepts includes terms and definitions as they apply to the 800xA system where the usage is different from commonly accepted industry standard definitions and definitions given in standard dictionaries such as ***Webster's Dictionary of Computer Terms*.**

Related Documentation

A complete list of all documents applicable to the 800xA Industrial^{IT} Extended Automation System is provided in Released User Documents, 3BUA000263*. This document lists applicable Release Notes and User Instructions. It is provided in PDF format and is included on the Release Notes/Documentation media provided with your system. Released User Documents are updated with each release and a new file is provided that contains all user documents applicable for that release with their applicable document number. Whenever a reference to a specific instruction is made, the instruction number is included in the reference.

Section 1 Libraries

Introduction

The libraries used in the control project must have a consistent appearance. The names of libraries, object types, data types, parameters, and variables must be used consistently. In addition to this, there must be well-defined color schemes, and a standard methodology for user interaction on the interface.

This manual is a style guide for developing libraries, data types, Control Module Types (CM) and Function Block types (FB) and Diagram types. The corresponding internal code for CM and FB is created with the programming language - Structured Text (ST). The suggestions regarding FB in Function Block Diagrams (FBD) are also applicable to Ladder Diagrams (LD).

Purpose

The recommendations provided in this manual apply to libraries, objects and data types, and not to hardware types.

This style guide helps in:

- Creating a compact graphic representation of object types, which facilitates efficient use of screen area and paper.
- Using consistent and clear naming of libraries, object types, data types, parameters, and variables, which complies with the existing standards.
- Creating object types with consistent engineering interface.
- Developing a code structure that is easy to understand.

Library Categories

There are two library categories:

- Standard Libraries
- User Defined Libraries

Each of these two categories in turn contain Object Libraries and Support Libraries.

Table 1 shows the relationship between the library categories.

Table 1. Relationship between library categories.

	Object Libraries	Support Libraries
Standard Libraries	ControlStandardLib *PidCC*	ControlSupportLib *AddRangeWithGain*
User Defined Libraries	BUXVesselLib *Tank*	BUXSupportLib *Agitator*

In Table 1, *Agitator* is an object which cannot be used as a stand-alone object in an application. In order to function, it needs to be part of a ready-to-use object, in this case, *Tank*. However, *Agitator* shall also be reused as part of other objects. Therefore, it is more suitable to place objects like *Agitator* in the support library (in this case, BUXSupportLib).

Standard Libraries

The Standard libraries are included as AC 800M system extensions for 800xA. They contain extended functionality designed by ABB. Almost all object types in the standard libraries are protected. This means that these types cannot be modified and cannot be copied to another library.

User Defined Libraries

User defined libraries should be used for customer or solution specific objects. For these libraries, it is recommended to include a prefix, which designates the origin of the library.

For example, the name for the Business Unit X's library can be BUXVesselLib.

Object Libraries

Object libraries contain templates or ready to use objects such as data types, function block types, control module types, and diagram types that can be used in applications created in the Control Builder.

Library Dependencies

The separation of common functions into a separate library makes code reuse possible. The complete functions, algorithms, and user interface objects can be placed in these common libraries.

Connected Libraries

If a library uses types from other libraries, these must be included in the list of Connected Libraries. This creates a dependency chain between the libraries. Several dependency levels can be obtained (for example, for licensing), if the libraries are connected.

A general rule for connecting libraries is that a library shall only depend on relevant libraries. It is also important to avoid circular dependencies.

The reuse of graphical elements in the IndustrialIT 800xA System Graphics does not require the creation of connected libraries.

Split Libraries

In many cases, dividing a library into several libraries has advantages.
Split libraries can be defined as:

- A specific set of libraries where some libraries are common to all applications, and they are complemented by more specialized libraries.
- A specific set of libraries with similar function, but using different external software or hardware. For example, the objects for communication are separated into several libraries according to the protocols used.

Split libraries belong to a family of the particular functionality. The structure of a split library can be flat or tree-like. A tree structure is created using dependencies within this family, and it is recommended that this tree structure is reflected in the naming of the libraries.

For example, consider the control libraries in which the total functionality is split into several libraries. They all have the control functionality in common, but are divided into categories like ControlStandardLib, ControlAdvancedLib, and so on.

Support Libraries

Libraries specially designated for code re-use are called Support Libraries. The content of a Support Library may be hidden, except for the System Graphics, which appears in the Object Type Structure in the Plant Explorer.

The Support Libraries are used by other libraries, in general or within its functionality family, and they are not directly connected to an application.

Naming Convention

The name of a library shall describe its purpose.

The name of a library shall consist of a mix of uppercase and lowercase letters forming the structure "*LibName*Lib". That is, the different parts of the name shall be separated with uppercase letters and the name string shall always end with "Lib". The maximum string length is 32 characters.

Some names are reserved. Examples of names used by the standard libraries are *BasicLib* and *IconLib*.

All libraries share the same name space. Therefore, it is strongly recommended that the user-defined libraries shall have a prefix in their library name. It is also important to avoid too general names.

A split library's name starts with the function family (for example “Control”), then the sub-category (for example “Advanced”), and ends with the suffix “Lib”.

For the support libraries, the naming convention is that the word “Support” should be included between the main part of the name and the *Lib* suffix. For example, for the user-defined library BUX, the support library could be named BUXSupportLib.

Table 2 provides a brief summary of the naming rules.

Table 2. Naming rules for libraries

Library Category	Naming Rule
General	The different parts of the name shall be separated with upper-case letters and the name string shall always end with "Lib". The maximum string length must not exceed 32 characters.
User-defined Libraries	The name string shall have a prefix (for example, the company name or an abbreviation of it).
Support Libraries	The word "Support" shall be included between the main part of the name and the *Lib* suffix.
Split Libraries	A split library's name starts with the function family (for example, "Control"), then the sub-category (for example,"Advanced"), and ends with the *Lib* suffix.

Section 2 Types

The choice of object type depends on the functionality that surrounds the object (that is, the overall plant design). For more information, refer to the *Basic Control Software, Introduction and Configuration* manual and also the *Application Programming, Introduction and Design* manual.

Naming

Object Types

Object types and parameters shall comply with relevant standards and form a consistent name space.

- The standard IEC 61131-3 shall be followed[1].
- The standard IEC 61131-5 shall be followed whenever possible[2].

As for library names, the object type name should start with an uppercase letter and different parts of the name should be separated by capital letters (for example, PidCascadeLoop and FFToCC).

1. IEC 61131 Programming Languages.
2. IEC 61131 Messaging service specification (Communication).

The length of object type names shall not exceed twelve characters, when this is not possible up to 20 characters are allowed. However, the use of short names is not as critical for control module types as for Function Blocks types. The following reasons are:

- Object type names and parameter names are not always shown in the CMD (Control Module Diagram) editor.
- The requirement that graphical representation of a Function Block Diagram shall be as clear and easy as possible to understand calls for short, descriptive and easily understandable names.

Still, when zooming in on the control module diagram, or when showing the parameter list for a control module type, the parameter names are shown, and they should therefore not be unnecessarily long.

The names of public and non-hidden function block types, control module types, and diagram types should clearly describe the actual function of the object type and not be too short and general. For example, Add4Int, not just Add. They may also have a common prefix that designates a group of object types, for example MMSRead, MMSWrite.

Names of control module types shall end with the suffix *M* if a function block type with identical functionality exists. Names of diagram types shall end with the suffix D if a function block type and/or if control module types with identical functionality exists.

Object types intended to be templates; - types that the user has to make a new definition of and rename before actual usage - shall always have the suffix Template in the object type name. For example, the object type name EquipProcedureTemplate.

Aspect objects shall be categorized in such a way that, example, all motors will contain the string Motor, all valves will contain the string Valve, a PID controller will contain the string PID, and so on.

Compound Words and Abbreviations

The following usage order is recommended:

1. Use the full name
2. Use the short name

If it is a compound word, try to use the following rules in the following order:

1. A. Word full name B. Word full name
2. A. Word full name B. Word short name
3. A. Word short name B. Word short name

There are no strict rules on how to build a short name, but the following methods should be considered:

Rule	Example
Use only a part of a whole word.	Acceleration Limiter -> AccelerationLimCC
Remove all vowels (and some consonants).	Square root -> SqrtCC
Use a new word.	Communication Link Read -> COMLIRead

If there is no risk for misunderstanding, different full names can have the same abbreviation. In Appendix A, Names and Abbreviations a list of names and their abbreviations are presented.

The IEC 61131-3 and IEC 61131-5 (the latter only for communication, excluding FOUNDATION Fieldbus) standards and guidelines shall be followed for naming of object types. This is also valid for libraries, parameters, as well as data types.

Note that some words, for example IF, THEN, ELSE (not case sensitive), are keywords recognized by the 61131-3 standard and cannot be used.

Suffixes

A group of object types may also have a common suffix describing the data type they should be applied to, for example Add4Int and Mult4Int. Most object types in the control libraries have the suffix -CC, which indicates that they should be applied to the data type Control Connection.

The following suffixes shall be recognized:

Table 3. Object Name Suffixes

Short Name	Remark
Real, Bool, Dint, CC	Used as suffix on function blocks that have same or similar function, but the data type of the main signal parameter differs. It can also be used to group objects using a main signal of a certain data type. For example: *PidSimpleReal*, *PidSimpleCC*, *SignalInBool*, and *SignalInReal.*
M	A control module type that has the same functionality as an existing function block type shall have the same name as that function block type plus the suffix “M”. For example: *AlarmCond* (Function block type) *AlarmCondM* (Control module type)
D	A diagram type that has the same functionality as an existing function block type or control module type shall have the same name as that function block type or control module plus the suffix "D" (for Diagram type). For Example: RemoteInput Control module type and RemoteInputD Diagram type
Core	Protected core functionality, typically re-used in several different object types. Functionality that may be subject to changes should be placed outside the core. For examples: *EquipProcedureCore*, *UniCore*
Template	Only for object types that the user must change and rename before use. For example: *EquipProcedureTemplate*

Other Naming Issues:

- Several suffixes are allowed but do not use _ (underscore) to separate abbreviations (occupies one position and may be visually unclear).
- Acronyms should be in capital letters, FBD (Function Block Diagram).

Name Space

The object type name should be unique, not only within a library, but among all libraries. Object types with the same name but belonging to different libraries can be accessed via dot notation, for example *MyLib1.MyFunctionBlock* and *MyLib2.MyFunctionBlock*. It is however strongly advised to have unique names, to minimize the risk of calling the wrong object type.

Section 3 Parameter Interface

The parameters of Control Modules and Function Blocks should follow the naming and usage convention described in this section. It will be easier for the application programmer to understand the purpose of the different parameters found in each object's connection list.

Naming Convention

It is important to have a standard for naming libraries, object types, data types, parameters, etc. For example, a control module type or a parameter with a standard name should always represent a specific function. This means that a user can connect the right variables and use the correct control module without knowing the exact function of their respective types.

When designing an object type, an excessive length of the name of one or a few parameters should not be allowed if it increases the total width of the object type graphical symbol considerably. This will be discussed more thoroughly in Section 4, Engineering Interface. In exceptional cases, longer names than specified below are allowed. In general, whenever a rule cannot be applied strictly, common sense should be used.

If names contain a part that is an abbreviation, for example MMS, these abbreviations should be written in capital letters.

Compound Words and Abbreviations

The following usage order is recommended:

1. Use the full name
2. Use the short name

If it is a compound word, try to use the following rules which was described in Compound Words and Abbreviations on page 21.

Names can consist of only one name, for example Enable, but also be composed by several words, InteractionPar. In the latter case it is often necessary to abbreviate one or several of the words into a short form. There are no strict rules on how to build a short name, but the following methods should be considered:

Rule	Example
Use only a part of a whole word.	Request -> Req
Remove all vowels (and some consonants).	Print -> Prt
Use a new word.	Communication Link -> COMLI

Parameter Properties

The parameters of function blocks, control modules, and diagrams have certain properties that should be set. The usage of these will be described in this sub section.

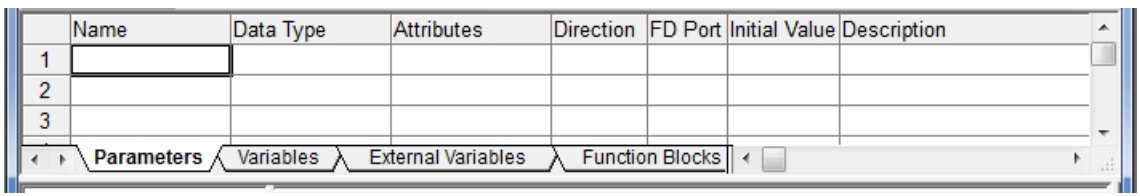


Figure 1. Parameter view in editor - Function Blocks

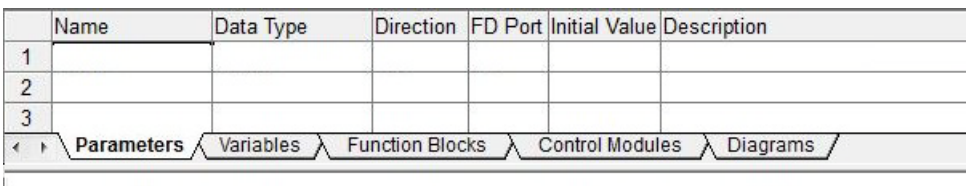


Figure 2. Parameter view in editor - Control Modules and Diagrams

Recommendations for the following properties will be presented:

- Name on page 33
- Data Type on page 27
- Attributes on page 27
- Parameter for Control Modules and Diagrams on page 28
- Parameter for Function Blocks on page 30
- FD Port on page 30
- Initial value on page 31
- Description on page 34

Data Type

It is recommended to avoid structured data types in Function Block Types, since it is not possible to connect the components of a structured data type directly to other function blocks.

If a component of a structured parameter is connected to another Function Block in FBDs, this must be made via a local variable. However, this kind of connections between two function blocks will not appear graphically, even though they are in fact connected.

When using the Diagram editor, a structured parameter component may be connected to an object, using that component data type by directly addressing it as `struct.component`. For example, a variable of type real can be connected to the value component of an object parameter of type RealIO.

Attributes

Control modules and diagrams do not have attributes on their parameters, (only on variables) since no local copy of the connected variable is made.

For function blocks the entries of the attribute column can be group into three categories:

- retain or cold retain
- nosort

- hidden
- by_ref
- constant
- Reverse attribute on data types in diagrams
- Display value attribute on data types in diagrams

Retain shall be used for In and Out parameters in Function Block Types. Whereas ColdRetain is the normal attribute for operator-settings parameters, such as operating time measurements, and parameters for operator interaction. Note that the (cold)retain handling does not support extensible parameters. When declaring a data type, it should be designed such that it is unaffected if the connected variable of an instance is declared coldretain/retain by the user.

The *nosort* attribute can be assigned to a parameter, but this is rarely used.

Parameters that should not be available via the OPC-server, must be given the attribute *hidden*.

The *by_ref* attribute can only be set for function block parameters with direction In or Out. This attribute specifies that the parameter value will be passed by reference instead of the value.

The attribute *Constant* can be used only for variables.

The reverse attribute works for control modules in diagrams. Split and Join functions cannot to be used with structured data types containing components with the reverse attribute. Functions and function blocks cannot handle reverse attributes. For diagrams there is also a attribute Displayvalue for datatype. If this attribute is set for a component in a datatype, the value will be shown in online mode for the connection.

Parameter for Control Modules and Diagrams

In control module and diagram types, a parameter can have the direction In, Out, In_Out or Unspecified. All of them are passed by reference, which means only a reference to the actual variable outside the control module is passed to and from the control module.

The differences between the four types include the different access rules from the code inside the control module and the limitations for connecting the parameters:

- *In* parameters can only be read, where as *Out*, *In_Out* and *Unspecified* parameters can be both read and written.
- The *In* parameter of a sub control module can be connected to either the *In* parameter or the *Out* parameter in the surrounding control module. Between control modules on the same level, it is only allowed to connect *In* to *Out*. Several *In* parameters can be connected to one *Out* (if it is not a structured type containing a reverse attribute). *In_Out* must be connected to a variable (on any level).
  These rules also apply to connecting parameters to communication variables. The *in* communication variable should be connected to *In* parameters and the *out* communication variable should be connected to *Out* parameters only.
- *Unspecified* parameters can be used without limitations when connected to other parameters with unspecified direction for compatibility reasons. However unspecified parameters cannot be connected to parameters with direction and vice versa. The *unspecified* parameters are not applicable for diagram type instances.

The direction *Unspecified* must not be used for parameters in a new control module type in library or a sub level control module.

The direction should be noted as a keyword in the description for each parameter. See also Type Description Keyword on page 32.

Parameter for Function Blocks

In function block types, a parameter can have the direction *In*, *Out* or *In_Out*. Use the direction In or Out whenever possible. Parameters of direction *In_Out* must be connected for function blocks.

The In_Out direction should be used only when specifically motivated (for example forcing). Ample use of In_Out may produce confusing FBDs. Note that such a parameter will be marked written for a Function Block, even for the components that are only read. This may result in code loops during compilation for structured data types, which can be avoided by setting the variable's attribute to nosort. If the data type AnyType is used, the direction must be *In_Out*.

FD Port

The FD Port column is valid for parameters in function block types, control module types, and diagram types. It is only significant for the types that are instantiated in a Diagram editor.

The normal choice is *Yes* or *No*. The value specifies if the parameter shall be visible when the function block type, control module type or diagram type is instantiated in an FD code block. The configured value represents the default, which can be changed on the instances.

Parameters with direction *in* are placed on the left side, and parameters with direction *out* are placed on the right side. Parameters with direction *In_out* can be placed on the left side, if the keyword *left* is added; else it is placed on both sides. Similarly, parameters with direction *unspecified* can be placed on the left, right, or on both sides. The *left* or *right* specification cannot be changed on the instance.

Parameters of bi-directional data types with no reverse defined components must be either an Unspecified parameter (control module types only) or an In_Out parameter.

The following list summarizes the use of the values in the FD Port column for control module parameters with direction Unspecified and function block parameters with direction In_Out:

- *No* - Not visible as a port.
- *No Left* - Not visible as a port. The parameter will be placed on the left side of the object if the user decides to show it later on.

- *No Right* - Not visible as a port. The parameter will be placed on the right side of the object if the user decides to show it later on.
- *Yes* - Visible as a port on the left side of the object.
- *Yes Left* - Visible as a port on the left side of the object.
- *Yes Right* - Visible as a port on the right side of the object.

Initial value

The initial value column for a parameter can be: empty, a value, or the value default. The value default is only applicable to Control Modules and Diagrams.

Parameters with type description IN or IN(OUT) can have all three alternatives; empty, a value, or the value default. For parameters with type description OUT and OUT(IN) either the empty field alternative, or the value default, is used. An empty initial value of a parameter of direction In_Out forces the application engineer to connect a variable to it, and the parameter then receives this parameter's initial value. Parameters marked default receives the initial value of its type.

For simple data type, it is advisable to use default only for *out* parameters. For structured data types it may be used both for *in* and *out*. For *in* parameters of simple data types it is advisable to instead use the intended default value example, false for a bool parameter.

The initial value of an instance can be set using the Control Properties aspect in the Control Structure of the Plant Explorer Workplace. This is, for example, useful for InteractionPar.

Description

Parameters in object types should have a line of text briefly describing its purpose/function. If a short name has been used for the parameter, use the description field to explain it with full names. For example, the description field for the parameter PrtAckAlarms may look like “Prints acknowledged alarms”. A real or integer input parameter may have a range. When the parameter value is out-of-range the parameter shall be assigned a specific value or the last good value.

Type Description Keyword

The description field can hold a keyword before the actual description. This keyword holds information about its usage for the application engineer. All parameters in Control Module types, Diagram types, and IN_OUT parameters should have at least one of the first four comments in Table 4.

Table 4. Type Description Keywords

Keyword	Description
IN	The parameter is only read.
OUT	The parameter is only written.
IN(OUT)	The parameter is both read and written, but mostly read.
OUT(IN)	The parameter is both read and written, but mostly written.
NODE	Used when the parameter has a graphical connection node (control modules only).
EDIT	The value of the parameter is used the first execution after transition from Edit to Run mode without initialization. Cannot be changed online.
NONSIL	The output parameter marked with NONSIL originates from a restricted marked sub-object and is not allowed to be used in the critical loop. If the object is set to SIL1-2 or SIL3, but the parameters are Non-SIL, then it is possible to obtain partial functionality in the SIL environment.
DEFAULT	The output parameters on SIL marked object where the information originates from restricted sub objects.

For Function Block Types parameters only the comments related to EDIT and NONSIL are applicable.

For control modules, it is very important that the IN and OUT markings of parameters are correct. The application programmer needs this information to avoid program loops and to know which parameters to write to and read from, respectively.

The parameters are shown as IN, OUT, or INOUT ports in Function Designer dependent of the first characters in the parameter description:

Parameter description starts with “IN” --> IN port

Parameter description starts with “OUT” --> OUT port

Parameter description starts with any other string --> INOUT port

Changing the first characters containing keywords of the parameter description should be considered an incompatible change. If the port direction is changed, e.g. from INOUT to IN, then the parameter must be **reconnected** in the function designer applications after the changed instances are downloaded.

Range Checking Description

A real, integer, or word input parameter may have a range. In that case the parameter description shall also state the range, the action for valid input values, and the action for out-of-range values (see description field in Table 5). This is mandatory for objects that are to be SIL classified.

Table 5. Example of description for range checking in a Function Block Type

Name	Data type	Attributes	Direction	Initial value	Description
AEConfig	dint	retain	IN	1	Config (0=None, 1=Alarm, 2=Event, 3=Event1, 4=Indication, Else Alarm + ParErr)

Parameters with Special Purposes

This sub section describes the most frequent parameters and their usage.

Name

The Name parameter is a string of 30 characters, holding the instance name. The Name parameter is used for several purposes:

- When performing a Name Upload in the Control Structure (Plant Explorer), the instances' Name parameter is passed to the corresponding Name aspects.

- If the object is an alarm owner, its Name should be used as source name when alarms are generated, and must therefore be unique. The Name parameter will then appear in the alarm list as the Source Name of an alarm. In such cases the Name parameter of the type should not have any initial value to force the application engineer to enter a name.
- If an interaction window is opened while the Control Builder is in on-line mode, the Name parameter is displayed in the window title bar. See Figure 3.
- Control module types having graphical connections and a Name parameter, should have the name visible in layer 1 together with the node names, Figure 5 in Point-to-Point Connection using ControlConnection Data Type on page 38.

Description

The Description parameter is a string of 40 characters, and is a short description of the object instance.

- When performing a Name Upload in the Control Structure of the Plant Explorer Workplace, the instance's Description parameter is passed to the Name aspect's description field.
- If the object generates alarms the Name aspect's description, which is the instance's Description parameter after a Name Upload, will appear as the Object Description in the alarm list.
- The Description parameter is displayed in some operator graphics windows for clarifying purposes.

The example object in Figure 3, is part of a standard PID controller with the Name parameter displayed in the Windows title bar and the Description parameter displayed in the control builder faceplate.

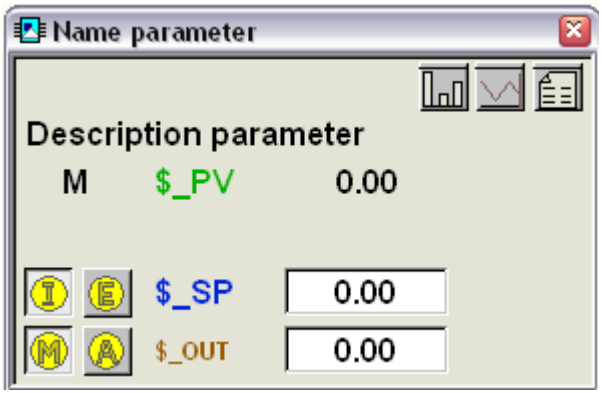


Figure 3. Example of usage and location of the Name and the Description parameter.

InteractionPar

The InteractionPar parameter is a structured data type where the components are all the parameters that can be changed via interaction objects. Note that it is only values that are written to/from the interaction objects that should be placed in this parameter. By putting these items in a parameter it is still possible to access them from application code. If it is judged that accessing the parameter from code is equally or more probable than from the interaction object, there should be both a component in InteractionPar and an additional parameter. However, this will require logic inside the type in order to handle the priority between the InteractionPar component and the parameter.

The interaction parameter data type shall be named by concatenating the object type name, where the parameter is used, with the word Par, ObjectNamePar. The components for configuration typically have the attribute ColdRetain and mode, state components have Retain attribute, whereas components for operator commands shall not have retaining attribute.

When creating a new object containing several sub-objects, which have their own InteractionPar, the new object's InteractionPar should have structured components. These components should be named according to the sub-objects functional purpose in the new object, to make its usage intuitive for an application engineer. Consider the case of a Control Module having one analog input and one analog output module, with some treatment in between. The compound object's InteractionPar could then look like Table 6.

Table 6. Structured components

Parameter name	Type	Component name	Type
InteractionPar	MyObjectPar	InputDevice	AnalogInputPar
		....	....
		OutputDevice	AnalogOutputPar

ParError

The ParError parameter is set true if any input parameter is out of range. This type of check is mandatory for SIL classified object. For non-SIL objects or SIL marked objects running in a non-SIL environment, the EnableParError parameter may be used. A real or integer input parameter may have a range. This means that it may be required to have a relation (for example, > or <) to one or several constants or other parameters.

The parameter description shall state the range, the action for valid input values, and the action for out-of-range values.

For SIL applications, the ParError is intended to be connected to Errorhandler(M).

Exception: A ParError parameter (or other parameters) shall not be added to types where the parameter interface is defined by IEC 61131-3, for example TOn.

Connections

A special sort of data types are the ones representing connections. These are used to connect control modules, function blocks, and diagrams. A connection carries information in more than one direction, why the structured data types are used. The naming convention for the data type is that they have the suffix Connection.

Main Signal Flow

The main signal flow of an object should have a name stating its purpose, like

- In, In1, In2, … - main input signals, if there are no other natural names.
- Out, Out1, … - main output signals, if there are no other natural names.
- Pv - process value signal, used for example for process feedback in control loops.
- Sp - set point signal, used in for example controllers.

This type of parameters do not need to reflect the data type used. The direction *in* is used for input signals and *out* for output signals. This kind of connections have the attribute *reverse* for the values that are sent backwards.

Point-to-Point Connection using ControlConnection Data Type

A natural way to represent the main signal flow of Control Modules is by structured connections. In this way a single connection, which preferably is done graphically, handles the main signal flow between two adjacent objects.

Bidirectional connections should have components for each direction. The substructures are preferably named Forward and Backward, but other names may be used if they better fit the context they are used in, for example Previous and Next, Upstream and Downstream, etc. This makes it easier for the programmer to handle communication between objects. It also supports data communication over task and system borders, since it is clear what components are sent in each direction.

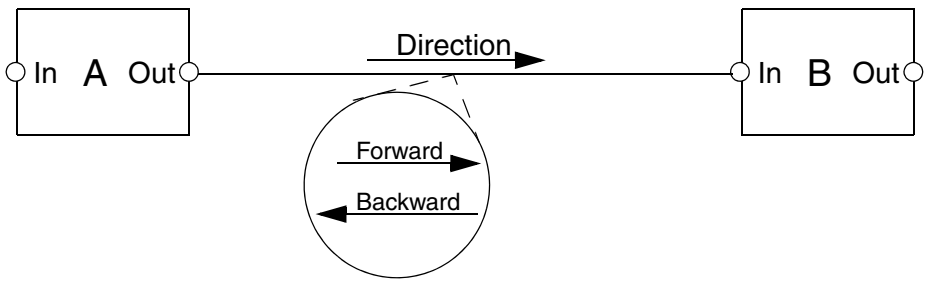


Figure 4. Point-to-point connection between two objects. The node names reflect the main direction of the signal. The blow-up shows in principle data flow.

In each of the directional substructures, for example Forward and Backward, a component Connected should exist. All objects shall set the Connected component true in the direction-component they are allowed writing to. The Connected component can then be checked by the object on the other side and determine if there is a connected object or not. This is useful when an object connection is optional, thus may not be connected.

Point-to-Point Connection Using Reverse Attribute

In bi-directional data types, the components used to transfer data in backward direction must have the *reverse* attribute. Otherwise the data type cannot be used for parameters with direction defined.

The *reverse* components must be grouped in consecutive order at the end of the data type. They can also be put in a sub structure at the end of the data type.

The *reverse* attribute can be used for components in the data types used in communication variables for point-to-point communication. If there is any reverse component in a communication variable, it is communicated from *out* backwards to *in* (opposite direction to the normal data flow). This also implies that it is only allowed to communicate one-to-one (an *out* communication variable can be connected to only one *in* communication variable if the data type with the *reverse* component is used).

Handling Communication Failure using ISP Value

The data type editor contains the column to define an ISP value to each component. These ISP values will be used in case of communication failure.

The rest of the components, which do not have any ISP values, will keep the existing value.

For safe communication using IAC in a SIL application ISP value must be defined for all components in a data type and for safe MMS it is on the user responsibility to arrange for safe values if the communication is broken.

Parameters for Alarm Handling

The alarm handling in object should be unified with possibility to assign class, severity, condition name, alarm and event configuration, as well as possibility to inhibit and disable alarms. Therefore, a number of parameters defined in Table 13 on page 98, have been reserved for this purpose.

Monitoring Continuous Execution

The parameters used for control of continuous functionality (for example, level monitoring) are described in Table 7. All of these parameters need not be defined at all times. If *Error* or *Warning* is defined, *Status* must also exist. If *Error* and *Warning* occur simultaneously, the error status code shall be given in *Status*.

For function block types, *Error* and *Warning* only last for one invocation. For control module types, Error and Warning are active for as long as the invocation lasts.

Add a suffix (for example, EnableReport, ErrorReport) in case of several activation signals.

Table 7. Parameters for continuous functionality in FB and CM

Parameter Name	Type	Direction	Comment
Enable	bool	In	Activates/deactivates continuous functionality.
Valid	bool	Out	Indicates that there is no error status and that the function is active. Warning status does not affect Valid.
Enabled	bool	Out	Indicates that the function is activated. This is not affected by error status or warning status.
Error	bool	Out	Indicates error (Status <0).
Warning	bool	Out	Indicates warning (Status >1).
Status	dint	Out	Indicates Status code.

Section 4 Engineering Interface

The engineering interface is the front-end towards the application engineer. In a library there are Control Module (CM), Function Block (FB) and Diagram (D) types that are ready to use, but also types which are intended to be modified before usage, so called templates.

The programming method for building an application is either text based or graphical. There are currently two graphical interfaces available for editing in Control Builder:

- The Diagram (D) editor
- The Control Module Diagram (CMD) editor.

The text based editor is called the POU editor or just editor. The POU editor is used to create code blocks inside the program, inside the control modules or inside the diagrams. This editor supports different programming languages:

Language name	Description
ST	Structured text. This language is pure text based.
IL	Instruction list. This language is pure text based.
FBD	Function block diagram that in fact is a kind of graphical editor for the FBD language.
LD	Ladder diagrams. This language has also a graphical visualization of the code.

Language name	Description
SFC	Sequential Flow Chart. In steps and transitions is ST used.
FD code block	This is used in the single graphical code block that only exist in the first tab of a diagram. Other code blocks in the diagram may be of any language.

The subjects on how Control Modules and Function Blocks are designed for usage in the graphical editor environments will be handled in this chapter.

The diagram editor is also able to visualize diagrams created in the functional designer environment but the code cannot be modified in the diagram editor.

General

Protection Attribute on Types

Function Block types, Control Module types, and Diagram types can be divided into three different categories:

- Function block types/control module types meant to be templates (have suffix “template”, must be changed by the user to function). These kinds are not possible to execute without any changes; Protection attribute has to be set to False.
- Function block types/control module types that could be changed by the user, these kinds are possible to execute without any changes; Protection attribute has to be set to False.
- Function block types/control module types that never should be changed or modified by the user must prevent copying of code; Protection attribute has to be set to True.

Template Design

Templates can be divided into two categories; types that can be edited and types that must be edited. To enable code editing in a template the application engineer must first make a copy of the type.

Templates enable the application engineer to customize a common solution. For example a motor may have slightly different properties depending on usage and/or model. By first copying a template the overall functionality is obtained and solution specific alteration can be made, and still preserving a common operator interface recognized by the user.

When you copy a SIL marked object template, it will be Non-SIL after pasting or during the copying process. The SIL level has to be set by the user manually on the new object type, if required.

Control Module Design

Programming with control modules is preferably done graphically in the Control Module Diagram (CMD) editor. The graphical connections between the modules give a good overview of the signal flow in the application.

This subsection will handle Control Module design requirements for usage in the Control Module Diagram editor. The graphic layers, the grid, and the coordinate system all affect how a control module behaves, appears, and is positioned on the screen, and therefore the major issues of concern. It is not always possible to build a control module type exactly according to these guidelines.

Graphical Layers

A control module type with basic functionality should typically be square-shaped, have two layers, and be zoomable. In normal cases layer 1 of a control module will be visible in the CMD editor. If the module is zoomed in, layer 2 will be visible instead of layer 1, at a certain zoom level. If the module is zoomed out, then layer 1 will become visible again. This shift point is normally 0.95, where 1.0 means the zoom level where the module fills the whole screen, or more exactly, one side of the module is equally long as the shortest side of the screen.

If the user wants to include other sub control module types, they should be placed in layer 2.

Icons shall be put in layer 1, and the instance name of the icon type should be Icon. When the “surrounding” control module type lacks graphical connections, the icon should not be zoomable. This means the icon inherits the zooming properties of layer 1, that is, it disappears at a zoom level of 0.95 (which means that the type covers more than 95% of the whole screen).

If graphical connections exist in the surrounding type, the icon instance should be zoomable and MaxSize of the icon instance set to 0.85. If such a type is zoomed in on, the icon is visible until the zoom level 0.85 is reached. Between zoom level 0.85 and 0.95, the graphical connections and the node names are visible. If the type is zoomed in on even more, layer 2 becomes visible.

If the control module type contains an interaction window in layer 2, there should be a non-visible toggle window interaction object in layer 1. The interaction object should be somewhat smaller than the control module type itself.

If the control module type has graphical connections and a Name parameter, the name should be in layer 1 to be visible together with the node names.

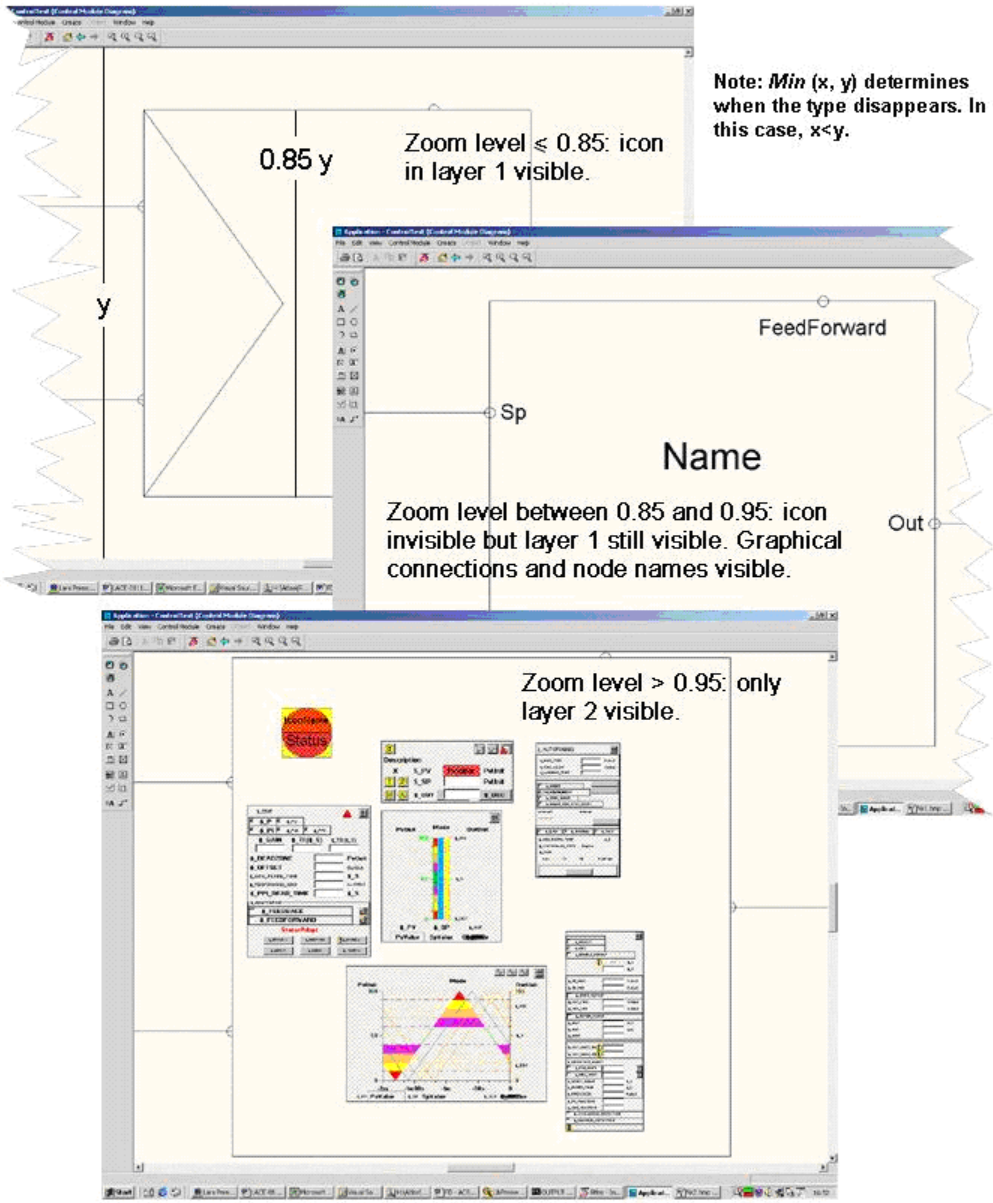


Figure 5. Recommended visibility of layers and icon at different zoom levels.

Grid and Coordinate System

A basic control module type should typically have four connection nodes and an internal coordinate system extending from (-1,-1) to (1,1). When working with the graphics inside control module type, it is advisable to check Show Grid. The default grid size of 0.02 should normally be used, but other sizes may be used as well.

Do not reshape instances of sub control module types because it will then have another shape in an operator window than in the main window. Use the Reset Shape command to ensure that all sub control module instances have their original shape.

Reshaping Issues

Normally, a control module type should not be subject to reshaping. In this case, make the module square-shaped and place the origin in the center. Place the corners at integer coordinate points in the internal coordinate system - also when a square shape is not possible - and use the module all the way out to its limits.

Note that it is possible to put graphical nodes on the clipping border. It is then easy to combine it graphically with other modules. The reason is that all of its four corners always will appear on grid points in the external grid. This method is valid for control module subject to reshaping as well, with the exception that the origin should be placed in the lower left corner instead. In icons, the lower left corner should be placed at the origin and the upper right corner at (1,1).

Graphical Connection Nodes

If a control module type has graphical nodes, it is desirable to let its instances appear on external grid points. Then, it will be easier to obtain good looking connection schemes.

The origin of the type (in the internal coordinate system) will in the instance always appear on a grid point of the external coordinate system at creation. Therefore, nodes that are put on the clipping border where the origin axes cross will always appear on the grid in the external coordinate system.

Nodes will appear on grid in the external coordinate system if the nodes are at the origin axes in the internal coordinate system.

If four nodes are not enough, it is advisable to use the corners. All the corners should be at integer coordinate points.

Layers and Interaction Windows

Control modules in layer 2

When a control module contains other modules, so called sub modules, it is recommended that they are placed in layer 2. Zooming in on the module will reveal the sub-modules, which gives a good overview and also fast access to the modules.

There are mainly three categories of sub-modules which are placed in layer two:

1. Interaction windows - used for operator interaction with the object. In this way the user gets fast access and overview of the windows belonging to the object.
2. Internal code structure - some internal control modules may be related to the connections of a module. By placing the control module close to its corresponding connection the user/developer gets a fast access to the correct module.
3. Applications - used for structuring application and/or reuse of application solutions. This gives a good overview of any applications residing inside the object, for example a control loop or a section of a process.

It is recommended that control modules representing interaction windows are visible in layer 2 of a control module. To make them visible, either create them in the CMD or set the property Visibility in Graphics to Visible on the object in the project explorer structure.

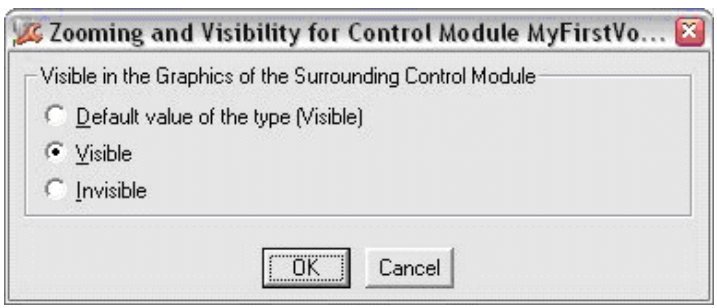


Figure 6. Visible dialog

A control module type that represents the interaction window should contain a rectangle with the following properties:

- Line color: -3 (Foreground color)
- Area color: -1 (Transparent)

This means that the area in the window will have window background color. It also means that the window border is visible in layer 2 of the control module.

Interaction object in layer 1

A non-visible interaction object should be somewhat smaller than the icon representing it. The reason is that the frame that indicates interaction should be clearly visible.

Texts

Text boxes should be used instead of vertical text lines to display variables. Otherwise the text may overwrite other graphical objects. The text box should be made wide enough for the text size to be determined by the height of the text box. It is then easier to combine it with text in other text boxes.

Texts that are too long will cause font reduction for text boxes. For text lines the font will remain constant, which means that the text will not fit in the operator window, which is worse than font reduction. Therefore text boxes should be used instead of text lines for presentation of variables in operator windows.

Even though it is possible for an operator to zoom the module, the operator must be able to control it without using zoom. The zoom facility is only intended for programmers and service personnel.

Icons

Icons are used to distinguish objects, for example control modules and interaction buttons, and to indicate their functionality.

For display elements icons and display element reduced icons, which are used when building graphics displays for operators.

Icons in Control Software for AC 800M

Control Software for AC 800M recognizes two kinds of icons: the basic icons of IconLib and the control module type icons. The basic icons are typically used for building up the latter, which may contain dynamics. For example, if the control module type is inhibited, a cross should cover the icon, or if the control module type malfunctions, an ErrorIcon should be visible in the control module type icon.

Another kind of dynamic icon indication is used when a number of control module types are connected in a chain and dynamic information is transferred backwards in the connected chain.

Suppose one control module type disregards the information from the previous ones. Then information about this may be sent backwards and indicated in the icons of the previous control module types. In this case, a dimmed pattern should be used as a background of the icon and all operator windows. Example: The control module types for analog control loops.

The Difference between basic Icons and Control Module Type Icons

The icons in IconLib (basic icons) do not have a background nor a border. The icons are transparent apart from the actual graphical objects in the icon. The reason for this is that it should be possible to build control module type icons by using several IconLib icons that may partly cover each other. It is therefore necessary to have a transparent background in the IconLib icons (see also Graphical Layers on page 43).

The control module type icons should not be transparent; they should have both a background and a border (important if modules are partly placed on top of each other). Therefore, the icons should contain a rectangle that has the same size as the icon, with the following colors:

- Line color: -3 (Foreground color)
- Area color: -2 (Background color)

Control Module Types for Control Module Icons

Normally the control module type icon should be a single control module. If the icon is used in several types in the library, the icon should be a private control module type.

If the icon is in a template type, the icon must be a public control module type. The name of the type should be NameIcon. For all icons, the control module instance name should be Icon.

Size of the ErrorIcon

If the symbol in the module icon is not large the ErrorIcon should be in the background of the module, and cover almost the whole icon. The ErrorIcon should be put behind the symbol.

If the symbol in the icon is large, the ErrorIcon should be smaller and put where it can be clearly visible.

Icons in IconLib

IconLib contains a number of general icons (control module types) that are common to all libraries. They have transparent background and frame because it is possible to combine them by putting one on top of the other. Another reason is that a red ErrorIcon should be visible in the control module type icon in case of error, and must not be covered by the other “sub icons”. The interior of a closed symbol in an icon should, however, not be transparent.

It is important that these icons are used in the same way in all the libraries. The following example categories describe how these icons should be used:

- Maneuver icons - The maneuver icons should be used together with interaction objects for maneuvering process objects or machines. The icons look like the actual buttons that can be found on a machine, and the result of clicking on this type of icon should give the same result as pressing a button on the real process object or machine.
- Mode icons - The mode icons should be used for presentation of the current state or mode of a process object. For example, a process object that is in manual mode can be indicated by the ManModeIcon.
- Command icons - A command icon should be used together with an interaction object. If the operator clicks on this type of icon, some function should be activated.

Note that the graphics of the icons may be changed in the future. The meaning of the icon should, however, not be changed.

Function Block Design

When programming using function block diagrams the layout of a function block is important. The layout of a function block is determined by the order of the parameters, the length of their name, and their direction.

The parameters of a function block type can be divided into groups reflecting their usage, for example alarm handling, or group start handling. The parameters within a group are order according to their direction as follows:

1. IN_OUT
2. IN
3. OUT

This order is also reflected in the layout of the function block since parameters with In direction appears on the left side of the function block, and those with Out direction on the right. Parameters with In_Out direction has a connection on both sides of the function block, see Figure 7. This is relevant only to the FBD programming language.

For all functions or function block types in the diagram editor it is recommended to always connect 'true' to the EN port. For SIL1-2 and SIL 3 diagrams it is mandatory to do so.

Parameter Names

The parameter names of the function blocks are visible in the function block diagram and control diagram editors, and thus affect the element size. It is therefore recommended that the names are as short as possible. The same applies to control module parameters in the control diagram editor.

Short names of Function Block types and parameters are important with respect to how many Function Block symbols that will fit on a screen or printed page in the Function Block Diagram (FBD) language. In FBD, the possibility to simultaneously see many symbols and their connections is essential for the easy reading of the logic.

Unnecessary paging (both on screen and print-out) has a most awkward effect on reading ease and requires more space for page references. Also note that using upper and lower case letters improves the readability of names, for example

ManMode is better than MANMODE. The POU editors allow use of upper and lower case letters for declaration of parameter names, and that the user refers to the name in any form, as long as the letters are the same (for example entering ManMode is the same as entering MANMODE).

A short name is more space efficient to use than a long name and easier to read. This assumes, of course, that the user knows what the name means. Standardized short names or acronyms are most helpful in this respect, for example Pv = Process Value, and T = time. It should also be kept in mind that a long name not necessarily provides more - or enough - information. Hence, a shorter name together with a good description proves often to be the best alternative.

In addition, in the editor, it is often possible to show the parameter description adjacent to the parameter name for more clarity. Seldom used or unusual parameter names may require longer names to be understandable (for example SourceSuffix), compared to traditionally used names (for example Min).

With the exception of InteractionPar (see further Section 5, Operator Interface), the length of parameter names in functions and Function Block types shall not exceed eight characters. When this is not possible, up to twelve characters are allowed. These restrictions apply equally to graphically connected parameters in control module types. Other parameter names in control module types shall be as short as possible, and easy to understand. Note, however, that for custom designed object types, the user may assign longer names to parameters, up to the system limit of 32 characters.

Avoid structured data types in Function Block types because these are not suitable in Functions Block Diagrams, since it is not possible to connect the components to other function blocks. The parameters in the connection list should be divided into a number of groups. The grouping of the parameters should be done in an object oriented way. This means for example that Min, Max and Unit of a signal should be grouped together with the signal itself.

Function Block Example

The declared parameter direction will affect the graphical layout of the function block. Each parameter will appear in the sequence it has been declared, whereas parameters of IN direction will appear on the left side, and OUT parameters on the right side in the FBD language.

An IN_OUT parameter has connections on both side of the function block, see Figure 7. A graphical consequence of this is that parameters of the direction IN and OUT will be grouped, between IN_OUT parameters, see Table 8.

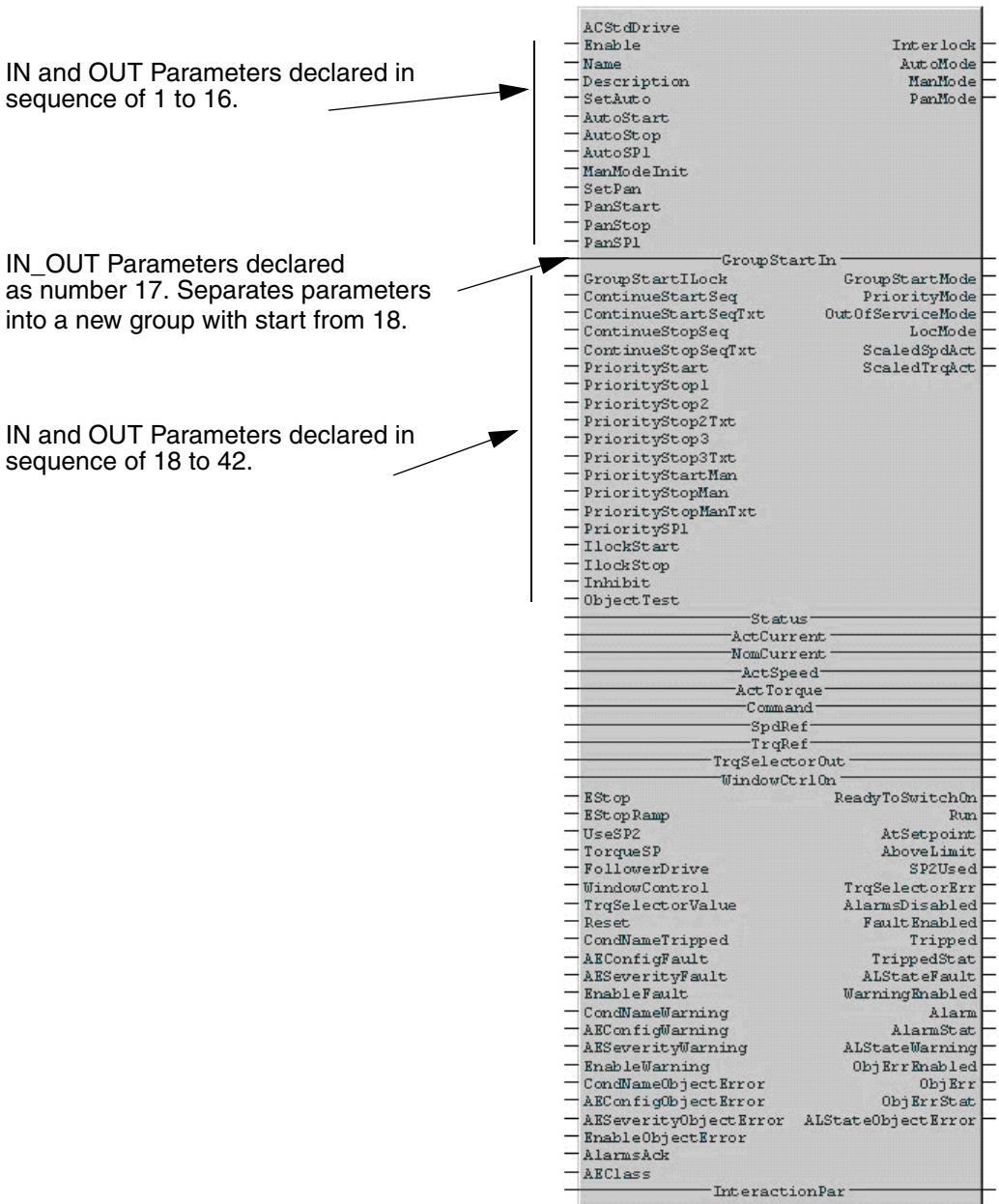


Figure 7. A graphical consequence for declared parameters.

Table 8. Parameter list

	Name	Data Type	Direction		Name	Data Type	Direction
1	Enable	bool	in	24	PriorityMode	bool	out
2	Name	string[30]	in	25	PriorityStart	bool	in
3	Description	string[40]	in	26	PriorityStop1	bool	in
4	Interlock	bool	out	27	PriorityStop2	bool	in
5	SetAuto	bool	in	28	PriorityStop2Txt	string	in
6	AutoMode	bool	out	29	PriorityStop3	bool	in
7	AutoStart	bool	in	30	PriorityStop3Txt	string	in
8	AutoStop	bool	in	31	PriorityStartMan	bool	in
9	AutoSP1	real	in	32	PriorityStopMan	bool	in
10	ManMode	bool	out	33	PriorityStopManTxt	string	in
11	ManModeInit	bool	in	34	PrioritySP1	real	in
12	PanMode	bool	out	35	OutOfServiceMode	bool	out
13	SetPan	bool	in	36	LocMode	bool	out
14	PanStart	bool	in	37	IlockStart	bool	in
15	PanStop	bool	in	38	IlockStop	bool	in
16	PanSP1	real	in	39	Inhibit	bool	in
17	GroupStartIn	GroupStartStep Connection	in_out	40	ObjectTest	bool	in
18	GroupStartMode	bool	out	41	ScaledSpdAct	real	out
19	GroupStartILock	bool	in	42	ScaledTrqAct	real	out
20	ContinueStartSeq	bool	in	43	Status	DintIO	in_out
21	ContinueStartSeqTxt	string[40]	in	44	ActCurrent	DintIO	in_out
22	ContinueStopSeq	bool	in	45	NomCurrent	DintIO	in_out
23	ContinueStopSeqTxt	string[40]	in	46	ActSpeed	DintIO	in_out

Table 8. Parameter list (Continued)

	Name	Data Type	Direction		Name	Data Type	Direction
47	ActTorque	DintIO	in_out	71	TrippedStat	bool	out
48	Command	DintIO	in_out	72	ALStateFault	dint	out
49	SpdRef	DintIO	in_out	73	AEConfigFault	dint	in
50	TrqRef	DintIO	in_out	74	AESeverityFault	dint	in
51	TrqSelectorOut	DintIO	in_out	75	EnableFault	bool	in
52	WindowCtrlOn	DintIO	in_out	76	CondNameWarning	string[15]	in
53	EStop	bool	in	77	WarningEnabled	bool	out
54	EStopRamp	bool	in	78	Alarm	bool	out
55	UseSP2	bool	in	79	AlarmStat	bool	out
56	ReadyToSwitchOn	bool	out	80	ALStateWarning	dint	out
57	Run	bool	out	81	AEConfigWarning	dint	in
58	AtSetpoint	bool	out	82	AESeverityWarning	dint	in
59	AboveLimit	bool	out	83	EnableWarning	bool	in
60	SP2Used	bool	out	84	CondNameObjectError	string[15]	in
61	TorqueSP	real	in	85	ObjErrEnabled	bool	out
62	FollowerDrive	bool	in	86	ObjErr	bool	out
63	WindowControl	bool	in	87	ObjErrStat	bool	out
64	TrqSelectorValue	int	in	88	ALStateObjectError	dint	out
65	TrqSelectorErr	bool	out	89	AEConfigObjectError	dint	in
66	Reset	bool	in	90	AESeverityObjectError	dint	in
67	AlarmsDisabled	bool	out	91	EnableObjectError	bool	in
68	CondNameTripped	string[15]	in	92	AlarmsAck	bool	in
69	FaultEnabled	bool	out	93	AEClass	dint	in
70	Tripped	bool	out	94	InteractionPar	ACStdDrivePar	in_out

Diagram Design

Using the Diagram editor allows mixing of functions, function blocks, diagram instances and control modules in the same diagram. The diagram is based on manual layout. This means the user can freely position blocks on the page.

The graphical connections are drawn using auto-routing. When two ports are graphically connected the line will automatically be drawn in a free area, so that blocks and labels are not crossed. When objects are moved the affected graphical connections will be re-drawn.

In a single diagram it is possible to declare and use variables and communication variables. In a diagram type it is possible to declare and use parameters and variables.

When invoking functions or function block types in the diagram editor it is recommended to not include the EN port, this ensures that the EN is set to true and that the functions or function block types will be executed.

The diagram can have up to 100 pages.

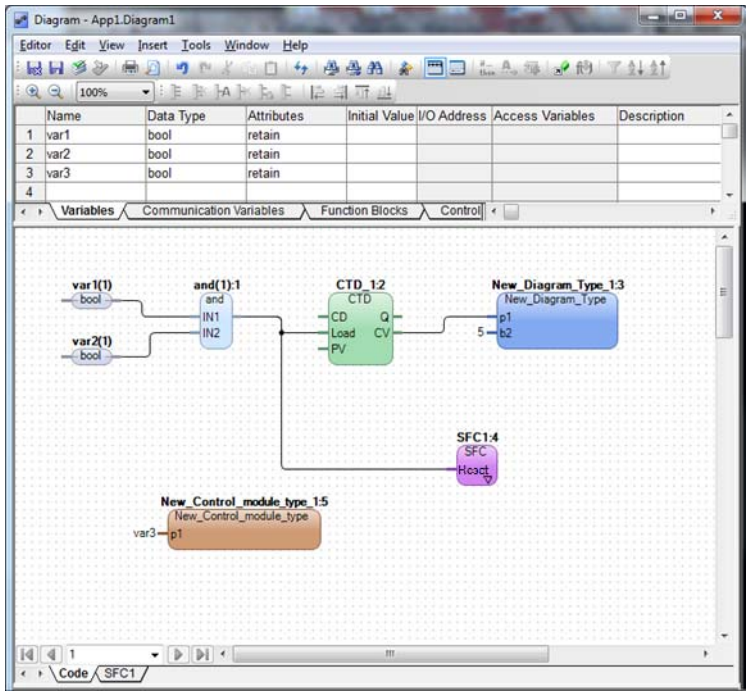


Figure 8. Diagram Example Containing Different Invocations

Data Flow Order

In the diagram all invocations are sorted in to a data flow order. The data flow order of the invocations are decided by the connections which depends on the data flow order they are connected to each other.

If it cannot be determined by the connection in what data flow order the invocations shall be sorted, for example if an invocation has two output that are separately connected to different invocations, it is the invocation closest to upper left corner in the diagram that will be sorted first in the data flow order.

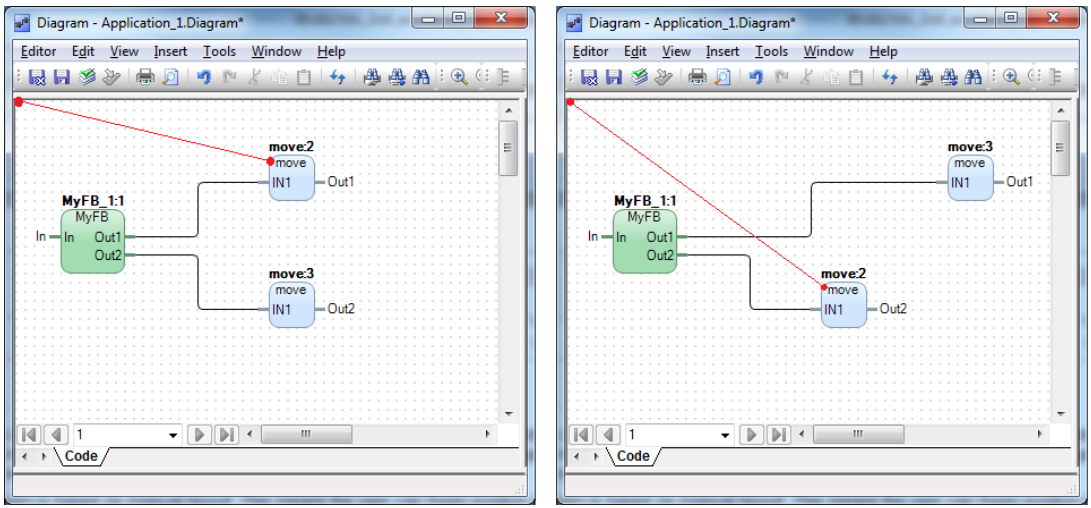


Figure 9. Flow Order Determined by Graphical Placement

Execution order

For control modules the execution order is determined by the normal code sorting. The numeric execution order indication (after the instance name) indicates the execution order of the forward code blocks.

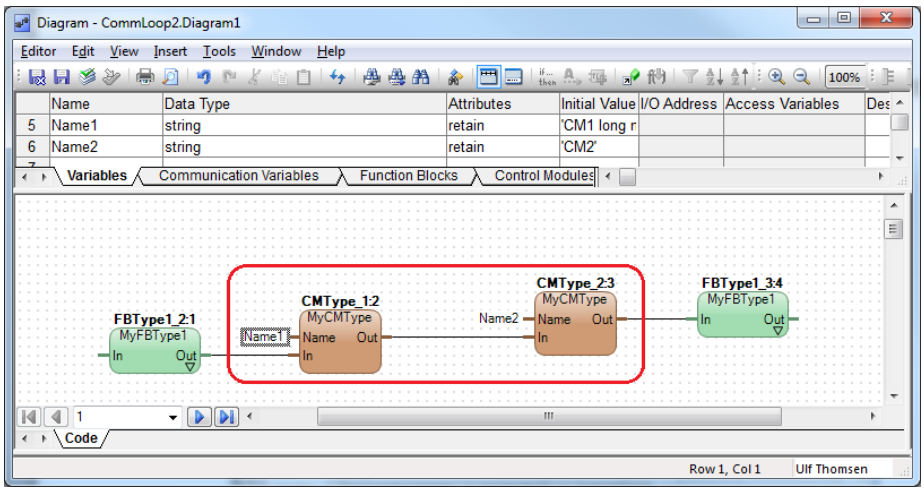


Figure 10. Flow Order and Control Module Code Sorting

In the Figure 10 the code sorting is done in two ways. The code execution of the two invoked CMTypes marked with the red rectangle will be code sorted as normal for control modules. This means that, for example a PID loop that are dependent on the code sorting for control modules to operate optimally will work in the Diagrams.

For the rest of the diagram the normal data flow order will determine the execution order of the Diagram. The data flow order in Figure 10 Flow order and Control module code sorting will be as follows.

1. FBtype1_2:1
2. (CMType_1:2 and CMType_2:3 sorted according sorting rules for control modules)
3. FBType1_3:4

Reverse and Display value attribute on data types in diagrams

As an result of the execution order described in Execution order on page 60, the reverse attribute will work for control modules in diagrams.

Split and Join functions shall not to be used with structured data types containing components with the reverse attribute. Functions and function blocks cannot handle reverse attributes.

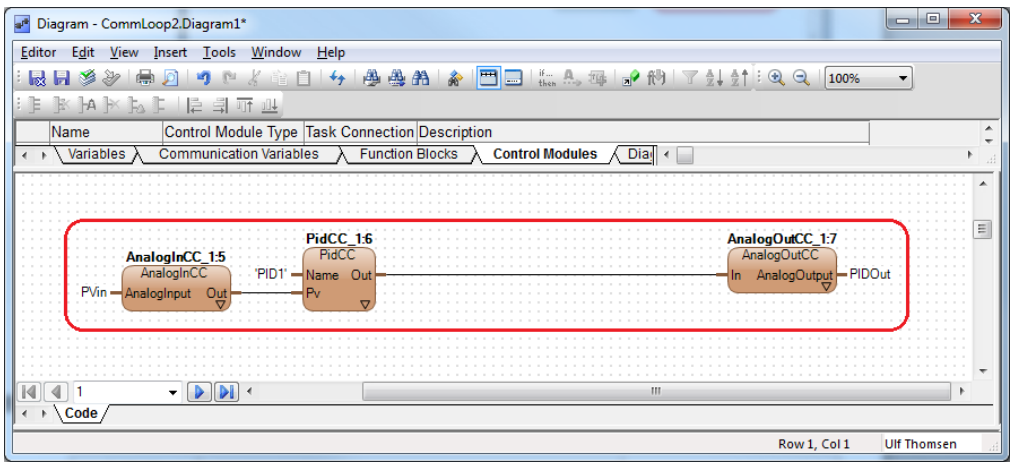


Figure 11. Control Modules in Diagrams

For diagrams there are also a attribute Display value for datatype. If the attribute are set for a component in a datatype this value will be shown in online mode for the connection.

Reverse attribute on data types in Diagram types

Bi-directional logic is handled by Control Module types, but it is possible to use a diagram type as well. The forward signal flow can be implemented in the diagram

code block, but the backward signal flow must be implemented in a separate code block.

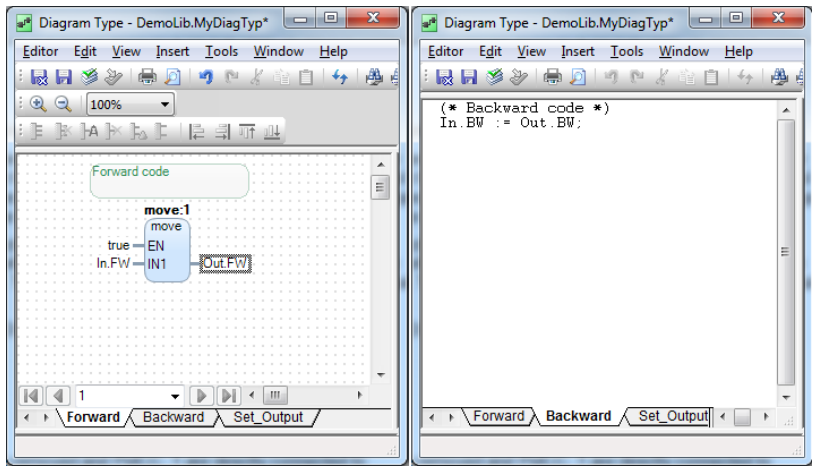


Figure 12. Diagram Type with Control Diagram (Forward) and Structured Text (Backward and Set_Output) code blocks

In Figure 12, the control diagram code represents the forward data flow and the ST code represents the backward data flow. The execution order of the code blocks will sorted according to the code sorting and algebraic loops for control modules.

Variable and Parameter

There are two ways of connecting parameters and variable in diagrams. Either by defining the parameter or variable name directly in the connection node of the

invoked object, or it is possible to insert a variable or parameter object in the diagram as in the lower example in the figure.

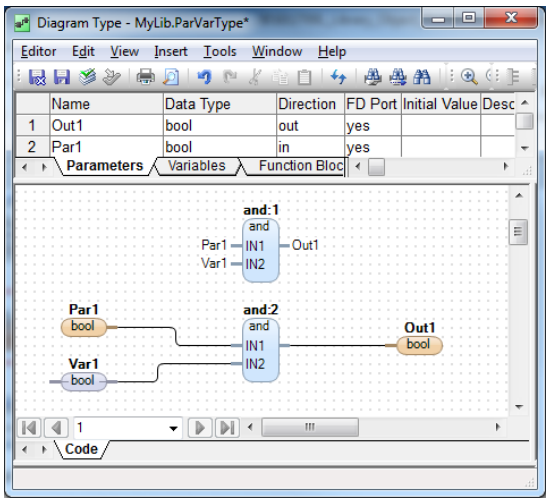


Figure 13. Connecting Parameters and Variables in Diagrams

If the parameters and variables are inserted in the diagram as objects, instead of just textual connections, then the load increases in the editor, but the controller load is not affected.

Interaction Windows in Online Mode

Introduction

There are two types of interaction windows for interaction in Control Builder; Interaction windows (primary) and Information windows (secondary).

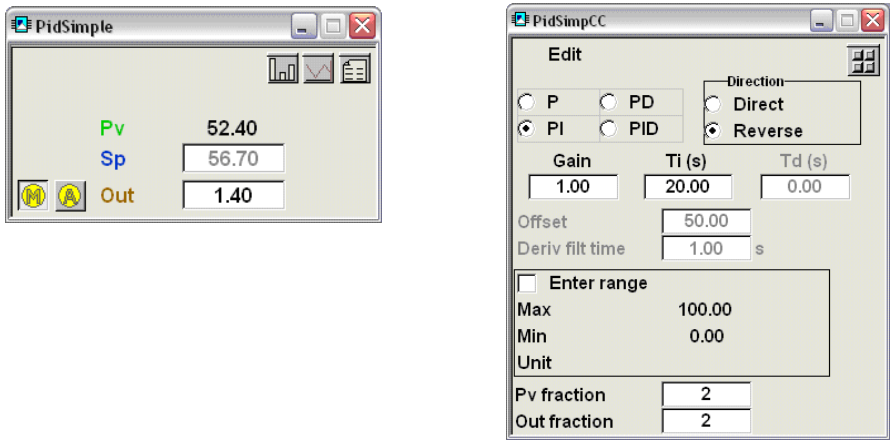


Figure 14. (left) Example of an Interaction window and (right) an Information window.

The instance name on the Interaction window (left image on Figure 14) is called Faceplate. The instance name on the Information window (right image on Figure 14), where interaction parameters are assigned, is called InfoPar. Additional Interaction windows shall have a name related to the functionality (for example, InfoAlarm, where alarm settings are to be done).

Information windows are activated from the Interaction window. Control modules, Function Blocks, and Diagrams designed for engineering interaction, shall have one or more interaction windows. The engineer may then control the control module or Function Block from these windows.

The information that can be contained in a window can be divided into three categories:

- Settings of functions to be used continuously which can be affected both by operator and program code.
  - Example: Integration time for a controller.
- Commands with or without associated input fields that can be affected both by operator and program code.
  - Example: Get a certain recipe (recipe name and the command itself needed).
- Presentation only.
  - Example: Actual value for a controller.

Interaction Windows

The interaction window contains the most important and frequent (mode changes, etc.) user interactions. There is also a sub window or a part of the interaction window which is called an Information window.

When the object type is selected, an interaction window shall be displayed. If an object type has more than one window, only the first one, that is the one that appears when the object type is selected, may be an interaction window. The windows that are reached from the main interaction window should all be information windows. If the main (interaction) window has been closed, it should be possible to re-open it from any of the secondary (information) windows via a button labeled with the ManoeuvreIcon.

Information Windows

A control module that is displayed as an information window should have the name Info and be accessible via the path string ControlModuleName*Info (where ControlModuleName refers to the parent control module) to simplify localization from outside. If a module has more than one information window their names should differ by a suffix, for example InfoBar, InfoHist.

When to use Interaction Windows

In general, windows are suitable when you have parameter values that should be tunable online (for example, controller gain) or if the user should be able to take control of the object (set to manual mode).

It is recommended to use interaction windows in Function Block Types when the number of parameters is so large that the FBD becomes confusing.

Typically, Control Module types are more complex than Function Block Types. Therefore, interaction windows are more often necessary for Control Module types (normally to facilitate configuring). Otherwise, the same rules as for function blocks apply.

Window Appearance

General Properties for interaction windows:

- Occupy a minimum of screen space
- If possible, group the window contents

The window width shall be a 0.192, 0.30, 0.35, 0.50 or 1.00 of the screen. The height of the window shall be set to 0. This means that the height is determined by the width and the shape of the control module.

The size of the window shall further be multiplied with the project constant *cWindowSizeFactor*. The *cWindowSizeFactor* is defined in a way that a window, showing a control module with width 4.0, has the same width as the screen if *cWindowSizeFactor* is 1.0. The project constant *cWindowSizeFactor* in BasicLib has a default value of 0.6.

The property *Owner* of interaction windows should normally not be used. The property should only be set if the operator window is a full size window (XSize = 1.00) that contains smaller windows. The attribute *RelativePos* is normally used if an interaction window is popped up from another window. The second window should be positioned relative to the first window using the attributes *xPos* and *yPos*, so that no part of the first window is covered by the second window. For full size windows the attribute *RelativePos* should be false.

An interaction window (popped-up from the control module instance in the control module editor) should appear with its lower left corner in the middle of the control module icon (achieved by setting relative position x = 0, y = 0). If it is possible to pop-up more windows from the interaction window, there top should be aligned with the interaction window top. They should also appear edge to edge with the interaction window. When window placement is made by adjusting relative x- and y-positions, the relative positions must be multiplied with *cWindowSizeFactor* to keep the alignment for different *cWindowSizeFactor*.

In exceptional cases there may be parameters to determine where the window should appear. There may also be a parameter, which determines if the position is relative to the module or to the screen.

Window positioning should be as follows: parameters to the right, trim curve to the left, and bar graph beneath the interaction window.

Design

Window layout

All interaction windows shall have equally high texts, regardless of the window size. To achieve this, the width of the control module that represents the interaction window has to be proportional to the width of the interaction window according to the following table:

Table 9. Width of windows and control modules

Window width	Control Module width	Used for windows with
0.192*WSF(1)	0.8	buttons, check boxes, option buttons, one vertical bar.
0.30*WSF	1.2	parameter lists, several vertical bars.
0.35*WSF	1.4	parameter lists with additional information or with long texts.
0.50*WSF	2.0	history graphs
1.00*WSF	4.0	configuration of large functions, for example fuzzy, batch

(1) WSF= cWindowSizeFactor

In some cases several input fields in an interaction window are logically connected to each other. Examples are option buttons where one and only one alternative in a group of alternatives is selected. The grouping should be done with a rectangle (similar to the ActiveX control Frame in standard MS-Windows programming) around the interaction fields in question. Use window background (-2) for the frame caption text to obtain look similar to a Frame ActiveX control. Note that it is

possible to have separate buttons (for example, pairs Apply/Undo) for the different logical groups.

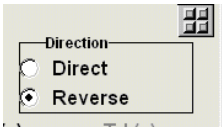


Figure 15. Grouping of option buttons.

The following rules should be used in the control module for the interaction window:

- Grid size = 0.02
- Text height = 0.1
- Text height for group = 0.08
- Distance to borders = 0.02
- Distance between texts in a group (vertically) = 0
- Distance between groups can vary, yet be at least 0.02
- Straight (aligned) columns in each group
- Check boxes, option buttons and input fields are used for inputs
- Icons can be used for distinct indications. It is also permitted to use check boxes and option buttons in the same way as a dynamic input field
- “Apply” / “Undo” buttons when two or more interdependent input fields. The buttons should have size 0.3x0.1
- Input fields should be center aligned
- For option buttons and check boxes, the text should be to the right of the interaction object.
- The caption texts in an operator window should have the height 0.16.
- A value may have a ResetIcon to the right. Selecting the ResetIcon will reset the value (to initial value) or, in some cases, other reasonable values.
- Color assignment shall follow common guidelines and regulations, for example red for alarm indication, and yellow for warning indication. Use project constants (cColor) to assign colors to objects.

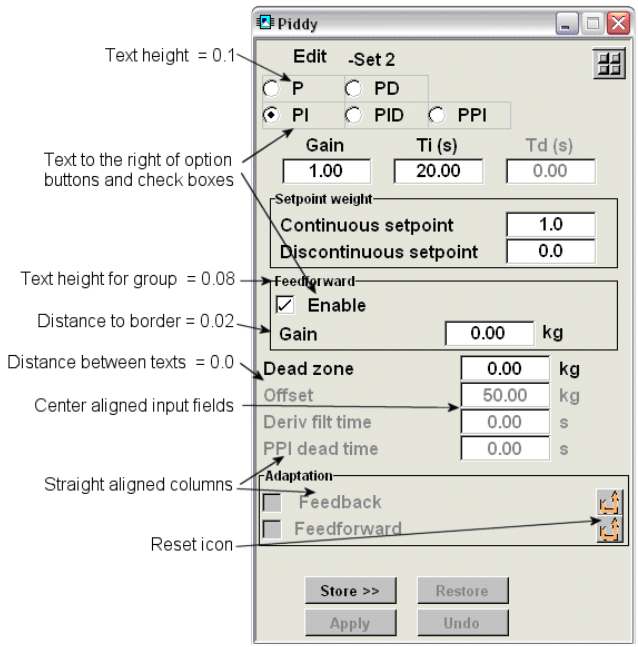


Figure 16. Interaction window design.

Interaction principles

Synchronization of data write

Sometimes several values should be applied simultaneously, for example the tuning parameters for a PID-controller. There is a built in solution for this in the Control Builder graphics. It is possible to assign OK/Cancel buttons that are associated with several input fields.

Another solution is that the values are stored in a temporary set of parameters when an Apply button is pressed. There should also be an Undo button to restore the used values. This method requires extra variables in the controller, but is good to use since the same method can be used for the Operator Workplace graphics.

Dimming of objects

Texts are dimmed by changing the color of the text. Normally the text has foreground color (-3). The alternative color, that is the dimmed color should be 4. The condition for using the alternative color should be NOT EnableName.

The project constant cColors.Disable function can be used for dimming the objects.

If the operator window has an input field (interaction object) for a physical value, this input field usually is preceded by a text presenting the name of the quantity, and succeeded by a text presenting the unit of the quantity. If the input field is disabled, both the preceding and succeeding texts should be dimmed. Static texts and text objects that present values that are not changeable from the user window should not be dimmed. Icons that cover interaction objects should always be dimmed if the interaction object is disabled.

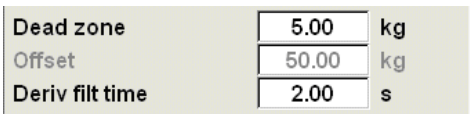


Figure 17. Dimming of an input field (here is the Offset field dimmed).

Section 5 Operator Interface

Operator refers to a user that is interacting with the system via a graphical interface in online mode. This includes actions during testing, commissioning, tuning, maintenance and daily operation via an operator‘s Workplace.

Introduction

Graphic interfaces fetch data via parameters and variables in the object type. In the sub-section InteractionPar on page 36, the concept of InteractionPar, a parameter of a structured data type was introduced. The InteractionPar concept is vital for the standard libraries operator interface.

Operator Workplace Interaction

It has to be decided for each object type if it should have operator graphics in the Workplace (Process Portal). Examples of object types that should have Operator Workplace graphics are high level objects such as Motors, Valves, PID-controllers etc. and objects that are alarm owners.

For objects that will have interaction possibilities in a Workplace, with the user or other objects, the attribute Aspect Object shall be set on the object type. Aspect Object is an attribute that decides whether the object will be visible in the Plant Explorer, or not.

Design

Reuse of graphic elements should be practiced in order to reduce maintenance. Since the system do not fully support reuse of graphic elements in combination with Library version handling, a specific method has been developed. The main idea is to put the reuse elements in a special support library from which they are dragged and dropped to be instantiated.

The Operator Workplace graphics can be divided in two categories, Faceplates and Display Elements. These two categories are described below.

Faceplates

Each object with Workplace graphics should have a faceplate named “MainFaceplate”. The faceplate is used for supervision and control of the object. The faceplate can have three different views; Reduced, Normal (Figure 18) and Extended (Figure 19).

The Normal view is mandatory and has a predetermined size to fit into group displays.

The Reduced view is default view and should contain the minimum of information that is needed for normal operation. The reduced faceplate can be omitted if the information will be the same or nearly the same as in Normal view and it is impossible to reduce the size.

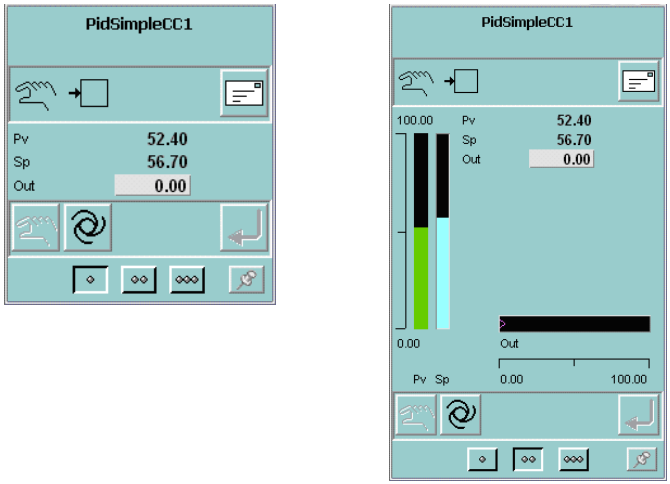


Figure 18. Example of (left) Reduced faceplate and Normal faceplate.

The Extended view is optional and should be used for additional information that does not fit in a Normal view and is not so often used during normal operation.

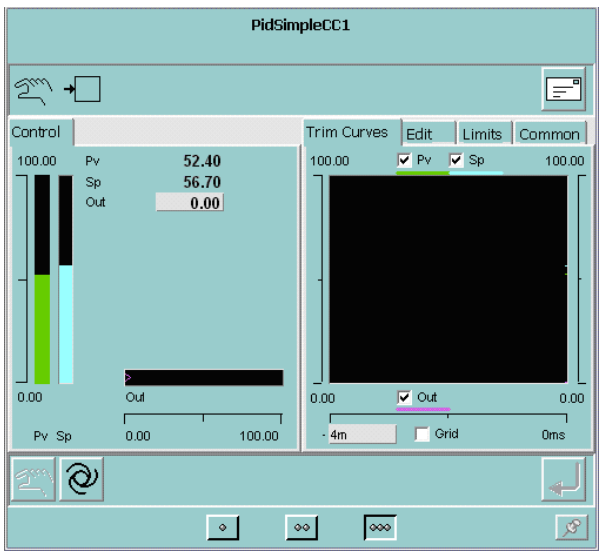


Figure 19. Example of Extended faceplate

Display Elements

The Display Elements are used for graphic displays. The following Display Elements aspects are mandatory for objects with Operator Workplace graphics:

- Reduced Icon
- Icon
- Tag

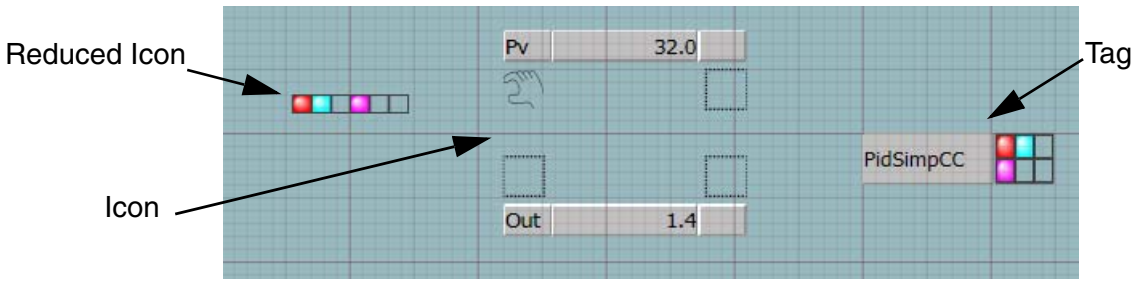


Figure 20. The mandatory display elements; (left) Reduced Icon, (middle) Icon and (right) Tag.

In addition, the following Display Element aspects can be used if suitable:

- Value
- Bar

Figure 21. The additional display elements; (left) Value and (right) Bar.

The Display Element Value and Display Element Bar aspects should be used when the object has one or more numerical values that can be considered as significant for the function, for example the output from a PID controller.

National Language Support (NLS)

All texts in the faceplates shall be NLS-strings. The NLS-translator is common for all standard libraries so that the total number of NLS-strings is limited. The NLS-translator should further be placed outside the libraries since there is no easy way to have version handled NLS-strings. Thus, since there is no version handling of NLS-texts, it is not allowed to delete or change the meaning of a text once it has been included in a release.

The alarm and event Message parameter and the initial value of the alarm ConditionName parameter should also be NLS-treated via an Alarm and Event translator. The Alarm and Event translator shall be common for all standard libraries and placed outside the libraries.

2	AlStateDevPos	uint	default	OUT Positive deviation alarm condition state (0-6)
3	CondNameNegDev	String[15]	'||SL_Neg_Dev'	IN EDIT Name of the condition for negative deviation alarm
4	EnableDevNeg	bool	true	IN Enable the negative deviation alarm

Figure 22. Example of NLS treated default value of Condition Name.

Interface between Control Builder and Operator Workplace

Control Builder parameters and variables are visible in the Operator Workplace via the OPC server. Variables that are not used in the Workplace graphics should be given the attribute hidden. Function block parameters should never be given the attribute hidden even if they are not used in the graphics. The reason for this is to be consistent with Control Module parameters, which cannot have any attributes. Extensible parameters cannot be accessed via the OPC server.

Naming of any non-hidden variables should follow the same rules as parameters, see Section 3, Parameter Interface (no restrictions on name length).

Permissions for Variables that are not Hidden

All parameters and variables that are not hidden are assumed to be read by everyone. Therefore, no read permissions are assigned. Table 10 describes the write permissions that can be used. These are restricted.

Table 10. Write permissions

Write Permission	Applied for
Administrate	Read only parameters / variables.
Tune	Tuning parameters. For example Pid parameters, filter time, etc.
Operate	Parameters which are expected to be edited in the daily operation of the plant. For example, Mode changes, start and stop, etc.

Interaction principles

The interaction principles in this section should be seen as a complement to the general 800xA process graphic principles.

Use of buttons and check boxes in faceplates

Check boxes shall be used to enable or disable functions and signals.

Buttons in the button row of the faceplate framework shall be used to control main functions of the object.

Buttons inside the faceplate elements shall be used to start or stop sub functions.

SIL considerations

Access Level

The Access Level defines the rules for changing online values of a running SIL application.

Access Level shall be configured on parameters used for interaction in faceplates of SIL1-2 and SIL 3 marked types.

For variables and parameters of simple data type the Access Level shall be configured on the variable/parameter (configured in the object type). For variables and parameters of structured data type, typically InteractionPar, the Access Level shall be configured on the variable/parameter and on the data type components.

Only variables/parameters (or their components) used for interaction shall be configured. The Access Level shall be set to *Confirm*. Other components, types, parameters, and variables shall not be configured and will keep the default Access Level *Read-Only*. When the system evaluates the Access Level for an OPC property, the whole path is evaluated and the most restrictive setting found along the path will be the setting for the whole path.

This means that the Access Level for sub-object instances shall be overridden to *Confirm* in case the sub-objects have a faceplate with interaction.

Support for Confirm Operation Dialog

The faceplate shall be designed so that it is possible to recognize Operation, Object, Property, and Value in the Confirm Operation dialog.

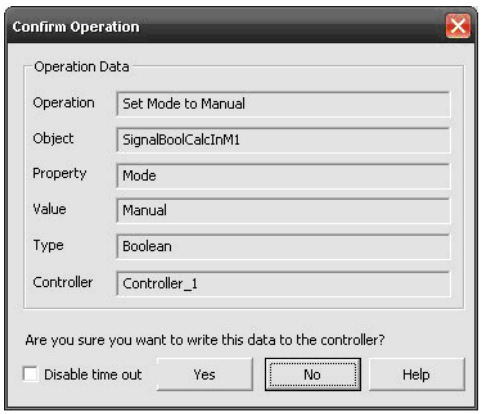


Figure 23. Confirm Operation dialog

For properties that can be changed through the faceplate (SIL marked types), a separate NLS string shall be defined for confirm write support in the NLS Manager for Access Management.

The Safe Online Write requires that the information displayed in the confirm write dialog is stored and handled separately from what is displayed in the faceplate (for example, using different NLS managers, in order to avoid systematic errors).

In the Confirmed Write Support aspect on the type, all changeable properties are configured with a Confirm Name, Confirm Value(s), and Operation.

The Confirm Name text shall describe what the operator changes. Examples: Mode, Command, HH Level.

The Confirm Value text shall describe the (process-) action of the operator. Examples: Set, Reset, Enable, Disable, Auto, Manual.

Confirm Value texts shall always be defined for Boolean values (one text for true and one text for false). Value texts for integer values shall be configured in case the integer value represents something except the value itself, for example, a mode.

Example:

Table 11.

Property: SignalErrorMode (dint)	
Value	**Description**
1	Through
2	Freeze
3	Pre-determined

For other data types, for example: real and time, no Value text is necessary. It is the value itself that is displayed in the Confirm Write dialog.

The mapping between the value and the Confirm Value NLS string is made in the Confirm Write Support aspect on the type.

The Operation has a default configuration which is "'Set ' %P ' to ' %V" where %P and %V refers to configured Property Name and Property Value, respectively.

The NLS identifiers for the Confirm Operation dialog shall be named as the identifiers for the faceplate, but with a '_AM' suffix.

For Control Builder standard libraries, the NLS identifiers for the Confirm Write dialog shall be configured in the 'NLS Resource Manager for AM' which is located under Object Type Structure - Libraries - Access Management.

For user defined libraries a corresponding NLS Resource Manager shall be added for example in a sub object to the Libraries object.

Example:

This example shows the configuration for the action when ordering the manual value to On. The involved property is InteractionPar.ManValue. When pressing the On button in the faceplate the dialog shown in Figure 24 appears.

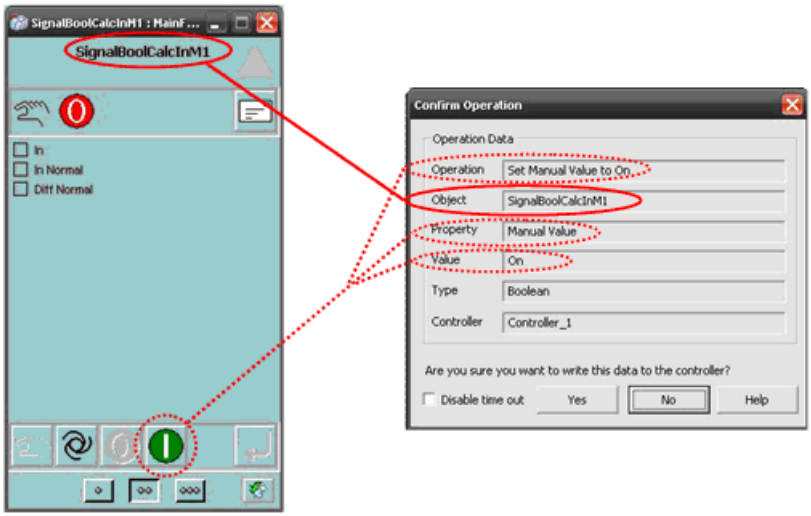


Figure 24. Relation between Faceplate information and Confirm Operation dialog

The Faceplate button in Figure 24 is configured in the Main Faceplate aspect with a Property name and Property Value to which it is set, see Figure 25.

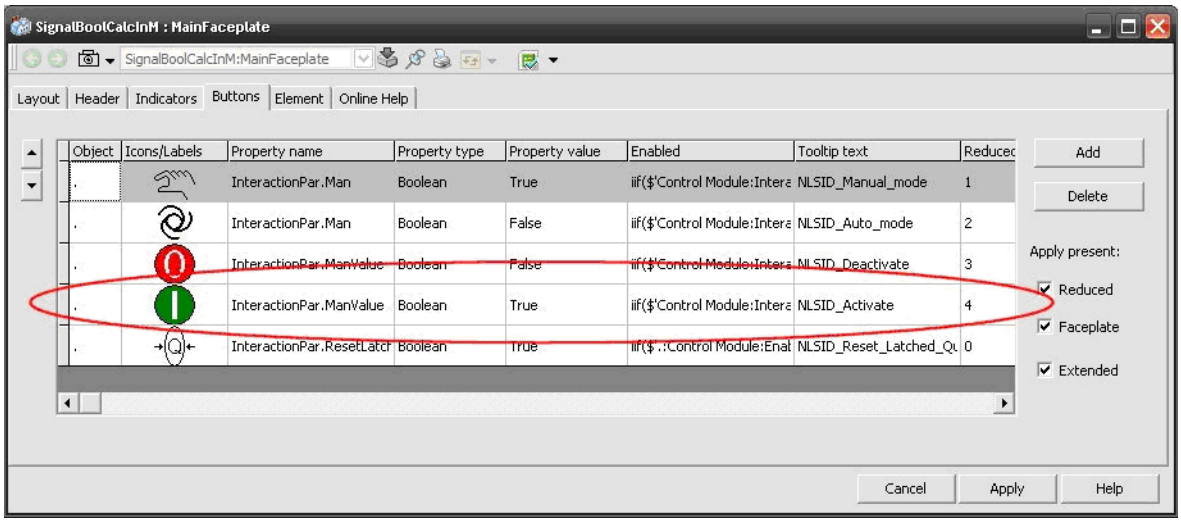


Figure 25. Faceplate button configuration showing associated Property Name and Property Value

In the Confirmed Write Support aspect on the type, the property is associated with a Confirm Name, Confirm Value(s), and Operation, which is displayed in the Confirm Operation dialog.

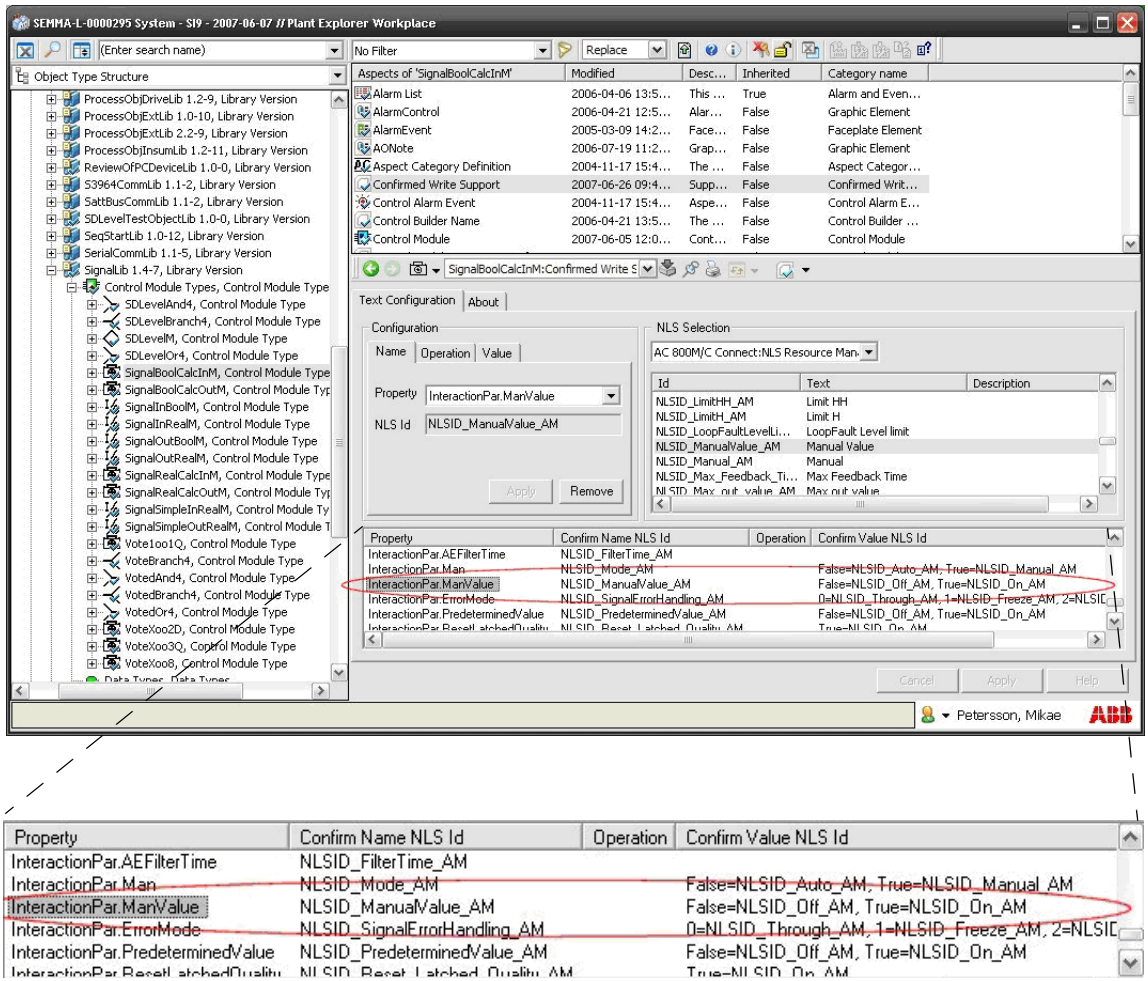


Figure 26. Configuration of the aspect Confirmed Write Support for the Confirm Operation dialog shown in Figure 24

Graphical Indication of ParError

The ParError parameter, introduced in the sub-section ParError on page 37, indicates when an input parameter of an object is out of range. In that case the erroneous value is not used in the code. Instead a documented 'safe' value is used.

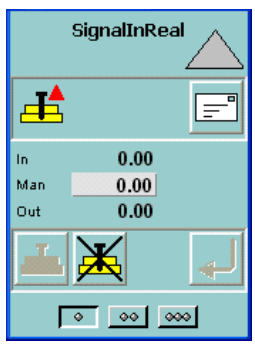
The graphics, Operator Workplace and Control Builder, shall always show how the object works. When ParError occurs the 'safe' values or last good values are used and the object works according to them. Therefore the graphics shall show the 'safe' values and the effect of them.

For example, if AEConfig has an out of range value the 'safe' value 1 is used. The value 1 means that the object is configured with an alarm. In this case an alarm icon and button shall be displayed as they are when AEConfig=1.

An exception from this rule is how the components of InteractionPar are displayed. The values are entered from the graphics and the entered value shall be displayed. In most cases this is not a conflict because it is not possible to enter out of range values from the graphics. The reason for the exception is that the value of the component in InteractionPar must be displayed, because the interaction is done with this variable.

The Operator Workplace Graphics

The parameter error situation shall be indicated in the faceplates and display elements. The faceplate indication shall be a small red triangle placed in a layer above the ordinary icon in the first position in the indicator row. In case of parameter error when no other icon is visible, the error is indicated with a bigger red triangle. The Figure 27 below show the different possibilities together with the Force icon.



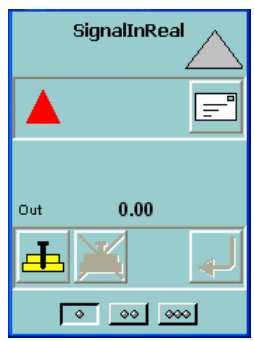


Figure 27. ParError indication on a Graphic Element Reduced Icon (left) and in Workplace faceplates (right).

The CB Graphics

Control Builder graphics have the corresponding red triangle displayed in the upper left corner of the interaction window Figure 28.

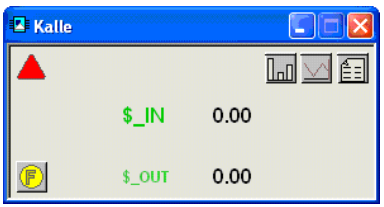


Figure 28. ParError indication in CB interaction window.

Section 6 Program Code Issues

Program Code

Structured Text shall be used for all program code in the library types. The only exception is the use of SFC in template types, see Templates on page 94.

The most important general requirement on the code is that it should be efficient. This means that it should use few local variables, execute fast and generate little communication. However, although high performance may be the most important factor in types (especially in those that will be invoked many times), the code should still be possible to interpret, also for persons who have not written it. Therefore, supply comments that clearly describe the different parts of the code. Do not forget that the identifiers of variables and parameters should be descriptive as well.

Comments are not downloaded to the controller; hence it does not occupy any controller memory space.

In addition, logical indents of loops etc. should be used at all times. Follow up copy/cut and paste operations carefully as these operations tend to undermine the indentation patterns.

Although SFC might be used in templates, it shall not be used in other library types since it is less efficient. An ordinary code block can be designed to behave as a sequence if the code uses an integer variable, preferable named *PrgStep*, as a sequence token. The integer variable value represents the step number of the code and the code is divided into sections that represent steps. The program steps are normally numbered in 100-steps. Conditional statements based on the token variable determine the execution of the different code sections. The principle can be seen in Figure 29.

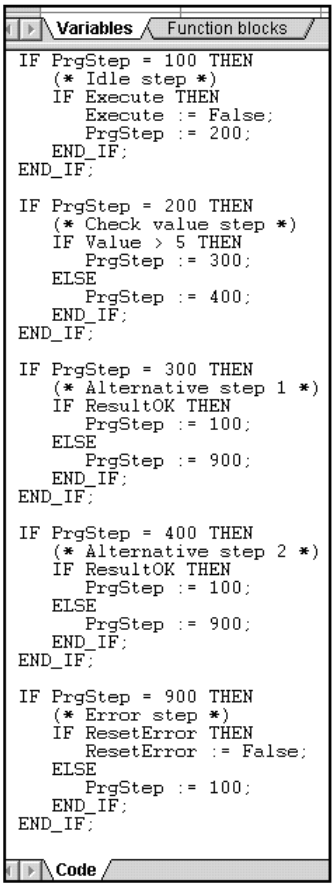


Figure 29. A code block using a sequence token variable called PrgStep.

Descriptions

A description should be provided wherever possible. This means that all object types, and data types shall have a brief (three to four short rows) and clear description to be shown under the Description tab in the lower pane of Project Explorer when the object is selected. This description text is used as base to the online help. It is also recommended to initiate the description of function document with this text for the specific object.

A structured data type component shall have a line of text briefly describing its purpose/function. Parameters in object types should be described similarly.

Do not write any reference to the online help in a parameter description in an object type. Normally, the online help describes important parameters that require additional explanation.

Variables and Project Constants

The following recommendations apply to the variable names:

- Variable names shall be descriptive.
- Underscore (_) shall not be used; instead, separate the different parts of a variable name with uppercase letters.
- Avoid Global/External variables whenever possible.
- Avoid Access variables whenever possible.
- Avoid very long names.
- Add *Old* to the variable name to create a variable that stores the old value.
- Add *Loc* or *Int* to the Avoid the use of the “Retain” attribute on a variable, when it is not necessary. For example, when the variable obtains its initial value at warm restart and when it is written to before it is read, “Retain” is not necessary (it only contributes to increased stop time during download of changes to the controller).

If it is sure that a value of a variable will not change, it shall be assigned the attribute “constant”. This solution also yields a slightly reduced CPU load.

It is not recommended to suffix or prefix variables or user defined variables to show it is a variable, for example, varName or Namevar.

During the development and testing phase of a new module, it is useful to have a lot of status variables in the code. It may be necessary to have one variable for each procedure call in order to make debugging easier. But, when the debugging phase is completed and the module works as expected, the number of variables shall be minimized. This can be achieved by using the same status variable for several procedure calls.

Project Constants

Project constants can be read in libraries and be changed project wide with a single operation. They are suitable for library items that the user may want to change. Examples are date and time formats, logical colors and names, alarm condition texts, and alarm and event texts.

Project constants shall not be used to change functionality of an object (for example, as initial values and as comparisons in code).

Since the project constants are stored on library level, it is important to consider incompatibilities at new library versions. There can be only one default value of a project constant with a certain name and since it must be assumed that a library version should co-exist with previous version, the default value should never be changed in new library versions.

Project constants shall begin with a lowercase “c”, followed by an uppercase letter. For structured project constants, this rule applies to the main name only, not to the individual components (for example cColors.Error).

Object Sub-Structures

Protection and Scope

Internal function block/control module types, that are used for code-reuse (see further Re-use of Code on page 89), shall have both Protection and Hidden attribute set to True. See Protection Attribute on Types on page 42, for the non-hidden function block types/control module types

All non-hidden object types shall have Public scope. The scope of an internal type depends on the usage. The scope shall be Private if the type only is used within the same library. Types used in other libraries must, of course, have Public scope. PPA aspect can be used for object oriented graphical design even if the types are protected, by using the option *Sub Objects visible in PPA*.

The non-hidden object types with Private scope shall be sub objects to the object type with Protection attribute.

Re-use of Code

Code re-use requires more variables and execution time, but facilitates maintenance. Internal (hidden) function block types should be used to improve the structuring of object types. It is also possible to implement functions by means of placing sub control module types within both Control Module- and Function Block types. Control Modules within Function Block Types are mutually sorted. The following shall be considered for internal types (yet, the benefits of efficient code should also be recognized).

Avoid use of very simple function blocks for structuring or re-use of code; the overhead caused by the resulting variable copying will be higher than writing the equivalent code in ST.

Naming of object type formal instances is realized by omitting the Name part from the corresponding type name. For example, if the object type name is PIDFaceplate, the instance name becomes Faceplate, BiCore becomes Core, and so on. It is desirable to have short, but descriptive, path strings in, for example error messages including variable names. Object type instances that can appear in loop messages should always have a descriptive name.

Names of object types on the same hierarchical level must be unique.

Simple data types should be used for parameters as far as possible in internal Function Block types since this gives more general types that can be used in different contexts with good performance. Only in rare cases may structured data types prove to be a more efficient solution. When using structured data types of direction Out, variable copying can be eliminated by declaring these as In_Out or By_ref. Instead of variable copying, a single pointer (address) reference is used for the entire structured data type and its components, something that saves memory and execution time. Note that these benefits are not valid for simple data types which should be declared according to their actual use. In-parameters should be declared as In and Out-parameters as Out. Also note that by declaring a parameter as In_Out, the compiler will consider it to be both read and written by the function block. This might cause some problems when using sub function blocks; in control modules as well as in function blocks. For control modules and diagram type instances it will affect the code sorting and might cause sorting loops. For function blocks it will be impossible to connect the parameter to a surrounding parameter declared as In.

It is possible to address sub function block parameters using dot notation. This can be used in order to save memory and execution time. One way to use the dot notation is for access of Out parameters. See the example in Figure 30, where the variable x can be omitted by using dot notation. It is not possible to use dot notation for parameters with By_ref attribute.

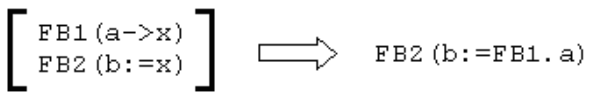


Figure 30. Example of dot notation for access of Out parameters.

Since the parameter value is stored in the function block, there is no need to assign all parameters in every function block call. See the example in Figure 31 where both memory and execution time is saved by the use of dot notation. Another variant of this solution can be used when the same function block is called several times. The fact that parameter values from the last call are stored can be used and not all parameters needs to be assigned in each call. However, this solution should be used with care, because one has to keep track of which values that actually are used.

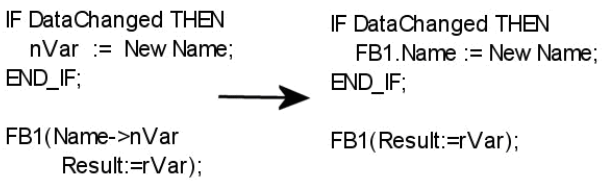


Figure 31. Example of conditional assignment of parameter using dot notation.

If the input parameter default value is satisfactory, the copying becomes unnecessary. It is therefore worthwhile to choose default values with care.

To connect a string literal to a function block, declare a variable of type *String[]* of correct length (if no value is specified, the string length defaults to 40 characters) with the attribute “Constant”.

Control Module Types

Code Sorting and Algebraic Loops

The code block sorting is a powerful tool that can be used to minimize delay between input and output signals. This is achieved by dividing the code into different code blocks in Control Builder. Code blocks can be used when information is transmitted between objects in both directions using a structured data type. By splitting the code into two or more code blocks it is possible to both send and receive information within the same scan.

However, the situation is different when it comes to connections for example reading and writing to single parameters. The basic principle is that an application programmer should be allowed to write any IN parameter and read any OUT parameter of a module without causing loops. This is achieved by writing to all OUT parameters in a specific code block. This code block shall not contain anything else but the writing to the OUT parameters. Local copies of the OUT parameters are used in the other code blocks. These variables shall have the attribute nosort to avoid loops. In order to secure that the output parameters are updated correctly, each of these internal variables must be assigned in one code block only. If an output parameter is dependant on calculations in more than one code block, there need to be one internal variable with the intermediate result per code block.

For the parameter **InteractionPar**, the attribute “Nosort” is used under some circumstances to prevent code loops. The code shall be written in such a way that no component of **InteractionPar** is marked as "written".

A problem occurs when **InteractionPar** is passed into an internal Function Block as an *IN_OUT* parameter. In this case, all components of **InteractionPar** shall have the attribute "Nosort". This works fine in all cases except, when the order of assignment of the components of **InteractionPar** is crucial. There is only one known case when this order is crucial and that is when an "Apply" component is set. In this case, all other components must be set before "Apply" is set.

Code tab names in control modules that begin with **Start_** are sorted separately and execute once before all other code in all other control modules in the application after warm or cold start. The “Start_” feature can be used when a different code sorting order is desired during initialization.

The code in the control module type should not be divided in code blocks for any other reasons than taking advantage of the code sorting. Unnecessary code blocks will cost extra execution time, especially in SIL-applications, and might danger the code sorting.

A compiler switch exists to rule the code sorting loop detection. Code sorting loops may hazard the object functionality at code modification elsewhere. It is recommended to avoid and remove all kind of sorting loops before any download to a controller takes place. It is also recommended to keep the compiler switch as 'Error' to avoid unforeseen errors.

The operation allowed on a parameter is governed by the type description keyword (see also Type Description Keyword on page 32). All parameters can be connected to the code block in question, but the rules for reading and writing is given in Table 12.

Table 12. Rules for code block read and write operation to a control module

Keyword	Rules for read/write operations to Control Module parameters from a external code block
IN	The parameter is written to and read from.
OUT	The parameter is only read.
IN(OUT)	The parameter is both read and written, but mostly written.
OUT(IN)	The parameter is both read and written, but mostly read.
NODE	Only connected. Read and write operation are done by other control module.
EDIT	Only written to, and only first scan value will affect the control module.

Some special purpose parameters do not obey the above rules.

- InteractionPar is used for operator interaction, and is allowed to be both read and write.
- I/O parameters of data types BoolIO, RealIO, DIntIO or DwordIO should only be connected.
- Connection parameters for communication between control modules, should only be connected.

Function Block Types

Even though there are no code sorting of code blocks within a function block type, the code might be split into more than one code block. This make the code more readable, especially for objects that implements different, non dependant functions. In SIL application there is an increase of execution time for each switch between code blocks, so the number of code blocks should be minimized.

Diagram Types

Diagrams are created under an application, and diagram types (which can be reused as instances in a diagram) are created under a library or under the same application as the diagram.

The FD code blocks in diagrams and diagram types allow mixing of functions, function blocks, control modules, and other diagrams, and graphically connect them to achieve a particular logic. The code in the function blocks is sorted according to the placement in the diagram, and the code in the control modules are sorted according to the data flow order. The actual execution order is indicated on each instance in the diagram. The code sorting of the diagram has to be considered when designing a diagram type. The use of function block types and control module types with a diagram type has to be done with a understanding of how the code are sorted in the diagram type. Signals in backward direction (components with reverse attribute) cannot be handled by Split or Join elements. Instead, these signals must be handled in a separate Structured Text code block in the same manner as in control modules.

Data Types

Data types common to several libraries shall be placed in a supporting library. Changing of initial values can be realized through project constants.

The data types can be divided into three categories:

1. Private data types, not used in parameters or in sub-data types in parameters. Should have attributes Private and Hidden and have a complete description of all components as well as the type itself.
2. Internal data types, where the content is of no importance for the user for example graphical connections where the user has no interest of the single

components in them. The data type should have attribute Protected and have a complete description of all components as well as the type itself.

3. All other data types are presumed to be Public data types. All the Public data types are important for the user. The Public data types should have a complete description of all components as well as the type itself.
4. ISP values must be defined for all components in a data type on order to be used for a communication variable in a SIL application.

It is not recommended to use the attribute State. Use a construction with a local variable with the suffix Old, which holds the value of the variable from last scan. The Old-variable should be assigned at the end of the code block in question.

Templates

There are two kinds of templates: those that are functional as is, and those that require modifications to work. The latter category shall be NameTemplate (for example, EquipProcedureTemplate). Code, intended to be changed by the user, shall be put in templates. All other functional code should be placed in protected object types. This strategy promotes safe future updates of programs.

Whether the “core” (that is, the code that does not have to be changed) of a template should be designed as a function block type or a control module type, has to be determined in each specific case.

In templates, it is recommended to use Structured Text. SFC may be used in templates under certain circumstances, for example Batch recipes.

Naming of variables is especially important in templates (compared to protected code), because the user shall be able to easily understand the design and function of any given template. Refer to Variables and Project Constants on page 87, for variable naming recommendations.

Task Considerations

The normal case is when a control module type inherits the task connection of the surrounding object type. There might however be some special cases where it is wanted to have a sub module/function block running in another task than its parent. This kind of constructions are allowed but should be used with caution. One special

feature of sub function blocks executing in another task than its parent is that it will not be executed at function block calls in structured text code.

Parameter Dependency on Tasks and Controllers

The tasks within a controller are recommended to have interval times that are multiples of each other, for scheduling purposes. The tasks in different controllers execute asynchronously.

For control modules and diagrams, the code is sorted to minimize delays in the signal flow. The code sorting has no effect on control modules connected to different tasks.

The task ownership of the parameter is divided between the tasks, according to the particular task that writes to the parameter at a time. Therefore, it is possible to have reliable execution of control modules over task and controller borders.

According to IEC 61131, function blocks are designed to execute in the same task as the program they are part of. For example, some parameters like *Done* are only set during one scan. This is the reason why an object in a slower task, reading this parameter, may not receive it.

The passing of parameters to and from the function block is controlled by the program task. The code in the function block that is connected to another task, runs according to its task, but the parameters are updated according to the program task. In such a situation, it is possible that the IN_OUT parameters of a function block are changed during its execution, if a task switch occurs.

This problem does not occur for *In* and *Out* parameters, since they make use of local copies within the function block, which remain unchanged during the execution of the whole function block.

An application using function blocks shall not be separated into several tasks.

The initialization of objects using an IF-clause may not receive necessary input parameters, if these come from objects residing in other tasks or controllers.

Calls to Asynchronous Functions

Use parameters like Enable to govern asynchronous Function Blocks. Do not use SFC or IF-clauses since the asynchronous FB can have an internal state which will influence next invocation.

Special functions

Handling of Input and Output Values

The code shall handle its input and output values in the following way:

The code shall have a predictable and reasonable behavior for all input values. Input values, which are considered invalid, shall be mapped to valid input values. It shall be described in the parameter description which values are considered invalid and which valid values these are mapped to. For further details see Range Check on page 103.

The object must have a predictable behavior when calculating output parameter values. Situations like divide by zero and sqrt() of negative numbers shall be prevented or handled in the code. The library object shall not give an overflow output value if none of the input parameters have an overflow value.

Parameters of direction out and Parameters of direction in_out with the description keyword OUT or OUT(IN) shall be written to in each scan.

Error Handling

If there is any risk that the type may end up in an error state or have a malfunction, it should have error handling.

If case of error, an ErrorIcon should appear in layer 1 of the module. An ErrorIcon should also appear in all operator windows.

If an error occurs there should also be an output, called Error from the module. This output should be boolean and be reset when the malfunction ends. In a program unit the Error output may be connected to an AlarmCond.

Some control modules, which utilize more complex functions, may require a more advanced error diagnostics. For these modules, an error display shall be included in an interaction window. This error display should contain a text explaining the error and the error code. The error code can come from a failed procedure call, or from the code itself (Number range -5000 to -6000 for standard library objects). It may also contain information about where the error has occurred in the code.

Alarm and Event Handling

If no specific reason exists, the more efficient type AlarmCondBasic shall be used.

If an object has alarm handling, the property Alarm Owner should be set.

The AEConfig parameter is used to define whether a supervised condition should generate an alarm or event. The possible settings are:

AEConfig = 0, Supervision disabled.

AEConfig = 1, Alarm.

AEConfig = 2, Event generated both on condition activation and deactivation.

AEConfig = 3, Event generated on condition activation only.

AEConfig = 4, Supervision only, that is neither alarm nor event generated.

There can be more than one AEConfig parameter per object. It is then possible to define different configuration for different condition within the object. For example; the AEConfigH parameter configures the High condition, the AEConfigHH parameter configures the High High condition etc.

Table 13 summarizes the Alarm and Event parameters for the Standard Library Objects and Figure 32 illustrates the alarm and event logic.

Table 13. Summary of Alarm and Event parameters

Parameter	Direction	Data type	Function
AEClass	In	Dint	The class for all alarms in the object.
AEConfigX	In	Dint	Defines whether condition X should generate an alarm or event. Possible values are 0-4 (see above). This parameter should be used as a configuration parameter.
AESeverityX	In	Dint	Severity for alarm condition X.
CondNameX	In	String[15]	Name of the condition X. Shall have a NLS-treated default value.
EnableX	In	Bool	If false, the XStat (and XAct) output is kept false and the alarm/event generation is disabled.
InteractionPar.EnableX	In	Bool	Same as EnableX but controlled from faceplate. This component is harmonized with EnableX and actions from the alarm list. For example, if condition X is disabled from the alarm list, InteractionPar.EnableX is set to false. When declaring the data type for InteractionPar, the EnableX component must be declared with the attribute “retain”.
InhXAct	In	Bool	Inhibits the XAct output. Only introduced for a selection of objects.

Table 13. Summary of Alarm and Event parameters (Continued)

Parameter	Direction	Data type	Function
InteractionPar.InhXAct	In	Bool	Same as InhXAct but controlled from faceplate. Only introduced for a selection of objects. When declaring the data type for InteractionPar, the InhXAct component must be declared with the attribute “retain”.
X	Out	Bool	Indicates active condition X. This output is enabled as long as AEConfigX <> 0. Example: GTH.
XStat	Out	Bool	Indicates active condition X. This output can be controlled via Enable.
XEnabled	Out	Bool	Indicates that the XStat and XAct parameters are enabled. This parameter can be used when additional actions should be performed at Enable/Disable.
XAct	Out	Bool	Indicates active condition X. This output can be controlled via Enable and Inhibit.
XActInh	Out	Bool	Indicates that the XAct parameter is inhibited, either from faceplate or code. This parameter can be used when additional actions are wanted at Inhibit. Only introduced for a selection of objects.

Table 13. Summary of Alarm and Event parameters (Continued)

Parameter	Direction	Data type	Function
AlStateX	Out	Dint	The alarm state for condition X. Valid if AEConfigX = 1
EnableSupOut	In	Bool	Default true. When false, out parameter X is false when XEnabled is false. This parameter is earlier introduced for a selection of objects and is kept for compatible reasons only.

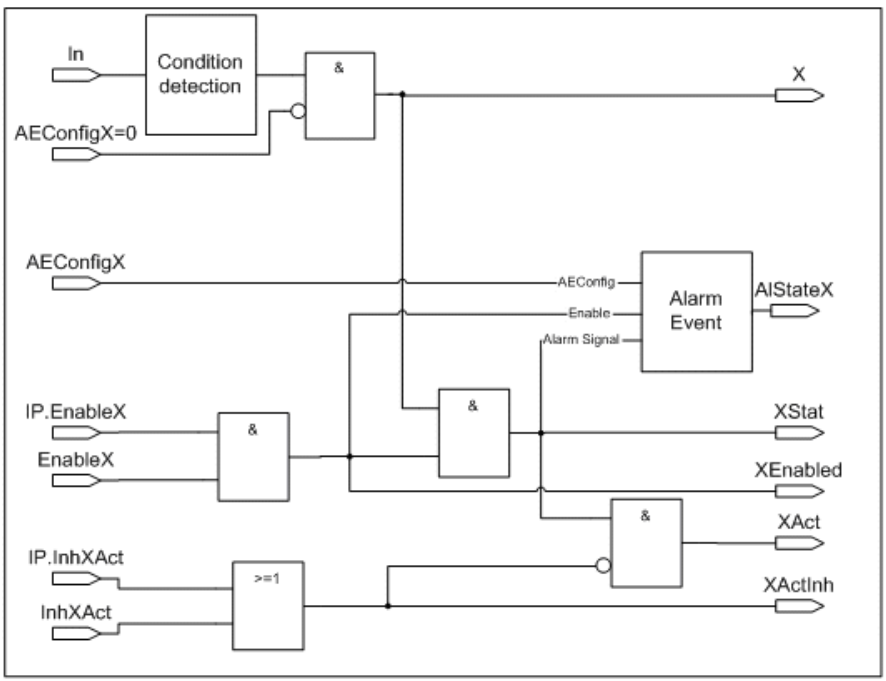


Figure 32. Alarm and event logic.

Source name

The source name parameter of the AlarmCondBasic instance shall not be connected to a parameter. By doing so, the Name parameter of the closest alarm owner will be used as source name. It is therefore important that all objects with alarms have a Name parameter. This Name parameter shall be assigned an unique value to have the alarm objects to work properly.

Enable and Disable alarm from alarm list vs. Faceplate

Disable from the faceplate means disabling of XStat and XAct as well as disabling of alarm/event. This means that the disabled alarm will be visible in the alarm list after a Disable from faceplate or code. The standard library objects should monitor the Alarm State and disable the alarm if the user then tries to enable the alarm from the alarm list. It is also possible to disable an active alarm from the alarm list. Doing so will however not influence the XStat or XAct outputs.

NLS for alarm and event

The NLS translator for alarms is discussed in sub-section National Language Support (NLS) on page 75.

The condition name for the alarms shall be NLS-treated. This is done by assigning an NLS-string as default value of the CondNameX parameter.

The parameter Message of AlarmCondBasic shall be connected to a string variable, called MessageAlarmCondition. The value of MessageAlarmCondition shall describe what has happened. If an alarm limit exceeded, the value of the limit shall be included. The value shall be NLS handled. Example:

```
||SL_Greater_than_{1}_{2}\5\Degrees
```

which gives the following result in English “Greater than 5 Degrees”. See National Language Support (NLS) on page 75.

Program Stop Complication

Just before the running program is stopped during a new download all the asynchronous procedures return -15 as error code (cErrProgramStopping, the requested operation is rejected because application program shutdown is in progress). This is a highly probable error code, which the programmer has to take

into consideration, when constructing the code of a control module or Function Block.

Three alternatives exist:

1. Accept the error code and return failure back to the caller.
2. Ignore the -15 error code and return success back to the caller.
3. Retry the asynchronous procedure when error code is -15 until success or another error emerges.

Which one of the alternatives to choose depends on the application. Often the only solution is to pass on the problem to the user in terms of an extra module parameter. In this case, an integer in the parameter ErrorHandling should be introduced with the above possibilities and a default value of 1.

Power Failure Behavior

Outputs shall be ramped when the application is restarted after a power failure and if an OSP value has been used.

State algorithms and bumpless parameters changes

Some parameter changes in algorithms that hold a state might cause a “bump” in the output if the update of parameters are not done with care. An example is change of filter time or gain for a filter algorithm. To avoid such bumps, the following method should be used.

The parameter values from the last scan is stored and used for the calculation for the new state. The output is computed and the last thing that is done is that the state is adjusted to the value that would have given the same output with the new parameters. The pseudo code below exemplifies the method:

```
NewState := UpdateState(OldState, In, SampleTime,
OldParameters);
Out := CalculateOut(NewState);
OldState := UpdateStateInverse(Out, In, SampleTime,
NewParameters);
OldParameters := NewParameters
```

Range Check

A *real*, *integer*, *data type,* or *time* input parameter may have a range. This means that it may be required to have a relation (for example > or <) to one or several constants or other parameters.

Generally, the object must have a predictable behavior for out-of-range input values. One example can be to use the closest range border value for internal calculations. What action that is most reasonable when out-of-range values are detected has to be decided from case to case.

If any real, integer, or time input parameter is out-of-range a graphical indication shall be visible in the faceplate and interaction window. In addition, a bool output parameter ParError shall be set to true (see also sub-section ParError on page 37).

Table 14. Example, Function Block Type

Name	Data type	Attributes	Direction	Initial value	Description
ParError	bool	Retain	Out		Indicates parameter range error.

The parameter description shall state the range, the action for valid input values, and the action for out-of-range values.

Table 15. Example, Function Block Type

Name	Data type	Attributes	Direction	Initial value	Description
AEConfig	dint	Retain	In	1	Config (0=None, 1=Alarm, 2=Event, 3=Event1, 4=Indication, Else Alarm + ParError)

Exception: A ParError parameter (or other parameters) shall not be added to types where the parameter interface is defined by IEC 61131-3, for example TOn.

Example of range test on AEConfig:

```
(* Parameter range test *)
ParError := false;
IF AEConfig < 0 or AEConfig > 4 THEN
   AEConfigInt := 1;
   ParError := true;
ELSE
   AEConfigInt := AEConfig;
END_IF;
```

After this, the variable *AEConfigInt* is used in the code and graphics since the input parameter AEConfig can not be modified.

When an input parameter is used in the code, the range check shall be made there. If the parameter is used in an internal FB, the range check shall be made there as well. In the main object the ParError should be calculated as the sum error of the ParError of the block itself (if any) and the ParError(s) of the sub function block(s).

Example:

Test of *AEConfigInt*, *InteractionPar.AEFilterTime* and *InteractionPar.AEHysteresis* have been tested as described above.

AEClass is tested in the function block Level6Alg.

```
Level6Alg(AEFilterTime := InteractionPar.AEFilterTime,
          AEClass := AEClass,
          AEHysteresis := InteractionPar.AEHysteresis,
          AEConfig := AEConfigInt);
ParError := ParError or Level6Alg.ParError;
```

This principle may be used in all levels of function block calls.

Conditional Range Check

If a conditional range check is applied, the execution time can be reduced in Non-SIL applications.

Table 16. Parameter interface for an example type with conditional range check and last good value

Name	Data type	Attributes	Direction	Initial value	Description
MyPar	dint	Retain	In	1	Config (<=0 = Event1, 1 = Event2, 2 = Event3, Else Last good value + ParErr)
EnableParError	bool	Retain	In	false	EDIT Enable par error calculation for Non-SIL applications.
ParError	bool	Retain	Out		Indicates parameter range error.

In the example code, the parameter **EnableParError** is copied during the first scan to a local variable **EnParError** for Non-SIL application, otherwise the local variable will be true. If the range check finds an out-of-range value, the last good value will be used.

```
if Init then
   Init := false;
   (* Force range checks if SIL application *)
   EnParError := EnableParError or GetApplicationSIL() <> 0;
   MyParInt:= MyLimit; (* Last good value *)
   ParErrorInt := false;
end_if;

if EnParError then
   (* Perform parameter range check each scan *)
   (* before assignment *)
   ParErrorInt := false;
```

```
   if MyPar > MyLimit then
      (* No assignment, use last good value *)
      ParErrorInt := true;
   else
      MyParInt := MyPar;
   end;

   (* More range checked assignments ... *)
else
   (* Assignment without range checks *)
   MyParInt := MyPar;
   (* More assignments ... *)
end_if;
```

Note that **MyPar** is copied to a local variable because it is not allowed to change IN parameters.

Overflow handling

The object must have a predictable behavior when calculating output parameter values. Situations like divide by zero and sqrt() of negative numbers shall be prevented or handled in the code.

Example:

Create a function Y := (1/T)*G, where T and G are parameters to a function block.

Solution:

When T is too small, the calculation generates an overflow in the arithmetical processing unit. To avoid this, the parameter T must be checked. As T is assumed to be a time, only positive and not too small values are allowed.

Code:

```
IF T < Ts THEN
   Y := G / Ts;
ELSE
   Y := G / T;
END_IF;
```

Where Ts is the selected sample time for the running task.

The requirement is that the library object shall not give an overflow output value if none of the input parameters have an overflow value.

SIL Mark Restrictions

Objects need to be restricted marked if any of the below is true:

- The object calls the functions that are not fully executed in the SM81x safety module.
- The output data from an unsafe source is not secured with a safety layer.

This is only relevant for 800xA internal libraries.

To obtain the main functionality of the object in the SIL environment, the whole object must be SIL Restricted.

If the object is set to SIL1-2 or SIL3, but the parameters are Non-SIL, then it is possible to obtain partial functionality in the SIL environment.

Appendix A Names and Abbreviations

Suggested Names

The recommendations consist of the following:

- Recommended Names and Abbreviations on page 109 lists recommended names and abbreviations for a number of common types and parameters.
- Standard Library Parameters on page 113 lists common parameters used in standard libraries. These are included for reference, since it is necessary to know about them when naming additional types and parameters when creating self-defined types.

Recommended Names and Abbreviations

If you do not have a naming strategy for types and parameters, the below table offers a good start.

Table 17. Names and recommended abbreviations of type and parameter names.

Full name	Short name	Remarks
Acknowledge	Ack	
Acknowledged	Ack	
Active	Act	
Activated	Act	
Alarm	Al	
Asynchronous	Async	
AutoDisabled	AutoDis	

Table 17. Names and recommended abbreviations of type and parameter names.

Full name	Short name	Remarks
Automatic	Auto	
Boolean	Bool	
Busy	Busy	
Cascade	Casc	
Channel	Chan	
Command	Cmd	
Communication	Comm	
Condition	Cond	
Connect	Conn	
Connected	Conn	
Connection	Conn	
Control	Ctrl	
Cyclic	Cyc	
Decrement	Decr	
Delay	Delay	
Description	Descr	
Detection	Dct	
Deviation	Dev	
Device	Dev	
Disable	Dis	
Distribute	Distr	
Duration	Dur	
Elapsed	Elap	

Table 17. Names and recommended abbreviations of type and parameter names.

Full name	Short name	Remarks
Enable	En	
Error	Err	
External	Ext	
Feed back	FB	
Feed forward	FF	
Fieldbus Foundation	FF	
Filter	Filt	
Force	Force	Do not abbreviate
Function block	FB	
Gain Scheduling	GS	
Hysteresis	Hyst	
In	In	Do not abbreviate
Integer	Int	
Inactive	Inact	
Increment	Incr	
Inhibit	Inh	
Interaction	Iact	
InteractionPar	InteractionPar	Do not abbreviate
Interlock	Ilock	
Limit	Lim	
Local	Loc	
Manual	Man	
Master	M	

Table 17. Names and recommended abbreviations of type and parameter names.

Full name	Short name	Remarks
Memory	Mem	
Message	Msg	
Mode	Mode	Do not abbreviate
Negative	Neg	
Number Of	NoOf	
Object	Obj	
Occurrence	Occ	
Out	Out	Do not abbreviate
Panel	Pan	
Parameter	Par	
Periodic	Periodic	Do not abbreviate
Positive	Pos	
Preset value	PV	61131-3 standard
Print	Prt	
Process value	PV	
Pulse	Pulse	
Ready	Rdy	
Reference	Ref	
Real	Real	
Relative	Rel	
Request	Req	
Reserve	Rsv	
Reset	Rst	

Table 17. Names and recommended abbreviations of type and parameter names.

Full name	Short name	Remarks
Select	Sel	
Setpoint	Sp	
Signal	Sig	
Simple	Smp	
Simulate	Sim	
Slave	S	
Source	Src	
Start	Start	Do not abbreviate
Status	Stat	
Stop	Stop	Do not abbreviate
String	Str	
Supervision	Sup	
Synchronous	Sync	
Support	Sup	In library names
Time	T	
Unacknowledged	Unack	
Update	Upd	
Value	Val	

Standard Library Parameters

The following list contains a number of important parameters that are used in the standard libraries. Please note that there are additional parameters for many Function Block- and Control Module types.

Table 18. Certain parameters and their descriptions used in standard libraries.

Parameter name	Description
Name	The name of the object. This name should appear in the window title of all the operator windows. It should also appear when the module is zoomed to layer 2. If the module has graphical connections, the name should be presented together with the node names.
Execute	Requests the control module or the diagram type to execute. The parameter is reset before next scan.
Req	Request for execution
Status	Status of the command set in Execute (M), (D), or Req (FB). The status can be set several scans after the command was set. 0 means that the module is pending, 1 means ready and OK, <0 means error.
Warning	True at a warning of unexpected operation. Status parameter >1.
Error	Indicates an error with True. Status parameter <0.
Ready	Indicates that the command was executed successfully.
Done (FB)	True when requested operation is performed successfully
NDR (FB)	True when new data has been received on each call after successful operation.
Enable	Enables the execution of the object code while True.
Enabled	Indicates that the function is activated. This is not affected by error status or warning status.
Valid	True when the output values are valid.
In, In1, In2, …	Main inputs, if there are no other natural names.
Out, Out1, Out2, …	Main outputs, if there are no other natural names.

Table 18. Certain parameters and their descriptions used in standard libraries.

InteractionPar	A parameter of type *ObjectNamePar*. The components are all the parameters that the operator can interact with via interaction objects. Via this parameter, all operator parameters are available in the surrounding program.
Period (FB)	Time interval between consecutive operations.
RemoteSystem	The address of a remote system.

In addition, the following abbreviations are recognized:

Table 19. Object name suffixes.

Full name	Short name	Remark
ControlConnection	CC	Used as suffix on all modules that have a parameter of the data type *ControlConnection*. For example: *PidCC*.
Control Module	M	A control module type that has the same functionality as an existing function block type shall have the same name as that function block type plus the suffix “M” (for Control Module type). For Example: *AlarmCond* – Function block type *AlarmCondM* – Control module type
Diagram	D	A diagram type that has the same functionality as an existing function block type or control module type shall have the same name as that function block type or control module plus the suffix "D" (for Diagram type). For Example: RemoteInput Control module type and RemoteInputD Diagram type
Template	Template	Only for object types that the user must change and rename before use. For example: *EquipProcedureTemplate*

INDEX

A

Access Level
    Confirm 77
    Read-Only 77
AEConfig 97
Area color 48
Aspect Object 72

B

basic icons 49
Bool 23
by_ref 28

C

CC 23
class 39
CMD 20
code block 91
cold retain 27
Command icons 50
condition name 39
constant 28
control module type
    graphical connections 44
    layers 43
    visible 43
    zoomable 43
control module type icons 49
Core 23
cWindowSizeFactor 66 to 67

D

D 23
Dimming 70
Dint 23
direction 38
Display Elements
    Icon 74
    Reduced Icon 74
    Tag 74

E

error display 97
ErrorIcon 50

F

Function Block Diagram 20

G

graphical nodes 46

H

hidden 28, 75

I

IEC 61131-3 22
IEC 61131-5 22
Information windows 63
internal coordinate system 46

K

Keyword 92
keyword 32

L

Line color 48

M

M 23

Maneuver icons 50
ManoeuvreIcon 65
MaxSize 44
Mode icons 50

N

Name Upload 33 to 34
NLS-strings 75
nosort 27

P

Project Constants 88
Protection 42

R

range 37
Real 23
Reset Shape 46
retain 27

S

Scope
    Private 88
    Public 88
severity 39
Source Name 34
string literal 90
structured data type 36
Structured Text 85

T

Template 23

V

value default 31
Visibility 47

W

Write Permission
Administrate 76
Operate 76
Tune 76

Contact us

www.abb.com/800xA
www.abb.com/controlsystems

3BSE042835-600

Power and productivity
for a better world™