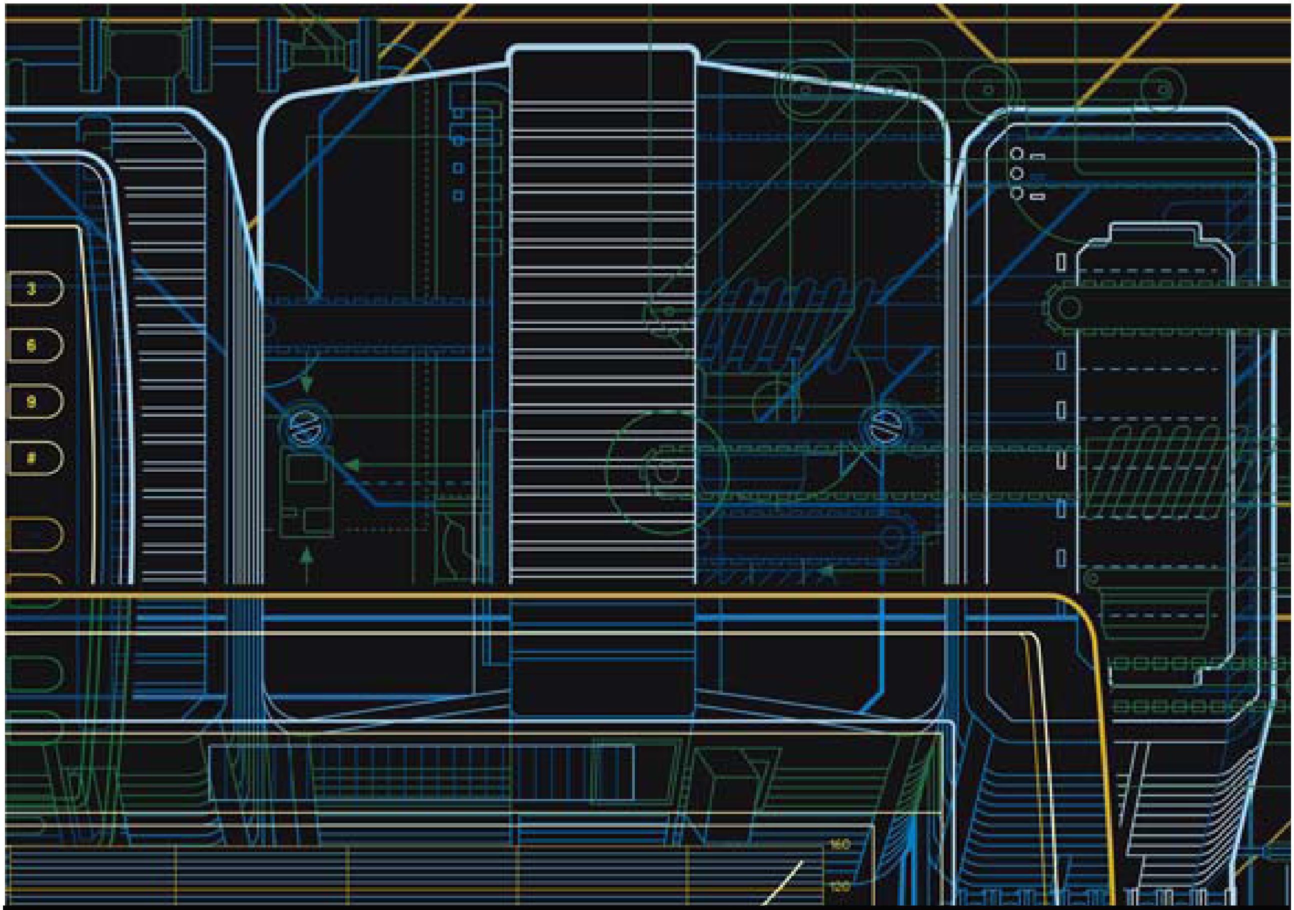


Compact Control Builder AC 800M

Binary and Analog Handling

Version 5.1.1

Power and productivity
for a better world™

Compact Control Builder AC 800M

Binary and Analog Handling

Version 5.1.1

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: April 2013
Document number: 3BSE041488-511

Table of Contents

About This User Manual

Section 1 - Introduction

Section 2 - Libraries

Section 3 - Standard Libraries

Section 4 - Analog Process Control

Section 5 - Binary Process Control

Section 6 - Synchronized Control

Section 7 - Surveillance

Appendix A - Customized Online Help

Appendix B - Library Objects Overview

INDEX

About This User Manual

General

This manual describes how to create re-usable automation solutions using the ABB standard libraries that are supplied with Compact Control Builder. It is a continuation of the two manuals *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*, and *Compact 800 Engineering Compact Control Builder AC 800M Planning (3BSE044222*)* , which describes basic configuration and programming functions that are accessed through the Project Explorer interface.

The libraries described in this manual conform to the IEC 61131-3 Programming Languages standard, except for control modules and diagrams, which are an extension to this standard.

This manual is organized in the following sections:

- Section 1, Introduction, a short introduction to the contents of the manual.
- Section 2, Libraries, which describes the concepts of using libraries when developing automation solutions.
- Section 3, Standard Libraries, which contains a general description, usage, and common properties of the libraries delivered with the system.
- Section 4, Analog Process Control, which describes creating control loops and other analog control functions using the Control libraries.
- Section 5, Binary Process Control, which describes creating binary control solutions using the Process Object and Signal Object libraries.
- Section 6, Synchronized Control, which describes how creating start and stop sequences for different automation solutions using the Group Start Library .
- Section 7, Surveillance, which describes the Signal and Vote Loop Concept.

User Manual 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 User Manual 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 that could result in *electrical shock.*

Warning icon indicates the presence of a hazard that 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 that 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

The following is a list of terms associated with Compact Control Builder. You should be familiar with these terms before reading this manual. The list contains terms and abbreviations that are unique to ABB, or have a usage or definition that is different from standard industry usage.

Term/Acronym	Description
(M)	(M) is used to refer to function block type and a control module type with similar functionality, for example, MotorBi(M).
Application	Applications contain program code to be compiled and downloaded for execution in a controller. Applications are displayed in Project Explorer.
Control Builder	A programming tool with a compiler for control software. Control Builder is accessed through the Project Explorer interface.
Control Module (Type)	A program unit that supports object-oriented data flow programming. Control modules offer free-layout graphical programming, code sorting and static parameter connections. Control module instances are created from control module types.
Industrial^{IT}	ABB’s vision for enterprise automation.
Interaction Window	A graphical interface used by the programmer to interact with an object. Available for many library types.
MMS	Manufacturing Message Specification, a standard for messages used in industrial communication.
OPC/DA	An application programming interface defined by the standardization group OPC Foundation. The standard defines how to access large amounts of real-time data between applications. The OPC standard interface is used between automation/control applications, field systems/devices and business/office application.
Process Object	A process concept/equipment e.g. valve, motor, conveyor or tank.

Term/Acronym	Description
Project Explorer	The Control Builder interface. Used to create, navigate and configure libraries, applications and hardware.
Type	A type solution that is defined in a library or locally, in an application. A type is used to create instances, which inherit the properties of the type.

Section 1 Introduction

Extended Control Software

This manual describes Extended Control Software. The term ‘extended’ comes from the fact that standard libraries that are not integrated with or based on AC 800M firmware can be seen as extensions to Compact Control Builder, and to AC 800M control software, see Figure 1.

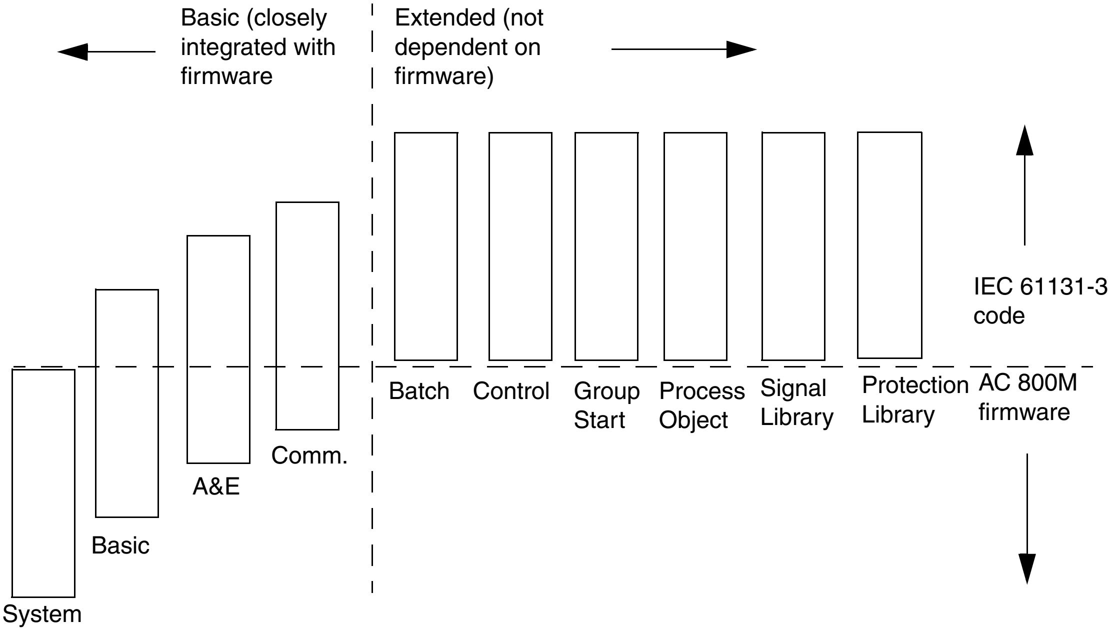


Figure 1. Basic and extended control software - standard libraries (some support libraries are not included)

Functions and types belonging to the Basic part of the system are described in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*).*

This manual contains the following sections and supporting appendixes.

- Section 2, Libraries discusses the considerations when to create own library structure, and using the standard libraries that are installed with Compact Control Builder.
- Section 3, Standard Libraries, contains an overview of the AC 800M standard libraries. It also explains the use of templates.
- Section 4, Analog Process Control, describes the types of the Control libraries and the methods to build control loops using these types.
- Section 5, Binary Process Control, describes the types of the Process Object libraries and the methods to build process object control solutions using these types.
- Section 6, Synchronized Control, gives a short introduction to the Group Start library.
- Section 7, Surveillance, describes the Signal and Vote Loop Concept.
- Appendix A, Customized Online Help, describes the requirements for creating help for the libraries and applications.
- Appendix B, Library Objects Overview, provides an overview of all library objects.

Libraries and Types

A library is a collection of type solutions, which can be connected to applications and other libraries, so that they can create instances from those types. However, the Library>Hardware folder, in Project Explorer, contains libraries with hardware types, which can be connected to controllers only. Libraries and types are discussed in detail in Section 2, Libraries.

Throughout this manual, there are two library categories:

Standard libraries are installed with Compact Control Builder. User-defined libraries are created to store own type solutions, so that they can be re-used.

Section 2 Libraries

Introduction

This section describes the library concept, as implemented in the Compact Control Builder. This section contains:

- An introduction (this subsection), which introduces some important concepts like type and library, different kinds of types and their intended use, and the appropriate use of type solutions and libraries.
- Advantages of Using Libraries and Types on page 25, which contains a summary of the advantages of using types and libraries as a basis for all automation system development.
- Building Complex Solutions With Types on page 27, which describes the three different basic ways of building automation solutions using types from the standard libraries. This topic covers important concepts such as templates.
- Library Management on page 31, which describes planning the library structure and maintaining the libraries over a longer period of time, including version handling. It also describes some risks and potential problems that the user must be aware of.
- Library Command Summary on page 36, which lists useful actions and commands when working with libraries.

Libraries and Types

A library is a collection of types, which can be connected to applications and other libraries, so that they can create instances from those types. However, a library in the Hardware folder contains hardware type(s), which can be connected to controller(s) only. In such a case, only the controller(s) can create instances of hardware types.

For a detailed discussion of types and objects, refer to the *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* manual.

Types Defined in Applications and Libraries

A type is a solution to a small or big automation problem. A type can be a simple counter or a complete control loop. It is defined in an application, or in a library.

Types contain instances of other types. These instances are often referred as *formal instances*. Types from another library can be used when building types, as long as this library is connected to the application and the library.

A type is used to create instances in an object oriented manner. Each instance points to the type on which it is based. When an instance is executed, the code stored in the type is executed every time. The variables and other data are instance-specific.

Since the same code is executed in all instances and the instances inherit the properties from their type:

- Re-use is made possible and this makes the automation solution flexible, since the behavior of many instances can be changed by changing the type.

The changes made to a type affect *all* instances.

- The memory consumption for each instance is smaller, compared to the memory needed to execute the type itself. For example, a MotorUni function block type consumes about 65 kB, while each additional instance only needs another 12 kB.

If types are created on a very high level and there is a need to change something for a particular object, this change affects other objects of the same type.

This can partly be solved by including copies of certain types (as some types in these libraries are templates, that is, they can be copied to the libraries and applications and the code can be modified to suit a particular process). These copies can then be changed without affecting the corresponding part of other types. However, these copies result in increased memory consumption, as well as create problems when upgrading types and libraries.

Hardware Types

Hardware types represent the physical hardware units and communication protocols that can be added to AC 800M. It can be a CPU unit, a communication interface or an I/O unit (see Hardware Overview on page 52). Hardware types can be defined in libraries only.

The following are the advantages when hardware types are organized in libraries:

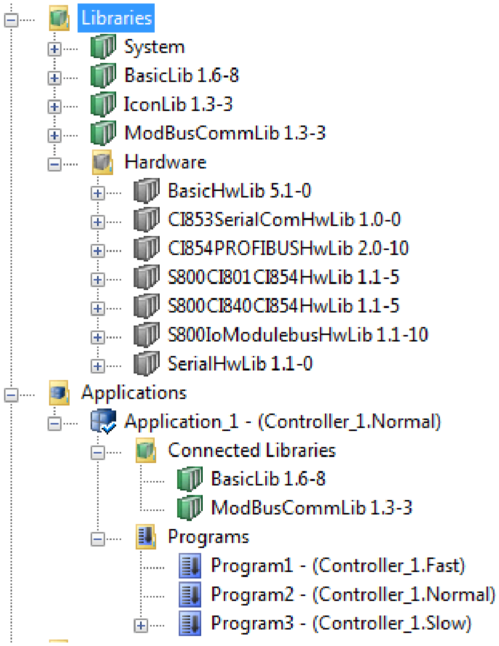
- Easy to upgrade to newer system versions
- Allows a new version of a hardware type to coexist with an older version (but in different versions of the library).
- Allows new library versions to be delivered and inserted to the system.
- Ensures that only used hardware types in controller configuration(s) allocate the memory in the system.

A hardware type contains a hardware definition file, which is the source code of the hardware unit. Changing and replacing a type in a library affects all instances of the hardware unit. For example, changing a hardware type of an I/O unit that is used in more than one positions in a controller, affects all positions where the I/O unit is used in the hardware tree (I/O connections and parameters may be incorrect).

The changes made to a hardware type affect all instances.

Libraries

A library is a collection of objects. Libraries are presented as objects in Project Explorer (Figure 2)



- Libraries
  - System
  - BasicLib 1.6-8
  - IconLib 1.3-3
  - ModBusCommLib 1.3-3
  - Hardware
    - BasicHwLib 5.1-0
    - CI853SerialComHwLib 1.0-0
    - CI854PROFIBUSHwLib 2.0-10
    - S800CI801CI854HwLib 1.1-5
    - S800CI840CI854HwLib 1.1-5
    - S800IoModulebusHwLib 1.1-10
    - SerialHwLib 1.1-0
- Applications
  - Application_1 - (Controller_1.Normal)
    - Connected Libraries
      - BasicLib 1.6-8
      - ModBusCommLib 1.3-3
    - Programs
      - Program1 - (Controller_1.Fast)
      - Program2 - (Controller_1.Normal)
      - Program3 - (Controller_1.Slow)

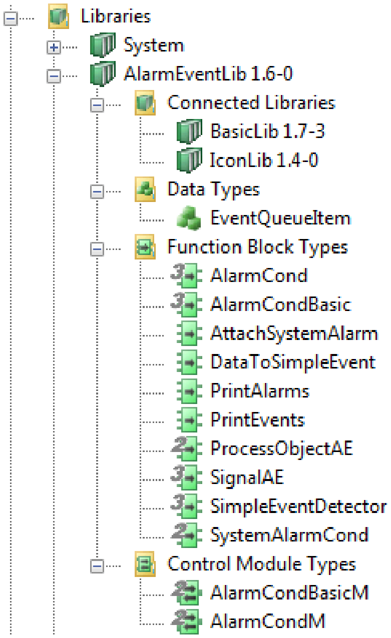
Figure 2. Libraries in Project Explorer

For a library to appear in Project Explorer, it must be added to the Libraries folder of the project (see Figure 2). Right-click the Libraries/Hardware folder and select **Insert Library**.

For a summary of useful library commands, see Library Management on page 31.

A library in the Libraries folder in Project Explorer may contain the following (see Figure 3):

- Data types
- Diagram Types
- Function block types
- Control module types
- Project constants (not shown in Figure 3).

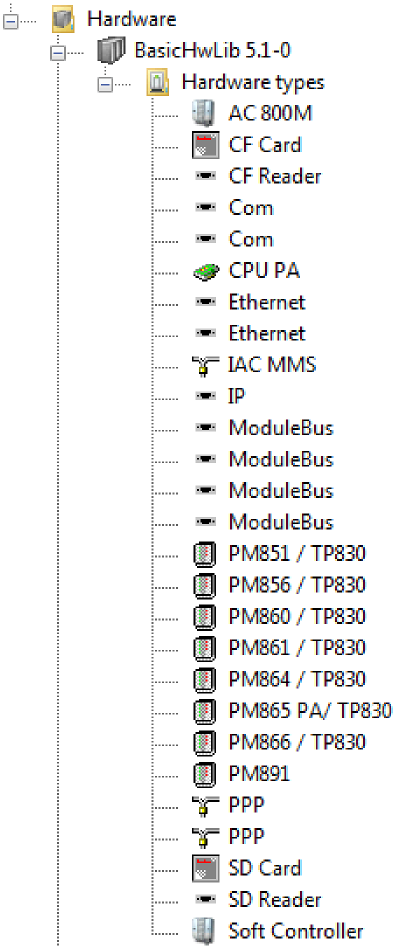


- Libraries
  - System
  - AlarmEventLib 1.6-0
    - Connected Libraries
      - BasicLib 1.7-3
      - IconLib 1.4-0
    - Data Types
      - EventQueueItem
    - Function Block Types
      - AlarmCond
      - AlarmCondBasic
      - AttachSystemAlarm
      - DataToSimpleEvent
      - PrintAlarms
      - PrintEvents
      - ProcessObjectAE
      - SignalAE
      - SimpleEventDetector
      - SystemAlarmCond
    - Control Module Types
      - AlarmCondBasicM
      - AlarmCondM

Figure 3. Contents of a library in the Libraries folder

A library in the Hardware folder contains (see Figure 4):

- Hardware types (*.hwd files)
- Device capability description files (for example*.gsd files)



- Hardware
  - BasicHwLib 5.1-0
    - Hardware types
      - AC 800M
      - CF Card
      - CF Reader
      - Com
      - Com
      - CPU PA
      - Ethernet
      - Ethernet
      - IAC MMS
      - IP
      - ModuleBus
      - ModuleBus
      - ModuleBus
      - ModuleBus
      - PM851 / TP830
      - PM856 / TP830
      - PM860 / TP830
      - PM861 / TP830
      - PM864 / TP830
      - PM865 PA/ TP830
      - PM866 / TP830
      - PM891
      - PPP
      - PPP
      - SD Card
      - SD Reader
      - Soft Controller

Figure 4. Contents of a library in the Hardware folder

If a type from a library is used, it has to be connected to the application, library or controller (libraries with hardware types) in which the type is used.

Right-click the Connected Libraries folder for the application, library or controller in question, and select **Connect Library**. If the library does not show up in the list, it must be added to the Libraries folder of the project.

Advantages of Using Libraries and Types

The advantage of using type solutions in the automation system is enhanced if the types are organized in libraries. This is true if the organization is big and develops automation solutions for a number of plants and processes. The following two subsections provide a summary of the advantages of using types and libraries.

Type Solutions

Use type solutions whenever an automation solution contains a large number of similar objects that perform similar functions, but in different locations or contexts.

Before programming the automation solution, identify the types needed, by considering the following:

- The parts of the plant that are likely to change. Typically, something might be added to a production line, or another production line might be added.
- The objects that can be variations on a theme (instances of a certain type solution). Typically, this would be objects such as motors, tanks, valves.
- The objects that correspond to the types already contained in the standard libraries that are installed with Compact Control Builder. If such objects are identified, configure them for use in the particular environment.
- The situations where one object changes, while all other similar objects remain the same.
- The standard libraries as well as the self-defined libraries might be upgraded, which causes problems in running applications.

The benefits of using type solutions are:

- Re-usable solutions save development time, as well as memory.
- Well-tested type solutions increase the reliability of the automation system. It is also easier to test a system that is based on type solutions.
- It is possible to change the type in one place and this affects all instances, instead of having to make the same change for many identical objects.

Libraries

Well-defined libraries provide the following advantages:

- All automation engineers within the organization have access to the same type solutions. This saves the development time and results in consistent automation systems.
- The knowledge of experienced programmers and specialists can be packaged and distributed to all automation engineers through libraries.
- A common standard can be implemented via the libraries (for example, a name standard).
- Complex solutions can be built with a high degree of control by using library hierarchies.
- A large number of applications can be updated by updating a library.
- Version control of standard libraries makes it possible to upgrade some applications, without affecting other applications. This applies if the major version number is changed between the both library versions. This is made only in cases when the compatibility is broken and the library guids are different.

Building Complex Solutions With Types

As mentioned in the introduction to this section, there are two basic ways to build automation solutions from the types in the standard libraries:

- Use ready-made types that only have to be configured and connected to the environment.
- Use template types that are modified to fit the process requirements. Using a template also requires adding functions by using other objects from the standard libraries, or by writing code. See Templates on page 58.

When you cannot find a ready-made type or a template that fits your needs, then you must build your own solution. A complex type or application-specific solution can be built using a number of types from the standard libraries, together with your own types. In some cases, the best option is to use a template and then add functions to it.

An example of a Complex solution, where types from the standard libraries are used as building blocks, is a cascade control loop.

The cascade loop in Figure 5 is an example of a complex solution. A similar cascade loop can be found in the Control Solution library.

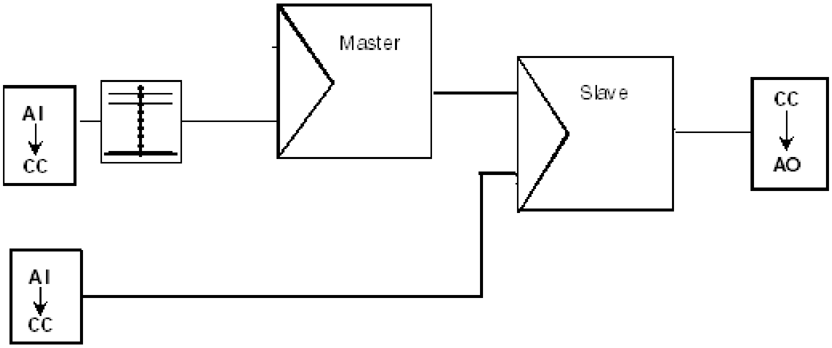
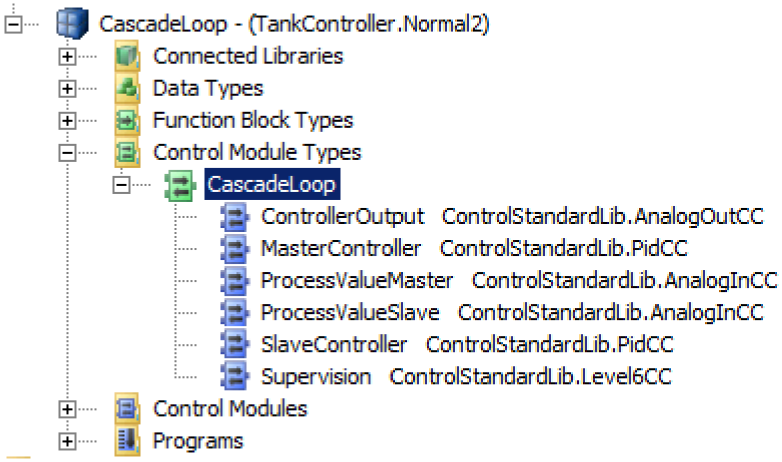


Figure 5. Cascade loop built from control modules

The cascade loop contains two control modules of type PidCC, one used as master, and the other used as slave (Figure 5). The input consists of two AnalogInCC control modules and one Level6CC control module. The output consists of an AnalogOutCC control module.



- CascadeLoop - (TankController.Normal2)
  - Connected Libraries
  - Data Types
  - Function Block Types
  - Control Module Types
    - CascadeLoop
      - ControllerOutput ControlStandardLib.AnalogOutCC
      - MasterController ControlStandardLib.PidCC
      - ProcessValueMaster ControlStandardLib.AnalogInCC
      - ProcessValueSlave ControlStandardLib.AnalogInCC
      - SlaveController ControlStandardLib.PidCC
      - Supervision ControlStandardLib.Level6CC
  - Control Modules
  - Programs

Figure 6. Cascade loop in Compact Control Builder

The heart of all automation solutions is the actual control of the process and the equipment. These types can be found in the Control, the Process Object and Supervision libraries.

However, there are a number of supporting libraries, which can be used to create specific solutions for part of the system, or to add functions by using other function blocks or control modules:

- **Signal Handling**
  Signal handling types are not only found in the Basic and Signal libraries, but also inside the Control and Process object libraries. It might, for example, be necessary to add a selector if there are several input signals to choose from. Other examples of signal handling objects that might be added are limiters and filters, if the input signals are outside the desired range or contain undesired components.

- **Alarm and Event Handling**
  The Alarm and Event library contains a number of types that can be added for alarm and event handling. These types can interact with existing alarms, or can be added as a separate alarm function.
- **Communication**
  Objects from the Communication libraries can be added to establish communication with other applications or even to other controllers. Communication variables may be used to establish communication between applications in the same controller where the communication variable is created, and also between applications in other controllers even if that controller is outside the actual configuration. Communcation variables may be used in top level Diagrams, Programs and top level Single Control Modules.

In addition to the types in the standard libraries, you can also define your own types, both on a higher level and low-level objects.

Most low-level objects are already available as types in the Basic library and via system firmware functions. Before designing new types, ensure that there is no similar type or function that fulfills the needs.

An application can be based on a mix of types from standard libraries, self-defined types from your own libraries, and locally defined types. See Figure 7 for an example.

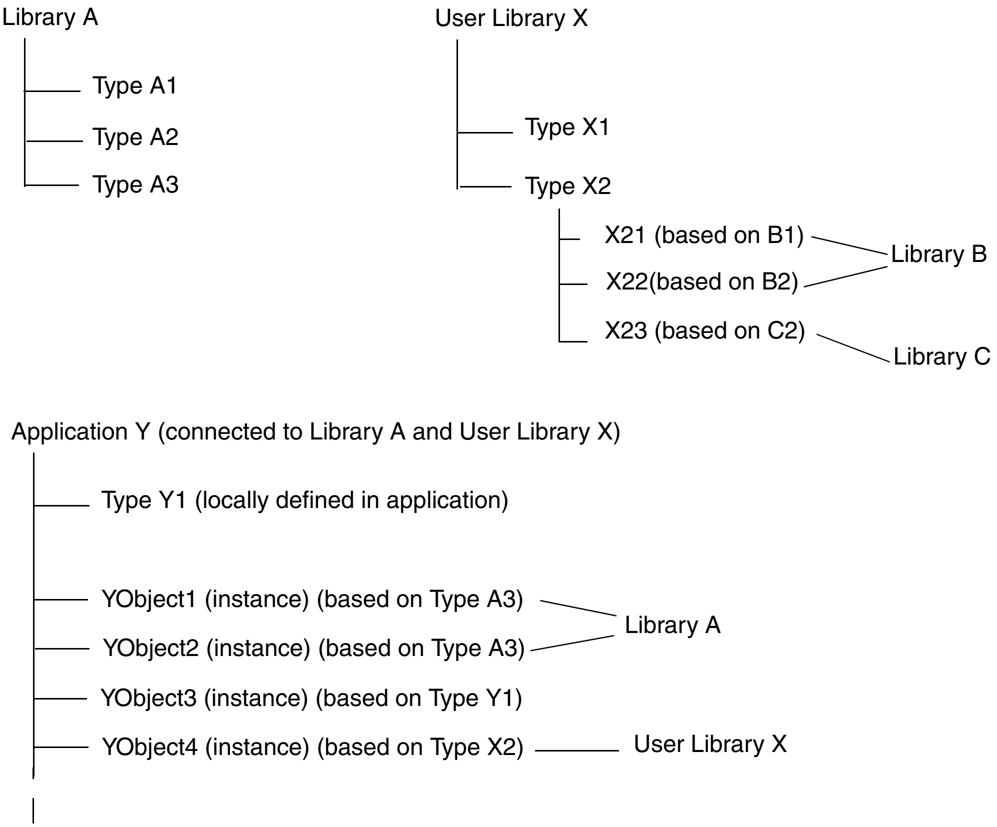

Library A
Type A1
Type A2
Type A3
User Library X
Type X1
Type X2
X21 (based on B1)
X22(based on B2)
X23 (based on C2)
Library B
Library C
Application Y (connected to Library A and User Library X)
Type Y1 (locally defined in application)
YObject1 (instance) (based on Type A3)
YObject2 (instance) (based on Type A3)
Library A
YObject3 (instance) (based on Type Y1)
YObject4 (instance) (based on Type X2)
User Library X

Figure 7. Building complex solutions based on standard libraries, self-defined libraries, and locally defined types

Library Management

When working with libraries, it is important to consider the following:

- Version handling
- Connection
- Change to a certain library

It is not possible to change library version of a library in the Libraries folder. However, library versions of libraries in the Hardware folder can be changed.

See also the manual *Library Objects Style Guide, Introduction and Design, (3BSE042835*)*.

The following important rules apply:

- Libraries may exist in different versions in the same project (only if the libraries have different GUID's). Different versions of libraries with hardware types may coexist in a controller, but it is not possible to connect libraries with different versions to the same application.

If you try to connect multiple versions of a library to an application, a warning dialog is displayed, showing the library and the versions that cause the problem. Multiple versions might occur through dependencies.

- Libraries in the Libraries folder may depend on each other in a number of layers.

Do not interconnect libraries unless it is absolutely necessary. There is always a risk of upgrade problems if there are many dependencies between libraries.

- Circular dependencies of libraries in that are used in applications are not allowed. Compact Control Builder checks and will warn you if you try to connect a library that create circular dependencies.
- Standard libraries cannot be changed by the user. This applies to both the library itself and the types inside.
- Self-defined libraries have three possible development states:
  - Open,
  - Closed,

  - Released.

- It is not possible to change a library with status Released. Make sure that a spare copy of the self-defined library is saved on local disc before changing to status Release.
- If a self-defined library has the status Open, it can be changed to Closed or Released.
- If a self-defined library has the status Closed, it can be changed to Open or Released.

Tips and Recommendations

The following list contains tips and recommendations intended to help you build an effective library structure and make good use of the standard libraries:

- All new libraries should have the suffix ‘Lib’, for example, ‘TankLib’.
- Libraries with hardware types should contain the suffix ‘HwLib’, for example, ‘S800ModulebusHwLib’.
- Libraries belonging to the same family should have a common prefix to their name, for example, ‘TankBasicLib’, ‘TankExtendedLib’.
- You can password protect your libraries, see Library Command Summary on page 36.
- All type names should follow the Control Builder naming standard and the IEC61131-3 standard.

For detailed information on naming conventions, see the manual *AC 800M Planning*.

- Short names are important for function block types since there is less space to show names in the Function Block Diagram (FBD) editor.
- Short names are important for function block types, control module types and diagram types since the names are displayed in the graphical code block of Diagram editor.
- When naming parameters, do not use very long names. This might have undesired effects in graphical displays.
- Use easy-to-understand and descriptive names.

- Avoid reserved names, such as IF, THEN, ELSE. See also Reserved Names on page 35.
- Make sure that descriptions for parameters provide the user with enough information. Also, see Parameter Keywords on page 35.
- Hide or protect objects that you do not want the user of your libraries to modify (or even see).

For detailed information on hiding and protecting types, see the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*.

- When creating data types, separate between two cases:
  - If a data type is closely connected to a certain type, store it in the same library as the type.

  - If a data type is used in many different types, and these types are stored in several libraries, there are two alternatives:

    a. Data types that are only used internally should be hidden and stored in a separate support library containing hidden types only. The name of the library should then include the word ‘Support’, for example, ‘TankSupportLib’.

    b. Data types that are used for parameters that are connected to other types and to the surrounding code should be stored in a common library containing visible types. The name of this library should then include the word ‘Basic’, for example, ‘TankBasicLib’.

- Simple function block types, control module types, and diagram types that are used as formal instances[1] in several complex types, in several libraries, should be placed in a separate support library (this library then has to be connected to all libraries where these types are used).
- Document your libraries. Use the Project Documentation function, see Where to Find Information About Standard Library Types on page 41.

[1] Formal instances are objects (instances of another type) that are located inside a type. Formal instances are executed when the objects based on the type are executed in applications.

Reserved Names

In addition to names reserved for use in code (IF, etc. see online help or *Compact Control Builder, AC 800M Planning (3BSE044222*)* Manual, the use of the following names is also reserved[1], and should be avoided for other purposes.

Table 1. Reserved names

Name	Description
Template	Use for templates only.
Core	Use for Core objects only.
HSI	Use for graphics calculation objects only.
Icon	Use for icons only.
Info	Use for interaction windows only.

Parameter Keywords

All parameters in control module types and diagram types, and all IN_OUT parameters in function block types, are recommended to contain an indication of its use in the corresponding Description field. The use is indicated by keywords, see Table 2. They must contain at least one of the first four keywords. The keyword should be placed before the following descriptive text, see Figure 8.

Table 2. Keywords for parameters

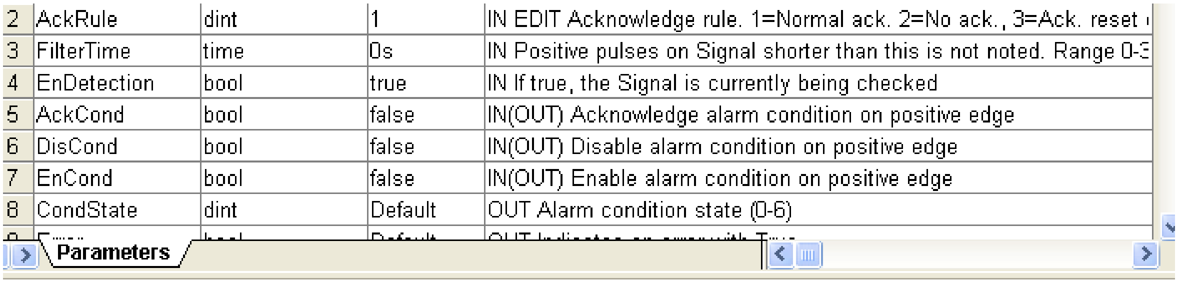
Keyword	Description (start parameter description with keyword)
IN	The parameter is only read.
OUT	The parameter is only written.
IN(OUT)	The parameter is both read and written, but mostly read.
OUT(IN)	The parameter is both read and written, but mostly written.

[1] These names are intended for use in instance names, for example, an UniCore instance would be called Core.

Table 2. Keywords for parameters (Continued)

Keyword	Description (start parameter description with keyword)
NONSIL	NONSIL is used in objects on output parameters where the output value originates from any internal restricted object.
NODE	Used when the parameter has a graphical connection node (control modules only).
EDIT	The value of the parameter is used the first scan after transition from Edit to Run mode without initialization. Online changes will not influence the executing code until a warm-start is performed.

Library Command Summary



2	AckRule	dint	1	IN EDIT Acknowledge rule. 1=Normal ack. 2=No ack., 3=Ack. reset
3	FilterTime	time	0s	IN Positive pulses on Signal shorter than this is not noted. Range 0-3
4	EnDetection	bool	true	IN If true, the Signal is currently being checked
5	AckCond	bool	false	IN(OUT) Acknowledge alarm condition on positive edge
6	DisCond	bool	false	IN(OUT) Disable alarm condition on positive edge
7	EnCond	bool	false	IN(OUT) Enable alarm condition on positive edge
8	CondState	dint	Default	OUT Alarm condition state (0-6)
9	Error	bool	Default	OUT Indicates an error with True

Figure 8. Keywords used in editor Description field

The following table is intended as a quick guide to library management. For detailed information on how to work with libraries, see the Control Builder online help and to the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*).*

Table 3. Library command summary

Action	**Command/Procedure**	**Comment**
Connect library to project	In Project Explorer, right-click the Libraries/Hardware folder and select **Insert Library**	
Connect library to application, library or controller	In Project Explorer, right-click intended Connected Libraries folder and select **Connect Library**	The library must have been inserted to the project.
Make spare copy of self-defined library (in Libraries folder)	In Project Explorer, right-click the library and select **Make Spare Copy**.	It is only possible to have one spare copy of each self-defined library.
Save copy of self-defined library (in Libraries folder)	In Project Explorer, right-click the library and select **Save Copy As...**	The self-defined library is saved in a file with a new name and a new directory with its source contents.
Change library state	In Project Explorer, right-click the library and select **Properties > State**, then set the new state	If the state is Released, it is only possible changing to state Open. In this case the index revision number is increased with one. Make sure that a spare copy of the self-defined library (in Libraries folder) is saved on local disc before changing to status Release.
Set protection for library	In Project Explorer, right-click the library and select **Properties > Protection**, then enter a password	If the library already is password protected, you must enter the old password before changing it.
Disconnect library from library, application or controller	In Project Explorer, go to the Connected Libraries folder, select the library and press Delete	If there are objects that use types from this library, a warning dialog is shown

Table 3. Library command summary (Continued)

Action	Command/Procedure	Comment
Remove library	In Project Explorer, go to the Libraries folder, select the library and press Delete	If there are applications or libraries that depend on this library, a warning dialog is shown
Library Usage	In Project Explorer, right-click the library and select **Library Usage**	The function shows if and where the library is connected to an application, library, or controller.

Section 3 Standard Libraries

Introduction

This section describes the standard AC 800M libraries, that is, the AC 800M libraries that are installed with Compact Control Builder.

- This part of the section, the Introduction, describes the different types in the standard libraries, including ready-made types, templates, and types intended to be used as building-stones in complex solutions.
- Library Overview on page 44 gives an overview of all standard libraries, with a short description of each.
- Hardware Overview on page 52 gives an overview of all libraries with hardware types.
- Templates on page 58 describes the template concept, and how to use template objects and libraries to create re-usable and flexible solutions.

The library concept and how to build and manage a library structure for your organization is described in Section 2, Libraries.

Ready-Made Objects, Templates and Building Stones

Standard AC 800M libraries contain:

- Ready-made objects that simply have to be connected to your environment to work. Typical examples are the simple control loops in the Control libraries, and some of the motor and valve objects in the Process Object libraries. See Standard Library Types on page 41.

- Objects that should be seen as templates. These template types are not protected and they can therefore be copied to your application, or to your own, self-defined library. They can then be modified to fit your specific requirements.

  In a template type, there are core functions that are protected. These core functions cannot be changed (with the exception of parameter connections), but you can add other functions, both by using other types from the standard libraries, and by adding code.

  Typical template objects are the objects in Control loop solution library. Other objects are Uni and Bi process objects in the Process Object Extended library, which can be used to build process control objects for any uni- or bi-directional object. See Group Start Library on page 47 and Control Libraries on page 48.

- Low-level objects that can be seen as building stones to be used for building more high-level, complex solutions. These objects can be used to add functions to an existing template, or to build a complex solution from scratch.

  Typical building stones are types for signal handling, which can be added to the output and input of, for example, control loops. See Building Complex Solutions With Types on page 27.

If the standard libraries do not contain any type that fits one of your specific requirements, you have two options:

- You can build your own type, based on objects from the standard libraries. If the type is application-specific, you can define it directly in the application. However, if it is likely that you in the future want to use it in other applications as well, then you should create a library and store your type solution in this library. Then, all you have to do to use the type in another application is connect the library to that particular application.

Say that you discover that you want to use a type in another application, but you have defined it in an application only. Then you should simply create a library and copy the type to this library (you can, of course, also copy it to one of your existing libraries). Then you can connect the library to all applications where you want to use the type and make sure that all instances refer to the library type. You can then delete the original type definition from your application.

- You can build your own type from scratch. This is not recommended, but might be necessary if you have a process with very specific requirements. In this case, it is strongly recommended that you store your types in a self-defined library.

Standard Library Types

When using ready-made types from the standard libraries, there are a number of things that you should know:

- When connecting parameters, the minimum requirement is that you connect the parameters that do not have default values.
- For control modules and diagram types, the information on individual parameters is given in the description field of the connection editor.
- For the in/out declared parameters in function blocks, the information on individual parameters is given in the description field of the connection editor.
- For more complex types, there is often additional parameter information in the corresponding online help file. Select the type and press F1 to display online help for a certain type.
- There is information for most ready-made types, both in manuals and in online help, see below.

Where to Find Information About Standard Library Types

For details on concepts, design and configuration for a specific type, there are several sources:

- How to use types from the Alarm and Event library and the Communication libraries is described in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*).*
- The Basic library and system firmware functions are also described in *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)..*

- The other standard libraries are all described in this manual:
  - For information on the Control libraries and the Signal library, see Section 4, Analog Process Control.
  - For information on the Process Object libraries, see Section 5, Binary Process Control.
  - For information on the Group Start Library, see Section 6, Synchronized Control.
  - For information on the Signal and Vote Loop Concept, see Section 7, Surveillance
- All libraries have a corresponding help file. Each object has context-sensitive help, which is accessed by selecting a type and pressing F1.
- It is also possible to generate project documentation for a library. In project Explorer, select the library and select **File > Documentation**. This will provide you with an MS Word file, containing short descriptions of all objects in the library, including a list of all parameter descriptions.

For more information on how to generate project documentation, see online help and the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*.

Common Properties

InteractionPar

Most function block types and control module types having an interaction window also have a parameter, called InteractionPar. This parameter is a structured data type with components where some of them have the attribute coldretain.

Things that can be done in interaction windows can also be done via the InteractionPar parameter from the surrounding application code. However, if no such code is implemented you should not connect the InteractionPar parameter (in the connection editor), just leave the connection field empty while using control modules. While using function blocks, you need to connect the InteractionPar parameter to a variable.

Consider InteractionPar as an option for connecting a local variable that can, from the application code, reach any of the components inside the InteractionPar parameter. But remember, connecting the InteractionPar to your code, means that you also take over the responsibility of handling coldretain values etc.

The main purpose of InteractionPar parameter is to manipulate values from graphics (interaction windows) only, thus not from code. Calling the InteractionPar (in code) will override any inputs given by the operator. The InteractionPar should be controlled by graphics, and only in exceptional cases from code.

Writing to components in InteractionPar must be done with care. If a user code continuously writes to it, the corresponding faceplate or interaction window entrance will be locked. Such writing need to be made only on an event to prevent the described unwanted behavior.

ParError

ParError parameter performs diagnostic tests inside an object in run-time mode. You recognize if an object contains ParError, by the output parameter ParError.

The parameter returns a Boolean output value if the object parameters are out of range'. The principle of the test on each such parameter is noted in the corresponding parameter description. For example, severity and class for alarms are wrong, or a high level input value is lower than the low level value in a level monitor module, etc. These are two common examples but basically it could test all kinds of input values.

The general idea of ParError is to provide you with a possibility to anticipate certain actions and handle them from your code. For instance one can call the Error handler function and perform a controller shut-down.

However, ParError requires some CPU load each time the diagnostics are executed. For that reason, all objects that contain ParError also may have an input parameter EnableParError which is set to false by default.

Library Overview

This part describes each standard library briefly. For a list of all types in a library and a short description of each type, see the Appendix B, Library Objects Overview.

In addition to the standard libraries, there are also firmware functions that can be used in your applications. You find these in the System folder in Project Explorer. For a complete list of the system functions, refer to Control Builder online help.

SupportLib

SupportLib is present in the Compact Control Builder to support the compatibility with the Control Builder, while transferring projects that have SupportLib as the connected library in the Control Builder.

Icon Library

The Icon Library (IconLib) contains icons that are used in interaction windows and CMD graphics in most other libraries.

The Icon library is automatically added to all control projects, via the control project template.

BasicGraphicLib

BasicGraphicLib contains Control Builder sub graphics that are mainly used in ControlObjectLib.

Basic Library

The Basic library (BasicLib) contains basic objects such as converters, counters, timers, pulse generators and edge detectors. This library is described in more detail in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*).*

The Basic library is automatically added to all control projects, via the control project template.

Alarm and Event Library

The Alarm and Event library (AlarmEventLib) contains function block and control module types for setting up alarm and event handling for objects that do not have built-in handling of alarms and events. This library is described in detail in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*.

Signal Libraries

SignalLib

The Signal library (SignalLib) contains types for adding supervision, alarm handling and error handling to I/O signals. SignalLib also contains types to define different rules that make it possible to control the process to predetermined states (vote objects). Types from this library can be used together with both binary and analog control applications.

SignalBasicLib

The SignalBasicLib library contains function block types suitable for safety applications. All objects in this library are without alarm and event handling. These simple function block types are used for overview and forcing of boolean and real signals. The easy design makes these function block types perform fast with low memory consumption.

SignalSupportLib

SignalSupportLib contains sub control builder objects, for example SignalBasicLib and SupervisionBasicLib. The function blocks are protected. They are used by, for example, SupervisionBasicLib objects to simplify code in these (parent) objects.

Graphics for objects in Signal Libraries

The indication of abnormal situations in an object is displayed in the interaction window indicator row on the object mode position. In case of abnormal situations, the ordinary icon appears with the warning color (yellow) as its background color.

An abnormal situation for an object occurs if:

- Any value in the object is simulated from the external environment.
- Any value from an input I/O or any value to an output I/O uses the predetermined value (ISP/OSP).
- The IO-channel is of redundant type and the inactive channel fails.
- The object is not a specialized I/O object and the signal is forced from the external environment.
  If the object is specialized, the force indication uses the force icon with a transparent background to indicate this situation, which is normal indication.

The information specific to the signal concerning the above abnormal situations is also displayed in the interaction window together with the signal value. The information appears as an yellow text string if the used value is of good quality. If the used value is of not of good quality, the information about the abnormal situations appears as a red text string.

Process Object Libraries

The Process Object libraries contain function block types and control module types for controlling motors, valves, ABB Drives and Insum Devices. Some types in these libraries are templates, that is, you can copy them to your own libraries and applications and modify the code to fit your particular process (see Templates on page 58). Only Core objects are protected.

There are a number of Process Object libraries:

Process Object Basic Library (ProcessObjBasicLib)

The Process Object Basic library contains the basic Core types that form a basis for valve and motor control objects in other Process Object libraries. It also contains two simple types with reduced functionality and lower memory consumption.

Process Object Extended Library (ProcessObjExtLib)

The Process Object Extended library contains a number of function block and control module types for general-purpose uni- and bi-directional control, and a number of types for valve and motor control. Most types in this library can be copied to your own libraries and be used as templates.

Process Object Drive Library (ProcessObjDriveLib)

The Process Object Drive library contains types for building ABB Drives control and supervision.

Process Object INSUM Library (ProcessObjInsumLib)

The Process Object INSUM library contains types for building INSUM control and supervision.

Synchronized Control Library

Group Start Library

The Group Start Library (GroupStartLib) contains control module types used to control and supervise the sequential startup of process objects.

The contains Function blocks and Control modules to control a configured and loaded SFC function inside the controller AC 800M.

Control Libraries

The Control libraries contain types and ready-made solutions for analog control. See Section 4, Analog Process Control. There are a number of Control libraries:

Control Simple Library (ControlSimpleLib)

The Control Simple library contains a number of types that are intended to be used for building simple control loops.

Control Basic Library (ControlBasicLib)

The Control Basic library contains function block types that are customized PID loops. These function blocks shall be connected to the I/O variables.

Control Standard Library (ControlStandardLib)

The Control Standard library contains control module types for building control loops, both stand-alone loops and cascade loops using master and slave configurations. They can be used together with types from other Control libraries, as well as together with objects from other libraries.

Control Extended Library (ControlExtendedLib)

The Control Extended library contains a number of control modules for arithmetics and signal handling. These types are intended to be used for building advanced control loops, together with objects from other Control libraries.

Control Advanced Library (ControlAdvancedLib)

The Control Advanced library contains control module types intended to be used to build control loops with advanced PID functions and decouple filter functions. The types from this library supports adaptive control and can be used to build dead-time control loops.

Control Object Library (ControlObjectLib)

The ControlObjectLib provides function blocks and control modules to define templates for using the control connection data type.

This library also contains template objects with control connection where user defined transfer function can be developed. It also contains voting input to alter the user defined functionality.

Control Solution Library (ControlSolutionLib)

The Control Solution library contains a number of ready-to-use control templates (for example handling cascade, feed-forward, mid-range, etc.). These templates are intended to be used directly in an application, as they are, but may also be copied to a self-defined library and modified, to comply an intended usage.

Control Fuzzy Library (ControlFuzzyLib)

The Control Fuzzy library contains types intended to be used for building fuzzy control solutions. It also contains a number of fuzzy control templates that you can copy to your own libraries, modify and use.

Control Support Library (ControlSupportLib)

The Control Support library is an internal library that stores the types used by other Control libraries. When the Control Support library is needed, it is automatically connected.

For a description of how to build analog control solutions from the types in these libraries (and from other types), see Section 4, Analog Process Control.

Supervision Library

Supervision Basic Library

Supervision Basic Library contains the function blocks intended for safety (shutdown) logic, which have one normal condition and one safe condition. The boolean activation signal is set, when an input object detects an abnormal condition. This signal is connected, through the shutdown logic, to the activation order input on an output object. When this is set, the output object is set to the defined safe condition. The central functionality is placed in the core function blocks in SignalSupportLib and AlarmEventLib.

The following blocks are not protected: SDBool, SDInBool, SDInReal, SDOutBool, SDReal and SDValve. This means that it is possible to make project specific copies.

Communication Libraries

The communication libraries contain function block types and control module types for reading and writing variables from one system to another. Typical communication function block types are named using the protocol name and function, for example, COMLIRead or INSUMConnect.

Communication is described in more detail in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*..

All supported protocols are described in the manual *AC 800M Communication Protocols (3BSE035982*)*.

There are a number of Communication libraries:

- COMLI Communication Library (COMLICommLib).
- Foundation FIELDBUS H1 Communication Library (FFH1CommLib).
- INSUM Communication Library (INSUMCommLib).
- MB300 Communication Library (MB300CommLib).
- MMS Communication Library (MMSCommLib).
- ModBus Communication Library (ModBusCommLib).
- Modbus TCP Library (ModBusTCPCommLib).
- MOD5-to-MOD5 Communication Library (MTMCommLib).
- Modem Communication Library (ModemCommLib).
- Siemens S3964 Communication Library (S3964CommLib).
- SattBus Communication Library (SattBusCommLib).
- Serial Communication Library (SerialCommLib).
- Self-defined UDP Communication Library (UDPCommLib).

- Self-defined TCP Communication Library (TCPCommLib).

Batch Library

The Batch library (BatchLib) contains control module types for batch control and for control of other discontinuous processes. It can be used together with any batch system which communicates via OPC Data Access and which supports the S88 state model for procedural elements.

The control module types in the Batch library are used for the interaction between the control application for an Equipment Procedure Element (for example, a phase or an operation) and the Batch Manager.

This library is described in more detail in Control Builder online help (select the library in Project Explorer and press F1).

The BatchLib also contains functionality for Batch Handling using batch advanced control modules. Templates for these control module types are provided in this library.

Hardware Overview

This part describes each standard hardware library briefly. For a list of all hardware types in a library and a description of each type, see Control Builder online help.

Basic Hardware

The Basic Hardware Library (BasicHWLib) contains basic hardware types such as controller hardware (for example, AC 800M), CPU units, Ethernet communication links, Com ports, ModuleBus, and so on.

The BasicHwLib is automatically inserted to all control projects and automatically connected to the controller, if the control project template AC 800M or SoftController is used.

PROFIBUS

The PROFIBUS hardware libraries contain PROFIBUS DP communication interfaces for the AC 800M.

- The CI854 PROFIBUS hardware library (CI854PROFIBUSHwLib) contains the communication interface for PROFIBUS DP, with redundant PROFIBUS lines and DP-V1 communication.

PROFIBUS Devices

The PROFIBUS device libraries contain hardware types that can be used to configure ABB Drive hardware and ABB Process Panels.

- The ABB Drive FPBA CI854 hardware library (ABBDrvFpbaCI854HwLib) contains hardware types to be used when configuring ABB Drive FPBA-01, using PROFIBUS DP (CI854).
- The ABB Drive NPBA CI854 hardware library (ABBDrvNpbaCI854HwLib) contains hardware types to be used when configuring ABB Drive NPBA-12, using PROFIBUS DP (CI854).

- The ABB Drive RPBA CI854 hardware library (ABBDrvRpbaCI854HwLib) contains hardware types to be used when configuring ABB Drive RPBA-01, using PROFIBUS DP (CI854).
- The ABB Process Panel CI854 hardware library (ABBProcPnlCI854HwLib) contains hardware types to be used when configuring ABB Process Panel, using PROFIBUS DP (CI854).
- The ABB Panel 800 CI854 hardware library (ABBPnl800CI854HwLib) contains hardware types to be used when configuring ABB Panel 800, using PROFIBUS DP (CI854).

PROFINET IO

The CI871 PROFINET IO hardware library, CI871PROFINETHwLib contains the communication interface for PROFINET IO. It also contains other hardwares that are used for configuring PROFINET.

PROFINET IO Devices

The PROFINET IO device libraries contain hardware types that can be used to configure ABB Drive hardware and ABB PROFINET IO device.

- The ABB Drive RETA-02 CI871 hardware library (ABBDrvRetaCI871HwLib) contains hardware types to be used when configuring ABB Drive RETA-02, using PROFINET IO (CI871).
- The ABB MNS *i*S CI871 hardware library (ABBMNSiSCI871HwLib) contains hardware types to be used when configuring ABB MNS *i*S, using PROFINET IO (CI871).
- The ABB Drive FENA-11 CI871 hardware library (ABBDrvFenaCI871HwLib) contains hardware types to be used when configuring ABB Drive FENA-11, using PROFINET IO (CI871).

Master Bus 300

The CI855 Master Bus 300 hardware library (CI855MB300HwLib) contains the communication interface (CI855) for and other hardware types to be used when configuring Master Bus 300.

INSUM

The CI857 INSUM hardware library (CI857InsumHwLib) contains the communication interface (CI857) and other hardware types to be used when configuring INSUM.

DriveBus

The CI858 DriveBus hardware library (CI858DriveBusHwLib) contains the communication interface (CI858) and other hardware types to be used when configuring DriveBus.

MODBUS TCP

The CI867 MODBUS TCP hardware library (CI867ModbusTcpHwLib) contains the communication interface (CI867 with two Ethernet ports) and other hardware types to be used when configuring MODBUS TCP.

IEC 61850

The CI868 IEC 61850 hardware library (CI868IEC61850HwLib) contains the communication interface (CI868 with two Ethernet ports) and other hardware types to be used when configuring IEC 61850.

CI868 interface is used for the horizontal communication between the AC 800M controller and different substation IEDs.

AF 100

The CI869 AF 100 hardware library (CI869AF100HwLib) contains the communication interface (CI869) and other hardware types to be used when configuring the Advant Fieldbus 100 bus.

MOD5

The CI872 hardware library (CI872MTMHwLib) contains the communication interface (CI872 with three optical ports) and the Remote MOD5 controller under each port.

EtherNet/IP and DeviceNet

The CI873 EtherNet/IP-DeviceNet hardware Library, CI873EthernetIPHWLib, integrated with AC 800M consists of the communication interface (CI873 with two Ethernet ports) and other hardware types to be used when configuring EtherNet/IP and DeviceNet.

LD 800DN is the linking device between EtherNet/IP and DeviceNet.

S100 I/O System

The CI856 S100 hardware library (CI856S100HwLib) contains the S100 communication interface (CI856), S100 Rack and S100 I/O units.

S200 I/O System

The S200 I/O libraries contain S200 adapter and S200 I/O units.

- S200 CI851 hardware library (S200CI851HwLib) contains S200 slave and I/O units for PROFIBUS DP-V0 (CI851).
- S200 CI854 hardware library (S200CI854HwLib) contains S200 slave and I/O units for PROFIBUS DP (CI854).

Satt Rack I/O System

- CI865 Satt ControlNet hardware library (CI865SattIOHwLib) contain the communication interface (CI865), S200 adapters, S200 units for Satt ControlNet, Satt Rack IO and 200RACN.

S800 I/O System

The S800 I/O libraries contain S800 adapters and S800 I/O units.

- The S800 I/O Modulebus hardware library (S800ModulebusHwLib) contains the S800 I/O units for ModuleBus.
- The S800 CI830 CI854 hardware library (S800CI830CI854HwLib) contains the adapter (CI830) and S800 I/O units for PROFIBUS DP (CI854).

- The S800 CI840 CI854 hardware library (S800CI840CI854HwLib) contains the adapter (CI840) and S800 I/O units for PROFIBUS DP (CI854).
- The S800 CI801 CI854 hardware library (S800CI801CI854HwLib) contains the adapter (CI801) and S800 I/O units for PROFIBUS DP (CI854).

S900 I/O System

The S900 I/O libraries contain field communication interfaces, adapters and S900 I/O units.

- The S900 CI854 hardware library (S900CI854HwLib) contains PROFIBUS DP fieldbus communication interface, adapter and S900 I/O units for PROFIBUS DP (CI854).

Serial Communication

The Serial Communication libraries contain hardware types for serial communication.

- The CI853 Serial Communication hardware library (CI853SerialComHWLib) contains the communication interface for RS-232C serial.
- The Serial hardware library (SerialHWLib) contain the serial communication protocol for SerialLib.
- The COMLI hardware library (COMLIHWLib) contain the serial communication protocol for COMLI.
- The ModBus hardware library (ModBusHWLib) contain the serial communication protocol for ModBus.
- The S3964 hardware library (S3964HWLib) contain the serial communication protocol for Siemens 3964R.

Self-defined UDP Communication

The UDP hardware library (UDPHwLib) contains the *UDPProtocol* hardware type that is used for self-defined UDP communication.

Self-defined TCP Communication

The TCP hardware library (TCPHwLib) contains the *TCPProtocol* hardware type that is used for self-defined TCP communication.

Printer and Modem

The Printer hardware library (PrinterHwLib) and Modem hardware library (ModemHwLib) contain the printer and modem protocol respectively.

FOUNDATION Fieldbus H1

The CI852 FOUNDATION Fieldbus H1 hardware library (CI852FFh1HwLib) contains the communication interface for the FOUNDATION Fieldbus H1 bus (CI852) and FF Devices.

The firmware available in Control Builder 5.1 does not support CI851. To run this module, the corresponding firmware available in Control Builder 5.0.2 must be downloaded to the connected PM8xx units.

Templates

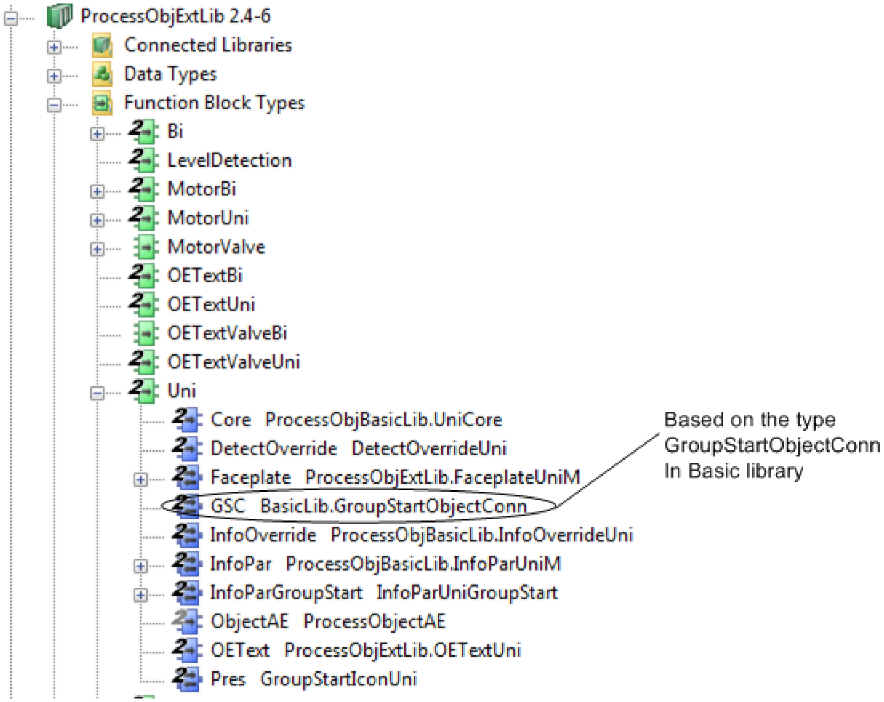
A template is characterized by the fact that it is not protected. It is intended to be copied to one of your own libraries, and modified inside that library. For an example of how to copy a template object to one of your own libraries, see Create a Library and Insert a Copy of a Type on page 304.

The moment you copy a type to your own library, the connection to the original template type is lost. This means that your copy does not reflect updates to the template.

However, a template type often consists of a number of objects from the standard libraries. Some of those might be protected (or even hidden), while some of them can be modified to suit the requirements of a particular organization, plant, or process. This also means that sometimes standard libraries still have to be connected to your library, due to the fact that they contain sub-types used inside the template type you copied. See Figure 13 on page 63

For information on the execution of objects based on template types and copies of template types, see Execution of Copied Complex Types on page 64.

To help you understand how this works, we will study a typical template type, the Uni function block from the Process Object Extended library.


ProcessObjExtLib 2.4-6
Connected Libraries
Data Types
Function Block Types
Bi
LevelDetection
MotorBi
MotorUni
MotorValve
OETextBi
OETextUni
OETextValveBi
OETextValveUni
Uni
Core ProcessObjBasicLib.UniCore
DetectOverride DetectOverrideUni
Faceplate ProcessObjExtLib.FaceplateUniM
GSC BasicLib.GroupStartObjectConn
InfoOverride ProcessObjBasicLib.InfoOverrideUni
InfoPar ProcessObjBasicLib.InfoParUniM
InfoParGroupStart InfoParUniGroupStart
ObjectAE ProcessObjectAE
OEText ProcessObjExtLib.OETextUni
Pres GroupStartIconUni

Based on the type
GroupStartObjectConn
In Basic library

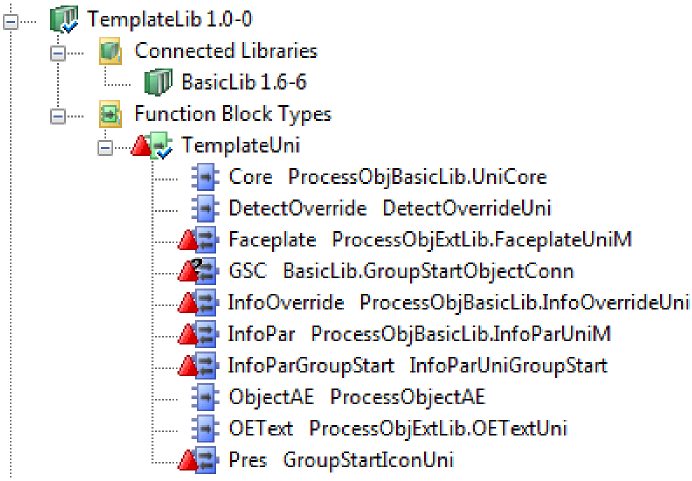
Figure 9. Uni function block type, with sub types and formal instances

The Uni function block type contains the following objects (formal instances):

- GSC (based on the type GroupStartObjectConn from the Basic library),
- Faceplate (based on the type FacePlateUni from the Process Object Extended library),
- InfoPar (based on the type InfoParUni, from the Process Object Basic library),
- InfoParGroupStart (based on the type InfoParUniGroupStart, from the Process Object Basic library),
- Pres (based on the type GroupStartIconUni),
- OEText (based on the type OETextUni from the Process Object Extended library)
- Core (based on the UniCore type from the Process Object Basic library),
- ObjectAE (based on the type ProcessObjectAE).

This means that the Uni type depends on the Basic library, the Process Object Basic library, and Process Object Extended library.

If we create our own library, TemplateLib, and copy the Uni function block type to this library, with the intention of modifying the Uni template into a uni-directional type that fits our process, it will look like Figure 10. The new function block type has been named TemplateUni.



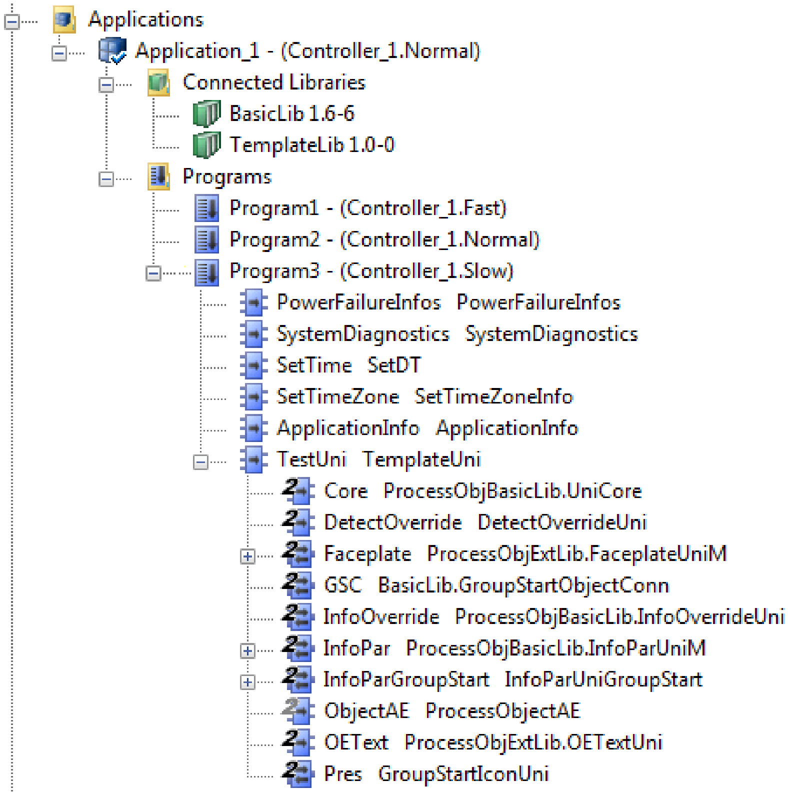
```
TemplateLib 1.0-0
    Connected Libraries
        BasicLib 1.6-6
    Function Block Types
        TemplateUni
            Core  ProcessObjBasicLib.UniCore
            DetectOverride  DetectOverrideUni
            Faceplate  ProcessObjExtLib.FaceplateUniM
            GSC  BasicLib.GroupStartObjectConn
            InfoOverride  ProcessObjBasicLib.InfoOverrideUni
            InfoPar  ProcessObjBasicLib.InfoParUniM
            InfoParGroupStart  InfoParUniGroupStart
            ObjectAE  ProcessObjectAE
            OEText  ProcessObjExtLib.OETextUni
            Pres  GroupStartIconUni
```

Figure 10. Uni, copied into a self-defined library TemplateLib. No connections to other libraries (red triangles on a number of types)

The red error triangles on the type and sub types appear because the new library, TemplateLib, is not connected to the libraries that contain some of the sub types.

If those libraries are connected to the new library, the red triangles disappear. After creating a copy of Uni, it can be modified to fit the specific requirements. For an example of how to add functions to a type, see Add Functions to Self-defined Types on page 309.

Once we are done adding to and modifying our type, we can use it in an application, see Figure 11. All we need to do to be able to use our new type in the application is to connect TemplateLib to the application and create an instance (TestUni) from the TemplateUni type.

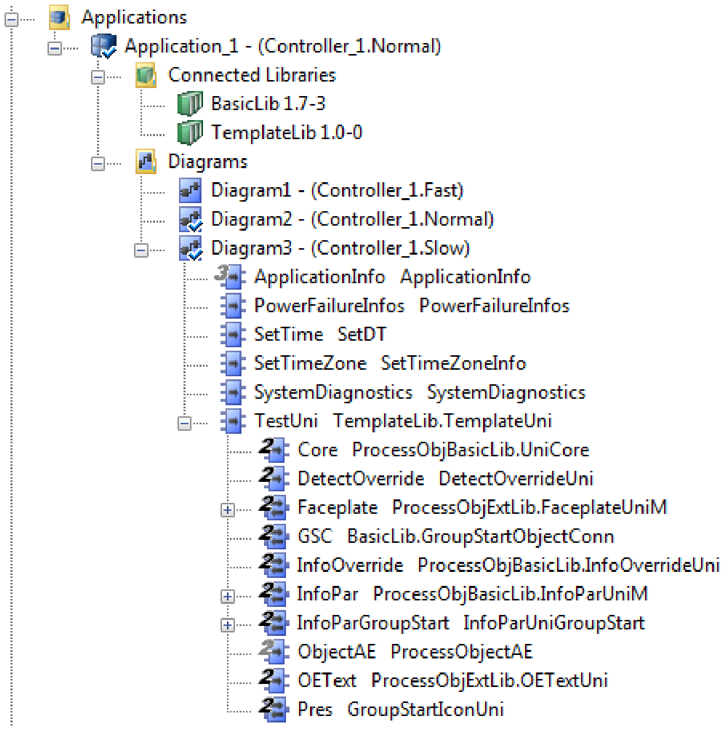


- Applications
  - Application_1 - (Controller_1.Normal)
    - Connected Libraries
      - BasicLib 1.6-6
      - TemplateLib 1.0-0
    - Programs
      - Program1 - (Controller_1.Fast)
      - Program2 - (Controller_1.Normal)
      - Program3 - (Controller_1.Slow)
        - PowerFailureInfos   PowerFailureInfos
        - SystemDiagnostics   SystemDiagnostics
        - SetTime   SetDT
        - SetTimeZone   SetTimeZoneInfo
        - ApplicationInfo   ApplicationInfo
        - TestUni   TemplateUni
          - Core   ProcessObjBasicLib.UniCore
          - DetectOverride   DetectOverrideUni
          - Faceplate   ProcessObjExtLib.FaceplateUniM
          - GSC   BasicLib.GroupStartObjectConn
          - InfoOverride   ProcessObjBasicLib.InfoOverrideUni
          - InfoPar   ProcessObjBasicLib.InfoParUniM
          - InfoParGroupStart   InfoParUniGroupStart
          - ObjectAE   ProcessObjectAE
          - OEText   ProcessObjExtLib.OETextUni
          - Pres   GroupStartIconUni

Figure 11. TemplateUni used in an application

Once we are done adding to and modifying our type, we can use it in a diagram under the application, All we need to do to be able to use our new type in the diagram is to connect TemplateLib to the application and create an instance

(TestUni) from the TemplateUni type in the diagram editor. Figure 12 shows the tree strucure after the instance is inserted in the diagram editor.

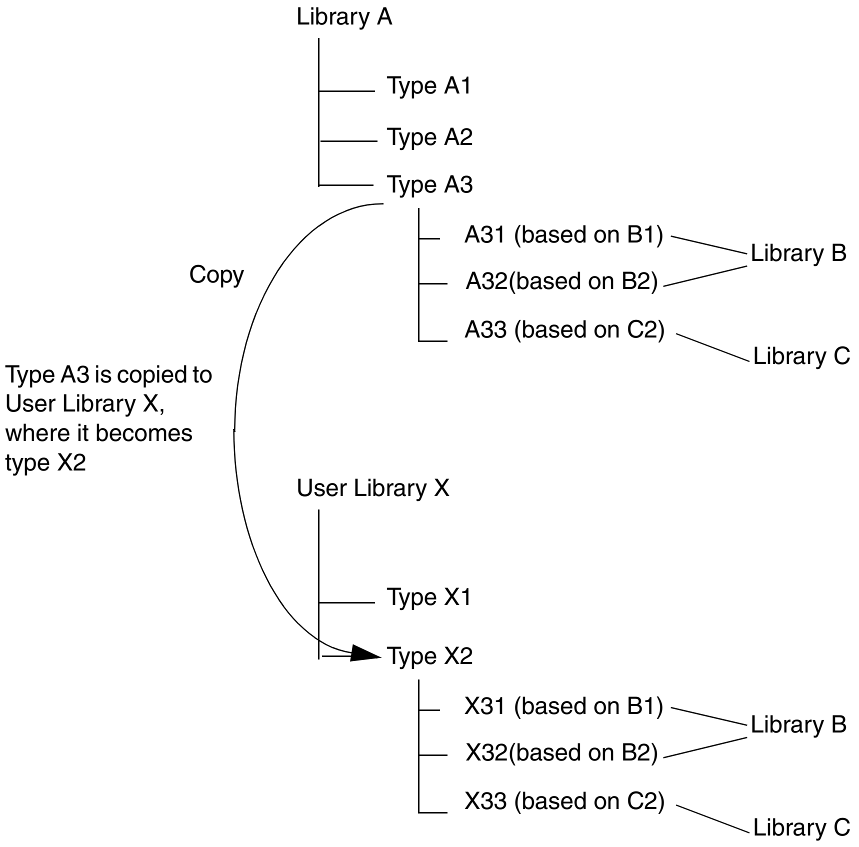


```
Applications
    Application_1 - (Controller_1.Normal)
        Connected Libraries
            BasicLib 1.7-3
            TemplateLib 1.0-0
        Diagrams
            Diagram1 - (Controller_1.Fast)
            Diagram2 - (Controller_1.Normal)
            Diagram3 - (Controller_1.Slow)
                ApplicationInfo  ApplicationInfo
                PowerFailureInfos  PowerFailureInfos
                SetTime  SetDT
                SetTimeZone  SetTimeZoneInfo
                SystemDiagnostics  SystemDiagnostics
                TestUni  TemplateLib.TemplateUni
                    Core  ProcessObjBasicLib.UniCore
                    DetectOverride  DetectOverrideUni
                    Faceplate  ProcessObjExtLib.FaceplateUniM
                    GSC  BasicLib.GroupStartObjectConn
                    InfoOverride  ProcessObjBasicLib.InfoOverrideUni
                    InfoPar  ProcessObjBasicLib.InfoParUniM
                    InfoParGroupStart  InfoParUniGroupStart
                    ObjectAE  ProcessObjectAE
                    OEText  ProcessObjExtLib.OETextUni
                    Pres  GroupStartIconUni
```

Figure 12. TemplateUni used in a diagram under the application

Note that there is no need to connect the libraries that are connected to TemplateLib (the reference from instances to types is there anyway). The only time this would be necessary is when a library contains a type that is used for a parameter connection to the surrounding code or to another object outside our type

Note that all sub types (the formal instances) retain a relation to their corresponding types. For example, a change to the OETextUni type in the Process Object Basic library will also affect the TemplateUni type, since this type contains an instance of OETextUni.



1.User Library X will still depend on Library B and Library C, since X31 and X32 are instances of B1and B2, and X33 is an instance of C2.

2.User Library X will **not** depend upon Library A. Changes to A3 will not affect X2.

Figure 13. Overview of the template concept

Execution of Copied Complex Types

It is important to understand what happens when you copy and modify a type that contains instances of other types (formal instances). We start with a template type from one of the standard libraries, as shown in Figure 14.

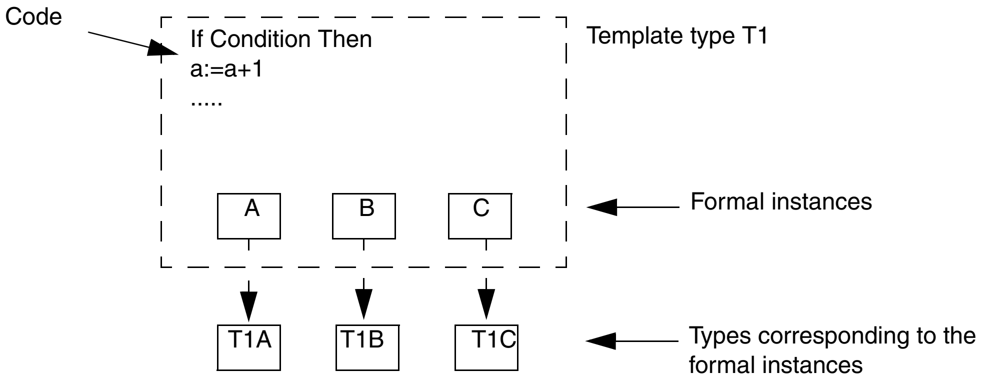


Figure 14. Template type with formal instances

Each formal instance has a corresponding type. These types are normally stored in the same library as the template object, or in a connected library (of the Basic or Support type).

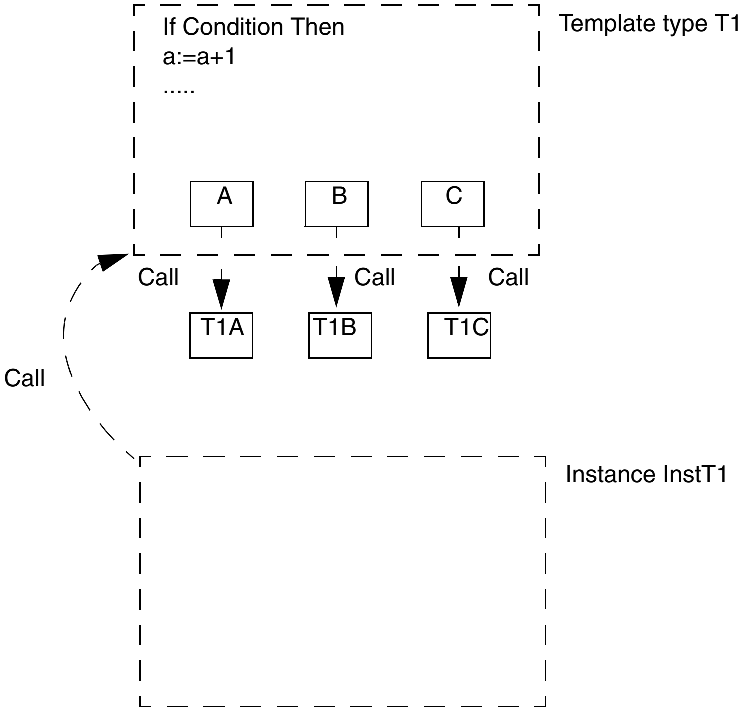


Figure 15. Execution of an instance of a template type with formal instances

When an object (a formal instance) is created from this type and the formal instance is executed, what happens is the following, see Figure 15:

1. The object (InstT1) calls the type (T1).
2. When the type T1 is called, the code executes and calls are made to all types (T1A, T1B and T1C) corresponding to the formal instances (A, B and C).
3. Each type that is called (T1, T1A, T1B, T1C) executes, operating on data from the corresponding object (T1) and formal instances (T1A, T1B, T1C).

This relation is affected if a copy of a Complex template type is created and the copied type is modified by adding code or by replacing one of the formal instances.

First, we create a new type (MyT1) by copying the template type (T1), see Figure 16.

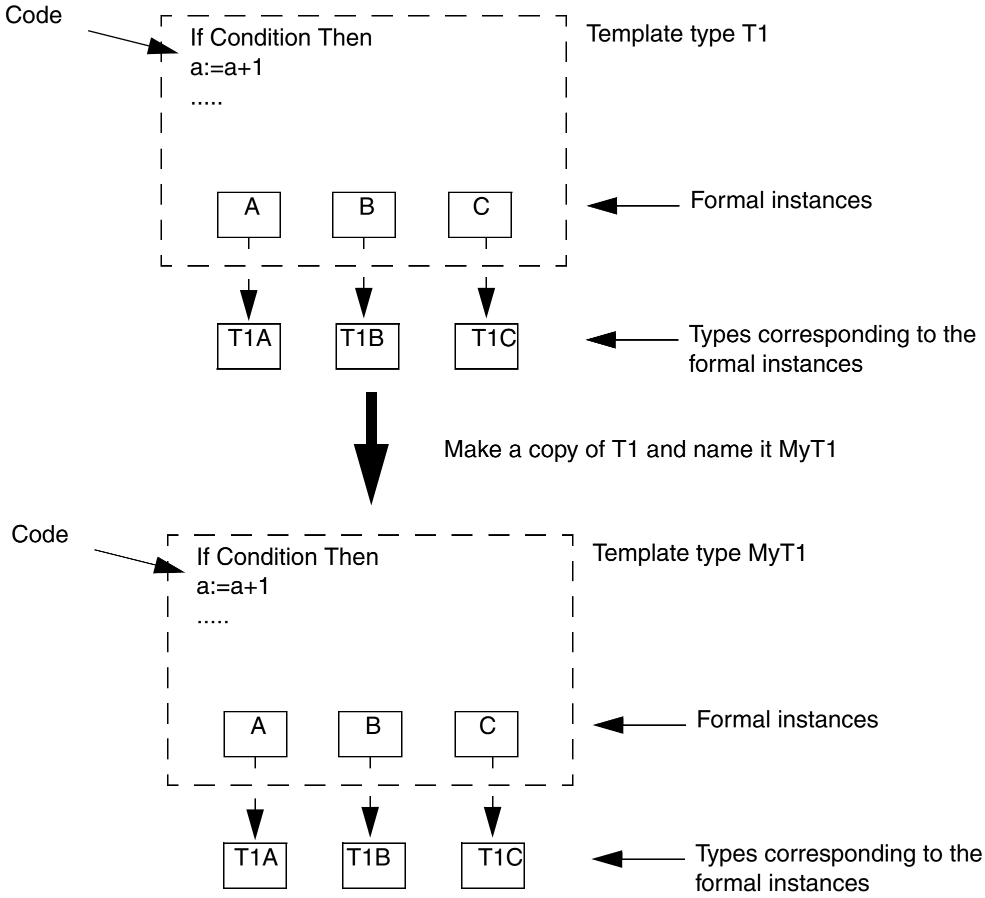


Figure 16. Copying a template type with formal instances

When an object based on MyT1 is executed, the call is to MyT1, and not to the type T1. However, each formal instance retains its connection to their corresponding type. The call to MyT1 will also generate calls to T1A, T1B and T1C, see Figure 17.

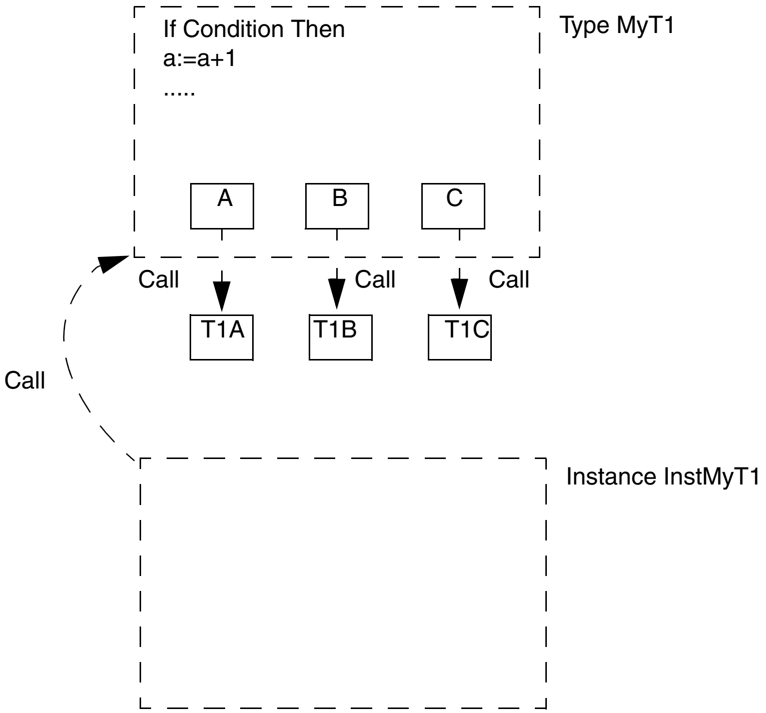


Figure 17. Execution of an object based on a template type copy

The purpose of copying a template type is to modify this type to fit your specific requirements. Say, for example, that we need an object that works differently from one of the formal instances. We might, for example, want to replace a valve with a valve of a different type than the original one.

If T1B is the original valve type, replace it with the new valve type V2B, and connect the new type to the MyT1 type, see Figure 18.

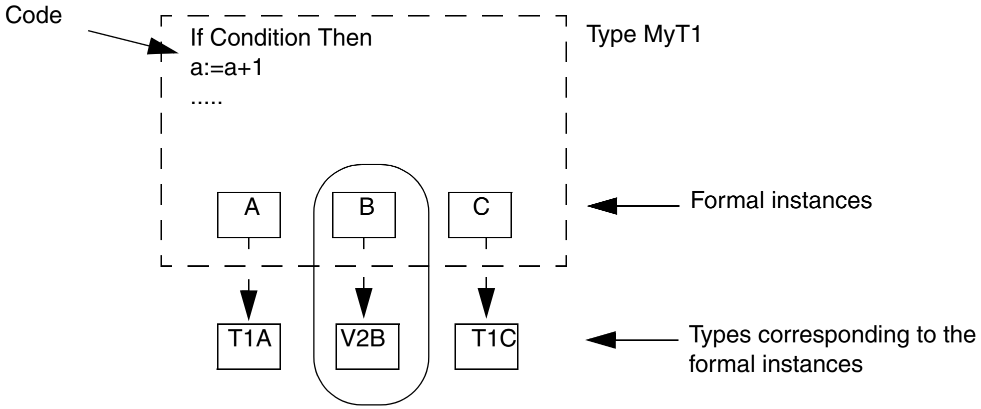


Figure 18. Copy of template type with formal instances, modified by replacing one of the formal instances (circled in the figure)

It is of course also possible not only to replace formal instances, but also to modify your copied type by adding or removing formal instances, and by adding to the code or changing it.

When an object based on this type is executed, what happens is the following, see Figure 19:

1. The object (InstMyT1) calls the type (MyT1).
2. When the type MyT1 is called, the code executes and calls are made to all types (T1A, V2B and T1C) corresponding to the formal instances (A, B and C).
3. Each type that is called (T1, T1A, V2B, T1C) executes, operating on data from the corresponding instance.

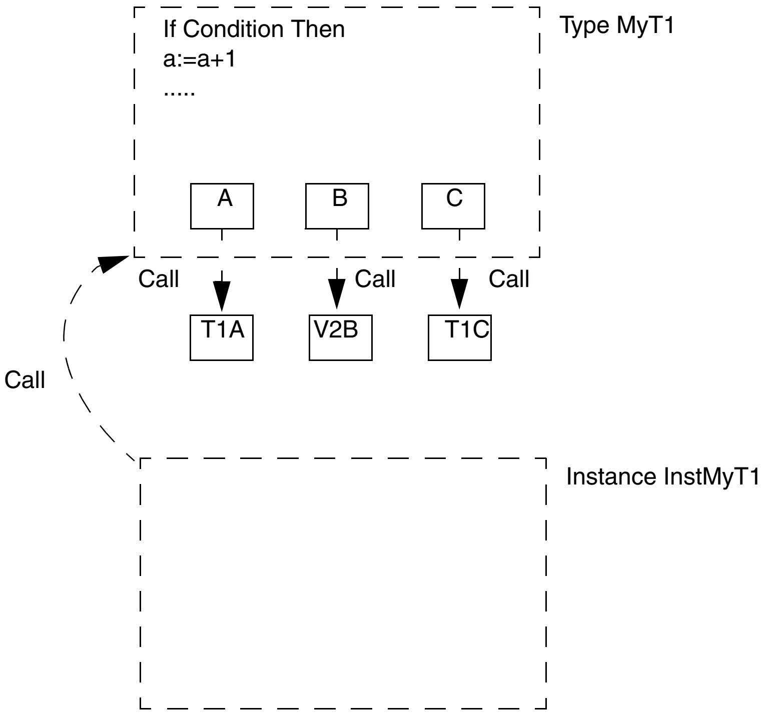


Figure 19. Execution of an instance of a modified template type copy

The following are the features of a copied template type:

- When a copy is made of a template type, the connection to the original template type is lost.
- However, all formal instances keep their connection to their corresponding types.
- Modifications to a copy do not affect the original type.
- Modifications to the template type do not affect the copy.
- Modifications to formal instances (that is, to their corresponding types), always affect both the original template types and its copies (as long as that particular formal instance is still used in the copy).

Section 4 Analog Process Control

Introduction

This section describes how to use types from the Control libraries to create analog control solutions for your automation system. The section contains:

- A description of the concept behind the Control libraries, see Concept on page 72.
- Advice and instructions on how to implement analog control solutions using the types in the Control libraries, see Design on page 107.
- Examples on how to implement analog control solutions using the types in the Control libraries, see Getting Started with ControlConnection on page 123.
- Detailed information on individual library types for signal handling, see Advanced Functions on page 152.A description and an example on how to use control module type templates from the Control Solution library (ControlSolutionLib), see Control Loop Solutions on page 226

For a discussion on the difference between function blocks and control modules, and how to choose between the two, see the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* .

Throughout this section, the word “controller” refers to a type used in control loops, for example, a PID controller.

Concept

The Control libraries contain a number of function blocks and control modules that are designed to help you construct complex signal systems and control loops with high functionality and flexibility. Some of them can be used as is, while some of them have to be combined to suit a specific application.

The Control libraries contain the PID controllers and analog signal handling functions. These functions are needed to handle analog signals and to construct control loops, both simple control loops (including cascade control loops) and very advanced ones.

This section describes the concept behind the Control libraries, split on the following sub-sections.

- Control Libraries Overview on page 72 gives an overview of all Control libraries.
- Functions and Other Libraries Used for Analog Control on page 75 is a summary of AC 800M firmware functions and functions from other libraries that can be used in connection with analog control. Here, you will find references to other parts of the manual that describe individual function and objects.
- ControlConnection on page 75 describes the ControlConnection structured data type, which is used to simplify communication between different control objects and their environment.

Control Libraries Overview

The Control libraries for AC 800M are standard libraries that are installed with the Compact Control Builder. There are a number of Control libraries:

- Control Basic library,
- Control Simple library,
- Control Standard library,
- Control Extended library,
- Control Advanced library,
- Control Object library,
- Control Solution library,
- Control Fuzzy library.

For a short description of each of these libraries, see the following sub-sections.

The Control libraries are also supported by a number of firmware functions that are included in AC 800M firmware and in other libraries. For information on those, see Functions and Other Libraries Used for Analog Control on page 75.

Control Basic Library

The Control Basic library contains function blocks for a number of ready-made complete control loops, simple as well as cascade, to be connected directly to I/O.

Control Simple Library

The Control Simple library contains function block types that can be used to build control loops using function blocks only. These have to be connected by the user, for forward and backward signal directions.

Control Standard Library

The Control Standard library has control module types for:

- a standard PID controller
- I/O signals
- signal conversion
- manual control
- branches, as well as supervision of levels, selections, and limitations.

Control Extended Library

The Control Extended library has control modules types for arithmetics and signal handling. Together with the Control Standard library control modules, it is possible to construct control loops with more control functions, for example, PID loops.

Control Advanced Library

The Control Advanced library has a control module type for an advanced PID controller, containing all the functionality of the PID controllers in the other control libraries and decouple filter functions. In addition, the controller may be configured for continuous adaptation of controller parameters. It may also be configured as a predictive PI, that is a PPI controller, and it has a gain scheduler. There is also a control module for adding a stiction compensator function and decouple filter to an output signal.

Control Object Library

The ControlObjectLib provides control modules to define templates for using the control connection data type. The control modules will have function blocks as sub objects. The library contains advanced multiple inputs/outputs with up to 4 inputs and outputs created as control module templates. These templates also provide manual override and bumpless transfer.

Control Solution Library

The Control Solution library (ControlSolutionLib) contains control module types for a number of ready-made complete control loop solutions. A control module solution provides a complete loop control solution with control, signal monitoring, alarm handling, cascade, feed-forward, mid-range, trending, operator graphics and also a possibility to add asset optimization functionality.

Control Fuzzy Library

The Control Fuzzy library contains control module types for definitions of fuzzy logic rules for process control, and for constructing multi-variable fuzzy controllers which are able to handle many inputs and outputs. The fuzzy controller also has the additional functions of a PID controller.

Functions and Other Libraries Used for Analog Control

Functions that can be used in connection with types from the Control libraries can be found in the Compact Control Builder (AC 800M firmware functions), as well as in the Basic library (counters, timers, latches) and in the Signal library.

For more information on system functions and basic functions in other libraries, see the *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* and online help. This manual also discusses how to use types from the Alarm and Event library to set up additional alarm and event handling, and how to set up control network communication using types from the Communication libraries.

Functions and library types that are not included in the Control libraries, but can be used for signal handling in connection with control loops, are described under Advanced Functions on page 152.

ControlConnection

ControlConnection is a data structure which contains all signals that are sent between the objects of a signal system or control loop. Some of the signals are sent in the forward direction of the loop and some are sent backwards, such as value, status and range. The complexity of the signal systems and control loops can then be reduced considerably for the signals between the objects.

Introduction

Data is generated by a source, computed in one or several objects, and finally forwarded to a consumer of data. This is the most common kind of data flow. If each object is to operate independently, and be able to connect with the other objects, a number of conditions must be fulfilled.

Learn how to build your own control module types with ControlConnection in Getting Started with ControlConnection on page 123 and Creating a Control Module with ControlConnection (CC template) on page 130.

Each object has to ask its succeeding object if it is ready to receive data and do whatever it is that the object does. The succeeding object must issue an acceptance when ready to receive data.

This means that the question must be computed, before it is possible to give the answer. This is achieved by using code sorting. To interconnect objects of this type, you have to use the type of parameters that simply carry data, irrespective of their direction. These type of parameters cannot have values of their own.

This is the basis for connections between the control modules in the control libraries, and the interconnecting data type called *ControlConnection*.

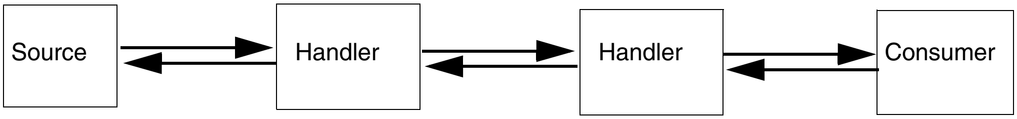


Figure 20. The principle of the data flow between control modules

Signal Flow between Control Modules

Simple and advanced control loops with various functions can be built from the control modules in the control libraries. A typical constellation is described in Figure 21. It consists of a number of input signal control modules, connected to calculation and controller modules, which are connected to an output signal control module.

There are two ways of sending information between control modules; via a graphical connection and via a parameter connection. Graphical connection is described in Graphical Connection of Control Loops on page 79.

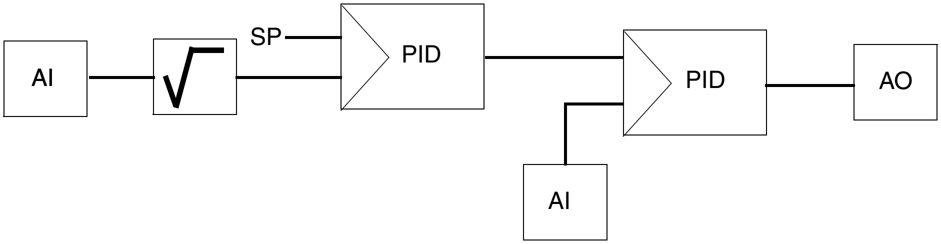


Figure 21. Example of graphical connections between control modules making a control loop

When connected to each other, the control modules have the following characteristics.

- Information is transferred between the modules, forward as well as backward in the control loop, during the same cycle of execution. This is used, for example, to achieve bumpless transfer upon a change from Manual to Auto mode, and to prevent integrator wind-up in the entire control loop.
- Signal flow without delay, in both directions, is obtained through automatic sorting of code blocks in the control modules.
- If a signal is not valid, for example < 4 mA, it is possible to consider this in succeeding control modules. Examples of this are transfer to Manual mode or setting a predetermined value on the output.

The chain of control modules in a control loop and/or during analog handling must start with a control module handling input signals, and the chain going to the right must end with a control module handling output signals.

A *ControlConnection* output from a control module must normally only be connected to one input in another succeeding control module.
A *ControlConnection* output from a control module must not be connected as a feedback to a previous control module in the chain, unless, in exceptional cases, a State control module is connected in between. See Miscellaneous Objects on page 225.

When using non-graphical connections of *ControlConnection* parameters you must be careful not to connect one output to several inputs, except, for example, for a presentation signal of Level6Connection type, which must be parameter-connected from the Level6CC control module to several control modules. See section, Miscellaneous Objects on page 225.

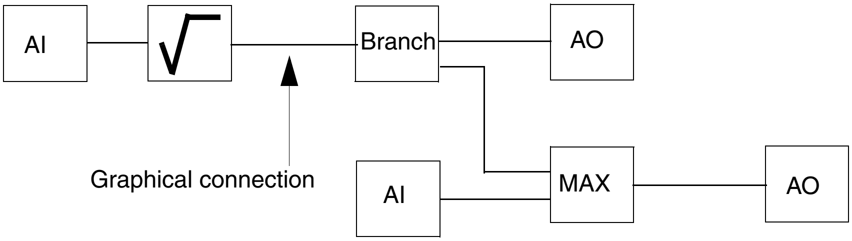


Figure 22. A chain of connected control modules for analog signal handling

In the chain of connected control modules, as seen in Figure 22, the main signal flow is from left to right, as illustrated by the bold arrows in Figure 23. The main signal flow may have divergent and convergent branches. Thin vertical arrows represent operator interactions.

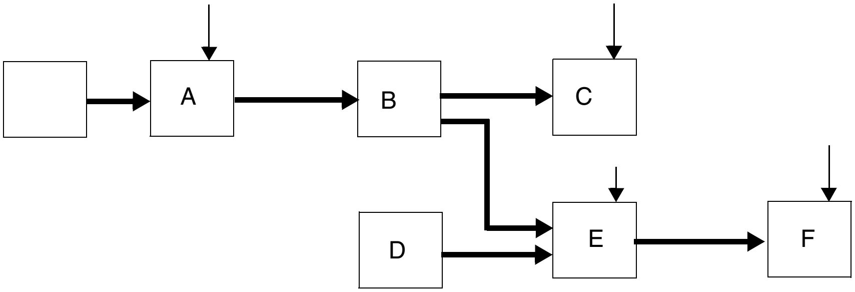


Figure 23. The main signal flow between the control modules

Information propagates during one scan, without any delay in the main signal flow. For example, the effect of an event in control module A is perceived in control module F in the same scan.

The connected control modules are influenced by their surroundings, for example, the operator interface or the surrounding application program. The effect of such an influence propagates ,without delay in the main signal flow, to outside of the control module in which the influence occurs.

The effect of such an influence also propagates, without delay, in the opposite direction to the main signal flow, to outside of the control module in which the influence occurs. This is called backtracking (see Backtracking on page 82). This effect does not, however, influence the main signal flow until one scan later. For example, the effect of an influence, shown by the thin arrow on block E, propagates to F in the same scan. It also propagates to D, B and A in the same scan, but not to C. In B, calculations are carried out on the effect before it is forwarded to A. In the next scan, the effect is used in the calculation of the main flow.

Graphical Connection of Control Loops

To create a control loop the control modules are connected to form chains by means of graphical connections from left to right, which is also the direction of the main signal flow.

Backtracking calculations are performed in all control modules in the control loop, when in Backtracking mode. The value may be transferred backward in the chain, if the chain before the backtracking-initiating control module has a member with an internal state able to collect the backtracked value. Information about the presence of such an internal state is given in the ControlConnection.

A control module has an internal state when its output is determined not only by the input signal, but also by its history.

In control modules with an internal state, the output signal might be limited, for example, when a succeeding control module is in Backtracking mode or has reached a maximum or minimum value. Information on this situation is passed backwards in the control loop chain in the ControlConnection data structure. The preceding control module with an internal state then stops further increase or decrease of the signal value.

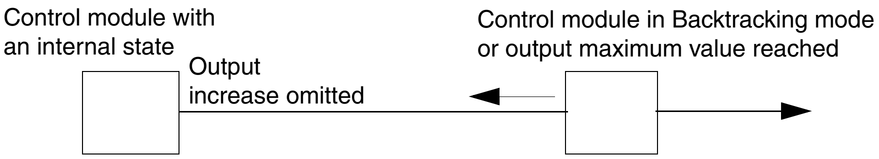

Control module with
an internal state
Output
increase omitted
Control module in Backtracking mode
or output maximum value reached

Figure 24. Example of when a succeeding control module has reached a limit value; the preceding control module stops further integration (anti-windup)

ControlConnection Data Type

The control-loop-specific ControlConnection data type handles both forward and backward signal flow and contains information about, for example, the signal value, status, and unit, as well as the measuring range of the signal to prevent the signal from exceeding its limits (in any situation), see Figure 25.

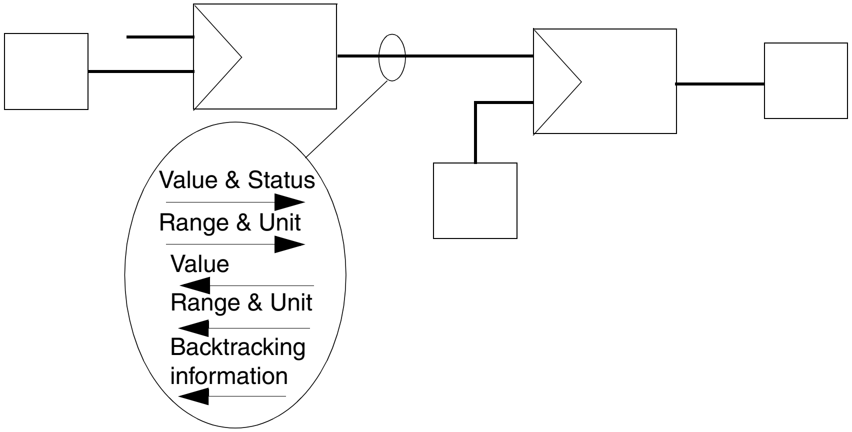


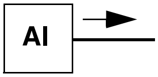
Figure 25. A graphical connection of ControlConnection type with its main signal flow forward and a flow backward

The *Value* component in the forward structure represents the main signal flow of the loop. The *Status* component contains information about the quality of the loop.

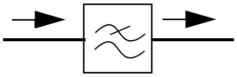
Ranges and Units

A control module limits its output signal to within the output range. When using graphical connection between control modules, signal ranges and units are generally calculated automatically, and sent forward, as well as backward, through the *ControlConnection* structure data type. according to the following rules.

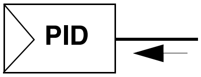
1. Ranges and units from inputs and outputs propagate forward and backward in the control module chain until they reach a control module whose output range and unit can not be automatically defined using the input range. Such control modules may be a controller, an integrator, or a filter. More information may then be supplied by manual setting of range and unit for this control module output. If range and unit values are not entered, the default values 0–100% are used.
2. If a control module receives overlapping ranges and units from the input and output directions, then the range and unit from the preceding control module are used.



The AI and AO control modules start to transmit their ranges in the forward and the backward directions, respectively.



Some control modules simply allow the ranges to pass through. Other control modules calculate and suggest a range. The range can also be set by the operator.



Control modules with an internal state receive the same output as the signal sent backward from the succeeding control module, unless a range and unit are entered manually.

The output range is visible in some control modules, such as PID controllers, and arithmetic modules etc., and may be changed. The range can be changed in the interaction window in Online or Test mode.

Backtracking

The behavior of a control module that has *ControlConnection* connections depends on which mode the succeeding control module in the chain is in. A control module in Backtracking mode when succeeding control modules indicate that they, for example, are in Manual mode. This means that integrator wind-up of a controller is prevented (see section, Anti-Integrator Wind-Up Function on page 153) and that bumpless transfer, for example, between Manual and Auto, is achieved (see Bumpless Transfer on page 154).

In Control Builder, control modules are dimmed if backtracking is active.

Enter Range Value and its Unit of Measure

In control modules where it is possible to enter an output range value and its unit of measure in the interaction window, you can override the propagated or calculated default values of the range value and its unit of measure. Select and enter maximum and minimum values and their unit of measure of the output signal.

Fraction

In all interaction windows of control modules, which have a ControlConnection, you can set the fraction, which is a local variable in each control module for setting of the decimals shown in the interaction window(s).

When an I/O signal from, for example a PT100 transmitter, has a wider physical range than desired you can enter a narrower range in the AnalogInCC control module.

When signals go through a number of arithmetic calculation control modules the ranges can easily take large positive or negative values. Also, units of measure may become long compound words that are unabridged, for example in multiplications. Therefore, it is important to set the range to acceptable values. Also, check that the unit of measure is correct and simplify it by abridgement. This must be checked in the last calculation control module before the signal goes to a control module that does not send range value and its unit of measure backward, such as the controller, derivative, integral or piecewise linear control module types.

If a constant value, unchangeable in Online mode, is required in the calculations, use a RealToCC control module in which you set the maximum and minimum values equal to the in value given for the constant, in Offline mode.

Limitation of Controller Output

It may be necessary to limit the controller output to a narrower interval in Auto mode, for example, when you test new controller parameter settings. Limitation of the controller output then ensures that the process is not upset if the controller is poorly tuned.

Do this limitation from the parameter interaction window and the limits are only active in Auto mode. When you deselect the limitation, the limits are returned to the normal endpoints of the range. Bumpless transfer is ensured whenever the limits are changed.

ControlConnection between Applications Using Control Modules

The MMSToCC control module along with the CCToMMS control module can be used when transferring signals of ControlConnection between applications.

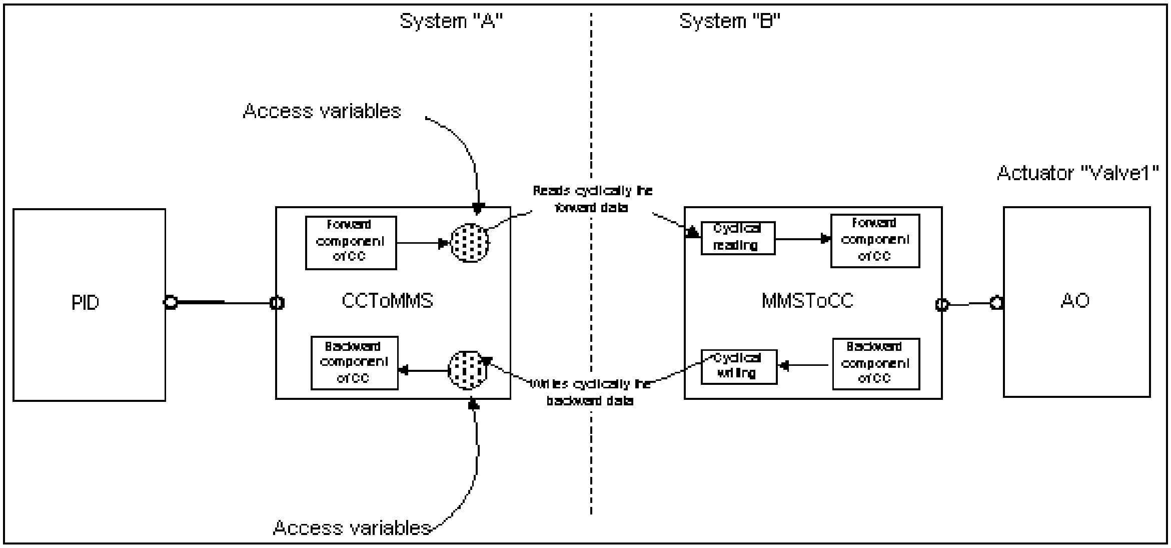


Figure 26. ControlConnection using control modules

It is recommended to parameterize the control connection to fulfil the IAC needs and to use IAC in prior of the MMS mechanism.

ControlConnection Using Communication Variables

The ControlConnection data type can be used for cyclic communication between top level diagrams, programs, and top level single control modules that use communication variables. The communication variables are declared in the top level diagram editor, program editor or top level single control module editor.

If the data type of the communication variable is ControlConnection, the backward component is marked with a *reverse* attribute, and a bidirectional communication is acheived.

Controller Types

This section describes controller principles and main controller functions. Advanced built-in functions and objects for signal handling are described under Advanced Functions on page 152.

A process may be of many types. The process may be rapid or slow, have dead time, involve non-linear process characteristics, have many different cases and/or conditions of operation, or depend on valve characteristics. The process may also involve viscous media, the process may be exothermic, or dependent on various calculations, etc.

The process requirements may also be to achieve a certain production quality. Manual intervention must be carried out in a smooth, so-called bumpless way. In addition, there are information and communication demands on the operators and/or other systems regarding momentary values, alarms, data history in short and long perspectives, and the presentation of these in a clear way.

To fulfill all the process demands, many functions must be carried out by the controller. The solution may involve anything from a single controller to several controllers with internal relationships, for which the system has complete control modules and function blocks.

The core of the PID controllers in the control libraries is based on PID algorithms. The only exception is the fuzzy controller, which has a design of its own. Additional functions are added by setting parameters, or by combining a controller with other control modules.

PID, PI, P, and PD Controllers

The basic, classical PID, PI, P, and PD controllers of ideal type are based on the control algorithms described in the section Controller Algorithms on page 89. When discussing these controllers as a group, the term PID controller is used.

Generally, with the aid of built-in functions, the system performs mode transfers and other changes in a bumpless way, see Bumpless Transfer on page 154. An anti-integrator (sometimes called reset) wind-up function is included, to prevent the output signal from drifting away, see Anti-Integrator Wind-Up Function on page 153.

In addition to controller algorithms and built-in functions, the standard libraries contain additional functions and types for creating almost any other controller type.

PPI Controller

If a process has long dead time in comparison with the process time constant, a predictive PI controller configuration, PPI (based on a simplified Otto Smith controller), can be used. The process dead-time delay is added, but the parameter values for P and I correspond to the same values as in a PI controller.

Fuzzy Controller

The fuzzy controller can handle one input and one output, as well as many inputs and many outputs. You may be able to use a fuzzy controller where PID control fails, or does not work well.

A fuzzy controller has most of the functions of a PID controller, together with the possibility of defining fuzzy logic rules for process control.

Functionality in PID Controllers

A survey of the following eight PID Controllers is presented in Table 4.

1.	PidLoop	(Function block)
2.	PidLoop3P	(Function block)

3.	PidCascadeLoop	(Function block)
4.	PidCascadeLoop3P	(Function block)
5.	PidCC	(Control module)
6.	PidAdvancedCC	(Control module)
7.	PidSimpleReal	(Function block)
8.	PidSimpleCC	(Control module)

Table 4. Functionalities in the PID Controllers

Included Function	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Belongs to ControlSimpleLib	No	No	No	No	No	No	Yes	Yes
Belongs to ControlBasicLib	Yes	Yes	Yes	Yes	No	No	No	No
Belongs to ControlStandardLib	No	No	No	No	Yes	No	No	No
Belongs to ControlAdvancedLib	No	No	No	No	No	Yes	No	No
PID algorithm	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Tunable Beta-factor	No	No	No	No	Yes	Yes	No	No
Backtracking	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Setpoint Backtracking	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Integrator wind-up prevention	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Bumpless transfer	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Tracking	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Internal setpoint ramping	No	No	No	No	Yes	Yes	No	No
Offset Adjustment	No	No	No	No	Yes	Yes	No	No
Deviation alarm limits	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Pv alarm limits	No	No	No	No	Yes	Yes	No	No
Limitation of output	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Feedforward	Yes	Yes	Yes	Yes	Yes	Yes	No	No
P-start	Yes	Yes	Yes	Yes	Yes	Yes	No	No

Table 4. Functionalities in the PID Controllers (Continued)

Included Function	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Predictive PI control	No	No	No	No	No	Yes	No	No
ERF	No	No	No	No	Yes	Yes	No	No
Disable PD at limited windup	No	No	No	No	Yes	Yes	No	No
Enable out ramp in manual	No	No	No	No	Yes	Yes	No	No
Autotuner relay	Yes	Yes	Yes	Yes	Yes	Yes	No	No
Autotuner extension (step)	No	No	No	No	No	Yes	No	No
Autotuner structure and design selection	No	No	No	No	Yes	Yes	No	No
Oscillation detection	No	No	No	No	No	Yes	No	No
Sluggish control detection	No	No	No	No	No	Yes	No	No
Gain scheduler	No	No	No	No	No	Yes	No	No
Adaptive control	No	No	No	No	No	Yes	No	No
Compensation for redundant I/O deviation on process value	Yes	Yes	Yes	Yes	No	No	No	No

Basic Controller Principles

Basically, a controller has three signals, the setpoint signal (Sp), the process value signal (Pv) and the output signal. The P controller, PI controller, and PID controller are different types of analog controllers. The basic controller configuration is the P controller, where P stands for proportional.

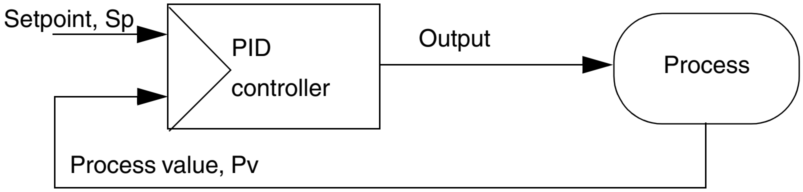
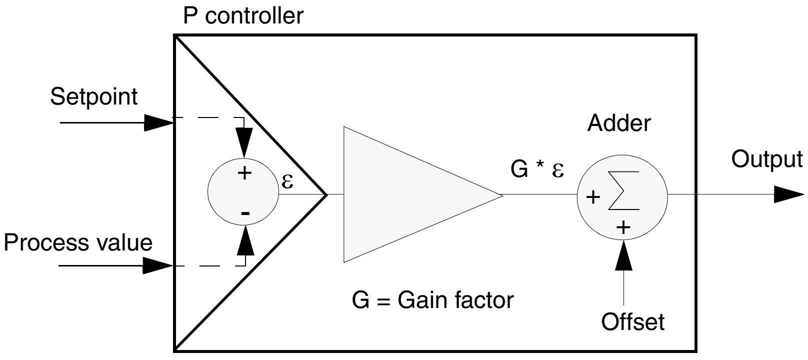


Figure 27. A control loop with the function block of a PID controller

The most simple P controller may be described as follows. The controller compares the setpoint value with the process value and the difference is called the control deviation, ε. This is amplified by G (the amplification or gain factor) and an offset signal is added to obtain a working point. See figure below. The result is the output signal.



ε= Setpoint – Process value
Output value = G * ε + Offset

Figure 28. The basic P controller

In a PI controller, the offset is replaced by the Integral part (I part).

In a PID controller the Derivative part (D part), acting on the filtered process value, is also added to the output.

Controller Algorithms

The PID controller algorithms used are of ideal type. The controller input from the process Pv and the setpoint Sp are regularly read by the controller. A read is also called a sample, and the time between two samples is called the sampling time. The required output signal value is calculated, for each sample, by comparing samples of the input and setpoint values. The sample time is equal to the task cycle time of the current task. The process value may be filtered before it enters the derivative part of a control algorithm, by a first-order, low-pass filter. See the algorithms below, and Figure 29 and Figure 30.

1. The P controller has the following algorithm:

$$Out = G \times (Sp - Pv) + Offset$$

2. The PD controller has the following algorithm:

$$Out\ =\ G \times ((Sp - Pv) + T_d \times \frac{d}{dt}(FilterOf(-Pv,\ T_{fil}))) + Offset$$

3. The PI controller has the following algorithm:

$$Out\ =\ G \times ((\beta \times Sp - Pv) + 1/T_i \times \int (Sp - Pv)dt)$$

4. The PID controller has the following algorithm:

$$Out\ =\ G \times ((\beta \times Sp - Pv) + 1/T_i \times \int (Sp - Pv)dt + T_d \times \frac{d}{dt}FilterOf(-Pv,\ T_{fil}))$$

5. The PPI controller has the following algorithm:

$$Out\ =\ G \times ((\beta \times Sp - Pv) + 1/T_i \times \int (Sp - Pv)dt) - 1/T_i \times \int (Out(t) - Out(t - L))dt$$

Abbreviations in the Algorithms	**Description**
Sp	Setpoint
Pv	Process value
G	G is defined as: *G = Gain x (OutRange) / (PvRange)*

Abbreviations in the Algorithms	**Description**
Gain	The gain you enter in the interaction window or by code via the *InteractionPar* parameter. Gain is normalized and dimensionless according to the above definition. Thus the gain can be influenced by the settings of the ranges for the process and the output values.
OutRange	The range (maximum – minimum) of the Out value
PvRange	The range (maximum – minimum) of the Pv process value
β	Setpoint weight
T_i	Integral time of the controller
T_d	Derivative time of the controller
T_{fil}	Filter time of the low-pass filter for the derivative part
FilterOf (x,y)	The expression x is sent through a low-pass filter. The filter time is equal to y.
Out	Output from the controller
Out(t-L)	Output value delayed by the dead time, L.

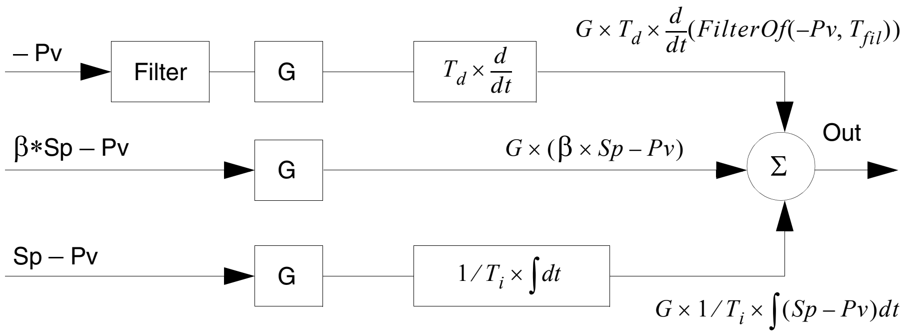

– Pv
Filter
G
$T_d \times \frac{d}{dt}$
$G \times T_d \times \frac{d}{dt}(FilterOf(-Pv, T_{fil}))$
β∗Sp – Pv
G
$G \times (\beta \times Sp - Pv)$
Σ
Out
Sp – Pv
G
$1/T_i \times \int dt$
$G \times 1/T_i \times \int (Sp - Pv)dt$

Figure 29. The principle of an ideal PID controller

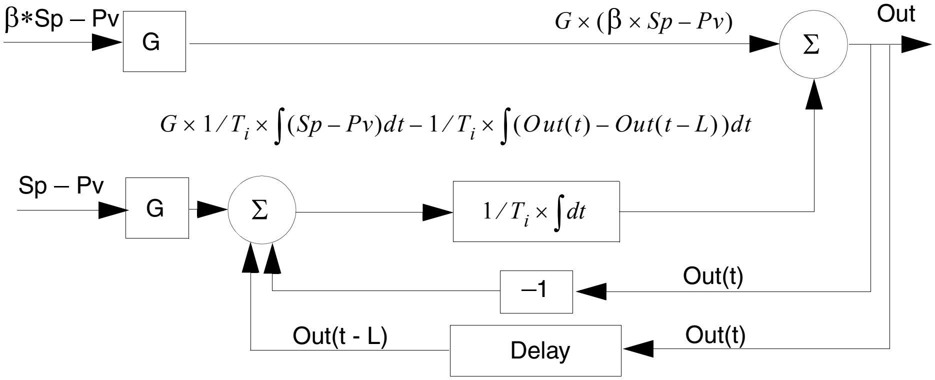

β∗Sp – Pv
G
$G \times (\beta \times Sp - Pv)$
Σ
Out
$G \times 1/T_i \times \int (Sp - Pv)dt - 1/T_i \times \int (Out(t) - Out(t-L))dt$
Sp – Pv
G
Σ
$1/T_i \times \int dt$
–1
Out(t)
Out(t - L)
Delay
Out(t)

Figure 30. The principle of a PPI controller

Parameter values can be set or changed via the interaction window, or in the application. When the control deviation is within a dead zone, specified in the parameter window, the output is constant.

Offset Adjustment

The P and PD controller types do not have an integrator, but they have an offset instead. The offset is a tuning parameter of the controller and used to determine its working point. It is normally constant, but may be automatically adjusted if the offset adjustment function is enabled.

If offset adjustment is enabled the offset is automatically adjusted in some modes and also when the controller parameters, for example the gain, is changed. The adjustment of the offset is always done in such a way that the output of the controller becomes continuous.

Details on how the offset is adjusted are discussed in the section Bumpless Transfer on page 154.

The offset adjustment function must be used with care. When offset is adjusted, the behavior of the controller changes. For example, the control deviation at a certain working point may change. This is not acceptable in many cases.

Internal Setpoint Backtracking

Internal setpoint backtracking adjusts the setpoint automatically, while in internal mode (provided that the function is enabled). The purpose is to make sure that the output of the controller is continuous at mode changes. In some cases, process value tracking is also achieved. See Bumpless Transfer when Enabling or Disabling the Limitation of the Output on page 158.

The adjustment is done the same way as in external mode, when a control module with backtracking capability is connected to the setpoint.

Reduced Effect of Setpoint Changes

For controllers with an integrator (PI, PID, PPI), the setpoint influence on the proportional term is governed by a setpoint weight, the beta (β) factor, to make the output as smooth as possible. These controllers have two setpoint weight factors which are used when the setpoint is considered continuous or discontinuous, respectively. An abrupt, discontinuous change in the setpoint should not be allowed to have full effect on the output. The setpoint is considered to be discontinuous, for example, when the internal setpoint is selected and the operator enters a setpoint value manually, or when a preceding control module is in Manual mode.

The switching between the two setpoint weight factors is automatic, depending on whether the setpoint signal is continuous or discontinuous. The setpoint weight is in the discontinuous case by default 0 (zero). Otherwise, at continuous setpoint signals, the setpoint weight factor is by default 1 (one).

At normal usage, the controller provides the appropriate default value for the setpoint weight factors. However, in PidCC and PidAdvancedCC*,* these two parameters are editable. In some special cases, the user may tune the setpoint weight factor by editing these parameters to real values between the 0 and 1 limits. An example is when an externally calculated tuning is to be applied to the controller.

Internal Setpoint Ramping

Internal setpoint ramping smoothens setpoint changes for PidCC and PidAdvancedCC. When activating this function, a target setpoint can be entered. The ramping can be started and stopped, only if internal setpoint is selected. Once the ramping is started the setpoint will change smoothly to the targeted setpoint. The ramp increase and decrease rates can be set separately. The time to reach the target setpoint is displayed. When the target is reached, the ramping is deactivated and stopped.

The ordinary internal setpoint field can be disabled to prevent discontinuous setpoint changes. The user can then only enter setpoints as target setpoints.

If, in the meantime, the setpoint is switched to external setpoint, the ramping is stopped. The transfer to the external setpoint will be bumpless if the external setpoint is able to receive a backtracking value. The target setpoint is left unchanged. If the controller enters Backtracking mode, ramping is aborted, and backtracking starts instead.

Limitation of Controller Output

Generally, maximum and minimum values for the controller output signal are specified using the I/O connection editor. These values are the endpoints of the vertical axis in bar graph and trend curve windows for the signal. In Auto mode, the user can limit controller output to a narrower interval from the parameter window, as long as the limiting functions are enabled. An example of when it may be necessary to limit the output is when testing new controller settings.

The controller output may then be limited, to ensure that the process is not badly disrupted if the controller is poorly tuned. The output range may also be changed in the controller parameter window. This is usually done when there is no I/O connection editor.

Hysteresis vs Dead Zone

The concepts hysteresis and dead zone are explained as follows.

Hysteresis

To avoid frequent activations at a level, a hysteresis value can be set in some control modules, for example, when a signal is close to an alarm limit.

In Figure 31, activation is desired when a signal exceeds a high alarm level. The activated and the deactivated conditions are separated by the hysteresis below the high alarm level. Depending on the direction of the signal, the hysteresis is added to either the activated or deactivated condition.

The alarm is first deactivated and the signal increases to the high alarm level. The alarm is then activated. It remains activated until the signal falls below the hysteresis and is then deactivated. The next time the signal exceeds the high alarm level, the alarm is activated. For a low alarm level, the situation is the reverse with the hysteresis above it.

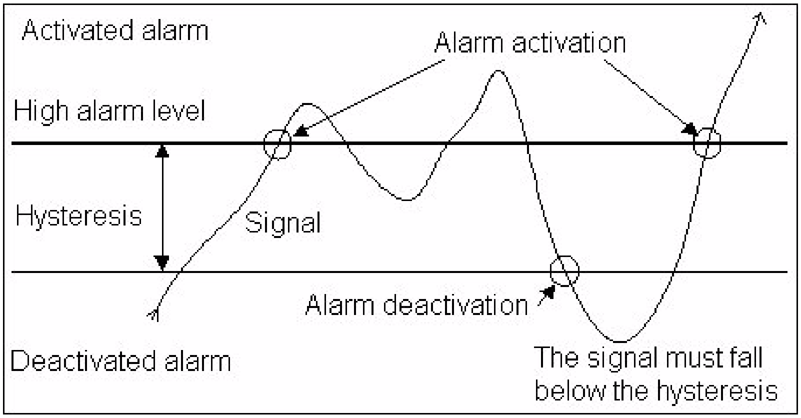


Figure 31. Activation is desired when a signal exceeds a high alarm level

Dead Zone

To allow a signal a certain noise level without causing activation, a dead zone can be set around it in some control modules, for example, for a control deviation. A small fluctuation in the signal is then allowed. The signal is not active when it is within the dead zone. When the signal exceeds or falls below the dead zone limits the signal is active. Figure 32 showing a dead zone on both sides of a desired signal value.

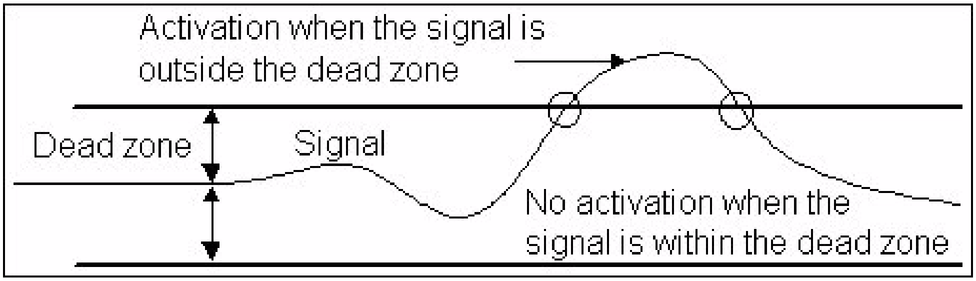

Activation when the signal is outside the dead zone
Dead zone
Signal
No activation when the signal is within the dead zone

Figure 32. A dead zone on both sides of a desired signal value

If PidCC or PidAdvancedCC is used, the calculation of the derivative part has no dependecy to the selected dead zone. This means that the output signal may change even if the difference between Sp and Pv is less than the dead zone value. Small changes in Pv is filtered by a special low pass filter assigned for the derivative part.

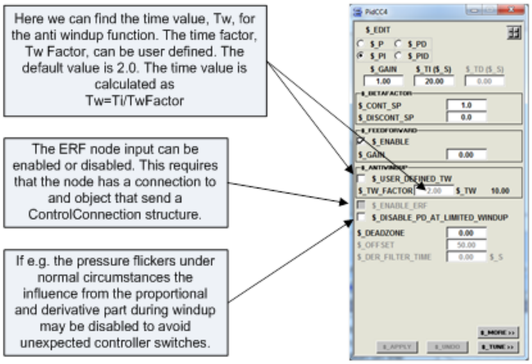
ERF

This input node of type ControlConnection can be found in the controllers PidCC and PidAdvancedCC. It is used in rare cases instead of the IRF Internal Reset feedback that is delivered in the backward direction of the Out node on the controllers. This signal is used as the limiting signal related to max and min reached situations and as a backtracking value when backtrackinf occurs.

Disable PD at limited windup

This is used in override control configurations, when known disturabances occurs on the process values on any of the constraint controllers. An example on such a situation is when the pressure suddenly changes, then the deviation of the constraint controller has moved this controller out of selection.

This setting is made either in the interaction windows or in the faceplate of the controller.



Here we can find the time value, Tw, for the anti windup function. The time factor, Tw Factor, can be user defined. The default value is 2.0. The time value is calculated as Tw=Ti/TwFactor

The ERF node input can be enabled or disabled. This requires that the node has a connection to and object that send a ControlConnection structure.

If e.g. the pressure flickers under normal circumstances the influence from the proportional and derivative part during windup may be disabled to avoid unexpected controller switches.

PidCC4

$_EDIT

○ $_P ○ $_PD
● $_PI ○ $_PID

$_GAIN 1.00 $_TI ($_S) 20.00 $_TD ($_S) 0.00

$_BETAFACTOR
$_CONT_SP 1.0
$_DISCONT_SP 0.0

$_FEEDFORWARD
☑ $_ENABLE
$_GAIN 0.00

$_ANTIWINDUP
□ $_USER_DEFINED_TW
$_TW_FACTOR 2.00 $_TW 10.00

□ $_ENABLE_ERF
□ $_DISABLE_PD_AT_LIMITED_WINDUP

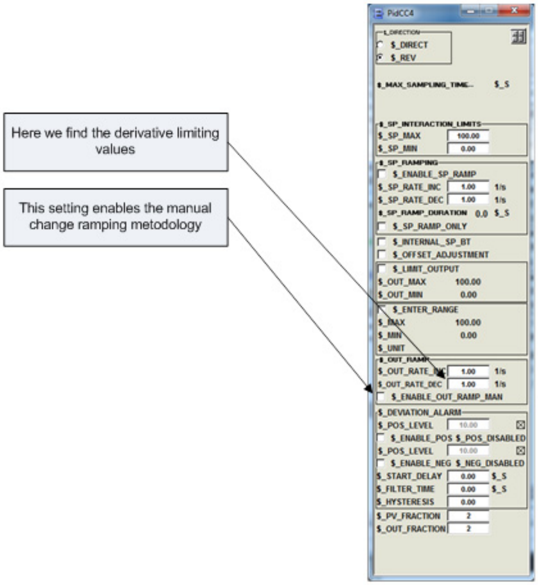
$_DEADZONE 0.00
$_OFFSET 50.00
$_DER_FILTER_TIME 0.00 $_S

Figure 33. The EDIT interaction window of a PidCC or PidAdvancedCC

Enable Out Ramp Manual

The setting for enabling ramping of the output signal of a controller is implemented in PidCC and PidAdvancedCC objects. The intension is to limit the derivative of the output in manual mode. The derivative is defined by the OutIncLim and the OutDecLim InteractionPar components.

This setting is to be made by the user either in the interaction windows or in the faceplate of the controller.



Here we find the derivative limiting values

This setting enables the manual change ramping metodology

PidCC4

$_DIRECTION
○ $_DIRECT
● $_REV

$_MAX_SAMPLING_TIME... $_S

$_SP_INTERACTION_LIMITS
$_SP_MAX 100.00
$_SP_MIN 0.00

$_SP_RAMPING
□ $_ENABLE_SP_RAMP
$_SP_RATE_INC 1.00 1/s
$_SP_RATE_DEC 1.00 1/s
$_SP_RAMP_DURATION 0.0 $_S
□ $_SP_RAMP_ONLY

□ $_INTERNAL_SP_BT
□ $_OFFSET_ADJUSTMENT

□ $_LIMIT_OUTPUT
$_OUT_MAX 100.00
$_OUT_MIN 0.00

□ $_ENTER_RANGE
$_MAX 100.00
$_MIN 0.00
$_UNIT

$_OUT_RAMP
$_OUT_RATE_INC 1.00 1/s
$_OUT_RATE_DEC 1.00 1/s
□ $_ENABLE_OUT_RAMP_MAN

$_DEVIATION_ALARM
$_POS_LEVEL 10.00 ☒
□ $_ENABLE_POS $_POS_DISABLED
$_POS_LEVEL 10.00 ☒
□ $_ENABLE_NEG $_NEG_DISABLED
$_START_DELAY 0.00 $_S
$_FILTER_TIME 0.00 $_S
$_HYSTERESIS 0.00

$_PV_FRACTION 2
$_OUT_FRACTION 2

Figure 34. The interaction window of a PidCC or PidAdvancedCC

Fuzzy Control

A fuzzy controller is based on fuzzy logic which is a generalization of the common Boolean logic.

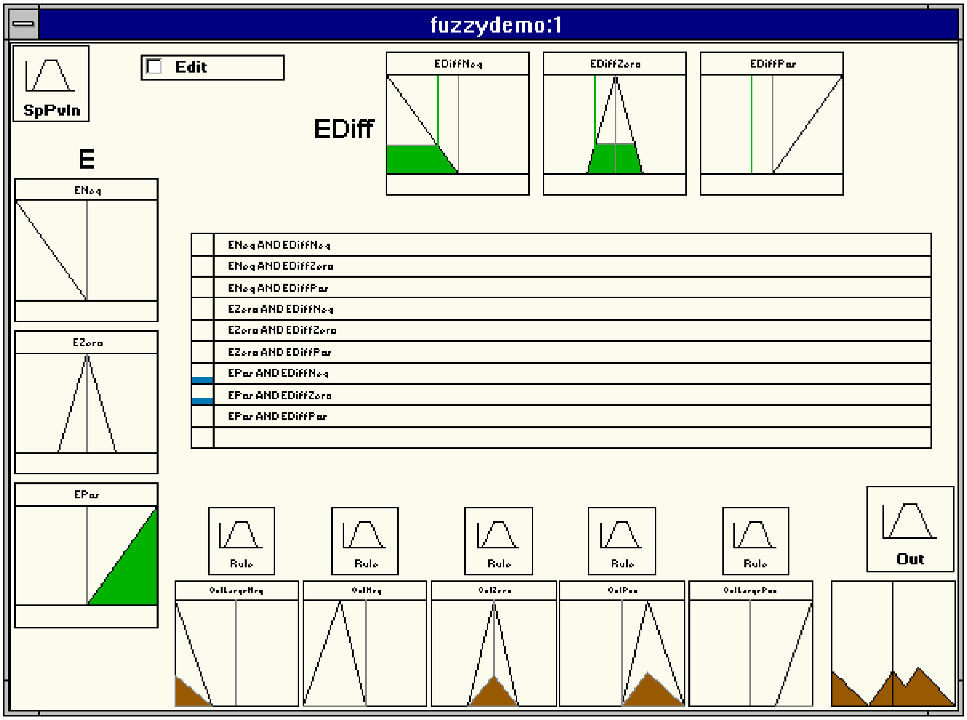


Figure 35. The fuzzy controller window

A fuzzy controller consists of a linear part and a fuzzy logic part. The linear part has many of the functions of the PID controller, for example:

- Computation of the control deviation ε = Setpoint – Process value and its derivative (even the second derivative).
- Computation of the derivative of the process value.
- A low-pass filter for derivative of the process value and the control deviation.
- Internal and external setpoint.

- Handling of absolute and relative alarms.
- An integrator with anti-integrator wind-up function.
- Manual and automatic output.
- Tracking function for the output.
- A feedforward function.

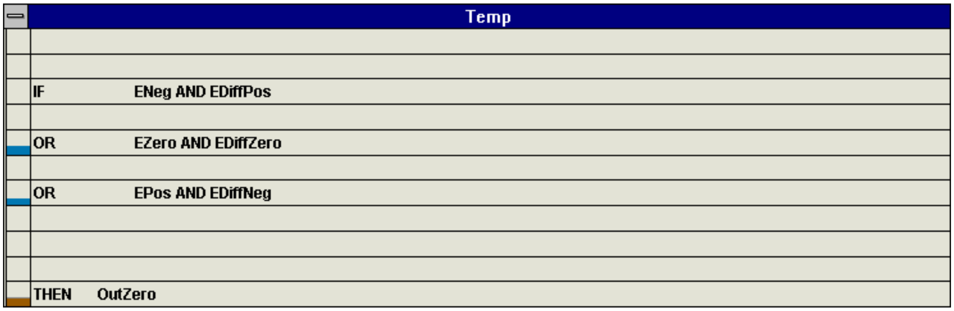
The fuzzy logic part of the controller contains the functions that define the rules for control of the process, for example:

- Computation of the degrees of membership of a number of signals to a number of fuzzy sets.
- Computation of fuzzy conditions.
- Computation of fuzzy rules.
- Computation of output membership functions for a number of controller outputs.
- Defuzzyfication of the output membership functions.

The fuzzy controller works as follows:

- One or more process values, and possibly also external setpoints, are entered into the linear part of the controller. The process values may be low-pass filtered. If no setpoint is used, the first derivative of each process value is computed. The result is made available to the fuzzy logic part of the controller.
- If setpoints are used, the control deviations, ε = Setpoint – Process Value, and their two first derivatives are computed for each process value. These two results are also made available to the fuzzy logic part of the controller.

- The fuzzy logic part of the controller receives one or more signals from the linear part. It may receive the control deviation, the process value or their derivatives from the linear part. Each of these signals is entered into a number of input membership functions. The output from an input membership function is a signal, which assumes values between 0 and 1. This value indicates the degree of membership of the signal from the linear part to this particular membership function.
- The outputs from the input membership functions are combined into fuzzy conditions using the fuzzy operators NOT and AND. The fuzzy operator NOT is defined as NOT X = 1 – X. The fuzzy operator AND is defined as X AND Y = Min(X,Y). The result has a value between 0 and 1.
- The fuzzy conditions may then be combined into fuzzy rules using the fuzzy operators NOT and OR. The fuzzy operator OR is defined as X OR Y = Max(X,Y). The output from a fuzzy rule also has a value between 0 and 1 and is called the degree of satisfaction of the rule.



Temp

IF ENeg AND EDiffPos

OR EZero AND EDiffZero

OR EPos AND EDiffNeg

THEN OutZero

Figure 36. A fuzzy rule

- The degree of satisfaction of each fuzzy rule is then used to compute the current output membership function for the rule.
- There may be a number of output membership functions associated with each output from the controller. All output membership functions associated with the same output from the controller are combined into one output membership function. This is done by computing the envelope (the maximum value of all the functions at every point) of all the membership functions.

- The resulting output membership function for a certain controller output is used to compute the value of the output. This is called defuzzyfication and is done by computing the center of gravity of the output membership function.
- The defuzzyfied outputs from the fuzzy logic part of the controller are then entered into the linear part of the controller.
- Each output may then be integrated and is limited by an anti-integrator wind-up function.

There are also functions for feedforward, output tracking and Manual mode.

The fuzzy controller also has a simulation facility. The values of the control deviation, the process value and their derivatives may be simulated. Simulation can be used to evaluate the behavior of the fuzzy logic part of the controller.

Relation to Other Libraries and Modules

The fuzzy controller is designed to operate together with the control library modules in the same way as the PID controller does. The fuzzy controller has, to a large extent, the same parameter interface as the PID controller. It should be connected to the other control modules in exactly the same way as the PID controller, i.e. using ControlConnection data type connections.

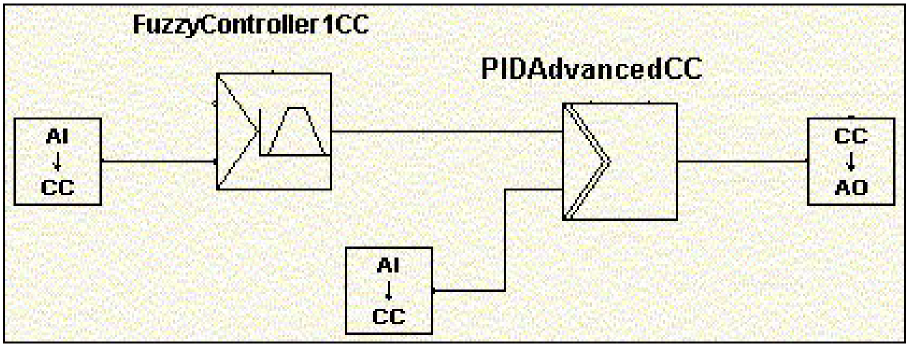


Figure 37. Fuzzy control relations

Typical configuration where a fuzzy controller is used as part of the control loop. A fuzzy controller operates as a master controller, the output of which is connected to the setpoint of a PidAdvancedCC.

How to Use Fuzzy Controller Templates

Introduction

Your copy of a template can be modified in the following ways: the number of inputs, outputs, membership functions, conditions and rules can be changed, and these items can be grouped in different ways.

The fuzzy control modules have one input and one output, but they can be configured for many inputs and many outputs.

Select the fuzzy control module which best suits your needs.

Step-by-Step Instructions for Using Templates

1. Copy a fuzzy controller template from the library. Select a template (for example FuzzyController1CC). For more information, refer to Templates on page 58.
2. Paste the copied fuzzy controller in the Control Module Types in the applications folder.
3. Rename it, for example, Own_FuzzyController1.
4. Create a new empty control module called, for example, SM1 and take Own_FuzzyController1 into use by the Create/Control Module command.

It is now possible to make changes to the fuzzy controller. You can change the number of inputs, outputs, membership functions, conditions or rules. These items can also be grouped in different ways.

The steps presented above describe how the user is able to make a new control module type in an application, but if the modified fuzzy controller is needed in many projects it is recommended that the user creates a module type in a new library which can then be included in many projects.

Internal Data Flow of Fuzzy Controllers

The sub-modules in the fuzzy controller templates are connected as shown in the illustration below.

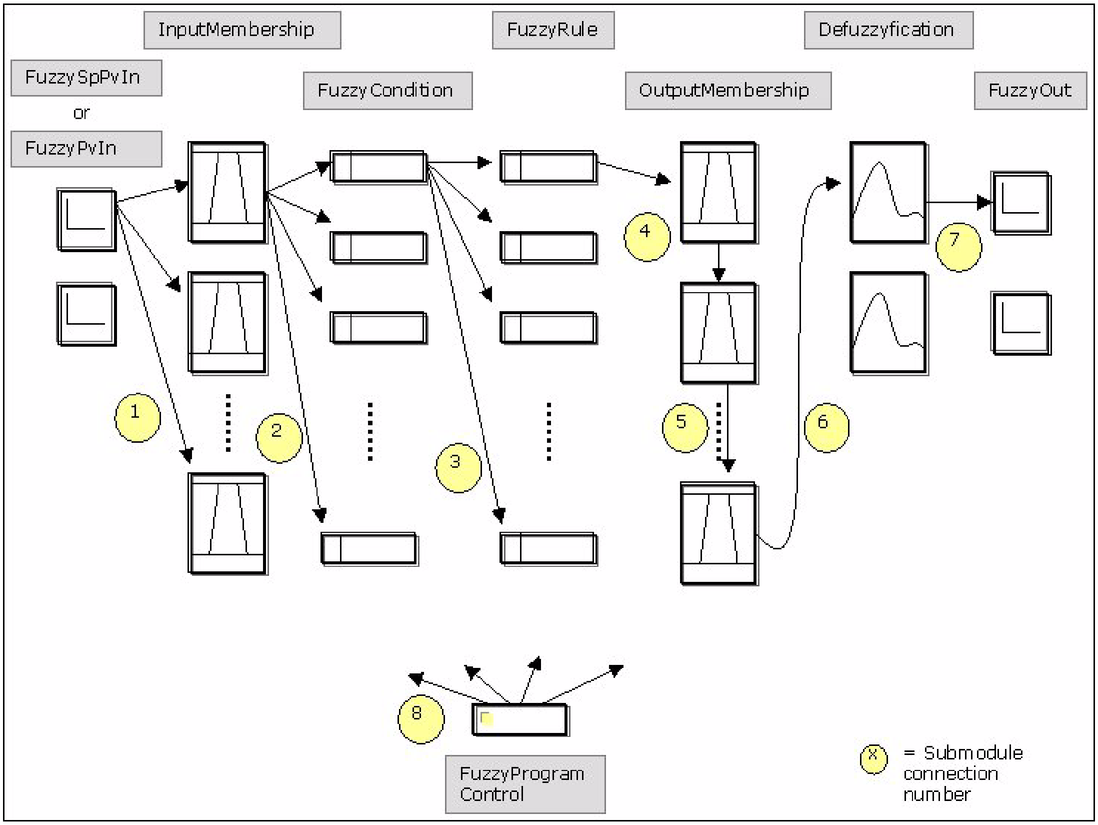


Figure 38. Sub-module connections in the fuzzy controller templates

1. The connection between FuzzySpPvIn and InputMembership: The FuzzySpPvIn control module computes the control deviation EOut (Setpoint-Process value) and its first and second derivatives. These signals are inputs to the InputMembership control modules. If a setpoint is not desired for some of the inputs FuzzyPvIn control modules are used instead of FuzzySpPvIn control modules.

2. The connection between InputMembership and FuzzyCondition: The control module defines an input membership function for the fuzzy logic part of the controller. The output, DegreeOfMembership, should normally be connected to all FuzzyCondition control modules. If the InputMembership control module is not connected to a certain FuzzyCondition control module then the membership function can, of course, not be used in the corresponding condition.
3. The connection between FuzzyCondition and FuzzyRule: The output parameter Condition of a number of FuzzyCondition control modules should be connected to the condition parameters of a number of FuzzyRule control modules. The fuzzy rules may then be defined from any of the connected fuzzy conditions.
4. The connection between FuzzyRule and OutputMembership: The output parameter DegreeOfSatisfaction should be connected to the corresponding parameter of an OutputMembership control module.
5. The connection between chained OutputMembership functions: The OutputMembership control modules are connected in a chain. The chain must always end with a Defuzzyfication control module. The control modules of the chain are connected via the parameters InputCurve and OutputCurve.
6. The connection between OutputMembership and Defuzzyfication: The Defuzzyfication control module should appear as the last link in a chain of OutputMembership control modules. The OutputMembership functions are connected to the chain via the parameter InputCurve.
7. The connection between Defuzzyfication and FuzzyOut: The parameter Output should be connected to the Input of the FuzzyOut control module.
8. The connection between FuzzyProgramControl and all the fuzzy logic part control modules: The Program control parameter should be connected to the corresponding parameter of all the InputMembership, FuzzyCondition, FuzzyRule and OutputMembership control modules.

Controller Modes

A controller has a number of different working modes. The controller may be switched from one mode to another with a minimum of disturbance in the process. The modes are listed in the table below with the lowest priority first.

Controller Mode	**Priority**	**Function**
Auto	Lowest	Auto is the normal automatic control mode.
Backtracking		The controller output is connected to a control module chain in which at least one of the succeeding controllers is in Manual mode.
Tracking		In Auto mode, the controller output tracks a signal value from the application when the *Track* parameter is True, except in Manual mode, which has higher priority. Upon changing from Auto mode to Tracking mode, or vice versa, the output is changed bumplessly, since it follows a ramp, limited by the maximum increase and decrease output ramp speed settings, until it reaches the track value.
Manual		The output signal can only be changed manually by a user in an interaction window, or via an application.
Tuning(1)	Highest	The Autotuner is started and active.

(1) Not available in PidSimpleCC and PidSimpleReal.

Design

Before using objects from the Control libraries, there are a number of choices that have to be made regarding which type of object to use for a specific purpose. The following information is designed to help you design reliable and effective analog control solutions:

- General Design Issues on page 107 describes things to consider and choices to be made before starting to create your analog control solutions.

For a more general discussion of design issues, see the *Compact Control Builder, AC 800M Planning (3BSE044222*)* Manual.

- Control Strategies on page 110 discusses what control strategy (what type of control loop, etc.) to use for different types of processes.
- Controller Types on page 113 contains an introduction to all controller types in the Control libraries.
- Industrial Controller Types on page 120 discusses how to build common industrial controller applications, such as cascade controllers, using types from the Control libraries.
- Signal Handling on page 123 gives an overview of where to find signal handling information.

General Design Issues

Analog signal handling and building of control loop applications using objects such as system functions and function blocks will often result in high functionality and a high degree of flexibility.

However, it requires a good deal of knowledge of control loop design and of the function of the participating objects, to construct and maintain signal systems and control loops.

Function Blocks or Control Modules?

An important choice is whether to use function blocks or control modules. For an extensive discussion of this topic, see the manual in *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*..

More specifically, you have to consider the following questions:

1. What is your general programming environment?
   What has already been done?
   What skills are required for present and future demands?
2. What kind of applications are you going to make, now and in the future? Is it
   a. signal handling
   b. advanced or simple control loop applications
   c. or a combination of these?

Table 5. Guide for choosing between function blocks and control modules.

Method	for signal handling	in control loop applications
Function Blocks	Suitable for calculations and when the signal information forwards is sufficient. Low memory consumption.	Suitable for control loops when the signal information forward is sufficient. Low memory consumption.
Control modules	Can be used when preceding and succeeding objects also are control modules.	Functions best when signal information forward and backward is required.

3. Which method is most suitable for designing a control loop?

   Basically, there are four methods of designing a control loop by means of the available libraries containing control functions, control function blocks, and control modules:

   – A ready-made control loop in function blocks,
   – System functions and function blocks,
   – Function blocks containing control modules,
   – Control modules.

A good basic strategy would in many cases be:

- Start with system functions and function blocks.

- When you need more control functions in many small, isolated islands of program code, you can create function blocks that contain control modules solving the control functions. Such a local group of control modules is then executed in the function block, according to its isolated terms, and blocks out influences from other function blocks.
- If your control system grows, so that it has to be coordinated and the code must be co-sorted, then programming with control modules is recommended, for example, in the case of several distributed cascade controllers.

4. Whether to use diagrams and diagram types for designing a control loop?

   If diagrams are used, it is possible to include the entire object-oriented design in a single diagram as it allows mixing control modules, function blocks, and functions, by means of graphical connections. The number of re-usable elements can also be reduced to a minimum as all of them can be included in a single re-usable diagram type, and used in many diagrams.

   Diagrams and diagram types allow you to configure the control logic in the project in a comprehensive graphical language called FD (Function Diagram). They allow mixing of the functions, function blocks, control modules, and other diagrams, in the same graphical editor. The diagrams also support cyclic communication between different applications, using communication variables.

   The diagrams provide a graphical overview of the application. In addition to the graphical code block that supports FD, the diagram and diagram type also supports SFC and ST code blocks, which are invoked from the main code block or sorted separately.

Control Modules and ControlConnection

In signal systems and control loops, a large amount of information is sent between different objects, both in the forward direction and backward. The main signal flow works well in normal operation. However, in exceptional situations ,there may be a need to handle, for example, the following situations.

- Integrator wind-up.
- Bumpless transfer.
- Signal quality.
- Signal measure ranges.

The complexity of such signal systems and control loops can be reduced considerably if the ControlConnection standard interface is used for signals between the objects, which then will have to be control modules. See ControlConnection on page 75.

By means of control modules it is possible, in addition to a high functionality and flexibility, to achieve a simplicity of configuration, which makes the control loops easy to configure and maintain. The risk of making mistakes when configuring control loops is drastically reduced, which increases the reliability of the loop.

Control Strategies

When a process is to be controlled, one of the most important questions is to select a controller strategy. Control strategies can be classified into the following main groups:

- Processes with no or short dead time, see Processes with No or Short Dead Time on page 111.
- Processes with long dead time, see Processes with Long Dead Time on page 111.
- Processes that do not fit the above two descriptions, see Special Processes on page 112.
- Rules of Thumb and Limitations on page 112 gives advice when using several controllers.

Processes with No or Short Dead Time

A process with no or short dead time can be of a number of types, for which the strategy is slightly different:

- **Process with Constant Process Dynamics**
  For a process with constant process dynamics and short dead times, that can have constant parameters in the whole working range, which is one of the most common processes, the proper strategy is to select a PID controller. Simpler variants P, PI or PD may also be sufficient.

  The process engineer's trimming tool for PID controllers is the Autotuner, which suggests settings for the parameters of the controller.
- **Process with Changing but Predictable Process Dynamics**
  For a process with changing but predictable process dynamics and short dead time that requires different parameters in different parts of the working range, the proper strategy is to use a PID controller with gain scheduling.

  The Autotuner is used to tune the parameters in each working range.
- **Process with Changing but Unpredictable Process Dynamics**
  For a process with changing but unpredictable process dynamics which vary slowly the proper strategy is to use an adaptive PID controller with or without gain scheduling.

  The Autotuner is used to tune the initial parameters. See the section Adaptive Control on page 169.

Processes with Long Dead Time

A process with long dead time can be of a number of types, for which the strategy is slightly different:

- **Process with Constant Process Dynamics**
  For a process with constant process dynamics with long dead times you may select a predictive PI controller, called a PPI controller.

  The PPI is started when configuring the PID controller and by selecting a maximum dead time. As a rule of thumb, a PPI controller is used when the dead time is longer than the dominating time constant in the process.

  When running a PPI controller it is still possible to run Autotuner and gain scheduling.

- **Process with Changing but Predictable Process Dynamics**
  For a process with changing but predictable process dynamics and long dead time, which requires different parameters in different parts of the working range, the proper strategy is to use a PPI controller with gain scheduling.

Special Processes

In special processes with several input and output signals which may not be possible to control, or when the strategies above, have proved unsuccessful, you may try a fuzzy controller.

The fuzzy controller is based on fuzzy logic which is a generalization of the common Boolean logic by something between true and false. See the section Fuzzy Controllers on page 119.

Rules of Thumb and Limitations

If you plan to use several controllers for a process, you should consider how the number of controllers influences the choice of processor for the control system. The following factors must be weighed against each other:

- The process time constant should not be less than 100 ms.
- When the time constant for a process is such that the controllers must execute faster than every 100 ms you should give execution times some extra thought, and ensure that the controller really has the time it requires, keeping in mind the associated program code.

Remarks on the Design of Control Loops

Configuration of Control Loops

The recommended configuration strategy for creating a control loop is to connect the control modules in chains from the left to the right, which is the direction of the main signal flow. If sufficient space is not available in the Control Module Diagram window, the control module icons may be turned in other directions. Their interaction windows are not influenced.

Control modules are preferably connected by means of graphical connections. This is done in Offline mode.

Connection to Tasks

The basic strategy for connections to tasks is to have all the control modules in a control loop running in the same task. If there is a need for faster action, particularly at the end of the control loop, these control modules can be connected to a quicker task, for example, in three-position control or for the slave controller in a cascade control loop.

Backtracking

In a control loop with several PID functions, where backtracking occurs, try to locate control modules that have an internal state as late as possible in the control loop chain. Otherwise when backtracking, a control module with internal state influences any succeeding TapCC and TapRealCC control module in a faulty way. The latter control modules would then tap off values and set levels based on the backtracked value (which is collected by the control module with an internal state) instead of a value from the preceding control module as expected.

Controller Types

Table 6 shows all controller types in the Control libraries

Table 6. Controller types in the Control libraries

Controller	Library	As Control Module Type	As Function Block Type
Simple	Simple Control		PidSimpleReal
	Standard Control	PidSimpleCC	
Standard	Basic Control		PidLoop PidLoop3P PidCascadeLoop PidCascadeLoop3P
	Standard Control	PidCC	
	Extended Control	BiasCC RatioCC	
Advanced	Advanced Control	PidAdvancedCC	

Table 6. Controller types in the Control libraries (Continued)

Controller	Library	As Control Module Type	As Function Block Type
Template[1]	Control Solution	SingleLoop CascadeLoop OverrideLoop FeedforwardLoop MidrangeLoop	
Template[1]	Control Object	Mimo22CC Mimo41CC Mimo44CC	
Fuzzy	Fuzzy Control	FuzzyController1CC, etc.	

[1] Control loop templates can be used directly in an application.

Getting Information on Individual Parameters

If you want to study individual parameters for a controller type, refer to online help for the type in question. To display online help for a controller type, select it in Project Explorer and press F1.

The corresponding online help topic will also contain an Editor button, which will open the corresponding editor, where you can see short descriptions for each parameter, as well as the corresponding data type.

It is also possible to generate project documentation for a library or part of it, by using the built-in Project Documentation function.

The Project Documentation is accessed from Project Explorer. Select a library or object and select **File > Documentation**. For more information on how to generate project documentation, see online help and the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)*..

Simple Controllers

Simple controller objects work according to Basic Controller Principles on page 87.

Table 7. Simple controllers

Type Name	Library	Type	Description
PidSimpleReal	Simple Control	Function Block	PidSimpleReal is a simple PID controller that supports backtracking, tracking and manual control. All transitions from limiting, tracking, and Manual mode are bumpless. Interaction graphics are also available, to support set-up and maintenance of the controller.
PidSimpleCC	Standard Control	Control module	PidSimpleCC is a low-functionality PID controller, which is less time and memory consuming than the full-functionality versions. Interaction graphics are also available, to support set-up and maintenance of the controller. The main inputs and the output are of *ControlConnection* type, which means that backtracking and limiting are managed automatically.

Standard Controllers

Standard controller types work according to Basic Controller Principles on page 87.

Table 8. Standard controllers

Type Name	Library	Type	Description
PidCC	Standard Control	Control module	PidCC is a full-function PID controller.
PidLoop PidLoop3P PidCascadeLoop PidCascadeLoop3P	Basic Control	Function block	PidLoop, PidLoop3P, PidCascadeLoop, and PidCascadeLoop3P are ready-made complete control loops that you can connect to I/O signals of *RealIO* type. The controllers in these control loops can be configured as P, PI, PD, or PID controllers, with the same functions as PidCC. However, these function block types cannot be connected to other function blocks or have any function block or control module inserted into the control loop.

The PidCC control module type has the following main functions:

- Autotuner of relay type, see Autotuning on page 162.
- Feedforward, see Feedforward on page 161.
- Tracking, see Backtracking on page 82.
- Deviation alarm generation, see Deviation Alarms on page 159.
- Limitation of output, see Limitation of Controller Output on page 94.
- Anti-integrator wind-up, see Anti-Integrator Wind-Up Function on page 153.
- Bumpless transfer, Bumpless Transfer on page 154.
- Dead zone for the control deviation, Additional Control Functions on page 177.
- Setpoint ramping, see Internal Setpoint Ramping on page 94.
- Autotuner structure selection, see Autotuning on page 162.
- Automatic offset adjustment, see Offset Adjustment on page 93.

- ERF, see ERF on page 96.
- Disable PD at limited windup, see Disable PD at limited windup on page 97.
- Enable Out Ramp Manual, see Enable Out Ramp Manual on page 97.
- Epsilon, see Gain Scheduling on page 173.

To supervise the control deviation, relative alarms can be given by two control deviation alarm limits, one for positive and one for negative deviation. Information is given for each level in two forms; as alarms and as Boolean parameters.

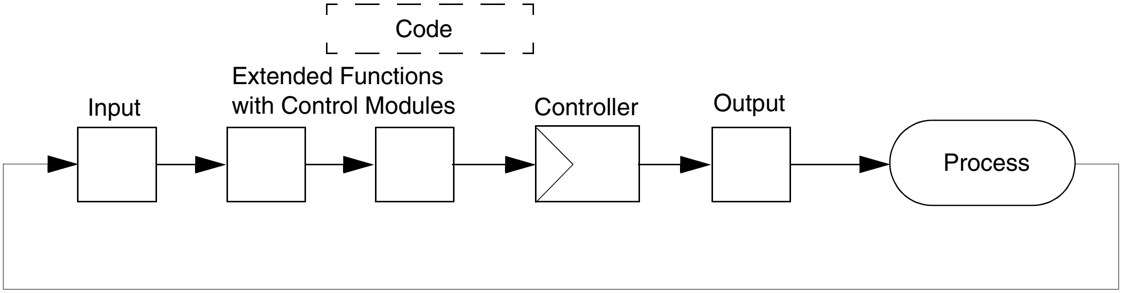


Figure 39. A control loop with all functions defined by control modules from the control libraries

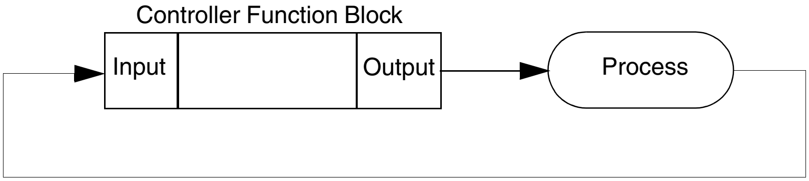


Figure 40. A control loop defined by a function block

Advanced Controllers

PidAdvancedCC works according to Basic Controller Principles on page 87.

Table 9. Advanced controllers

Type Name	Library	Type	Description
PidAdvancedCC	Advanced Control	Control module	PidAdvancedCC is the most advanced controller in the Control libraries, see list below.

In addition to the main functions of PidCC, PidAdvancedCC has the following, more advanced functions:

- Configurable as a PPI (Predictive PI controller), see PPI Controller on page 85.
- Autotuning using relay and step response methods, see Autotuning on page 162.
- Gain scheduling, see Gain Scheduling on page 173.
- Adaptive control, Adaptive Control on page 169.
- Oscillation detection, see Additional Control Functions on page 177.
- Sluggish control detection, see Additional Control Functions on page 177.

! PidAdvancedCC generates an event each time a parameter is changed. This means that you must be careful so that you do not flood the alarm and event servers by connecting a parameter to a variable that changes very often.

Fuzzy Controllers

A fuzzy controller may handle the case of one input and one output, as well as many inputs and many outputs. It has most of the functions of a PID controller together with the possibility of defining fuzzy logic rules for process control.

Table 10. Fuzzy controllers

Type Name	Library	Type	Description
FuzzyControllerXCC	Fuzzy Control	Control module	For additional information, see online help for the Fuzzy Control library and Fuzzy Control on page 99.

A fuzzy controller should not be used in cases where PID control works well. In these cases it is much easier to tune a PID controller.

However, in cases where PID control fails or works poorly, a fuzzy controller may be successful. For example, the fuzzy controller may be successful:

- When the process is truly multi-variable, with many coupled inputs and outputs.

This may also have a solution using two PID controllers combined with a decoupling filter (PidCC and DecoupleFilterCC).

- When the process has non-linearities that are difficult to handle with PID control and gain scheduling.
- When the process is difficult to describe analytically, and operators control it manually, by experience.

For a short introduction to fuzzy control, see Fuzzy Control on page 99.

Industrial Controller Types

Controllers regarded as common in industry can easily be constructed by means of the control modules in the Control libraries according to the typical examples below. For more in formation, see Control Loop Solutions on page 226.

Cascade Controller

A cascade controller is constructed as a combination of control modules, using input and output modules, and two controller modules. Two controllers are connected in cascade; the output of one controller, called the master, is connected to the external setpoint of the other controller, called the slave.

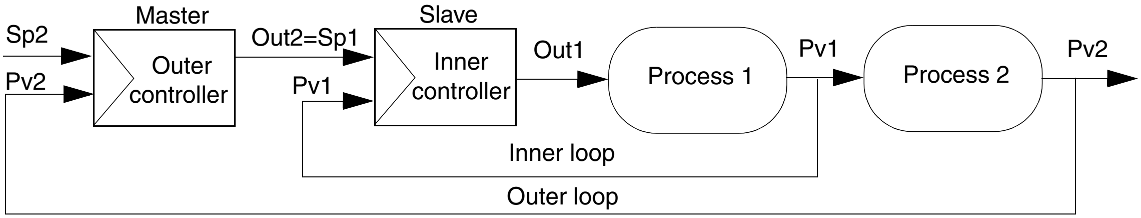


Figure 41. Illustration of two controller modules connected in cascade

Two controllers connected in cascade must be tuned in the correct sequence. The inner loop should be faster than the outer loop. Ready-made function blocks are available for applications in which a fixed cascade loop is suitable.

Three-Position Controller

A three-position controller is constructed as a combination of control modules: a controller module, an analog input module and a three-position output module, which gives two digital output signals. Use a three-position controller when digital output are required for an increasing, decreasing or no digital signal at all to be sent, for example, to an electrical actuator. See the section Three-Position Output on page 177. For setting of ranges in a SplitRange or a MidRange object, see Split Range Examples on page 238.

Pulse Controller

A pulse controller is constructed as a combination of control modules: a controller module, an input module and an output module, which gives a digital output signal with a pulse width proportional to the analog controller output.

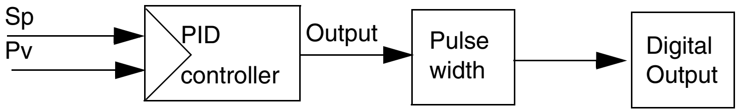


Figure 42. Pulse control with the digital pulse width output proportional to the analog output

Ratio Controller

A ratio controller can be constructed from a combination of control modules: input and output modules, controller modules, arithmetic modules and tap modules. The ratio between two different process values may be controlled by two or more controller modules, according to Figure 43. A RatioCC control module is then used for the ratio between the setpoints. Ratio controllers are often used for recipe handling.

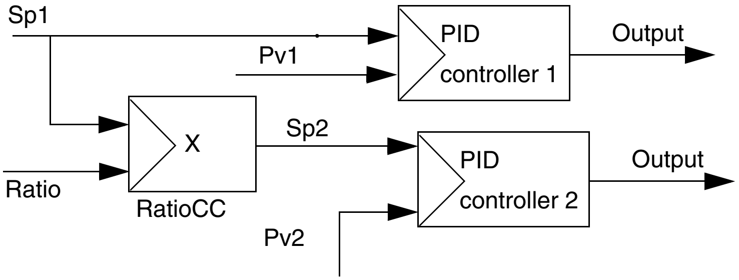


Figure 43. Example of ratio controllers where the setpoint from the first controller is multiplied by a factor to obtain the setpoint for the second controller

Split-Range Controller

A split-range controller is constructed as a combination of the following control modules: input and output modules, a controller module and any of the branching modules. The output from a controller is then divided into two ranges which can be set independently, and may overlap each other. An example of the use of split-range control is when two control valves are used. The valves may be of different sizes working in the same direction. The lower range is connected to the smaller valve and when it is fully open, the upper range opens the larger valve. The valves may also work in opposite directions. For example, when a tank level is to be controlled. At lower levels, one valve opens, and at higher levels, the other valve opens.

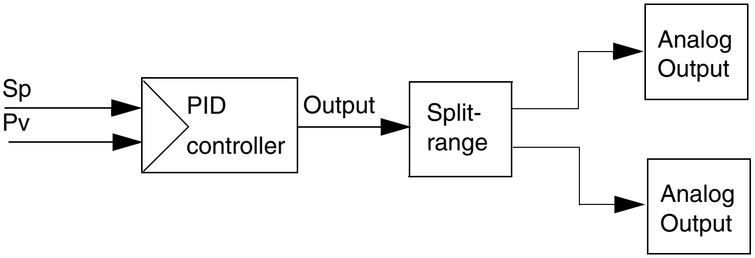


Figure 44. Example of split-range control where the output range from the controller is divided into two branches with different ranges

Signal Handling

The Control libraries contain a number of function blocks and control modules for signal handling. There are also types for signal handling in the Basic and Signal libraries.

For a list and description of available signal handling objects, see Advanced Functions on page 152.

Control Builder online help also contains additional information on specific signal handling objects. Select the object in Project Explorer, and press F1 to display online help for a type.

Getting Started with ControlConnection

This section describes how to create a control module template that can connect to ControlConnection, thus talk to other objects with ControlConnection.

After reading this subsection you will learn:

- The relationship between code-blocks and data flow directions.
- The concept of ControlConnection Gate modules.
- Step-by-step for creating a ControlConnection template.

What is ControlConnection?

ControlConnection is a structured data type for handling signals between control modules in both forward and backward directions. It is a very effective structured data type from the outside, but difficult to connect inside an object.

The difficulties lie in the structure itself, which means that other control modules must fulfil the relation and condition specification for (ControlConnection) signal traffic. For that reason you are going to be introduced to the ControlConnection Gate modules that will transform your local signals automatically to ControlConnection.

At the end of this subsection you will also learn how to create a ControlConnection template (CC template) from scratch. The template will help you overcome the most common difficulties there is by having local code reading/writing signals to/from ControlConnection.

For a more theoretical presentation of ControlConnection, see ControlConnection on page 75.

Dealing with Data Flow Directions

In order to process signals of ControlConnection your control module must be designed to handle data in both forward and backward directions. The best way to accomplish this is to assign one code block for each direction. Thus one code block *Forward* and one code block *Backward*.

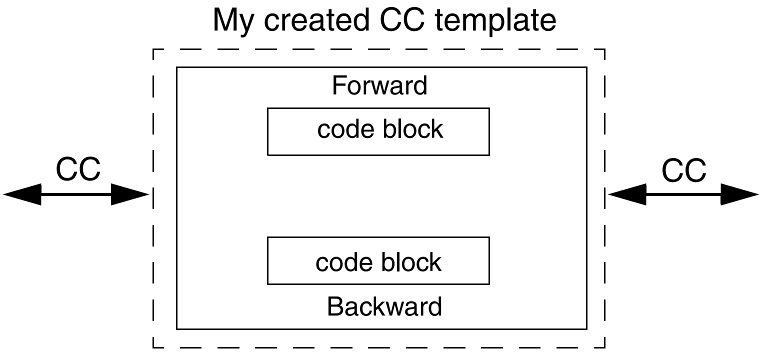


Figure 45. A control module prepared with two code blocks for handling forward and backward directions

This guideline of having one code block for each direction, should apply to all control modules that are processing signals of Control Connection. However, there are some exceptions (as always), first object in the chain, the *Source* (AI object) and the last object the *Sink* (AO object) only needs one code block. You will learn more about this under Code Sorting Order Backwards on page 125.

Code Sorting Order Forward

After establishing the need for two code-blocks (Forward and Backward) in the control module, it is time to study how the code sorting order works for ControlConnection. Remember, it is not just the two (Forward and Backward) blocks alone that should be sorted, but all the outer code blocks as well. However, the key is the ControlConnection's sort order mechanism which is very helpful. Provided that we have made all the necessary in/out arrangements, it will always execute forward directions before backward directions.

You will learn all about In/Out parameter connections later in Creating a Control Module with ControlConnection (CC template) on page 130.

This means that ControlConnection will always start from left with the first forward block in a chain of control modules (Figure 46). As long as the module is positioned correctly in the control module chain, it will be automatically sorted.

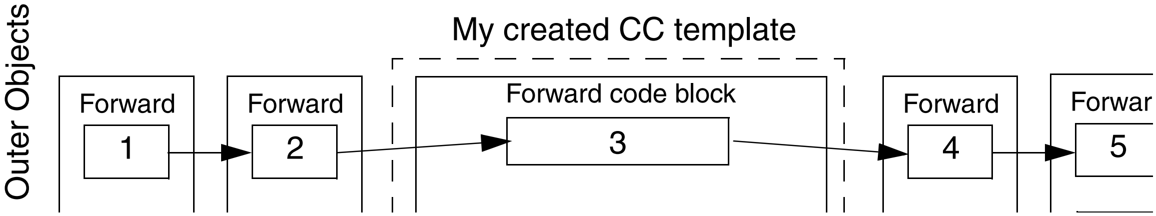


Figure 46. ControlConnection will always start with forward block (1), and then execute forward block (2), your forward block (3) and then (4), (5) etc.

Code Sorting Order Backwards

Next step is to learn which object that will change the data flow direction, or start passing information backwards via the backward blocks. Just as before with the forward block's code sorting order, the same principle applies for the backward blocks, but now only backwards.

There are two (predictable) things that can stop further forward executions. It is either a Sink object at the end of the chain (most common solution) or a code sorting variable in the backward block.

Therefore, before building your ControlConnection module, consider if your control module should have a generic solution (continue to passing forward information through the chain) or if your module should be the end object.

Generic Solution with a Sink Object

A generic solution continues to passing the information to the next forward block, which means that it relies on a Sink object (AO) at the end of the chain. A Sink Object (AO) is the most common to use at the end of the chain in a generic solution. It is the Sink that will change the data flow direction to go backwards (Figure 47).

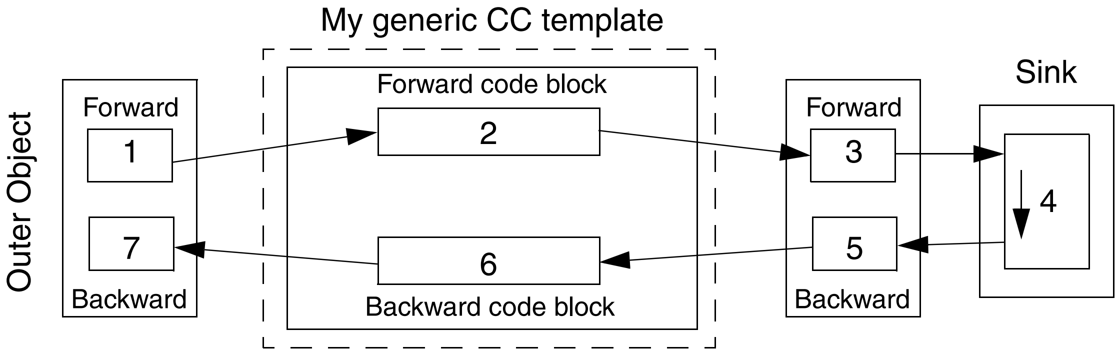


Figure 47. My generic CC template object has a generic solution which simply passes information forward to the next forward object

As you can see in Figure 47 the Sink object contains both forward code and backward code in one code block. The backward direction sort order starts at the end of the Sink's common code block. The Sink is writing to the backward block, in this case No (5) in Figure 47.

End Object Solution

To build an end object means basically two things;

1. Your forward block will be executed *last* among all outer forward blocks in the chain of control modules.
2. Your backward block will be executed *first* among all outer backward blocks.

Building an end object solution is done by start declaring a "dummy" variable (for example CodeSortVar). Then you let the variable CodeSortVar read (in the backward code block) a value that was previously written (for example OldValue1) in the corresponding forward code block. See Figure 48.

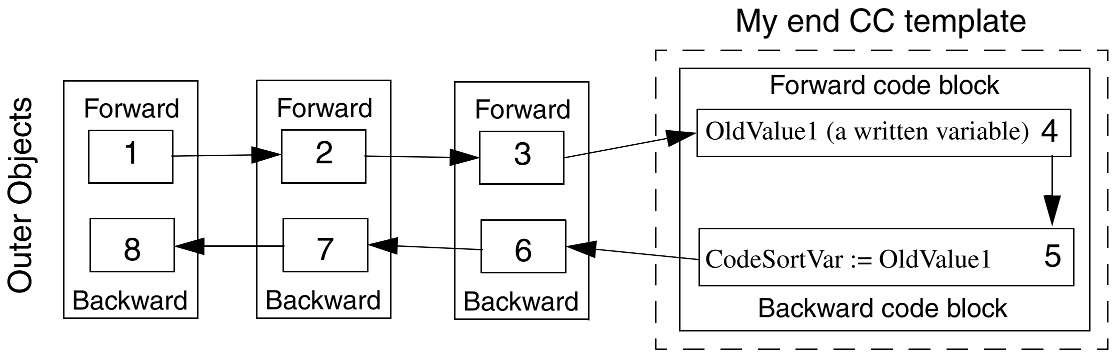


Figure 48. My end CC template object has an end solution. It stops passing information forward and begins passing information backwards to the next backward block

The end CC template object has created a dependency between the forward block and the backward block. Since the backward block is reading the variable (*OldValue1*) must the forward block be executed before the backward block. The variable *OldValue1* was previously written in the forward block.

You will be able to study more of this in the subsection Creating a Control Module with ControlConnection (CC template) on page 130.

Next, you will learn about the Gate modules and how they transform local signals to ControlConnection.

Open the Gates to ControlConnection

The main difficulties of having own control modules talking ControlConnection is to fulfil the specification for ControlConnection. However, by using Gate modules you do not have to worry about that. As the name applies a Gate module check the signals that come in and out from your control module and capsulated your execute code safely between them.

An IN Gate module will for example read an In signal of ControlConnection and pass it over to one of your local variables. An OUT Gate will transform the executed code value to a ControlConnection signal.

In short, you use the Gate modules to fulfil the specification for Control Connection.

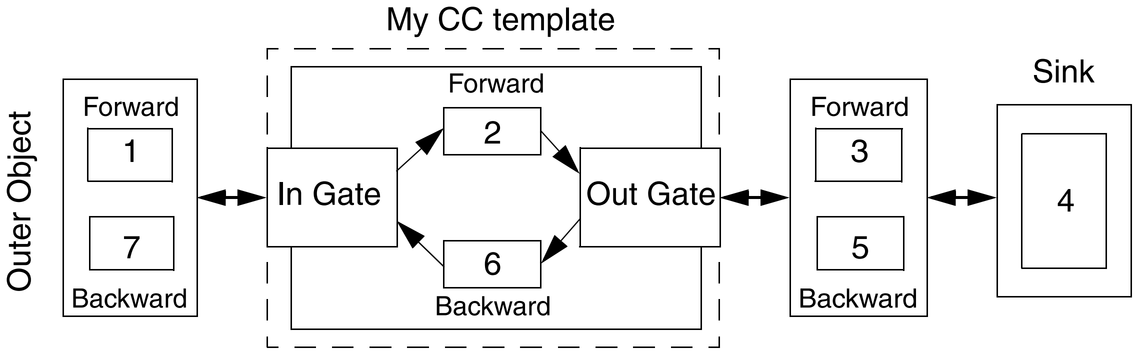


Figure 49. My CC template object is protected by the Gate modules on both sides to ensure a secure signal traffic with ControlConnection

There are no code-blocks illustrated in the Gate modules (see Figure 49) although they contain both forward and backward blocks. They are merely there in the background for syntax control reasons.

Ownership of the Gate Variables

When you are working with the Input Gate and the Output Gate you are going to deal with two variables for each Gate. These two variables will have different ownership, or write permission when reading and writing to your code blocks (see Figure 50).

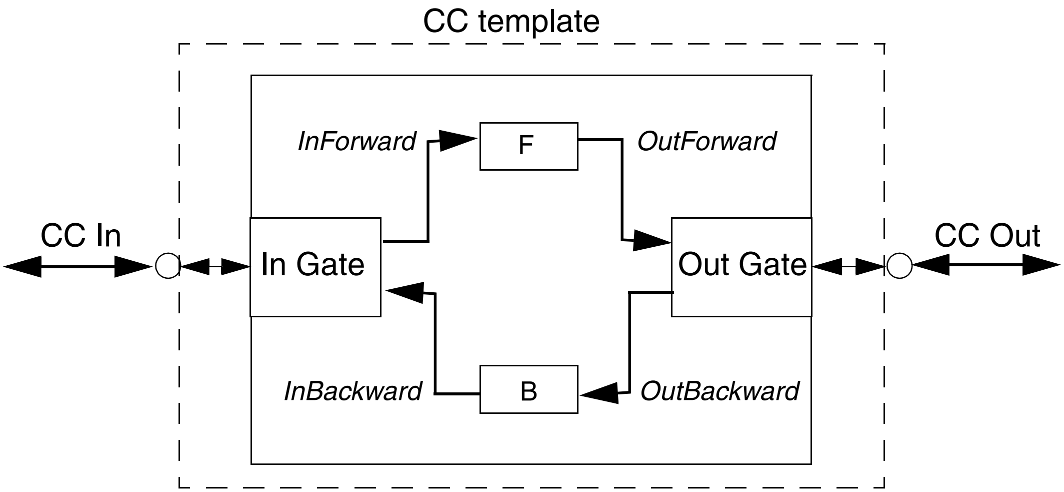


Figure 50. Gate Variables writing permission

These four local variables (*InForward*, *InBackward*, *OutForward* and *OutBackward*) are local variables that are connected to the Gate parameters *Forward* and *Backward* respectively. However, these local variables have different permissions.

From Figure 50:

- **InForward** which is connected to the In Gate's *Forward* parameter, is own by the Gate, thus writing is **not** allowed in the forward code block.
- **InBackward** which is connected to the In Gate's *Backward* parameter, is own by your control module (CC template), thus writing is allowed in backward code block.
- **OutForward** which is connected to the Out Gate's *Forward* parameter, is own by your control module (CC template), thus writing is allowed in forward code block.
- **OutBackward** which is connected to the Out Gate's *Backward* parameter, is own by the Gate, thus writing is **not** allowed in the backward code block.

There are four Gates provided for you in the standard library BasicLib, *CCInputGate*, *CCOutputGate*, *CCInputGateExtended* and *CCOutputGateExtended*. You can learn the differences between the Gates in Control Builder online help.

Creating a Control Module with ControlConnection (CC template)

This step-by-step example builds a control module that calculates the average value of the four latest forward values on a ControlConnection node. The ControlConnection node always involves only the control module types, and not the function block types.

Although you can choose a single control module, it is strongly recommended that you create your CC template from a control module type. A control module type can be re-used in many applications, but a single control module cannot be reused.

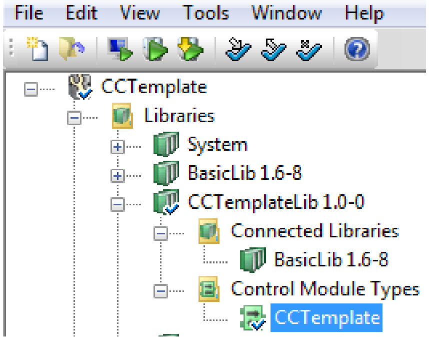
This example assumes that you have experience of (at least) basic Control Builder skills, involving creating and connecting new objects based on types, etc.

The instructions in this example are merely providing you with an idea of a working order. They do not always represent the exact order of events you will meet in Control Builder (instructions handling context menus, buttons, Save before close etc. have been intentionally neglected).

Create a new CC template object

From the Project Explorer:

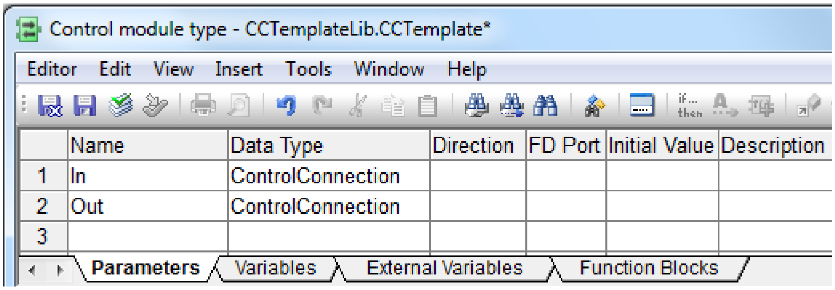
1. Create a new Project with a AC 800M template and name it CCTemplate.
2. Create a New Library and name it CCTemplateLib.
3. Connect the BasicLib into your new CCTemplateLib.
4. Create a new control module type and name it CCTemplate.



Declare parameters and variables

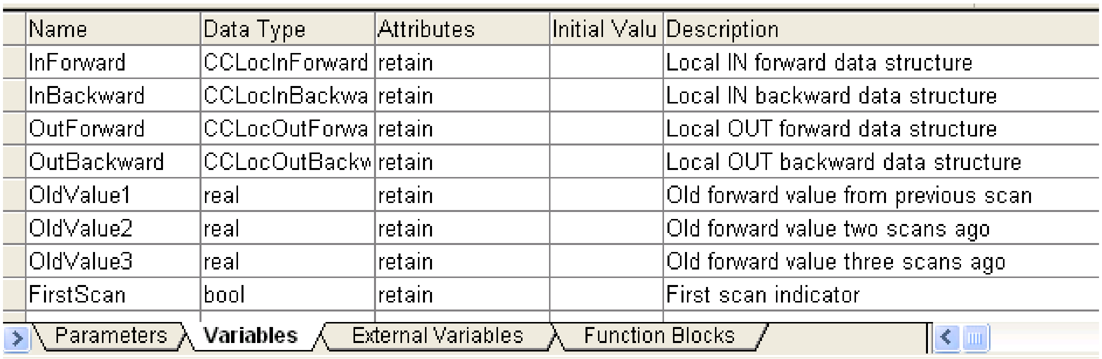
Open the declaration editor for the CCTemplate object.

1. Declare an **In** and **Out** parameter of data type ControlConnection.



	Name	Data Type	Direction	FD Port	Initial Value	Description
1	In	ControlConnection				
2	Out	ControlConnection				
3						

2. Next, declare your local variables according to Figure 51.



Name	Data Type	Attributes	Initial Valu	Description
InForward	CCLocInForward	retain		Local IN forward data structure
InBackward	CCLocInBackwa	retain		Local IN backward data structure
OutForward	CCLocOutForwa	retain		Local OUT forward data structure
OutBackward	CCLocOutBackw	retain		Local OUT backward data structure
OldValue1	real	retain		Old forward value from previous scan
OldValue2	real	retain		Old forward value two scans ago
OldValue3	real	retain		Old forward value three scans ago
FirstScan	bool	retain		First scan indicator

Figure 51. Declared variables in the CCTemplate object

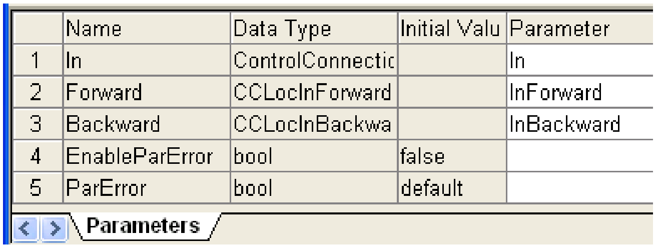
As you can see, the Gate variables *InForward*, *InBackward*, *OutForward* and *OutBackward* are of structured data types. It is these four local variables that will talk directly to the Gate modules. You will learn about their components when you are programming the forward and backward code blocks. The other four variables are used in the code blocks.

You can also learn the naming convention for parameters and variables in the manual *Library Objects Style Guide, Introduction and Design (3BSE042835*)*.

Connecting the Gate modules

Next, create instances of the Gate modules in the CCTemplate. The Gate modules (CCInputGate and CCOutputGate) are located in the BasicLib.

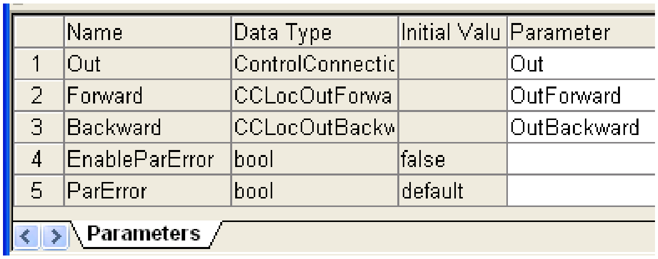
1. Right-click CCTemplate and create an instance of CCInputGate. Name it CCInputGate.
2. Connect the Input Gate module according to Figure 52.



	Name	Data Type	Initial Valu	Parameter
1	In	ControlConnectic		In
2	Forward	CCLocInForward		InForward
3	Backward	CCLocInBackwa		InBackward
4	EnableParError	bool	false	
5	ParError	bool	default	

Figure 52. CCInputGate connected to the CCTemplate module

3. Right-click CCTemplate and create an instance of CCOutputGate. Name it CCOutputGate.
4. Connect the Output Gate module according to Figure 53.



	Name	Data Type	Initial Valu	Parameter
1	Out	ControlConnectic		Out
2	Forward	CCLocOutForwa		OutForward
3	Backward	CCLocOutBackw		OutBackward
4	EnableParError	bool	false	
5	ParError	bool	default	

Figure 53. CCOutputGate connected to the CCTemplate module

After connecting the two Gate modules to the CCTemplate, the result in Project Explorer should look like Figure 54.

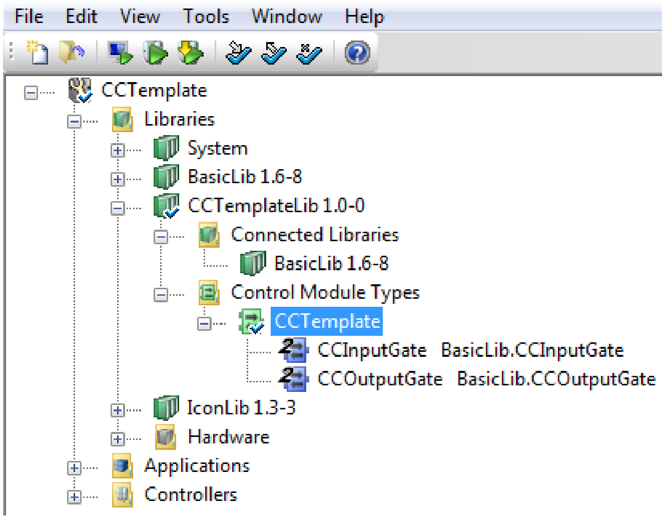


Figure 54. The Gates to ControlConnection are connected to the CCTemplate module

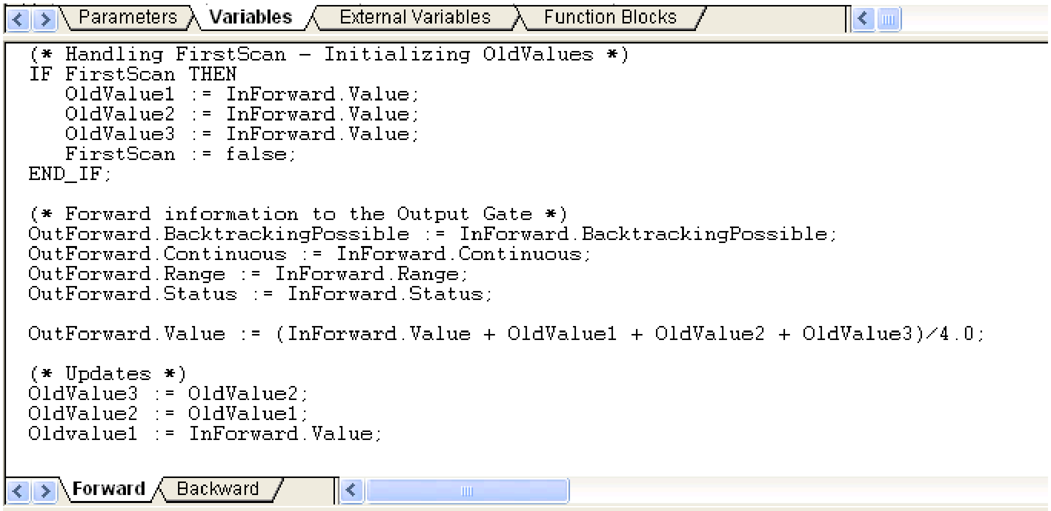
Programming Forward and Backward Code

1. Open the Programming Editor for CCTemplate.
2. Re-name the code block *Code* to **Forward**.
3. Right-click the **Forward** tag and select **Insert** from context menu. A dialog will open.
4. Accept default Languages selection (ST) and name the new code block **Backward**.
5. Select the Forward tag and write the following programming code.

Each variable of a structured data type (e.g. InForward) has a component menu attached. Open the menu by typing a dot (InForward**.**) directly after the variable in the code block and select the component.

```
(* Handling FirstScan - Initializing old values *)
IF FirstScan THEN
```

```
    OldValue1 := InForward.Value;
    OldValue2 := InForward.Value;
    OldValue3 := InForward.Value;
    FirstScan := false;
END_IF;

(* Forward the information to the Output Gate *)
OutForward.BacktrackingPossible := InForward.BacktrackingPossible;
OutForward.Continuous := InForward.Continuous;
OutForward.Range := InForward.Range;
OutForward.Status := InForward.Status;

OutForward.Value := (InForward.Value + OldValue1 + OldValue2 +
OldValue3)/4.0;

(* Updates *)
OldValue3 := OldValue2;
OldValue2 := OldValue1;
OldValue1 := InForward.Value;
```

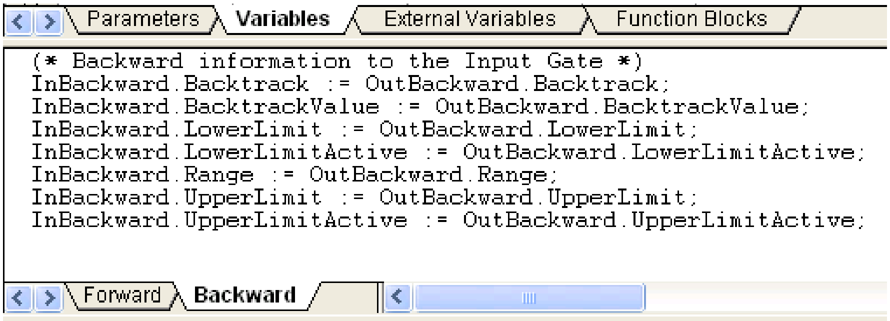


```
(* Handling FirstScan - Initializing OldValues *)
IF FirstScan THEN
   OldValue1 := InForward.Value;
   OldValue2 := InForward.Value;
   OldValue3 := InForward.Value;
   FirstScan := false;
END_IF;

(* Forward information to the Output Gate *)
OutForward.BacktrackingPossible := InForward.BacktrackingPossible;
OutForward.Continuous := InForward.Continuous;
OutForward.Range := InForward.Range;
OutForward.Status := InForward.Status;

OutForward.Value := (InForward.Value + OldValue1 + OldValue2 + OldValue3)/4.0;

(* Updates *)
OldValue3 := OldValue2;
OldValue2 := OldValue1;
Oldvalue1 := InForward.Value;
```

Figure 55. Programming code in the Forward code block

6. Select the Backward tag and write the following programming code.

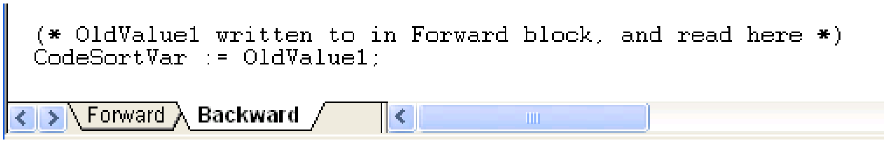
```
(* Backward information to the Input Gate *)
InBackward.Backtrack := OutBackward.Backtrack;
InBackward.BacktrackValue := OutBackward.BacktrackValue;
InBackward.LowerLimit := OutBackward.LowerLimit;
InBackward.LowerLimitActive := OutBackward.LowerLimitActive;
InBackward.Range := OutBackward.Range;
InBackward.UpperLimit := OutBackward.UpperLimit;
InBackward.UpperLimitActive := OutBackward.UpperLimitActive;
```



```
(* Backward information to the Input Gate *)
InBackward.Backtrack := OutBackward.Backtrack;
InBackward.BacktrackValue := OutBackward.BacktrackValue;
InBackward.LowerLimit := OutBackward.LowerLimit;
InBackward.LowerLimitActive := OutBackward.LowerLimitActive;
InBackward.Range := OutBackward.Range;
InBackward.UpperLimit := OutBackward.UpperLimit;
InBackward.UpperLimitActive := OutBackward.UpperLimitActive;
```

Figure 56. Programming code in the Backward code block

If you need the End module functionality, then add the following to your code. Declare the CodeSortVar variable as a real with no attribute (empty field) in the declaration editor.



```
(* OldValue1 written to in Forward block, and read here *)
CodeSortVar := OldValue1;
```

Figure 57. Code added for End module functionality

Adding Graphical Nodes

This topic is not applicable if only the diagram editor, and not the CMD editor, is used for creating the control loop. See Create an Instance of CCTemplate in the Diagram on page 140.

After you are done with this subsection your CC Template will contain a name area, two connected graphical nodes in the CMD Editor (Figure 58).

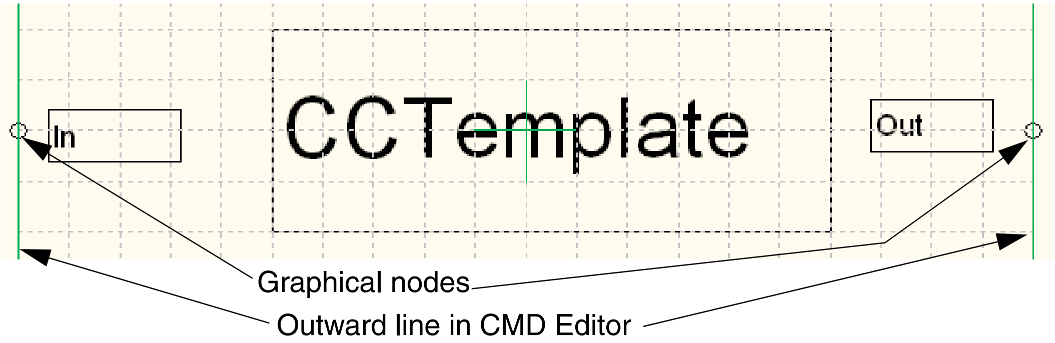


Figure 58. CCTemplate in the CMD Editor

From Project Explorer with the programming editors closed.

1. Right-click **CCTemplate** module and select **CMD Editor** in the context menu.
2. Select icon for Rectangle (Figure 59) and mouse-click a rectangle over the outward line (see the outward line in Figure 58).

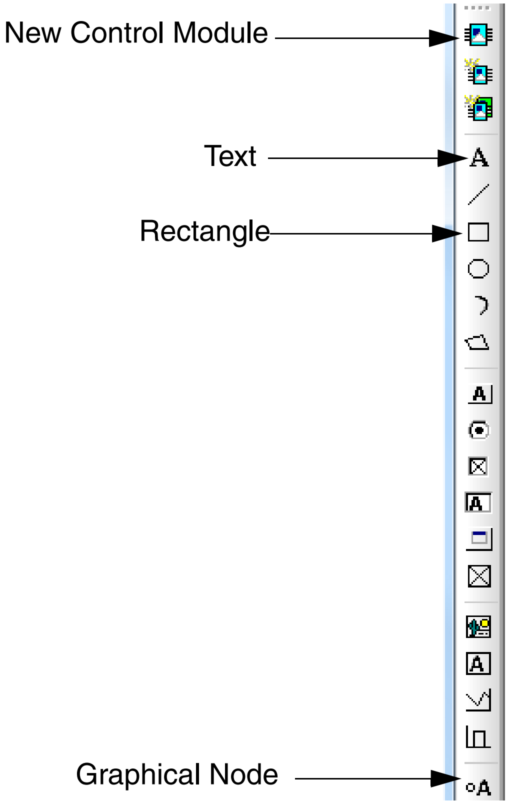


Figure 59. Icon menu in the CMD Editor

3. Select icon for Text (Figure 59) and write CCTemplate (Figure 58).

Before you can add a graphical node, first declare the corresponding parameters. In this case you have already declared your parameters (In and Out) in the CCTemplate parameter editor.

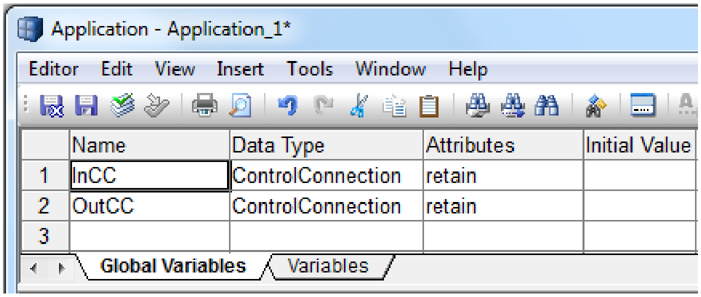
Adding a graphical node is done with three (left) mouse-clicks. First click will add a node, second click will start a rectangle field (move the cursor), and third mouse-click will release the rectangle. After the third click, type in the parameter name.

4. Select icon for Graphical Node (Figure 59) and add two nodes on both sides of your Text area (see exact location in Figure 58). Type in parameter **In** and **Out** in the rectangle.
5. Close the CMD Editor when done. The CMD Editor should look like Figure 58.

Create an Instance of CCTemplate in the Application

Next, you will learn how to create an instance (control module) of your CCTemplate control module type in the application.

1. Connect your CCTemplateLib to the application.
2. Declare two global variables (InCC and OutCC of ControlConnection) in the Application according to Figure 60.


Application - Application_1*
Editor Edit View Insert Tools Window Help
Global Variables
Variables

	Name	Data Type	Attributes	Initial Value
1	InCC	ControlConnection	retain	
2	OutCC	ControlConnection	retain	
3				

Figure 60. Global variables for connecting the CCTemplate object in the application

3. Open the CMD Editor in the Application and select the icon for New Control Module (see Figure 59).
4. In the dialog select **CCTemplateLib** and then select **CCTemplate** as your control module type. Name your Instance to **CCTemplate**.
5. Click **OK**.
6. Left mouse-click a box to a suitable size and release left mouse-click. A connection window opens.
7. Connect In and Out with InCC and OutCC, respectively. Save and Close.

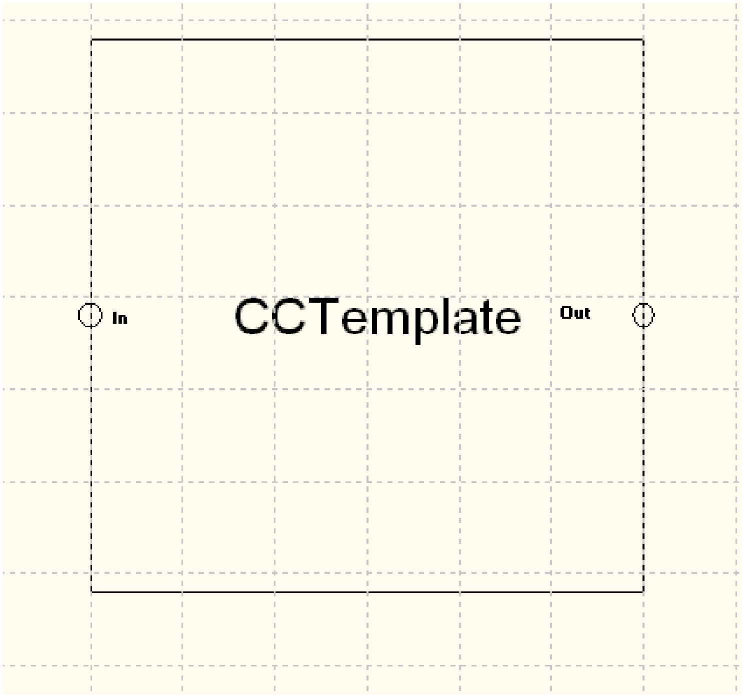


Figure 61. Instance of CCTemplate in the CMD editor of application.

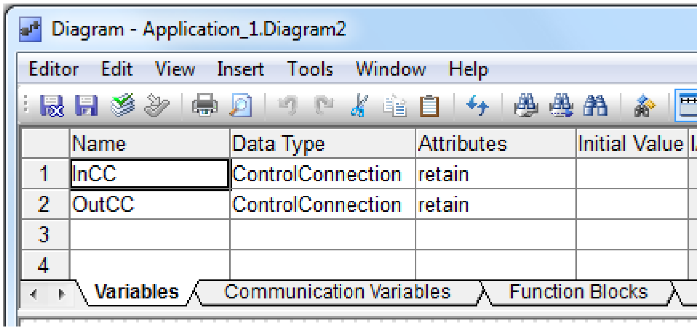
Create an Instance of CCTemplate in the Diagram

In this topic, you will learn how to create an instance (control module) of CCTemplate control module type in a diagram under the application.

The default application, *Application_1*, contains three diagrams. Let us create an instance of CCTemplate in *Diagram2*.

1. Connect CCTemplateLib to the application.
2. Right click Diagram2 and select Editor.

3. In the declaration pane of diagram editor, declare two variables (InCC and OutCC of ControlConnection) according to Figure 62.



	Name	Data Type	Attributes	Initial Value	I
1	InCC	ControlConnection	retain		
2	OutCC	ControlConnection	retain		
3					
4					

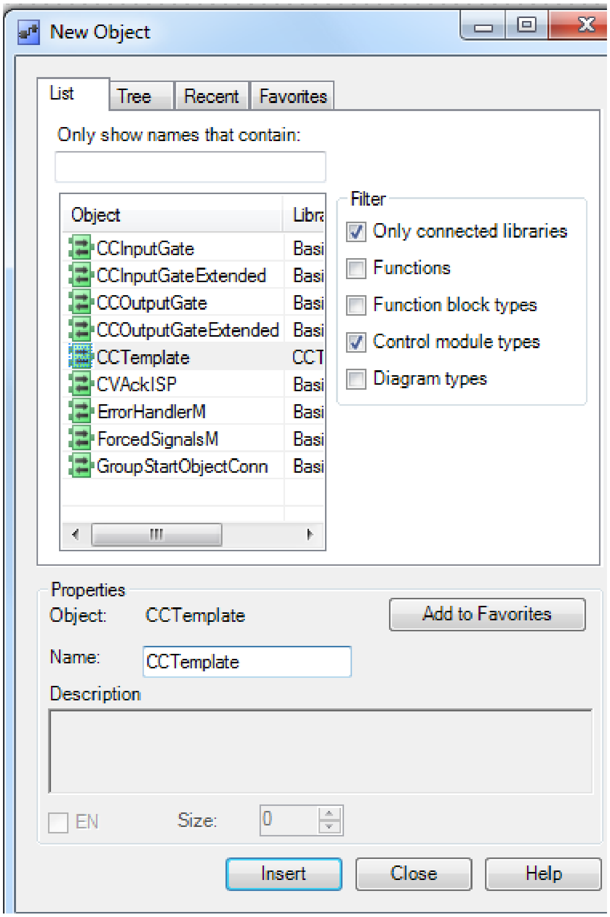
Figure 62. Variables for connecting the CCTemplate object in the diagram

4. In the graphical code block *Code*, insert an instance of CCTemplate control module type:

   a. Right click in the grid area, and select **New > Object**.

   b. In the New Object dialog box, select **CCTemplate** under the List tab. In the Name field, enter the name of the instance as *CCTemplate* (see Figure 63)

   c. Click **Insert**. The CCTemplate control module is inserted as shown in Figure 64.



Only show names that contain:

Object	Libr
CCInputGate	Basi
CCInputGateExtended	Basi
CCOutputGate	Basi
CCOutputGateExtended	Basi
CCTemplate	CCT
CVAckISP	Basi
ErrorHandlerM	Basi
ForcedSignalsM	Basi
GroupStartObjectConn	Basi

Filter

- ☑ Only connected libraries
- □ Functions
- □ Function block types
- ☑ Control module types
- □ Diagram types

Properties

Object: CCTemplate

Name: CCTemplate

Description

□ EN Size: 0

Figure 63. New Object dialog box

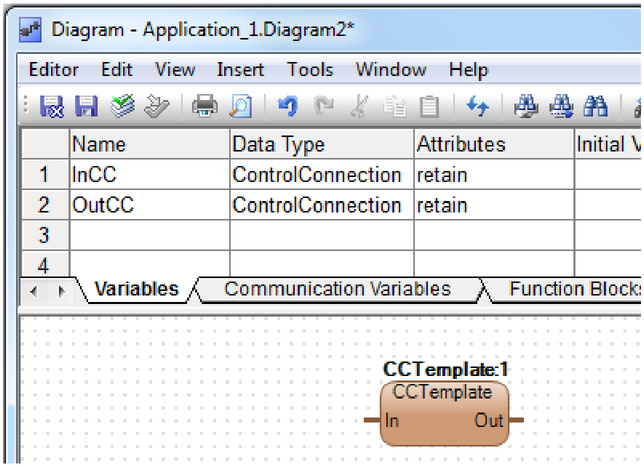


Diagram - Application_1.Diagram2*
Editor Edit View Insert Tools Window Help
Variables
Communication Variables
Function Block:
CCTemplate:1
CCTemplate
In
Out

	Name	Data Type	Attributes	Initial V
1	InCC	ControlConnection	retain	
2	OutCC	ControlConnection	retain	
3				
4				

Figure 64. CCTemplate inserted in the graphical code block

5. Connect the In and Out ports to the variables InCC and OutCC, respectively:

   a. Right click In port, and select **Connect**. In the Connect dialog, type InCC. Click **OK**.

   b. Right click Out port, and select **Connect**. In the Connect dialog, type OutCC. Click **OK**.

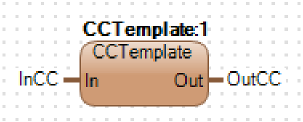


Figure 65. CCTemplate object with variables connected

6. Save and Close the diagram editor.

What next?

After completing the CC template example, you learned how the Gate modules work and how to adapt the CC template to your own solutions on ControlConnection.

However, if you need more functionalities, the Control Object library contains three additional ControlConnection templates (Mimo22CC, Mimo41CC, and Mimo44CC) with more advanced functionalities. See Control Object Control Modules on page 474 for more details on these templates.

Connecting a Mimo22CC Object to a New Application

This topic is not applicable if only the diagram editor, and not the CMD editor, is used to connect the MimoXXCC object. See Connecting a Mimo22CC Object in a Diagram under the Application on page 145.

As an example, follow the steps below to connect a Mimo22CC object to a new application:

1. Create the new application in Control Builder.
2. Connect the application to the controller.
3. Connect the application to a task.
4. Connect BasicLib, ControlStandardLib and ControlObjectLib to the application.
5. Instantiate the Mimo22CC object (in the ControlObjectLib).
6. Instantiate two AnalogInCCs and two AnalogOutCCs (in the ControlStandardLib).
7. Create one variable for each AnalogInCC or AnalogOutCC of type "RealIO" in the application, and connect these to the AnalogInCCs and AnalogOutCCs.
8. Connect the control modules correctly. Figure 66 shows the completed connection for the Mimo22CC object.

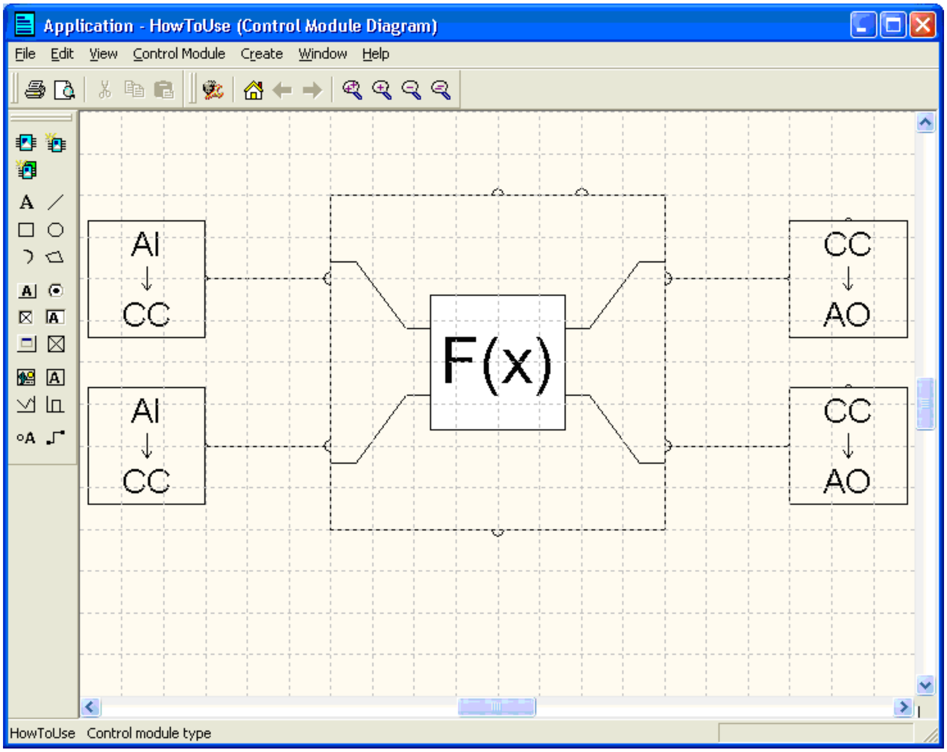

Application - HowToUse (Control Module Diagram)
File Edit View Control Module Create Window Help
AI ↓ CC
AI ↓ CC
F(x)
CC ↓ AO
CC ↓ AO
HowToUse Control module type

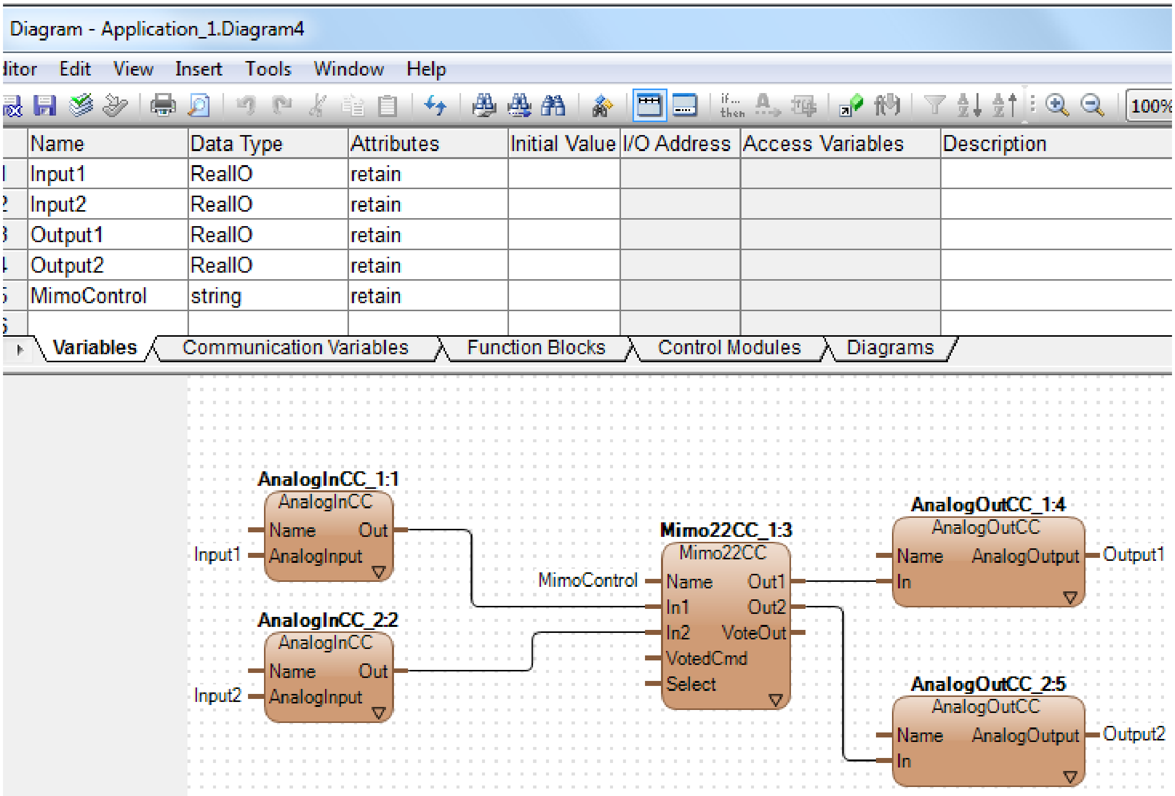
Figure 66. Connection of the Mimo22CC object

Connecting a Mimo22CC Object in a Diagram under the Application

As an example, follow the steps below to connect a Mimo22CC object in a new diagram under the application:

1. Create a new diagram under the application.
2. Connect the application to the controller.
3. Connect the diagram to a task.
4. Connect BasicLib, ControlStandardLib and ControlObjectLib to the application.

5. Right click the new diagram, and select Editor.
6. In the graphical code block of diagram editor, instantiate the Mimo22CC object (from the ControlObjectLib).
7. Instantiate two AnalogInCCs and two AnalogOutCCs (from the ControlStandardLib).
8. Create graphical connections (drag-and-drop) from the output ports of the two AnalogInCCs to the two input ports of Mimo22CC.
9. Create graphical connections (drag-and-drop) from the input ports of the two AnalogOutCCs to the two output ports of Mimo22CC.
10. Create one variable for each AnalogInCC or AnalogOutCC of type "RealIO" in the diagram, and connect these to the AnalogInCCs and AnalogOutCCs.
11. Connect the control modules correctly. Figure 67 shows the completed connection for the Mimo22CC object in the diagram editor.



	Name	Data Type	Attributes	Initial Value	I/O Address	Access Variables	Description
1	Input1	RealIO	retain				
2	Input2	RealIO	retain				
3	Output1	RealIO	retain				
4	Output2	RealIO	retain				
5	MimoControl	string	retain				
6							

Figure 67. Mimo22CC connected in diagram editor

Creating a New MimoXXCC Object from Another MimoXXCC object

Follow the guidelines below to create a new MimoXXCC object from another MimoXXCC object:

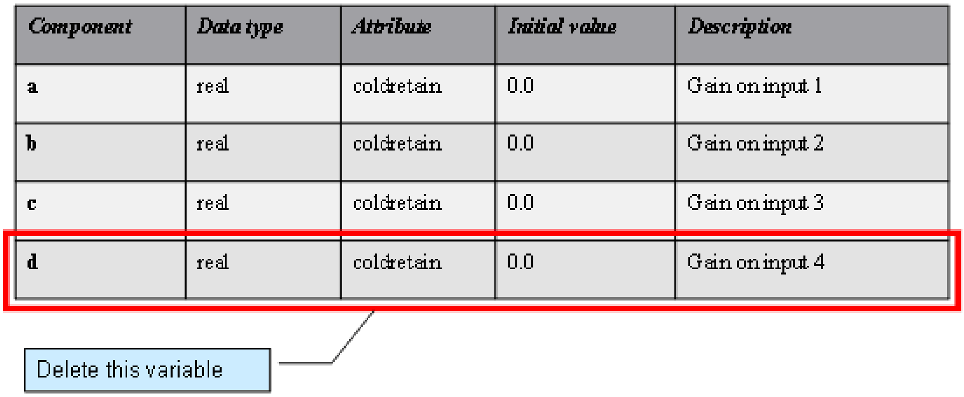
- Create appropriate variables, parameters, and data types to pack the data to different channels to gain an easy overview of the control module structure.
- Create function blocks for CC-component calculation in both forward code block and backward code block. Create additional function blocks in both forward and backward direction when dealing with voted functionality.

  The new code must be easy to understand and well structured, with the required inputs and outputs.

Creating a new Mimo33CC object from a Mimo44CC object

Follow the steps below to create a new Mimo33CC object from a Mimo44CC object:

1. Create datatypes
   a. Create the Coeff3 datatype by deleting a variable from Coeff4.

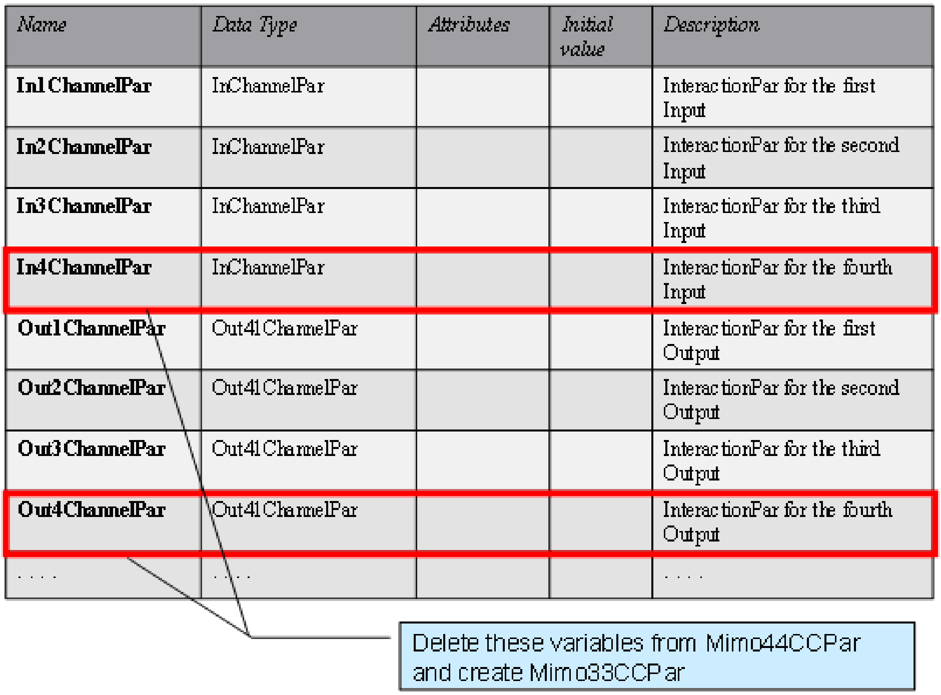


Component	Data type	Attribute	Initial value	Description
a	real	coldretain	0.0	Gain on input 1
b	real	coldretain	0.0	Gain on input 2
c	real	coldretain	0.0	Gain on input 3
d	real	coldretain	0.0	Gain on input 4

Delete this variable

Figure 68. Coeff4 before it is changed to become Coeff3

   b. Create the Mimo33CCPar data type by deleting variables from Mimo44CCPar.

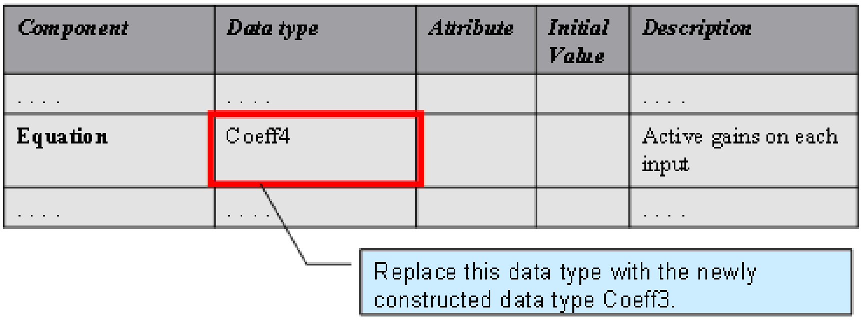


Name	*Data Type*	*Attributes*	*Initial value*	*Description*
In1ChannelPar	InChannelPar			InteractionPar for the first Input
In2ChannelPar	InChannelPar			InteractionPar for the second Input
In3ChannelPar	InChannelPar			InteractionPar for the third Input
In4ChannelPar	InChannelPar			InteractionPar for the fourth Input
Out1ChannelPar	Out41ChannelPar			InteractionPar for the first Output
Out2ChannelPar	Out41ChannelPar			InteractionPar for the second Output
Out3ChannelPar	Out41ChannelPar			InteractionPar for the third Output
Out4ChannelPar	Out41ChannelPar			InteractionPar for the fourth Output
. . . .	. . . .			. . . .

Delete these variables from Mimo44CCPar and create Mimo33CCPar

Figure 69. Creating Mimo33CCPar from Mimo44CCPar

c. Create the Out31Channel data type by changing the Out41Channel data type.

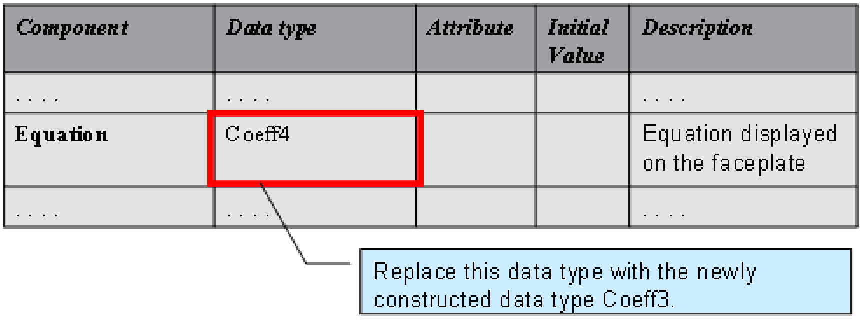


Component	Data type	Attribute	Initial Value	Description
. . . .	. . . .			. . . .
Equation	Coeff4			Active gains on each input
. . . .	. . . .			. . . .

Replace this data type with the newly constructed data type Coeff3.

Figure 70. Changing the Out41Channel to create Out31Channel

d. Create the Out31ChannelPar by changing the Out41ChannelPar.



Component	Data type	Attribute	Initial Value	Description
. . . .	. . . .			. . . .
Equation	Coeff4			Equation displayed on the faceplate
. . . .	. . . .			. . . .

Replace this data type with the newly constructed data type Coeff3.

Figure 71. Changing Out41ChannelPar to Out31ChannelPar

2. Create the following necessary parameters:
   - In1
   - In2
   - In3
   - Out1
   - Out2
   - Out3

   Many other parameters, which are common for all MimoXXCC objects, must be already present.

3. Create the following variables:
   - In1Channel
   - In2Channel
   - In3Channel
   - Out1Channel
   - Out2Channel
   - Out3Channel

   Many other variables, which are common for all MimoXXCC objects, must be already present.

4. Modify the following function blocks to reflect the changes from Mimo44CC object to Mimo33CC object:
   - AssignBTInputsX
   - OutX1BackwardFunction
   - OutX1Function

   For example, change the Out41Function to Out31Function by following the same principles for CC component calculation, but considering only three inputs instead of four inputs.

5. Modify the code to suit the 33CC object, by deleting or replacing the lines in the existing code of 44CC object:

   a. Modify the forward code.

   b. Modify the backward code.

   c. Modify the Set_Outputs code block.

Advanced Functions

This section describes a number of functions that are built into the types in the Control libraries. It also describes of a number of functions and library types from other standard libraries that can be used when building control loops. The description is split on the following functional areas:

- Anti-Integrator Wind-Up Function on page 153 describes the anti-integrator windup function that is built into the control types.
- Bumpless Transfer on page 154 describes the bumpless transfer function, which is used to smoothen controller output.
- Deviation Alarms on page 159 describes the alarm and event functions that are built into standard and advanced controller types.
- Feedforward on page 161 describes the feed-forward function, which is used to accelerate controller response by adding to or subtracting from controller output.
- Autotuning on page 162 describes how to use autotuning functions to improve controller settings.
- Adaptive Control on page 169 describes how to achieve adaptive control, for complex processes.
- Gain Scheduling on page 173 describes how to use gain scheduling to adapt settings to predictable variations in your process.
- Gain Scheduling versus Adaptation on page 175 discusses when to use gain scheduling, and when to use adaptation.
- Additional Control Functions on page 177 collects information on a number of special functions that are offered by the Control library types, such as three-

position output, stiction compensation, oscillation detection, reduction of friction influence, and detection of sluggish control.

- Input and Output Signal Handling on page 184 describes objects used for input and output signal handling.
- Supervision on page 195 describes objects used for supervision, that is, level detectors, supervision objects, and signal objects.
- Calculation on page 198 describes objects used for calculations of medians, mean, and majority, as well as other mathematical calculations. The Compact Control Builder in itself also contains a number of basic mathematical calculations, such as trigonometry, logarithms, exponentials, etc.
- Signal Handling on page 200 describes objects used to detect changes in signals, in order to be able to predict control actions, such as derivative objects, integrating objects, flow calculators.
- Branch Objects on page 209 describes objects used to split signals into several components.
- Selector Objects on page 213 describes objects used to select one out of several signals.
- Limiter Objects on page 219 describes objects used to limit signals.
- Conversion on page 222 describes objects used to convert signals from one data type to another.
- Miscellaneous Objects on page 225 describes some additional functions that might be useful in control loops, for example, an object that can be used to break up control loops.

Anti-Integrator Wind-Up Function

The anti-integrator wind-up function is an internal function in the controller modules that stops the integral part in certain situations. It is used, for example, in a cascade (master/slave) configuration, when the slave is in Manual mode, to prevent the master from integrating.

Problems with integrator wind-up may occur when a controller containing an integrator is not able to bring the control deviation (Sp – Pv) to zero fast enough, compared with the integral action of the controller. The controller output would probably reach one of its limits and remain there for a while, even after the control deviation has changed sign once ,after the process value has passed the setpoint. The result would be a large overshoot and therefore a slow response.

The reason for this unfavorable behavior is that the integrator winds up to a large (positive or negative) value when the control deviation has the same sign for a long time and the controller output reaches its limit.

When the control deviation changes its sign, it may take a long time for the integrator to wind down enough for the controller output to leave its limit.

To prevent this, integrator wind-up is limited by the anti-integrator wind-up function. A small wind-up is allowed to avoid the risk of small oscillations of the controller output, close to its limit. The size of the allowed integrator wind-up is determined by the size of the control deviation and the integration time of the controller. This is to initially achieve a fast response from a maximum (or minimum) value of the controller output.

When the anti-integrator windup is active, this is indicated in the interaction window by means of an icon that also shows the direction of the windup.

External Reset Feedback for Handling Anti-Integrator Wind-up

In PidCC and PidAdvancedCC control modules, it is also possible to use an external value, ERF (External Reset Feedback), instead of using the limiting value for anti-integrator wind-up,

If ERF option is used, ensure that the gain of the feedback path is the inverse of the signal path. The feedback path indicates backtracking because it is not an output path.

Bumpless Transfer

Bumpless transfer means that the controller output is made as smooth as possible, even when conditions within the controller change abruptly. Examples of such changes are mode changes and parameter value changes.

Bumpless transfer may be achieved in different ways. First of all, the integrator, if one exists, of the controller is adjusted so that the output becomes as smooth as possible. For controllers without integrator, the same effect is achieved (if offset adjustment is enabled) by adjusting the offset.

If the controller has no integrator, but any of the control modules preceding the controller has an integrator, this integrator is adjusted instead. If none of these options are available, the output may be temporarily ramped to achieve smoothness. In some cases, discontinuities in the output may be accepted.

The maximum increase and decrease ramping speed must be adapted to the process. If they are too slow, it might take a very long time before the ramp terminates. If they are too fast, the control actuator may be damaged.

Bumpless Transfer during Mode Changes

The result of changes from one mode to another is described in Table 11, where Auto has the lowest priority, and Tuning the highest. The numbers refer to the outcome described in the list below the table. Impossible changes are indicated with an X.

There is, however, one exception. It is not possible to go to Tuning mode when Backtracking is requested (that is, the object would have been in Backtracking mode, were it not in Tracking or Manual mode).

Table 11. Bumpless transfer during mode changes

From/To	Auto	Backtracking	Tracking	Manual	Tuning
Auto	–	4	1	2	5
Backtracking	3	–	1	2	X
Tracking	3	4	–	2	5
Manual	3	4	1	–	5
Tuning	6	4	6	6	–

1. The output is ramped, at the rate of change set by the parameters *OutIncLim* and *OutDecLim*, until the output tracking value (*TrackValue*) is reached.
2. The manual value attains the value of the output upon the change to Manual mode.
3. The mode change can behave in any of the following ways:
   a. If the controller has an integrator (PI, PID, PPI) and the *Pstart* parameter is disabled, or if offset adjustment is enabled for P and PD controllers, the controller starts controlling from the value of the output, before the mode change. In this case, Pv Tracking may occur. See Process Value Tracking on page 157.

   b. If the controller has an integrator (PI, PID, PPI) and the *Pstart* parameter is enabled, then the case is the same as above, but with the *Pstart* function added. At the instant of the mode change, *Pstart* internally adds a step (= *G(Sp – Pv)*), to which the output is then ramped.

   c. If the controller has no integrator (P, PD), and offset adjustment is disabled, the following will occur:

      If external setpoint is used and the connected control module can backtrack, or if internal setpoint is used and internal setpoint backtracking is enabled, then the setpoint will be adjusted so that the output becomes continuous.

      Otherwise the output may be discontinuous.
4. The output of the controller becomes equal to the backtracking value.
5. Tuning starts from the current value of the output.
6. The output returns to the value before tuning started.

Process Value Tracking

Tracking of the process value, Pv Tracking, is an internal function in the controller that copies the process value Pv to the value sent back to an external setpoint Sp, or if enabled to the internal setpoint. Pv Tracking may occur for controllers with integrator or with offset adjustment enabled. It occurs when the controller is in Backtracking, Tracking or Manual mode.

Output Change Rate Parameters

Two controller InteractionPar components, *OutIncLim* and *OutDecLim*, determine the output change rate of the ramp used during the mode changes described above. They are also used for the ramp, which prevents the output from changing abruptly, when output limits are narrowed.

These parameters do not limit the velocity change rate of the output in general, the change rate is only affected temporarily, in the cases described above.

In the PidCC and PidAdvancedCC controllers, the change rate in manual mode may be limited by these two interaction parameters. This setting is found in 'Enable out ramp man(ual)'

Bumpless Transfer during Parameter Changes

If the value of a controller parameter changes, for example the gain, the output will be continuous if the controller has an integrator, or if offset adjustment is enabled. Otherwise the output may be discontinuous.

Bumpless Transfer during Internal and External Setpoint Changes

Bumpless transfer between internal and external setpoints is achieved in the following ways. See first Reduced Effect of Setpoint Changes on page 93 and then Internal Setpoint Ramping on page 94.

1. Upon transfer to internal setpoint:

   The internal setpoint value is initially set equal to the current value of the setpoint.

2. Upon transfer to external setpoint:

   If the setpoint is connected to a preceding control module with an internal state, it will be continuous. The internal state is adjusted so that the setpoint becomes continuous.

Otherwise the setpoint is in general not continuous.

Bumpless Transfer when Enabling or Disabling the Limitation of the Output

Bumpless transfer is obtained when the limitation of the controller output is enabled or disabled in the following way. If you narrow the limits, and the controller output is outside the new limits, the output follows a ramp until it reaches the new limit, using the set change rate. See the section Output Change Rate Parameters on page 157. When you expand the limits, controller output is continuous for a controller with an integrator. It may be discontinuous for a controller without an integrator.

Bumpless Transfer when Forcing the I/O Signal to the Process

When the output I/O enters Forced mode, it will request the controller to go into Backtracking mode. The reaction of the controller depends on the priority of the modes for the controller, as described above.

When the output I/O leaves Forced mode, it will no longer require the controller to be in Backtracking mode.

If two PidLoop function blocks are used to build a cascade loop, the bumpless transfer function does not work properly. Use the PidCascadeLoop function block instead.

Bumpless Transfer at Switchover to Redundant I/O

To get bumpless transfer of I/O signals of *RealIO* data type, at switchover from active to redundant I/O, a RedundantIn function block can be used. It is used in the standard control modules and function blocks using the *RealIO* data type as an input parameter. To achieve the bumpless transfer, RedundantIn ramps the *RealIO* signal by using a *real* input value for the change rate of the signal.

All controller types that have an in signal of the type RealIO have a built-in function block of this type. The only exception is MotorBi, MotorUni, MotorBiM, and MotorUniM, where the RealI/O signal is used for surveillance only.

Deviation Alarms

Deviation alarms are generated by the standard and the advanced controller objects, but not by the simple ones. The control deviation is defined as the difference between the process value and the setpoint value.

An alarm condition state and a Boolean alarm condition parameter are set when the deviation is higher or lower than the positive or negative limits set. To prevent alarm flicker, a suitable time filter and degree of hysteresis are used. Before going to Auto mode, you can set a certain start delay time, to give the controller time to tune before alarms are activated.

For information on the use of the inhibit and disable parameters for the alarm functions, see alarm and event information in the *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* manual.

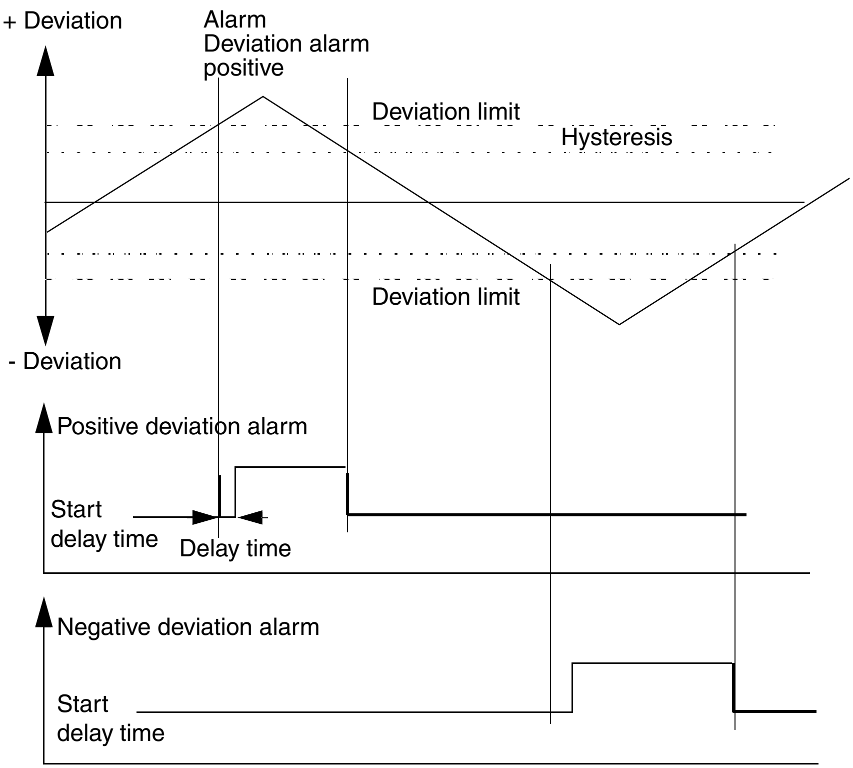


Figure 72. Overview of the controller deviation alarm limits

Feedforward

The feedforward signal is used to compensate for measurable disturbances, to achieve faster and smoother control of a process. Feedforward means that a signal is either added to or subtracted from the output signal of the controller. The feedforward signal may also be amplified or reduced.

The feedforward process accelerates the controller response by anticipating changes and acting to neutralize any disturbance, before it occurs.

Feedforward can also be used to suppress changes in the input signal that must not be allowed to influence the controller output.

Feedforward is selected as a positive (+) or a negative (–) value of FF_{Gain} in the algorithm.

$$Out_{PID} = Out_{from\ PID\ algorithm} + FF_{Gain} * FF$$

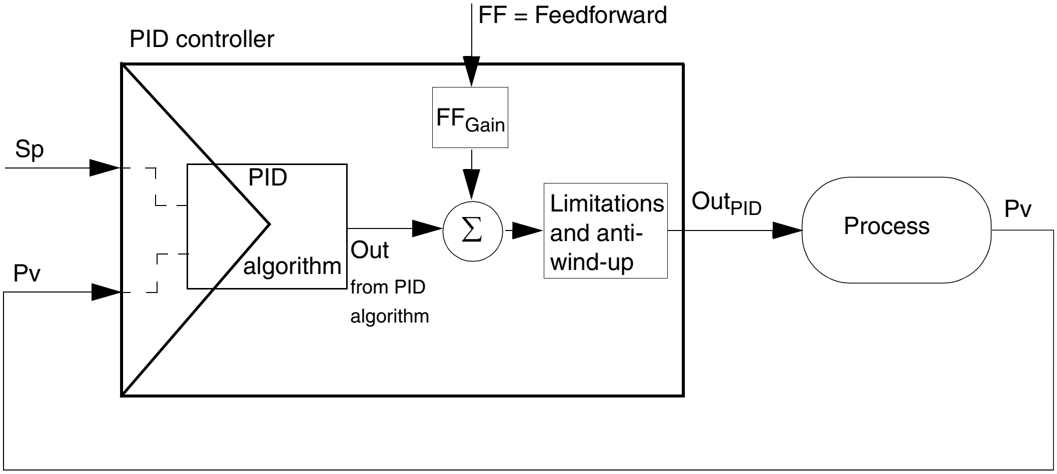


Figure 73. The feedforward principle in the controller

Autotuning

Introduction

Autotuning is a simple way to obtain suitable controller parameters. It is recommended to use the Autotuner function, otherwise, a great deal of time can be spent in manual tuning of many controllers in large process plants. Manual tuning time can be increased even more when retuning becomes necessary, due to changes in the process conditions.

Several autotuning iterations do not improve the information from one tuning to the next iteration. However, it does increase the speed for next autotuning iteration.

You are advised to repeat autotuning a couple of times to rule out possible disturbances that might have affected the first autotuning iteration. Furthermore, if a number of controllers affect the same process, it is necessary that all controllers have been correctly autotuned and holds accepted process values while autotuning a single controller.

When the process is in steady state, start the Autotuner. It then identifies the dynamic parameters of the process automatically, and from these, the Autotuner calculates and suggests appropriate PID parameters. When autotuning is complete, the controller reverts to previous mode. It uses the old controller parameters, but suggests the new autotuned parameters, and you have the choice to apply them.

The user may also select another controller structure and design than used for calculating the controller parameters from the autotuning results. Some users want a specific controller structure, for example, a PI controller, and that the result of an autotuning should comply with this selection. Then, the autotuner recalculates the controller parameters based on the autotuning results. This means that any new tuning is not necessary as the already executed tuning has measured the dynamics of the process. The autotuner uses these measured values while re-calculating the changed controller algorithm.

Autotuning is based on a relay (ON/OFF) identification method, with feedback measurements, as illustrated in Figure 74. To obtain extended autotuning, it is also possible to complete process identification by means of an automatic subsequent setpoint step. Choose between the following three autotuning methods.

1. Relay only. This normally gives acceptable controller parameters, particularly if the time needed for autotuning is critical.
2. Setpoint step only. After you have performed autotuning with the relay method, you may, at a later time, perform setpoint step identification, when you want to compensate for dead time in the process.
3. Relay and setpoint step. This is the complete autotuning alternative.

Perform autotuning when the process is in steady state only.

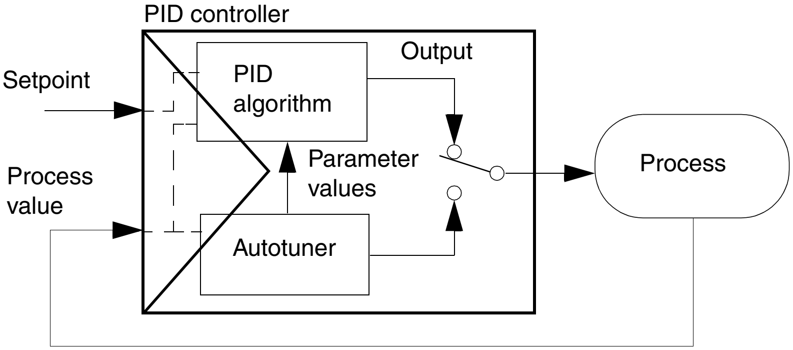


Figure 74. The principle of autotuning in a PID controller with the Autotuner function

Autotuning with Relay Method

When the system is in steady state, and the Autotuner has been started, the PID controller is temporarily disconnected.

First, the Autotuner measures the noise of the process value.

Secondly, the output is generated and changed by the relay, with a hysteresis function, to implement a disturbance in the process, of a small amplitude, according to the figure below. The effect of the relay function is an ON/OFF control which, by means of a square wave signal, generates a controlled and stable oscillation in the process value. The response is observed, and the amplitude of the oscillation is automatically controlled to a minimum value by adjustment of the relay amplitude.

From the period and amplitude of the process value oscillation, suitable P, I and D parameters are calculated. The controller is then ready to operate and the PID algorithm is reintroduced into the control loop.

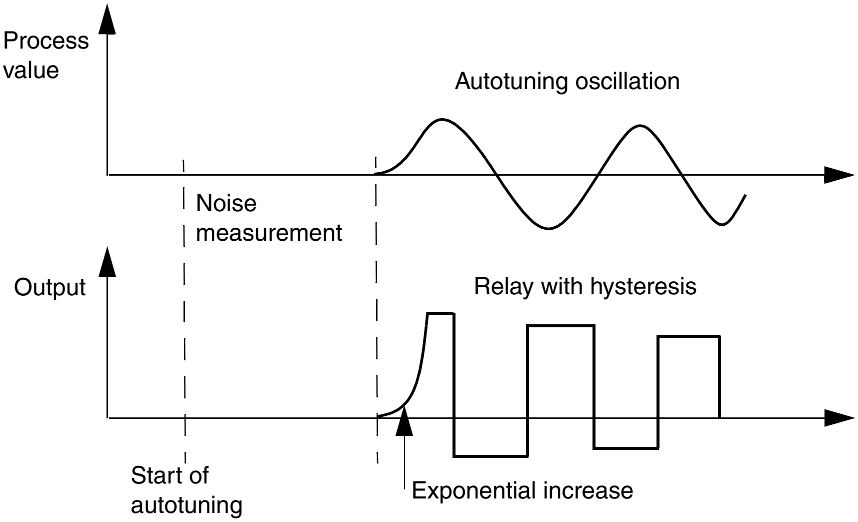


Figure 75. The process value oscillation

Extended Autotuning with the Setpoint Step Method

To improve the autotuning, a small setpoint step can be carried out automatically, or at your request, with the relay autotuned PID controller. Static gain, dead time and the time constant of the process are obtained from the setpoint step response, and the PID parameters can be adjusted.

The step tuning method is only available in the PidAdvancedCC controller as stated in Table 4.

Autotuning Process

Autotuning can be started with the controller in Manual or Auto mode. During the autotuning process, the Autotuner controls the output. The following three conditions must be checked before starting autotuning:

- The process must be in steady state. It is not possible to start the Autotuner during a load disturbance or a setpoint change.
- It is also important that no major load disturbance occurs during the autotuning process.
- The control deviation (Sp – Pv) must be less than 5% of the actual Pv range.

The value of control deviation or error (Sp -Pv) with respect to the dead zone value is also available as an output in PidCC and PidAdvancedCC control modules.

When these conditions are fulfilled, you can start the Autotuner. If the process is not in a steady state, autotuning may fail. Autotuning is interrupted by a load disturbance.

In PidCC and PidAdvancedCC control modules, there is also an output parameter that indicates whether the autotuner is active or not.

During the first part of the autotuning process, the output signal is kept constant and the noise level is measured, in order to calculate the necessary oscillation amplitude. Note that it is important to choose a shorter sampling time (task cycle time) for fast processes than for slow processes, otherwise, the period used for noise calculation will be unnecessarily long, autotuning will be less accurate, and the resulting control will be unnecessarily slow. If the process is not stationary, the Autotuner will interrupt and give a warning that the noise level is higher than the true level.

When the noise level has been calculated, the Autotuner determines the relay hysteresis, no larger than necessary, but sufficiently above the noise level. Subsequently, the output from the relay is introduced into the loop, but no larger than the maximum relay value set. This causes the process value to oscillate around the setpoint, and the relay output amplitude is adjusted to give the desired amplitude of the process value. It may be necessary to limit the amplitude of the first output signal increase, for example, in processes with significant dead times.

The period and amplitude of the oscillation are determined for the process value. Slow processes can have oscillation periods between minutes and hours, while fast processes have oscillation periods of a few seconds. When the oscillation amplitude is stable, the PID parameters are calculated. If the autotuning method selected is relay only, autotuning is complete at this point. The new parameter values may be applied. If tuning fails, the controller continues to use the old parameters.

After the relay method has been used, you may select setpoint step identification only, or relay and setpoint step identification. After the user has started a setpoint step, the process value will finally reach the new setpoint according to the figure below. When steady state is reached, the output signal is restored to its previous value. The process goes back to its initial state and autotuning is complete. The process gain, time constant and dead-time are calculated from the setpoint step response. With these process parameters identified, the Autotuner recalculates the PID parameters obtained from the relay method. When autotuning is complete, the new parameters are shown in the interaction window. If you want to accept the suggested PID parameters, apply them before closing the interaction window.

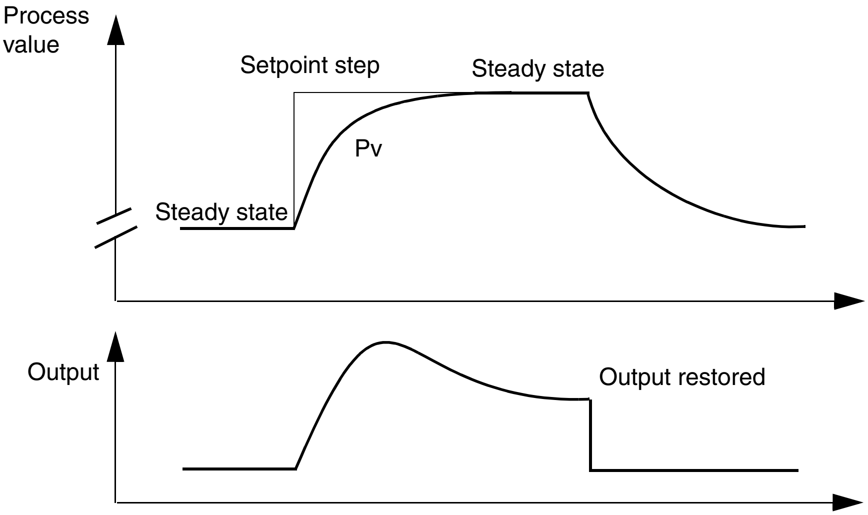


Figure 76. Setpoint step identification and output restoration

The Autotuner saves the values of the noise level and the relay amplitude from the previously performed autotuning. Autotuning may then be repeated more quickly. To start from the beginning, reset the Autotuner. Autotuning using the relay and/or step method can be made individually in each part of a controller with gain scheduling. This applies only for the PidAdvancedCC controller type.

PI or PID Controller

During relay tuning, the Autotuner chooses a controller type, PID or PI, automatically. The normal Autotuner choice is a PID controller. In some cases, where processes contain integrators, for example, for level control, the Autotuner may decide to use a PI controller.

PPI Controller

If Setpoint step only, or a complete relay and Setpoint step autotuning is performed, the Autotuner compares the process dead time with the process time constant. If the dead time dominates (about twice the time constant) the Autotuner may suggest the PPI design. A PPI controller is never chosen if autotuning is configured for relay only. The Autotuner detects the process dead-time during the setpoint step method only. However, you may manually select the PPI type to handle processes with a known dead-time, which then has to be specified.

Controller Response Speed

The choice of controller speed influences the behavior of the control loop. In certain processes, high speed is necessary and overshoots are acceptable, whereas in other cases, a slower control sequence can be accepted. In the Autotuner, it is possible to select one of three controller response speeds: Slow, Normal, or Fast, and thus determine the method of operation. Upon speed changes, the controller PID parameters are updated immediately. Apply the new parameters to accept them.

Pre-settings

For successful autotuning, some pre-settings can be made as follows.

- The maximum limit of the relay amplitude, expressed in engineering units, is initialized to 10% of the output range. The Autotuner automatically chooses a suitable relay amplitude, so that the parameter for maximum relay amplitude needs to be used only if too high output signal levels cause critical situations.
- The maximum limit of the setpoint step, expressed in engineering units, is initialized to 10% of the process value range. The Autotuner automatically chooses a suitable step amplitude, so that the parameter for maximum step amplitude needs to be used only if too high setpoint values cause critical situations.
- Warning time is selected, if you want a warning for excessive autotuning time.

Resetting

If you set the InteractionPar component *Reset*, the values of the noise level and the relay amplitude saved by the Autotuner from the previous autotuning are rejected. A new estimate of the noise level is then made. Reset is recommended when a condition of the process, such as dynamics or noise properties, has changed. It should also be used when earlier autotuning has failed.

Direct or Reverse Direction

The direction of the process gain is either direct or reverse. The default direction is reverse. This means that when the process value increases, the controller output decreases. If the direction you have set is not the same as that automatically detected by the Autotuner, a warning text is displayed, indicating that the controller direction may be wrong.

However, in cases where the process is of extreme “non-minimum phase” type, and the process starts to respond in the wrong direction to an output step, the Autotuner will also give a warning.

Maximum Sampling Time

When you have autotuned a controller, the Autotuner calculates a maximum sampling time and indicates it in the interaction window. This time is 1/8 of the process oscillation time. If the current sampling time (task cycle time) is longer than the calculated maximum sampling time, then you should decrease the current sampling time.

If your sampling time is too long, the suggested maximum sampling time may be shorter than the current sampling time. This means that the current sampling time is too long in relation to the signal changes the Autotuner has detected.

A suitable strategy for decreasing the sampling time is to halve the current sampling time, and autotune again, to see the new maximum sampling time given by the Autotuner.

This method can be repeated until you reach the point where the current sampling time is shorter than or equal to the maximum sampling time.

Adaptive Control

There are many kinds of processes. Some are very simple to control, and some are far more complex, with changing dynamics. An example of a complex system is maintaining a constant value of the pH in a tank. A combination of an adaptive controller and gain scheduling gives good results in such applications.

An adaptive controller is used to continuously update controller parameters. The variations in process dynamics must, however, be slow in comparison with the time constant of the process. An adaptive controller adapts the PID and feedforward gain parameters.

The adaptation function is enabled by the operator. The operator must first perform an initial start-up autotuning. When the tuned parameters have been accepted, adaptive supervision is started by continuously monitoring the input and output signals to/from the process. Adaptation is then activated only when both signal values exhibit large enough variations. The activated adaptation function then calculates and implements new controller parameters.

Enabled, ongoing adaptation is deactivated on the following occasions.

1. The operator disables adaptation.
2. The Autotuner is activated.
   When autotuning is complete, adaptation continues, either with the new initial tuning values, if they have been accepted, or the old ones, if no choice of tuning values was made by the operator.
3. Upon changes to Manual mode.
4. During backtracking.
5. Upon output tracking.
6. When the sampling time is too long.
7. For a feedback adaptive controller also:
   – in the case of load disturbance,
   – when there is no integrator (I) part,
   – when a PPI controller is chosen.

Feedback Adaptive Controller

Feedback adaptation modifies the PID parameters of the controller. The feedback adaptive controller has the ability to continuously follow a specified point on a Nyquist curve, as the process dynamics change. The principle of the feedback adaptive controller is shown in the following figure.

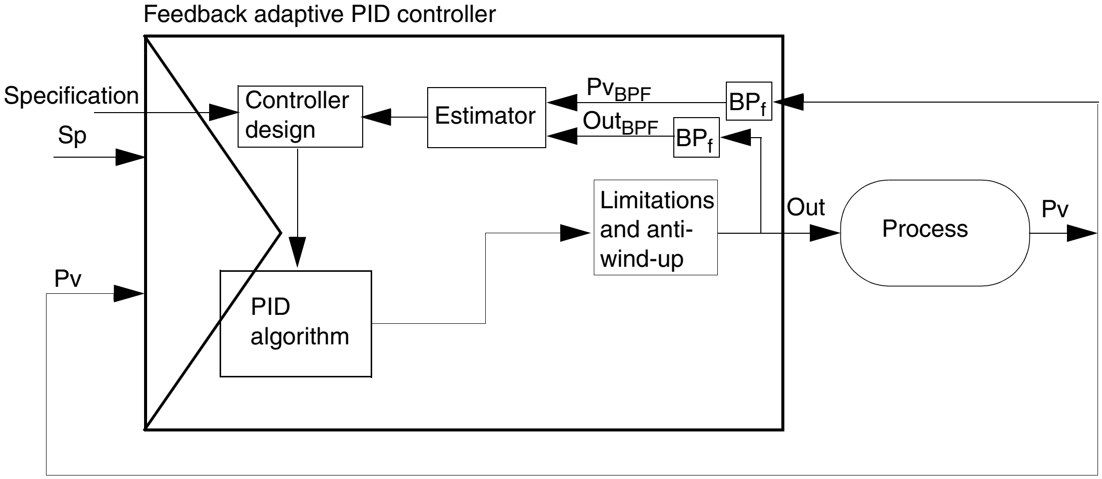


Figure 77. The principle of the feedback adaptive PID controller

After initial autotuning, feedback adaptive supervision is achieved by monitoring the band-pass-filtered PID controller Out_{BPF} signal and the process value Pv_{BPF}.

The user specifies Slow, Normal, or Fast response. The adaptive controller then gives the resulting PID or PI parameters.

Enabled, ongoing adaptation is deactivated when a load disturbance is detected in the Pv signal.

The reason for this is that the process value (Pv) is not relevant in relation to the Out signal from the PID controller, and would give incorrect values for PID parameters.

When the load disturbance has disappeared, adaptation supervision continues.

Feedforward Adaptive Controller

If it is possible to measure load disturbances in the process, you can use standard feedforward control. If the relation between the measured and the real load disturbance varies, you can use a feedforward adaptive controller as shown in the figure below, for example, when the flow characteristics of a pump are changed, due to fouling in the pipe system. Feedforward adaptation then modifies the feedforward gain, FF_{Gain}, of the controller.

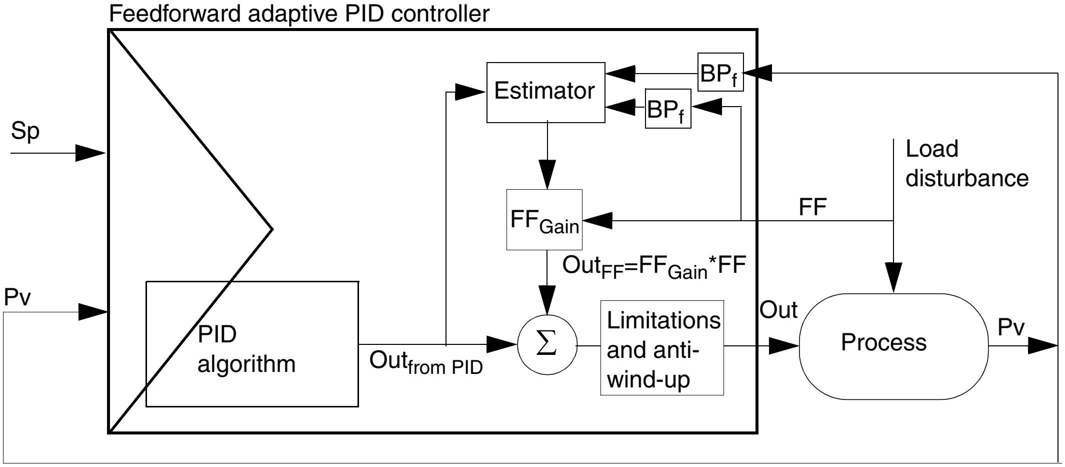


Figure 78. The principle of the feedforward adaptive PID controller

Feedforward adaptive supervision is carried out by monitoring the process value, Pv, and the feedforward signal, FF, representing the load disturbance. This is done through the high-band-filters, BP_f. The parameter estimator is also influenced by the PID controller $Out_{from\ PID}$ signal. Adaptation starts when both the filtered signals are large enough.

The feedforward gain, FF_{Gain}, which can be positive or negative, is continuously calculated as long as the feedforward adaptive function is active.

The signal, $Out_{FF} = FF_{Gain}*FF$, is added to the PID controller output signal $Out_{from\ PID}$ to compensate for the load disturbance.

Gain Scheduling

Gain scheduling can be used when the process has predictable non-linear dynamics, time variations, or demands on changes in operating conditions. To use the gain scheduling technique, you first have to choose a reference signal that correlates well with the changes in process dynamics. The reference signal can be:

- Pv – the process value signal
- Out – the output signal
- Sp – the setpoint signal
- Ext – an external signal
- Epsilon - control error (Sp - Pv)

The reference signal can be divided into up to five ranges, separated by adjustable limits. The gain scheduling function is a table, containing one set of all the parameters for the PID controller for each range. One set of parameters is active when the reference signal is within the current range. When the reference signal passes a value between two parameter set ranges, the next set of parameters takes over.

Parameter Set Ranges

As soon as the gain scheduling is selected in the interaction window, two parameter sets are available to start with. If more parameter sets are needed, insert a new one above the one selected. The limit value is given between the ranges as half the previous range. It is possible to change the limit manually. The selected parameter sets can also be deleted in the same way. The range then includes the deleted range.

Tuning the Parameter Sets

The Autotuner (see Autotuning on page 162) should be used to set controller parameters in each parameter range set. A parameter set is active when the reference signal is between its range limits. Autotuning can only be performed in an active parameter set range. When the reference signal is close to a limit, autotuning may give poor results. All tuning values, including adaptive controller values, are stored in the gain scheduling table. You may also set the controller parameters manually.

A small hysteresis function is built in, to avoid frequent switching between two parameter sets when a noisy reference signal passes a limit.

Example of Inserting and Tuning Parameter Sets

The following are the examples of inserting and tuning parameter sets.

1. Initially, we have a single parameter set (Set 1) that is Autotuned to T1. When gain scheduling is activated, a second set (Set 2) is added above Set 1, with the same Autotuned T1. You can then select Set 2 and Autotune this to T2. The limit is by default set to half the height of the set that is divided. You can change this before autotuning the new set.

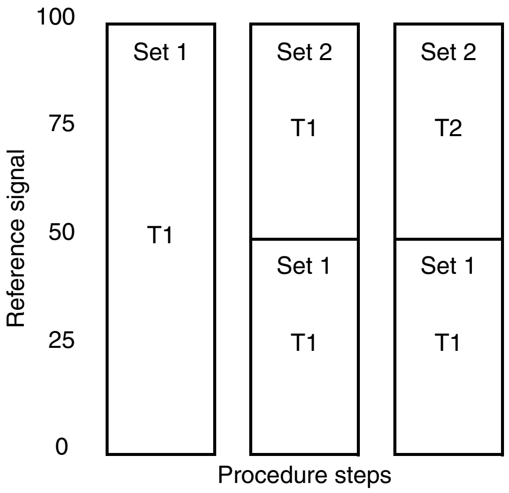


Figure 79. Example of the procedure for gain scheduling in two sets

2. You can then split Set 2 in half. Set 3 is added above Set 2, with the same autotuned T2. You can then select Set 3 and autotune this to T3. The limit is by default set to half the height of the set that was divided. You can change this before autotuning the new set.

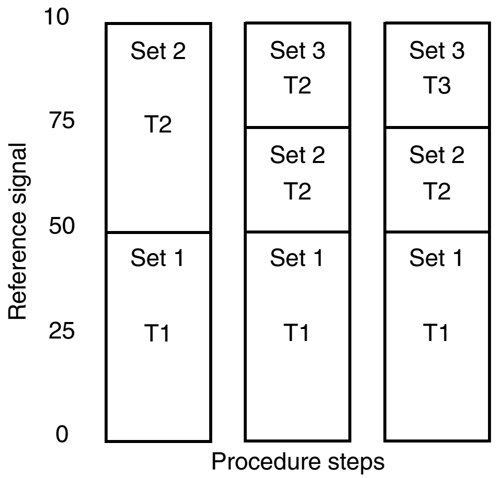


Figure 80. Example of the procedure for gain scheduling in three sets

Gain Scheduling versus Adaptation

When configuring a controller, you can choose between constant controller parameters, gain scheduling, adaptation, or a combination of those, depending on the process dynamics, as follows, and according to Figure 81.

Process with Constant Process Dynamics

For a process with constant process dynamics, which is the most common, a controller with constant parameters can be chosen. The correct strategy is then to select a PID controller or a PPI controller if the dead time is long.

The process engineer’s trimming tool for PID and PPI controllers is the Autotuner, which suggests settings for the parameters of the controller.

As a rule of thumb, a PPI controller is used when the dead time is longer than the dominant time constant in the process.

Processes with Changing but Predictable Process Dynamics

For a process with changing but predictable process dynamics, which requires different parameters in different parts of the working range, the proper strategy is to use a PID controller or a PPI controller with gain scheduling. See the section Gain Scheduling on page 173. Use the Autotuner to tune the parameters in each parameter set range.

Processes with Changing and Unpredictable Process Dynamics

For a process with changing but unpredictable process dynamics, which vary slowly, the proper strategy is to use an adaptive PID controller. See the section Adaptive Control on page 169. The Autotuner is used to tune the initial parameters. A PPI controller is able to run gain scheduling, but not adaptation.

Processes with Changing and Partly Predictable Process Dynamics

For a process with changing, unpredictable process dynamics, which vary slowly, and partly predictable process dynamics, the proper strategy is to use a combination of adaptation and gain scheduling.

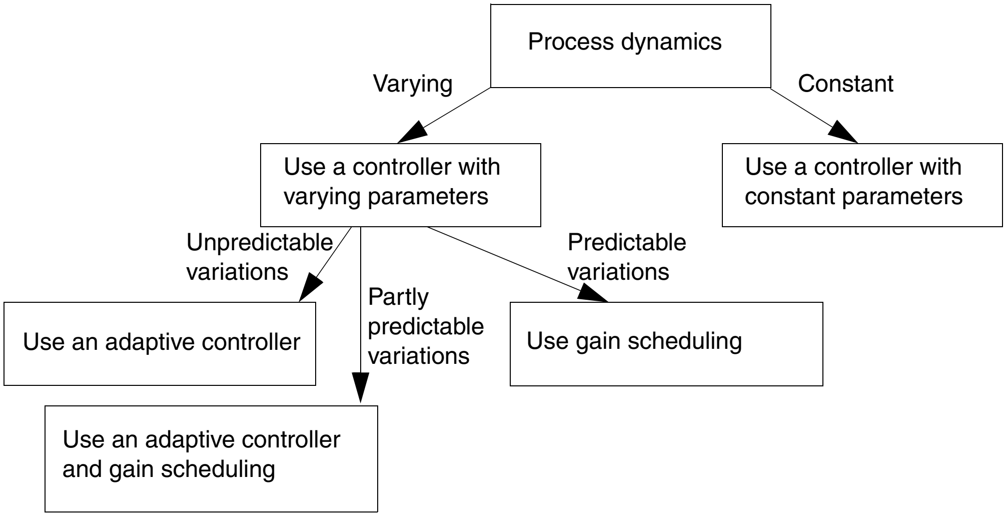

Process dynamics
Varying
Constant
Use a controller with varying parameters
Use a controller with constant parameters
Unpredictable variations
Partly predictable variations
Predictable variations
Use an adaptive controller
Use gain scheduling
Use an adaptive controller and gain scheduling

Figure 81. Procedure used to decide which controller to use, adaptive control and/or gain scheduling

Additional Control Functions

Three-Position Output

Three-position action from a controller with increasing, or decreasing, or no signal at all, for example, to an electrical motor actuator, is achieved by a function with two digital output signals, which are never active at the same time.

This three-position output control module is an extension of a controller, when two digital outputs are required. A comparison is made between the controller’s analog output signal and an analog signal from the control device or actuator, which gives the so-called *position feedback* signal.

When the difference is greater than a set dead zone, either of the two digital output signals, *Increase* (increment output) or *Decrease* (decrement output) of *BoolIO* type, is activated in the following manner, see Table 12.

Table 12. Three-position output.

Comparison	**Increase**	**Decrease**
Output=Position feedback	False	False
Output>Position Feedback	True	False
Output<Position Feedback	False	True

A position feedback signal is not always available. It can, however, be estimated internally by the module, to represent the current position of the control device by the following calculation. The time during which the increasing or decreasing pulse has been active is divided by the total action time between the actuator end positions which you can declare, and then multiplied by the controller output range. The minimum output signal pulse length that you can set is the sampling time.

The dead zone is the tolerated difference between the output signal from the controller and the position feedback signal. A difference within the dead zone will not affect any digital output.

The minimum time for switching between the two output signals can be set in seconds, as short as the sampling time, or longer, depending on the actuator.

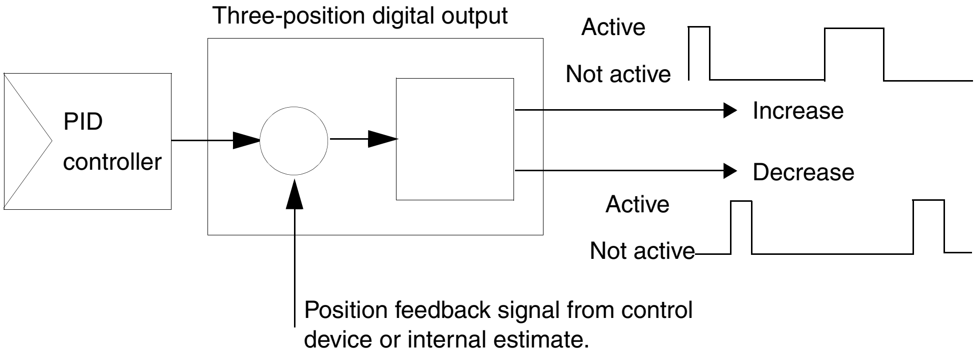

Three-position digital output
PID
controller
Active
Not active
Increase
Decrease
Active
Not active
Position feedback signal from control device or internal estimate.

Figure 82. The principle of the three-position digital output function

Stiction Compensator

It is always important to know how a control loop will perform, because it influences the process output. Performance checks may be carried out in many ways. One method of detecting deficiencies in the process control is to detect oscillations. Oscillations above a certain amplitude and within a certain frequency range are probably caused by sticking control valves, due to too high static friction, called stiction. This phenomenon usually increases gradually during operation with fluids that are difficult to handle, for example, viscous fluids. Stiction then gives rise to oscillations of a particular character in the process control loop.

There may also be other reasons for the oscillations, for example, badly tuned control loops or oscillating load disturbances. However, in this section, only methods of detecting and minimizing stiction problems will be dealt with. Methods implemented in the PID controllers and as an add-in control module, are described in the figure below and in the succeeding sections.

If the process handles products which cause friction problems in a pneumatic control valve, an add-in function, called a stiction compensator, should be added to the analog output signal used in the control loop.

This method of keeping pneumatic valves free from clogging and seizing involves activating them regularly by adding short pulses, to “knock” the valve. The stiction compensator function compensates for static friction and hysteresis which may increase gradually with time.

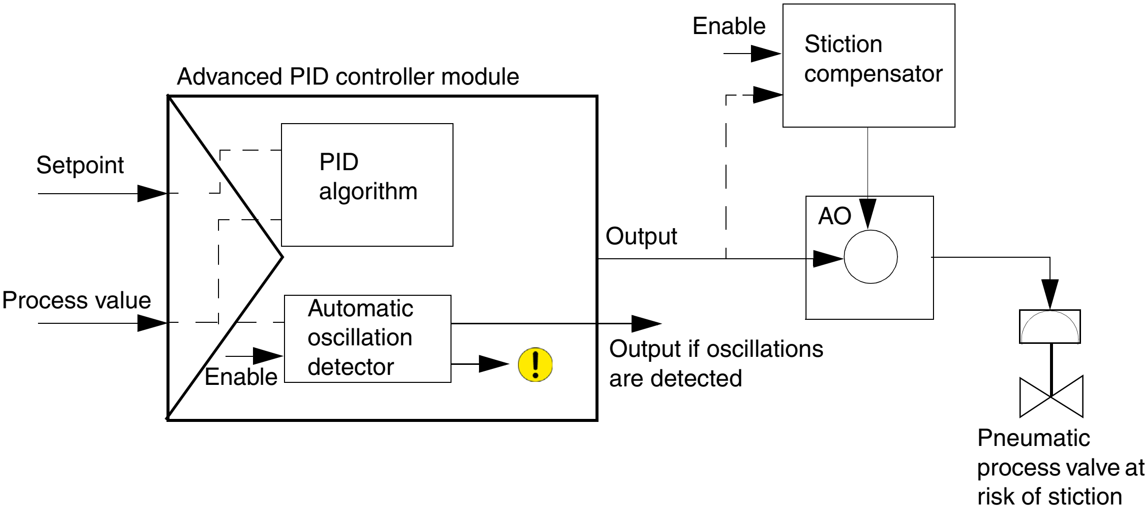

Enable
Stiction compensator
Advanced PID controller module
Setpoint
PID algorithm
AO
Output
Process value
Automatic oscillation detector
Enable
!
Output if oscillations are detected
Pneumatic process valve at risk of stiction

Figure 83. Principles for detecting and solving friction problems

Oscillation detection starts when the user has enabled the automatic oscillation detector in the PID controller module. If an oscillation is detected an output is set and a warning is given. If it is obvious that there is friction in the valve, the user should enable the stiction compensator function to keep the valve moving until it can be repaired or replaced.

The StictionCompensater object is designed to be added to the AnalogOutCC output control module.

Oscillation Detection

Automatic monitoring of control loop performance is built into the advanced PID controller module. When activated, this oscillation detector function detects oscillations in the process value around the setpoint, often caused by friction in a control valve. Oscillation is detected when the process value oscillates a certain number of times around the setpoint with an amplitude of about 1% or greater, and with a period of about the length of the process time constant.

If you are uncertain about the cause of oscillation, you may undertake a diagnostic procedure according to the flow chart in Figure 84, which helps you to find and eliminate the source of oscillation.

Oscillation detection can be sent from the PID AdvancedCC object (parameter *VoteOut*) to receiving Voting objects (parameter *Inx*). You then configure the value of *Inx* parameter to be for example oscillation detection from the Vote object's parameter *InxLevelConfig*. See also Signal and Vote Loop Concept on page 367.

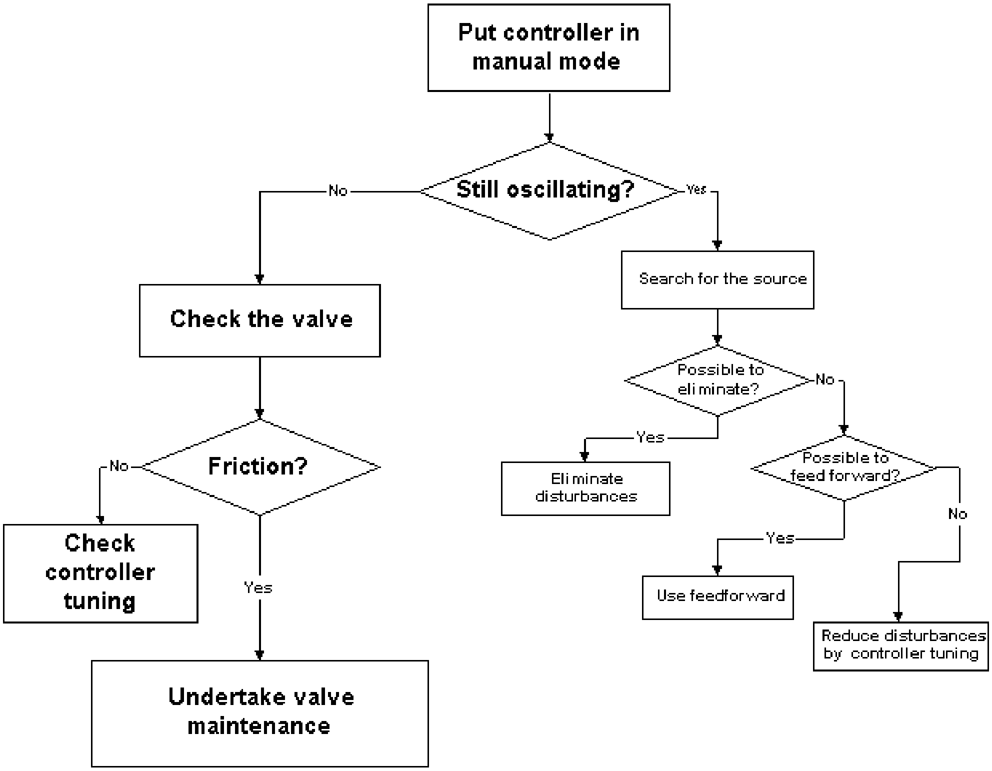


Figure 84. Flow chart for oscillation diagnosis

Perform the following steps to determine what kind of oscillation has been detected:

1. Activate the oscillation detector function in the loop assessment settings of the advanced PID control module, to detect any oscillation in the process value around the setpoint.
2. If an oscillation is detected, a warning text is shown in the More parameters interaction window and an output signal from the PID control module is set to true.
3. If you are uncertain of the reason for the oscillation you may undertake a stiction diagnostic procedure according to the succeeding steps. These guide you in finding and eliminating the oscillation.
4. Put the output signal to the pneumatic valve into forced mode.
5. If the oscillation stops, check the pneumatic valve for friction. If the valve is sticking, perform the required maintenance to retrieve the problem, or replace the valve.
6. If it is not suitable at the moment to carry out maintenance on the valve, wait for a later occasion. Meanwhile, you are advised to activate the stiction compensator to reduce the influence of static friction in a pneumatic valve.
7. If there is no friction, check the tuning of the controller. There may have been accidental changes in the process parameters.
8. If the oscillation persists, the process value may be influenced by a disturbance. Search for the source. It may be useful to use the feedforward function.

Reduction of the Influence of Friction

The stiction compensator function signal is superimposed on the analog output signal to the process, according to the figure below. A short pulse sequence is added to the controller output signal, inside the analog output control module. This signal is of equal amplitude and duration in the direction of the output signal's change rate. Thus, when the signal increases, the pulse is directed upwards (and vice versa). See Figure 85. In this way, it is possible to handle sticky valves. (Industrial tests show that the procedure reduces the control deviation during stick-slip motion significantly, compared with standard control without friction compensation.)

The stiction compensator function may be varied and the following parameters can be set: pulse amplitude, pulse width, and the pulse period factor multiplied by the pulse width, giving the pulse period time. A stiction compensator pulse is only given when the output signal changes by an amount greater than a set hysteresis limit.

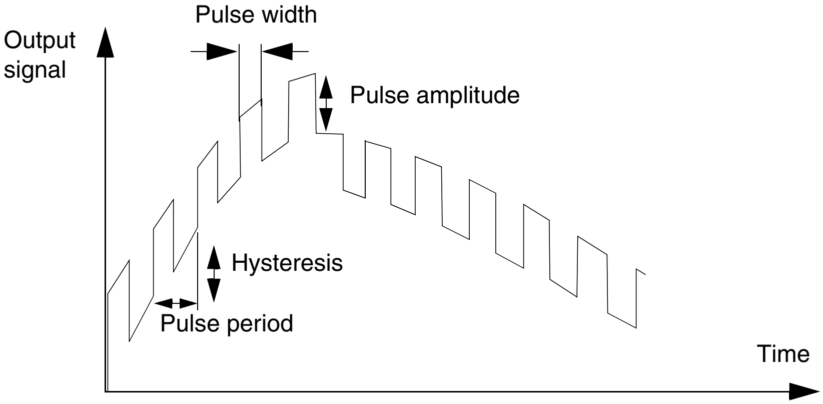


Figure 85. The stiction compensator signal is superimposed on the analog output signal

Sluggish Control Detection

Sluggish control, which should be avoided, means that a controller responds too slowly to load disturbances or setpoint changes, as in the figure below.

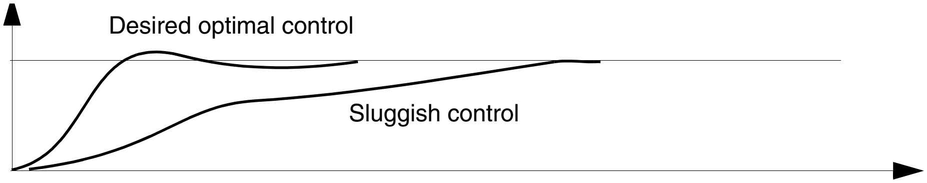


Figure 86. Illustration of sluggish control

Sluggish control loops may occur with conservatively or poorly tuned controllers. This may cause losses in production and quality. A sluggish response to load changes or disturbances is therefore undesirable. Slow behavior with unnecessarily large and long deviations from the setpoint should be avoided. A well-tuned controller gives a fast response to load disturbances. The loop assessment function, which works according to the Idle index, can detect sluggish control.When you have completed the commission of a control loop and you have tuned it, you can supervise the loop for the detection of sluggish control. Sluggish control may occur after a certain operating time.

Perform the following steps to detect sluggish control.

1. Activate the sluggish control detector function in the loop assessment settings of the advanced PID control module, to detect any sluggish control in the process.
2. If sluggish control is detected, a warning text is shown in the More parameters interaction window and an output signal from the PID control module is set to true.

Perform a new autotuning sequence and ensure that faster control is achieved.

Sluggish control can be sent from the PID AdvancedCC object (parameter *VoteOut*) to receiving Voting objects (parameter *Inx*). You then configure the value of *Inx* parameter to be for example sluggish control from the Vote object's parameter *InxLevelConfig*. See also Signal and Vote Loop Concept on page 367.

Input and Output Signal Handling

Signals start and end in I/O units with I/O channels of the *RealIO* data type. Between input and output I/O units, signals are handled in I/O function blocks of the *RealIO* data type, or directly in various function blocks, or in control modules of the *ControlConnection* data type.

In open loop control, information mainly goes forward, for example, formula calculations, indications, comparisons, or presentations.

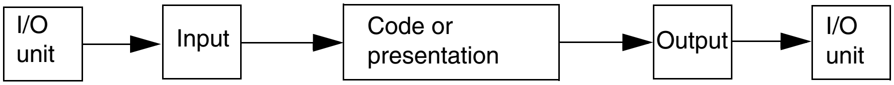


Figure 87. Signal handling in open loop control

In closed-loop control, applications that contain one or several controllers, it is necessary for information to go both forward and backward. This places much higher demands on the solution of such applications.

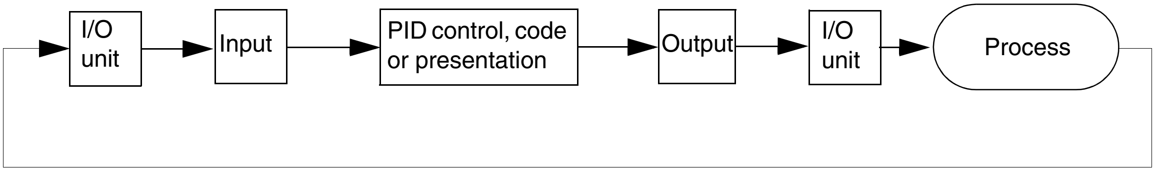


Figure 88. Control loop application in closed-loop control

When combining and connecting various objects, you should be able to predict the resulting functions and behavior.

It is often necessary to measure values for later calculations in the application and for presentation. Analog signals are then transferred from measurement transmitters in a process, to I/O units. The signal interface objects for input and output signals read values from, and write values to, the I/O systems, respectively. Further on in the loop, the signal is directed to application code, or to a presentation. The signal connected to an analog input interface which transforms it into a ControlConnection type signal for further direction, for example, to a PID controller, application code, or a presentation, see ControlConnection on page 75.

Over and under range measurement

Signal objects of real type and AnalogInCC in ControlStandardLib are equipped with an option to increase the signal range with a fixed pre-selected factor of +-15% of the specified range.

You can select individual Signal Objects connected to variables of data type RealIO on the controller and set the input parameter **EnableOverUnderRange** to true.

The default value on EnableOverUnderRange depends on a global project constant from BasicLib. The default value for this project constant is **false** and Over and Under range feature is disabled.

The Signal Object enabled with over and under range feature, displays the output parameter **OverUnderRangeEnabled** as true to inform the surrounding code about the extended range.

Input objects connected to I/O.

To enable signal range extensions on input signals, in Project Explorer, refer *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* .

Connected PID controllers need to be re-tuned for optimized operation.

The default value of project constant for inputs is set to false.

The objects of SignalLib and ControlStandardLib supporting signal range extension feature are:

- Functional blocks:
  - SignalInReal
  - SignalSimpleInReal
- Control modules
  - SignalInRealM
  - SignalSimpleInRealM
- AnalogInCC in ControlStandardLib

Output objects connected to I/O.

To enable signal range extensions on output signals, in Project Explorer, refer *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* .

The default value of project constant for outputs is set to false.

The extended range is also applicable in forced mode. The operator can set forced values directly to the IO-unit from the operator interaction windows.

The objects of SignalLib and ControlStandardLib supporting signal range extension feature are:

- Functional blocks:
  - SignalOutReal
  - SignalSimpleOutReal
- Control modules:
  - SignalOutRealM
  - SignalSimpleOutRealM
- AnalogOutCC in ControlStandardLib

Input Signal Handling

Input objects receive a value from the I/O unit, which receives it from the process. I/O input units are represented in the hardware configuration section in the Project Explorer, where you can configure the measuring range and units of measurement.

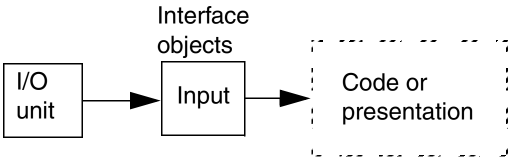


Figure 89. Handling of input signals

Table 13. Standard library types for input signal handling

Type Name	Library	Type	Description
SignalInReal(M)	SignalLib	Function block and Control module((1))	SignalInReal has an analog input, of *RealIO* data type, with several supervision functions, such as alarm and event levels, and interaction windows. SignalInReal has a first-order, low-pass filter built in. The input is intended to be connected to an analog input I/O variable. The signal output is of *real* data type (Function blocks) and ControlConnection (Control modules).
SignalSimpleInReal(M)	SignalLib	Function block and Control module(1)	SignalSimpleInReal is a version of SignalInReal (SignalSimpleInRealM is a version of SignalInRealM) that only handles one high and one low level. This simple type consumes less memory than SignalInReal.

Table 13. Standard library types for input signal handling (Continued)

Type Name	Library	Type	Description
SignalBasicInReal	SignalBasicLib	Function block	SignalBasicInReal is used for overview and forcing of analog input signals of data type RealIO. The input signal value is filtered, i.e. rapid changes are delayed according to the FilterTime value in InteractionPar. If a redundant switchover occurs, the output value change is smoothened according to the RedIncDecLim parameter. Error is set to true when input IO status is error marked.
AnalogInCC	ControlStandard Lib	Control module[1]	AnalogInCC receives the measured analog input value from the I/O unit and converts the input signal of RealIO data type to the common ControlConnection data type. AnalogInCC has a built-in first-order, low-pass filter. The analog input signal may be supervised by a Level6CC control module type with alarm levels, see Supervision on page 195. The analog input signal may also be supervised in bar graphs or histograms and controlled manually. In a control loop application, AnalogInCC normally precedes a PidCC controller.

[1] The control module type has a voting parameter that can be connected to a vote control module type. See Vote Control Module Types on page 381.

Output Signal Handling

The chain of objects in a control loop must end with one of the following objects (excluding function blocks) for the output signals, see Table 14 and Table 15.

Signals or values from the code can go directly to an I/O unit, or be handled in control modules before the output signal goes further to an output interface and then on to an I/O unit, and finally out to an actuator in the process.

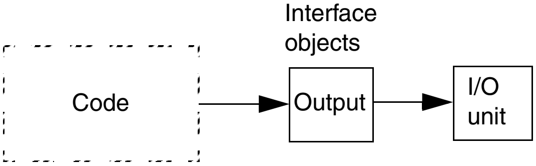


Figure 90. Handling of output signals

Table 14. Standard library types for analog output

Type Name	**Library**	**Type**	**Description**
SignalOutReal(M)	SignalLib	Function block and Control module (1)	SignalOutReal has an analog input of real data type and SignalOutRealM has an input of ControlConnection data type. Both object types have outputs of RealIO data type and are equipped with signal quality supervision with alarm functions and faceplates. The signal output, of RealIO data type, is intended to be connected to an analogoutput I/O variable.
SignalSimpleOutReal(M)	SignalLib	Function block and Control module 1	SignalSimpleOutReal has analog output of RealIO (Function blocks) and ControlConnection (Control modules) SignalSimpleOutReal is a version of SignalOutReal and SignalSimpleOutRealM is a version of SignalOutRealM. This simple type consumes less memory.

Table 14. Standard library types for analog output (Continued)

Type Name	Library	Type	Description
SignalBasicOutReal	SignalBasicLib	Function block	The function block SignalBasicOutReal is used for overview and forcing of analog output signals of data type RealIO. The input value is transferred to the output signal value and is limited within the range of the output parameter. Error is set to true when output IO status is error marked. Warning is set when the input is out of range defined by the output parameter. ParError is set to true when the range components of the output parameter are erroneous like that the maximum is less than the minimum.
AnalogOutCC	ControlStandardLib	Control module	AnalogOutCC writes, scales, or converts the following, from the *ControlConnection* signal type, to the RealIO signal type: - analog output signals to actuators via I/O units, - variables to the local system or a distributed system. The analog output may be supervised with a control module of the Level6CC, Level4CC, or Level2CC type, see Supervision on page 195. AnalogOutCC has an extension possibility for stiction compensation, see Stiction Compensator on page 179. The analog output may also be supervised in bar graphs or histograms and controlled manually in a interaction window. AnalogOutCC often succeeds a PID control module.

(1) The control module type has a voting parameter that can be used for connections to a vote control module type. See Vote Control Module Types on page 381.

Backtracking function in analog output objects

A backtracking function can be activated in Local mode via the parameter FeedbackPos. A local value from (for example) a level indicator can be sent (via an analog input object) with ControlConnection back to the analog output object's backtracking function.

This function will provide a bumpless transfer when the analog output object switches back from local mode to auto mode.

This function is valid only for control modules SignalOutRealM, SignalSimpleOutRealM and AnalogOutCC.

Converting Controller Output to a Digital Output Signal

Sometimes it might be desirable to convert controller output to a digital output signal, see Table 15.

In some cases, PulseWidthCC or ThreePosCC can be used for signal handling. ThreePosCC or PulseWidthCC usually follow upon a PID control module.

Table 15. Standard library types for digital output signals

Type Name	Library	Type	Description
ThreePosCC	Control StandardLib	Control module	ThreePosCC should be used as the end of a three-position control loop. (This is described in more detail in section Additional Control Functions on page 177. Digital signals are used to modify the state of connected devices. The digital outputs cannot be activated at the same time and when there is no need to change the output, neither of them is activated.

Table 15. Standard library types for digital output signals

PulseWidthCC	Control StandardLib	Control module	PulseWidthCC converts the analog signal from the control loop into a digital output signal. The digital signal is periodic with a selectable pulse width proportional to the value of the analog signal.
ThreePosReal	Control SimpleLib	Function block	ThreePosReal is a three-position converter from a real input to two Boolean outputs (increase/ decrease), similar to the ThreePosCC control module. The function block can be used with or without feedback from the actuator.

Manual-Auto Control

The ManualAutoCC control module lets you view the status of a signal of *ControlConnection* data type, at any location, but preferably before an output, and then change its value in Manual mode.

Table 16. Standard library objects for Manual-Auto control

Type Name	Library	Type	Description
ManualAutoCC	Control StandardLib	Control module	ManualAutoCC makes is possible to enter values manually into a control loop, for example, range, units of measurement or limits, and to supervise the control values graphically in bar graphs and trim curves. Normally, ManualAutoCC is configured in series on the ControlConnection line between two other control modules. If ManualAutoCC is located first in a control loop, it can be in Manual mode only, and when located last in a control loop, it can be in Auto mode only. After many different calculations, the unit of measurement of a signal may require simplification, which can be entered in a ManualAutoCC control module. See Table 41 on page 378.

Supervision

A level detector is a trip switch (low or high) for supervision of an analog signal. A low-level trip indicates when the input signal drops below any of several defined low detect levels, and a high trip correspondingly indicates when the supervised signal exceeds any of several defined high detect levels.

Table 17. Standard library types for supervision

Type Name	Library	Type	Description
LevelHigh and LevelLow	BasicLib	Function block	LevelHigh and LevelLow are trip switches for the supervision of an analog signal of real type at an optional number of levels. The input signal may be given a certain degree of hysteresis, which prevents the level detector output signal from repeatedly changing state when the supervised input signal varies near the detection level.
SignalReal	SignalLib	Function block	SignalReal has an analog input and an output, both of real data type, with several supervision functions, such as alarm and event levels, and interaction windows. The input and output are intended to be connected to real variables in an application.
SignalRealCalcOutM(1)	SignalLib	Control Module	SignalRealCalcOutM is a version of SignalReal that handles input connections from a vote control module. Input/Output is ControlConnection.
SignalRealCalcInM[1]	SignalLib	Control Module	SignalRealCalcInM is a version of SignalReal that handles connections to a vote control module. Input/Output is ControlConnection.

Table 17. Standard library types for supervision (Continued)

Type Name	Library	Type	Description
Level6CC[1]	Control Standard Lib	Control module	Level6CC is a supervisor object for level detection of a *ControlConnection* signal with six configurable alarm and event detection levels: H (High), HH, HHH, L (Low), LL, and LLL. Supervision may be absolute or relative to a reference signal. Level6CC also has hysteresis and filter time for alarm and event levels. The H and L levels are mainly used for logical circuits. The HH and LL levels are intended to be used as conditions for alarm generation. The HHH and LLL levels are intended to stop processes, but there are no limitations on their use. All levels may generate alarms. The presentation color of levels and graphs is defined by project constants. Each level has a logical color, and for each color, there is a color setting. See online help for more information. Three information types are given for each level: alarm condition state, a Boolean alarm condition parameter, a presentation signal of *Level6Connection* type. The latter, which is parameter connected only, may be used in control objects, in analog and digital interface control modules, and in ManualAutoCC, to show alarm levels in graph windows. Level6CC handles connections to vote control module.
Level4CC[1] and Level2CC[1]	Control Standard Lib	Control module	Level4CC and Level2CC are simplified versions of Level6CC, restricted to 4 and 2 levels, respectively. These types consume less memory and should be used when 4 or 2 level supervision is enough. Level4CC and Level2CC handles connections to vote control module.

(1) See also Vote Control Module Types on page 381.

Signal Quality and Status

The supervision of signal quality and handling of signal errors from, for example, a transmitter, or from the I/O interface system, is important in many processes.

Normally, the quality of a signal has its origin in the signal interface. The quality of a signal is defined as either GOOD, UNCERTAIN or BAD. A signal underflow or overflow gives an UNCERTAIN signal quality. A hardware error gives a BAD signal quality.

Table 18. Standard library objects for signal quality supervision

Type Name	**Library**	**Type**	**Description**
SignalSupervisionCC	Control StandardLib	Control module	SignalSupervisionCC supervises the *ControlConnection* signal quality and handles signal errors with configurable alarm and event settings. SignalSupervisionCC has three different modes. • Through mode lets the signal pass without any action. • Freeze mode may freeze the output. If the input signal is not of GOOD quality, the output is frozen and an alarm is given. • Predetermined mode. If the input signal is not of GOOD quality, the output is set to a preset value and an alarm is given. The pre-determined value is reached by ramping.

Calculation

There are a large number of system functions and control modules for mathematical calculations of signals as well as mean, median and majority calculations.

Table 19. Standard library objects for mathematical calculations

Type Name	**Library**	**Type**	**Description**
MedianReal and MedianDint	Basic	Function block	MedianReal and MedianDint calculate the median value of an optional number of input values of real and dint types, respectively.
MajorityReal	Basic	Function block	MajorityReal calculates the mean value of a number of signals of the real type, within a selectable deviation value. MajorityReal can, for example, be used to exclude a divergent value in a redundant calculation or to measure a number of signals.
AddCC	Control ExtendedLib	Control module	AddCC executes the addition Out = In1 + In2.
SubCC	Control ExtendedLib	Control module	SubCC executes the subtraction Out = In1 - In2.
MultCC	Control ExtendedLib	Control module	MultCC executes the multiplication Out = In1 * In2.
XRaisedToYCC	Control ExtendedLib	Control module	XRaisedToYCC executes the xy function Out = In1 raised to the power of In2.
SqrtCC	Control ExtendedLib	Control module	SqrtCC executes the square root Out = Sqrt(In).
DivCC	Control ExtendedLib	Control module	DivCC executes the division Out = In1 / In2.

Table 19. Standard library objects for mathematical calculations (Continued)

Type Name	Library	Type	Description
Mean4ExcludeBadCC Mean8ExcludeBadCC Mean12ExcludeBadCC	Control ExtendedLib	Control module	Bad values collected by several transmitters can be excluded by using MeanXExcludeBadCC control modules to analyze the status and value of a signal. Which module type to use depends on the number of input signals. Bad signals are excluded, and the mean value is calculated for the remaining signals. If status is not GOOD or if a value is extreme, it is omitted. The mean value of the remaining inputs is then taken as the output value. When only one valid input value exists, this value is used as the output signal. If no input signal is accepted, the output is a value with status BAD. For example, this might be useful when a number of temperature transmitters are placed at the bottom of a boiler, and some have been covered with dust, and therefore return significantly higher or lower values than the others. If a transmitter is broken, the corresponding valid information is false and this value is omitted by this reason. If the transmitter however transmits a valid signal but the value has a significant difference from the others, this value will also be omitted.

MeanXExcludeBadCC may also be used for processing values to controllers.

Signal Handling

Derivative Objects

Derivative objects are normally used to detect changes in a signal, to predict a control activity. A derivative object may also act as a high-pass filter.

Table 20. Derivative standard library objects

Type Name	Library	Type	Description
DerivativeReal	Control SimpleLib	Function block	DerivativeReal is a combined first-order, low-pass filter and a differentiator. The filter is used to smoothen the derivative action. The output may be forced to track an external signal. Transition from tracking is bumpless. The transition to the tracking value is dependent on the deviation from the current output value when tracking is enabled. DerivativeReal and DerivativeCC (below) have similar functionality.
DerivativeCC	Control ExtendedLib	Control module	DerivativeCC is a derivative control module with adjustable filter time. The input is filtered by a first-order, low-pass filter which determines the time during which the derivative action is to decline. The sampling time must be considerably shorter than the filter time, at least 3 to 10 times. The filter output is then differentiated. During feedforward control, when the signal changes, there is a need for an amplification of short duration. The DerivativeCC filter can be used to smoothen the derivative action.

Integrator Objects

Integrators accumulate the input signals and present the sum as an output. For instance, a flow may be integrated, in order to compute a volume. The input signal is integrated as long as the integrator is enabled. See Figure 91.

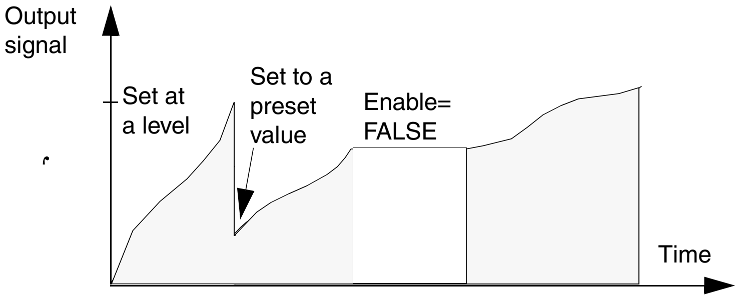


Figure 91. The integrator function sums the input signal (In) value over time

Table 21. Integrator standard library objects

Type Name	Library	Type	Description
IntegratorReal	Control SimpleLib	Function block	The IntegratorReal output can be limited, and may be forced to track an external signal. Transitions from tracking and limiting are bumpless. Further increase or decrease of the output can be inhibited. At reset, the integral part is set to zero.
IntegratorCC	Control ExtendedLib	Control module	IntegratorCC offers the same functions as the IntegratorReal function block type. It can also be reset to a selectable, predetermined output value.

Flow Calculation

Flow measurements can be made by meters giving an analog signal directly proportional to the flow, or by differential pressure measurement across a measuring flange.

Table 22. Standard library objects for flow calculation

Type Name	Library	Type	Description
FlowCC	Control ExtendedLib	Control module	FlowCC calculates the value directly proportional to the flow, or by differential pressure measurement across a orifice plate. FlowCC has compensation inputs for the surrounding temperature and pressure. Given the maximum flow, the flow can be calibrated for other operational cases with real measurements. FlowCC may be used as input to various calculations, or as a process value in a controller.

Time Average

The time average value of the input over a specified number of samples can be calculated.

Table 23. Standard library objects for time average calculation

Type Name	Library	Type	Description
TimeAverageCC	Control ExtendedLib	Control module	TimeAverageCC reads a parameter of ControlConnection data type, and calculates the moving average value over a specified number of samples. There is also a parameter in TimeAverageCC that helps to configure the module to calculate the average in three different ways: • Continuous sampling based on the configured number of samples • Setting a sample time for calculation. This sample time can be equal to or greater than the modules’s execution scan time • Sampling based on request.

Signal Reshaping

If a transmitter signal is non-linear, a piecewise linear signal object can be used to reshape and linearize it. Linearization is performed before the signal is connected to the controller or a calculator function. A piecewise linear signal object is also useful in cases of non-linear relations between values in one or two dimensions, for example, absolute and relative humidity, or pressure versus density for a liquid or for steam.

Table 24. Standard library objects for flow signal reshaping

Type Name	Library	Type	Description
PiecewiseLinearReal	Control SimpleLib	Function block	PiecewiseLinearReal has a number of predefined input-output pairs. Values between these pairs are calculated by linear interpolation. PiecewiseLinearReal can be used to define a non-linear function y=f(x). The maximum number of data points is 21, and there is an interaction window making data input easier. Intervals between different break points do not have to be equal. X values must be increasing. Below the first point, and above the last point, linear extrapolation to infinity is used. The pseudo inverse of the defined function can be calculated for a given input.
PiecewiseLinear2DReal	Control SimpleLib	Function block	The PiecewiseLinear2DReal function block type takes two inputs, which means that a non-linear surface, z=f(x,y) can be specified. The restriction on the x, and the y values is that they must be increasing. A maximum of 21 x values and 11 y values can be specified, that is 231 data points. An interaction window can be used to edit the data. The functionality of PiecewiseLinear2DReal is the same as for the PiecewiseLinear2DCC control module below.

Table 24. Standard library objects for flow signal reshaping (Continued)

Type Name	Library	Type	Description
PiecewiseLinearCC	Control Extended Lib	Control module	PiecewiseLinearCC has the same functions as the PiecewiseLinearReal (see above). If a transmitter signal is non-linear, PiecewiseLinearCC can be used to linearize it. Linearization is performed before the signal is connected to a controller. The controller can then be tuned for optimized function, independent of the non-linearity of the transmitter. If, in the future, the transmitter is replaced, the new one may have other characteristics. The new function, given in the transmitter manufacturer's technical information data sheet, may then be entered in this PiecewiseLinearCC control module and the control loop will work the same with the new transmitter. You can also connect PiecewiseLinearCC to a controller output to linearize a non-linear valve characteristic.
PiecewiseLinearExtension	Control Extended Lib	Control module	PiecewiseLinearExtension modules are used as add-ons to PiecewiseLinearCC, in order to add multiples of 20 points, for large numbers of data.
PiecewiseLinear2DCC	Control Extended Lib	Control module	PiecewiseLinear2DCC has the same functions as PiecewiseLinear2DReal (see above).

The above control modules are suitable for changing signals according to non-linear static functions.

A setpoint curve may be generated by a PiecewiseLinearCC control module to a succeeding controller by having a preceding IntegratorCC control module.

The inverse of the f(In) function can be calculated with the *InInverse* and *OutInverse* parameters:

- The relation is *OutInverse* = f^{-1}*(InInverse)*. The inverse calculation is performed on the specified data points, where *OutInverse* is in the interval x1 - xn, and xn is the last point used. All functions can of course not be inverted. In such cases a pseudo inverse is calculated using the curve between the maximum and minimum defined values of y.
- If the inverse is still not unique, the smallest value is chosen. This is illustrated in Figure 92, which shows the inverse calculation for two different *InInverse* values, y1 and y2. For y1 there are two possible inverse values. The rightmost is chosen, since the inverse is calculated from the curve between the maximum and minimum values. For y2, there are four possible inverse values, three of which are inside the inverse range. The leftmost of these is chosen. If *InInverse* is greater than the maximum defined value of y or less than the minimum defined value of y, the inverse calculation is based on using the maximum or minimum value respectively.

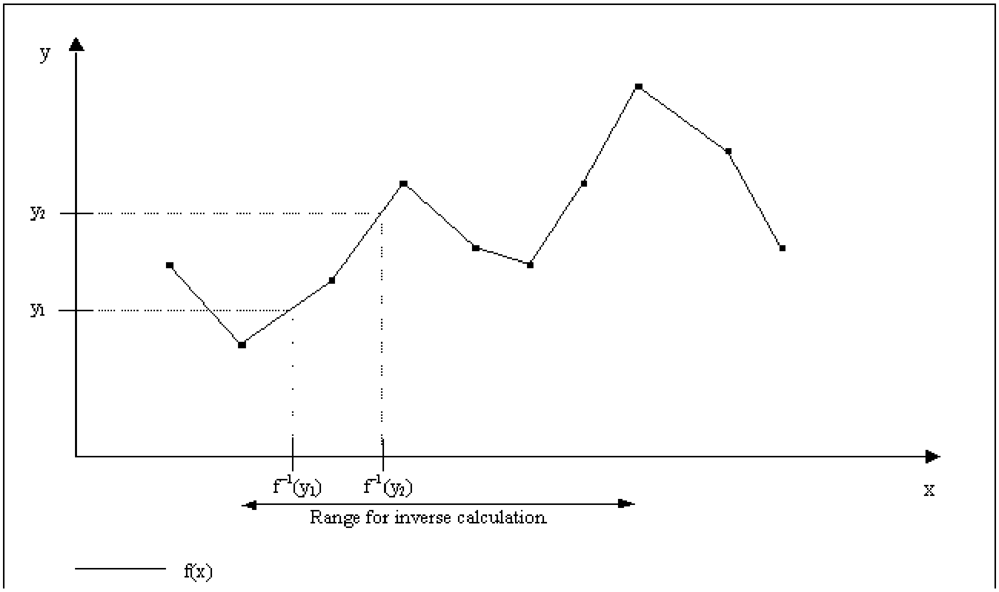


Figure 92. Calculation of the inverse function f(x)

Filters

First-order, low-pass filters can be used to, for example, reduce the amount of high-frequency noise in analog signals created by an analog transmitter or the control system environment. A filtering time can be set.

Table 25. Standard library objects for filtering

Type Name	Library	Type	Description
FilterReal	Control SimpleLib	Function block	FilterReal is a single-pole low-pass filter. The transfer function is: *G(s) = 1 / (1 + s*FiltT)* where *FiltT* is the filter time constant.
Filter2Real	Control SimpleLib	Function block	Filter2PReal is a low-pass filter with one zero and two complex poles. Their outputs can be forced to track an external signal. The transition from tracking is bumpless. The transfer function is *G(s) = (1+s*ZFiltT) / (1+ s*2*Damping*PFiltT+s2*PFiltT2)* where *ZFiltT* is the time constant for the derivation. *Damping* is the damping factor and *PFiltT* is 1/the resonance angular frequency.
LeadLagReal	Control SimpleLib	Function block	LeadLagReal is used as a lead or a lag function, that is, a derivative or an integration limiter, respectively, determined by the relation between two input time constants. LeadLagReal, of real type, can be forced to track an external signal and the transition from tracking is bumpless. The transfer function is: *G(s) = (1+s*LeadT)/(1+s*LagT)* where *LeadT* is the time constant for the derivative lead and *LagT* is the time constant for the integration lag.
FilterCC	Control ExtendedLib	Control module	FilterCC is a first-order, low-pass filter for *ControlConnection* signals. The transfer function is the same as in FilterReal (see above).

Table 25. Standard library objects for filtering (Continued)

Type Name	Library	Type	Description
Filter2PCC	Control ExtendedLib	Control module	Filter2PCC is a low-pass filter with one zero and two complex poles. The transfer function is the same as in Filter2PReal (see above).
DecoupleFilterCC	Control AdvancedLib	Control module	The DecoupleFilterCC is used to dynamically decouple cross coupled systems with two input signal. In a system with two inputs and two outputs which are cross coupled i.e. that one input affects both outputs, this leads to a difficult controlling problem. By introducing a DecoupleFilterCC, the interaction between these inputs and outputs can be shaped so that only one input affects one output. Thus by using DecoupleFilterCC, one can transform a Two In - Two Out system (TITO) into two Single In - Single Out systems (SISO).
LeadLagCC	Control ExtendedLib	Control module	LeadLagCC has the same functions and transfer function as LeadLagReal (see above).

Low-pass filter control modules may be used to flatten a signal step with a high derivative component into a continuous signal, without steps. This filter function is also included in the analog input control module described in Input and Output Signal Handling on page 184.

Delays

Dead-time control and loop calculation are required in control systems with long transport lags, such as conveyor feed systems.

Table 26. Standard library objects for signal delays

Type Name	Library	Type	Description
DelayCC	Control ExtendedLib	Control module	DelayCC, in combination with other control loop control modules, delays a *ControlConnection* signal for a predetermined time. The delay time may also be a calculated variable, or a controller output.

Branch Objects

There are a number of branch objects that split signals into two or several branches and manage different aspects of the splitting procedure.

Also, many processes work with large differences in product flows. It may then be necessary to use two valves in parallel, one for small flows and one for large flows. In other processes, there might be a need to split a signal into two branches, one slow and one fast.

To ensure that two process valves working in parallel, together give the required flow a signal range might need to be divided into two output signal ranges, each of which is an output part of the input signal.

The Control libraries contain a number of types for these purposes.

Table 27. Standard library signal branch objects

Type Name	Library	Type	Description
BranchCC Branch4CC	Control StandardLib	Control module	BranchCC and Branch4CC split a signal of ControlConnection type into two or four branches, respectively, with output signals equal to the input signal, with the same backtracking functions. A signal of ControlConnection type can be duplicated for calculation with several other signals using these control modules. A measured signal value can be duplicated for use as an input signal to several controllers using these control modules.
SplitRangeCC	Control StandardLib	Control module	SplitRangeCC splits the output signal to the valves into two output signal ranges; one branch for each valve. You can scale the output signal ranges independently. A SplitRangeCC split range control module can be used in a controller output signal.

Table 27. Standard library signal branch objects (Continued)

Type Name	**Library**	**Type**	**Description**
MidRangeCC	Control StandardLib	Control module	MidRangeCC splits a *ControlConnection* signal into two branches, one slow and one fast branch. For example, if two valves act in parallel on the same flow, there is a need to split a signal into two branches. One valve may be bigger and slower and have a larger operating range. The other may be small and fast and used to control small perturbations in the flow. The fast branch reacts faster to changes in the signal and then works around an approximate middle setpoint for its operating range. Meanwhile, the slower branch takes control.
CommonRangeCC	Control StandardLib	Control module	CommonRangeCC splits a *ControlConnection* signal into two branches which, when added together, give the value of the input signal. CommonRangeCC is used to ensure that two valves in parallel, with a specified ratio between the outputs, give the required flow, when added. The input signal range is divided into two output signal ranges, each of which is an output part of the input signal, for example, in a 20/80% ratio. If one output signal is in Manual mode and is changed (for example, 20% is changed to 10%), the other output signal overrides its default value (80%) and sets its output to 90% in order to maintain the total output (100%). CommonRangeCC can also be used for quotient control. In such applications the input for the first output is connected to an output from a quotient controller. The input is connected to an output from a controller for the addition of the outputs.

Signal Tapping

There are functions for tapping off signals in the same signal range. One function taps off signals of the *ControlConnection* type. Another can tap off signals of the *real* type from a signal of the *ControlConnection* type. A signal tap is a kind of listening control module on the main signal flow. The tapped signal is an exact copy of the input signal. Modules connected to the tap output must not, under any circumstances, influence the main signal flow.

Table 28. Standard library objects for signal tapping

Type Name	Library	Type	Description
TapCC	Control StandardLib	Control module	TapCC divides a *ControlConnection* signal into two branches, where one has backtracking capability, and the other does not. The latter is called the tapped signal. Use this control module type with care.
TapRealCC	Control StandardLib	Control module	TapRealCC extracts the value component, as an exact copy, from a *ControlConnection* signal to a real value. Use this control module type with care. Use TapRealCC to use extract the *real* value of a signal in, for example, calculations.

Selector Objects

In signal handling and in control loop applications it is often necessary to choose between two or more signals. This subsection describes a number of functions, function blocks and control modules for this purpose.

Table 29. Standard functions and library objects for signal selection

Type Name	Library	Type	Description
sel *mux*	(system function)	Function	*sel* selects one signal of two depending on whether a given expression is evaluated as True or False. *mux* works as a multiplexor, with several inputs and one output, of most data types, but inputs have to be of the same type. The user selects the input signal to be forwarded to the output. For signal handling, it is sufficient to use the *sel* or *mux* system functions, for example, to select a signal for indication. When choosing between signals from, for example, two controllers, consider that the non-selected signal branch is open, and no information is sent backwards. This may require some means to remedy controller output signal drifting, such as backtracking.

Table 29. Standard functions and library objects for signal selection (Continued)

Type Name	Library	Type	Description
SelectorCC Selector4CC	Control StandardLib	Control module	The SelectorCC and Selector4CC selector control modules select one of two or several inputs, of ControlConnection type, for example, setpoints. In SelectorCC you select which of two input signals is allowed through, by means of a Boolean signal. To select between four input signals a Selector4CC selector control module with four inputs is used. For more than four input signals, several Selector4CC control modules can be connected in a chain. The output of the first Selector4CC goes to the input of the second, and so on. By means of an integer signal going to and between the control modules it is possible to select which of the input signals to allow through. For signal handling only it is possible to use the SelectorCC and Selector4CC selector control modules. High demands on information in both directions are fulfilled. For example, backtracking of the not-selected inputs is supported automatically.
SelectGoodCC SelectGood4CC	Control StandardLib	Control module	The SelectGoodCC and SelectGood4CC control modules can be used for selection of the first GOOD signal between either two or up to four inputs of ControlConnection type. If no GOOD signal exists, the output used is a copy of the first input. For signal handling only, it is possible to use the SelectorGoodCC and SelectorGood4CC selector control modules. High demands on information in both directions are fulfilled. For example, backtracking of the not-selected inputs is supported automatically. An I/O signal is considered to be GOOD when there is no overflow, no underflow and no missing interface.

Table 29. Standard functions and library objects for signal selection (Continued)

Type Name	**Library**	**Type**	**Description**
max *min*	(system function)	Function	*max* and *min* select the signal with the maximum or the minimum value. *max* and *min* are extensible for several input signals of many types, but the compared signals have to be of the same type.
MaxReal MaxDint MinReal MinDint	BasicLib	Function block	MaxReal, MaxDint, MinReal, and MinDint select the input signal with the maximum or the minimum value among an optional number of inputs. The selected input is written to the output signal. The MaxReal, MaxDint, MinReal, and MinDint function blocks compare signals of real or dint type. The function blocks also have a dead zone functionality for the inputs and an output that specifies the chosen input. Observe that the output signal will be discontinuous if the dead zone differs from zero (0). For signal handling only it is sufficient to use the min or max system functions or the MaxReal, MaxDint, MinReal, or MinDint function blocks. When the signal selection involves controllers in a control loop, high demands on information through the signals are required, for example, when using function blocks to solve a selection of controller outputs for override control of a heat pump.

Table 29. Standard functions and library objects for signal selection (Continued)

Type Name	Library	Type	Description
MaxCC Max4CC MinCC Min4CC	Control StandardLib	Control module	MaxCC, Max4CC, MinCC, and Min4CC calculate and select the maximum and minimum value of 2, 3, or 4 input *ControlConnection* signals with a certain tolerance. The selected input is written to the output signal. For MinCC and Min4CC, the value of inputs that are not connected to the output is limited. If they deviate more than their tolerance, their value is limited to the output plus the tolerance. In this way, the output of preceding modules is limited to the output plus the tolerance. As long as an input is within its tolerance, no value is sent to preceding control modules. The same thing applies for MaxCC and Max4CC, but with the opposite functionality. MaxCC, Max4CC, MinCC, and Min4CC can be used for signal handling.

The MinCC and Min4CC selector control modules select the minimum value of two and up to four connected input signals, respectively. MaxCC and Max4CC select the maximum value of two and up to four connected input signals, respectively. At least two inputs have to be connected but unconnected inputs are not considered in the selection. The inputs may have different measuring ranges.

If the output is backtracked from succeeding control modules all connected inputs are backtracked to the same value.

The user defines a tolerance for each input. An unselected input can deviate at most by that tolerance from the output. When an input is limited by the tolerance, that input value is sent to preceding control modules having an internal state. No value is sent backward as long as the input is within its tolerance from the output.

Assume for a MinCC that input one is the minimum signal and is consequently sent to the output.

To avoid input two drifting away it is limited by its tolerance from the output. If input two exceeds the output plus the input two tolerance, input two is limited to that value. This means that outputs of preceding control modules with internal states do not exceed that limited input two value. Consequently, if the input two value is within its tolerance of the output signal, no value is sent to preceding control modules. The opposite is true if instead input two signal is the smallest.

The same is valid for MaxCC. It selects the maximum of two input signals and sends it to the output. Accordingly, if the smaller input signal reaches its tolerance limit, the smaller input is assigned the output signal value minus its tolerance value. This means that outputs of preceding control modules with internal states do not fall below that limited input two value.

In control modules with four inputs the discussion is exactly the same. The three unselected inputs are treated in the same way as described above. These control modules can be used if, for example, controllers are connected to the inputs. The unselected controller output signal will stay close (differing by the tolerance at most) to the selected controller output. It is then ready to be the active controller. See the example for maximum and minimum selectors.

Do not set the tolerance values too low. Problems may occur if any of the input signals starts to oscillate or has a noise jamming overlay. If the tolerance parameters are set too narrow and a controller starts to oscillate, with an amplitude greater than the tolerance, problems may occur. The output signal from MaxCC and Max4CC can start to "integrate" up to 100% (MinCC and Min4CC lead to integration to 0).

Limiter Objects

In many cases it is necessary to limit a signal value, or to limit the rate of change of a signal.

Table 30. Standard library objects for limiting a signal

Type Name	Library	Type	Description
limit	(system function)	Function	*limit* works as a delimiter between a set minimum and a set maximum signal value of several data types. If the input is smaller than or equal to the minimum value, the output is assigned the minimum value. If the input is greater than or equal to the maximum value, the output is assigned the maximum value. An input value between the minimum and a maximum values is not changed.
VelocityLimiter Real	Control SimpleLib	Function block	VelocityLimiterReal is a ramp function that is used to limit the velocity of change for a signal of real type. The output can be forced to track an external signal and the transition from tracking is bumpless.

Table 30. Standard library objects for limiting a signal (Continued)

Type Name	Library	Type	Description
AccelerationLim Real	Control SimpleLib	Function block	AccelerationLimReal is a ramp function that is used to limit the velocity and acceleration of change for an input signal of real type. The maximum acceleration is given as the time allowed from steady state to maximum velocity of the output signal. It is also possible to configure AccelerationLimReal so that overshoots are minimized when the input is changed in discrete steps. The output may also be tracked and held at a certain value. For signal handling only it is sufficient to use the *limit* system functions or VelocityLimiterReal and AccelerationLimReal. When the limiter involves controllers in a control loop, high demands are placed on information through the signals.

Table 30. Standard library objects for limiting a signal (Continued)

Type Name	Library	Type	Description
LimiterCC LimiterHighCC LimiterLowCC	Control StandardLib	Control module	LimiterCC keeps the signal between a high limit and a low limit. LimiterCC is, in the first place, used to set high and low limits on the output from controllers. The controller itself has changeable limits on its output, but in a loop with one or more control modules on the controller output, the limitation is often required at the end of a chain of control modules. LimiterHighCC and LimiterLowCC maintain the signal below a high limit and above a low limit, respectively. The Limiter types have no internal state and cannot allow backtracking from the output, unless a preceding control module has the feature. If a high or low limit is reached, information is sent backward to preceding control modules to prevent the input from being increased. An output also indicates that the limit has been reached. If a succeeding control module orders this control module into Backtracking mode, this order is also transferred to preceding control modules.
VelocityLimiterCC	Control StandardLib	Control module	VelocityLimiterCC is used to limit the change rate of an increasing or decreasing signal. This will slow down changes in the output signal. The maximum rate of change for increasing and decreasing signals can be set independently. The output follows the input signal but the rate of change is limited. If the input signal changes faster than the output is allowed to change, the change rate of the output will be constant until the output has reached the input value.

Table 30. Standard library objects for limiting a signal (Continued)

Type Name	Library	Type	Description
AccelerationLimCC	Control StandardLib	Control module	AccelerationLimCC is a ramp function that is used to limit the velocity and acceleration of change for an input signal of ControlConnection type. The maximum acceleration is given as the time allowed from steady state to maximum velocity of the output signal. It is also possible to configure AccelerationLimCC so that overshoots are minimized when the input is changed in discrete steps. The output may also be tracked and held at a certain value.

Conversion

The conversion function blocks in the Basic library convert signals of various data types to and from signals of other data types.

The conversion control modules in the Standard Control library convert data types used in control loops to and from other data types for connections to application programs. These control modules are like an adaptor interface between an ordinary data type signal and a control loop signal of *ControlConnection* data type.

The following conversion functions are available. See also online help for detailed information on individual objects (select the object in Project Explorer and press F1).

Table 31. Conversion objects in standard libraries

Type Name	Library	Type	Description
CCToReal	Standard Control library	Control module	Convert a *ControlConnection* signal to a *real* signal.
CCToInteger	Standard Control library	Control module	Convert a *ControlConnection* signal to an *integer* signal.

Table 31. Conversion objects in standard libraries (Continued)

Type Name	Library	Type	Description
RealToCC	Standard Control library	Control module	Add a *real* signal to a *ControlConnection* signal.
RealIOToInteger	Standard Control library	Function block	Converts a scaled *RealIO* value to an integer value in the range 0-65535.
IntegerToRealIO	Standard Control library	Function block	Converts an integer value in the range 0-65535 to a scaled *RealIO* value with a measuring range, units and decimals. Can be treated as a physical *RealIO* value in the application.
BcToDint	Standard Control library	Function block	Converts data from an optional number of binary coded Boolean inputs and a sign input into a *dint*.
DintToBc	Standard Control library	Function block	Converts data from dint to an optional number of *Boolean* outputs, using binary coded conversion, and a sign output.
FirstOfNToDint	Standard Control library	Function block	Converts data from 1-of-N format with an optional number of *Boolean* inputs and a sign input, into a *dint*.
DintToFirstOfN	Standard Control library	Function block	Converts data from *dint* to an optional number of *Boolean* outputs using 1-of-N conversion, and a sign output.

Table 31. Conversion objects in standard libraries (Continued)

Type Name	Library	Type	Description
NBcdToDint	Standard Control library	Function block	Converts data from an optional number of *Boolean* inputs in groups of four, coded as BCD, and a sign input, into a *dint*.
DintToNBcd	Standard Control library	Function block	Converts from *dint* to an optional number of *Boolean* inputs in groups of four, coded as BCD, and a sign output.
GrayToDint	Standard Control library	Function block	Converts data from gray code with an optional number of *Boolean* inputs and a sign input, into a *dint*.

There are more functions for similar conversions in the Basic library. See online help for complete information.

Miscellaneous Objects

There are some additional types in the standard libraries that are useful when building control loops.

Table 32. Standard library objects for demultiplexing

Type Name	Library	Type	Description
StateCC	Control ExtendedLib	Control module	StateCC is used to break up control loops and create one scan delay. (For simple data types this problem is solved by means of variable:new and variable:old.) StateCC is needed since it is not possible to break code loops in a controller loop consisting of control modules only. StateCC delays the forward and backward components by one cycle scan. StateCC does not influence either the forward or the backward component of *ControlConnection*. StateCC is normally not used, because the system does not check that the code is executed in the correct order. However, if you are forced to break the automatic sorting of the input to output order and you have to make a connection of *ControlConnection* type backwards in the control loop, it is possible to use it. Do not use StateCC in any other case.
Threshold	Basic	Function block	Threshold can have an extensible number of inputs. Threshold determines when more than, or equal to, a given number of *Boolean* input values are True.

Control Loop Solutions

Introduction

This section describes how to use control module type templates from the Control Solution library (ControlSolutionLib) to create control solutions for commonly occurring customer processes. The section contains:

- A description of the Control Solution concept and the types in the Control Solution library, see Concept on page 226.
- Instructions on how to implement control loop solutions using the types in the Control Loop Solution library, see Design on page 226.
- An example on how to implement a control loop solution using the types in the Control Loop Solution library and change it to fit an intended usage, see Example on page 228.

Concept

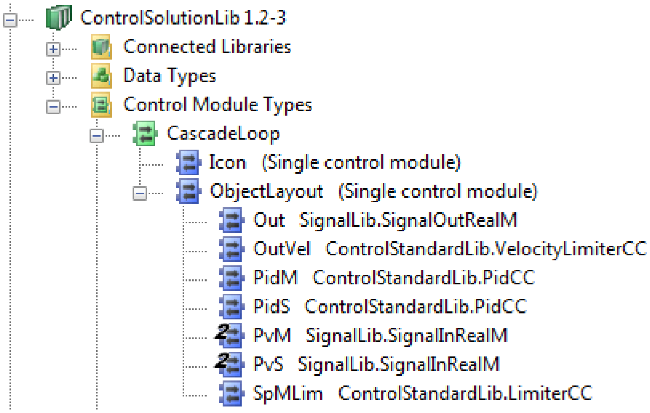
All the control module types in Control Solution library (ControlSolutionLib) provide a complete control solution, intended to be used directly in an application. The user requires only to connect the control module to I/Os and in some cases set some configuration parameters.

The control module types are ready-to-use solutions for frequently occurring control processes found at customers. They consist of a control solution with basic control module types and alarm handling.

The users may use the solutions directly, or create own types by making copies and change these to fit an intended usage, which may be level control, flow control, etc. These new types can then be populated with specific default values for controller tuning, alarm limits etc.

Design

A control solution template consists of several basic library types (sub control modules) from the standard libraries.



```
ControlSolutionLib 1.2-3
    Connected Libraries
    Data Types
    Control Module Types
        CascadeLoop
            Icon   (Single control module)
            ObjectLayout   (Single control module)
                Out   SignalLib.SignalOutRealM
                OutVel   ControlStandardLib.VelocityLimiterCC
                PidM   ControlStandardLib.PidCC
                PidS   ControlStandardLib.PidCC
                PvM   SignalLib.SignalInRealM
                PvS   SignalLib.SignalInRealM
                SpMLim   ControlStandardLib.LimiterCC
```

Figure 93. Standard types in a control loop solution (CascadeLoop)

Parameters

The parameters of the sub control modules are connected to variables and given appropriate default values. In a user specific object these variables, for example parts of the alarm configuration, can be moved to the parameter list. The default values of the variables/parameters, such as alarm limits, can also be changed, to fit an intended usage.

The variables of a sub control module can be moved to the parameter list and vice versa.

All control solution templates have a Name parameter with the length of 22 characters. The Name parameters of the sub objects will be a name that is a concatenation between the name of the solution object and a suffix. The used suffix is the instance name of the corresponding sub object.

- **Example**

  If the Name parameter of a CascadeLoop instance of Figure 93 is FIC1234, the sub object PidM gets FIC1234PidM as Name parameter.

Alarm Handling

The solutions are configured with an appropriate set of level and deviation alarms.

The following sub control modules with alarms are used in the solutions:

- SignalInRealM
  Alarm configured for Error: Alarm levels of HH, H, L, and LL parameters are set to 95%, 90%, 10%, and 5% of the range, respectively. HHH and LLL parameters are disabled.
- SignalOutRealM
  Alarm configured for Error: All alarm levels are disabled.
- PidCC
  Alarm configured for DevPos and DevNeg: The deviations are set to ±10%.

Parameters of a control solution object are connected to the sub control module ObjectLayout (a single control module). The main solution object is Alarm Owner of all alarms from the solution, since Alarm Owner property of ObjectLayout is not enabled. For further information about the Alarm Owner Concept, See *AC 800M Configuration manual (3BSE040935*).*

The connection list of ObjectLayout must be updated when a parameter is renamed. In the most cases the corresponding parameter of ObjectLayout also must be changed, to reflect the change in the parameter interface of the solution object. When a parameter of ObjectLayout is renamed, the corresponding connection needs to be reconnected. The connections of the sub objects of ObjectLayout need to be changed to the new parameter name.

Example

This example shows how to customize a control solution template, by using types in the Control Loop Solution library and how to change the new object to fit an intended usage. It also illustrates some important control loop solution concepts and relations.

- Adding Object gives an example of how to add a basic control object to a new created control solution object.

Adding Object in Control Builder

In this example a new control solution object, based on the SingleLoop template, is created. The new control solution template is extended with a delay (SpDelay of DelayCC type) after the setpoint limiter (SpLim) to delay the setpoint value.

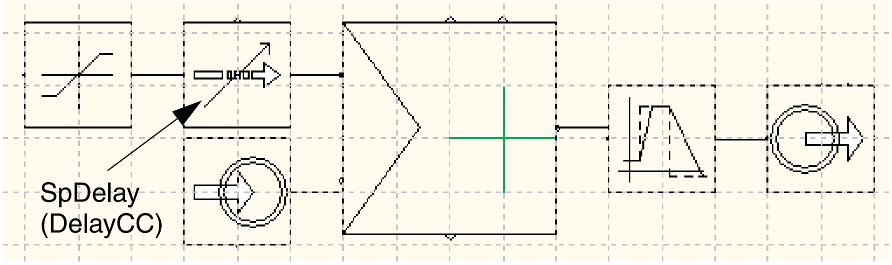


Figure 94. The result in of the new control solution in CMD editor

1. In Control Builder, create a new library (user-defined library).
2. Make a copy of the SingleLoopPar data type, from ControlSolutionLib and paste it into the user-defined library (in this example MySolutionLib).

By copying the InteractionPar (SingleLoop) data type before the main control module type minimizes parameter connections.

3. Make a copy of the SingleLoop control module type, in ControlSolutionLib, and paste it into MySolutionLib.
4. In MySolutionLib, rename SingleLoop control module type to DelayLoop and rename SingleLoopPar data type to DelayLoopPar.

5. Connect libraries, that the solution template DelayLoop is dependent on, to MySolutionLib.

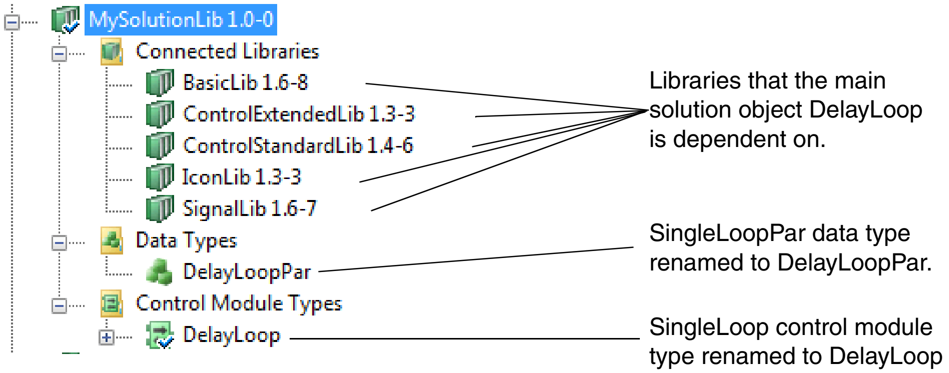
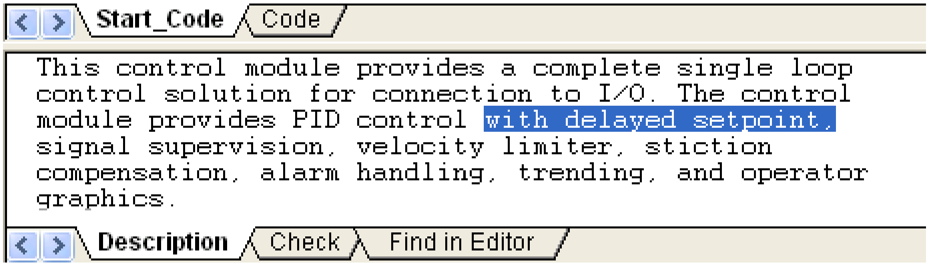


Figure 95. Connected Libraries and renamed objects

6. Right-click the DelayLoop and select **Editor**.

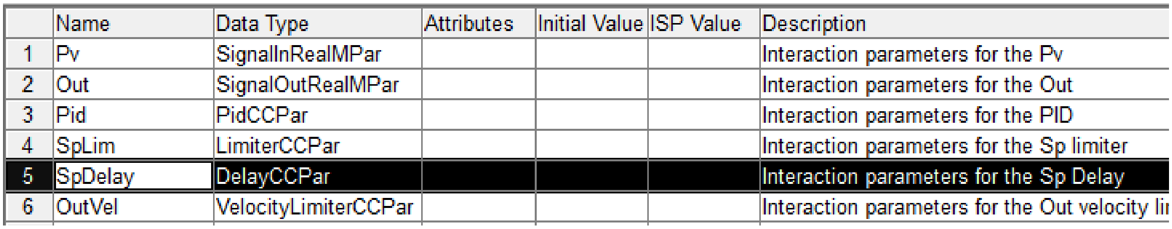
7. Update description of the solution object to reflect the new added functionality. In Message pane, select Description tab and add text about the delay functionality.



This control module provides a complete single loop control solution for connection to I/O. The control module provides PID control with delayed setpoint, signal supervision, velocity limiter, stiction compensation, alarm handling, trending, and operator graphics.

Figure 96. New added text in Description tab of DelayLoop

8. Expand Data Types folder and double-click the DelayLoopPar to open editor. Add SpDelay of data type DelayCCpar as a new component (interaction parameters of the setpoint delay). Save and close the editor.



	Name	Data Type	Attributes	Initial Value	ISP Value	Description
1	Pv	SignalInRealMPar				Interaction parameters for the Pv
2	Out	SignalOutRealMPar				Interaction parameters for the Out
3	Pid	PidCCPar				Interaction parameters for the PID
4	SpLim	LimiterCCPar				Interaction parameters for the Sp limiter
5	SpDelay	DelayCCPar				Interaction parameters for the Sp Delay
6	OutVel	VelocityLimiterCCPar				Interaction parameters for the Out velocity li

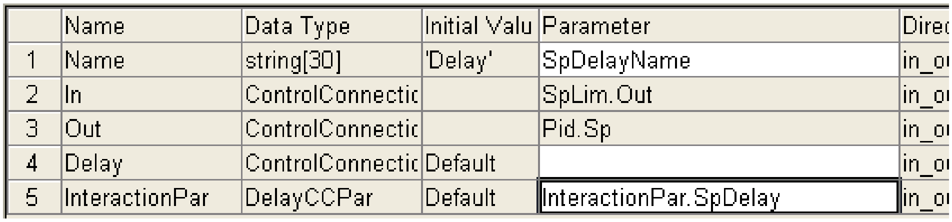
Figure 97. SpDelay added as new component for DelayLoopPar

9. Expand DelayLoop folder, right-click the ObjectLayout single control module and select the CMD editor. Provide space for the DelayCC control module by moving SpLim (LimiterCC) to the left.

10. Create an instance of DelayCC type (ControlExtendedLib) with instance name SpDelay and add it between the SpLim and Pid control modules. Make sure that the graphical connections are correct. See Figure 94.

In the next following steps the SpDelayName parameter is connected to NameSpDelay parameter in ObjectLayout. NameSpDelay parameter is then connected to the NameSpDelay variable of DelayLoop.

11. Connect parameters of SpDelay according to Figure 98 below. Save and close the Connections editor.

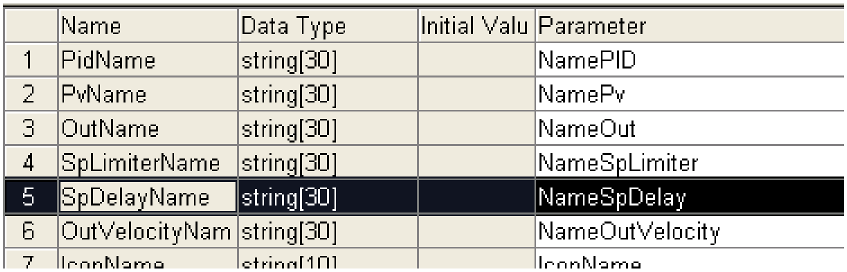


	Name	Data Type	Initial Valu	Parameter	Direc
1	Name	string[30]	'Delay'	SpDelayName	in_o
2	In	ControlConnectic		SpLim.Out	in_o
3	Out	ControlConnectic		Pid.Sp	in_o
4	Delay	ControlConnectic	Default		in_o
5	InteractionPar	DelayCCPar	Default	InteractionPar.SpDelay	in_o

Figure 98. Connections of SpDelay parameters

12. Right-click the ObjectLayout and select **Editor**. Add SpDelayName of string[30] data type as parameter. Save and close the editor.

13. Right-click the DelayLoop and select **Editor**. Add NameSpDelay of string[30] data type as variable.
14. Right-click ObjectLayout and select **Connections.** Connect NameSpDelay variable to SpDelayName parameter.



	Name	Data Type	Initial Valu	Parameter
1	PidName	string[30]		NamePID
2	PvName	string[30]		NamePv
3	OutName	string[30]		NameOut
4	SpLimiterName	string[30]		NameSpLimiter
5	SpDelayName	string[30]		NameSpDelay
6	OutVelocityNam	string[30]		NameOutVelocity
7	IconName	string[10]		IconName

Figure 99. Connection of NameSpDelay variable

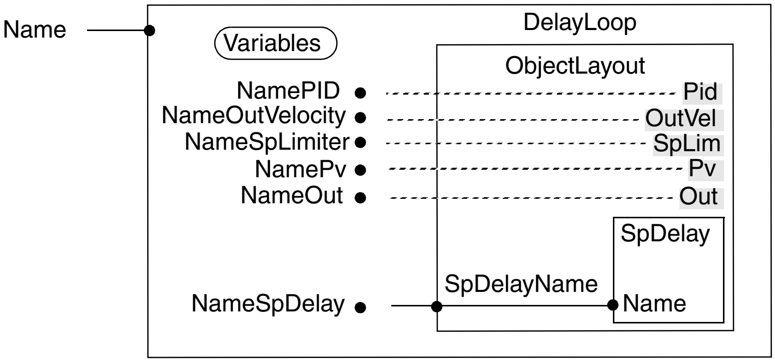
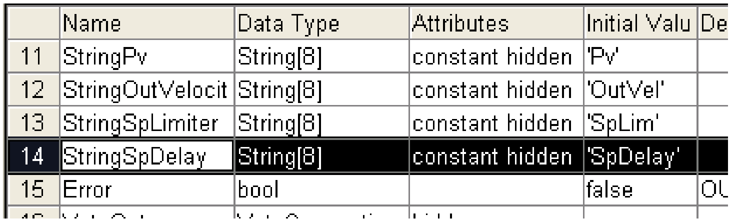


Figure 100. The result of DelayLoop parameter and variable connections

In the next following steps initial values for max. delay and delay (5.0 s) of the setpoint value are set to 30.0 s and 5.0 s respectively. The initial values can then be changed via the interaction parameters.

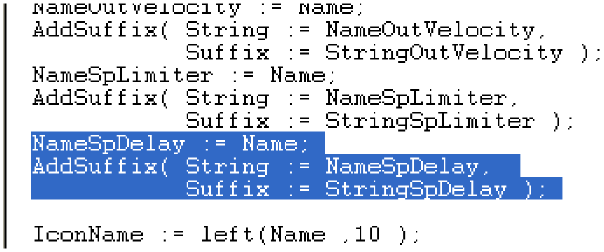
15. Right-click the DelayLoop and select **Editor**. Add StringSpDelay of string[8] data type with the initial value ‘SpDelay’ as variable.



	Name	Data Type	Attributes	Initial Valu	De
11	StringPv	String[8]	constant hidden	'Pv'	
12	StringOutVelocit	String[8]	constant hidden	'OutVel'	
13	StringSpLimiter	String[8]	constant hidden	'SpLim'	
14	StringSpDelay	String[8]	constant hidden	'SpDelay'	
15	Error	bool		false	OU

Figure 101. StringSpDelay added as data type with ‘SPDelay’ as Initial Value

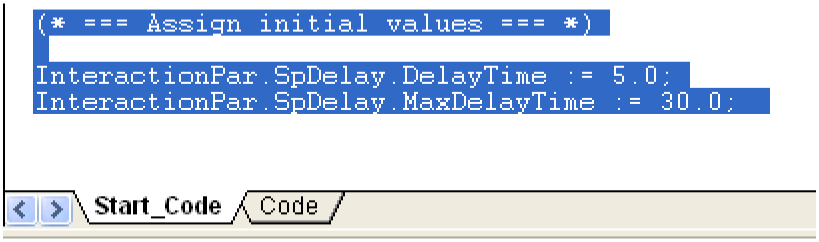
16. In the code pane, add code for name assignment.



```
NameOutVelocity := Name;
AddSuffix( String := NameOutVelocity,
           Suffix := StringOutVelocity );
NameSpLimiter := Name;
AddSuffix( String := NameSpLimiter,
           Suffix := StringSpLimiter );
NameSpDelay := Name;
AddSuffix( String := NameSpDelay,
           Suffix := StringSpDelay );

IconName := left(Name ,10 );
```

Figure 102. Code for name assignment

17. At the bottom of the code pane, add code to set the initial value of delay time to 5.0 s and initial value of max. delay time to 30.0 s.



```
(* === Assign initial values === *)

InteractionPar.SpDelay.DelayTime := 5.0;
InteractionPar.SpDelay.MaxDelayTime := 30.0;
```

Figure 103. Assigning of initial values

Figure 104. Extended Faceplate of DelayLoop

Basic Control Loop Examples

This sub section contains a number of examples that have been included to show how to implement analog process control using objects from the Control libraries, and to illustrate some important concepts and relations:

- Basic Control Loop Examples show two basic cascade loops, and an input selection function, all based on types from the Control libraries.

The following two examples show cascade loops built from basic components in the Control libraries.

Cascade Loop with Functions and Function Blocks

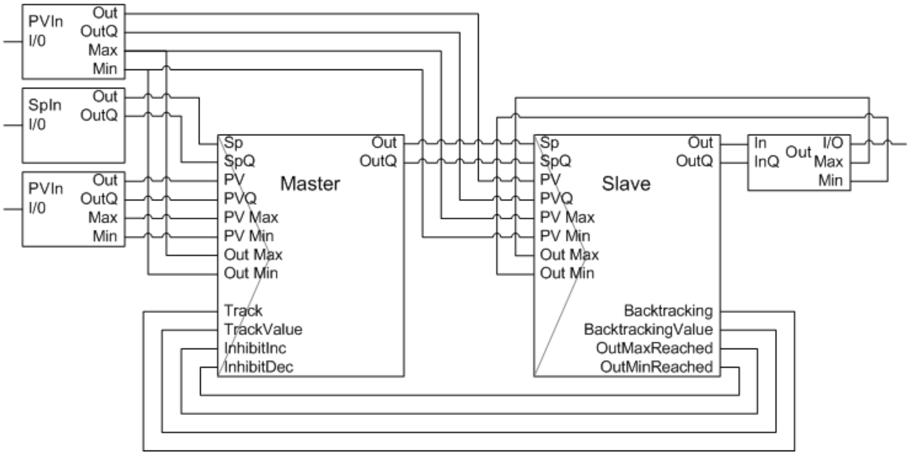


Figure 105. Example of a cascade control loop consisting of function blocks, with signal ranges added

Cascade Loop with Control Modules

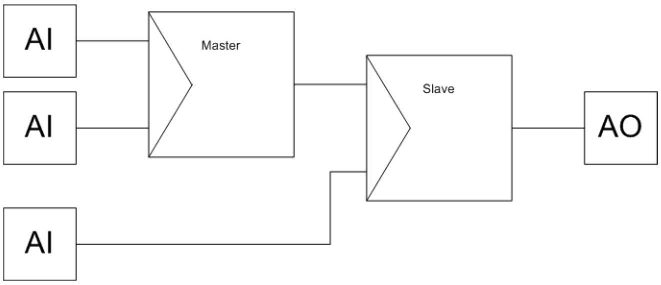


Figure 106. Example of a cascade control loop consisting of control modules

Signal Selection Example

When signal selection involves controllers in a control loop, higher demands are placed on the signal information. Use the MaxCC, Max4CC, MinCC, and Min4CC control modules for selection. This example shows how to use the control module Max4CC to select between the input from several control loops.

Information backward (backtracking), units of measurement and range, etc., transferred using structured signals of *ControlConnection* type, and in the control modules for these purposes. Information on which signal is selected is also given. If the output is backtracked from succeeding control modules, all connected inputs are backtracked to the same value.

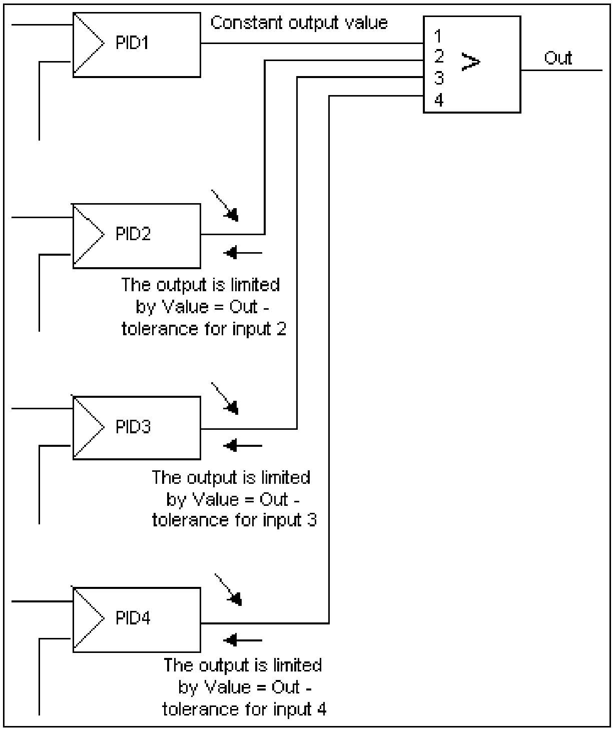

PID1
Constant output value
1
2
3
4
>
Out
PID2
The output is limited by Value = Out - tolerance for input 2
PID3
The output is limited by Value = Out - tolerance for input 3
PID4
The output is limited by Value = Out - tolerance for input 4

Figure 107. Example of a Max4CC control module with inputs from controller outputs

In Figure 107, controller PID1 has no control deviation, its inputs are stable and the PID1 output signal is constant. The other three controllers have a control deviation causing the controller outputs to show a decreasing tendency. This situation will force each of the non-selected PID2, PID3, and PID4 outputs to a value, assigned for each input to Max4CC, close to the selected controller output determined by the PID1 output minus the tolerance value. The controller output signals will then remain close to the selected controller output, ready to be the selected one. For example, if you use control modules to solve a selection of controller outputs for override control of a heat pump, the solution with a minimum selector control module may be as shown in Figure 108.

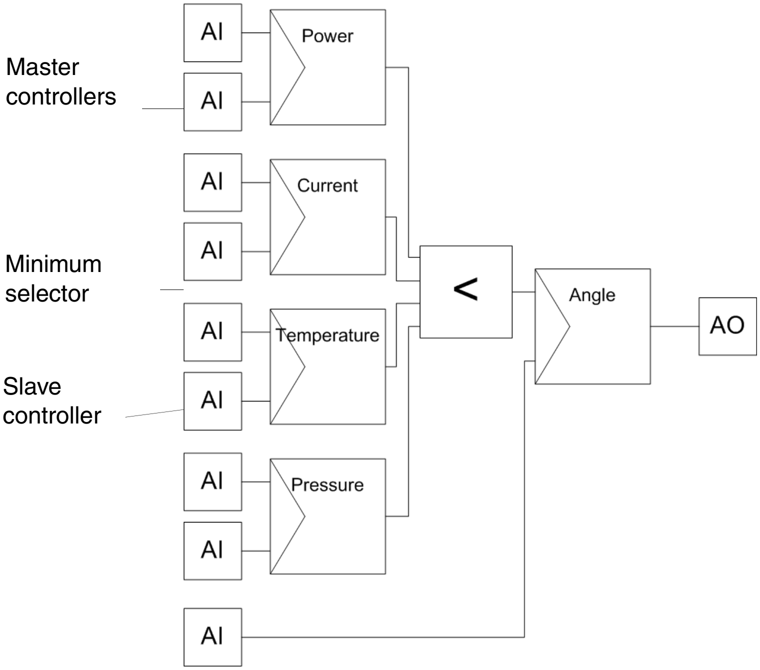


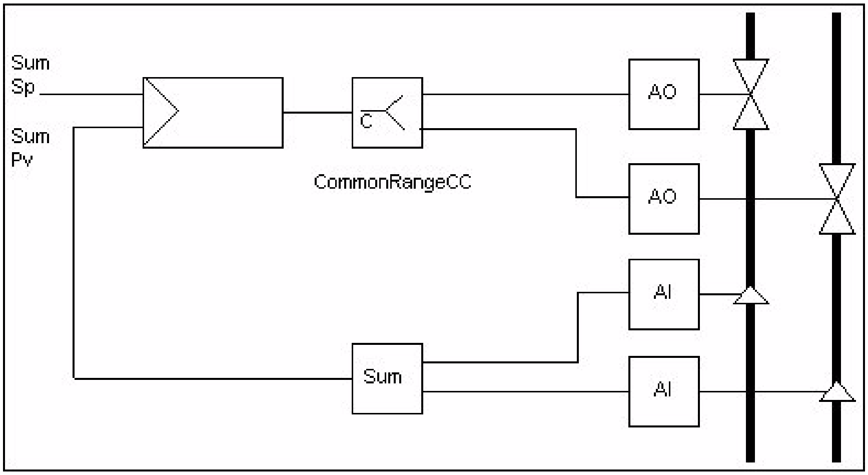
Figure 108. Example using control modules for override control in a heat pump using a minimum selector control module

All the components of *ControlConnection* are transmitted unaffected between the selected master controller and the slave controller. The backward component, of the non-selected inputs to the minimum selector are computed from the forward components of the selected input in such a way that integrator wind-up is managed, even in noisy cases. The inputs to the minimum selector are computed from the backward components of the slave controller output in order to handle bumpless transfer when the slave controller goes from manual to automatic mode.

In the forward direction, the range of measurement of the minimum selector output is computed as the union of the ranges of the minimum selector inputs. In the backward direction, the range of measurement of the output is transmitted to all the inputs.

Common Range Example

A CommonRangeCC control module is used to ensure that two valves in parallel together give the required flow. The signal is divided into two output signals that each transmit part of the input signal. The two output signals also work together in the following way. If one is in Manual mode, the other output signal takes over and compensates its output signal so that the sum of the output signals becomes equal to the input signal. The ratio between the output signals may be selected from the interaction window, or, from an input of ControlConnection type. This allows the output signal's quotient to be controlled according to the example in the figure below.



Split Range Examples

Example 1

Figure 109 describes the conditions in a SplitRangeCC control module when the input value giving the maximum output 1 is greater than the input value giving minimum output 1, that is imax1 > imin1. The input value giving maximum output 2 is also greater than the input value giving minimum output 2, that is imax2 > imin2.

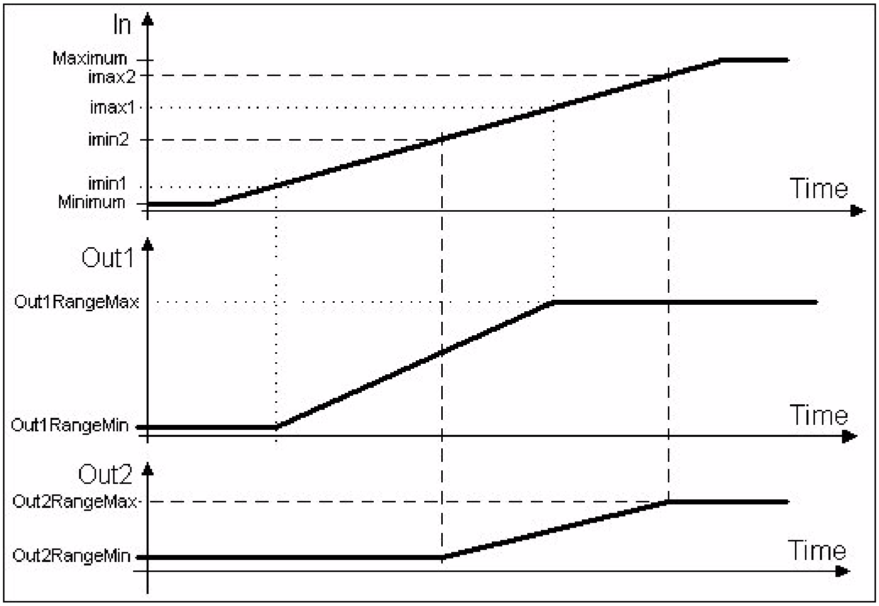


Figure 109. Split range

Example 2

An example of a case in which the SplitRangeCC control module may be used is heating or cooling of a material flow. In the setup illustrated below in Figure 110, there are two valves, one for the cooling medium and one for the heating medium. If both valves are closed, there is no need to change the temperature. A low-level input signal means cooling and a high-level signal means heating.

In the temperature control case, imin1 and imin2 may be equal so that heating takes over when cooling stops.

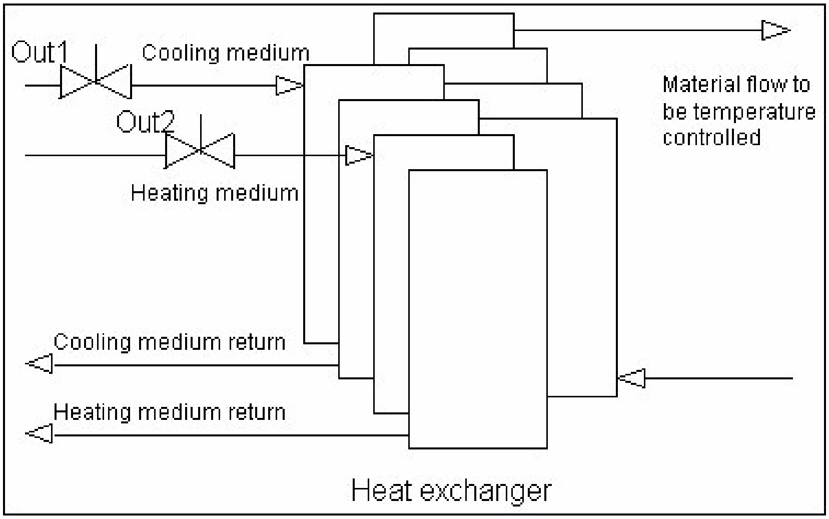


Figure 110.

In applications such as this, the input value giving the maximum value of output 1 is smaller than the input value giving the minimum value of output 1, that is imax1 < imin1. When the input value giving maximum output 2 is greater than the input value giving minimum output 2, that is imax2 > imin2, the conditions are as shown in Figure 111. However, imin1 and imin2 are not equal, as suggested above. The figure shows possible overlapping valve openings.

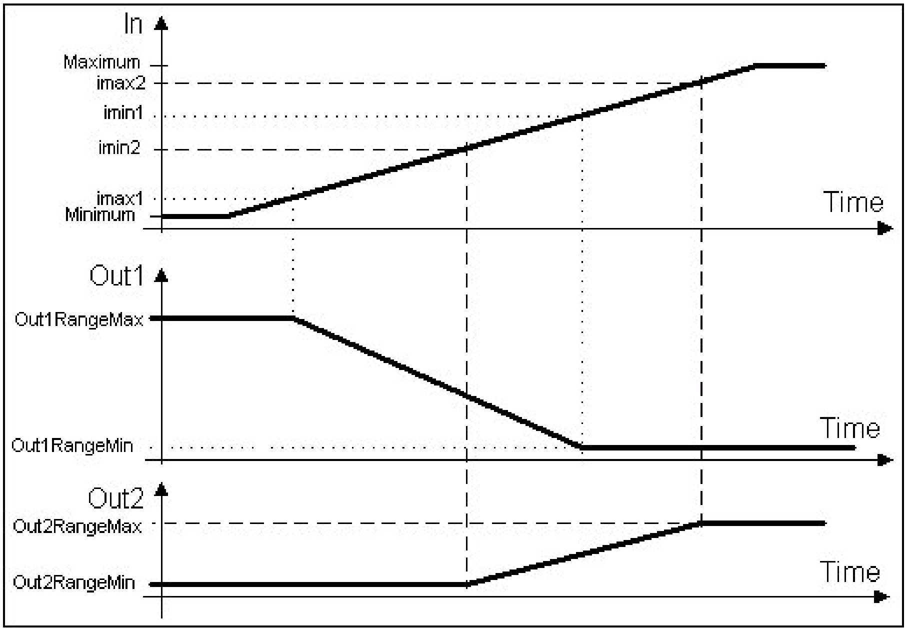


Figure 111.

Level Detection Example

A level detection example with a Level4CC control module type is described in the following figure. The alarm conditions are shown at the various HH, H and L, LL levels, with their common hysteresis, of the In minus (-) the Reference signal. FilterTime is the common time-delay for the alarms. The two HHH and LLL levels are not shown.

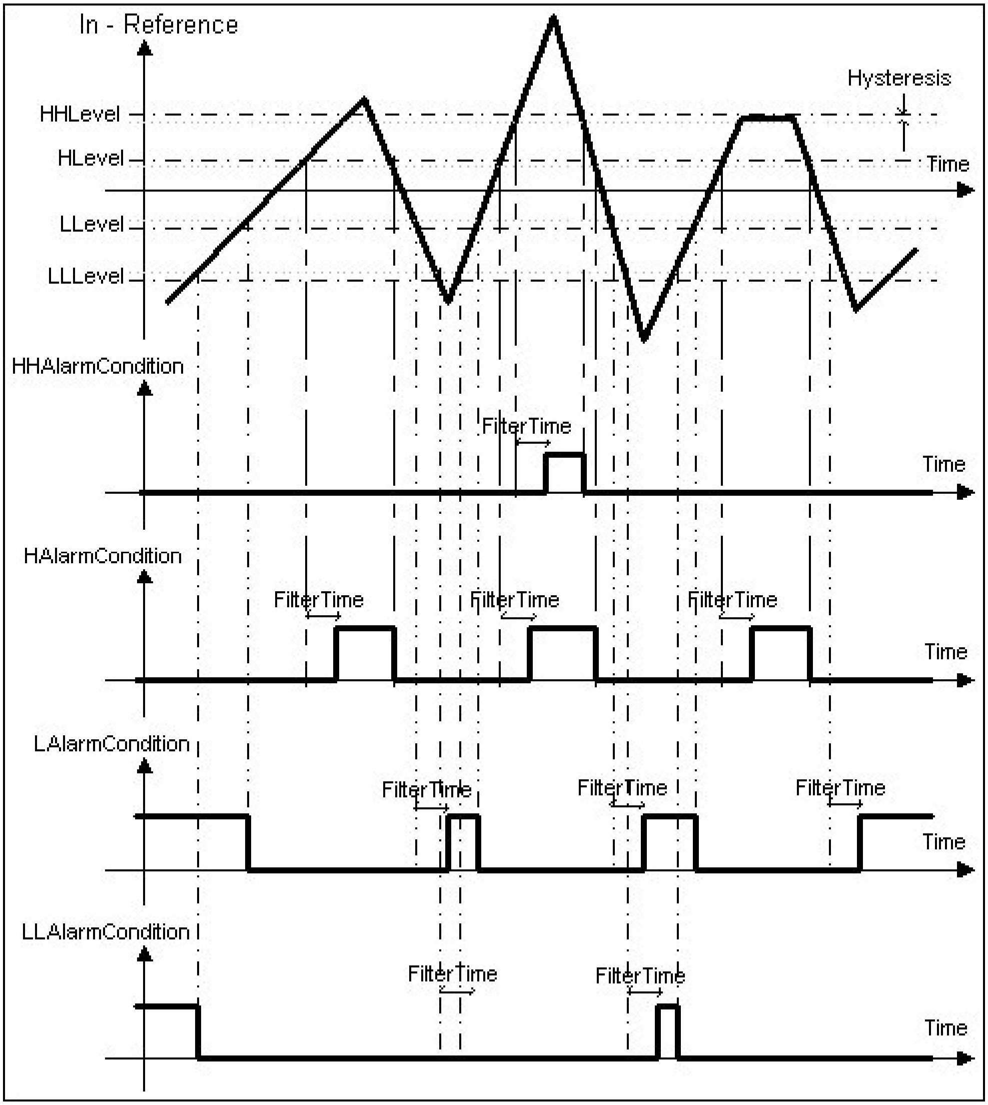


Figure 112. Level Detection example

Application Examples

The examples in this section are installed with your Compact Control Builder. They are located in the ControlExamples folder which is installed in the *Examples* folder.

For more information on how to open the examples, see the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* .

The control loop example applications all reside in the *TankControl* project, which is created when the ControlExamples file is imported. Each application is implemented in four ways, using:

- control modules
- function blocks
- control modules in a function block
- a control loop function block.

The variables of each application are connected to simulated processes, with I/O units according to Figure 113.

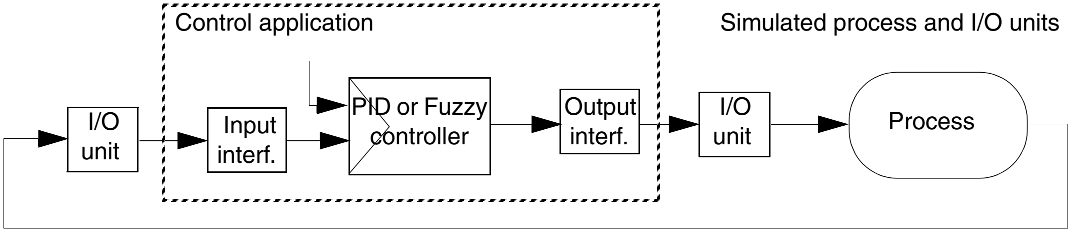
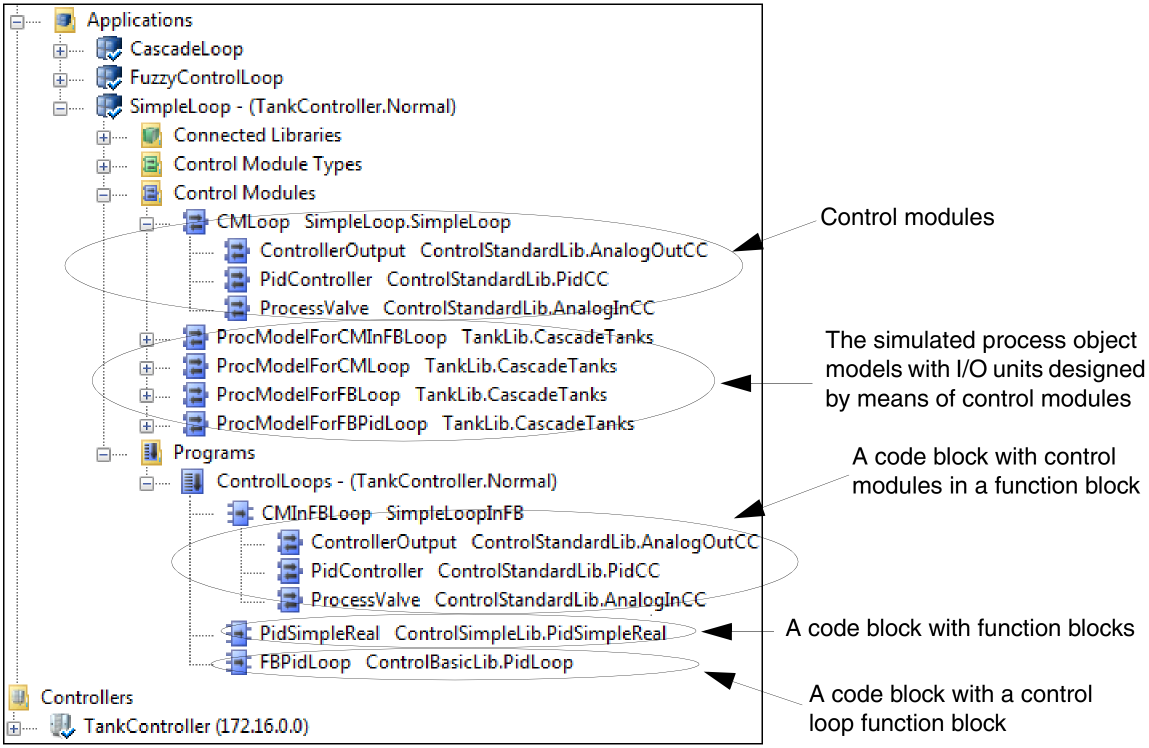


Figure 113. The principle for control applications, the simulated processes and I/O units

The first application example consists of basic, simple control loops. The second example is based on the simple example but expanded to include a cascade controller. The third application example has fuzzy controllers. See Figure 114.

The processes used in the examples are simulated process models, all designed by means of control modules. The process object models, such as the tanks and valves, have their own interaction windows. The advanced user may then alter these objects and see the resulting behavior of the control loop.



```
Applications
    CascadeLoop
    FuzzyControlLoop
    SimpleLoop - (TankController.Normal)
        Connected Libraries
        Control Module Types
        Control Modules
            CMLoop   SimpleLoop.SimpleLoop
                ControllerOutput   ControlStandardLib.AnalogOutCC
                PidController   ControlStandardLib.PidCC
                ProcessValve   ControlStandardLib.AnalogInCC
            ProcModelForCMInFBLoop   TankLib.CascadeTanks
            ProcModelForCMLoop   TankLib.CascadeTanks
            ProcModelForFBLoop   TankLib.CascadeTanks
            ProcModelForFBPidLoop   TankLib.CascadeTanks
        Programs
            ControlLoops - (TankController.Normal)
                CMInFBLoop   SimpleLoopInFB
                    ControllerOutput   ControlStandardLib.AnalogOutCC
                    PidController   ControlStandardLib.PidCC
                    ProcessValve   ControlStandardLib.AnalogInCC
                PidSimpleReal   ControlSimpleLib.PidSimpleReal
                FBPidLoop   ControlBasicLib.PidLoop
Controllers
    TankController (172.16.0.0)
```

Control modules

The simulated process object models with I/O units designed by means of control modules

A code block with control modules in a function block

A code block with function blocks

A code block with a control loop function block

Figure 114. The four methods of designing an application and the simulated process

The purpose of these control loop examples is to show the functions of the function blocks and control modules from the Control libraries, and their use. It is assumed that the user has basic knowledge about the system, Project Explorer and application graphics, as well as general knowledge about process control systems.

The purpose is also to show the user how to use controller interaction windows, perform tuning and parameter settings, apply and view the result of disturbances, etc. The examples will increase your understanding of the process dynamics. Run the *TankControl* project and gradually find and explore the features on your own.

If you want to construct a similar control system application yourself, start from these control examples. You may, in offline mode, also insert any of the application examples into your own project. Alternatively, you can copy some example objects, with their connections, to your own application, from an inserted application example.

What You Can Do?

Start the examples in offline mode, run them in test or online mode, and then perform the following operations:

- Open the interaction windows for the controllers from the Project Explorer.
- Change their internal or external setpoints.
- Go to manual mode and control the process manually.
- Tune the controllers with the Autotuner, or manually.
- Change other controller parameters.
- Change the I/O range or force an I/O signal.
- Open the interaction windows for the process objects from the application windows, and change their characteristics.

Start the Examples

This section describes the control examples in the *Examples* folder, which is installed with the Compact Control Builder.

Carry out the following steps to startup the examples:

- Begin with the Simple Loop example, and then go to the Cascade Loop, followed by the Fuzzy Control Loop.
- If you want to run the project in a controller, you have to select the *TankController* controller and enter *System Identity* as the identity of the controller. In this example, an AC 800M controller with a PM860/TP830 CPU has been selected. You have to enter the same System Identity that you used for the controller as the IP address for *Ethernet*. You should then save the *TankController* controller.

Simple Loop Examples

These first Simple Loop examples contain a process model tank, as shown in the figure below, connected to a level PID controller in the Project Explorer. They are also the basis for the subsequent examples. The task is to keep the level in the upper tank constant.

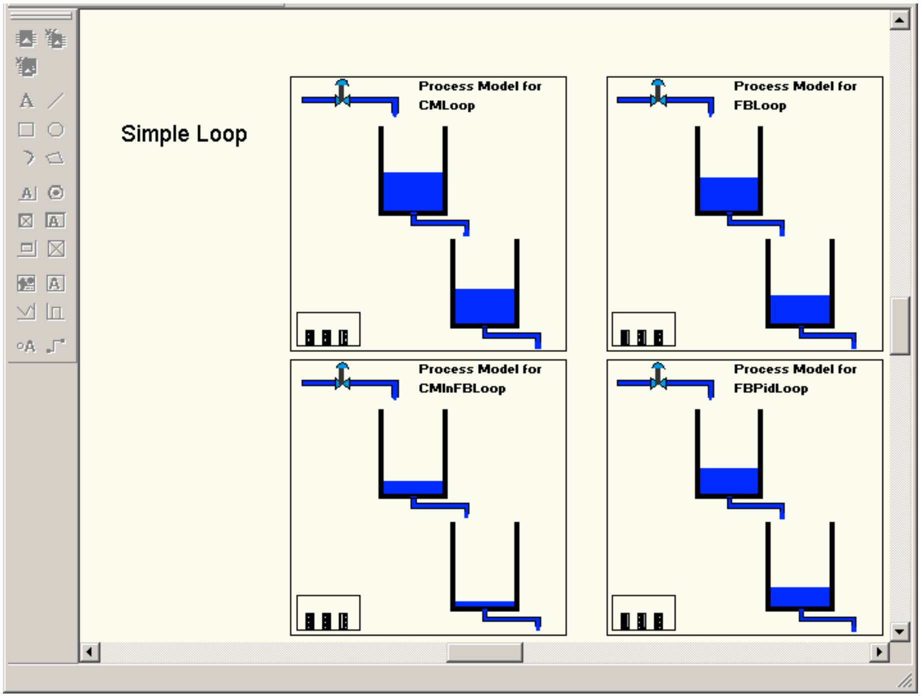


Figure 115. The Simple Loop application in online mode with its four methods of programming

Tank Process

The example consists of two tanks. The upper tank has an inlet pipe. The flow in the inlet pipe is controlled by a control valve. The outlet from the upper tank is a free drain outlet to another lower tank, also with a free drain outlet.

The tank process is created, in each of the four methods, with control modules from the TankLib library; one control module for each tank and one for the valve, which can be seen in the Project Explorer. In the application interaction window, you have a view of the process.

The input and output of the simulation model are of *real* data type. Because the input and output control signals to and from the control loop are of *RealIO* type they are connected via control modules from the TankLib to simulate the I/O system. You can see these control module types, AnalogInIOSim and AnalogOutIOSim, in the lower left hand corners of the application window. In offline mode, you can view their connections. In test and online mode, you can view them in the interaction windows.

Simple Loop with Control Modules

In Test or Online mode in the Project Explorer in the **Applications > SimpleLoop > Control Modules > CMLoop** control module, open the interaction window of the PIDController control module for exploration.

Simple Loop with Function Blocks

In Test or Online mode in the Project Explorer in **Applications > SimpleLoop > Programs > ControlLoops**, open the interaction window of the PIDSimpleReal function block for exploration.

Simple Loop with Control Modules in Function Blocks

In Test or Online mode in the Project Explorer in **Applications > SimpleLoop > Programs > ControlLoops**, open the interaction window of the CMInFBLoop function block. CMInFBLoop contains the CMLoopInFB control module which contains similar control modules to CMLoop above. Open the interaction window of the PIDController control module for exploration.

Simple Loop with a Function Block Loop

In Test or Online mode in the Project Explorer in **Applications > SimpleLoop > Programs > ControlLoops**, open the interaction window of the FBPidLoop function block, which is a ready-made complete control loop, for exploration.

Cascade Loop Examples

The Cascade Loop examples consist of two process model tanks, according to the figure below, connected to two level PID controllers in cascade in the Project Explorer. The examples are based on the Simple Loop example above. The task is to keep the level in the lower tank constant.

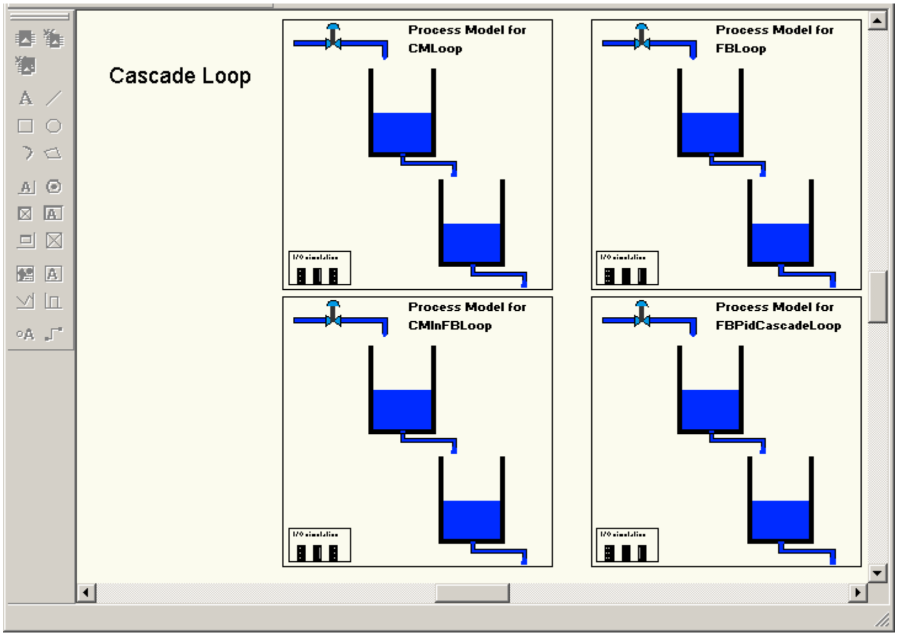


Figure 116. The Cascade Loop application in online mode showing four methods of programming

Tuning of Controllers Connected in Cascade

A cascade controller consists of a combination of input and output signals and two function blocks or control modules, connected in cascade. The output of one controller, called the master, is connected to the external setpoint of the other controller, called the slave.

Two controllers connected in cascade, according to the following figure, must be tuned in the correct sequence.

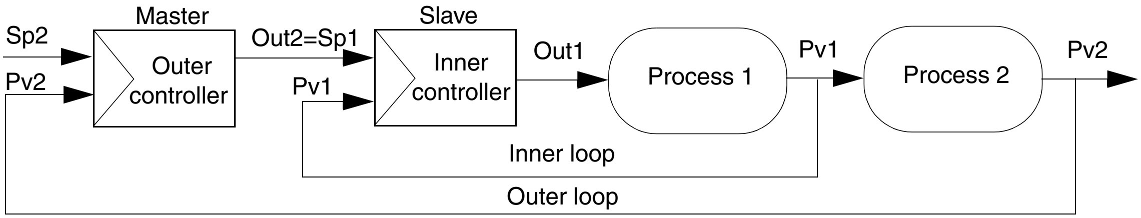


Figure 117. Illustration of two PID function blocks or controller modules connected in cascade

The inner loop should be faster than the outer loop. It should also be possible to have high gain in the inner loop. Starting from scratch, perform the tuning of two controllers in cascade according to the following basic steps.

1. Put both controllers in Manual mode.
2. Begin with the inner controller and adjust the internal setpoint, Sp1, to the inner process value, Pv1.
3. Start the Autotuner of the inner controller, and accept the tuned PID parameters. See Autotuning on page 162.
4. Select the external setpoint for the inner controller and set it to Auto mode.
5. In the outer controller adjust the setpoint, Sp2, to the outer process value, Pv2.
6. Start the Autotuner of the outer controller and accept the tuned PID parameters. See Autotuning on page 162.
7. Switch the outer controller to Auto mode.

Cascade Loop with Control Modules

In Test or Online mode in the Project Explorer in the **Applications > CascadeLoop > Control Modules > CMLoop** control module, open the interaction windows of the SlaveController and MasterController control modules for exploration.

Cascade Loop with Function Blocks

In Test or Online mode in the Project Explorer in **Applications > CascadeLoop > Programs > ControlLoops**, open the interaction windows of the PIDSimpleMaster and PIDSimpleSlave function blocks for exploration.

Cascade Loop with Control Modules in Function Blocks

In Test or Online mode in the Project Explorer in **Applications > CascadeLoop > Programs > ControlLoops**, open the interaction window of the CMInFBLoop function block. CMInFBLoop contains the CMLoopInFB control module, which contains similar control modules to CMLoop above. Open the interaction windows of the SlaveController and MasterController control modules for exploration.

Cascade Loop with a Function Block Loop

In Test or Online mode in the Project Explorer in **Applications > CascadeLoop > Programs > ControlLoops**, open the interaction window of the FBPidCascadeLoop function block, which is a ready-made complete control loop, for exploration.

Fuzzy Control Loop Examples

The Fuzzy Control Loop examples contain a tank with a level controller. The task is to keep the level in the upper tank constant by means of a fuzzy controller according to the FuzzyControlLoop application, as follows. It is implemented in two ways:

- with control modules,
- with control modules in a function block.

These two examples are identical to the Simple Loop examples, except that the PID controller in the control loop has been replaced by a fuzzy controller. It should, however, be emphasized that the tank control loop is not a good example of a loop where a fuzzy controller should be used. In this case, a PID controller works well and should be used instead of a fuzzy controller.

Despite this, we will use the tank control application to demonstrate the fuzzy controller. The reason is that the loop is simple to control and well known to most control engineers. It also provides a means of to comparing the fuzzy controller to the PID controller.

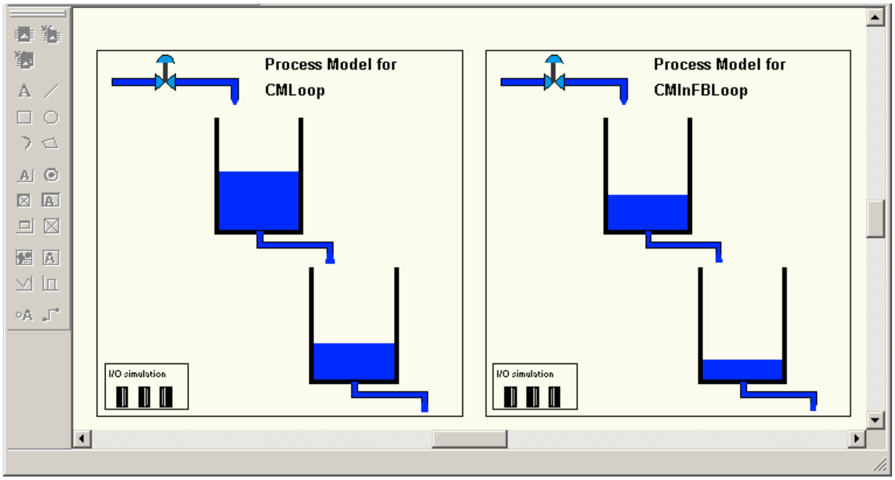


Figure 118. The Fuzzy Control Loop application in online mode showing two methods of programming

Fuzzy Control Loop with Control Modules

In Test or Online mode in the Project Explorer in the **Applications > FuzzyControlLoop > Control Modules > CMLoop** control module, open the interaction window of the FuzzyController control module for exploration. Start by putting the fuzzy controller in Auto mode and enter a setpoint value, for example, setpoint = 4.

Fuzzy Control Loop with a Control Module in the Function Block

In Test or Online mode in the Project Explorer in **Applications > FuzzyControlLoop > Programs > ControlLoops**, open the interaction window of the CMInFBLoop function block. CMInFBLoop contains the CMLoopInFB control module which contains similar control modules as CMLoop above. Open the interaction window of the FuzzyController control module for exploration.

Section 5 Binary Process Control

Introduction

This section describes how to use types from the Process Object libraries to create binary control solutions for your automation system. The section contains:

- A description of the Process Object concept that will help you understand the thinking behind the library types in the Process Object libraries, see Concept on page 254.
- Advice and instructions on how to implement binary control solutions using the types in the Process Object libraries, see Design on page 286.
- Examples on how to implement binary control solutions using the types in the Process Object libraries, see Examples on page 304.
- Detailed information on how to implement ABB Drives and INSUM control, see Advanced Functions on page 317.

All other Uni and Bi types that are based on the UniCore and BiCore types exist in a function block and a control module version. Throughout this section, the notation UniSimple(M) etc. is used when referring to both the function block type and the control module type.

For a discussion on the difference between function blocks and control modules, and how to choose between the two, see the *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* manual.

Concept

A process object is a representation of a real physical object, which can be in different states and can be controlled by commands. Throughout this section, process objects are usually motor or valve objects, but they can also represent other objects, such as a tank.

With process objects, there is a need for scalability, down to single process objects. However, they must never be too specific or too complex, they must be so small and fit so smoothly that they do not interfere with the programmer’s own valve or motor parameters, but at the same time, they must be intelligent enough to work directly by themselves, without any additional programming efforts.

In order to satisfy this need, the Process Object libraries contain a number of types that are designed to be either:

- Used in the application as is, in which case you only have to connect it to the application, to interaction windows, etc., see Process Object Libraries Overview on page 255.
- Used as a template, when new valve object types or motor object types are to be created in your libraries and applications, see Process Object Template Concept (Core Objects) on page 262.

When using process objects, as is or as templates, you will also need to know:

- How to use the basic parameters of the core objects to configure the different types in the Process Object library types, see Core Object Functions and Parameters (UniCore and BiCore) on page 266.
- Which graphics and icons that are used to represent control modules, Control Module Icons on page 283.
- How the interaction windows that can be used in Control Builder work, see Interaction Windows on page 284.
- How communication with graphics and code works, see Interaction Parameters on page 285.

Process Object Libraries Overview

The Process Object libraries provide function blocks and control modules to define valve and motor objects, as well as objects to control and supervise ABB Drives, standard INSUM (Integrated system for user optimized motor management) MCUs (Motor Control Units) and trip units for circuit breakers. The Process Object libraries are standard AC 800M libraries that are installed with the Compact Control Builder.

A schematic overview of the relations between a function block or control module and the remote objects can be seen in Figure 120.

Some of the types in these libraries are not protected, which means that you can copy any type to your library, and then modify the unprotected code in the type and use it.

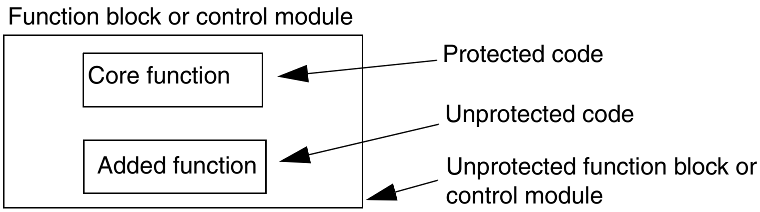

Function block or control module
Core function
Added function
Protected code
Unprotected code
Unprotected function block or control module

Figure 119. The function blocks and the control modules in this library are unprotected, they can be used and modified in your own library

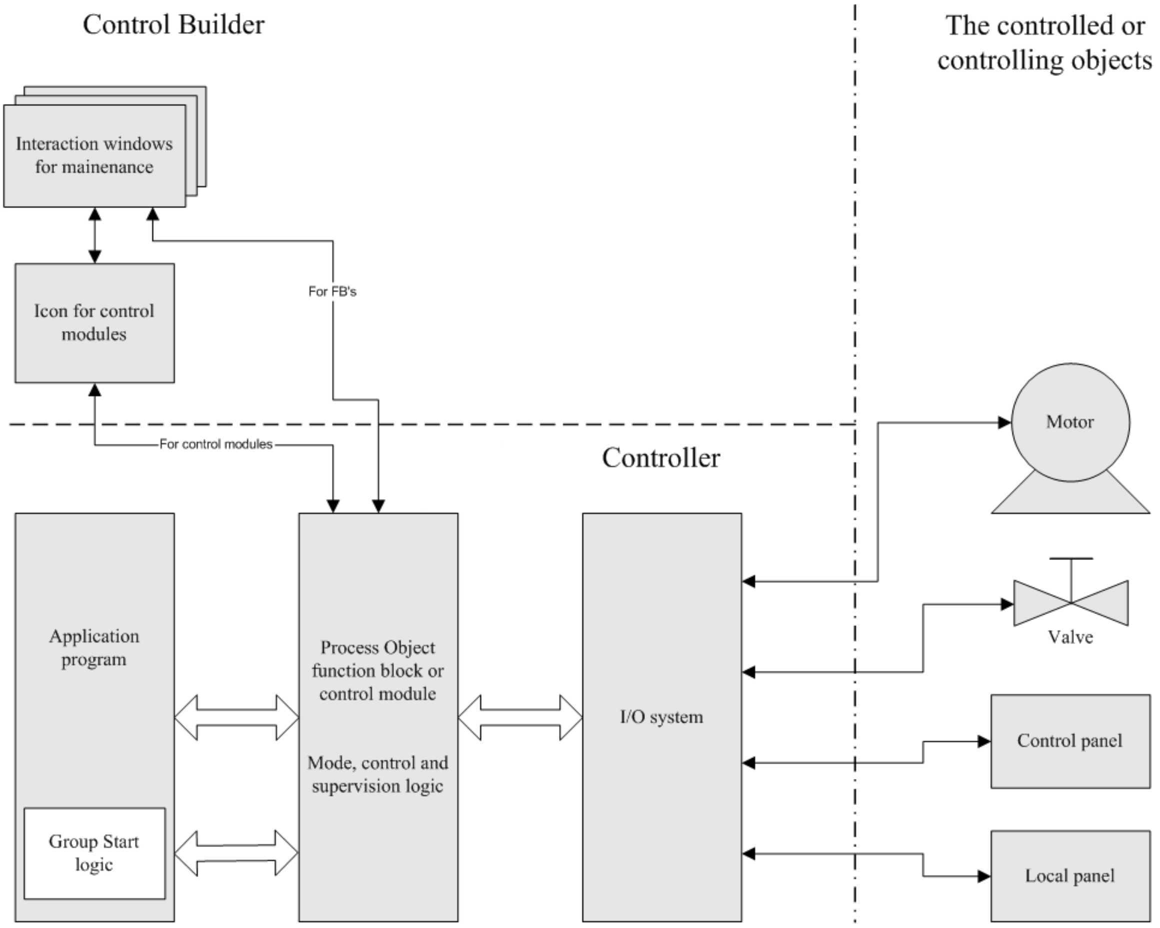


Figure 120. Relations between a function block or control module and the remote objects

There are four different Process Object libraries:

- Process Object Basic library,
- Process Object Extended library,
- Process Object Drive library,
- Process Object INSUM library.

For a complete list of all objects in the Process Object libraries, see Process Object Libraries on page 488. For information on parameters and detailed instructions on how to configure individual types, see Process Object libraries online help.

Process Object Basic Library

The Process Object Basic library contains template (core) objects for building your own process objects. They are intended to be embedded in application-specific types, see Process Object Template Concept (Core Objects) on page 262. It includes additional function blocks for high-level graphical configuration, which are integrated with both Control Builder Professional and faceplates in the Plant Explorer. This library also contains function block types for control of ABB Drives, objects for delaying commands in auto mode, and priority handling as following:

- UniCore: Uni directional core object can be used to control a generic process object through a number of predefined outputs and feedback signals (inputs). UniCore means that the process object can be either activated or deactivated, for example, a motor that can run in one direction (Unidirectional) or it can be stopped.
- BiCore: Bi directional core object can be used to control a generic process object through a number of predefined outputs and feedback signals (inputs). BiCore means that the process object can be in either of two activated positions or it can be deactivated, for example, a motor that can run either forwards or backwards (bidirectional) or be stopped.
- UniSimple and UniSimpleM: Uni directional objects are suitable for HMI control and supervision of a unidirectional (one activated and one deactivated position) process object. UniSimple extensions to the basic UniCore type comprises the graphics functionality for function block. UniSimpleM

extensions to the basic UniCore type comprises control module graphics and interaction windows.

- BiSimple and BiSimpleM: Bi directional objects are suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) process object. BiSimple extensions to the basic BiCore type comprises alarm and graphics functionality. BiSimpleM extensions to the basic BiCore type comprise control module graphics and interaction windows.
- DriveStatusReceive and DriveCommandSend: The Drive functionality is divided into two function blocks, one to receive data from the drive and one to send data to the drive. These can be used as a base for control of ABB Drive, ACS800, ACS600 & ACS400 and their corresponding DC drives.
- IED Command Send: IEDCommandSend functionality has two modes of operation.
  - Direct Mode - Is active when Mode is 1

    In Direct mode, the function block reads the Open or Close command input from IEC 61131-3 application logic and operates irrespective of selection status feedback (SelStatus). The status of the feedback is monitored and compared to the actual output commands. Mismatch is generated as Boolean outputs (CloseFailed/OpenFailed) after the configured time limit expires (PosTimeout).
  - SBO Control Mode - Is active when Mode is 2

    In SBO Control mode, the function block generates Select and Cancel or Operate on Open or Close input command to confirm the selection of IED device. On command input, the select output pulse is generated and the status is monitored. Select timeout output is generated after a time delay (SelTimeout) on failure of select status feedback, then the Cancel command input generates the Cancel output pulse, else the Operate output pulse is generated. On generation of Operate output, the corresponding Open or Close output is generated based on Open/Close command input. The status of the feedback is monitored and compared to the output commands. Mismatch is generated as Boolean outputs (CloseFailed/OpenFailed) after the configured time limit expires (PosTimeout).

- IED Status Receive: IEDStatusReceive function block receives status information from IED logical node through I/O channel and delivers the actual feedback status FB0 and FB1 with respect to input status (PosStatus). The quality status of the input (PosStatus) is copied into outputs FB0 and FB1.
  - Open feedback status (FB0) is set when PosStatus =1
  - Close feedback status (FB1) is set when PosStatus =2
- UniDelayOfCmd: Delays the start and stop command in auto mode. This is a kind of filter to avoid undesired start and stop of the object. This object also contains a selection whether the command shall be edge or level detected.
- BiDelayOfCmd: Functions as an UniDelayOfCmd but extended with another start signal. This object also contains a selection whether the command shall be edge or level detected.
- PrioritySup: Indicates Priority mode, supervise the Priority command inputs and takes care of automatic object stop when errors indications are present.

Process Object Extended Library

The Extended library is more comprehensive than the Basic library. It comprises ready-to-use valve and motor objects, alarm handling functionality and templates for user-designed valve and motor objects. It includes additional control modules for high-level graphical configuration, which are integrated with Compact Control Builder.

- Uni: The function block type are suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) process object. Extensions to the basic UniCore type is alarm and graphics functionality.
- Bi: The function block types is suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) process object. Extensions to the basic BiCore type is alarm and graphics functionality.

The types in this library also have a group start interface, which can be used to connect them to a group start configuration.

All the control module types in Process Extended library has a vote parameter (*VoteOut and VotedCmd*) that is used for connections to vote control module types (see Vote Control Module Types on page 381).

- MotorUni: The function block type is suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) motor. It is based on UniCore and extended with alarm and interaction windows.
- MotorUniM: The control module type is suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) motor. It is based on UniCore and extended with alarm and interaction windows.
- MotorBi: The function block type MotorBi is suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) motor. It is based on BiCore and extended with alarm and interaction windows.
- MotorBiM: The control module type is suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) motor. It is based on BiCore and extended with alarm and interaction windows.
- ValveUni: The function block type is suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) valve. It is based on UniCore and extended with alarm and interaction windows.
- ValveUniM: The control module type is suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) valve. It is based on UniCore and extended with alarm and interaction windows.
- MotorValve: This function block type is suitable for graphical control and supervision of a bidirectional (two activated position) motor valve. It is based on BiCore and is extended with alarm and interaction windows.
- MotorValveM: This control module type is suitable for graphical control and supervision of a bidirectional (two activated position) motor valve. It is based on BiCore and extended with alarm and interaction windows.
- MotorValveCC: This control module type is suitable for graphical control and supervision of a motor controlled valve of open/close type. It is based on BiCore and is extended with alarm and interaction windows. The input interface is of type control connection that can be connected to an output object from a control loop

Process Object Drive Library

The Process Object Drive library contains function block and control module types for controlling and supervising ABB Drives via ModuleBus, DriveBus, or PROFIBUS-DP. Drives process objects are based on UniCore. Communication is based on the DriveStatusReceive and DriveCommandSend function blocks, which are included in all Drives process objects.

All the control module types in Process Object Drive library has a vote parameter (*VoteOut and VotedCmd*) that is used for connections to vote control module types (see Vote Control Module Types on page 381).

- DriveStatusReceive and DriveCommandSend: The two function blocks gives the user the opportunity to add user code with input from the DriveStatusReceive block and with output to the DriveCommandSend block without delays.

Process Object INSUM Library

The ProcessObjInsumLib library contains Function Blocks and Control Modules to control and supervise the standard INSUM (Integrated system for user optimized motor management) devices MCU (Motor Control Units) and trip unit for Circuit Breakers. INSUM process objects are based on BiCore. Communication is based on the INSUMWrite and INSUMRead function blocks from InsumCommLib, which are included in all INSUM process objects.

All the control module types in Process Object INSUM library has a vote parameter (*VoteOut and VotedCmd*) that is used for connections to vote control module types (see Vote Control Module Types on page 381).

- McuBasic, based on the BiCore, INSUMWrite and INSUMRead.
- McuExtended, based on the same Function Blocks as McuBasic and also extended with alarms.
- InsumBreaker, based on UniCore, INSUMWrite and INSUMRead Function Blocks and extended with alarms for trips, warnings and other errors like communication errors and feedback errors.
- McuBasicM, Control Module with the same functionality as the corresponding Function Block, McuBasic.

- McuExtendedM, Control Module with the same functionality as the corresponding Function Block, McuExtended.
- InsumBreakerM, Control Module with the same functionality as the corresponding Function Block, InsumBreaker.

INSUM devices are connected via an INSUM Gateway and a CI857 hardware unit to the AC 800M. All objects include interaction windows for Compact Control Builder.

The Process Object INSUM library also contains a number of predefined data types, used for communication to the interaction windows.

Process Object Template Concept (Core Objects)

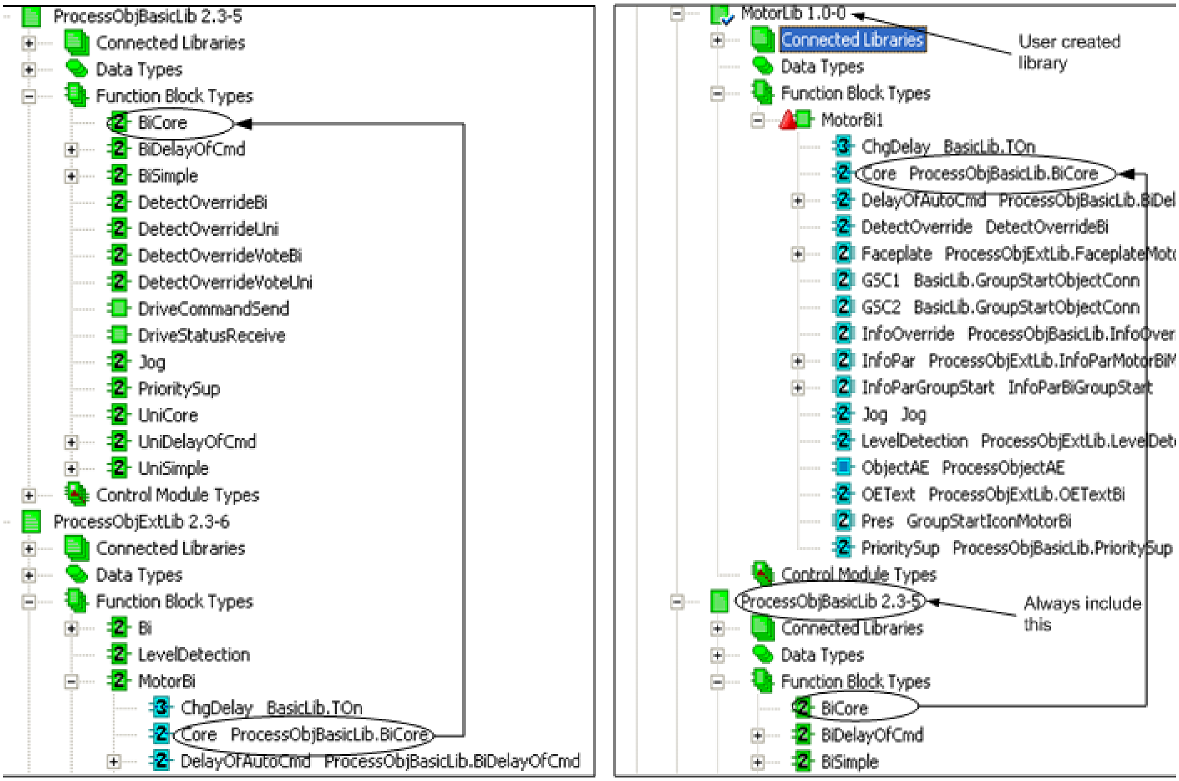
When using types from the Process Object libraries, it is important to understand the *template* concept. When working with process objects, there is a need for a general function block base, a *Core function* that fits all valve and motor objects in all industrial control applications. Since all valve and motor objects need at least uni- or bi-directional control, objects with these properties make up the core of all types in the Process Object libraries.

The smallest common denominator is:

- The UniCore function block type, for uni-directional process objects.
- The BiCore function block type, for bi-directional process objects.

The above core function blocks (BiCore and UniCore*)* are protected and form a basis for all object types. The core is not copied when a process object type is copied, instead it refers (points) to its type in the Process Object Basic library. This means that then using types that have a core object embedded, this core object is referenced.

Figure 121 shows how core objects form the basis of other, application-specific objects.



```
ProcessObjBasicLib 2.3-5
  Connected Libraries
  Data Types
  Function Block Types
    BiCore
    BiDelayOfCmd
    BiSimple
    DetectOverrideBi
    DetectOverrideUni
    DetectOverrideVoteBi
    DetectOverrideVoteUni
    DriveCommandSend
    DriveStatusReceive
    Jog
    PrioritySup
    UniCore
    UniDelayOfCmd
    UniSimple
  Control Module Types
ProcessObjExtLib 2.3-6
  Connected Libraries
  Data Types
  Function Block Types
    Bi
    LevelDetection
    MotorBi
      ChgDelay   BasicLib.TOn
      Core   ProcessObjBasicLib.BiCore
      DelayOfAutoCmd   ProcessObjBasicLib.BiDelayOfCmd
```

```
MotorLib 1.0-0
  Connected Libraries
  Data Types
  Function Block Types
    MotorBi1
      ChgDelay   BasicLib.TOn
      Core   ProcessObjBasicLib.BiCore
      DelayOfAutoCmd   ProcessObjBasicLib.BiDel
      DetectOverride   DetectOverrideBi
      Faceplate   ProcessObjExtLib.FaceplateMot
      GSC1   BasicLib.GroupStartObjectConn
      GSC2   BasicLib.GroupStartObjectConn
      InfoOverride   ProcessObjBasicLib.InfoOver
      InfoPar   ProcessObjExtLib.InfoParMotorBiM
      InfoParGroupStart   InfoParBiGroupStart
      Jog   Jog
      LevelDetection   ProcessObjExtLib.LevelDete
      ObjectAE   ProcessObjectAE
      OEText   ProcessObjExtLib.OETextBi
      Pres   GroupStartIconMotorBi
      PrioritySup   ProcessObjBasicLib.PrioritySup
  Control Module Types
ProcessObjBasicLib 2.3-5
  Connected Libraries
  Data Types
  Function Block Types
    BiCore
    BiDelayOfCmd
    BiSimple
```

User created library

Always include this

Figure 121. The core functionality in ProcessObjExtLib and in a user's self-defined library

The Process Object Basic library must always be connected to your project when using self-defined types based on types from the Process Object libraries. This is necessary since this library contains data types used that are used in the Process Object Extended library. It is necessary even if parameters of those data types not are explicitly used, due to their DEFAULT declaration.

Core objects can only be updated when the standard libraries are upgraded. If core objects are updated, the core functions of all types based on these core objects are updated as well.

All types based on the core objects exist in two variations, as a function block type and a control module type.

Core Object Properties

All core objects (sub-objects) have protected code. The user can neither see nor access the code, modify it nor change it; instead the function blocks have parameters for providing a means of interaction in the code.

The two basic function blocks, UniCore and BiCore, do not contain any Control Builder graphics. However, they have parameters for interaction with the graphics operator panel.

UniCore or BiCore are used in all process objects and may be combined with some or several of the other above mentioned core objects. The process objects UniSimple(M) and BiSimple(M) are open for the user and may be used as templates and copied to a user library, where modifications may be performed in the code, the Control Builder graphics or used as object types and instantiate in the user application like other closed library objects. The code content of these objects is mostly calls to core objects and the direct code is minimized.

Interaction with core objects uses a structured parameter named *InteractionPar*, containing the information that is written from the graphics. As this parameter also may be written from the user code, care have to be taken not to lock out the graphic interaction possibilities.

Commands to UniCore and BiCore objects are reset inside the core, and process objects shall only set them to activate operation. Examples on commands are switch to auto mode or manual mode, but also the manual operation commands like *ManCmdx*.

Types Based on UniCore and BiCore

There are a number of types that are based on UniCore and BiCore:

- The function block types Uni, Bi, ValveUni, MotorUni, MotorBi, and MotorValve.
- The control module types UniM, BiM, ValveUniM, MotorUniM, MotorBiM, MotorValveM and MotorValveCC.

The Process Object Basic library contains two additional function block types and two control module types that are based on the core objects, but contain fewer functions than the Uni and Bi types:

- UniSimple(M) is based on Uni(M), but does not have any alarm handling.
- BiSimple(M) is based on Bi(M), but does not have any alarm handling.

For information on how to configure simple objects, see Generic Uni- and Bi-Directional Control on page 293 and Process Object library online help.

Some of the Objects also use Uni and BiCore, in Process Object INSUM Library and Process Object Drive Library.

All these types can be used as is, or form the basis for your own types. Objects based on these types will still reference the UniCore or BiCore function block type. This means that they may be affected by upgrades to the Process Object standard libraries.

To avoid upgrading problems, place a copy of the process object type in your own library and then make the changes.

Core Object Functions and Parameters (UniCore and BiCore)

In order to understand how to use the process objects in the Process Object libraries, it is necessary to know how the different parameters interact and how they should be used.

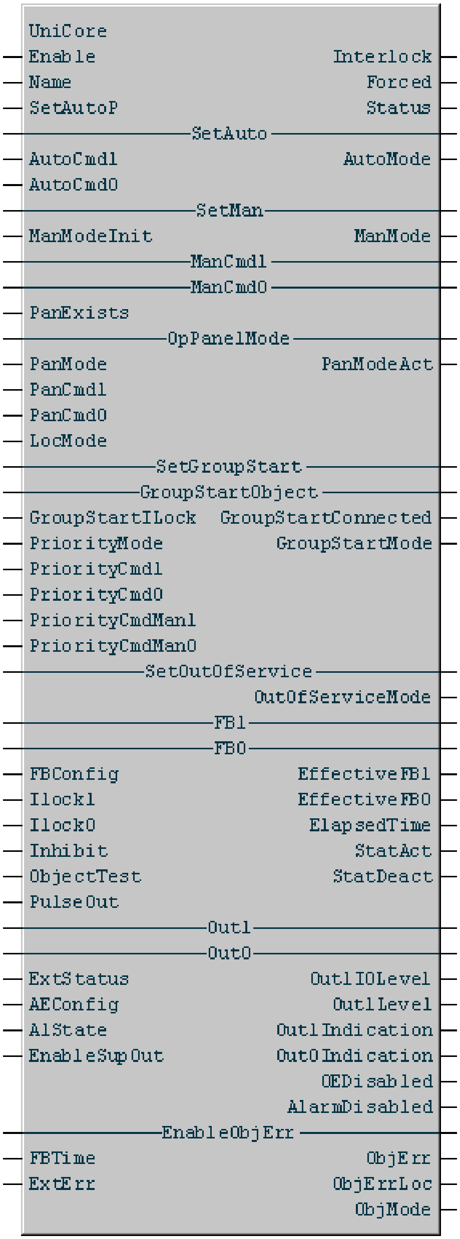
This text is based on the UniCore function block type, but most of the information applies to BiCore as well. At the end, there is a short section which discusses the most important differences between UniCore and BiCore, see Differences Between UniCore and BiCore on page 278.

UniCore can be used to control a generic process object via a number of predefined outputs and feedback signals (inputs). Uni means that the process object can be either activated or deactivated. For a valve (or a motor) this would mean open (running in one direction, uni-directional) or being closed (stopped).

UniCore has pulse output selection and feedback configuration, to allow configuration of the number of feedbacks and inversions.

UniCore parameters are divided into the following types:

- **Operation**
  - Mode,
  - Configuration of feedback signals,
  - Object test.
- **Interlocking**
  - Ilock,
  - Priority commands,
  - Inhibit.
- **Feedback error**
- **Effective feedback**
- **Output IOLevel**



UniCore function block

Operation Parameters

The operation parameters of the UniCore object can be used to configure feedback signals, set feedback error time, and disable error handling. The *PulseOut* parameter governs whether outputs should be pulsed or level, depending on the hardware used for the process object in question. UniCore also calculates information to be presented in interaction windows.

- **Modes**

  An object is activated when the *Enable* parameter is set to true, signifying that the function block will be executed. When the parameter Enable is false, *Out1*, *Out0*, *Out1Level*, *StatAct*, *StatDeact,* and *ObjErr* will be set to false, regardless of the status of other signals. The object can be activated or deactivated in all modes. After deactivation, the object is switched into Manual mode.

  UniCore has seven different operation modes:
  Manual, Auto, Panel, Priority, Group Start, Local and Out of Service Mode.

  Manual mode and Auto mode are examples of output mode indication parameters (*ManMode* and *AutoMode*).

  Manual mode is set as the default start-up mode using the parameter *ManModeInit*. The initial value is set to true, meaning that Manual will be the default mode. The parameter *ManModeInit* is copied to the parameter *ManMode* at every cold start.

  The output parameter *ObjMode* also indicates the different modes:

  ObjMode = 0 -> Local mode

  ObjMode = 1 -> Priority mode

  ObjMode = 2 -> Panel mode

  ObjMode = 3 -> Manual mode

  ObjMode = 4 -> Auto mode

  ObjMode = 5 -> Group start mode

  ObjMode = 6 -> Out of service mode

  Figure 122 shows the different operation modes and their relationships. The longer from the middle the higher the priority.

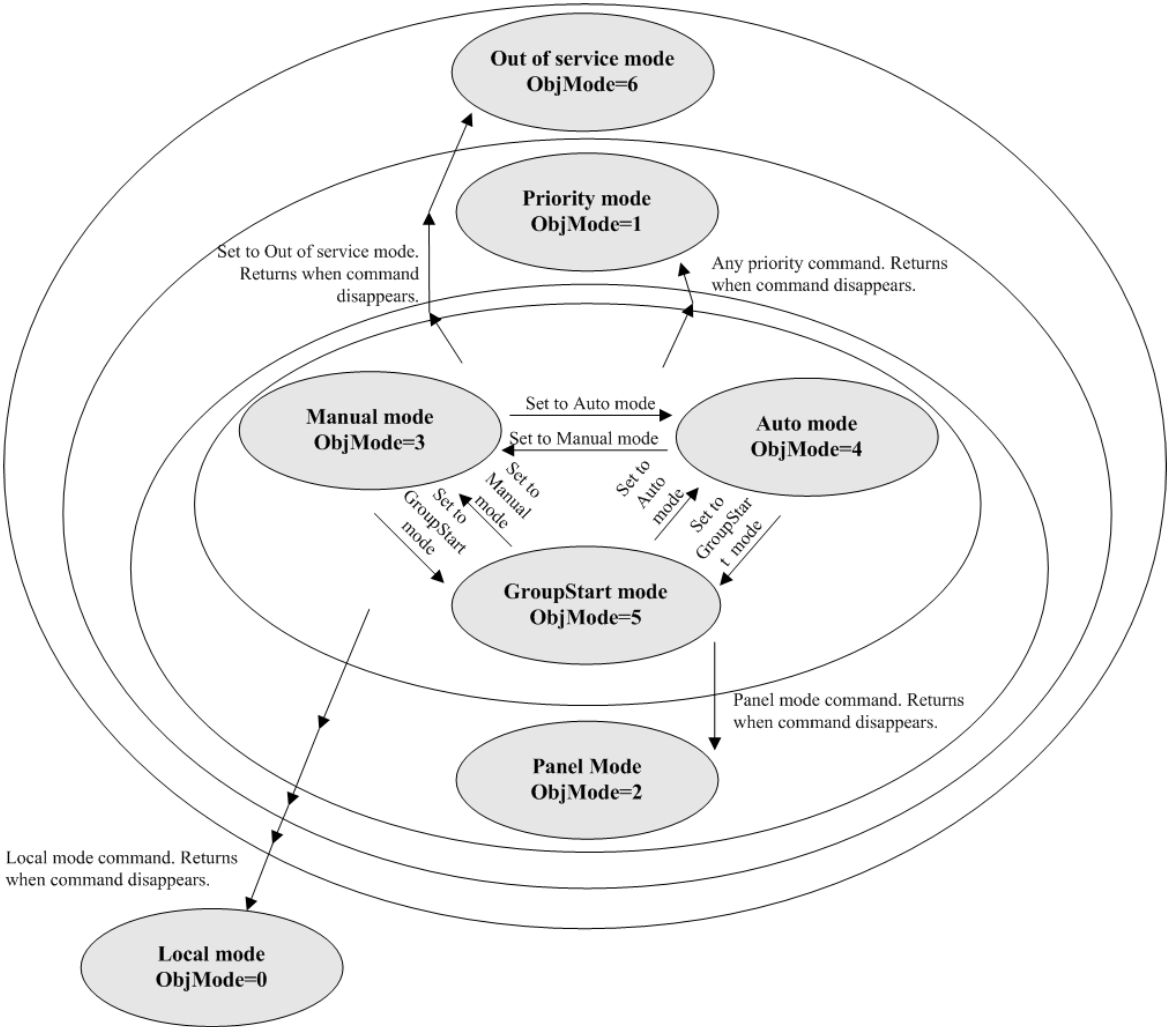


Figure 122. Core object modes and their relationship

– **Manual Mode** - A user operates the object from a workstation

To change to Manual mode, trigger the *SetMan* parameter and the *ManMode* parameter will automatically be set to true. Output signals (*Out1*, *Out0*, *Out1Level*) will retain their status from the previous mode.

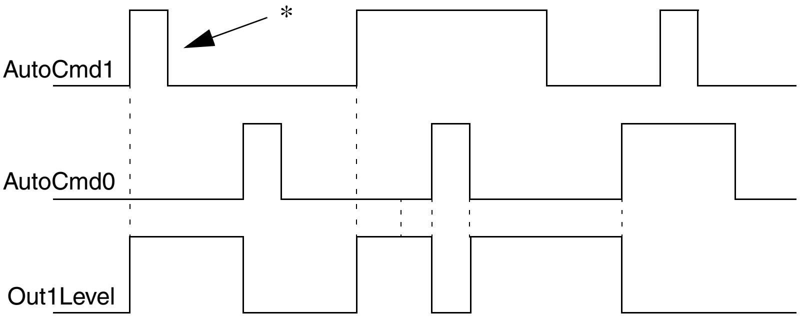
In Manual mode, the status of the output signal is controlled by the parameters *ManCmd1* and *ManCmd0*. These parameters have rising edge detection.

All parameters mentioned above (*SetMan*, *ManCmd0*, *ManCmd1*) are normally connected to the interaction window via the application.

– **Auto Mode** - The program controls the object

Auto mode is set by means of the parameter *SetAuto*. The status of the output signals (*Out1*, *Out0*, *Out1Level*) in Auto mode is controlled by the parameters *AutoCmd1* and *AutoCmd0*. *AutoCmd0* has higher priority than *AutoCmd1*.

The *AutoCmd1* and *AutoCmd0* parameters are level detected, therefore these parameters affect the output signal, as long as they are active. The programmer is required to reset these parameters from the application program outside the function block. These parameters act on *Out1Level* according to Figure 123.



* Reset after function block in application program. *AutoCmd0* has priority.

Figure 123. Status relations for Out1Level and the Auto commands

When returning to Auto mode from another mode, the status of the output signals will be returned from the previous mode, but adjusted directly by the signals *AutoCmd1* or *AutoCmd0,* if one or both are set to true.

– **Panel Mode** - The object is controlled locally from a control panel, via the UniCore function block

  The function block has a set of signals for maneuvering the object from a control panel. Setting the *PanMode* parameter activates Panel mode. The *PanMode* parameter is level detected.

  This mode is active only as long as the *PanMode* parameter is true. If *PanMode* is changed to false, the system exits Panel mode immediately, and return to the previous mode.

  For examples of Panel mode, see Connect to a Control Panel in Panel Mode on page 313.

– **Priority Mode**

  The object is in Priority Mode when any of *PriorityCmd0*, *PriorityCmd1*, *PriorityCmdMan0* and *PriorityCmdMan1* are active. This is described in section Priority and Interlocking Parameters on page 274.

– **Group Start Mode**

  Changing mode to Group start mode is done via the structured parameter *GroupStartIn* of type *GroupStartObject*. In this mode the object is controlled via this parameter. The parameter *GroupStartIlock* prevents the possibility to transfer to Group start mode. In this mode output parameter *GroupStartMode* is active.

– **Local Mode** - The object is controlled locally from a local control panel, bypassing the UniCore function block

  Local mode is used when the object is controlled locally from a local control panel and the function block does not have any signals for controlling the object from the local control panel. All signals from the local control panel are physically connected directly to the object (motors, valves, etc.), see Figure 124.

  In this mode, the object statuses are updated using the feedback signals. Objects return to the previous mode when local mode is disabled.

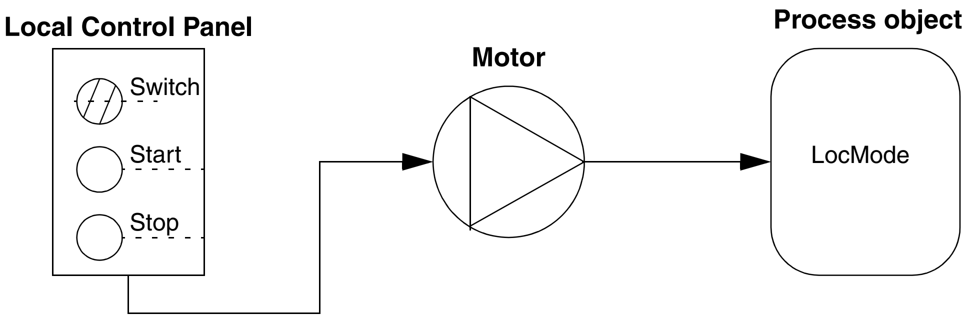


Figure 124. Control steps in Local mode

- **Out Of Service Mode**

  Out of Service mode is only available when the object is stopped. The operator controls the mode and the object cannot be maneuvered. It is possible to transfer to other modes.

  If the command signal *SetOutOfService* is true the mode sets. When this command has been executed the function block resets the command. The object must be in position 0 (stopped, closed) to enter the Out of Service mode.

- **Feedback Signals**

  The parameter *FBConfig* informs the function block of how feedback is configured, by transferring the combination of feedback signals of the object to the function block. Possible combinations of feedback signals are listed in Table 33.

Table 33. Possible combinations of feedback signals

FBConfig	Feedback from activated position	Feedback from deactivated position
0	FB1	FB0
1	FB1	(none)

Table 33. Possible combinations of feedback signals

FBConfig	Feedback from activated position	Feedback from deactivated position
2	FB1 inverted	(none)
3	FB1 inverted	FB0 inverted
4	(none)	FB0
5	(none)	FB0 inverted
6	(none)	(none)

The values of the feedback signals *FB1* and *FB0* are transferred to the *StatAct* and *StatDeact* output parameters respectively, in condition with *Out1Level*.

If an object has no, or only a single, feedback signal, *StatAct* and *StatDeact* will still be set. The signals *StatAct* and *StatDeact* are set to zero if double feedback is used and both are true at the same time.

- **Output Settings**

The output signals include *Out1*, *Out0* and *Out1Level*. *Out1* and *Out0* can be configured as pulsed command or level-detected command, by means of the *PulseOut* parameter.

The pulse is sustained by the parameters *Out1* and *Out0*, until the corresponding feedback is detected or the maximum feedback error time is exceeded. *Out1Level* is intended to indicate the output state when the *Out1* and *Out0* parameters are pulsed. If there is no feedback from a position, the pulse duration is set to the object error time. Only one pulse is generated upon each status change.

- **Object Test**

When the parameter *Object Test* is activated, the feedback error calculation is disabled and the *Out1* and *Out0* signals are set to false. The signals *StatAct* and *StatDeact* are set according to the status of *Out1Level*.

When *Object Test* is deactivated, the status of the object reverts to that in the mode prior to *Object Test* and the *Out0* signal is set to true.

Priority and Interlocking Parameters

Priority and interlocking parameters can be used to control the behavior of process objects in certain situations, for example, to stop an object from entering a certain state or forcing it to a certain state. This might be needed to ensure accurate operation with critical process values, such as high levels and temperatures, Motor Control Center interface, safety devices, etc.

The main difference between interlocking and priority parameters lies in that interlocking parameters stop the object from entering a certain state, while priority parameters are used to force an object to a certain state.

- **Ilock** - Prevents the object from entering a certain state

  The *Ilock1* and *Ilock0* signals block commands in all modes to switch the object to the activated and deactivated state respectively. The *Inhibit* signal will suppress interlock signals, when true.

- **Priority Command** - Compels the object to enter a certain state

  The *PriorityCmd1* and *PriorityCmd0* signals force the output to the respective status with priority over other signals, except when the *Inhibit* signal is set to true. When these signals disappear, the status from Priority state is transferred to the current mode.

- **Priority Manual Command** - The commands *PriorityCmdMan1* and *PriorityCmdMan0* have the same functionality as the Priority Commands with the exception that the object switch into manual mode after the priority man command is completed.

- **Inhibit** - Suppress all *Ilock* and *Priority(Man)Cmd* signals

  When the *Inhibit* signal is set to true, the program will ignore all *Ilock* and *PriorityCmd* signals. This parameter can be used when there is an absolute need for running the object, although all *Ilock* and *PriorityCmd* signals are suppressed. When the *Inhibi*t signal returns to false, the *Ilock* and *PriorityCmd* are re-activated.

  When *Object Test* is de-activated, the status of the object reverts to that in the mode prior to *Object Test* and the *Out0* signal is set to true.

Feedback and Output Parameters

The feedback error (*ObjErr*) is supervised in all modes except when the parameter *Enable* is false and in Local mode, based on the values of the parameters *Out1Level*, *FB1* and *FB0*.

The parameter *ExtErr* provides the possibility of connecting other errors to the object. This parameter does not affect the object status. If required, it must be implemented outside the UniCore, for example, by connecting a variable to both parameters *ExtErr* and *PriorityCmd0*.

The information required for the feedback error calculation is the feedback error time and the number of feedbacks: these two parameters have to be configured by the user (the parameters *FBTime* and *FBConfig*).

FBTime has an initial value of 5 seconds. In the Extended Library an interaction parameter is connected to *FBTime*. This value can be altered online, in the interaction windows (Compact Control Builder). The interaction parameter has a cold retain attribute to retain the value following cold or warm restarts.

ParError is set to True when a parameter takes an illegal value (goes outside the allowed range). This is also indicated by a red triangle in interaction windowsof the parent objects.

EnableParError is set to false by default.

See also ParError on page 43.

The object internal status is updated from the feedback signals after a warm restart:

- **Effective Feedback**

  The *EffectiveFB1* and *EffectiveFB0* parameters, of the *in/out* type, give the calculated result from *FBConfig* and the values of *FB1* and *FB0*.

  Effective feedback signals are connected to the interaction window and are displayed under the Status tab in the parent objects.

- **Output IOLevel**

  The parameter *Out1IOLevel,* of the in/out type, is calculated from the actual values of the binary I/O (*Out1IO*, *Out0IO*), in relation to the selected pulse functionality.

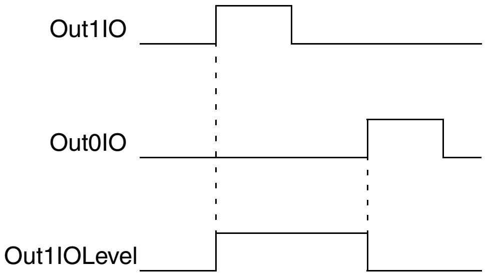


Figure 125. OutputIOLevel state diagram

  The output signals are connected to and displayed in the interaction window.

- **Interlock**

  The parameter *Interlock*, of the in/out type, is the sum of parameters *PriorityCmd* and *Ilock*. The *Interlock* parameter provides information when any type of interlocking is active (for example, *PriorityCmd* or *Ilock*).

  The interlock signals are connected and displayed in the interaction window.

- **Forced Actions**

  The parameter *Forced*, of the in/out type, indicates the forced status of the *Out0IO* and *Out1IO* signals at *FB0* or *FB1*. If one of these four I/O signals is blocked/forced, the parameter *Forced*, of the in/out type, will be true.

Power Fail Recovery

After a power failure, the objects are always set in Manual mode. If the desired behavior is different from this, the code to obtain the demanded state needs to be added.

The parameter *ManModeInit* placed on the UniCore or BiCore functions block instance is only used at a cold start and is therefore not used during power fail recovery. This is also indicated in the parameter description.

During the startup after a power fail, it is during the second scan that the objects are updated to reflect the actual state of the connected feedback signals. This means that the backtracking from output take place during the second scan after power up, and not during the first scan.

For example, during the second scan after power up, a function block is updated to the real state of the connected objects, which means that the feedbacks are read and the internal states of the function block are set according to the read values. The corresponding updates are executed when the quality of the connected I/O channels becomes good from being non good. This is made to keep the states on the connected objects that may have entered any OSP setting during the non good quality period.

I/O Quality Change

Independent of operation mode, the outputs from the objects are updated to reflect the actual state of the connected feedback signals when the quality of the connected I/O channels changes from bad to good.

If bad to good signal quality transfer is not detected, outputs from the objects are set to deactivate/close. Bad I/O quality may not be seen if the I/O remains powered when the CPU is switched off after a CPU power failure detection.

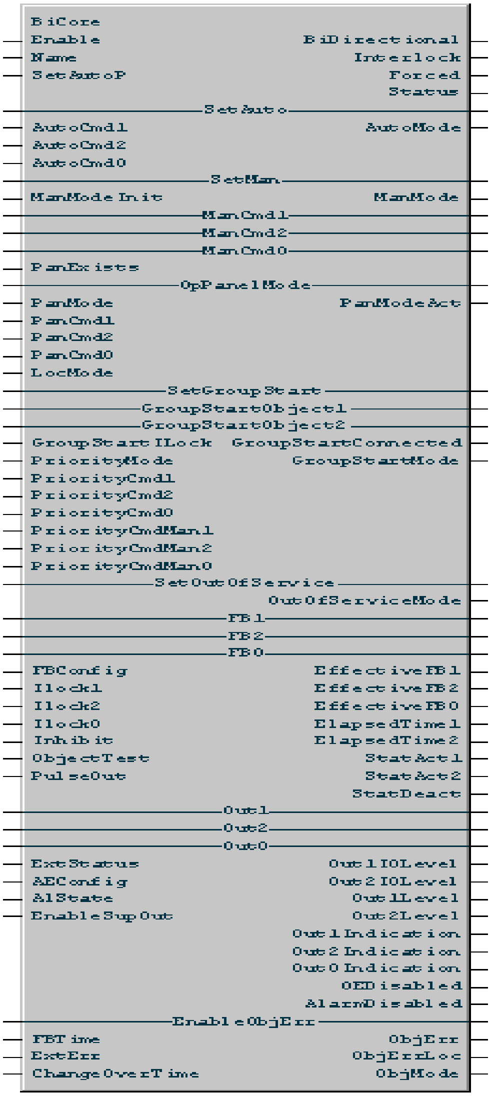
Differences Between UniCore and BiCore

The only difference between UniCore and BiCore is that BiCore is bi-directional. UniCore is designed for control of process objects with two states (such as stopped and running), while BiCore is designed for control of process objects that have three states (such as stop, start/forward and start/back, which correspond to 0, 1, and 2).

Examples of bi-directional applications are two-speed motors and forward/backward motors.

Compared to UniCore, Bi Core has the following extra parameters (all relating to the extra state, hence they all carry the number 2):

- In Manual mode, there are additional parameters *ManCmd2, Out2* and *Out2Level*.
- In Auto mode, there are additional parameters *AutoCmd2*, *Out2* and *Out2Level*. The influence of *AutoCmd* parameters on *OutLevel* parameters can be seen in Figure 126 on next page.
- In Panel mode, there is an additional *PanCmd2* parameter.



BiCore function block

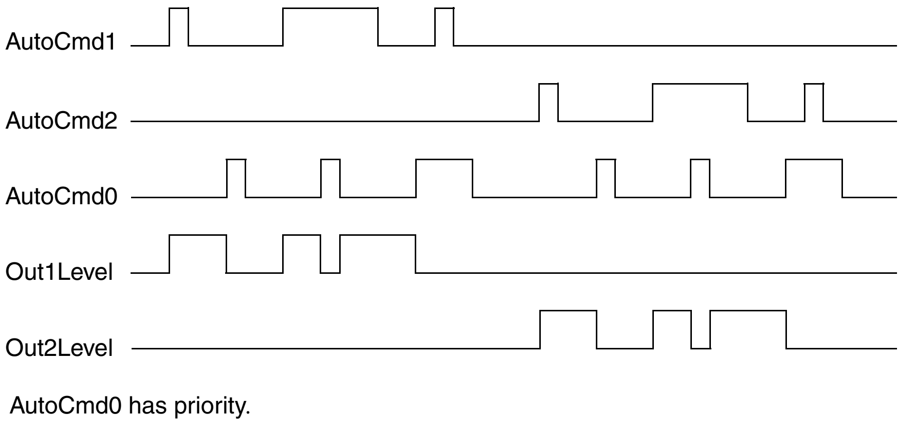


Figure 126. Status relations of OutLevel parameters and Auto commands (BiCore)

- There are additional parameters for feedback configuration: *FB2, EffectiveFB2* and *StatAct2*, see Table 34. If an object has no, or only one or two, feedback signals, *StatAct1, StatAct2* and *StatDeact* are still set. The signals *StatAct1, StatAct2,* and *StatDeact* are set to false, if *FBConfig* = 0, 1, 2 or 3, and more than one feedback is True at the same time.

Table 34. Possible combinations of feedback signals (BiCore)

FBConfig	Position 1	Position 2	Deactivated position
0	FB1	FB2	FB0
1	FB1	FB2	(none)
2	FB1 inverted	FB2 inverted	(none)
3	FB1 inverted	FB2 inverted	FB0 inverted
4	(none)	(none)	(none)

- For implementation of a bi-directional motor, the BiCore function block has a parameter named *ChangeOverTime* (*Time* data type). The *ChangeOverTime* parameter is used for large motors in order to secure priority operation (switching forward/reverse). The parameter *ChangeOverTime* sets a delay time, before the direction can be changed (see Figure 127).

The *ChangeOverTime* parameter has an initial value of 5 seconds.

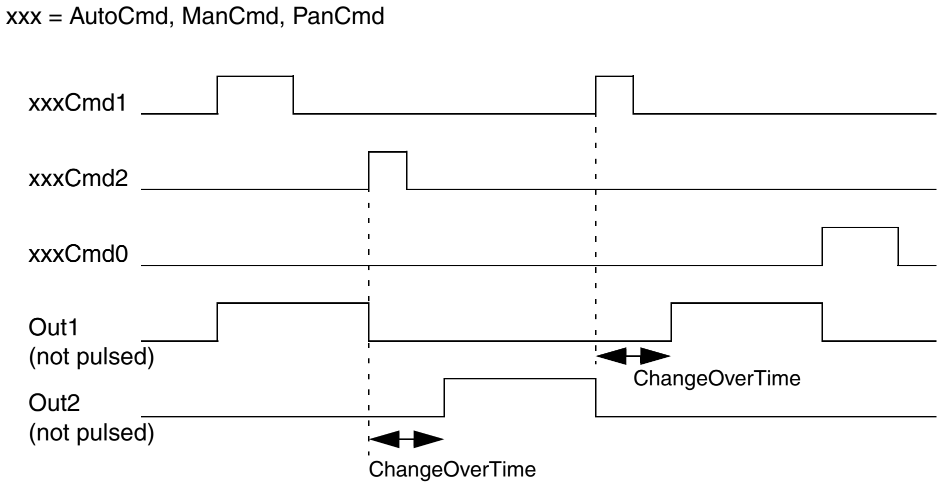


Figure 127. Change-over action state diagram

ChangeOverTime is only applicable to *Out1*, *Out2,* and *Out0*. It is not applicable to *Out1Level* or *Out2Level*, as they present the actual command.

It is possible to configure all Bi Process Objects to work in two-speed mode instead of bi-directional mode (switching forward/reverse). This is done by setting the *ChangeOverTime* parameter to zero.

- The *Ilock1*, *Ilock2* and *Ilock0* interlocking parameters block commands in all modes, so that the object cannot be forced to the activated or deactivated states, respectively. The *Inhibit* signal will suppress interlock signals when True. Figure 127 shows which changes between states that are blocked by interlocking parameters.

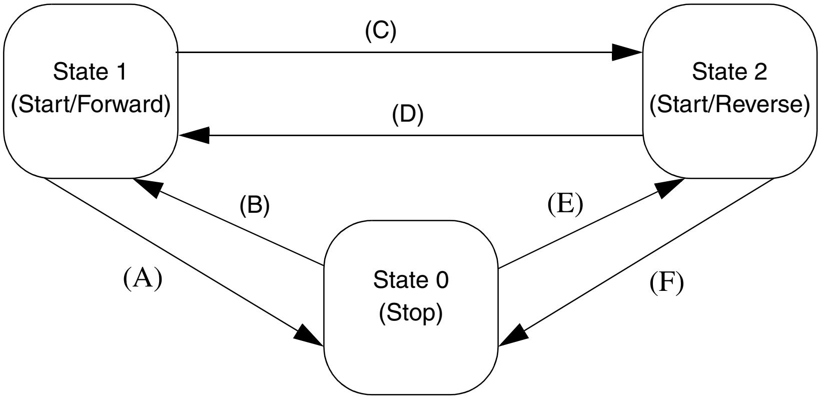


Figure 128. Manual maneuvers interlocking signals

(A) - Condition for jump to state 0: State 0-maneuver and NOT *Ilock0*.

(B) - Condition for jump to state 1: State 1-maneuver and NOT *Ilock1*.

(C) - Condition for jump to state 2: State 2-maneuver and NOT *Ilock2*.

(D) - Condition for jump to state 1: State 1-maneuver and NOT *Ilock1*.

(E) - Condition for jump to state 2: State 2-maneuver and NOT *Ilock2*.

(F) - Condition for jump to state 0: State 0-maneuver and NOT *Ilock0*.

PriorityCmd0 has the highest priority of the three priority commands and affects *Out0*. Similarly, *PriorityCmd1* and *PriorityCmd2* affect *Out1* and *Out2*.

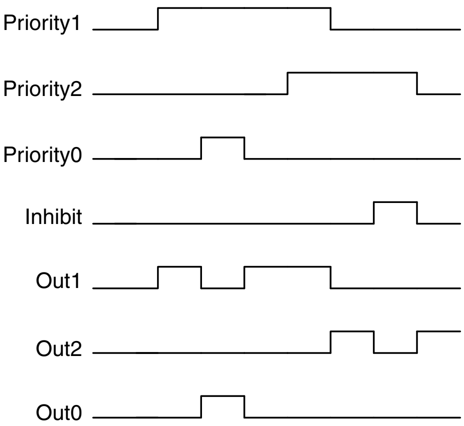


Figure 129. State diagram showing Priority command priorities and output signals with the effect of Inhibit

The *Inhibi*t signal overrules all priority commands. When the *Inhibit* signal is active, the object ignores the status of *PriorityCmd* (the Priority commands do not affect the output signals).

- The parameter *Out2IOLevel,* of the in/out type, is the result calculated from the actual values of the binary I/O (*Out2IO*, *Out0IO*) in relation to the selected pulse functionality.

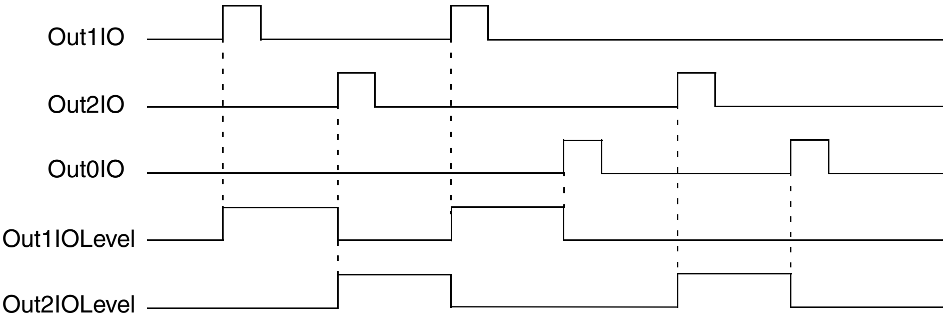


Figure 130. OutputIOLevel state diagram

The output signals are connected to and displayed in the operator workplace.

Control Module Icons

In Control Builder, a control module is represented by an icon that shows the most important module characteristics dynamically. The interaction window can be opened by clicking this icon. Connections to a group start environment is made using normal parameter connections for the process objects. For control modules, it is also possible to make graphical connections.

Figure 131 shows a control module icon in Control Builder.

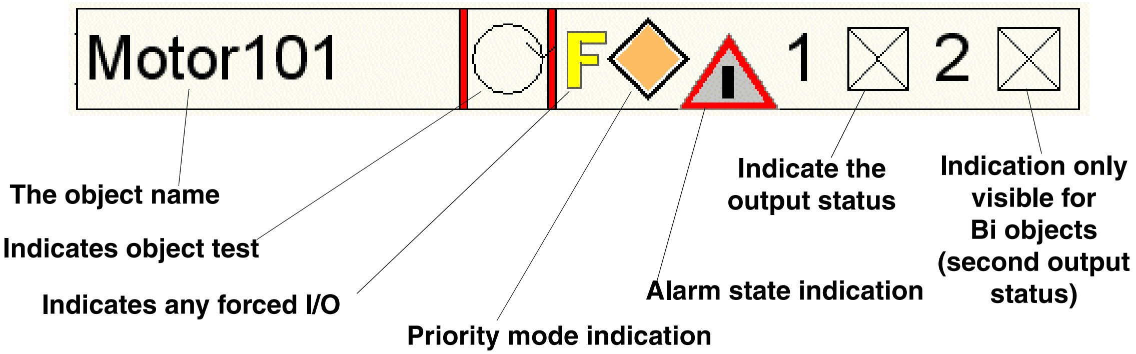


Figure 131. Graphical representation and explanations of the control module icons

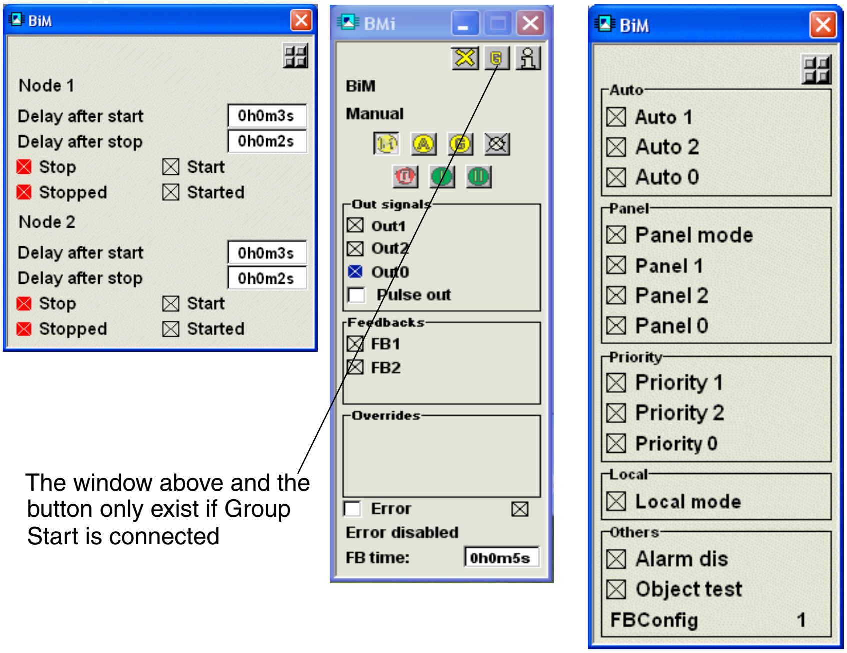
When modifying a process object, changes may also have to be made to the graphics. When additions are made in Control Builder graphics, window positioning might have to be changed. To change the position of an interaction window, double-click the faceplate (for example FaceplateBi) in the control module types folder. Select Variables for x and y position and change their value. After a positioning change, change Initial Value to Value. The values are relative and can be set for both the x and y position.

Interaction Windows

Interaction windows are used for maintenance purposes. Through an interaction window, the service engineer or programmer may manipulate the process object. All types in the Process Object standard libraries have at least one pre-made interaction window.

For example, the Bi process object has three Control Builder interaction windows, one for manual control, one for indications and one for Group Start (see Figure 132). The main interaction window is displayed first.

The extended interaction window is displayed by a click on the information icon in the main interaction window and the Group Start window is displayed by a click on the G-button.



BiM

Node 1
Delay after start 0h0m3s
Delay after stop 0h0m2s
☒ Stop ☒ Start
☒ Stopped ☒ Started
Node 2
Delay after start 0h0m3s
Delay after stop 0h0m2s
☒ Stop ☒ Start
☒ Stopped ☒ Started

BMi

BiM
Manual
Out signals
☒ Out1
☒ Out2
☒ Out0
□ Pulse out
Feedbacks
☒ FB1
☒ FB2
Overrides
□ Error ☒
Error disabled
FB time: 0h0m5s

BiM

Auto
☒ Auto 1
☒ Auto 2
☒ Auto 0
Panel
☒ Panel mode
☒ Panel 1
☒ Panel 2
☒ Panel 0
Priority
☒ Priority 1
☒ Priority 2
☒ Priority 0
Local
☒ Local mode
Others
☒ Alarm dis
☒ Object test
FBConfig 1

The window above and the button only exist if Group Start is connected

Figure 132. Control Builder interaction windows for Bi(M)

Interaction Parameters

Interaction parameters are used to interact with the process object. Interaction parameters can be accessed from the code (this is not recommended, since it might shut out the operator interface) and from the graphical interface. Interaction parameters are identified by the syntax *InteractionPar.ComponentName*.

For more information on interaction parameters, see Control Builder online help for the Process Object libraries. Interaction parameters for process objects are also described in connection with instructions on how to configure the objects, see Advanced Functions on page 317.

Design

The types in the Process Object libraries can be used in a number of ways. There are also a number of choices that have to be made regarding which type of object to use for which purpose.

This sub section contains information designed to help you:

- Decide which type to use, see Choose the Correct Type on page 287.
- Configure standard types to be used in your application as is, see Use Standard Library Types on page 288.
- Create your own types based on core objects, see Use Standard Library Types to Create Self-defined Types on page 289.
- Connect process objects to a group start configuration, see Group Start Interface on page 289.
- Configure alarm handling, see Alarm Handling on page 291.
- Create uni-directional and bi-directional control solutions using the Uni(M) and Bi(M) template objects, see Generic Uni- and Bi-Directional Control on page 293.
- Configure motor control and valve control solutions using objects from the Process Object libraries, as templates or as is. See Motor and Valve Control on page 296.

Choose the Correct Type

In choosing a representation of a specific process object, you must decide on the following:

- Should I use a function block or a control module? See Function Blocks vs Control Modules on page 287.
- Which process object type should I use? See Type Selection Chart on page 287.

Function Blocks vs Control Modules

All process objects (except the core function blocks) are delivered in two versions, function blocks and control modules. Control modules have an suffix M in the type name.

In simple applications (or small ones) that will not need to be modified at a later date, traditional function block solutions may be used. In more complex applications, control modules are preferred.

One of the benefits of using control modules, is that they allow the user to insert many similar objects, as the code sorting routine ensures that variables are dealt within the correct order. The more objects to be inserted, the more you gain by using control modules.

Function block parameters are copied at each call, while control module parameters are defined at compilation and set up once, prior to execution. Control modules have a performance advantage, especially when large structures (for example, structured data types) are used as parameters, of the in and/or out type, in function blocks, or when parameters are transferred through deep hierarchies.

Type Selection Chart

If the process object is intended to be applied to a bi-directional object (an object with three states) and it is a motor, choose the MotorBi(M) type. If it is a bi-directional object, but not a motor, choose the Bi(M) type or the BiSimple(M) type if you do not need alarm handling. If it is a bi-directional object and a motor valve, choose MotorValve(M). If it is a bi-directional object and a motor controlled valve of open/close type then choose MotorValveCC.

If the process object is to be used for a uni-directional object, (an object with two states) several choices are possible. If it is not a motor or a valve, choose the Uni object type or the UniSimple(M) type if you do not need alarm handling. If it is a uni-directional motor object, choose MotorUni(M) and if it is a uni-directional valve object, choose ValveUni(M).

Figure 133 contains a flow chart for selecting the correct type for your process objects.

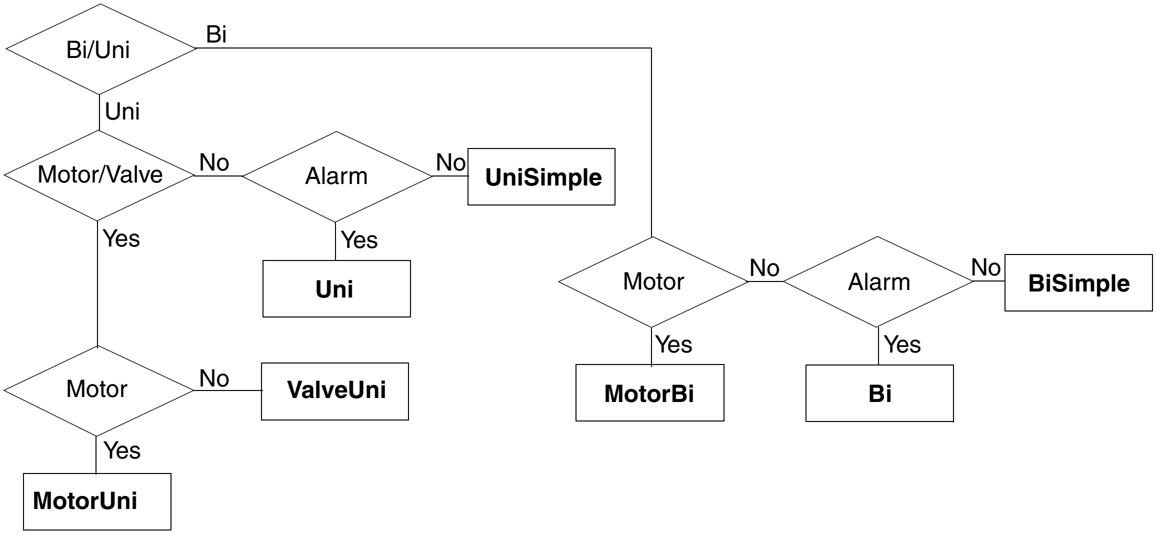


Figure 133. Flowchart of possible process object type choices

Use Standard Library Types

Some of the process objects are delivered as template objects. This means that the code is open and readable for the user. However, process objects may be instantiated as they are, if no addition or change is needed.

If you want to use one of the types in the Process Object libraries without any modification, simply connect the library to your application and create one or several instance(s) of it. The only thing you have to do is configure the instances by connecting their parameters to your application.

For examples of how to do this, see Examples on page 304. For detailed information about parameters for the different types, see Level Detection, Commands and Alarm Texts on page 317.

Use Standard Library Types to Create Self-defined Types

It is possible to create your own types, based on the types in the Process Object libraries. However, to make it possible to modify and add to your self-defined types, you need to create copies of the Process Object library types you want to modify, and store them in a library that you have created for this purpose. For an example of how to create a new library and copy types to it from the Process Object libraries, see Create a Library and Insert a Copy of a Type on page 304.

If you use a type from a Process Object library as a formal instance in one of your self-defined types, and this sub type is updated (for example, through an upgrade of your system), the changes are reflected in your self-defined type(s) as well.

You can add functions to the copied object types that you store in self-defined libraries. For an example of how to do this, see Add Functions to Self-defined Types on page 309.

For detailed information about parameters for the different process object types, see Level Detection, Commands and Alarm Texts on page 317.

Group Start Interface

All process objects in the Process Object library (except ProcessObjBasicLib) have the group start interface where connections may be done to the group start environment.

The Group Start interface consists of the following parameters:

- *GroupStartIn* *,
- GroupStartMode,
- GroupStartILock.

* One for each connection (Uni types have one connection and Bi types have two connections).

The parameter *GroupStartIn* is the structured parameter, that connects the process object to the group start environment and is in the control module cases implemented as a node to allow a graphical connection. The out parameter *GroupStartMode* indicates Group Start mode with a true value in the corresponding boolean parameter. When the parameter *GroupStartILock* is true, transfer to group start mode is inhibited.

These parameters are optional and only implemented in the more complex motor objects. The purpose is to halt the start or stop sequence if any external signal requests it and all Txt-parameters are strings, telling the Group Start environment the reason of the halt.

- *ContinueStartSeq* *
- *ContinueStartSeqTxt* *
- *ContinueStopSeq* *
- *ContinueStopSeqTxt* *

* One for each connection (Uni types have one connection and Bi types have two connections).

For a description of the Group Start library, see Section 6, Synchronized Control.

Voting Interface

Several process objects support Voting; for more information about these objects and their corresponding Voting parameters, see Signal and Vote Loop Concept on page 367.

Alarm Handling

For more information on alarm and event handling, see the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* , and online help for the Alarm and Event library.

Some process objects contain an alarm and event handling function block. The alarm handling interface consists of the following parameters:

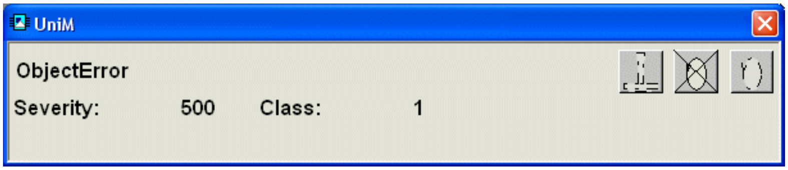
- AlarmDisabled
- ExtErr
- ObjErr
- AlState
- AlarmAck
- AEConfig
- AESeverity
- AEClass
- EnableSupOut

When an error occurs, motor objects normally and automatically enter Priority mode and reset the start signals. It is possible to leave the motor running, by setting the interaction parameter *KeepAtErr*. For other template objects, this is the normal functionality.

For the alarm to work properly, the *Name* parameter of each object has to be unique throughout the whole plant.

The alarm function is based on the AlarmCond function block, with acknowledge rule (*AckRule*=1). This acknowledge rule includes six possible alarm states. For further information about alarm states, see online help on the AlarmCond function block. The alarm is activated by a feedback error.

The AlarmCond function block incorporates an alarm Control Builder interaction window, displayed by a clicking the alarm icon in the Control Builder interaction window. See Figure 134.



ObjectError

Severity: 500 Class: 1

Figure 134. An alarm Control Builder interaction window

If the object is in Disabled or in Local mode, or if the feedback error is disabled, the alarm function is disabled.

An alarm can be acknowledged from the program via the parameter *AlarmAck*, from the alarm Control Builder interaction window.

Error Texts

Error text strings can be added by the OETextUni, OETextBi, and OETextValveUni function block that are included in their corresponding process object types. These function blocks generate a text message from the feedback and output signals of the AlarmCond function block. The text message is displayed in the Alarm Control Builder interaction window.

The text is put together by adding the name of the object that created the alarm to information about status and feedback. If the error has been generated from an *ExtErr* parameter, an external error text is also displayed.

For more information on the OEText function blocks, see Level Detection, Commands and Alarm Texts on page 317. More information about parameters is also found in online help for the Process Object libraries.

Generic Uni- and Bi-Directional Control

The Process Object Extended library contains two objects that are intended as generic template objects for uni- and bi-directional applications:

- Uni(M), see Uni(M) on page 294
- Bi(M), see Bi(M) on page 295

There are two versions of each object, one function block type and one control module type (indicated by the letter M).

There are also two objects in the Process Object Basic library that are simplified versions of the above two:

- UniSimple(M), see UniSimple(M) on page 296
- BiSimple(M), see BiSimple(M) on page 296

As for the Uni(M) and Bi(M) objects, there are two versions, one function block type and one control module type.

All the above objects are based on UniCore and BiCore. Before reading this section, you should always be familiar with the functions and parameters of the core objects, see Core Object Functions and Parameters (UniCore and BiCore) on page 266.

All the above function block and control module types contain function blocks from the Process Object Basic and/or Process Object Extended libraries. For a description of those function blocks, see Level Detection, Commands and Alarm Texts on page 317.

Some parameters in the function blocks are different from the ones in the control modules. For example, the voting interface in the control modules do not exist in the function blocks.

Uni(M)

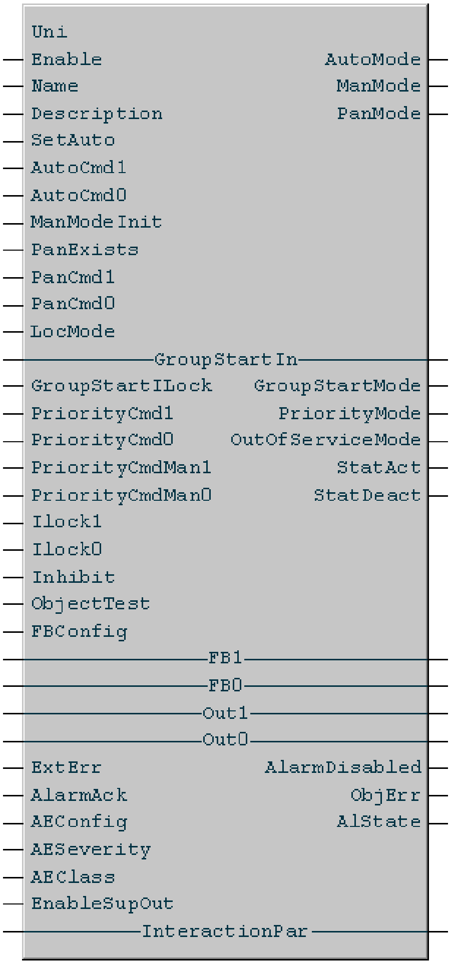
Uni(M) object is based on a UniCore. Uni(M) includes an alarm function, Control Builder interaction windows .

Uni(M) is intended for uni-directional control and can be used as a template for developing your own uni-directional types, see Create a Library and Insert a Copy of a Type on page 304.

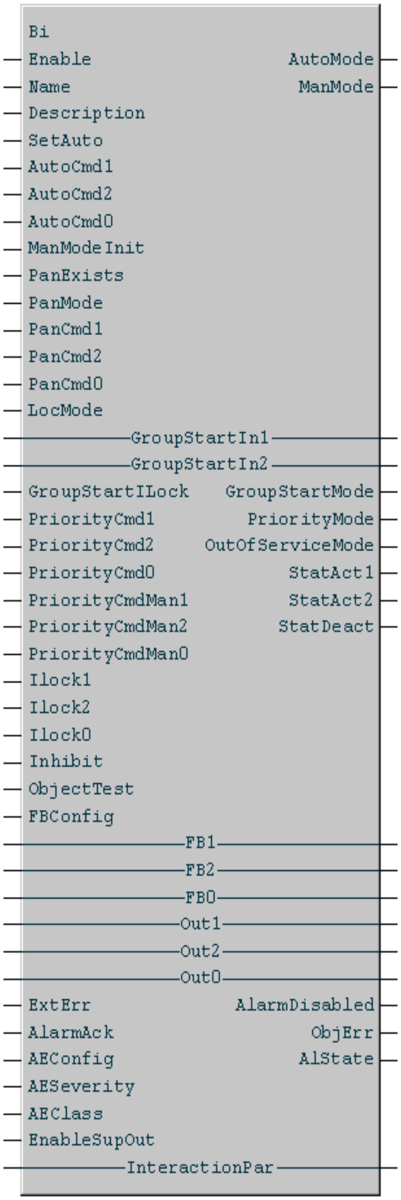
For Uni(M), it is possible to control Panel mode of the object from both a workplace, and from a switch on a control panel. The interaction window has a button that can toggle Panel mode.

There is also a parameter (*PanMode*) intended to be connected to an activation signal from the panel. If you connect *PanMode* to the panel, it is still possible to use the button in the interaction window to activate Panel mode.

It is not possible to deactivate Panel mode from the interaction window if it has been activated from the control panel. For an example of a panel control configuration, see Connect to a Control Panel in Panel Mode on page 313.



Bi(M)



Bi(M) is an example of a BiCore. Bi(M) includes an alarm function, Control Builder interaction windows.

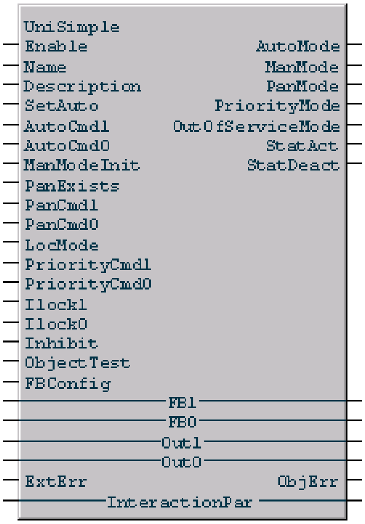
Bi(M) is intended for bi-directional control and can be used as a template for developing your own bi-directional types, see Create a Library and Insert a Copy of a Type on page 304.

For Bi(M), it is possible to control Panel mode of the object from both a workplace, and from a switch on a control panel. The interaction window has a button that can toggle Panel mode.

There is also a parameter (*PanMode*) intended to be connected to an activation signal from the panel. If you connect *PanMode* to a panel it is still possible to use the button in the interaction window to activate the Panel mode.

It is not possible to deactivate Panel mode from the interaction window if it has been activated from the control panel. For an example of a panel control configuration, see Connect to a Control Panel in Panel Mode on page 313.

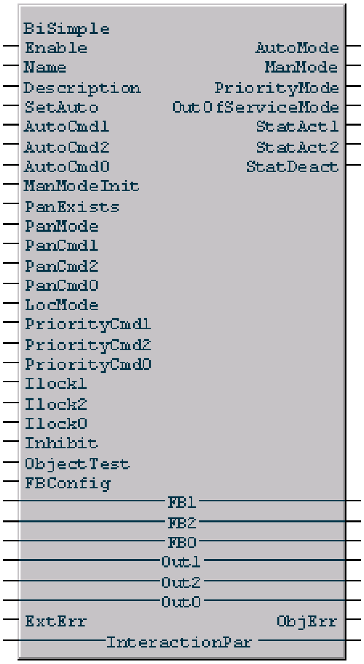
UniSimple(M)



UniSimple(M) is a UniCore application. It includes Control Builder interaction windows.

UniSimple(M) has the same functions as Uni(M), except for the fact that there is no alarm and event handling.

BiSimple(M)



BiSimple(M) is a BiCore application. It includes Control Builder interaction windows.

BiSimple(M) has the same functions as Bi(M), except for the fact that there is no alarm and event handling.

Motor and Valve Control

The Process Object Extended library contains two motor control objects (each in a function block type and a control module type version), one valve object (in a function block type and a control module version), and three motor valve objects (in one function block type and two control module types):

- MotorUni(M) for uni-directional motors, see MotorUni(M) on page 298.
- MotorBi(M) for bi-directional motors, see MotorBi(M) on page 300.
- ValveUni(M) for valves, see ValveUni(M) on page 301.
- MotorValve(M) for motor valves, see MotorValve(M) on page 302.
- MotorValveCC for motor valves, see MotorValveCC on page 302.

For a description of how to configure ABB Drives and INSUM control, see Advanced Functions on page 317.

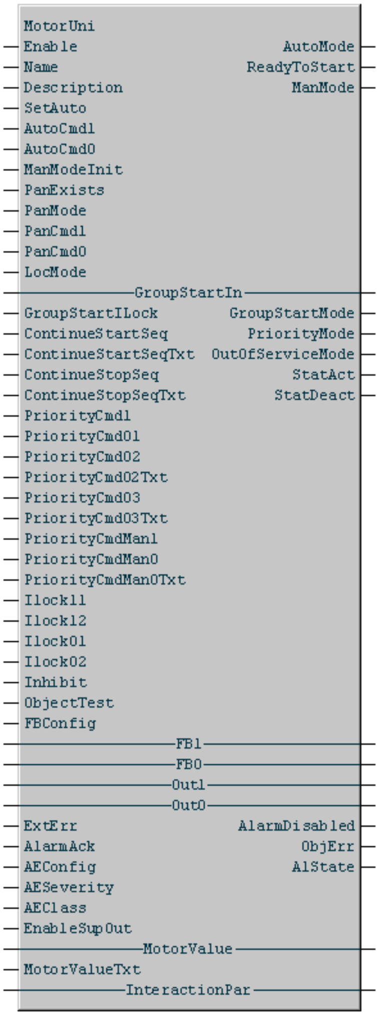
All the above objects are based on UniCore and BiCore. Before reading this section, you should always be familiar with the functions and parameters of the core objects, see Core Object Functions and Parameters (UniCore and BiCore) on page 266.

All the above function block and control module types contain function blocks from the Process Object Basic and/or Process Object Extended libraries. For a description of those function blocks, see Level Detection, Commands and Alarm Texts on page 317.

These types can be used as is, or as templates. If you want to use them as templates, they have to be copied to your own library and then modified, see Create a Library and Insert a Copy of a Type on page 304.

When using the motor and valve process objects as is, the only thing you have to do is to connect the parameters that do not have default values.

MotorUni(M)



MotorUni(M) is an example of a UniCore application. It includes an alarm function, Control Builder interaction windows. It is intended to be used to control a uni-directional motor object (stop and run).

This section only discusses functions that have been added, compared to the functions of UniCore and Uni(M). You should also read Core Object Functions and Parameters (UniCore and BiCore) on page 266.

Auto mode can be set from the program, interaction windows. Since Auto mode implies automatic operation, the program controls the object via *AutoCmd1* and *AutoCmd0*.

Each of these parameters is supplied with a value of *FBTime*, connected to the interaction window, via interaction parameter components. The value of *FBTime* can be changed from the corresponding graphical window. The interaction parameter components have the cold retain attribute to retain their values following a cold restart. *FBTime* for *AutoCmd1* has the same setting as *AutoCmd1* and *AutoCmd0*, because the same local variable is used.

For MotorUni(M), it is possible to control Panel mode of the object from both a workplace, and from a switch on a control panel. The interaction window has a button that can toggle the Panel mode.

There is a parameter (*PanMode*), which is intended to be connected to an activation signal from the panel. If you connect PanMode to the panel it is still possible to use the button in the interaction window to activate the Panel mode. It is, however, not possible to deactivate Panel mode from the interaction window if it has been activated from the control panel.

Modification of the interlocking function is performed to extend some parameters, to apply user-defined text, and to implement the combination of *PriorityCmd0* and an alarm function via *ExtErr*. Further improvement is possible, based on this example.

MotorUni(M) object has four interlocking parameters: *Ilock01*, *Ilock02*, *Ilock11*, and *Ilock12*, that set the condition of *Ilock0* or *Ilock1* in the UniCore function block. The *Ilock0* condition is set by *Ilock01* or *Ilock02*. Similarly, the *Ilock1* condition is set by *Ilock11* or *Ilock12*.

The *PriorityCmd0* parameter is divided into three parameters: *PriorityCmd01*, *PriorityCmd02*, and *PriorityCmd03*. *PriorityCmd02* and *PriorityCmd03* have associated parameters: *PriorityCmd02Txt*, and *PriorityCmd03Txt*, to provide a descriptive text in the alarm list and the interaction window. The text is displayed in the Priority Interlock menu area.

The extended *PriorityCmd0* (-*Cmd01*, -*Cmd02* and -*Cmd03*) has a hold function with an alarm. When the priority command is active, the feedback error is generated by an alarm and can only be released by alarm acknowledge or inhibit.

The *Inhibit* signal for MotorUni(M) process objects works exactly the same as described earlier in Core Object Functions and Parameters (UniCore and BiCore) on page 266, with one exception: *Inhibit* releases a feedback error, activated by *PriorityCmd0*.

MotorBi(M)

MotorBi(M) is an example of a BiCore application, with an alarm function, interaction windows.

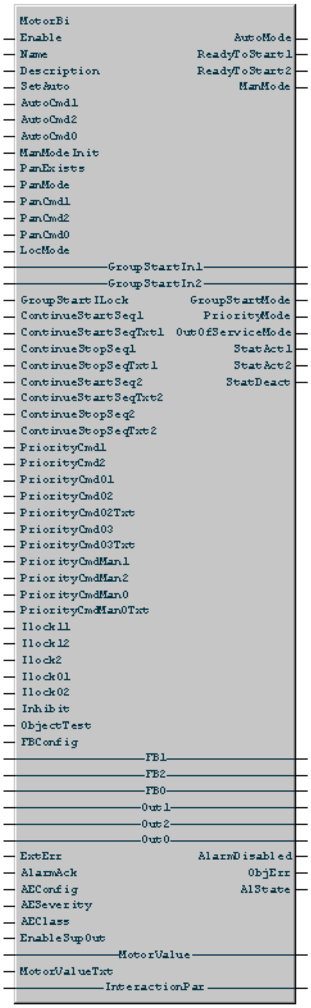
This section only discusses functions that have been added, compared to the functions of BiCore and Bi(M).You should also read Core Object Functions and Parameters (UniCore and BiCore) on page 266.

Auto mode can be set from the program, interaction windows. Since Auto mode is the automatic operation, the program controls the object via *AutoCmd1*, *AutoCmd2* and *AutoCmd0*.

AutoCmd1 and *AutoCmd2* are supplied with an *OnDelayTime* interaction parameter, *AutoCmd0* is supplied with an *OffDelayTime* interaction parameter, connected to the interaction window, via interaction parameter components. The value of *OnDelayTime* and *OffDelayTime* can be changed in the graphical window.

Interaction parameter components have the cold retain attribute to retain their values at cold restart. *OnDelay* for *AutoCmd1* and *AutoCmd2* has the same setting as *AutoCmd1*, *AutoCmd2* and *AutoCmd0*, since the same local variable is used.

Just like for MotorUni(M), it is possible to control Panel mode of the object from both a workplace, and from a switch on a control panel. The interaction window has a button that can toggle the Panel mode.



MotorBi(M) implements the BiCore change-over function. A parameter is provided for setting the change-over time, and it is connected to the corresponding parameter in BiCore. The interaction parameter component for the change-over time is also connected to the interaction window. This makes it possible to change the value in the corresponding graphical window. The interaction parameter components have the cold retain attribute to retain the values following a cold restart.

Interlocking and priority commands work as for MotorUni(M), but with additional parameters for the second output and the extra state.

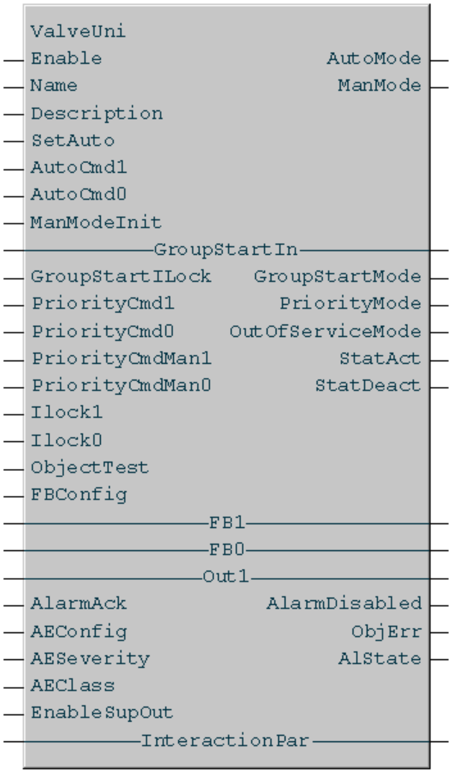
Error texts are generated in the same way as for MotorUni(M).

ValveUni(M)

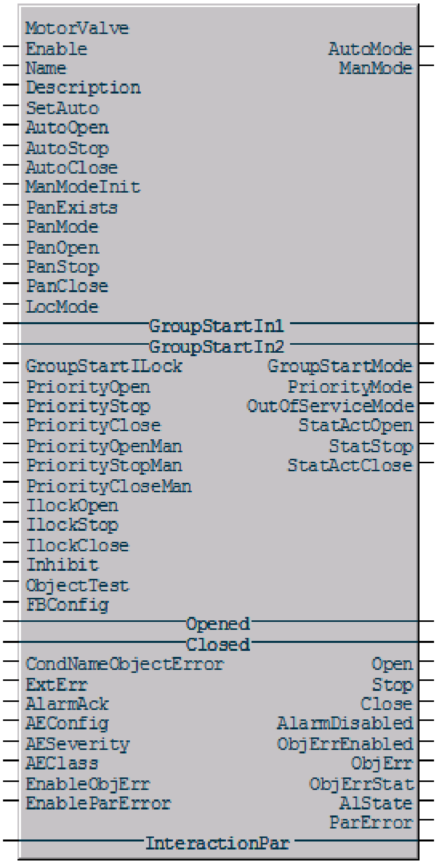
The ValveUni(M) process object is a simple example of the UniCore function block. ValveUni(M) includes an alarm function, Control Builder interaction windows.

The ValveUni(M) object is incorporated with only one output I/O for open command.

When using the ValveUni(M) as is, the only thing you have to do is to connect the parameters that do not have default values.



MotorValve(M)



The MotorValve(M) and MotorValveCC process objects are based on BiCore and is extended with alarm and interaction windows.

MotorValve is suitable for graphical control and supervision of a bidirectional (two activated position) motor valve. It is used for motorized valves that are to be maneuvered to any position.

The object has two feedback signals in the Opened and Closed position. When the valve is moving towards its Closed position, the valve may be stopped at any position and also manoeuvered towards the Open position. The stop may occur both in the beginning of the movement where the closed feedback is still true and between both the feedback switches. The opposite is relevant for the opening command. When the Opened position has been reached, the Open output remains true until the stop command is given. When the Closed position has been reached, the Close output remains true until the stop command is given. During the Open and Close operations the transition time is supervised.

When the Opened position is reached, the Open output remains true until the Stop command is given. When the Closed position is reached, the Close output remains true until the Stop command is given. During the open and close operation, the transition time is supervised.

MotorValveCC

The MotorValveCC control module type is suitable for graphical control and supervision of a motor controlled valve of open/close type. It is based on BiCore and is extended with alarm and interaction windows. The input interface is of type control connection that can be connected to an output object from a control loop.

The object is a composite object mainly consisting of *ThreePosCC* from ControlStandardLib, and *MotorValveM* from the ProcessObjExtLib.

Examples

This sub section contains a number of examples that have been included to show how to implement process objects, how to create your own, application-specific types, and to illustrate some important concepts and relations:

- Create a Library and Insert a Copy of a Type on page 304 gives an example of how to create a library and copy Process Object library types into this library.
- Add Functions to Self-defined Types on page 309 gives an example of how functions can be added to a type.
- Connect to a Control Panel in Panel Mode on page 313 shows how to connect UniCore and BiCore to a control panel (Panel mode).

Create a Library and Insert a Copy of a Type

This example shows how to copy a type from the Process Object Extended library to a user-defined library.

1. In Project Explorer, right-click the Libraries folder and select **New Library...**
2. Type the name of the library to be created in the Name field, for example, **`MotorLib`**. Click **OK**.
3. In the Process Object Extended library, locate MotorBiM (in the Control Module Types folder).
4. Right-click the type and select **Copy** (Ctrl+C).
5. Right-click the your newly created library (in this example MotorLib), and select **Paste** (Ctrl+V). The copy of the object is created under the Control Module Types folder.

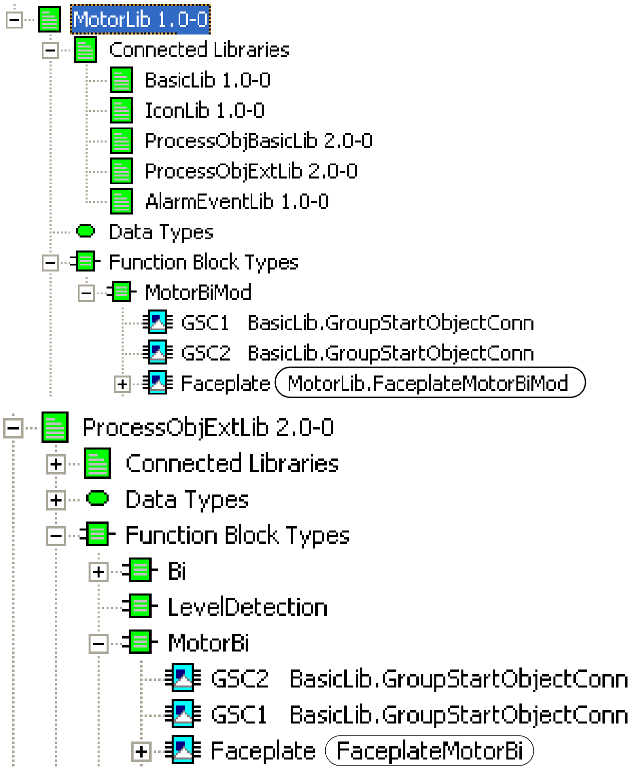
   The default name of the copied object is the name of the copied original object type.

To change the name of the object, proceed as follows:

6. Right-click the object and select **Rename...**
7. Type the desired name of the object, in the New name text field, in this example, MotorBiMod. Click the **OK** button.

   Process object types in the Process Object libraries contain a number of control modules, for example, the FaceplateMotorBi control module, which can be used as an interaction window for the MotorBi(M) process object type. The control modules refer (or point) to their types in ProcessObjExtLib.

   If these control module types are to be changed in your application, the updated control module types must also to be copied to your library.
8. Expand the folder of the copied process object type, and identify the control module types that refer to the control modules used in the process object type. For example, the control module type MotorLib.FacePlateMotorBi refers to control module FaceplateMotorBi used in the MotorBi(M) process object type.



```
MotorLib 1.0-0
    Connected Libraries
        BasicLib 1.0-0
        IconLib 1.0-0
        ProcessObjBasicLib 2.0-0
        ProcessObjExtLib 2.0-0
        AlarmEventLib 1.0-0
    Data Types
    Function Block Types
        MotorBiMod
            GSC1   BasicLib.GroupStartObjectConn
            GSC2   BasicLib.GroupStartObjectConn
            Faceplate MotorLib.FaceplateMotorBiMod
ProcessObjExtLib 2.0-0
    Connected Libraries
    Data Types
    Function Block Types
        Bi
        LevelDetection
        MotorBi
            GSC2   BasicLib.GroupStartObjectConn
            GSC1   BasicLib.GroupStartObjectConn
            Faceplate FaceplateMotorBi
```

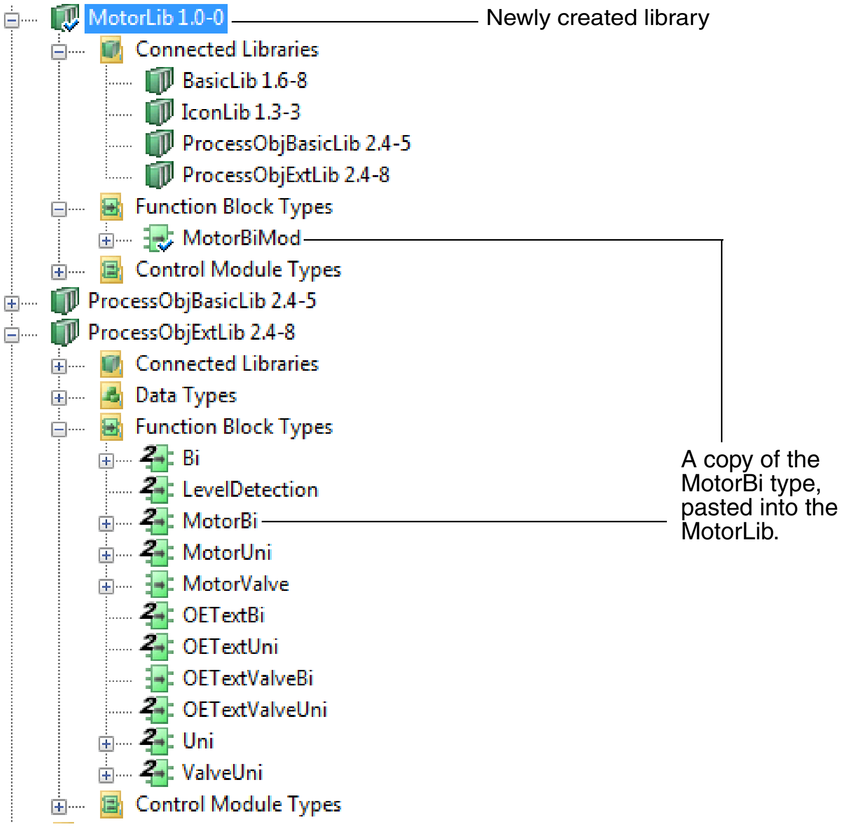
Figure 135. The control module type is copied to your own created library

9. In the Control Module Types folder of the Process Object Extended library, select the control module type that is to be copied (for example, FaceplateMotorBi).

10. Right-click on the control module type and select **Copy** (Ctrl+C).

11. Right-click on the Control Module Types folder in your newly created library (in this example MotorLib). Select **Paste** (Ctrl+V). (See steps 6 and 7 if you wish to change the name of an object type.)

12. Select the control module type that is to be replaced, under the process object type, in your own created library (MotorLib), in this example, FaceplateMotorBiMod under MotorBiMod. See Figure 136.



- MotorLib 1.0-0
  - Connected Libraries
    - BasicLib 1.6-8
    - IconLib 1.3-3
    - ProcessObjBasicLib 2.4-5
    - ProcessObjExtLib 2.4-8
  - Function Block Types
    - MotorBiMod
  - Control Module Types
- ProcessObjBasicLib 2.4-5
- ProcessObjExtLib 2.4-8
  - Connected Libraries
  - Data Types
  - Function Block Types
    - Bi
    - LevelDetection
    - MotorBi
    - MotorUni
    - MotorValve
    - OETextBi
    - OETextUni
    - OETextValveBi
    - OETextValveUni
    - Uni
    - ValveUni
  - Control Module Types

Newly created library

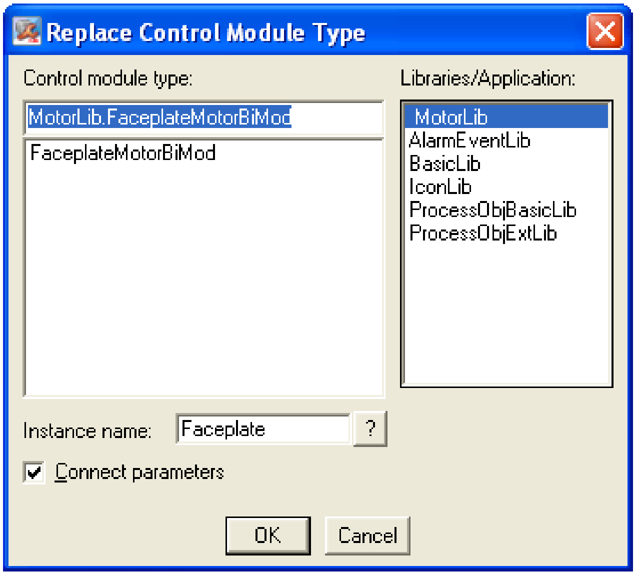
A copy of the MotorBi type, pasted into the MotorLib.

Figure 136. Copying of object types to a new library, MotorLib

13. Right-click and select **Replace Type**.

14. In the Libraries/Application list, select your self-defined library and in the Control module type list, select the control module type that is to replace the

original type. (It is possible to rename the control module in the Instance name field). Click the **OK** button.



Replace Control Module Type

Control module type: MotorLib.FaceplateMotorBiMod

FaceplateMotorBiMod

Libraries/Application:

- MotorLib
- AlarmEventLib
- BasicLib
- IconLib
- ProcessObjBasicLib
- ProcessObjExtLib

Instance name: Faceplate

☑ Connect parameters

Figure 137. Replace Control Module Type dialog

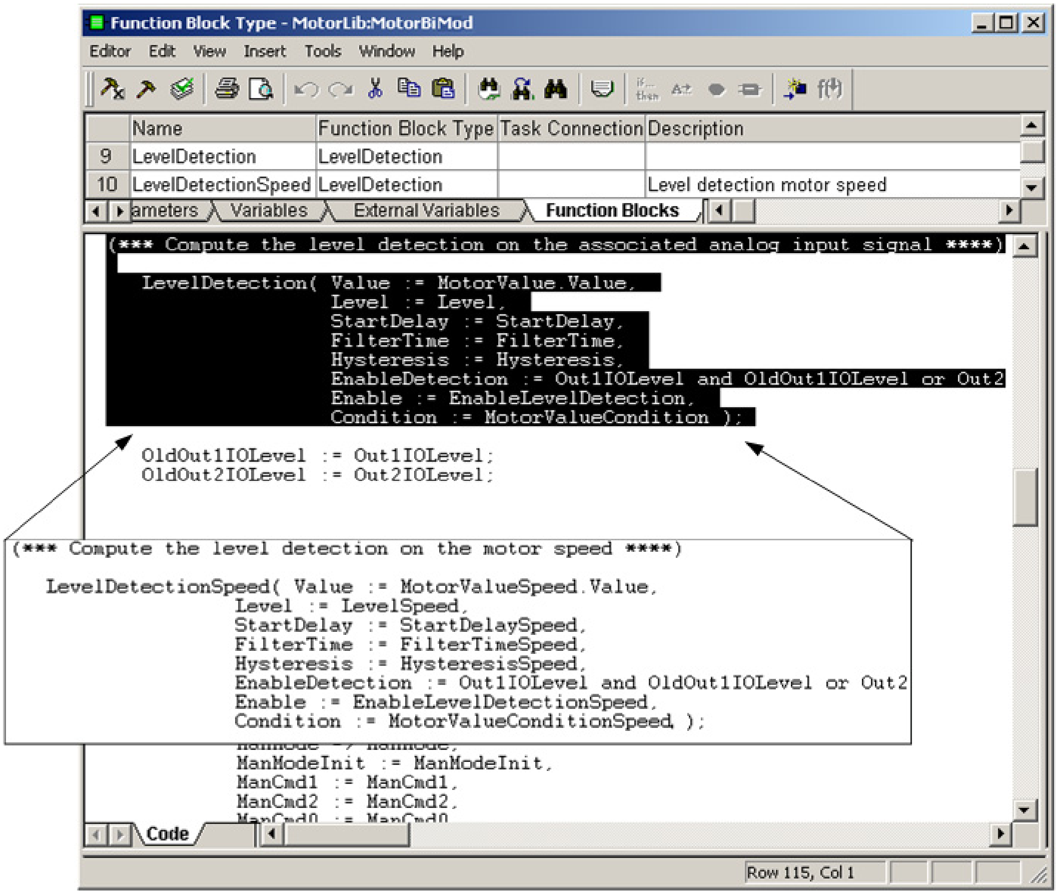
Add Functions to Self-defined Types

In the example below, a level detection for motor speed is added to the object type that was copied above, MotorBiMod.

In this example, no functionality for alarm text handling has been added.

1. In the Function Block Types folder in MotorLib, right-click on the MotorBiMod type and select **Editor** (ENTER).
2. Place the cursor in the code pane and choose **Edit>Find...** (Ctrl+F) from the menu. Search for the text **`level detection`**.
3. Make a copy of the code concerning `***Compute the level detection on the associated analog input signal***` and paste it into the code pane.
4. According to Figure 138, declare the new function block (LevelDetectionSpeed), change the function block name in the code pane and declare the required variables. Connect the parameters by right-clicking on the function block in the code pane and selecting **Edit>Parameter list** (Ctrl+M).

5. Select **Editor>Save and Close** (Ctrl+U) to implement the changes made in the MotorBiMod object type.



	Name	Function Block Type	Task Connection	Description
9	LevelDetection	LevelDetection		
10	LevelDetectionSpeed	LevelDetection		Level detection motor speed

```
(*** Compute the level detection on the associated analog input signal ****)

  LevelDetection( Value := MotorValue.Value,
                  Level := Level,
                  StartDelay := StartDelay,
                  FilterTime := FilterTime,
                  Hysteresis := Hysteresis,
                  EnableDetection := Out1IOLevel and OldOut1IOLevel or Out2
                  Enable := EnableLevelDetection,
                  Condition := MotorValueCondition );

  OldOut1IOLevel := Out1IOLevel;
  OldOut2IOLevel := Out2IOLevel;
```

```
(*** Compute the level detection on the motor speed ****)

  LevelDetectionSpeed( Value := MotorValueSpeed.Value,
                       Level := LevelSpeed,
                       StartDelay := StartDelaySpeed,
                       FilterTime := FilterTimeSpeed,
                       Hysteresis := HysteresisSpeed,
                       EnableDetection := Out1IOLevel and OldOut1IOLevel or Out2
                       Enable := EnableLevelDetectionSpeed,
                       Condition := MotorValueConditionSpeed );
```

```
                       ManModeInit := ManModeInit,
                       ManCmd1 := ManCmd1,
                       ManCmd2 := ManCmd2,
```

Figure 138. Level detection functionality added to the MotorBiMod function block type

It is now possible to use the MotorBiMod function block type in a program, with the new added functionality.

6. In the Program folder under Applications, right-click on a program, for example Program2, and select **Editor** (ENTER).
7. Declare a function block of the type MotorBiMod (in this example called MotorBi).
8. Insert the function block in the code pane, connect the desired parameters and declare the required variables. (In this example the Function Block Diagram language is used).

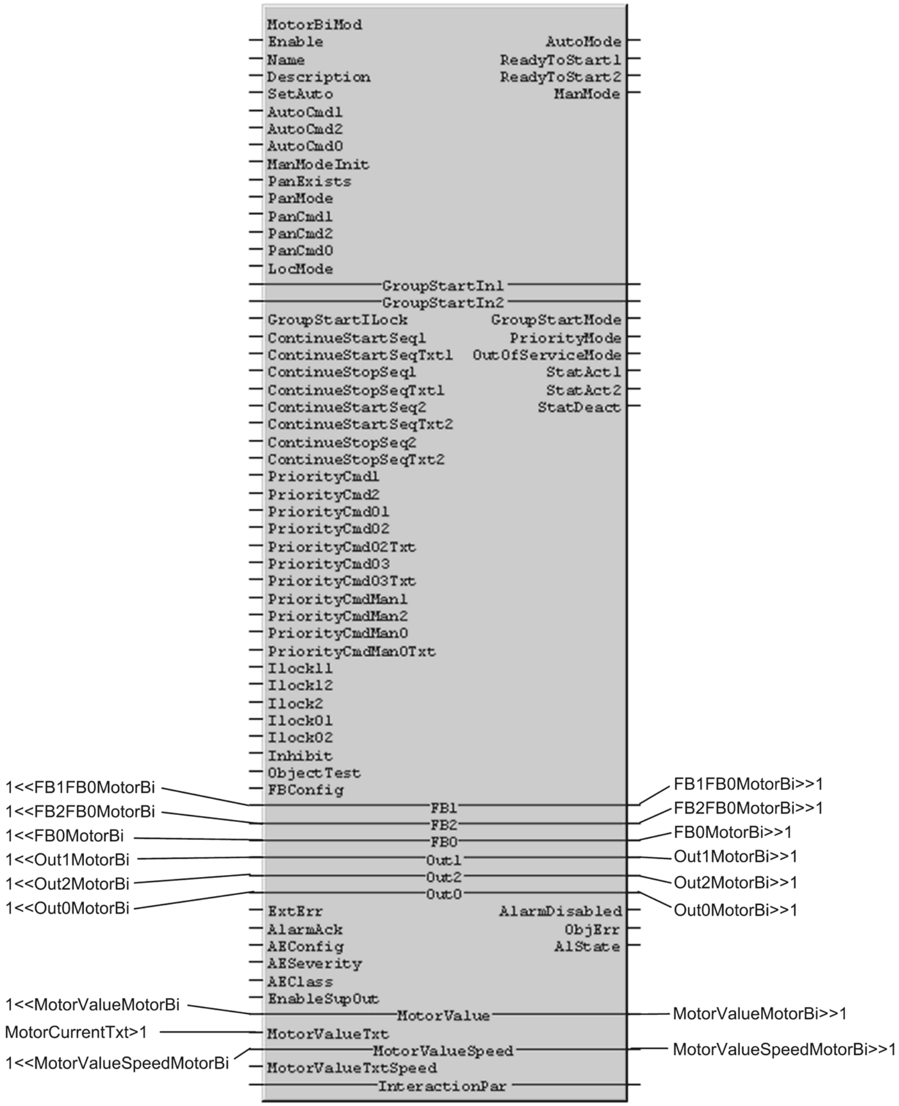


Figure 139. MotorBi function block with connections to variables

9. Select **Editor>Save and Close** (Ctrl+U) to implement the changes.

Connect to a Control Panel in Panel Mode

Whether or not Panel mode is used, depends on how comprehensive the application is. The question is whether you prefer to initiate control (the parameter *PanMode*) of the object from a workplace, or from a switch on a control panel.

When the Panel mode is active, the control panel takes control of the object and errors are calculated according to the status signals from the control panel. The output signal retains its status from the previous mode. The following examples show how to connect UniCore and BiCore to a control panel.

UniCore Examples

Figure 140 shows how to connect UniCore to a control panel in a small application.

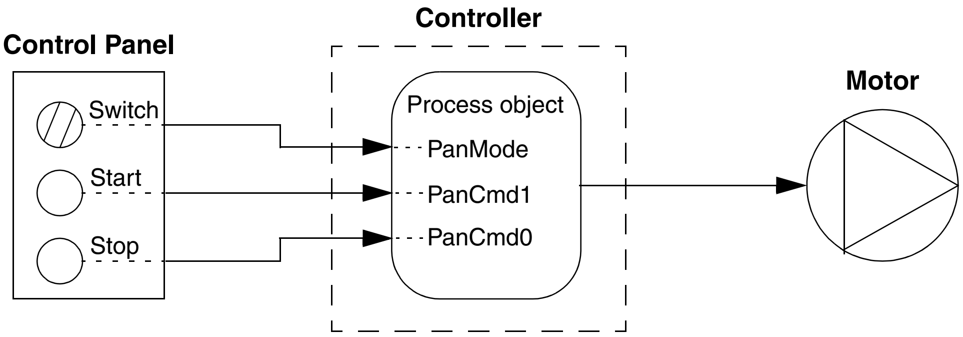


Figure 140. Control steps in Panel mode. This solution is recommended for small applications, where all three panel parameters are connected from a control panel

In large applications, the best solution is to connect the *PanMode* parameter, via variables, to interaction window. The *PanCmd1* and *PanCmd0* parameters are still connected to the physical Start/Stop buttons (see Figure 141). Changes in the Panel mode are therefore approved centrally, that is, a local operator must first obtain permission from the central control operator.

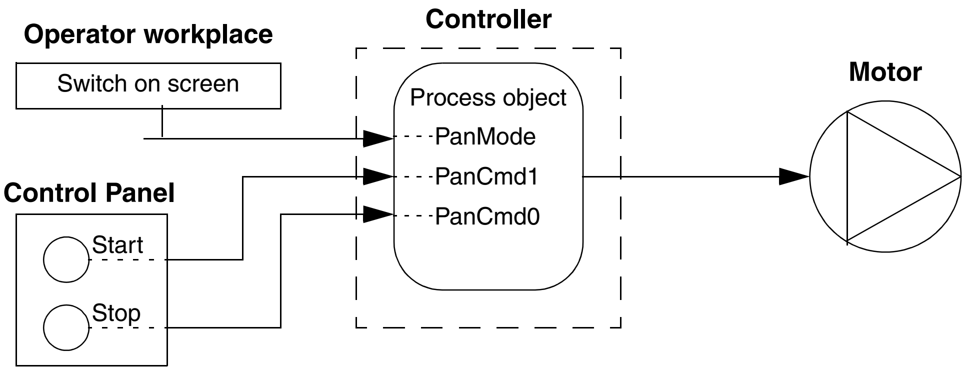


Figure 141. Control steps in Panel mode. This solution is recommended for large applications, where the PanMode parameter is connected to an operator workplace

The status of the object is controlled by the signals *PanCmd1* and *PanCmd0*, which are level signals and function in the same way as *AutoCmd* signals, as illustrated in Figure 141. The *PanCmd1* and *PanCmd0* parameters should be connected to push buttons. It may be advisable to use an *R_Trig* function block (trigger, parameter and push button) between *PanCmd1* and *PanCmd0*, in case the push button malfunctions.

BiCore Examples

A control panel is a natural choice for smaller applications. The *PanMode* parameter is connected to the switch on the control panel; *PanCmd1, PanCmd2* and *PanCmd0* are connected to the Start/Stop buttons (see Figure 142).

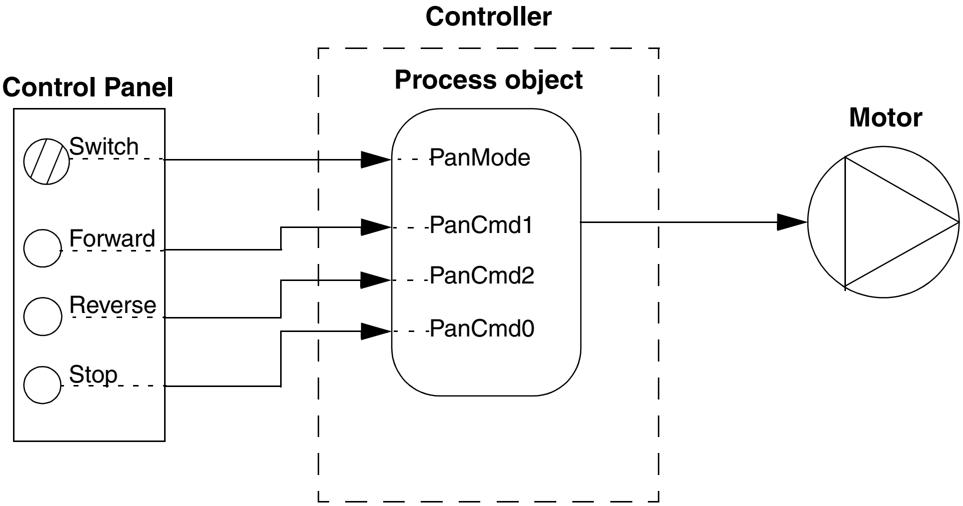


Figure 142. The Panel mode control diagram. A general solution for smaller applications, where all four Panel parameters are connected from a control panel

The *PanMode* parameter is connected via variables to interaction window. The *PanCmd1*, *PanCmd2* and *PanCmd0* parameters are still connected to the physical Start/Stop buttons. Changes in the Panel mode are therefore approved centrally, that is, a local operator must first obtain permission from the central control operator (see Figure 143).

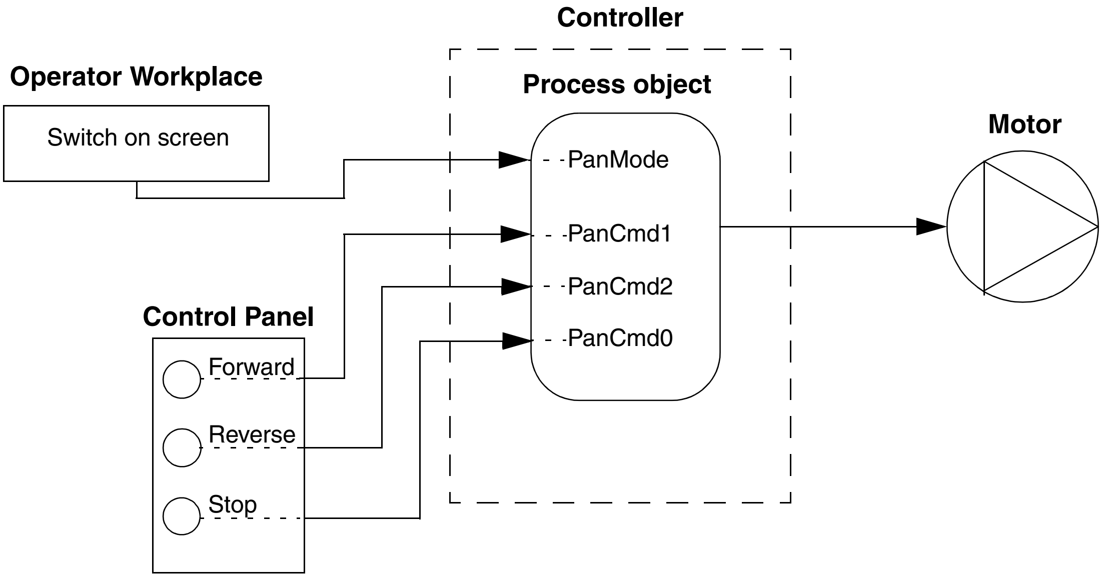


Figure 143. The Panel mode control diagram. A general solution for large applications, where the PanMode parameter is connected to an operator workplace

The status of the object is controlled by the signals *PanCmd1, PanCmd2* and *PanCmd0*, which are level detected signals and function in the same way as *AutoCmd* signals, as illustrated in Figure 143. The *PanCmd1*, *PanCmd2* and *PanCmd0* parameters should be connected to push buttons. It may be advisable to use an *R_Trig* function block (trigger parameter and push button) between *PanCmd1, PanCmd2* and *PanCmd0*, in case the push button malfunctions.

Advanced Functions

This sub section contains information about the more advanced functions of the process objects. It also contains reference information for those who need to use all functions offered by the Process Object libraries:

- Level Detection, Commands and Alarm Texts on page 317 contains information on the use of parameters for all types in the Process Object libraries.
- ABB Drives Control on page 320 explains how to configure process objects for controlling ABB Drives.

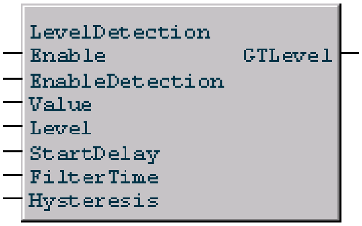
INSUM Control on page 340 describes how to configure process objects for INSUM control.

For detailed information on the use of individual parameters, beyond the contents of this manual, see online help and the Control Builder editor. To open the editor, right-click the type and select **Editor**.

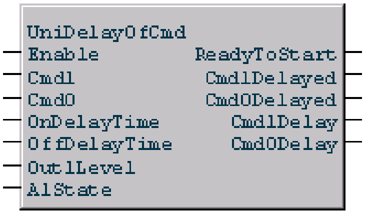
Level Detection, Commands and Alarm Texts

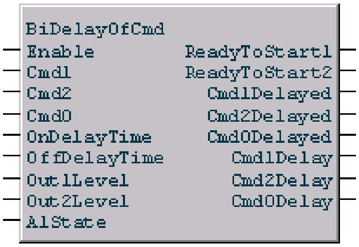
In addition to the UniCore and BiCore core objects (see Process Object Template Concept (Core Objects) on page 262), the Process Object Basic and Process Object Extended libraries also use a number of function block types that can be used to enhance the functions of the process objects:

- The LevelDetection function block is be used to supervise a signal of type real. When the in signal *Value* has been above the value of *Level* during the set *FilterTime*, *GTLevel* is set to True.



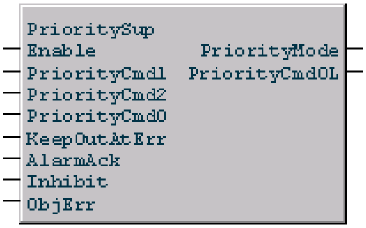
- The UniDelayOfCmd and BiDelayOfCmd function block types are used to avoid false commands in Auto mode. The command signal is delayed, to avoid repeated starts and stops.





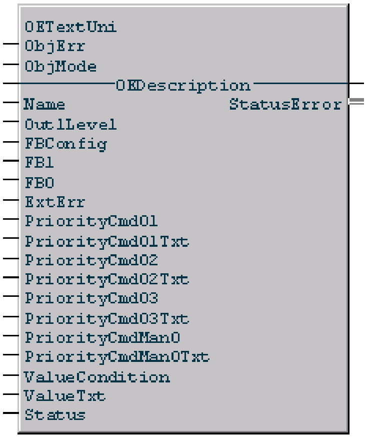
If, for example, a level detector informs an object to start, a disturbance pulse should not be able to start the object. The object shall not start until the detector delivers a reliable, constant signal stating that the start level is reached. If the object is a motor, this behavior is very important, in order not to wear it out.

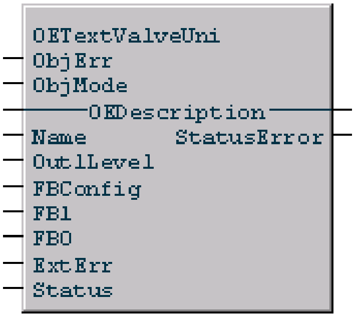
- The DriveStatusReceive and DriveCommandSend function block types are used for ABB Drives communication, see Examples on page 304.
- The PrioritySup function block type supervises the commands and sets the mode to Priority mode if any of the inputs are active. It also supervises the alarm status. Some are active and automatic priority to zero is performed, if *KeepOutAtErr* is false.

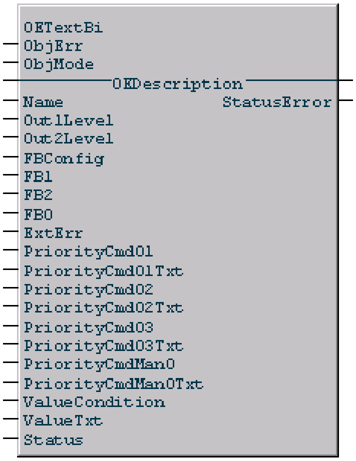


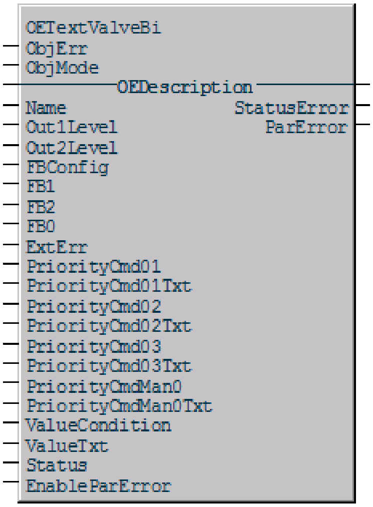
It can be used, together with the OEText function blocks, to generate error messages.

- OETextUni, OETextBi, OETextValveBi, and OETextValveUni function block types (which are available in the Process Object Extended library), DetectOverrideBi, DetectOverrideUni, DetectOverrideVoteBi, DetectOverrideVoteUni and Jog function block types (which are available in the Process Object Basic library), and ProcessObjectAE (which is available in Alarm Event library) can be used to generate error-text strings for the corresponding objects.









The following situations are taken care of and the corresponding texts are built:

– 'Name' Unit error
– 'Name' Channel error
– 'Name' Too low value

- 'Name' Too high value
- 'Name' Underflow
- 'Name' Overflow
- 'Name' Out of service
- 'Name' OE External error;
- 'Name' OE 'PriorityCmd02Txt'
- 'Name' OE 'PriorityCmd03Txt'
- 'Name' OE 'MotorValueTxt'
- 'Name' OE Out1/Out0; FB1=1/0; FB0=1/0

The OEText functions block have open code and may be copied and changed in a user-defined library.

ABB Drives Control

This section describes things you have to consider when you want to control ABB drives using process objects:

- ABB Drives Process Objects on page 321 introduces the ABB Drives Process objects that are found in the ABB Drives Process Object library.
- More information on the behavior, configuration and operation of the ABB Drives process objects can be found under Operation Modes on page 323, Drive States on page 325, Drive Speed References on page 326, Drive Torque Selector for ACStdDrive(M) on page 327, Drive Torque Selector for DCStdDrive(M) on page 328, Drive Torque Selector for EngDrive(M) on page 328, Priority and Interlocking on page 329, and Alarm and Event Handling on page 330.
- The function blocks DriveStatusReceive and DriveCommandSend are included in all ABB Drives process objects. For a description of those, see DriveStatusReceive and DriveCommandSend on page 332**.**
- ABB Drives Communication on page 335 shows how to configure communication between an AC 800M controller and ABB Drives, using the ABB Drives process objects.
- ABB Drives Interaction Windows on page 339 provides Drives-specific information on interaction windows.

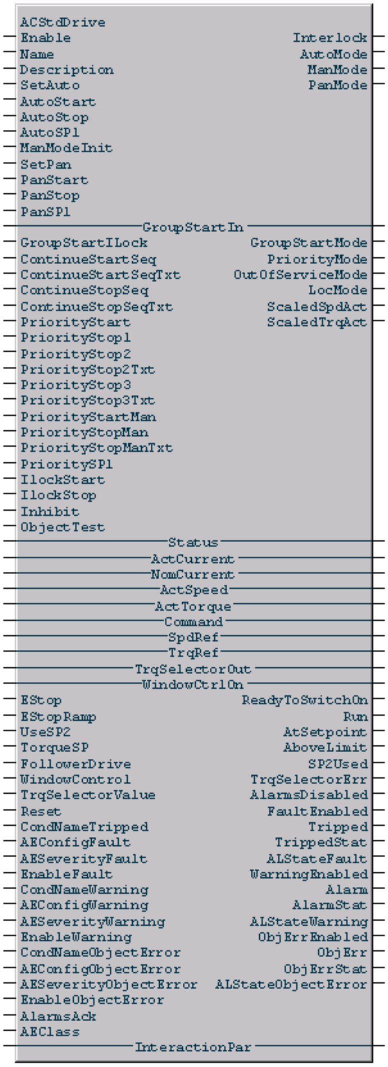
ABB Drives Process Objects

There are three ABB Drives process objects:

- **ACStdDrive(M)**

  Supports the control and supervision of ABB AC Standard Drives.

  It is based on UniCore for Process logic handling, and on DriveCommandSend and DriveStatusReceive for handling the ABB Drive interface/communication. Blocks for alarm handling are also included, for display of drive trips, drive alarms and object errors, such as communication and feedback errors from the device.

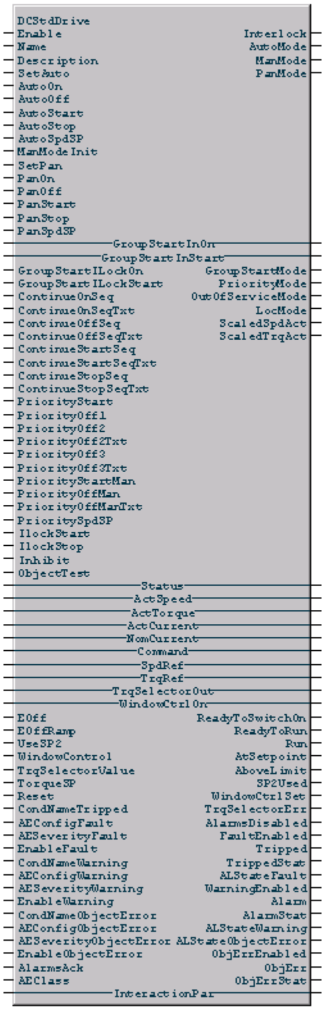


- **DCStdDrive(M)**

  Supports the control and supervision of ABB DC Standard Drives.

  It is based on UniCore for process logic handling, and on DriveCommandSend and DriveStatusReceive for handling ABB Drive interface/communication. Blocks for alarm handling are also included, for display of drive trips, drive alarms and object errors, such as communication and feedback errors from the device.

  The function of DCStdDrive(M) and ACStdDrive(M) is very similar.

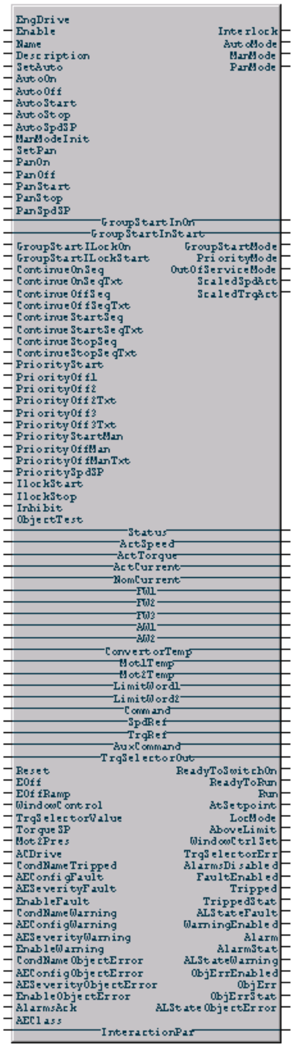


- **EngDrive(M)**

  Supports the control and supervision of ABB Engineered Drives.

  EngDrive(M) is based on UniCore for process logic handling, and on DriveCommandSend and DriveStatusReceive for handling ABB Drives interface/communication. Blocks for alarm handling are also included, for display of drive trips, drive alarms and object errors, such as communication and feedback errors from the device.

  EngDrive(M) operation modes work the same way as for DCStandardDrive(M).



Operation Modes

The *Enable* parameter is by default set to true, meaning that the function block or control module is executed. When *Enable* is False, the function block or control module is disabled, including all internal instances. All output signals are inactivated or set to zero.

ABB Drives process objects are based on UniCore, with the following operation modes.

- **Manual Mode**

  In Manual mode, the operator controls the drive from the operator station, or from interaction windows in Control Builder.

  Manual mode can be set via the interaction parameter. In Manual mode, the drive can only be started and stopped by setting the parameters *InteractionPar.Manstart* and *InteractionPar.ManStop*.

  The output parameter *ManMode* is true when the drive is in Manual mode. The speed reference is set with *InteractionPar.ManSpdRefInput*.

The default startup mode after a cold start is set via *ManModeInit*. The initial value is set to true, which means that manual mode will be the default mode.

By writing 'false' in the actual parameter column, you can change to auto mode instead. The parameter *ManModeInit* is copied to parameter *ManMode* at every cold start. Manual mode is always active when the *Enable* signal state goes from False to True.

- **Auto Mode**

  Auto mode is set by setting *InteractionPar.SetAuto* or *SetAuto* to True.

  In Auto mode, the status of the drive (start or stop) is controlled via the parameters *AutoStart* and *AutoStop*. The output parameter *AutoMode* is True when an object is in Auto mode. The speed reference is set using the parameter *AutoSP1*.

- **Panel Mode**

  Panel mode is set by setting *InteractionPar.SetPan* or *SetPan* to True.

  In Panel mode, the drive is controlled via the parameters *PanStart* and *PanStop*. The speed reference is set using the parameter *PanSP1*. The output parameter *PanMode* is active when the object is in this mode.

  Sometimes a panel connection is not available in the system. The displays and control logic for *PanMode* must not be activated. This is prevented by setting the parameter *InteractionPar.PanelExists* to False.

- **Priority Mode**

  The drive is in Priority mode whenever a priority Start or Stop interlock is active. See Priority and Interlocking on page 329.

- **Group Start Mode**

  Group Start mode is set by setting the parameter *InteractionPar.SetGroupStart*.

  In Group Start mode, the object is controlled via the parameter *GroupStartIn*. The interlocking parameter *GroupStartIlock* prevents transfer to Group Start mode. In this mode, the output parameter *GroupStartMode* is active. The speed reference in this mode is *AutoSP1*.

- **Local Mode**

  In Local mode, the drive is controlled directly from CDP312 on the drive, bypassing the controller. Local mode can only be set and reset at the Drive. In Local mode, all inputs from Auto mode and Manual mode, as well as all priority interlocks, are ignored. The object will return to the previous active mode when Local mode is disabled. The output parameter *Remote* is false when a drive is in Local mode.

- **Out of Service Mode**

  Out of Service mode is set by setting the parameter *InteractionPar.SetOos*.

  In Out Of Service Mode, the drive is stopped, and the operator controls the on the machine to which the drive belongs.

  To exit Out of Service mode, change the mode to Manual, Auto, Group Start or Panel. The output parameter *OutOfService* is active when the object is in this mode.

Drive States

The current state of a drive is available in the Status Word received from the drive (see DriveStatusReceive and DriveCommandSend on page 332). The drive can be in any one of the following states, the active state is set to true in the object output:

- *ReadyToSwitchOn*: The drive is ready to be switched On.
- *ReadyToRun*: The drive is energized and ready to run (EngDrive(M) only).
- *Run*: The drive is operating with a speed reference.
- *Fault*: The drive has tripped on an internal error.
- *Alarm*: The drive is reporting an internal warning, but it is not severe enough to stop the operation of the drive.
- *AboveLimit*: The actual frequency/speed is equal to or above supervision limit set by a parameter (for example, parameter 32.02 in ACS800).

Drive Speed References

ABB Drives objects transmit one speed reference, called *SpdRef*, to the drive.

There are the following input speed references:

- *AutoSP1* in Auto mode & Group Start mode.
- *PanSP1* in Panel mode.
- *InteractionPar.ManSpdRefInput* in Manual mode. This reference is set from the operator interface.
- *PrioritySP1* in Priority mode. This value is set to half of *InteractionPar.MaxSpeed*, in case it is set to zero in the application. This ensures that the motor runs when *PriorityStart* is true.
- *AutoSP1CC*, a control connection is available in the control modules. This reference is used in AutoMode only, if this is not connected then *AutoSP1* is used as speed reference. (this reference is valid for EngDrive(M) only.)

The correct reference, depending on the active mode, is transmitted to the drive. The speed and torque set points in the object are set in engineering units selected by the system engineer.

The string for unit of speed is entered in *InteractionPar.SpeedUnit*. The string for unit of torque is entered in *InteractionPar.TorqueUnit*.

The object scales the speed signal to the drive units using *InteractionPar.MaxSpeed* and *InteractionPar.EnableNegSpeed* and torque signals using *InteractionPar.MaxTorque* and *InteractionPar.EnableNegTorque*. See DriveStatusReceive and DriveCommandSend on page 332.

The speed set point in Manual mode is input from the operator station in the selected engineering units, for the convenience of the operator. *InteractionPar.ManSpdRefInput* is used for this.

During a mode change, the drive maintains its previous state. It is the responsibility of the new state to ensure 'bumpless' transfer of the speed reference, when the drive is running.

In Manual mode, the reference is always initialized with *ActualSpeed*.

Drive Torque Selector for ACStdDrive(M)

ACS800 drives have parameters for Master/Follower applications. The master and follower drives are connected by a fiber optic channel. The Master drive controls the start/stop of the follower drive. The object can be used to generate the desired torque reference for the follower drive.

The object and interaction window of the follower drive only display status and references. They do not control the drive in any manner, the master drive controls the follower drive.

Six options are available for setting the source of torque reference. The drive controller selects different torque references based on the input parameter *TorqueSelectorValue*.

1 -> Speed Controlled

2 -> Torque Controlled

3 -> Min (minimum logic with speed error comparison)

4 -> Max (maximum logic with speed error comparison)

5 -> Window Control (Window control mode)

6 -> Zero Control

The selection of torque reference is made using parameters *WindowControl* and *TorqueSelectorValue*. The parameters *TrqSelectorOut*, *WindowCtrlSet* and *TrqSelectorErr* are the outputs of the object.

The default mode is Speed Control. In *AutoMode*, the drive can be run in different torque selector modes. In other modes like *ManMode* and *PanMode*, only speed control mode is possible.

Drive Torque Selector for DCStdDrive(M)

The torque reference *TrqRef* is transmitted to the Drive.

DCS500B drive has a torque reference chain. This can be used to provide desired torque reference for the torque regulation. Six options are available for setting the source of torque reference. The drive controller selects different torque references based on the input parameter *TorqueSelectorValue*.

0 -> Zero Control

1 -> Speed Controlled

2 -> Torque Controlled

3 -> Min (minimum logic with speed error comparison)

4 -> Max (maximum logic with speed error comparison)

5 -> Window Control (Window control mode)

The selection of torque reference is made using parameters *WindowControl* and *TorqueSelectorValue*. The parameters *TrqSelectorOut*, *WindowCtrlSet* and *TrqSelectorErr* are the outputs of the object. The default mode is Speed Control. In Auto mode, the drive can be run in different torque selector modes. In other modes like Manual mode and Panel mode, only speed control mode is possible.

Drive Torque Selector for EngDrive(M)

An Engineered Drive has a torque reference chain. This can be made use to provide desired torque reference for the torque regulation. Six options are available for setting the source of torque reference. The drive controller selects different torque references based on the input parameter *TorqueSelectorValue*.

- 1 -> Zero Control
- 2 -> Speed Controlled
- 3 -> Torque Controlled
- 4 -> Min (minimum logic with speed error comparison)
- 5 -> Max (maximum logic with speed error comparison)
- 6 -> Window Control (Window control mode)

The selection of torque reference is made using parameters *WindowControl* and *TorqueSelectorValue*. The parameters *TrqSelectorOut*, *WindowCtrlSet* and *TrqSelectorErr* are the outputs of the object.

The default mode is Speed Control. In Auto mode, the drive can be run in different torque selector modes. In other modes like Manual mode and Panel mode, only speed control mode is possible.

Priority and Interlocking

Interlocking is used to stop process objects from entering a certain state. Priority parameters are used to force an object to a certain state. The following interlocking and priority functions are available for ABB Drives process objects:

- **Priority Start (On) Interlocks**

  The parameter *PriorityStart* forces the drive to start. The process interlock can only be overridden by the priority Stop commands and the *Inhibit* parameter.

  The speed reference when *PriorityStart* is active during *PriorityMode* is set to *PrioritySP1*.

  The parameter *PriorityStartMan* forces the drive to start and sets it to Manual mode.

- **Priority Stop (Off) Interlocks**

  There are three different parameters, which can be used for priority Stop (Off) interlocks, forcing the drive to stop (stop and switch off), *PriorityStop1*, *PriorityStop2* and *PriorityStop3 (PriorityOff1*, *PriorityOff2* and *PriorityOff3*). The priority Stop (Off) can only be overridden by the *Inhibit* parameter.

  The parameter *PriorityStopMan* (*PriorityStopMan*) forces the drive to stop and sets it to Manual mode.

- **Start/Stop Interlocks**

  The *IlockStop* and *IlockStart* are Start/Stop interlocks that prevent the drive from being manually forced to certain states. *IlockStop* blocks the drive from being stopped in *ManMode* and *IlockStart* blocks a manual start signal. *IlockStart* does not stop a drive that is already running. The *Inhibit* parameter overrides these Interlocks also.

- **Inhibit**

  The *Inhibit* parameter overrides all active interlocks when set to True.

Alarm and Event Handling

The following alarms are generated for ABB Drives objects:

- **Warnings**

  Warnings are alarms from a drive that do not trip the motor. These need to be acknowledged by operator and are time stamped in the controller.

- **Faults**

  Faults are alarms from the drive which trip the motor. These need to be acknowledged by the operator and are time stamped in the controller.

- **Object Errors**

  Object errors can be any of the following, or a combination:

  - Communication error from the drive.
  - Feedback error from the drive.
  - *PriorityStop2* or *PriorityStop3* interlock active.
  - *EmergencyStop* and *EmergencyStop* with ramp, issued to the drive.

  Details of the particular internal error that has occurred can be found in the description field in the alarm list.

Alarms are acknowledged via the input parameter, *AlarmsAck*. The operator acknowledges it in the alarm list. All alarms can be disabled by *InteractionPar.DisableAlarms*.

Both a feedback error from the drive and drive having tripped will force the object to go to state Stop (*Run* is false). EngDrive(M) will switch off, that is, *ReadyToSwitchOn* is False. That means that whenever the drive returns to normal mode, a start command will never be active until the command has been reactivated.

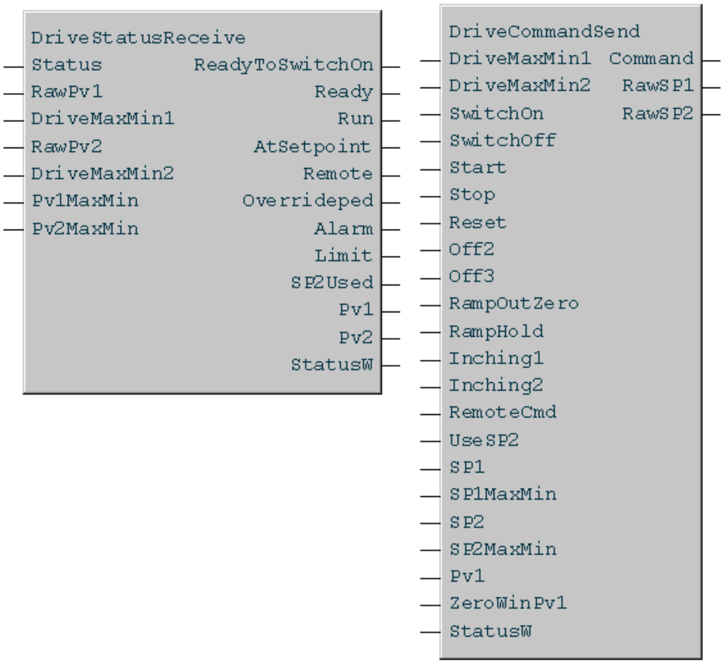
The following parameters are associated with alarm handling:

- *AEConfigX* (where X is any of the alarms above), sets the behavior when there is an active alarm.
- *AESeverityX* (where X is any of the alarms above), sets the severity of the alarm.
- *ALStateX* (where X is any of the alarms above), displays the state of the alarm.
- *AEClass*, the class which all alarms belong to.
- *Warnings*, active output when there is a Warning alarm active.
- *Trips*, active output when there is a Trip alarm active.
- *ObjErr*, active output when there is an ObjectError alarm active.

For more information on alarm and event parameters and alarm and event handling, see alarm and event information in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* , and online help for the object in question.

DriveStatusReceive and DriveCommandSend

The function blocks DriveStatusReceive and DriveCommandSend are included in the drive control types ACStdDrive(M), DCStdDrive(M), and EngDrive(M).



DriveStatusReceive are used to retrieve the status from a drive, after which the program or operator makes a decision based on this information. A command can then be issued to the drive using DriveCommandSend.

ABB Drives process object types can also be used to build custom ABB Drives control solutions. In this case, you should consider the following:

- **Execution Order**
  The execution order is always DriveStatusReceive, UniCore, DriveCommandSend. In this way, communication delays are avoided. The start parameter (*Start*) is a level signal with a hold function.
- **Status and Control Word**
  The *Command*, *RawSP1*, *RawSP2*, *Status*, *RawPV1* and *RawPV2* parameters are to be connected to the drive, independent of the media through which data is transferred. The drive communicates with the controller though drive data sets and the parameters are to be connected to the drive according toTable 35 .

The drive data set number may differ between different types of ABB Drives, see ABB Drives documentation for the drive in question.

Table 35. Drive data sets and parameters

Drive Data Set 1			Drive Data Set 2		
Data Words			Data Words		
1.1	1.2	1.3	2.1	2.2	2.3
Command	RawSP1	RawSP2	Status	RawPV1	RawPV2

- **Emergency Stop**
  There are two command parameters that can cause an emergency stop, *Off2* and *Off3*. If the application demands an emergency stop through one of these parameters, the drive stops according to local emergency settings on the drive. The priority for emergency stops is controlled by the drive. The priority of emergency stop is higher than that of the stop and start commands.

  The *Off2* and *Off3* parameters do not have any effect on the drive, if it is not in its remote state (locally controlled drive).

- **Controlling the Drive**
  There are two sets of commands that start/stop a Drive:
  - *On*/*Off*:
    In a DC Drive, *On*/*Off* switches on/off the main contactor/circuit breaker and also energizes/deenergizes the field, motor fan and drive module cooling fan. In an AC engineered Drive, the incoming DC converter bridge is switched on, and the DC bus energized.
  - *Start*/*Stop*:
    Start/*Stop* releases/blocks the pulses to the output power bridge. On the start command, the drive controllers release torque and speed references and start the motor.

  For an AC Standard Drive, a start/stop command is sufficient. The same command can be connected to Switch On and Start, Switch Off and Stop.

  When both start (Switch On) and stop (Switch Off) orders are given though the input parameters, the stop (Switch Off) order has the higher priority.

  In order to prepare a drive for start/switch on, the following must be fulfilled:
  - The drive must be configured to receive commands from the Fieldbus.
  - Communication must be running (Drive<->Controller).
  - No activated emergency stop (*Off2*, *Off3*).
  - After drive fault, the Drive must be reset before continuing operation.

- **Scaling of Drive Values**
  It is possible to scale all reference and set point values according to local settings in the drive unit. These variables are used in the Function Blocks DriveStatusReceive and DriveCommandSend.

  DriveMaxMin1 is the maximum numerical value of the first process variable, for example speed reference and actual values in a Drive. This value for speed is +/- 20000 for ACS800.

  DriveMaxMin2 is the maximum numerical value of the second process variable, for example torque reference and actual values in a Drive. This value is +/-10000 for ACS800.

  Pv1MaxMin is the maximum value of the first process variable (for example speed) in engineering value. If a processing line has the speed of 1500 m/s, this value is set to 1500. The value 1500 is then scaled using *DriveMaxMin1*.

Pv2MaxMin is the maximum value of he second process variable (for example torque) value in engineering value. Scaling is similar to *Pv1MaxMin*.

If the application requires only unipolar values (for example no negative references), then both *Pv1MaxMin* and *Pv2MaxMin* have to be limited externally. *SP1MaxMin* and *SP2MaxMin* are the corresponding set point values for speed and torque in the DrivesCommandSend.

ABB Drives Communication

ACStdDrive(M), DCStdDrive(M), and EngDrive(M) all contain the DriveStatusReceive and DriveCommandSend function blocks. For more information on those function blocks, see DriveStatusReceive and DriveCommandSend on page 332.

Figure 144 and Figure 145 shows communication between an AC 800M Controller and an ABB Standard Drive.

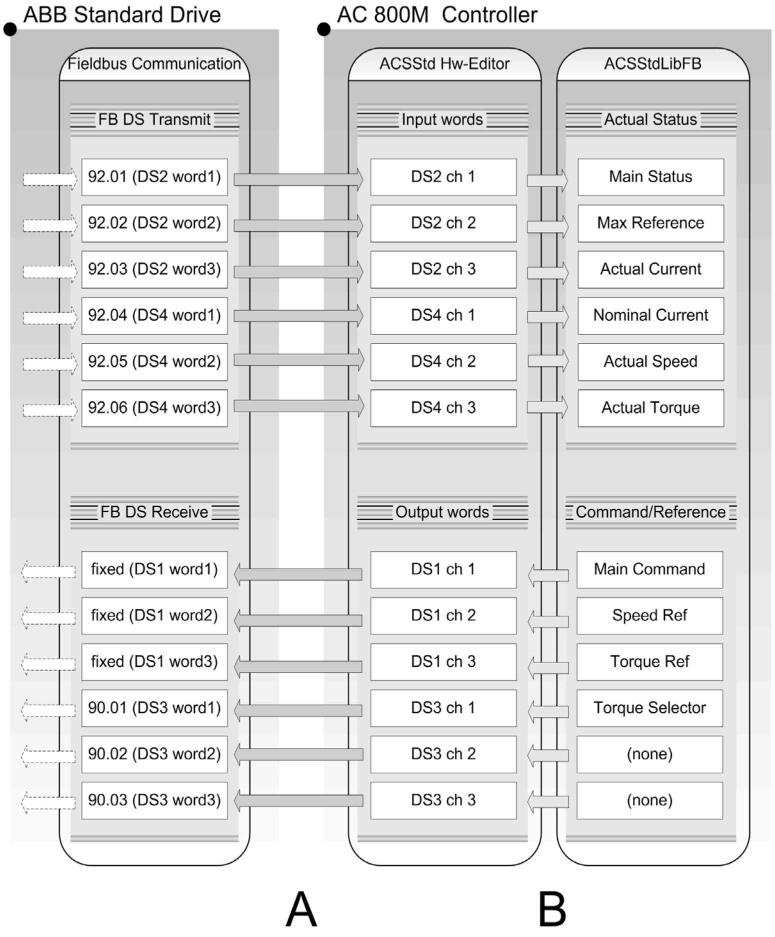


Figure 144. Overview of the connections between an ABB Standard Drive and an AC 800M Controller, part A-B

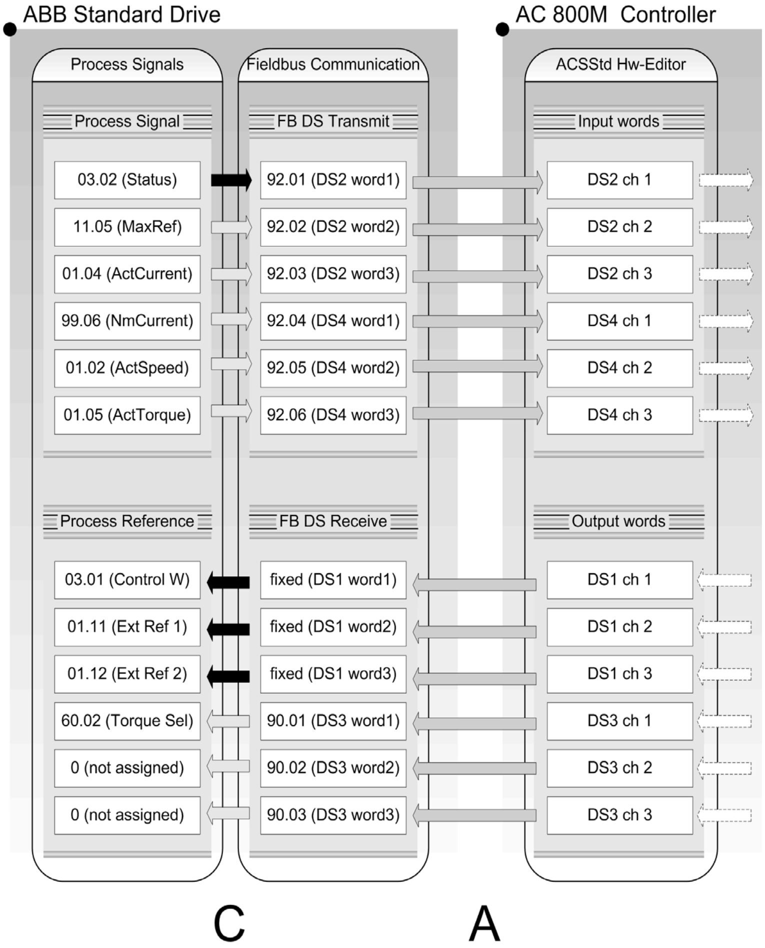


Figure 145. Overview of the connections between ABB Drive and AC 800M Controller, part C-A (black arrow = fixed connection)

To configure communication:

a. Establish the communication channels (part A in Figure 144 and Figure 145).

   The communication protocols between AC 800M and ABB Drives may be:

   - ModuleBus,

   - PROFIBUS DP-V0,

   - PROFIBUS DP-V1,

   - PROFINET IO,

   - DriveBus.

   The setting must be done either in AC 800M, or in the ABB Drive, so that they use the same protocol and define the same communication channels. For AC 800M, the setting is defined in the Hardware Editor of the applied communication interface. Refer to online help and the manual *Communication, Protocols and Design* for further information. The Hardware Editor for different protocols might use different names for the same communication channel. For ABB Drive configuration, refer to ABB Drives documentation.

b. Define the data to be sent and received in AC 800M Controller (part B in Figure 144).

   In AC 800M, this is done by making variable connections between the ACStdDrive function block and the Hardware Editor of the applied communication interface, using variable connections. The most important thing is that the required process data on the Drive side must be connected on the same communication channel used by the corresponding data in ACStdDrive function block. For example, in Figure 144, Actual Speed in ACStdDrive function block must be connected to DS4 ch 2, so that the parameter can get the correct information.

c. Define the data to be sent and received in the ABB Standard Drive (part C in Figure 145).

For example, define the connection between parameter index 01.02 (ActSpeed) with parameter index 92.05 (DS4 word 2) in ABB Standard Drive. How to connect those parameter indexes in the Drive is described in ABB Drives documentation. A black arrow in Figure 145 indicates a fixed connection that is not configurable.

The same method is also used to establish communication with EngDrive(M). The differences are only the number of communication channels and the data to be sent and received.

For newer drives (like the ACS880 series), when used with PROFIBUS DP or PROFINET IO (via FENA-11), the start trigger must be set to "Level" instead of "Trig". To do this, from the Main Menu in the drive, browse to Parameter -> Complete list -> 20 Start/stop/direction. Then, set:

- 20.01 -> Ext1 commands -> **Fieldbus A**
- 20.02 -> Ext1 Start Trigger -> **Level**

ABB Drives Interaction Windows

The Drives control types in the Process Object Drives library have four interaction Windows, the main for manual control and some supervision, one for configuring the object, one to display Process interlock and Priority signal status and one for Group Start mode. The main window is displayed first, the others can be displayed by clicking the corresponding Icons on the main window.

DCStdDrive(M) has two additional parameters that can be set in Control Builder using interaction parameters. The parameters are:

- *DriveSpeedScale*: Maximum speed of Motor in Drive Units (for example. 20000 for DCS500B).
- *DriveTorqueScale*: Nominal Torque in Drive Units (for example 4000 for DCS500B).

All ABB Drives internally scale speed to +/-20000 and torque to +/- 10000. This unit is different for DCS500B. However, future DC Drives can be expected to follow ABB Drives scaling.

INSUM Control

This section describes control INSUM devices using the process objects InsumBreaker(M), MCUBasic(M), and MCUExtended(M):

- INSUM Process Object Types on page 341 contains a short description of each of the INSUM process objects in the INSUM Process Object library.
- Additional information on the behavior, configuration and operation of the INSUM process objects can be found under Operation Modes on page 344, Circuit Breaker and MCU States on page 346, MCU Types on page 347, Motor Starter Types on page 347, Priority and Interlocking on page 348, Priority and Interlocking on page 348, Alarm and Event Handling on page 349, Supervision on page 355, and Control Builder Interaction Windows on page 357.

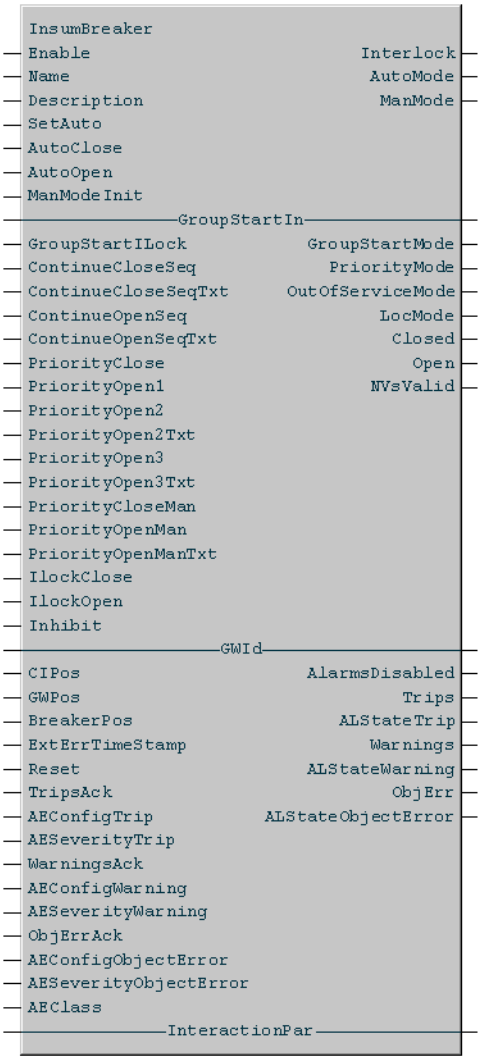
INSUM Process Object Types

There are three INSUM process object types:

- **INSUMBreaker(M)**

  Used to control and supervise an INSUM trip unit for circuit breakers.

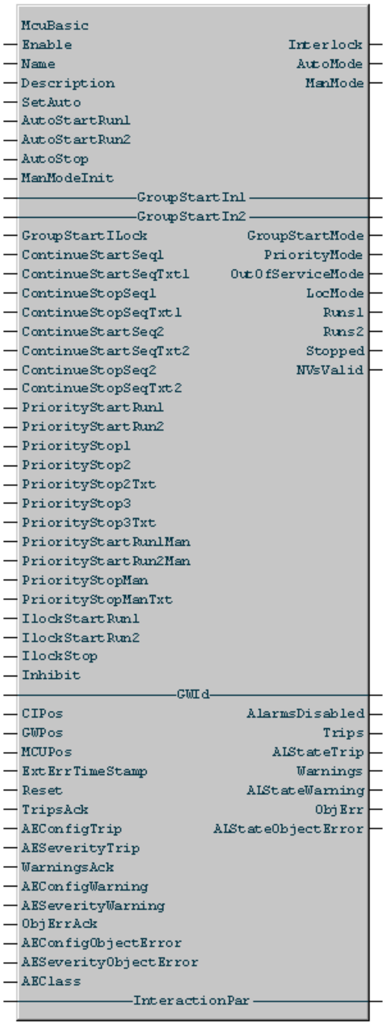
  INSUMBreaker(M) is based on UniCore for process logic and INSUMRead and INSUMWrite blocks for communication with the device. Blocks for alarm handling are also included, for the display of trips, warnings and other errors, such as communication errors and feedback errors from the device.



- **MCUBasic(M)**

  Used for control and supervise an INSUM MCU. It supports the control of two starter types, NR-DOL and REV-DOL and two MCU types, MCU1 and MCU2.

  MCUBasic(M) is based on BiCore for process logic and INSUMRead, and on INSUMWrite blocks for the communication with the device. Blocks for alarm handling are also included, for the display of trips, warnings and other errors, that is, communication errors and feedback errors from the device.

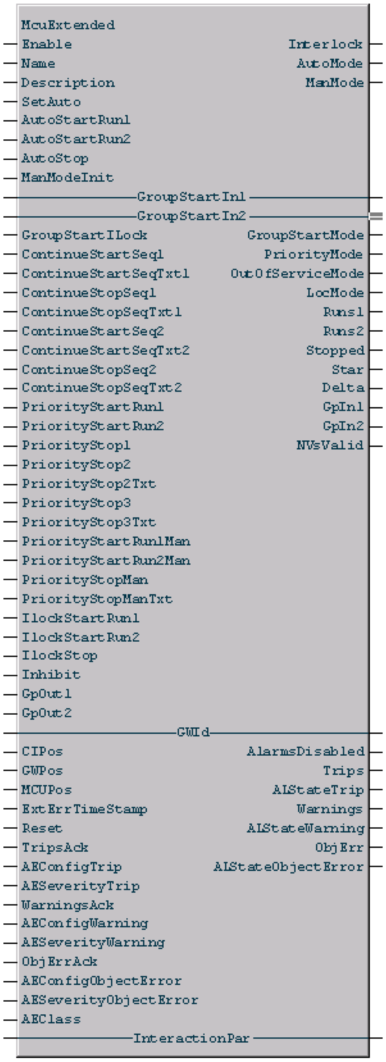


- **MCUExtended(M)**

  Used to control and supervise an INSUM MCU. It supports the control of four starter types, NR-DOL, REV-DOL, NR-SD and NR-2N.

  McuExtended(M) is developed for the control of an MCU2 and has options of reading/writing to general purpose I/Os. It is based on BiCore for process logic, and includes INSUMRead and INSUMWrite blocks for communication with the device.

  Blocks for alarm handling are also included, for the display of trips, warnings and other errors, such as communication errors and feedback errors from the device.



Operation Modes

The *Enable* parameter is by default set to True, meaning that the object is executed. When *Enable* is False, the object is disabled, including all internal types. All output parameters are inactivated or set to 0.

All three INSUM process objects have the same modes (with the exception of Soft Local mode, which is not valid for circuit breakers). There are only minor differences, which are pointed out below.

- **Manual Mode**

  The input parameter *ManModeInit* sets the default startup mode after a cold start. If this parameter is True, control will be manual (Manual mode), otherwise it will be automatic (Auto mode). When an object returns from being disabled, it is always in Manual mode.

  Manual mode is activated via *InterationPar.SetMan*. In Manual mode, the parameter *ManMode* is True.

  Open and Close commands to circuit breakers in manual mode are sent via the *InteractionPar* variables *InteractionPar.ManOpen* and *InteractionPar.ManClose*.

  Manual control of MCUs is possible via the parameters *InteractionPar.ManStartRun1*, *InteractionPar.ManStartRun2* and *InteractionPar.ManStop*.

  In Manual mode, the operator can switch to Auto, Soft Local (McuBasic(M) and MCUExtended(M) only), Out of Service, and Group Start mode. The operator is also able to reset trips from a circuit breaker or MCU (*InteractionPar.Reset*) and activate TOL bypass for MCUs (set *InteractionPar.TOLBypassActive* to True).

- **Auto Mode**

  Auto mode is activated via the parameter *SetAuto* or the interaction parameter *InteractionPar.SetAuto*. In Auto mode, the status of a circuit breaker (open or closed) is controlled by the parameters *AutoOpen* and *AutoClose,* while the status of an MCU is controlled by the parameters *AutoStartRun1*, *AutoStartRun2*, and *AutoStartStop*.

In Auto mode, active trips can be reset and it is possible to override the parameters in *InteractionPar*. The output parameter *AutoMode* is True as long as the object is in Auto mode.

- **Priority Mode**

  Priority mode is active when any of the process interlock or priority interlock parameters described in Priority and Interlocking on page 329 are True. Priority mode can be overridden by the *Inhibit* parameter, by manual commands, or by setting the circuit breaker or MCU in Local mode.

- **Group Start Mode**

  Group Start mode is activated via the *InteractionPar.SetGroupStart* parameter. In Group Start mode, the object is controlled by the parameter *GroupStartIn* (circuit breakers), or by the parameters *GroupStartIn1* and *GroupStartIn2* (MCUs). The parameter *GroupStartIlock* prevents changes to Group Start mode. In Group Start mode, the output parameter *GroupStartMode* is True.

  For more information about the Group Start library, see Section 6, Synchronized Control.

- **Local Mode**

  Local mode means that the circuit breaker or MCU is not controlled from the process object, but via direct input, bypassing the controller. Local mode can only be set or reset directly at the circuit breaker or MCU. In this mode, all input from Auto mode and Manual mode is ignored.

  When local mode is disabled, the object will return to its previous mode. The output parameter *LocMode* is True when in Local mode.

  Priority mode and priority manual commands cannot be executed in this mode.

- **External Mode**

  If you activate External mode it will provide other objects like the MMI all rights reserved of sending commands to a MCU. Thus, preventing a user (via faceplates) to interfere with an action given from the MMI. Although, an Insum object in External mode cannot send commands to a MCU, it can still read the MCU status.

  External mode is activated via the *InteractionPar.SetExternal* and is indicated in the parameter External mode.

- **Soft Local Mode (MCU control only)**

  Soft Local mode works as Local mode, but with the difference that Soft Local mode is activated from a remote point of control, for example, an interaction window. Remote control is also reset directly from the interaction window. When an MCU is in Soft Local mode, an indication is shown on the interaction window (the same indication as for Local mode, but the difference is that the push button for setting the MCU to remote control is active).

- **Out of Service Mode**

  Out of service mode is entered using the command *InteractionPar.SetOutOfService*. It is only possible to enter this mode when the status of a circuit breaker is Open (when the status of an MCU is Stopped). In this mode, it is not possible to maneuver the circuit breaker or MCU, but it is possible to change modes and exit Out of service mode. The output parameter *OutOfServiceMode* is active when in this mode.

Circuit Breaker and MCU States

The current state of a circuit breaker, Open or Closed, is shown by the output parameters *Open* and *Closed,* respectively. More detailed information is displayed at the interaction window.

The current state of the MCU is shown by the output parameters *Runs1*, *Runs2* and *Stopped*. *Runs1* and *Runs2* basically mean that the MCU is in any of its running states, and *Stopped* that the MCU is stopped. Changing states for an MCU is handled via its associated commands. More detailed information is displayed in the interaction window.

MCUExtended(M) has two additional states: Star and Delta. The *Star* and *Delta* output parameters have different meaning depending on the motor starter type used, see Motor Starter Types on page 347.

MCU Types

As McuBasic(M) supports the use of both MCU1 and MCU2, the configuration in the object must indicate which one is being used (via *InteractionPar.MCU*) as some functionality provided by McuBasic only is available for MCU2 types. This is not necessary for McuExtended(M), since this type is intended for use with MCU2 only.

McuBasic(M) supports the object is made via *InteractionPar.StarterType* (0 = NR-DOL, 1=REV-DOL). As NR-DOL is a starter type that only activates two states (Runs1 and Stopped), all commands and input parameters associated with output parameter *Runs2* do not have any function for this starter type, that is, if the *InteractionPar.StarterType* is set to 0, activating *AutoStartRun2* will not affect the object in any way.

Motor Starter Types

Table 36 shows supported motor starter types.

Table 36. Supported motor starter types. X=supported, -=not supported

	NR-DOL	REV-DOL	NR-SD	NR-2N
MCUBasic(M)	X	X	-	-
MCUExtended(M)	X	X	X	X

The motor starter type is selected via *InteractionPar.StarterType* (0 = NR-DOL, 1=REV-DOL, 2=NR-SD and 3=NR-2N):

- NR-DOL only activates two states (Runs1 and Stopped). Parameters associated with *Runs2* do not have any effect for this motor starter type.
- For NR-SD, all commands and input parameters associated with *Runs2* have no function (as for NR-DOL). For this starter type, output *Star* means that the starter is star connected and, accordingly, output *Delta* means that the starter is delta connected.
- For NR-2N, the commands and input parameters associated with *Runs2*, that is, *AutoStartRun2*, *PriorityStartRun2*, *GroupStartIn2* etc., puts the starter in its high-speed state. However, *Runs2* itself has no function for this configuration. Instead, an active output *Runs1* indicates that the MCU is in its running state

and the speed is indicated via the outputs *Star* and *Delta,* where *Star* is the low-speed and *Delta* the high-speed indication.

Priority and Interlocking

Interlocking is used to stop process objects from entering a certain state. Priority parameters are used to force an object to a certain state. The following interlocking and priority functions are available for INSUM process objects:

- **Process Interlocks**

  Setting the parameter *PriorityClose* to True forces a circuit breaker to close. For MCUs, the parameters *PriorityStartRun1* and *PriorityStartRun2* forces the MCU to go to state Runs1 and Runs2, respectively.

  Process interlocks can only be overridden by Priority interlocks, the *Inhibit* parameter and by manual commands.

- **Priority Interlocks**

  For circuit breakers, the parameters *PriorityOpen1*, *PriorityOpen2* and *PriorityOpen3* can be used to force the circuit breaker to open. For MCUs, the parameters *PriorityStop1*, *PriorityStop2* and *PriorityStop3* can be used to force the MCU to stop.

  PriorityOpen2, *PriorityOpen3, PriorityStop2* and *PriorityStop3* all generate an alarm (ObjectError) when activated.

  Priority Interlocks can only be overridden by the *Inhibit* parameter and manual commands.

- **Priority Manual Commands**

  For circuit breakers, *PriorityCloseMan* and *PriorityOpenMan* force the circuit breaker into the corresponding state (Closed or Open), and sets the process object in Manual mode. For MCUs, *PriorityStartRun1Man*, *PriorityStartRun2Man* and *PriorityStopMan* force the MCU into the corresponding state and then sets the process object in Manual mode. *PriorityCloseMan* and *PriorityStopMan* also generate an alarm.

  Priority manual commands have no function when a circuit breaker or MCU is in Local mode or Priority mode.

If priority manual commands are executed while in Priority mode, an MCU will change to the corresponding state, but the change to Manual mode will not take place until Priority mode is left (that is, all Priority commands are inactivated or a possible feedback error is acknowledged).

- **Open/Close and Start/Stop Interlocks**

  For circuit breakers, *IlockClose* and *IlockOpen* are prevent circuit breakers from being transferred to certain states manually. *IlockClose* blocks the circuit breaker from manual closure and *IlockOpen* blocks manual open signals.

  IlockClose does not open an already closed circuit breaker.

  For MCUs, *IlockStartRun1*, *IlockStartRun2* and *IlockStop* prevent the MCU from being transferred to certain states manually. *IlockStartRun1* blocks the MCU from manual start to Run1 state, *IlockStartRun2* blocks manual start to Run2 state, and *IlockStop* blocks a manual stop command. *IlockStartRun1* and *IlockStartRun2* do not stop MCUs that are already in a running state.

- **Inhibit**

  Setting the *Inhibit* parameter to True overrides all active interlocks. *Ilock* parameters prevent both manual control and control in Auto mode.

Alarm and Event Handling

The following alarms can be generated for INSUM objects:

- **Warnings**

  Alarms from a circuit breaker or MCU can be warnings, for example, alarms that do not trip the breaker or MCU. Warnings are acknowledged by the operator in the operator workplace alarm list, or by activating the parameter *WarningsAck* at the object. This alarm is time-stamped in the controller.

- **Trips**

  Trips are alarms that indicate that a circuit breaker or motor has tripped. If the parameter *ExtErrTimeStamp* is True, trips might also be a warning with an external time stamp from the breaker or MCU. Which type of alarm a trip is can be seen in the description field in the operator workplace alarm list, where all warning and trip details are displayed. A trip is acknowledged by the operator in the alarm list, or by activating the parameter *TripsAck* at the object.

- **Object Errors**

  Object errors can be any of the following, or a combination:

  – Communication read error in an INSUMRead function block.

  – Communication write error in an INSUMWrite function block.

  – Feedback error from the circuit breaker or MCU.

  – Any of *PriorityOpen2* or *PriorityOpen3* is True.

  – *PriorityCloseMan* or *PriorityStopMan* generates an alarm

  More information about which ObjectError has occurred is displayed in the description field in the alarm list. As with other alarms, ObjectError can be acknowledged from the alarm list or from the object (via *ObjErrAck*).

Both a feedback error from the device and the circuit breaker having tripped will force the breaker/MCU to go to state Open/Stopped and the object will go to Priority mode. That means that whenever the circuit breaker or MCU returns to normal mode, a start command will not become active until the command has been re-activated.

Whether alarms should be time-stamped in the circuit breaker or MCU, or not, is controlled by the parameter *ExtErrTimeStamp*. If *ExtErrTimeStamp* is True, the inputs *CIPos* and *GWPos* must be set in order to get the alarms from the correct circuit breaker or MCU. Setting this parameter disables the Warning alarm, since both warnings and trips will be included in the alarm Trip, as described above.

It is possible to acknowledge alarms from Control Builder interaction windows. To open an interaction window, click on the relevant alarm triangle icon in the interaction window.

The following parameters are associated with alarm handling:

- *AEConfigX* (where X is any of the alarms above), sets the behavior when there is an active alarm.
- *AESeverityX* (where X is any of the alarms above), sets the severity of the alarm.
- *ALStateX* (where X is any of the alarms above), displays the state of the alarm.
- *AEClass*, sets the class to which an alarm belongs.
- *Warnings*, active output when there is a Warning alarm active.
- WarningsStat, same as Warnings but can be disabled,
- *Trips*, active output when there is a Trip alarm active.

- TripsStat, same as Trips but can be disabled.
- *ObjErr*, active output when there is an ObjectError alarm active.
- ObjErrStat, same as ObjErr but can be disabled.

For more information on alarm and event parameters and alarm and event handling, see alarm and event information in the manual *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* , and online help for the object in question.

All alarms can be disabled by activating the configuration parameter *InteractionPar.DisableAlarms*.

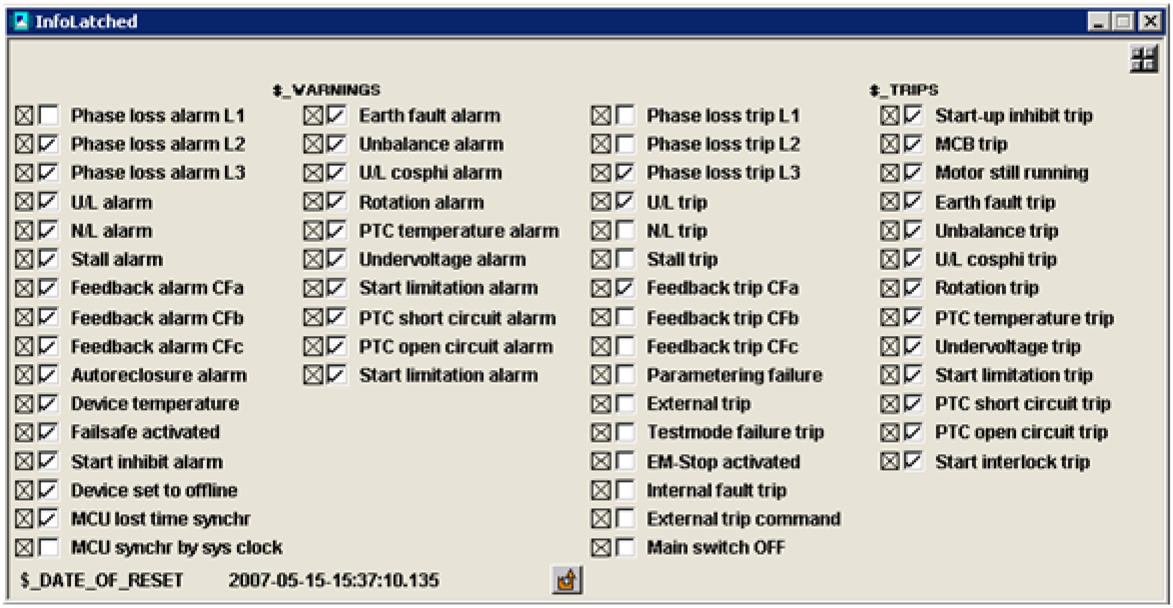
- **Alarm Handling**

A time stamp exists in the maintenance tab in the interaction window which displays the date and time when the trips and warnings latches were cleared/reset the last time.

You need to define which trips and warnings that will produce an alarm, by checking the check boxes for the trips and warnings of interest.

This example will explain how to handle trips and warnings, same logic applies to the objects McuExtended(M) and InsumBreaker(M).

The trips "Phase loss trip L3", "U/L trip" and "Feedback trips Cfa" will generate an alarm the other trips will not. See Figure 146



☒ □ Phase loss alarm L1
☒ ☑ Phase loss alarm L2
☒ ☑ Phase loss alarm L3
☒ ☑ U/L alarm
☒ ☑ N/L alarm
☒ ☑ Stall alarm
☒ ☑ Feedback alarm CFa
☒ ☑ Feedback alarm CFb
☒ ☑ Feedback alarm CFc
☒ ☑ Autoreclosure alarm
☒ ☑ Device temperature
☒ ☑ Failsafe activated
☒ ☑ Start inhibit alarm
☒ ☑ Device set to offline
☒ ☑ MCU lost time synchr
☒ □ MCU synchr by sys clock

$_WARNINGS

☒ ☑ Earth fault alarm
☒ ☑ Unbalance alarm
☒ ☑ U/L cosphi alarm
☒ ☑ Rotation alarm
☒ ☑ PTC temperature alarm
☒ ☑ Undervoltage alarm
☒ ☑ Start limitation alarm
☒ ☑ PTC short circuit alarm
☒ ☑ PTC open circuit alarm
☒ ☑ Start limitation alarm

☒ □ Phase loss trip L1
☒ □ Phase loss trip L2
☒ ☑ Phase loss trip L3
☒ ☑ U/L trip
☒ □ N/L trip
☒ □ Stall trip
☒ ☑ Feedback trip CFa
☒ □ Feedback trip CFb
☒ □ Feedback trip CFc
☒ □ Parametering failure
☒ □ External trip
☒ □ Testmode failure trip
☒ □ EM-Stop activated
☒ □ Internal fault trip
☒ □ External trip command
☒ □ Main switch OFF

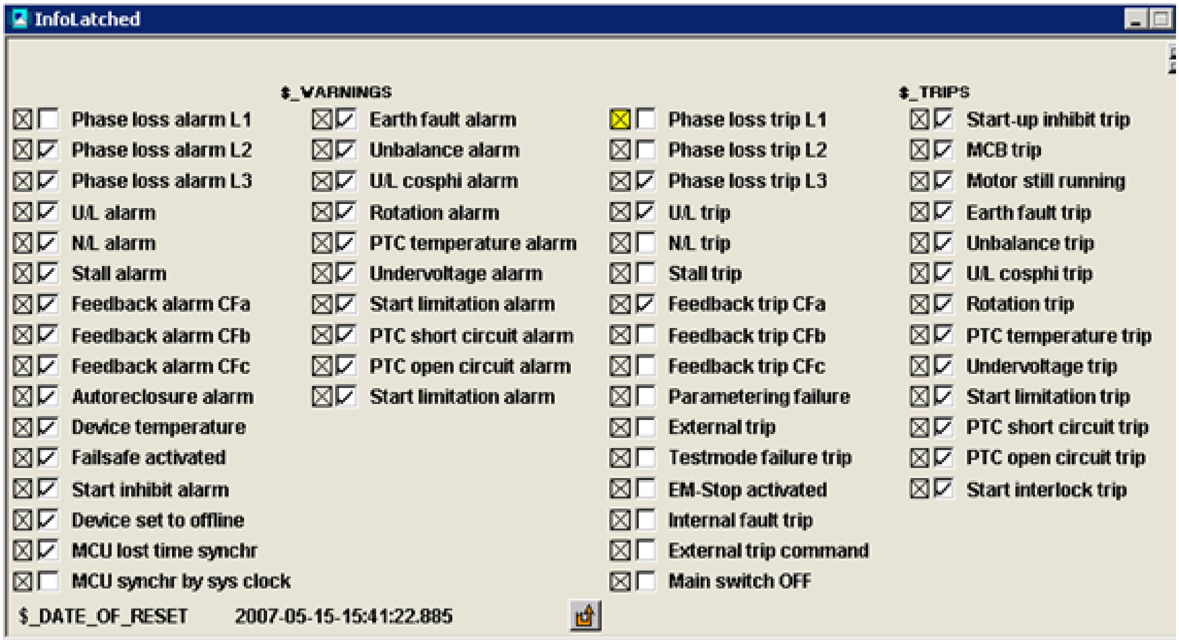
$_TRIPS

☒ ☑ Start-up inhibit trip
☒ ☑ MCB trip
☒ ☑ Motor still running
☒ ☑ Earth fault trip
☒ ☑ Unbalance trip
☒ ☑ U/L cosphi trip
☒ ☑ Rotation trip
☒ ☑ PTC temperature trip
☒ ☑ Undervoltage trip
☒ ☑ Start limitation trip
☒ ☑ PTC short circuit trip
☒ ☑ PTC open circuit trip
☒ ☑ Start interlock trip

$_DATE_OF_RESET 2007-05-15-15:37:10.135

Figure 146. MCUExtended Main Interaction Window- Alarms

In this example, for Phase loss trip L1, a trip will be present but the object will not generate an alarm, the trip is indicated as a warning or event (yellow). See Figure 147.



InfoLatched

☒ □ Phase loss alarm L1

☒ ☑ Phase loss alarm L2

☒ ☑ Phase loss alarm L3

☒ ☑ U/L alarm

☒ ☑ N/L alarm

☒ ☑ Stall alarm

☒ ☑ Feedback alarm CFa

☒ ☑ Feedback alarm CFb

☒ ☑ Feedback alarm CFc

☒ ☑ Autoreclosure alarm

☒ ☑ Device temperature

☒ ☑ Failsafe activated

☒ ☑ Start inhibit alarm

☒ ☑ Device set to offline

☒ ☑ MCU lost time synchr

☒ □ MCU synchr by sys clock

$_VARNINGS

☒ ☑ Earth fault alarm

☒ ☑ Unbalance alarm

☒ ☑ U/L cosphi alarm

☒ ☑ Rotation alarm

☒ ☑ PTC temperature alarm

☒ ☑ Undervoltage alarm

☒ ☑ Start limitation alarm

☒ ☑ PTC short circuit alarm

☒ ☑ PTC open circuit alarm

☒ ☑ Start limitation alarm

☒ □ Phase loss trip L1

☒ □ Phase loss trip L2

☒ ☑ Phase loss trip L3

☒ ☑ U/L trip

☒ □ N/L trip

☒ □ Stall trip

☒ ☑ Feedback trip CFa

☒ □ Feedback trip CFb

☒ □ Feedback trip CFc

☒ □ Parametering failure

☒ □ External trip

☒ □ Testmode failure trip

☒ □ EM-Stop activated

☒ □ Internal fault trip

☒ □ External trip command

☒ □ Main switch OFF

$_TRIPS

☒ ☑ Start-up inhibit trip

☒ ☑ MCB trip

☒ ☑ Motor still running

☒ ☑ Earth fault trip

☒ ☑ Unbalance trip

☒ ☑ U/L cosphi trip

☒ ☑ Rotation trip

☒ ☑ PTC temperature trip

☒ ☑ Undervoltage trip

☒ ☑ Start limitation trip

☒ ☑ PTC short circuit trip

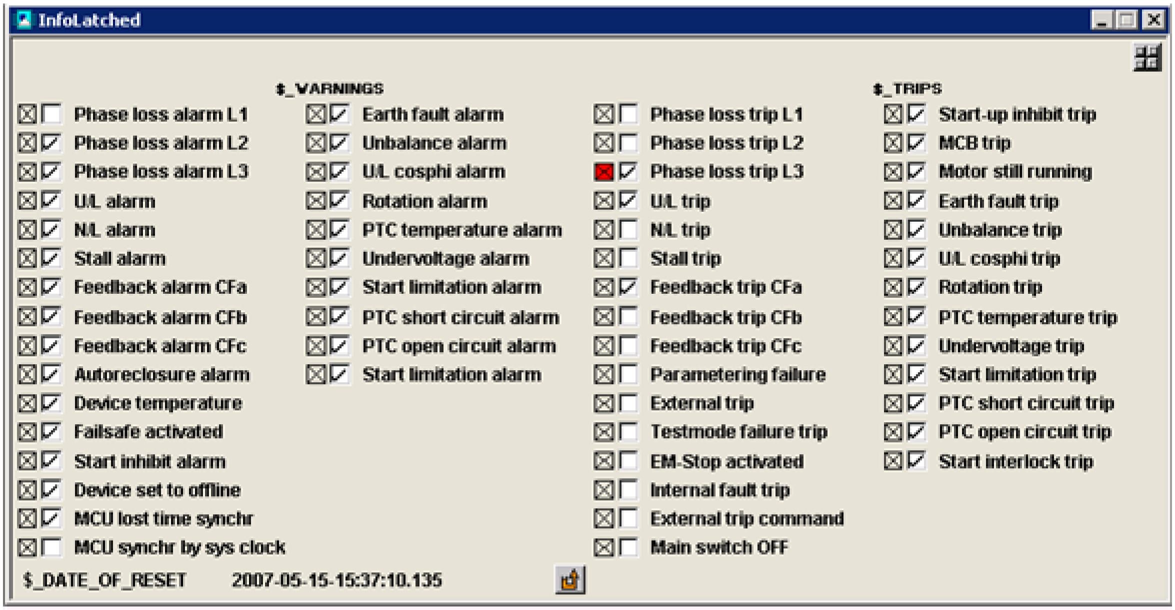
☒ ☑ PTC open circuit trip

☒ ☑ Start interlock trip

$_DATE_OF_RESET 2007-05-15-15:41:22.885

Figure 147. MCUExtended Main Interaction Window - Trips

When a trip is present which generates an alarm, the alarm producing trip is indicated in red color. See Figure 148. Also notice the alarm icon in the upper right corner of the interaction window, the alarm icon now indicates that there is an alarm active.



InfoLatched

☒ □ Phase loss alarm L1
☒ ☑ Phase loss alarm L2
☒ ☑ Phase loss alarm L3
☒ ☑ U/L alarm
☒ ☑ N/L alarm
☒ ☑ Stall alarm
☒ ☑ Feedback alarm CFa
☒ ☑ Feedback alarm CFb
☒ ☑ Feedback alarm CFc
☒ ☑ Autoreclosure alarm
☒ ☑ Device temperature
☒ ☑ Failsafe activated
☒ ☑ Start inhibit alarm
☒ ☑ Device set to offline
☒ ☑ MCU lost time synchr
☒ □ MCU synchr by sys clock

$_VARNINGS

☒ ☑ Earth fault alarm
☒ ☑ Unbalance alarm
☒ ☑ U/L cosphi alarm
☒ ☑ Rotation alarm
☒ ☑ PTC temperature alarm
☒ ☑ Undervoltage alarm
☒ ☑ Start limitation alarm
☒ ☑ PTC short circuit alarm
☒ ☑ PTC open circuit alarm
☒ ☑ Start limitation alarm

☒ □ Phase loss trip L1
☒ □ Phase loss trip L2
☒ ☑ Phase loss trip L3
☒ ☑ U/L trip
☒ □ N/L trip
☒ □ Stall trip
☒ ☑ Feedback trip CFa
☒ □ Feedback trip CFb
☒ □ Feedback trip CFc
☒ □ Parametering failure
☒ □ External trip
☒ □ Testmode failure trip
☒ □ EM-Stop activated
☒ □ Internal fault trip
☒ □ External trip command
☒ □ Main switch OFF

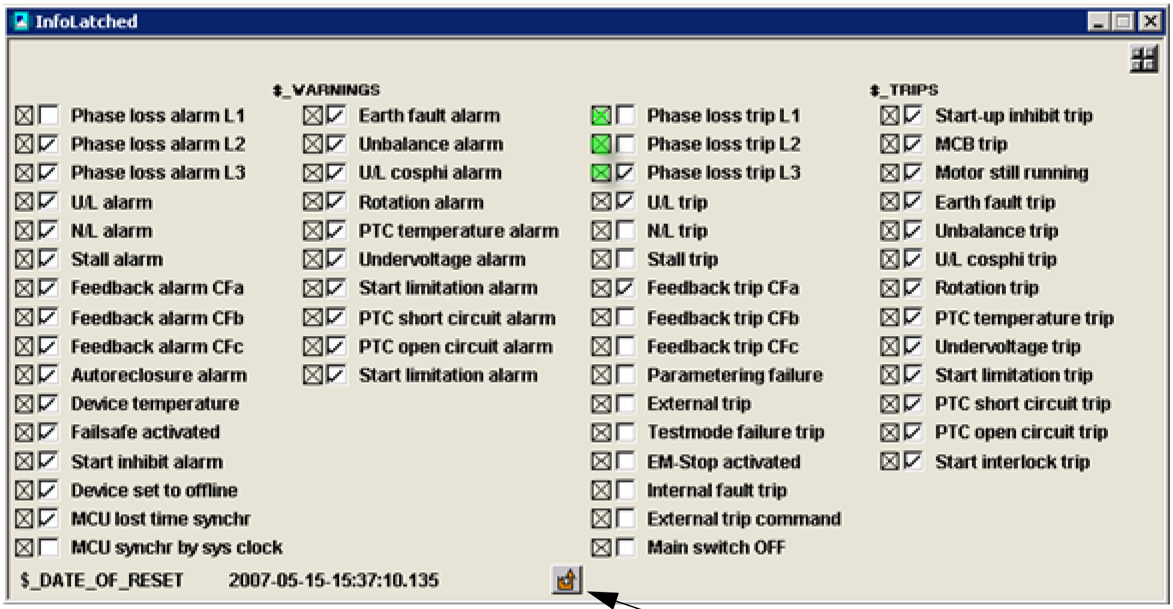
$_TRIPS

☒ ☑ Start-up inhibit trip
☒ ☑ MCB trip
☒ ☑ Motor still running
☒ ☑ Earth fault trip
☒ ☑ Unbalance trip
☒ ☑ U/L cosphi trip
☒ ☑ Rotation trip
☒ ☑ PTC temperature trip
☒ ☑ Undervoltage trip
☒ ☑ Start limitation trip
☒ ☑ PTC short circuit trip
☒ ☑ PTC open circuit trip
☒ ☑ Start interlock trip

$_DATE_OF_RESET 2007-05-15-15:37:10.135

Figure 148. MCUExtended Main Interaction Window- Alarm Indication

When there have been trips but the trips are no longer active then the interaction window indicates a green color, See Figure 149. These are Latched Trips. The latched trips can be reset by pressing the reset latches button.



InfoLatched

$_VARNINGS

☒ ☐ Phase loss alarm L1
☒ ☑ Phase loss alarm L2
☒ ☑ Phase loss alarm L3
☒ ☑ U/L alarm
☒ ☑ N/L alarm
☒ ☑ Stall alarm
☒ ☑ Feedback alarm CFa
☒ ☑ Feedback alarm CFb
☒ ☑ Feedback alarm CFc
☒ ☑ Autoreclosure alarm
☒ ☑ Device temperature
☒ ☑ Failsafe activated
☒ ☑ Start inhibit alarm
☒ ☑ Device set to offline
☒ ☑ MCU lost time synchr
☒ ☐ MCU synchr by sys clock

☒ ☑ Earth fault alarm
☒ ☑ Unbalance alarm
☒ ☑ U/L cosphi alarm
☒ ☑ Rotation alarm
☒ ☑ PTC temperature alarm
☒ ☑ Undervoltage alarm
☒ ☑ Start limitation alarm
☒ ☑ PTC short circuit alarm
☒ ☑ PTC open circuit alarm
☒ ☑ Start limitation alarm

$_TRIPS

☒ ☐ Phase loss trip L1
☒ ☐ Phase loss trip L2
☒ ☑ Phase loss trip L3
☒ ☑ U/L trip
☒ ☐ N/L trip
☒ ☐ Stall trip
☒ ☑ Feedback trip CFa
☒ ☐ Feedback trip CFb
☒ ☐ Feedback trip CFc
☒ ☐ Parametering failure
☒ ☐ External trip
☒ ☐ Testmode failure trip
☒ ☐ EM-Stop activated
☒ ☐ Internal fault trip
☒ ☐ External trip command
☒ ☐ Main switch OFF

☒ ☑ Start-up inhibit trip
☒ ☑ MCB trip
☒ ☑ Motor still running
☒ ☑ Earth fault trip
☒ ☑ Unbalance trip
☒ ☑ U/L cosphi trip
☒ ☑ Rotation trip
☒ ☑ PTC temperature trip
☒ ☑ Undervoltage trip
☒ ☑ Start limitation trip
☒ ☑ PTC short circuit trip
☒ ☑ PTC open circuit trip
☒ ☑ Start interlock trip

$_DATE_OF_RESET 2007-05-15-15:37:10.135

Reset Button

Figure 149. MCUExtended Main Interaction Window- Latched Trips

Supervision

The following NVs are read to the object from InsumBreaker(M):

- *NV NodeStatus,*
- *NV CurrentReport,*
- *NV AlarmReport.*

NodeStatus and *AlarmReport* are read every execution cycle, while *CurrentReport* is read with a cyclic interval that is set by the parameter *InteractionPar.ProcessDataScanTime*.

The following MCU NVs are read to the object from McuBasic(M):

- *NV CurrentReport*
- *NV CalcProcValue*
- *NV TimeToReset*
- *NV TimeToTrip*
- *NV CumRunT*
- *NV OpCount1*
- *NV OpCount2*
- *NV OpCount3*
- *NV AlarmReport*
- *NV MotorStateExt*
- *NV ActualCA1*

MotorStateExt, *AlarmReport* and *ActualCA1* are read every execution cycle, while the others are updated with a cyclic time interval set by the parameter *InteractionPar.ProcessDataScanTime*.

The following MCU NVs are read to the McuExtended(M) object:

- *NV CurrentReport*
- *NV VoltageReport* (optional)
- *NV PowerReport*
- *NV CalcProcValue*
- *NV TimeToReset*
- *NV TimeToTrip*
- *NV CumRunT*
- *NV OpCount1*
- *NV OpCount2*
- *NV OpCount3*
- *NV GpIn1*
- *NV GpIn2*
- *NV GpOut1Fb*
- *NV GpOut2Fb*
- *NV AlarmReport*
- *NV MotorStateExt*
- *NV ActualCA1*

MotorStateExt, *AlarmReport*, *ActualCA1*, *GpIn1*, *GpIn2*, *GpOut1Fb* and *GpOut2Fb* are read every execution cycle, while the others are updated with a cyclic time interval set by the *InteractionPar.ProcessDataScanTime* parameter.

Control Builder Interaction Windows

All three types in the Process Object INSUM library has five interaction windows: a main window for manual control and some supervision, one for extended supervision, one for configuring the object, one for group start, and one for alarm reset/latching. The main interaction window is displayed first, and the others are displayed by clicking the corresponding icons at the top of the main INSUM Control window.

Section 6 Synchronized Control

Introduction

Group Start library can be used to build procedures for starting and stopping processes. The library is intended to be used together with objects from the Process Object Basic, Process Object Extended, Project Object INSUM, and Project Object Drive libraries.

The Group Start library is based on a control module philosophy. The stop sequence is always the reverse of the start sequence. This makes the Group Start library suitable for creating start and stop procedures for machines. Group Start supports several initial head control modules.

For more information about the Function block types and Control module types see the online help.

Group Start Library

The Group Start library (GroupStartLib) contains objects to control and supervise the sequential startup of process objects as well as other units that may be seen as process objects.

The GroupStartLib organizes the control modules needed to build Group Start applications. One exception is the GroupStartObjConn control module, which is placed in BasicLib in order to avoid unnecessary dependencies between the process object libraries and the GroupStartLib.

Group Start Concept

By using the Group Start library in the Control Builder, it is easy to build complex configurations to handle the start and stop of objects. The user gets an overview of the connected object, and the object status and control is centralized to the head of the library. The information about the object being started as well as the next object to be started is shown to the user.

The starts can be build hierarchically (one after the other). The hierarchy can be made at several levels across many objects.

The connected objects may either be standard process objects (from the ProcessObj libraries) or complete control loops that have an on/off situation associated with them. In a control loop, the On/Off switch may consist of a set point change from one value to another.

Object administration using standby objects is also implemented using GroupStartLib, to make sure that a given number of objects are running. This means if one of the objects stops by any reason, another one starts up to recover the lost capacity.

A build in switch between the objects makes it possible to have all the objects running in a predefined sequence, which is set up by a maximum time interval for each object.

Group Start Configuration

The configuration of the Group Start library can be done using control modules with graphical connections. This method simplifies and reduces the need to declare local variables.

To get the Group Start configuration working, only the names of the objects have to be connected. There are no default names, as the alarms may not work properly if the connected names are unique.

A template configuration, placed in a separate library called GroupStartExampleLib, shows the configuration possibilities. The other libraries needed for the configuration are the ProcessObjectLib (if any connected object is a motor or valve) and ControlLib (if any control loop is a group start member). The connection to process objects is build on a connection control module placed in BasicLib.

Example for Template Configuration

This template configuration example is part of GroupStartExampleLib.

The characteristics of this template configuration example are:

- This configuration consists of a configuration layout, which has no connection to any kind of real application.
- The process objects are replaced by the test control module.
- There are no parameters present in this configuration.
- There is no connection to the I/O system. If connections are to be made, some variables must be defined in the template module that are connected to the I/O system.
- The template has primarily two group starts defined, and consists of two other sub groups looked upon and connected as ordinary objects. The connected leaf objects are of test type. In a real application, these can be exchanged with ordinary process objects of any kind.
- This example has a GroupStartOr control module implemented. The bottom right group looked upon as an object can be started from the upper left OR right group start.

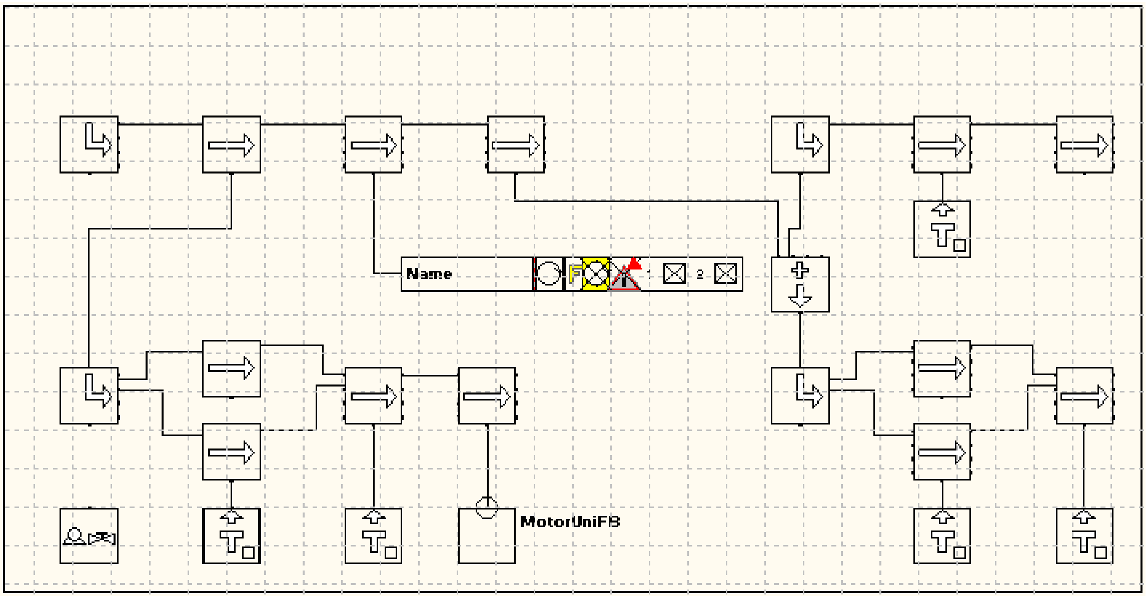


Figure 150. Template Configuration Example

The template has primarily two group starts defined, and consists of two other sub groups connected as ordinary objects. The connected leaf objects are of test type, and in a real application can be exchanged with ordinary process objects of any kind. This example has a GroupStartOr control module implemented. The bottom right group looked upon as an object and it will be started from the upper left OR right group start. See Figure 151.

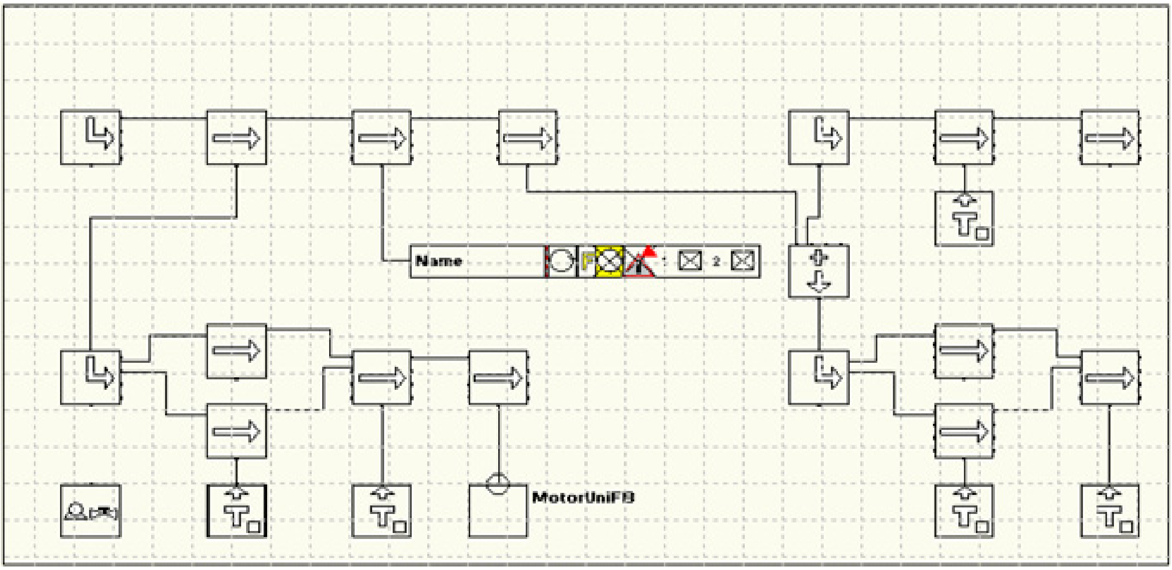


Figure 151. Template Configuration Example

Group Start Objects

The following objects are present in the GroupStartLib:

- GroupStartHead
  GroupStartHead supervises the entire group, keeps track of the alarms generated in the group and detects the connected objects not ready for start in group start mode.
- GroupStartStep
  GroupStartStep is used to define a step in a Group Start sequence.
- GroupStartAnd
  GroupStartAnd can be used when two or more Group Start groups need to be synchronized before a Group Start object/sub-group is started.

- GroupStartOr
  GroupStartOr can be used when a Group Start object/sub-group is started from two or more Group Start groups.
- GroupStartObjectTemplate
  GroupStartObjectTemplate is used to connect a generic process object to the group start, using the structured data type *ProcObjConnection*.
- GroupStartTestObject
  GroupStartTestObject can be used during design / commissioning to test the Group Start before all process objects are in place.
- GroupStartStandby4, GroupStartStandby8, GroupStartStandby12
  These objects can be used for standby purposes and to activate a desired number of objects, all working together in the process. Refer GroupStartStandby Object on page 365 for details.
- InfoParGroupStartObjectTemplate
  This control module type contains the graphics of the interaction window of the GroupStartObjectTemplate type.

GroupStartObjectTemplate

The GroupStartObjectTemplate control module is used to represent the connected device in the group start environment. This control module can be used with interaction windows to supervise the connection to the group and to enter start and stop delay times, and also to enter user defined interaction data. This control module encapsulates the generic connection control module.

The GroupStartObjectTemplate, which supports the structured data type *ProcObjConnection*, functions as an embedded object to connect other types of process objects to the group start to form the complete control loop.

The code in this control module has to be written to define start and stop of the control loop. The code, for example, can be used to switch between different values of the master controller setpoint. The feedback *stopped* and *started* must also be defined. The connection information is defined by a sub control module to the connection module.

GroupStartStandby Object

This object supervises the connected process objects. If any of the running objects fails, this object tries to keep the desired number of objects running.

The following are the characteristics of the connected system, if the GroupStartStandby object is used:

- If the desired number of required objects cannot be kept running, an alarm is generated to indicate that few objects remain in good condition. This alarm function may be switched off.
- To maintain the required number of running objects, the next ready object makes a start attempt to create an even wear on all objects.
- When the required number of objects does not change, the user can change the active objects just to keep an even wear on them.
- Each object is supervised by a timer and when a configured maximum time is reached, this objects stops and the next object is started. This object change (when the maximum time is reached) may be switched off.
- When a new object is to be started as a result of another running on maximum time, the start and stop order can be selected. When such a change of running objects takes place, an event is fired (if selected).
- The first object in the queue stops when the required number of objects decreases.
- The standby object can be connected and controlled by a group start sequence. If this is the case, it responds immediately on a start request from the group start administrative objects (Head or Step), and later the standby object controls the connected objects autonomously.
- The standby object can be switched between manual mode and auto mode. In auto mode, the object is controlled by the group start sequence. In manual mode, the start and stop action can be performed locally.
- If the input parameter is not connected, the standby object reacts as if it has got a start order, and controls the connected object with respect to the required number of started objects. If it is in manual mode, the entire group of connected process objects may be switched on and off, based on the required number of started objects.

- All the connected process objects can be switched into GroupStart mode using the Group Start button (G) in the faceplate.

Section 7 Surveillance

Introduction

This section presents a short description about the signal and vote loop concept in the SignalLib.

Signal and Vote Loop Concept

Vote control module types may be connected to objects with vote logic (control module types with a voting parameter). The vote control module types are used to define different rules that make it possible to control the process to predetermined states. This means that the process can behave differently, depending on valid rule, for example shutdown the process.

Overview

The SignalLib library consists of control modules for signal input, voting and output signals and the ProcessObjLib libraries consist of control modules for process control. The ControlLib libraries consist of control modules for control loops and calculation.

A typical usage for vote control modules are applications that are divided into signal loops, where voting with signal status diagnostics and communication to external applications is an essential and integrated part of the application. Each loop contains input signal control modules, vote control modules and output control modules. The output signals in the vote control modules are latched and can be reset from the process output signal control module or from any connected vote control module in the loop. The latched command signals are sent via MMS to a separate application in addition to a local alarm/event.

Figure 152 shows the principal voting data flow for a SIF (Safety Instrumented Functions) loop and Figure 154 shows some combination possibilities between SIF loop logic and common programmer calculation code, using the input signals from signal objects real as well as boolean types. The vote control modules have a possibility to combine the action together with a command number inside the structured component VotedConnection. Then, the receiving object as e.g. a process object like a motor or a valve may decode this command into different actions inside the process object like PriorityCmd0 or/and perhaps ILock1. PriorityCmd0 or/and ILock1 are defined by an input parameter (*xxxConfig*) for each possible action where *xxx* is representing the specific action, for example PriorityCmd0Config. This gives the possibility for different Vote control modules to take different (or equal) actions in the process object.

The selections in *xxxConfig* are displayed in the interaction window of the process object and the signal output control modules.

The voted commands are numbered from 1 to 32 to define a corresponding bit in the *xxxConfig* word.

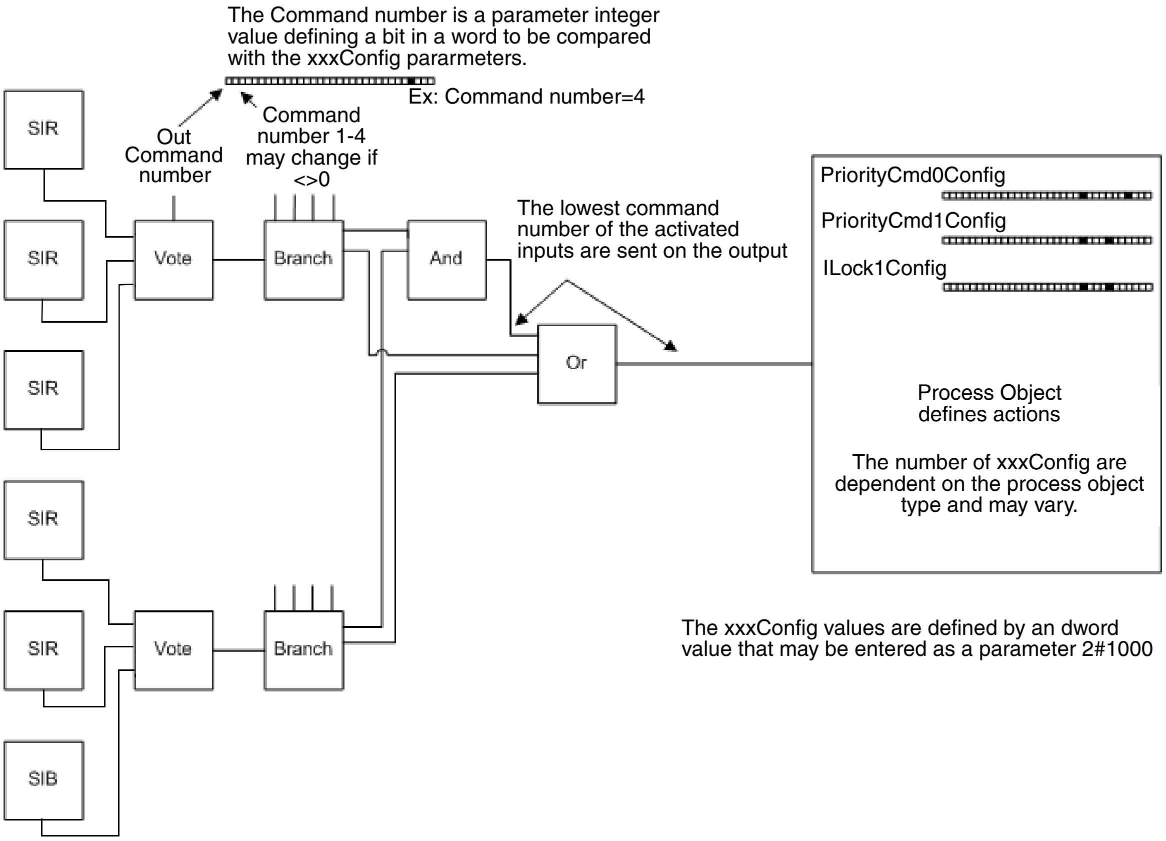
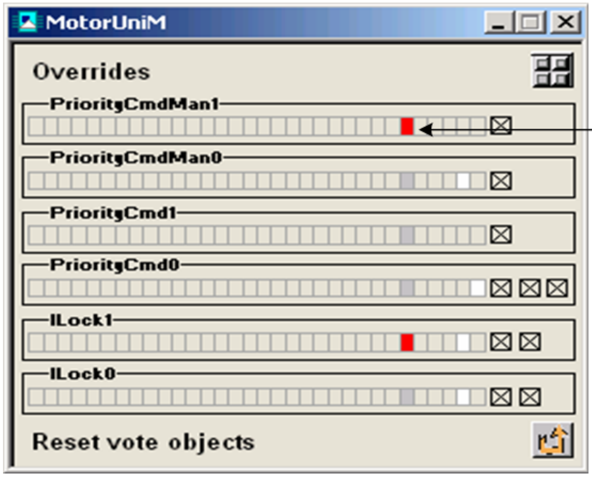

The Command number is a parameter integer value defining a bit in a word to be compared with the xxxConfig pararmeters.
Ex: Command number=4
Out Command number
Command number 1-4 may change if <>0
SIR
SIR
SIR
SIR
SIR
SIB
Vote
Branch
And
Vote
Branch
Or
The lowest command number of the activated inputs are sent on the output
PriorityCmd0Config
PriorityCmd1Config
ILock1Config
Process Object defines actions
The number of xxxConfig are dependent on the process object type and may vary.
The xxxConfig values are defined by an dword value that may be entered as a parameter 2#1000

Figure 152. Configuration example of vote logic. (SIR=Signal In Real, and SIB=Signal In Bool)

The output from a vote control module may also be connected to other objects defined in Table 41, for example, an output signal object that defines the action of the output signal when the voted signal is activated.

The coding method of the *xxxConfig* parameters is based upon which state the process is to enter when the command is received. To enter a specific process state, the process object has to give one or several commands. The different commands may be individual for different output control modules types, as listed in Table 41.



The square color changes when the bit pattern matches the received command number.

Red color is used when both Cmd and LatchedCmd are true.

Green color is used when Cmd component has returned to false state.

White squares indicate the value of the received command number.

Black color indicate the configured behavior of different actions.

Figure 153. Example of Voted commands indications in a Interaction Window of a process object (MotorUniM).

The process object in the example has three different command types that are affecting the process object in three different ways:

- PriorityManCmd 0 and 1
- PriorityCmd 0, and 1
- Lock0 and 1

The corresponding config parameters contains a bit pattern that is compared to the command bit position and if they match the command is given to the process object.

These commands are or-ed with the corresponding parameters and finally the process object priorities what actually shall be performed. If for example the PriorityCmdMan0 command is given from the voting logic and the input parameter PriorityCmdMan1 also is true, the PriorityCmdMan0 will be the result. The *xxxConfig* parameters together with the command bit pattern is displayed in the interaction window.

Table 37 describes the standard library types for supervision.

Table 37. Standard library types for supervision

Type Name	Library	Type	Description
SignalBool	SignalLib	Function block	SignalBool has a digital input and an output, both of bool data type, with several alarm and event functions when input value differs from normal value, and interaction windows. The input and output are intended to be connected to bool variables in an application.
SignalBoolCalcOutM[1]	SignalLib	Control module	SignalBoolCalcOutM is a version of SignalBool that handles input connections from a vote control module. Input/output is of Boolconnection.
SignalBoolCalcInM[1]	SignalLib	Control module	SignalBoolCalcInM s a version of SignalBool that handles connections to a vote control module. Input/output is of Boolconnection.

Table 38 describes the standard library types for input signal handling.

Table 38. Standard library types for input signal handling

Type Name	Library	Type	Description
SignalInBool(M)	SignalLib	Function block and Control module(1)	SignalInBool has a digital input, of BoolIO data type, with several supervision functions, such as alarm and event levels, and interaction windows. The signal input is intended to be connected to a digital input I/O variable. The signal output is of bool data type (Function blocks) and BoolConnection (Control modules).
SignalBasicInBool	SignalBasicLib	Function block	SignalBasicInBool is used for overview and forcing of boolean input signals of data type BoolIO. The input signal value is transferred to the output value. Error is set to true when input IO status is error marked.

Table 39 describes the Standard library types for digital output.

Table 39. Standard library types for digital output

Type Name	Library	Type	Description
SignalOutBool(M)	SignalLib	Function block and Control module[1]	SignalOutBool has a digital output of BoolIO data type (Function Blocks) and BoolConnection (Control modules), with several supervision functions, such as alarm and event levels, and interaction windows. The signal input is intended to be connected to a digital input I/O variable. The signal output is of bool data type.
SignalBasicOutBool	SignalBasicLib	Function block	SignalBasicOutBool is used for overview and forcing of boolean output signals of data type BoolIO. The input value is transferred to the output signal value. Error is set to true when output IO status is error marked.

Example

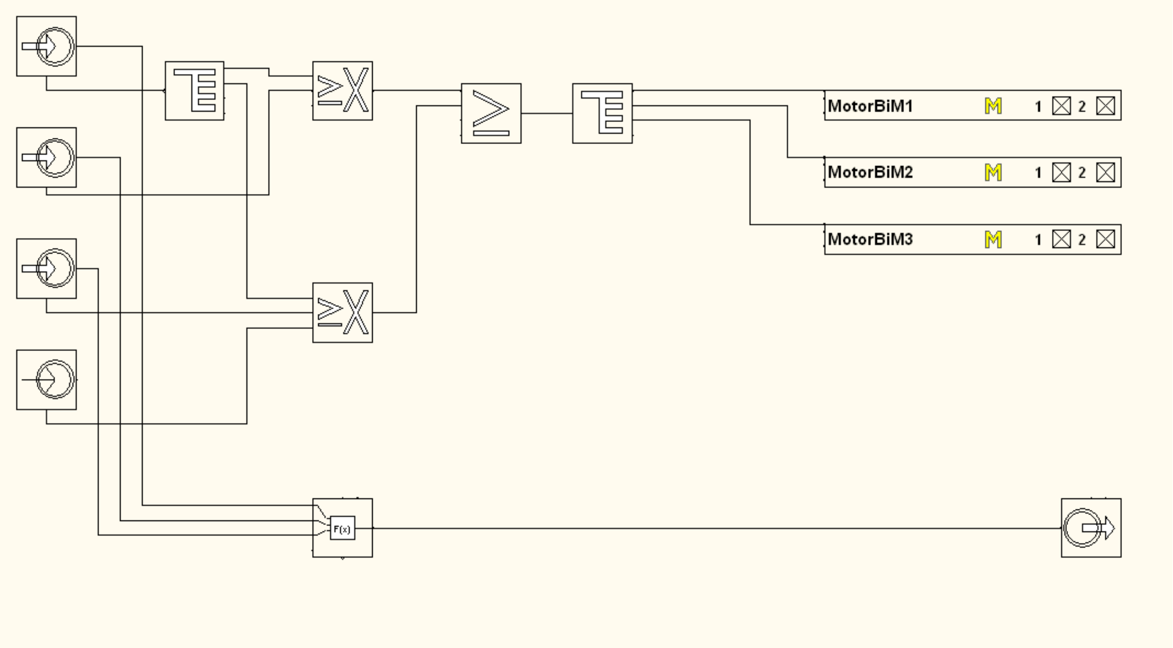


Figure 154. Configuration example of vote logic

As a result of the upper vote control module in the control module diagram above, the process is to **shut down** completely, and as a result of the second vote control module the process is to be placed in a **stand by** mode.

Solution

Start to define the behavior of the different process objects or connected outputs at **shutdown** and **standby**, according to following:

Behavior	**Action**
Shutdown	The upper process object shall run (use PriorityCmd)
	The middle process object shall prevent the automation program to transfer the object into the stopped position (use ILock0)
	The lower process object shall stop and when the vote condition has been released be placed in manual mode (use PriorityManCmd0)
Standby	The upper process object shall run (Use PriorityCmd1)
	The middle process object shall not be affected.
	The lower process object shall prevent the automation program to transfer the object into the started position (Use ILock1)

This gives the following settings on the objects:

Object	**Setting**
Upper vote	OutCommandNumber: 1
Lower vote	OutCommandNumber: 2
Upper process object	PriorityCmd0Config: 2#1 PriorityCmd0Config: 2#10
Middle process object	ILock0Config: 2#1
Lower process object	PriorityManCmd0Config: 2#1 ILock1Config: 2#10

Standard Object Types that Support Voting Logic

There are two types of objects that support Voting logic:

- Objects determining specified situations like too high values, sending it to the voting objects (sending objects).
- Objects receives voted commands and are responding to them (receiving objects).

The column 'Reset of overrides' in Table 40 and Table 41 means that the commanded overrides functions (Force, Inhibit and Disable) can be reset from outside the object using the VoteConnection structured data type for sending objects and the VotedConnection data type for receiving objects. SDLevelM initiates this type of action that is distributed through the Branch, And and Or object for each type of vote connection.

Sending Objects

Sending objects send the detected information into a node of VoteConnection. The table below displays sending objects and also describes the possible detected situations related to the *InxLevelConfig* parameter of the voting objects. Values of *InxLevelConfig* outside the described range or on objects where the value is marked with a “x” gives ParError in ParError detecting objects.

Table 40. Sending VoteConnection standard objects

***InxLevelConfig* in Voting object** **Object**	**Reset of Overrides**	**Real Value Out(1)**	**= -3**	**= -2**	**= -1**	**= 0**	**= 1**	**= 2**	**= 3**
SignalInBoolM	Yes	x	x	x	x	DiffNormal	x	x	x
SignalBoolCalcInM	Yes	x	x	x	x	DiffNormal	x	x	x
SignalInRealM	Yes	Yes	LLL	LL	L	x	H	HH	HHH

Table 40. Sending VoteConnection standard objects (Continued)

***InxLevelConfig* in Voting object** **Object**	**Reset of Overrides**	**Real Value Out[1]**	**= -3**	**= -2**	**= -1**	**= 0**	**= 1**	**= 2**	**= 3**
SignalSimpleInRealM	Yes	Yes	x	x	L	x	H	x	x
SignalRealCalcInM	Yes	Yes	LLL	LL	L	x	H	HH	HHH
PidCC	x	Yes	x	x	DevNeg	x	DevPos	x	x
PidAdvancedCC	x	Yes	x	x	DevNeg	x	DevPos	Oscillation detected	Sluggish control detected
Level2CC	x	Yes	x	x	L	x	H	x	x
Level4CC	x	Yes	x	LL	L	x	H	HH	x
Level6CC	x	Yes	LLL	LL	L	x	H	HH	HHH
GSHead	x	X	X	X	Time error	Stopped	Started	X	X
BiM	x	x	x	x	Object error	Off	On Pos 1	On Pos 2	x
UniM	x	x	x	x	Object error	Off	On	x	x
ValveUniM	x	x	x	x	Object error	Closed	Opened	x	x
MotorBiM	x	x	x	x	Object error	Stopped	Runs Pos1	Runs Pos2	x
MotorUniM	x	x	x	x	Object error	Stopped	Runs	x	x

Table 40. Sending VoteConnection standard objects (Continued)

***InxLevelConfig* in Voting object** **Object**	**Reset of Overrides**	**Real Value Out**[1]	**= -3**	**= -2**	**= -1**	**= 0**	**= 1**	**= 2**	**= 3**
MotorValveM	x	x	x	x	Object error	Stopped	Opened	Closed	x
MotorValveCC	x	x	x	x	Object error	Stopped	Opened	Closed	x
InsumBreakerM	x	x	Trip	Warn ing	Object error	Opened	Closed	x	x
McuBasicM	x	x	Trip	Warn ing	Object error	Stopped	Runs Pos1	Runs Pos2	x
McuExtendedM	x	x	Trip	Warn ing	Object error	Stopped	Runs Pos1	Runs Pos2	x
ACStdDriveM	x	x	Trip	Warn ing	Object error	Stopped	Runs	Stopped	Runs
DCStdDriveM	x	x	Trip	Warn ing	Object error	Stopped	Runs	Current Off	Current On
EngDriveM	x	x	Trip	Warn ing	Object error	Stopped	Runs	Current Off	Current On
VoteXoo1Q	x	x	x	x	x	Action	Latched Action	x	x
VoteXoo2D	x	x	x	x	x	Action	Latched Action	x	x
VoteXoo3Q	x	x	x	x	x	Action	Latched Action	x	x
VoteXoo8	x	x	x	x	x	Action	Latched Action	x	x

Table 40. Sending VoteConnection standard objects (Continued)

***InxLevelConfig* in Voting object** **Object**	**Reset of Overrides**	**Real Value Out[1]**	**= -3**	**= -2**	**= -1**	**= 0**	**= 1**	**= 2**	**= 3**
VotedOr	x	x	x	x	x	Action	Latched Action	x	x
VotedAnd	x	x	x	x	x	Action	Latched Action	x	x
VotedBranch	x	x	x	x	x	Action	Latched Action	x	x

[1] Sends a real value to be used in the statistic calculation of the vote object (highest, lowest, median and average value).

Receiving Objects

The sending objects receive the voted information and reacts on it, dependent of the nature and possibilities of the individual object. The table below displays the sending objects and also describes the possible different behavior.

Table 41. Receiving VotedConnection standard objects

Object	**Reset of Overrides**	**Voting Action**		
SignalBoolCalcOutM	Yes	Freeze	Predetermined Value	
SignalOutBoolM	Yes	Freeze	Predetermined Value	

Table 41. Receiving VotedConnection standard objects (Continued)

Object	Reset of Overrides	Voting Action		
SignalRealCalcOutM	Yes	Freeze	Predetermined Value	
SignalOutRealM	Yes	Freeze	Predetermined Value	
SignalSimpleOutRealM	Yes	Freeze	Predetermined Value	
SDLevelM	x	Activate		
PidCC	x	->Auto mode	-> External setpoint/ Internal setpoint	-> Tracking mode
PidAdvancedCC	x	->Auto mode	-> External setpoint/ Internal setpoint	-> Tracking mode
SelectorCC	x	Select predefined input channel	-> Internal setpoint	
Selector4CC	x	Select predefined input channel also in chained configuration	-> Internal setpoint	
ManualAutoCC	x	-> Auto mode		
SignalSupervisionCC	x	Freeze	Predetermined Value	
GSHead	x	Start	Stop	
BiM	x	PriorityMan 0, 1, 2	Priority 0, 1, 2	Interlock 0, 1, 2
UniM	x	PriorityMan 0, 1	Priority 0, 1,	Interlock 0.1
ValveUniM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1

Table 41. Receiving VotedConnection standard objects (Continued)

Object	**Reset of Overrides**	**Voting Action**		
MotorBiM	x	PriorityMan 0, 1, 2	PriorityMan 0, 1, 2	Interlock 0, 1, 2
MotorUniM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1
MotorValveM	x	PriorityManOpen, Stop,Close	PriorityOpen,Stop, Close	InterlockOpen, Stop,Close
MotorValveCC	x	-> Auto mode	PriorityOpen,Stop, Close	PriorityValue
InsumBreakerM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1
McuBasicM	x	PriorityMan 0, 1, 2	PriorityMan 0, 1, 2	Interlock 0, 1, 2
McuExtendedM	x	PriorityMan 0, 1, 2	PriorityMan 0, 1, 2	Interlock 0, 1, 2
ACStdDriveM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1
DCStdDriveM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1
EngDriveM	x	PriorityMan 0, 1	PriorityMan 0, 1	Interlock 0, 1
Mimo22CC	x	Selects a specified equation		
Mimo41CC	x	Selects a specified equation		
Mimo44CC	x	Selects a specified equation		

Vote Control Module Types

Type Name	**Library**	**Description**
Vote1oo1Q	SignalLib	Performs voting of one input signal, where the signal quality is considered together with the activation signal from the input module, for example from SignalInRealM. The vote output is set if the input signal if either activated or have bad quality.
VoteXoo3Q	SignalLib	Performs voting of up to three input signals, where the signal quality is considered together with the activation signal from the input modules, for example from SignalInRealM. The vote output is set if X (X parameter) number of input signals are either activated or have bad quality.
VoteXoo2D	SignalLib	Performs voting of up to two input signals, where the signal quality is diagnosed together with the activation signal from the input modules, for example from SignalInRealM. If X parameter =1; the vote output is set if any input signal is activated and quality is good, or if all connected signals have bad quality. If X parameter =2; the vote output is set if both input signals are activated and quality is good, or if one input signal is activated and quality is good and the other input signal have bad quality, or if all connected signals have bad quality.
VoteXoo8	SignalLib	Performs voting of up to eight input signals from input modules, for example from SignalInRealM. The vote output is set if X (X parameter) number of input signals are activated. The signal quality is not considered.

Vote Structure Control Module Types

Type Name	Library	Description
VotedAnd4	SignalLib	Makes an AND function between four signal of VotedConnection from a vote control module.
VotedOr4	SignalLib	Makes an OR function between four signals of VotedConnection from vote control modules.
VoteBranch4	SignalLib	Branches a signal of VoteConnection, for example an output from a signal object control module, into four signals of the same data type.
VotedBranch4	SignalLib	Branches a signal of VotedConnection, from a vote control module, into four signals of the same data type

SDLevelM Control Module Types

This control module provides an easy way to structure the Emergency or Process shutdown logic in the commonly used level hierarchy. There are activation input parameters from surrounding objects as well as from superior levels, and output parameters to effect process objects as well as subordinate levels. These four input and output possibilities may be inhibited individually.

The level is activated by an input signal from superior level objects, the *cause* input from a voting circuit, or the *Activate* interaction command from the object interaction window.

An example of such a hierarchal structure is (see Figure 155):

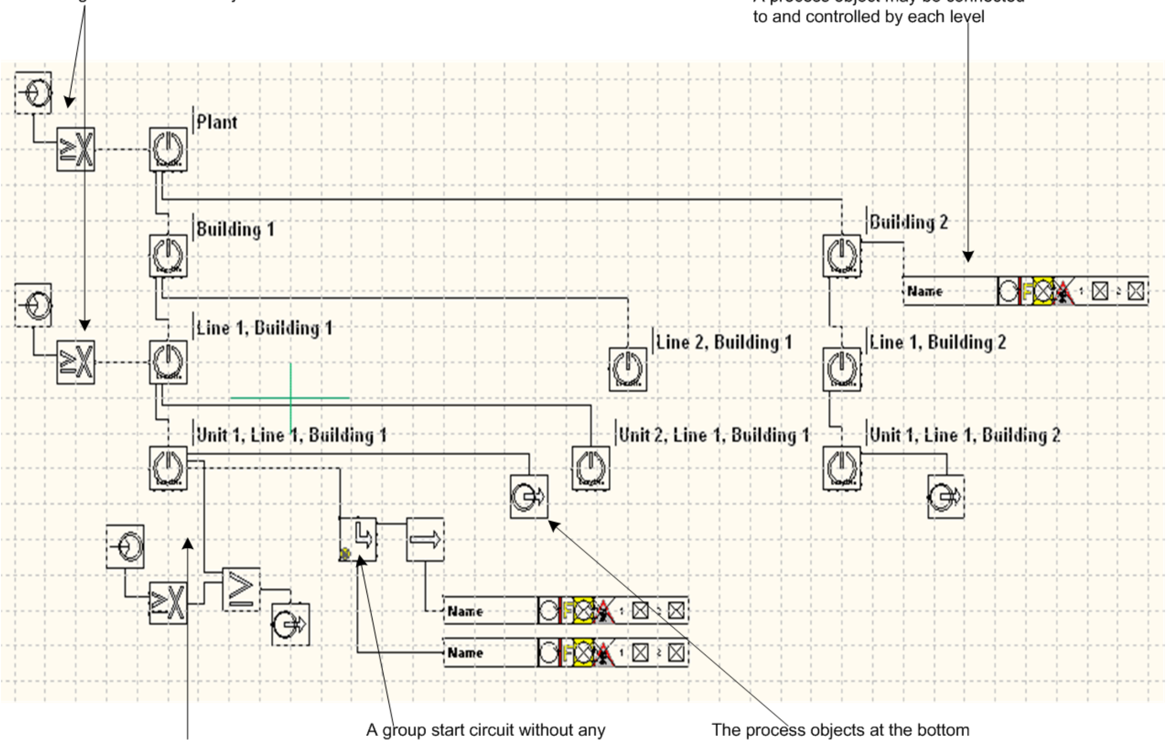
- A plant that consists of several buildings.
- A building that has several production lines.
- A production line that has several batch processes.
- A batch process that has several units.
- A unit that has several process objects.

When the total plant shuts down, all equipment in the plant also shuts down. At the base structure, when a unit shuts down, all process objects in the unit shuts down.

These shut downs may occur according to the connected process objects in different levels. The behavior of the output process object is decided in the object itself, described in the example of voting mechanisms, see Signal and Vote Loop Concept on page 367.

The cause input to this level is the voted result and the action is configured in the level object.

A process object may be connected to and controlled by each level



The level output may be logically connected in a SIS loop

A group start circuit without any previously connect group headers may act as a process object and take effect of the voted commands

The process objects at the bottom level are controlled by the bottom level objects, where the action of the object is defined

Figure 155. Level Usage Example

Overrides

The overrides that exist in this object are the inhibitions of the input and output signals. These inhibitions can be performed using parameters with the corresponding name or components in the InteractionPar data structure. The inhibitions from the interaction can be enabled or disabled by controlling the parameter *EnableOverride*. An interaction command or parameter input signal is used to reset the overrides in the object.

The input signals *In* and *Cause*, and the output signals *Outx* and *Effectx* are connected graphically as shown in the Figure 156.

Activation signals

The *In* parameter is connected to any of the *Outx* parameters of superior level SDLevelM control module. Hence, the *Outx* parameters is connected to the *In* parameter of the subordinate level SDLevelM control module.

The *Cause* parameter is connected to the voted output parameter of SIF loop representing the level activation. The *Effectx* parameters is connected to the VotedCmd input parameter of the process object to be controlled.

When a Level is activated, all connected subordinate levels and all connected effects will be activated except those having the corresponding inhibition signal activated.

The communication signaling to the surrounding objects is distributed in the connection structured data types, for example reset of overrides or reset of latches.

Override control signals

The *ResetOverride* parameter is connected to an appropriate signal, for example, an input connected to a push button and/or the *ResetAllApplForced* parameter of a ForcedSignals function block.

The *EnableOverride* parameter is connected to an appropriate signal, for example, an input connected to a key switch and/or the *AccessEnable* parameter of a ForcedSignals function block.

The interaction push buttons for reset of overrides and reset of latches are enabled when any overrides or latches that can be reset exist.

Set the parameter *EnableOverride* to true, to enable the control of inhibitions in this object. If this input parameter is false, the entry in the interaction window disabled.

Object node connections

The objects of control module type are equipped with nodes for graphical connection in the CMD Editor.

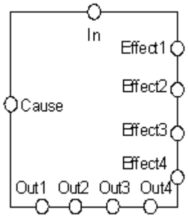
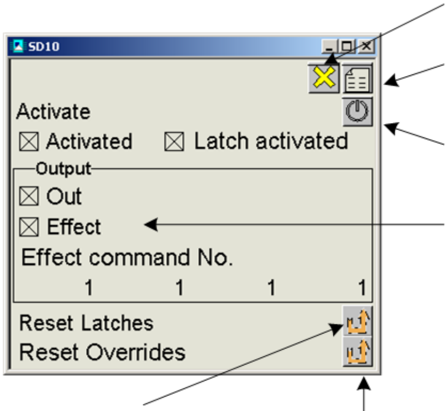


Figure 156. The node connection for SDLevelM

Control Builder Interaction Window



SD10

Activate

☒ Activated ☒ Latch activated

Output

☒ Out

☒ Effect

Effect command No.

1 1 1 1

Reset Latches

Reset Overrides

This button invokes the Cause input vote window

This button invokes the information window

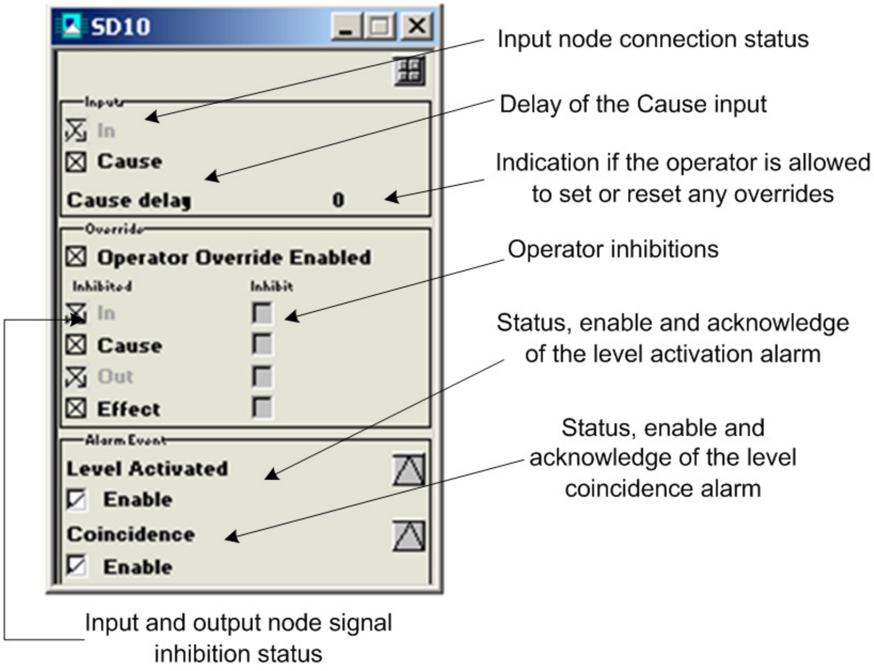
This button creates a manual cause command

The Effect output command numbers used for decoding of desired activation of the connected receiving process objects

The buttons used to reset latched information in the level and connected sub levels

The buttons used to reset overrides in the connected process objects

Figure 157. The Control Builder Interaction Window



Input node connection status

Delay of the Cause input

Indication if the operator is allowed to set or reset any overrides

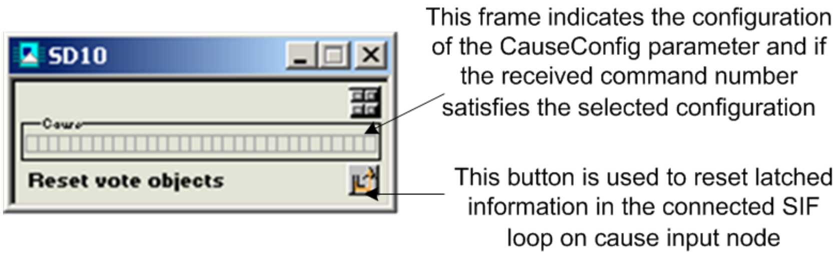
Operator inhibitions

Status, enable and acknowledge of the level activation alarm

Status, enable and acknowledge of the level coincidence alarm

Input and output node signal inhibition status

Figure 158. The Control Builder InfoPar



This frame indicates the configuration of the CauseConfig parameter and if the received command number satisfies the selected configuration

This button is used to reset latched information in the connected SIF loop on cause input node

Figure 159. The Cause Input Vote Window

SDLevelOr4

This control module makes an OR function between four signal of SDLevelMConnection.

Object node connections

The objects of control module type are equipped with nodes for graphical connection in the CMD Editor.

In the Forward calculation:

- The smallest .LevelNo value is transferred to the output structure.
- The output .Level, .LatchedLevel and .Reset are or'ed from the inputs.

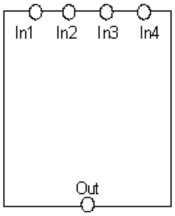


Figure 160. The node connection for SDLevelOr4

SDLevelAnd4

This control module makes an AND function between four signal of SDLevelMConnection.

Object node connections

The objects of control module type are equipped with nodes for graphical connection in the CMD Editor.

In the Forward calculation:

- The smallest .LevelNo value is transferred to the output structure
- The output .Level and .LatchedLevel are and'ed from the inputs.

- The output .Reset is or'ed from the inputs.

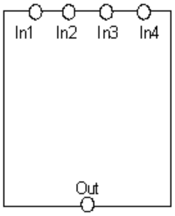


Figure 161. The node connection for SDLevelAnd4

SDLevelBranch4

This control module splits a signal of SDLevelMConnection.

Object node connections

The objects of control module type are equipped with nodes for graphical connection in the CMD Editor.

The input .ResetEnabled is or'ed from the outputs in the backward calculation.

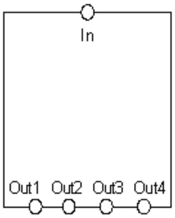


Figure 162. The node connection for SDLevelBranch4

Latching input object quality information

The latch mechanism is activated when the quality information distributed to the VoteConnection has been better that the lowest value again (e.g. raised from bad to uncertain or good, or raised from uncertain to good).The interaction window indicates that the quality information is latched by textual information.

This Information window has different layout depending of the signal object type.

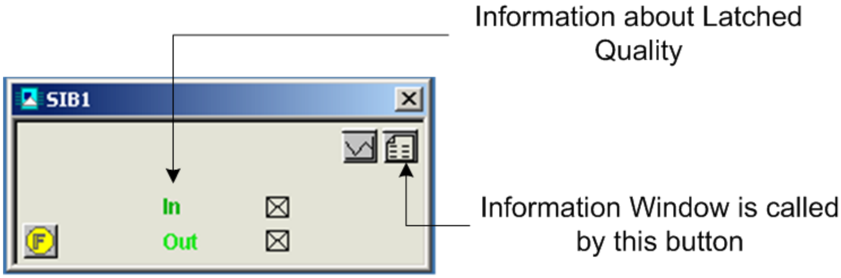
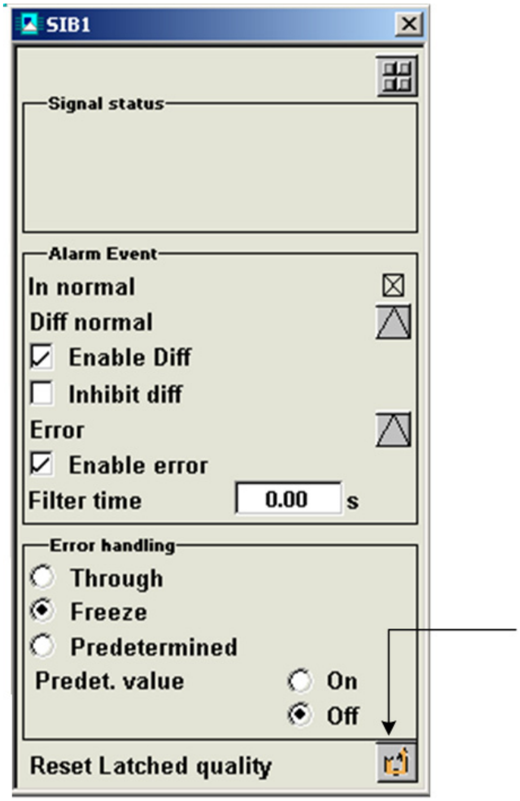

Information about Latched Quality
SIB1
In ☒
Out ☒
F
Information Window is called by this button

Figure 163. Interaction Windiow-Input Quality Latched

The operators can acknowledge by calling up the information window.

-



SIB1

Signal status

Alarm Event

In normal ☒

Diff normal

☑ Enable Diff

□ Inhibit diff

Error

☑ Enable error

Filter time 0.00 s

Error handling

○ Through

● Freeze

○ Predetermined

Predet. value ○ On ● Off

Reset Latched quality

This Reset button line is visible when the parameter 'EnableLatchQuality' is true. The Operator can reset the latched information.

Figure 164. Resetting Latched Information

The latched information is visible, only when the VoteOut parameter is connected.

Appendix A Customized Online Help

In this section you will find requirements on customized help for self-defined libraries, applications and components of externally added applications, as well as for non-standard hardware. Customized help can be produced using any online help authoring tool.

How to add customized help for user-defined libraries with hardware and non-standard hardware types, differ from how to add customized help for user-defined libraries (with data types, function block types and control module types) and applications.

Online Help Files for User-defined Libraries and Applications

The following requirements must be fulfilled on customized online help for user-defined libraries (with data types, function block types and control module types) and applications.

1. Any online help authoring tool that produces Microsoft HTML Help 1.3 can be used when producing the external help files.
2. Your help files should be placed in the *UserHelp* folder*,* which is located in the standard help file folder of the Control Builder product, *ABB Industrial IT\...Help\UserHelp.* (For example, *c:\Program files\ABB Industrial IT\Engineer IT\Compact Control Builder 5.1\Help\UserHelp*).
3. The help file should be of the Microsoft HTML help file type, and should have the same name as the library or application (for example, *MyLib.chm).*

4. Context-sensitive help (F1 help) must always use A keywords that are based on the Project Explorer object name.The name of the library or application in Project Explorer is used by Control Builder, when calling the online help file. See External Help Files via F1 on page 396.,

Only Microsoft HTML Help files (*.chm) of version 1.3 are supported.

Online Help Files for User-defined Libraries with Hardware and Non-standard hardware

The following requirements must be fulfilled on customized online help for user-defined libraries (with hardware types) and non-standard hardware.

1. Any online help authoring tool that produces Microsoft HTML Help or WinHelp can be used when producing the external help files.
2. The help file can be of HTML (*.chm) or WinHelp (*.hlp) type.
3. The external help file should be added to the library with hardware or to the hardware type in Project Explorer. How to add a help file to a library with non-standard hardware (with, for example, I/O units) and a specific hardware type in a library, see *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* .

Context-sensitive help (F1) works with any help file name of the added help file, as long as it is a HTML or WinHelp help file. F1 on a hardware type without any added help file calls the help file added (if any) to the user-defined library (with the hardware type).

Access Customized Online Help from Control Builder

There are three ways to access online help from Control Builder. You can use the menu option **Help...**, you can select an item in Project Explorer and press F1, or you could click the **Help** button on the tool bar, or in a dialog.

Help Menu

The Show Help About dialog provides access to help files that have been added by the user (if such files exist in the *UserHelp* folder). Use this dialog box to access external, inserted help files. The dialog box is opened from the Project Explorer: **Help > Show Help About...**, as illustrated in Figure 165.

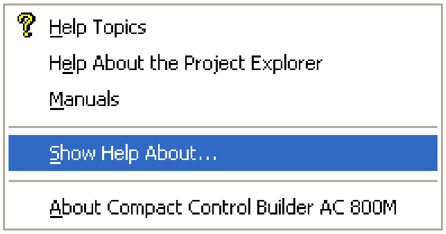
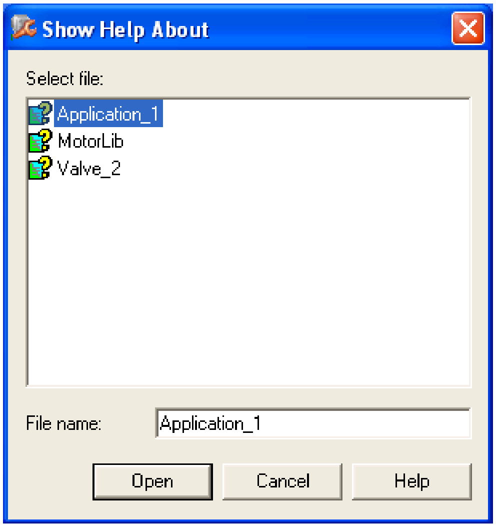

Help Topics
Help About the Project Explorer
Manuals
Show Help About...
About Compact Control Builder AC 800M

Figure 165. Help menu in the Project Explorer window

All externally added *.chm files residing in the predefined help file folder, *...Help\UserHelp*, are listed in the **Show Help About** dialog box. See Figure 166.



Select file:

Application_1
MotorLib
Valve_2

File name: Application_1

Figure 166. The Show Help About dialog box

To open a file from the dialog box, select (or double-click) a file from the list, or type the name in the **File name** field, and click **Open**.

Help files displayed in the Show Help About dialog box are not a part of Control Builder online help. This means that if a project is transferred to a new computer, you manually will have to copy these help files from the *UserHelp* folder on the old computer to the *UserHelp* folder on the new computer. This also means that these help files are not included when a system backup is performed.

Context-Sensitive Help (F1)

If you select an item in Project Explorer and then press the F1 key, help on the selected item will be displayed. The F1 key can be used on all items in applications, libraries, user-defined libraries, hardware, and externally added hardware. It is also possible to get F1 help on error messages in the editor message pane, or in the project explorer pane.

External Help Files via F1

Control Builder supports context-sensitive help on user-defined library (with data types, function block types and control module types) and components of externally added applications, provided that the corresponding help file is placed in the *UserHelp* folder.

Context-sensitive help on user-defined libraries with hardware and non-standard hardware is available if a help file (HTML or WinHelp file with any file name) is added to the library or to the hardware type. See *Compact 800 Engineering Compact Control Builder AC 800M Configuration (3BSE041488*)* manual.

External help files for user-defined library (with data types, function block types and control module types), and components of externally added applications, must have exactly the same name as the corresponding component file, as it appears in the Control Builder tree, with the extension *chm* (refer to Table 42). Otherwise, context-sensitive help will not work.

Table 42. Control Builder files and corresponding help files (examples)

Object Type	Control Builder File	Help File
Library	Foocos.lbr	Foocos.chm
Application	Application_1.app	Application_1.chm

Context-Sensitive Linking

Context-sensitive linking between the help project topics and user-defined libraries (with data types, function block types and control module types), and components of externally added applications, in Project Explorer is done by A keyword linking. Project Explorer objects should have their exact names specified as an A keyword in the corresponding topic. An A keyword is a non-language-dependent text string, sent from Control Builder (at an F1 call) to the online help system.

You must add the name of the object as an A keyword to the help topic describing the object. Control Builder uses the name of the currently selected item and tries to find a corresponding A keyword in the help system. If a call fails, the Control Builder keyword is displayed under the **Index Tab** in the HTML Help Viewer.

Appendix B Library Objects Overview

This section gives an overview of all library objects, such as data types, functions, function block types, and control module types that can be used in applications created using the Control Builder engineering tool. Refer to the appropriate manuals and Control Builder online help for detailed descriptions of the libraries.

Almost all library types are protected. This means that the types cannot be copied to your own library and then modified. To determine if a library object is protected, select the object in the Project Explorer and then try to copy it. If the object is dimmed, it is protected.

System

The System contains IEC 61131-3 data types and functions, as well as data types and functions with extended functionality designed by ABB.

IEC 61131-3 Standard Functions

Table 43. IEC 61131-3 standard functions

Function	TC	Description
Type conversion functions		
*_TO_**	Y(1)	Type conversion from * to ** The following type conversions are implicit (that is, a call to a type conversion function is not needed): bool -> word, bool -> dword word -> dword int -> dint, int -> real uint -> dint, uint -> real dint -> real explicit conversion functions: <bool, int, dint, uint, words, dwords time>_to_real <int, dint, uint, word, dword, real>_to_bool <bool, uint, dint, word, dword, real>_to_int <bool, int, uint, word, dword, real, time>_to_dint

Table 43. IEC 61131-3 standard functions (Continued)

Function	TC	Description
		<bool, int, dint, word, dword, real>_to_uint <bool, int, dint, uint, real, dword>_to_word <bool, int, dint, uint, real, word>_to_dword
	Y	string_to_date_and_time, string_to_dint, string_to_dword, string_to_int, string_to_real, string_to_time, string_to_uint, string_to_word
	N	dword_to_string, int_to_string, real_to_string, time_to_string, uint_to_string, word_to_string, date_and_time_to_string, bool_to_string, dint_to_string
	N	string_to_bool
General functions		
ABS	Y	Absolute value
SQRT	Y	Square root
Logarithmic functions		
EXP	Y	Natural exponential
LN	Y	Natural logarithm
LOG	Y	Logarithm to base 10
Trigonometric functions		
ACOS	Y	Principal arc cosine
ASIN	Y	Principal arc sine
ATAN	Y	Principal arc tangent
COS	Y	Cosine in radians
SIN	Y	Sine of input in radians
TAN	Y	Tangent in radians

Table 43. IEC 61131-3 standard functions (Continued)

Function	TC	Description
Extensible arithmetic functions		
ADD	Y	Addition (OUT:= IN1 + IN2 + … + Inn)
MUL	Y	Multiplication (OUT:= IN1 * IN2 * … * INn)
Non-extensible arithmetic functions		
DIV	Y	Division (OUT := IN1 / IN2)
EXPT	Y	Exponentiation (OUT := IN1 raised to IN2)
MOD	Y	Modulus (OUT := IN1 modulo IN2)
MOVE	Y	Move (OUT := IN)
SUB	Y	Subtraction (OUT := IN1 – IN12)
Standard bit shift functions		
ROL	Y	Rotate bits left (OUT := IN left-rotated by N bits, circular)
ROR	Y	Rotate bits right (OUT := IN right-rotated by N bits, circular)
SHL	Y	Shift bits left (OUT := IN left-shifted by N bits, zero-filled on right)
SHR	Y	Shift bits right (OUT := IN right-shifted by N bits, zero-filled on left)
Standard bit-wise Boolean functions		
AND	Y	Boolean AND (OUT := IN1 AND IN2 AND … AND INn)
NOT	Y	Boolean negation (OUT := NOT IN1)
OR	Y	Boolean OR (OUT := IN1 OR IN2 OR … OR INn)
XOR	Y	Boolean Exclusive OR (OUT := IN1 XOR IN2 XOR … XOR INn)

Table 43. IEC 61131-3 standard functions (Continued)

Function	TC	Description
Standard selection functions		
LIMIT	Y	Delimiter between a minimum, min, variable value and a maximum, max, variable value (non-extensible).
MAX	Y	Select the largest of the input variables (extensible)
MIN	Y	Select the smallest of the input variables (extensible)
MUX	Y	Multiplexer which selects the variable pointed out by the input variable (extensible)
SEL	Y	Binary selection (non-extensible)
Standard comparison functions		
EQ (=)	Y	Equality
GE (>=)	Y	Monotonic decreasing sequence
GT (>)	Y	Decreasing sequence
LE (<=)	Y	Monotonic increasing sequence
LT (<)	Y	Increasing sequence
NE (<>)	Y	Inequality (non-extensible)

Table 43. IEC 61131-3 standard functions (Continued)

Function	TC	Description
Standard character string functions		
CONCAT	N	Extensible concatenation
DELETE	N	Delete L characters of IN, beginning at the Pth character position
FIND	Y	Find the character position of the beginning of the first occurrence of IN2 in IN1. If no occurrence of IN2 is found, then OUT := 0.
INSERT	N	Insert IN2 into IN1 after the Pth character position
LEFT	N	Left-most L characters of IN
LEN	Y	String length function
MID	N	L characters of IN beginning at the Pth character position
REPLACE	N	Replace L characters of IN1 by IN2, starting at the Pth character position
RIGHT	N	Right-most L characters of IN
Functions of time data types		
ADD	Y	Add time variables
SUB	Y	Subtract time variables

(1) Conversion functions can be used in time-critical tasks, except for the functions: bool_to_string, int_to_string, dint_to_string, uint_to_string, word_to_string, dword_to_string, real_to_string, time_to_string, date_and_time_to_string and string_to_bool.

Non-IEC 61131-3 Conversion Functions

Table 44. Non-IEC 61131-3 conversion functions

Function	TC	Description
Type conversion functions		
addsuffix	Y	Add a suffix to a String.
ASCIIStructToString	N	The ASCIIStructToString function takes a struct of *dint*s, which contains the codes for an ASCII string, and reconstructs the string from the values in the components of the struct. The component values of the integer, DintStruct, are read and translated into the value of the destination string, String.
BCDToDint	Y	BCDToDint converts a BCD value into an integer value (*dint*).
Bool16ToDint	Y	Bool16ToDint converts a Boolean struct BoolStruct with 16 items into a *dint*.
Bool32ToDint	Y	Bool32ToDint converts a Boolean struct BoolStruct with 32 items into a *dint*.
CalendarStructTo Date_and_time	Y	This function converts a CalendarStruct to a date_and_time value.
Date_and_timeTo CalendarStruct	Y	This function converts a date_and_time value to a CalendarStruct.
DIntToBCD	Y	DintToBCD converts an integer value into a BCD value.
DIntToBool16 and DIntToBool32	Y	The DIntToBool16 and DIntToBool32 functions convert a *dint* into a Boolean struct BoolStruct with 16 or 32 items, respectively.
DIntToGraycode	Y	This function converts a *dint* value to a Graycoded value.

Table 44. Non-IEC 61131-3 conversion functions (Continued)

Function	TC	Description
GraycodeToDInt	Y	This function converts a Graycoded value to a *dint* value.
MaxStringLength	N	The MaxStringLength function returns the maximum length of a string; that is, the allocated size of the string variable, as an integer value.
serial_string_append_ASCII	N	The function append a character to a string.
serial_string_append_Hex	N	The function appends the hexadecimal representation of a DWord to a string.
serial_string_find_ASCII	N	This function returns the position of a specified character within a string.
serial_string_get_ASCII	N	This function returns the ASCII code of a character in a string.
serial_string_Hex_to_DWORD	N	The function converts a hexadecimal string to a DWord.
serial_string_left	N	This function extracts the leftmost characters from a string.
serial_string_mid	N	This function extracts a substring from a string.
serial_string_put_ASCII	N	The function replaces a character in a string.
serial_string_replace_Hex	N	The function replaces a substring in a string with the hexadecimal representation of a DWord.
serial_string_right	N	This function extracts the rightmost characters from a string.
StringToASCIIStruct	N	This function converts a string to an ASCIIStruct.

Table 44. Non-IEC 61131-3 conversion functions (Continued)

Function	TC	Description
DWordToIPString	N	This function converts a DWord, containing the binary information of an IP address, to a string value. The first part of the IP address string is the most significant byte in the DWord and the other three parts are the remaining three bytes in the DWord placement order.
IPStringToDWord	N	This function converts a IP address string to a DWord containing the binary information of the IP address. The most significant byte in the DWord contains the first part of the IP address and the other three bytes in the DWord placement order contain the other three parts of the IP address.

Non-IEC 61131-3 bit string functions

Table 45. Non-IEC 61131-3 bit string functions

Function	TC	Description
SetBit (WORD, UINT) or SetBit (DWORD, UINT)	Y	Sets a bit to 1 in the WORD (or DWORD) specified by the first bit string argument, and returns the value. The bit to set is specified by the second argument. The bit in the second argument is a number from 0 to *N-1*, where N is the number of bits used to represent the first argument (16 for WORD, and 32 for DWORD). 0 denotes the least significant bit, and *N-1* denotes the most significant bit. If the second argument specifies a bit >= N, the first argument (a WORD or DWORD) is unaffected by the call.
SetBits(WORD, WORD) or SetBits(DWORD, DWORD)	Y	Sets a number of bits in the WORD (or DWORD) specified by the first bit string argument, to bits specified by the second argument, and returns the value. The bits are specified using a mask. For example, SetBits(2#1111000011110000, 2#0101) results in 2#1111000011110101.

Table 45. Non-IEC 61131-3 bit string functions (Continued)

Function	TC	Description
ClearBit(WORD, UNIT) or ClearBit(DWORD, UNIT)	Y	Sets a bit to 0 in the WORD (or DWORD) specified by the first bit string argument, and returns the value. The bit to be set to 0 (cleared) is specified by the second argument. The bit in the second argument is a number from 0 to *N-1*, where N is the number of bits used to represent the first argument (16 for WORD, and 32 for DWORD). 0 denotes the least significant bit, and *N-1* denotes the most significant bit. If the second argument specifies a bit >= N, the first argument (a WORD or DWORD) is unaffected by the call.
ClearBits(WORD, WORD) or ClearBits(DWORD, DWORD)	Y	Sets a number of bits to 0 in the WORD (or DWORD) specified by the first bit string argument, and returns the value. The bits to be set to 0 (cleared) are specified in the second argument using a mask. For example, ClearBits(2#11111111, 2#101) results in 2#11111010.

Table 45. Non-IEC 61131-3 bit string functions (Continued)

Function	TC	Description
TestBit(WORD, UINT) or TestBit(DWORD, UINT)	Y	Tests whether a bit is set (the value is 1) in the WORD (or DWORD) specified by the first bit string argument, and returns a BOOL value (true or false). The bit to test is specified by the second argument. The function returns *true* if the bit is set, or *false* if the bit is not set. The bit in the second argument is a number from 0 to *N-1*, where N is the number of bits used to represent the first argument (16 for WORD, and 32 for DWORD). 0 denotes the least significant bit, and *N-1* denotes the most significant bit. If the second argument is >= N, the function returns *false*.
TestBits(WORD, WORD) or TestBits(DWORD, DWORD)	Y	Tests if all of a number of bits are set (the value is 1) in the WORD (or DWORD) specified by the first bit string argument, and returns a BOOL value (true or false). The bits to be tested are specified in the second argument using a mask. If all bits specified by the mask are set, the function returns *true*, otherwise it returns *false*. For example: TestBits(2#1011, 2#1001) returns true. TestBits(2#1011, 2#1101) returns FALSE.

Other Functions

Table 46. Other functions

Function	TC	Description
CheckSum	N	The CheckSum function calculates checksums used for ASCII protocols written in the programming language.
EqAnyType	N	The EqAnyType function compares two variables of any type.
ExecuteControlModules	Y	This function is used in function blocks that contain control modules. When the function is called from the function block, all control modules in that function block are executed.
GetCVStatus	Y	The GetCVStatus function accepts the communication variable as input, and provides the complete status and extracted statuses of the variable through different output parameters. These output parameters for extracted statuses can be connected to variables to control the logic.
GetDTQuality	N	Returns the quality of the system time. It may be GOOD, UNCERTAIN or BAD.
GetStructComponent	N	This function reads (copies) values from a struct component.
GetSystemDT	Y	Returns the system time when current task was started.
InhibitDownload	Y	This function is used to prevent download to a controller.
InitAnyType	N	The InitAnyType function initiates all components of a structured data type variable.
LocalDTToSystemDT	N	Returns the system time for the specified local time.

Table 46. Other functions

Function	TC	Description
Match	Y	The Match function returns the position of a string within another string. Unlike the Find function, a wildcard can be used with the Match function.
Modp	Y	The Modp function returns the remainder after integer division and is related to the Mod function. The functions differ on negative values. Mod follows the IEC 61131 standard and Modp follows the behavior in SattLine version 2.2 or earlier.
MoveAnyType	N	The MoveAnyType function copies the Source parameter of any type into the Destination parameter.
NationalLowerCase	N	The NationalLowerCase function sets upper-case letters to lower-case letters.
NationalUpperCase	N	The NationalUpperCase function sets lower-case letters to upper-case letters.
PutStructComponent	N	This function writes (copies) values into a struct component.
RandomNorm	Y	The RandomNorm function returns a normally distributed random number.
RandomRect	Y	The RandomRect function returns a random number uniformly distributed between 0.0 and 1.0.
RandomSeed	Y	The RandomSeed function initializes the RandomGenerator to a random value.
ReadStatusZeroDivInt	Y	The ReadStatusZeroDivInt function checks for zero division exceptions for integer values.
ReadStatusZeroDivReal	Y	The ReadStatusZeroDivReal function checks for zero division exceptions for real values.

Table 46. Other functions

Function	TC	Description
RealInfo	N	This function is used to get information about a real value, that is, if it is within the allowed range. It is used to check overflow/underflow for integer arithmetic operations. The function checks if a result is valid or invalid (invalid results include infinity and NaN (not a number)).
ReservedByTool	Y	This function is used to check whether download to the controller is in progress. The method returns True if download is in progress, else False.
ResetForcedValue	-	This function is not used in Compact Control Builder.
Round	Y	The Round function rounds a real value to the nearest integer.
SetFalse	Y	The Boolean operator will be set to false.
SetSeed	Y	The SetSeed function initializes a random generator using a specific start value.
SetTrue	Y	The Boolean operator will be set to true.
SystemDTToLocalDT	N	Returns the local time for the specified system time.
Timer	Y	The Timer function controls a timer.
TimerElapsed	Y	The TimerElapsed function returns the elapsed time of a timer as a time value.
TimerElapsedMS	Y	The TimerElapsedMS function returns the elapsed time of a timer in milliseconds as a *dint* value.
TimerHold	Y	The TimerHold function stops a timer.

Table 46. Other functions

Function	TC	Description
TimerReset	Y	The TimerReset function stops and resets a timer.
TimerStart	Y	The TimerStart function starts a timer.
Trunc	Y	The Trunc function truncates a real value to an integer.
WriteVar	N	Enables a system to write to a variable in a controller, or to transfer variables between controllers, without owning the variable.

Array and Queue Functions

Table 47. Array and queue functions

Function	**TC**	**Description**
		Arrays
CreateArray	N	Creates an array of elements of the same type as ArrayElement.
DeleteArray	N	Deletes the array *Array* and deletes the whole tree structure of arrays.
GetArray	N	Gets the contents of the data type at position Index in the array *Array* into *ArrayElement*.
InsertArray	N	Inserts a new element in an array. All successive elements are moved one step, and the last element is overwritten.
PutArray	N	Puts the contents of *ArrayElement* into the data type at position Index in the array.
SearchArray	N	Searches the array ArrayName for a certain component in an array element.
SearchStructComponent	N	A boolean function which searches for a specific part in a structured component.
		Queues
ClearQueue	N	This procedure clears the queue *Queue*.
CreateQueue	N	Creates a queue of elements of the same type as *QueueElement*.
CurrentQueueSize	N	This integer function returns the current number of elements in the queue.
DeleteQueue	N	This procedure deletes the queue.

Table 47. Array and queue functions (Continued)

Function	TC	Description
GetFirstQueue	N	Puts the contents of the first element from the queue into *QueueElement*.
GetLastQueue	N	Puts the contents of the last element from the queue into *QueueElement*.
PutFirstQueue	N	Puts the contents of *QueueElement* into the first element of the queue.
PutLastQueue	N	Puts the contents of *QueueElement* into the last record element of the queue.
ReadQueue	N	Reads the contents of the specified element number from the queue and puts them into *QueueElement*.

Task Control Functions

Table 48. Task control functions

Function	TC	Description
FirstScanAfter ApplicationStart	Y	The FirstScanAfterApplicationStart function checks if the current scan/execution is the first one performed by the current task.
FirstScanAfterPowerUp	Y	The FirstScanAfterPowerUp function checks if the controller has been warm started.
GetActualIntervalTime	Y	Use the GetActualIntervalTime function to get the actual interval time of the current task.
GetApplicationSIL	-	This function is not used in Compact Control Builder.
GetIntervalTime	Y	Use the GetIntervalTime function to get the requested interval time of the current task.

Table 48. Task control functions (Continued)

Function	TC	Description
GetPriority	Y	Use the GetPriority function to obtain the priority of the current task.
SetIntervalTime	Y	Use the SetIntervalTime function to set the requested interval time of the current task.
SetPriority	N	Use the SetPriority function to set the priority of the current task.

Basic Library

The BasicLib is the basic library for the *Control Builder M* software. It contains data types, function block types and control module types with extended functionality designed by ABB.

IEC 61131-3 Function Block Types

Table 49. IEC 61131-3 function block types

Function Block Type	TC	Description
Standard bistable function block types		
SR	Y	Bistable function block (set dominant)
RS	Y	Bistable function block (reset dominant)
Standard edge detection function block types		
R_TRIG	Y	Rising edge detector
F_TRIG	Y	Falling edge detector
Standard counter function block types		
CTU	Y	Up-counter

Table 49. IEC 61131-3 function block types

Function Block Type	TC	Description
CTD	Y	Down-counter
CTUD	Y	Up-down counter
Standard timer function blocks type		
TP	Y	Pulse timer
TON	Y	On-delay timer
TOF	Y	Off-delay timer
RTC	Y	The RTC (Real-Time-Clock) function block performs display and date and time setting.
SEMA	Y	SEMA, the semaphore function block, is designed to allow competing tasks to share a particular resource.

Process Object Function Block Types

Use the ACOF (Automatic Check Of Feedback) functions primarily for supervision of process objects. ACOF function blocks can monitor up to three output signals and up to three feedback signals.

Use ACOF when a feedback signal is expected within a certain time after an output to the process object has been activated or deactivated. If the feedback signal is not detected, an alarm signal is given.

Table 50. Process object function block types

Function Block Type	TC	Description
ACOFAct	Y	Object control with two limit switches and automatic return. This object has one stable position.
ACOFActDeact	Y	Object control with two limit switches. This object has two stable positions.

Table 50. Process object function block types (Continued)

Function Block Type	TC	Description
ACOFAct 3P	Y	Object control with three limit switches and automatic return. This object has one stable position.
ACOFActDeact 3P	Y	Object control with three limit switches. This object has three stable positions.

Other Function Block Types

Table 51. Other function block types

Function Block Type	TC	Description
Conversion function block types		
BcToDint	Y	BcToDint converts data from an optional number of binary coded bool inputs and a sign input into a *dint*.
DintToBc	Y	DintToBc converts data from *dint* to an optional number of bool outputs, using binary coded conversion, and a sign output.
DintToFirstOfN	Y	DintToFirstOfN converts data from *dint* to an optional number of bool outputs using 1-of-N conversion, and a sign output.
DintToNBcd	Y	DintToBcd converts from *dint* to an optional number of bool (in groups of four coded as BCD) and a sign output.
FirstOfNToDint	Y	FirstOfNToDint converts data from 1-of-N format with an optional number of bool inputs and a sign input into a *dint*.
GrayToDint	Y	GrayToDint converts data from gray code with an optional number of bool inputs and a sign input into a *dint*.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
IntegerToRealIO	Y	IntegerToRealIO converts a raw integer value to a scaled *RealIO* value with a measuring range, units and decimals. It can be treated as a physical *RealIO* value in the application.
NBcdToDint	Y	NBcdToDint converts data from an optional number of bool inputs (in groups of four coded as BCD) and a sign input into a *dint*.
RealIOToInteger	Y	The RealIOToInteger function block converts a scaled *RealIO* value to a raw integer value.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
Diagnostics function block types		
PowerFailureInfos	N	PowerFailureInfos provides information on power failure status, such as number and duration. In addition, the total number of power failures since the last reset and the duration of these can also be presented.
SystemDiagnostics	N	SystemDiagnostics can be used to measure and display the following: 1. Stop time and memory usage during a controller download 2. Current memory in use 3. Maximum memory used since the last cold start 4. Alarm and Event statistics 5. Ethernet statistics
SystemDiagnosticsSM	-	This function block is not used in Compact Control Builder.
System time function block types		
GetDT	N	GetDT retrieves the system time from the controller at which the current task was started. The function block converts the system time to local time so that both are available as parameters.
GetTimeZoneInfo	N	GetTimeZoneInfo retrieves the currently used time-zone settings.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
SetDT	N	SetDT sets the system time on the controller. It is possible to set the time directly as system time or indirectly as local time and the function block will perform the conversion to system time. The time can be set as relative or as absolute time. This functionality is available through parameters and an Interaction window. The Interaction window also shows the current system time and local time.
SetTimeZoneInfo	N	SetTimeZoneInfo sets the time-zone information. StandardDT and DaylightDT can be specified with the absolute or the day-in-month format. This functionality is available through parameters and an Interaction window. This function overwrites the time-zone information downloaded by the ACB.
Timer function block types		
TimerD	Y	TimerD measures the elapsed time in the negative direction and indicates when a specified level has been exceeded.
TimerOffHold	Y	TimerOffHold is an Off-delay timer function block with a hold input function.
TimerOnHold	Y	TimerOnHold is an On-delay timer function, with a hold input function.
TimerOnOffHold	Y	TimerOnOffHold is an On-delay and Off-delay timer function block type with a hold input function.
TimerPulseHold	Y	TimerPulseHold is a pulse timer with a hold function.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
TimerPulseHoldDel	Y	TimerPulseHoldDel is a pulse timer with a hold input function and a delayed start to generate Q output pulses.
TimerU	Y	TimerU measures the elapsed time in positive direction and gives indication when a specified level is past.
Level detector function block types		
LevelHigh	Y	LevelHigh is used to compare a real value with an optional number of high limits with the possibility of specifying hysteresis. A high-level detector is a trip for supervising an analog signal. A high-level trip indicates when the input signal exceeds the selected high-detect level. The detector has hysteresis on the input signal, which prevents the level detector output signal from repeatedly changing state when the supervised input signal varies near the detection level.
LevelLow	Y	LevelLow is used to compare a real value with an optional number of low limits with the possibility of specifying hysteresis. A low-level detector is a trip for supervising an analog signal. A low-level trip indicates when the input signal drops below the preset low-detect level. The detector has a hysteresis on the input signal, which prevents the level detector output signal from repeatedly changing state when the supervised input signal varies near the detection level.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
Threshold	Y	Threshold is used to determine when more than, or equal to, a given number of Boolean values are true.
Calculator function block types		
Median	Y	Median is used to calculate the median value of an optional number of input values. The function block supports *real* and *dint*: MedianReal and MedianDint.
MajorityReal	Y	Used to calculate the mean value of a number of real numbers.
Selector function block types		
DeMux	Y	DeMux can handle the data types *real*, *dint* and *bool*: DeMuxReal, DeMuxDint and DeMuxBool.
Max	Y	Max is used to select the largest value of an optional number of inputs. The Max function block exists for *real* and *dint*: MaxReal and MaxDint.
Min	Y	Min is used to select the lowest value of an optional number of inputs. The function block exists for *real* and *dint*: MinReal and MinDint.
Bistable function block types		
RSD	Y	RSD is used as a memory for bool variables. Besides the RS function, it can also override this function with a write function.
Generator function block types		
PulseGenerator	Y	PulseGenerator is a pulse generator used to create a continual pulse signal.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
PulseGeneratorAcc	Y	PulseGeneratorAcc is a pulse generator used to create a continual pulse signal with high long term accuracy.
SinGen	Y	SinGen is used to generate a sinus signal. The frequency and amplitude are controlled by inputs.
SqGen	Y	SqGen is used to generate a square wave with an optional number of amplitudes. Each amplitude is determined by an input and for each amplitude the time during which it is to be maintained is specified.
TimePulses	Y	The pulse generator creates a pulse on the outputs every hour on the Hour output and every new day on the Day output, synchronized to the real-time clock. There is also a parameter in this function block, which determines whether the pulse need to be generated either based on UTC or based on the local time.
Register function block types		
Fifo	N	Fifo is a queue register of First-In-First-Out type. Fifo can handle the data types real, dint and bool: FifoReal, FifoDint and FifoBool
FifoRW	N	FifoRW is a queue register of First-In-First-Out type. In addition, data can be changed and deleted at any position in the queue. FifoRW can handle the data types *real*, *dint* and *bool*: FifoRWReal, FifoRWDint and FifoRWBool.

Table 51. Other function block types (Continued)

Function Block Type	**TC**	**Description**
Register	Y	Register is used as a memory function block with an optional number of positions. The function block exists for data types *real*, *dint* and *bool*: RegisterReal, RegisterDint and RegisterBool.
Shift	N	Shift is used as a shift register of optional length. The function block exists for data types *real*, *dint* and *bool*: ShiftReal, ShiftDint and ShiftBool.
ShiftL	Y	ShiftL is used as a shift register of optional length. All positions can be written and read. The function block exists for data types *real*, *dint* and *bool*: ShiftLReal, ShiftLDint and ShiftLBool.
Controller interaction function block types		
ErrorHandler	-	This function block is not used in Compact Control Builder.
ForcedSignals	-	This function block is not used in Compact Control Builder.
SaveColdRetain	N	This function block copies cold-retain variables from RAM to backup media. This is done in order to make the cold-retain values survive a restart of the controller. The function block processes all the applications that are executing in the controller.
Various function block types		
ApplicationInfo	N	ApplicationInfo gives information about the application where it executes in e.g. name and state.

Table 51. Other function block types (Continued)

Function Block Type	TC	Description
EvalRestartInhibit	N	EvalRestartInhibit may inhibit restart of application in evaluation mode.
PrintLines	N	PrintLines prints text lines on a printer.
RedundantIn	Y	When the system switches from ordinary to redundant I/O there may be a jump in the signal which may not be good for the algorithms used. This function block type prevents this by ramping the signal using a real value as the change of the signal per second.
SampleTime	Y	SampleTime is a function block to measure the sampling period and to acquire the requested sampling period.
Trigger	Y	Trigger provides a combined time and event trigger functionality.

Control Module Type

Table 52. Control Module Types

Control Module Type	TC	Description
ErrorHandlerM	-	This control module is not used in Compact Control Builder.
ForcedSignalsM	-	This control module is not used in Compact Control Builder.
GroupStartObjectConn	N	This is the connection module between a group start sequence and object to be started or stopped in the sequence.

Table 52. Control Module Types (Continued)

Control Module Type	TC	Description
CCInputGate	Y	The CCInputGate makes sure that the ControlConnection specification if fulfilled on the input otherwise ParError will be true. This module should be used inside of a control module together with the CCOutputGate control module.
CCOutputGate	Y	The CCOutputGate makes sure that the ControlConnection specification if fulfilled on the output otherwise ParError will be true. This module should be used inside of a control module together with the CCInputGate control module.
CCInputGateExtended	N	Same as the CCInputGate control module but with additional functionality. This module should be used inside of a control module together with either the CCOutputGateExtended or CCOutputGate.

Table 52. Control Module Types (Continued)

Control Module Type	TC	Description
CCOutputGateExtended	N	Same as the CCOutputGate control module but with additional functionality. This module should be used inside of a control module together with either the CCInputGateExtended or CCInputGate.
CvAckISP	N	This control module is used to reset the latched ISP values in communication variables. One control module instance is used to reset all such latched values in one group of communication variables. The communication variables with latched functionality are defined in groups. The CVAckISP control module is used to initiate the operator reset action. If several such groups are to be reset simultaneously, the control module instances for each group are interconnected in a cascade configuration. The reset order is distributed to all members in that kind of configuration.

Communication Libraries

The libraries MMSCommLib, ModemCommLib, COMLICommLib, ModBusCommLib, MB300CommLib, S3964RCommlib, SattBusCommLib, SerialCommLib, InsumComLib, FFHSECommLib and FFH1CommLib contain a number function block types and control module types that provide external variable communication with protocols such as MMS, FOUNDATION Fieldbus, SattBus, COMLI, Siemens 3964R MasterBus 300 and INSUM devices.

There are also function block types for self-defined UDP communication (in UDPCommLib), self-defined TCP communication (in TCPCommlib), and modem connection (in ModemCommLib).

MMSCommLib

The library MMSCommLib uses the MMS function block types and control modules to establish communication with a system supporting the MMS protocol.

When transferring variables it is important to use data types with the same range on the client as on the server. It is, however, possible to connect variables with different ranges, such as a *dint* variable on the server and an *Int* variable on the client. This will only work as long as the variable values are within the range of the *Int* variable.

Table 53. MMS function block types

Function Block Type	TC	Description
MMSConnect	N	Initiates a communication channel and establishes a connection with a remote system.
MMSRead	N	Reads one or several variables.
MMSReadCyc	N	Reads one or several variables cyclically.
MMSWrite	N	Writes to one or several variables.
MMSWriteDT	N	Transmits date and time.

Table 53. MMS function block types (Continued)

Function Block Type	TC	Description
MMSDefAccVar	N	This function block is used to create an access variable, which is connected to a defined variable in the executing system. The defined variable is then accessible for both reading and writing from a remote system and also within its own system.
MMSDef4Bool	N	Used for safe communication of data between SIL applications. MMSDef4Bool operates in a pair with MMSRead4Bool, and is used to transfer the values of four different Boolean variables. MMSDef4Bool is used in the server application, while MMSRead4Bool is used in the client application.
MMSDef4BoolIO	N	Used for safe communication of data between SIL applications. MMSDef4BoolIO operates in a pair with MMSRead4BoolIO, and is used to transfer the values of four different Boolean IO variables. MMSDef4BoolIO is used in the server application, while MMSRead4BoolIO is used in the client application.
MMSDef4Dint	N	Used for safe communication of data between SIL applications. MMSDef4Dint operates in a pair with MMSRead4Dint, and is used to transfer the values of four different Double Integer variables. MMSDef4Dint is used in the server application, while MMSRead4Dint is used in the client application.

Table 53. MMS function block types (Continued)

Function Block Type	TC	Description
MMSDef4DintIO	N	Used for safe communication of data between SIL applications. MMSDef4DintIO operates in a pair with MMSRead4DintIO, and is used to transfer the values of four different Double Integer IO variables. MMSDef4DintIO is used in the server application, while MMSRead4DintIO is used in the client application.
MMSDef4Real	N	Used for safe communication of data between SIL applications. MMSDef4Real operates in a pair with MMSRead4Real, and is used to transfer the values of four different Real variables. MMSDef4Real is used in the server application, while MMSRead4Real is used in the client application.
MMSDef4RealIO	N	Used for safe communication of data between SIL applications. MMSDef4RealIO operates in a pair with MMSRead4RealIO, and is used to transfer the values of four different Real IO variables. MMSDef4RealIO is used in the server application, while MMSRead4RealIO is used in the client application.
MMSRead4Bool	N	Forms a pair for safe communication of data between SIL applications, together with MMSDef4Bool. See the above description of MMSDef4Bool.
MMSRead4BoolIO	N	Forms a pair for communication of data between applications, together with MMSDef4BoolIO. See the above description of MMSDef4BoolIO.

Table 53. MMS function block types (Continued)

Function Block Type	TC	Description
MMSRead4Dint	N	Forms a pair for safe communication of data between SIL applications, together with MMSDef4Dint. See the above description of MMSDef4Dint.
MMSRead4DintIO	N	Forms a pair for communication of data between applications, together with MMSDef4DintIO. See the above description of MMSDef4DintIO.
MMSRead4Real	N	Forms a pair for safe communication of data between SIL applications, together with MMSDef4Real. See the above description of MMSDef4Real.
MMSRead4RealIO	N	Forms a pair for communication of data between applications, together with MMSDef4RealIO. See the above description of MMSDef4RealIO.

Table 54. MMS control module types

Control Module Type	TC	Description
MMSToCC	N	The communication protocol is MMS. The forward and the backward structure of *ControlConnection* is handled separately in MMS variable groups.
CCToMMS	N	CCToMMS is used together with MMSToCC. The communication protocol is MMS. The forward and the backward structures of *ControlConnection* are handled separately in MMS variable groups.

Table 54. MMS control module types (Continued)

Control Module Type	TC	Description
MMSRead16BoolM	N	Reads 16 Bool variables from a remote controller.
MMSRead64BoolM	N	Reads 64 Bool variables from a remote controller.
MMSRead128BoolM	N	Reads 128 Bool variables from a remote controller.
MMSReadHI	N	Reads variables from a remote system or from an application within the same system. The result of the read operation is put in the Any Type parameters Out1 and Out2.
MMSDefHI	N	Defines the variables connected to the Any Type parameters In1 and In2 as access variables available for remote access.

ModemCommLib

The library ModemCommLib library contains function block types used to establish communication with a modem.

Table 55. Modem function blocks

Function Block Type	TC	Description
ModemConnStat	N	ModemConnStat is used to obtain the current status of a modem connected to a selected channel. The current status is given by the Status parameter.

Table 55. Modem function blocks

Function Block Type	TC	Description
ModemDialUp	N	ModemDialUp is used to connect a modem via a defined communication channel.
ModemHangUp	N	ModemHangUp is used to disconnect a modem via a defined communication channel.

COMLICommLib

The library COMLICommLib contains the function block types to establish communication with a system supporting the COMLI protocol.

Function block types with the COMLI prefix support both the address-oriented COMLI and SattBus protocols. When a SattBus channel is used, the COMLI are packed within SattBus. The protocol to be used (COMLI or SattBus) is defined by the *Channel* parameter of the *COMLIConnect* function block.

Communication via a TCP/IP network is also supported.

Table 56. COMLI function block types

Function Block Type	TC	Description
COMLIConnect	N	Connects to a defined communication channel.
COMLIRead	N	Reads one or several variables.
COMLIReadCyc	N	Reads variable data cyclically.
COMLIReadPhys	N	Requests physical values from a SattConXX system.
COMLIWrite	N	Writes to one or several variables.
COMLIWriteDT	N	Transmits date and time of master to the slave.

ModBusCommLib

The library ModBusCommLib contains the function block types to establish communication with a system supporting the MODBUS protocol.

Table 57. MODBUS functions

Function Block Type	TC	Description
MBConnect	N	Initiates a communication channel and establishes a connection with a remote system.
MBRead	N	Reads one or several variables.
MBWrite	N	Writes to one or several variables.
MBException	N	Reads the ModBus exception coils.

MTMCommLib

The MOD5-to-MOD5 Communication Library, MTMCommLib, contains Function Block types to establish communication with a system supporting the MOD5-to-MOD5 protocol.

Table 58. MOD5 function blocks

Function Block Type	TC	Description
MTMConnect	N	Initiates a communication channel and prepares for a logical connection for communication with MOD5 controller.
MTMReadCyc	N	Is used to request named variables cyclically. This is done by activation of the **Enable** parameter.

Table 58. MOD5 function blocks (Continued)

Function Block Type	TC	Description
MTMDefCyc	N	Is used to transfer the named variable data cyclically to the CI872 to be available for reading from a MOD5 controller.
MTMDefERCyc	N	Is used to transfer the named variable data cyclically to the CI872 to be available for reading from a MOD5 controller. It creates Access Variables internally. It is recommended to be used in application for PA controller only

MB300CommLib

The library MB300CommLib contains objects that can be used for set up communication with MasterBus 300 (MB300). MB300 can be used with AC 400 and AC 800M. A CI855 communication unit for AC 800M provides connectivity to AC 400 via MB300. Refer to the relevant manuals regarding the process interface that can be used with legacy controller AC 400.

Table 59. MB300 function block types

Function Block Type	TC	Description
MB300Connect	N	MB300Connect is used to establish connection between the calling communication partner and the remote communication partner.
MB300DSSend	N	The MB300Send function block is used to send a DataSet to a node on MB300.
MB300DSReceive	N	The MB300Receive function block is used for reception of a DataSet sent by a node on MB300.

ModBusTCPCommLib

The ModBusTCPCommLib contains Function Blocks types to establish communication with a system supporting the MODBUS TCP protocol.

Table 60. MODBUS TCP function block types

Function Block Type	TC	Comment
MBTCPConnect	N	Initiates a communication channel and prepares for a connection
MBTCPRead	N	Is used to request variable data. This is done by activation of the Req parameter
MBTCPReadCyc	N	This function block is used to cyclically request variable data. This is done by activation of the Enable parameter.
MBTCPReadFileRecord	N	Is used to read the File Records (supports FC20 Function Code). Used to Read time stamp Events from Switchgear Device supporting Modbus protocol.
MBTCPException	N	Reads the status of the exception coils
MBTCPWrite	N	Is used to send variable data. This is done by activation of the Req parameter
MBTCPWriteFile Record	N	Is used to write the File Records(supports FC21 Function Code). Used to Write time stamp Events to Switchgear Device supporting Modbus protocol.
MBTCPReadWrite	N	Used to write to registers and read from registers (supports FC23 Function Code) in remote devices using MODBUS TCP, in a single MBTCP transaction.

S3964RCommLib

The library S3964RCommlib contains the function block types to establish communication with a system supporting the Siemens 3964R protocol.

Table 61. Siemens 3964R function block types

Function Block Type	TC	Description
S3964RConnect	N	Connects to a defined communication channel.
S3964RRead	N	Reads one or several variables.
S3964RReadCyc	N	Reads variable data cyclically.
S3964RWrite	N	Writes to one or several variables.

SattBusCommLib

The library SattBusCommlib contains the function block types supporting SattBus. They are used to communicate through a SattBus channel using the SattBus name-oriented model. To communicate through a SattBus channel using the address-oriented model, COMLI function block types are used. Communication via a TCP/IP network is also supported.

Table 62. SattBus function block types

Function Block Type	TC	Description
SBConnect	N	Connects to a defined communication channel.
SBRead	N	Reads one variable value.
SBReadCyc	N	Reads variable data cyclically.
SBWrite	N	Writes one variable value.

Table 62. SattBus function block types (Continued)

Function Block Type	**TC**	**Description**
ComliSB		*These following (ComliSB) Function Blocks ONLY support communication over a TCP/IP network, thus cannot work when running on a physical COMLI- port.*
ComliSBConnect	N	Use the ComliSBConnect function block type to initiate a communication channel and establish a connection to a slave system with a unique node/slave address on a TCP/IP network.
ComliSBRead	N	The ComliSBRead function block supports the address- oriented transfer model. Use it to request variable data from a remote system on a TCP/IP network.
ComliSBReadCyc	N	The ComliSBReadCyc function block supports the address-oriented transfer model. Use it to cyclically read one or several variables from a slave system on a TCP/IP network.
ComliSBReadPhys	N	Use the ComliSBReadPhys function block to request physical measuring ranges (that is, scaling factors) of registers or analog signals from a slave system on a TCP/IP network.

Table 62. SattBus function block types (Continued)

Function Block Type	TC	Description
ComliSBWrite	N	The ComliSBWrite function block supports the address-oriented transfer model. Use it to write to one or several variables in a slave system on a TCP/IP network.
ComliSBWriteDT	N	Use the COMLIWriteDT function block to transmit date and time from the local system to a remote SattCon system on a TCP/IP network. (Or a system supporting COMLI) and to update the system time. The function block uses the local time.

SerialCommLib

The library SerialCommLib library contains function block types for communication with external devices via serial channels with user-defined protocols; for example, printers, terminals, scanner pens, etc.

Table 63. Serial channel function block types

Function Block Type	TC	Description
SerialConnect	N	Opens and closes a defined serial communication channel.
SerialSetup	N	Changes serial communication settings.
SerialWriteWait	N	Writes a string and waits for a reply.
SerialListenReply	N	Listens for a string and sends a reply.
SerialWrite	N	Writes a string.
SerialListen	N	Listens for a string.

INSUMCommLib

The InsumCommLib library contains function block types for communication with INSUM devices. INSUM (Integrated System for User-optimized Motor control) is a system for protection and control of motors and switch gear. The communication interface CI857 provides communication for the AC 800M controller with the INSUM system via TCP/IP.

Table 64 describes the function block types present in INSUMCommLib.

Table 64. INSUM function block types

Function Block Type	TC	Description
INSUMConnect	N	The INSUMConnect function block establishes a connection to a concerned INSUM Gateway. This connection can be used as a base for other INSUM specific IEC 61131-5 function blocks to access the proper INSUM data. The connection can be established between the calling communication partner and the remote communication partner. The 'Id' output of INSUMConnect provides the communication channel description, which can be used as the 'Id' input to other communication function blocks. INSUMConnect also gives a status information about the connected gateway, for supervision.

Table 64. INSUM function block types (Continued)

Function Block Type	TC	Description
INSUMReceive	N	The INSUMReceive function block is used to read a point out variable of a specified type in an INSUM device. There are two parameters that point out the data to be read. The DevicePos parameter specifies the position of the INSUM Device in the HW tree below the INSUM Gateway. The value of this parameter ranges from 101 to 432. The NVIndex parameter is the index of the Network Variable. These parameter Ids must be connected to the Id parameter of INSUMConnect.
INSUMWrite	N	The INSUMWrite function block writes data from an IEC 61131 variable to a network variable in an INSUM device. For example, INSUMWrite writes an MCU data that is triggered from an OperateIT, which is used for sending command (Start/Stop/Reset) and handling Pass Control Access.

UDPCommLib

The UDP Communication Library (UDPCommLib) contains function block types that are used for self-defined UDP communication. These function blocks are used when the controller needs to communicate with external equipment. The used protocol is UDP, running on Ethernet.

Table 65. UDP function block types

Function Block Type	TC	Description
UDPConnect	N	This function block is used to open and close a self-defined UDP communication channel. It allocates a local UDP port (in parameter) when enabled. The port is also opened in the controller's firewall. The UDPConnect is the first function block in the chain when implementing a self-defined UDP communication.
UDPWrite	N	The UDPWrite function block writes a structure of dints or dwords to a receiving device. It is used to write/send a UDP datagram on the network. The UDPWrite must be combined with the UDPConnect in order to send UDP messages on the network.
UDPRead	N	The UDPRead function block reads a struct of dints or dwords from a sending device. It is used to read/receive a UDP datagram from the network. The UDPRead must be combined with the UDPConnect in order to recieve UDP messages from the network.

TCPCommLib

The TCP Communication Library (TCPCommLib) contains function block types that are used for self-defined TCP communication. These function blocks are used when the controller needs to communicate with external equipment. The used protocol is TCP, running on Ethernet.

Table 66. TCP function block types

Function Block Type	TC	Description
TCPClientConnect	N	The TCPClientConnect function block is used to open and close a TCP connection to a remote TCP server. The server's port is also opened in the controller's firewall. The local port used will be automatically chosen by the controller. The TCPClientConnect is the first function block in the chain when implementing self-defined TCP client for communication with a remote TCP server.
TCPServerConnect	N	The TCPServerConnect function block is used to let the controller become a TCP server waiting for connection requests initiated by other TCP clients on the network. It allocates a local TCP port (in parameter) when enabled. The port is also opened in the controller's firewall. The TCPServerConnect is the first function block in the chain when implementing self-defined TCP server for communication with one or upto ten remote TCP clients.
TCPWrite	N	The TCPWrite function block writes a structure of dints or dwords to a receiving device. It is used to write/send a TCP message on the network. The writing is acknowledged by the remote node. The TCPWrite must be combined with the TCPClientConnect or TCPServerConnect in order to send TCP messages on the network.

Table 66. TCP function block types (Continued)

Function Block Type	TC	Description
TCPRead	N	The TCPRead function block reads a struct of dints or dwords from a sending device. It is used to read/receive TCP messages from the network. The TCPRead must be combined with the TCPClientConnect or the TCPServerConnect in order to recieve TCP messages from the network.

Alarm and Event Library

The library AlarmEventLib contains function block types and control module types for alarm and event handling, which include detection and notification. Alarm state handling and alarm acknowledgement are also included.

An alarm data model based on OPC Alarms and Events is used.

Table 67. Function block types in AlarmEventLib

Function Block Type	TC	Description
AlarmCondBasic	Y	This function block should be used to monitor a boolean signal for which changes need to be acknowledged to ensure that attention is paid to a problem. The function block support only internal conditions and it is not possible to invert the signal.
AlarmCond	Y	Defines an alarm condition that follows a condition state diagram. It monitors the changes in an input parameter (boolean type) to detect an abnormal condition. Other inputs include acknowledge, disable and enable. A condition state output parameter presents the state of the alarm. The parameter *AckRule* (integer) defines the properties for acknowledgement handling. The source name *SrcName* identifies the name of the object in which the alarm occurred. *Class* and Severity are inputs that can be used to categorize the event that occurs when the alarm changes state. It is possible to monitor a signal on an I/O device which reads the time stamp of the I/O changes on the device. TransitionTime determines the time of the event occurrence when the signal changes. If the value is equal to the default value (the time), then it is read inside this FB.
AttachSystemAlarm	Y	Makes it possible to present the current condition state of a specified system alarm.

Table 67. Function block types in AlarmEventLib (Continued)

Function Block Type	TC	Description
PrintAlarms	N	Prints alarm conditions. On request, this function block prints a list of the alarms currently defined in the control system where the block executes. The printer should be connected directly to a serial port, on the control system, corresponding to the *Channel* parameter, and should support the 8-bit character set.
PrintEvents	N	Prints events continuously. This function block prints both simple events and condition-related events to a printer connected locally to the control system. This means that as soon as an *AlarmCond* changes state (for example, from inactive to active) this information can be sent to the printer. However, only events generated in the system to which the printer is connected can be printed. The printer should be connected to the port corresponding to the *Channel* parameter, and should support the 8-bit character set.

Table 67. Function block types in AlarmEventLib (Continued)

Function Block Type	TC	Description
SimpleEventDetector	Y	Generates a simple event on a Boolean type condition. This function block supervises a Boolean type signal. When the signal changes value, a simple event is generated. You can use this function block to detect, for example, the start and stop of process objects. There is no acknowledgement handling; otherwise the function block resembles *AlarmCond*. *Severity* and *Class* are inputs that can be used for sorting the events. It is possible to monitor a signal on an I/O device which reads the time stamp of the I/O changes on the device.
DataToSimpleEvent	N	This function block type generates a simple event with additional data that is user-defined. It is possible to subscribe to simple events using OPC Alarm and Event. You can also print out the simple events on a locally connected printer, but the user-defined data is not printed out. If the function block is used in a Batch object, its recipe parameters can be logged together with user-defined data.
SystemAlarmCond	Y	Internal monitored Signal.

Table 67. Function block types in AlarmEventLib (Continued)

Function Block Type	TC	Description
ProcessObjectAE	Y	This function block generates alarm or events depending on the input parameter AEConfig. It has two condition inputs that generates alarms or event on the egdes of the condition signals. This function block is called from inside the template objects of ProcessObjBasicLib and ProcessObjExtLib.
SignalAE	Y	This function block is used along with AlarmCond and SimpleEventDetector objects to handle alarm and event.

Table 68. Control module types in AlarmEventLib

Control Module Type	TC	Description
AlarmCondBasicM	Y	Defines an alarm condition and detects condition state changes. *AlarmCondBasicM* has reduced functionality compared to *AlarmCondM*. In return, it consumes less memory. This is the control module equivalent of the function block type *AlarmCondBasic,* described above.
AlarmCondM	Y	Defines an alarm condition that follows a condition state diagram. This is the control module equivalent of the function block type *AlarmCond,* described above. TransitionTime determines the time of the event occurrence when the signal changes. If the value is equal to the default value (the time), then it is read inside this CM.

Control Libraries

The libraries ControlBasicLib, ControlSimpleLib, ControlStandardLib, ControlObjectLib, ControlExtendedLib, ControlAdvancedLib and ControlFuzzyLib contain predefined function block types and control module types. The library ControlSupportLib contains invisible objects and is used by the objects in other Control libraries.

ControlBasicLib

The library ControlBasicLib contains predefined function block types. These are complete working modules that can be used as-is. Therefore, you should not use these modules to create new ones. The PID functions have feedforward, Tracking, 3-position output and Autotuner control functions. Function blocks have been used to construct this library.

PID type Function Blocks

Table 69. PID type function blocks

Function Block Type	TC	Description
PidLoop	N	This function block type defines a simple control loop with a PID controller and a filter. The function block is to be connected directly to I/O via structured variables of the predefined data type *RealIO*. The PID controller of the control loop has feedforward and Tracking functions and an Autotuner. The Autotuner calculates the controller gain, integration time and derivation time based on a simple relay experiment. The PID controller has integrator wind-up prevention and bumpless transfer between modes. It also has built-in deviation alarm limits.
PidLoop3P	Y	This function block type defines a simple control loop with a three-position controller. It is identical to PidLoop except that the analog output has been replaced by two binary outputs, to increase or decrease the actuator position, or to keep it constant.

Table 69. PID type function blocks (Continued)

Function Block Type	TC	Description
PidCascadeLoop	N	This function block type defines a cascade control loop with two PID controllers. The function block is to be connected directly to I/O via structured variables of the predefined data type *RealIO*. The PID controller is identical to that in PidLoop. The integrator wind-up prevention is extended to also prevent wind-up in the master controller when the output of the slave controller is limited.
PidCascadeLoop3P	Y	This function block type defines a cascade control loop with a three-position controller as a slave controller. It is identical to PidCascadeLoop except that the analog output from the slave controller has been replaced by two binary outputs, to increase or decrease the actuator position, or to keep it constant.

ControlSimpleLib

The library ControlSimpleLib contains a number of function blocks that are intended to be used for designing simple control loops. All function blocks in this library can be used in time-critical tasks.

Table 70. Simple Control function blocks

Function Block Type	**TC**	**Description**
PidSimpleReal	Y	This function block is a simple PID controller with less functionality than the PidLoop function blocks and the PidCC module. PidSimpleReal is, however, less time and memory consuming. This controller supports backtracking, tracking, manual control and output limiting. All transitions from limiting, tracking and manual mode are bumpless. Interactive graphics facilitate set-up and maintenance of the controller.
LeadLagReal	Y	This function block is used either as a Lead or Lag function; that is, as a derivative or integrating limiter. The actual function (Lead/Lag) is determined by the relation between the two input time constants, *LeadT* and *LagT*. The function block can be forced to track an external signal. Transition from tracking is bumpless.
FilterReal	Y	This function block is a single-pole, low-pass filter. The functionality obtained is mainly the same as with the control module FilterCC. The output can be forced to track an external signal. Transition from tracking is bumpless. This function block can be used in a time-critical task.

Table 70. Simple Control function blocks (Continued)

Function Block Type	**TC**	**Description**
Filter2PReal	Y	This function block is a low-pass filter with one zero and two complex poles. The output can be forced to track an external signal. Transition from tracking is bumpless.
PiecewiseLinearReal	Y	This function block is a look-up table with a number of predefined input-output pairs. Values between these pairs are calculated by linear interpolation. The function block can be used to define a non-linear function y = f(x). The maximum number of data points is 21 and an Interaction window facilitates data input. The only restriction on the data points is that the x-values must be increasing. It is also possible to calculate the pseudo-inverse of the defined function for a given input, *InInverse*. The functionality of this function block is the same as that of the Control module PiecewiseLinearCC.
PiecewiseLinear2DReal	N	This function block is an extension of the PiecewiseLinearReal function block. It accepts two inputs, which means that a non-linear surface, z = f(x,y) can be specified. The restriction on the x-, and the y-values is that they must be increasing. A maximum of 21 x-values and 11 y-values can be specified; that is, as 231 data points. An Interaction window can be used to edit the data.
VelocityLimiterReal	Y	This function block is a ramp function used to limit the velocity of the change of a signal. The output can be forced to track an external signal. Transition from tracking is bumpless.

Table 70. Simple Control function blocks (Continued)

Function Block Type	TC	Description
AccelerationLimReal	Y	This function block is a ramp function used to limit the velocity of the change of a signal. The output can be forced to track an external signal. Transition from tracking is bumpless. It limits both the velocity and acceleration.
IntegratorReal	Y	This function block is a regular integrator with the same functionality as the control module IntegratorCC. The output is limited and may be forced to track an external signal. All transitions from tracking and limiting are bumpless.
DerivativeReal	Y	This function block is a combined first-order low-pass filter and a differentiator. The filter is used to smooth the derivative action. The functionality is similar to the existing control module DerivativeCC. The output may be forced to track an external signal. Transition from tracking is bumpless.
ThreePosReal	Y	This function block is a three-position converter from a *real* input to two Boolean outputs (increase/decrease). It can be used with or without feedback from the actuator. It is similar to the existing control module ThreePosCC.

ControlStandardLib

The library ControlStandardLib contains control module types that can be used when designing your own standard control loops. They are used for continuous control; for example, PID loops. They can be used for stand-alone or cascade control in master/slave configurations. The PID functions have Feedforward, Tracking, Backtracking, Three-position output, Hand/Auto and Autotuner control functions. Control modules have been used to create this library, and they have associated engineering and operator graphics.

The control modules can be connected to other control modules in ControlExtendedLib, ControlAdvancedLib, or ControlFuzzyLib, in order to construct more advanced control loops. The control modules are connected using graphical connections. Information is sent forwards as well as backwards in the control loop. The automatic code sorting is used to obtain good performance related to bumpless transfer and integrator wind-up in the entire control loop.

PID Control Modules

Table 71. PID control modules

Control Module Type	TC	Description
PidCC	N	This is a standard PID controller. It has all the functions of the PID controllers of the function blocks described above. But since it is a control module, it can be connected to other control modules in order to create more advanced control loops than those that can be obtained with the function blocks described above.See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept on page 367.*
PidSimpleCC	Y	This control module is a low-functionality PID-controller compared to the already-existing PID modules, PidCC and PidAdvancedCC. The PidSimpleCC module, however, consumes less time and memory. Interactive graphics facilitate set-up and maintenance of the controller. The main inputs and the output are of ControlConnection type, which means that backtracking and limiting are handled automatically.

Process I/O Control Modules

Most of the process I/O modules used for continuous control operate with the data type Control Connection (“CC”).

Table 72. Process I/O control modules

Control Module Type	TC	Description
AnalogInCC	Y	Analog Input
AnalogOutCC	N	Analog Output
ThreePosCC	Y	This control module is used as the end of a three-point control loop with digital outputs. The control module input is an analog signal that is compared with a feedback signal from the valve position, or is generated internally.
PulseWidthCC	N	This control module is used as the graphical end of a control loop with pulse-modulated digital output. The control module input is an analog signal that generates the duty cycle of the output signal. The cycle time is defined via the parameter interface.

Manual Control Modules

Table 73. Manual control modules

Control Module Type	**TC**	**Description**
ManualAutoCC	N	With this control module it is possible to enter values into a control loop manually, and supervise the control values graphically in histograms. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*

Conversion Control Modules

Table 74. Conversion control modules

Control Module Type	**TC**	**Description**
RealToCC	Y	This control module is designed to collect each component to form a data type ControlConnection.

Table 74. Conversion control modules (Continued)

Control Module Type	TC	Description
CCToReal	Y	This control module functions as an adapter from a signal of data type ControlConnection. It divides the ControlConnection into its components to give a signal of data type *real*.
CCToInteger	Y	This control module functions as an adapter from a signal of data type ControlConnection. It divides up ControlConnection into its components, to give a signal of data type *integer*. The *real* value of the parameter *In* is converted to the integer parameter *Out*. At conversion the hysteresis specified by the parameter *Hysteresis* is used.

Branch Control Modules

Table 75. Branch control modules

Control Module Type	TC	Description
BranchCC	Y	This control module divides the control loop connection structure into two equal branches.
Branch4CC	Y	This control module divides the control loop connection structure into four equal branches.
SplitRangeCC	Y	This control module divides the control loop connection structure into two branches in relation to their ranges.
MidRangeCC	Y	This control module is a ControlConnection with two branches, one fast and one slower branch. The fast branch acts more equal to changes in the signal, and then it is forced to work around the mid-point of its operating range, as the slower branch takes over control. This control module can be used in cases where, for example, two valves are acting on the same flow. One of the valves is a smaller, but faster valve, used to control small disturbances in the flow. The other valve is a larger valve that cannot work quickly, but has a wider operating range.
CommonRangeCC	Y	This control module divides the control loop connection structure into two branches with a specified ratio between the signal levels.

Table 75. Branch control modules (Continued)

Control Module Type	TC	Description
TapCC	Y	This control module divides the control loop connection structure into two branches, one in which backtracking is possible, and the other in which it is not.
TapRealCC	Y	This control module extracts the value component from the control loop connection structure to produce a real value.

Supervisory Control Modules

Table 76. Supervisory control modules

Control Module Type	TC	Description
Level2CC	N	This control module type is for level detection and alarm purposes and has two detection levels, H (High), and L (Low). Supervision may be absolute or relative to a reference signal. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
Level4CC	N	This control module type is for level detection and alarm purposes and has four detection levels, H (High), HH, and L (Low), LL,. Supervision may be absolute or relative to a reference signal.See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*

Table 76. Supervisory control modules (Continued)

Control Module Type	TC	Description
Level6CC	N	This control module is used for level detection purposes. It has six detection levels (LLL, LL, L, H, HH, HHH). L = Low, H = High See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
SignalSupervisionCC	N	This control module is used to manage erroneous signal status collected from the transmitters or from the interface system. Three different modes are available: allow the signal to pass through without any interference, freeze the output, or switch over linearly to a predetermined value. The latter two cause an alarm condition to be sent, if configured. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*

Selector Control Modules

Table 77. Selector control modules

Control Module Type	TC	Description
SelectorCC	Y	This control module selects one of two inputs of data type *ControlConnection*. Selection is made based on a Boolean signal. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
Selector4CC	Y	This control module selects one out of four inputs of data type *ControlConnection*. Selection is based on an integer signal. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
SelectGoodCC	Y	The first detected valid signal of type *ControlConnection* is selected. If no valid signal is detected, *Out* is a copy of *In1* structure.
SelectGood4CC	Y	The first detected valid signal of type *ControlConnection* is selected. If no valid signal is detected, *Out* is a copy of *In1* structure.
MaxCC	Y	The control module MaxCC computes the larger (maximum) value of two input signals of data type *ControlConnection* and writes it to the output signal.

Table 77. Selector control modules (Continued)

Control Module Type	**TC**	**Description**
Max4CC	Y	The control module Max4CC computes the largest (maximum) value of four input signals of data type *ControlConnection* and writes it to the output signal.
MinCC	Y	The control module MinCC computes the smaller (minimum) value of two input signals of data type *ControlConnection* and writes it to the output signal.
Min4CC	Y	The control module Min4CC computes the smallest (minimum) value of four input signals of data type *ControlConnection* and writes it to the output signal.

Limiter Control Modules

Table 78. Limiter control modules

Control Module Type	TC	Description
LimiterCC	Y	LimiterCC limits the signal so that it does not increase above the upper limit, or decrease below the lower limit.
LimiterHighCC	Y	LimiterHighCC limits the signal so that it does not increase above the upper limit.
LimiterLowCC	Y	LimiterLowCC limits the signal so that it does not decrease below the lower limit.
VelocityLimiterCC	N	VelocityLimiterCC limits the velocity of the signal. It can be used, for example, to create a linear movement function between a starting point and a target. This will slow down changes in the output signal to avoid rapid steps.
AccelerationLimCC	Y	AccelerationLimCC limits the velocity of the signal. It can be used, for example, to create a linear movement function between a starting point and a target. This will slow down changes in the output signal to avoid rapid steps. It limits both the velocity and acceleration.

ControlObjectLib

The Control Object Library (ControlObjectLib) provides function blocks and control modules.

All the control modules inside the ControlObjectLib work as templates. The control modules (Mimo22CC, Mimo41CC, and Mimo44CC) handle multiple inputs and multiple outputs in both forward and backward communication direction.

Control Object Function Blocks

Table 79. Control Object Function Blocks

Function Block Type	TC	Description

AddRangeWithGain	Y	This function is used when the user wishes to calculate the output-range if the inputs added are affected by gains, i.e.
		Out = a*In1 + b*In2, in this case the Out-Range is:
		OutMax = a*In1Max + b*In2Max and
		OutMin = a*In1Min + b*In2Min

AssignBTInputs2	Y	This function checks if the input is backtracking or not recursively.
AssignBTInputs4	Y	This function block assigns which input is backtracking.
CalcBackValue	Y	The Relative value of the backtracked value is passed to this function block to calculate the backward value of the input. If the input is used for backtracking then *InUsedForBT* is set to 'true'.
LevelHL	Y	Produces a warning if Output reaches a certain High or Low value.

Table 79. Control Object Function Blocks (Continued)

Function Block Type	TC	Description
Out21BackwardFunction	Y	For every user defined function block in the forward direction a corresponding function block in the backward direction is needed. In this case Out21BackwardFunction is the corresponding function block for Out21Function.
Out21Function	Y	This is an example of a user defined function. In this example the function is linear. Out1 := a1*In1 + b1*In2
Out41BackwardFunction	Y	For every user defined function block in the forward direction a corresponding function block in the backward direction is needed. In this case Out41BackwardFunction is the corresponding function block for Out41Function.
Out41Function	Y	This is an example of a user defined function. In this example the function is linear. Out1 := a1*In1 + b1*In2 + c1*In3 + d*In4
OutVotedBackwardFunction	Y	The corresponding backward function block to OutVotedFunction.
OutVotedFunction	Y	This user defined function block is executed if Voted.

Table 79. Control Object Function Blocks (Continued)

Function Block Type	TC	Description
PrepareBacktrack	Y	The equation in backward direction is adjusted. All the inputs which cannot be used for backtracking will affect the backtracked value for the remaining Inputs. Out := a*In1 + b*In2, If In2 cannot be used then the new Out := Out - b*In2; so that the remaining system is Out := a*In1;
VotedCmdHandler	Y	This function block shall take care of the voted actions.

Control Object Control Modules

Table 80. Object control modules

Control Module Type	TC	Description
Mimo22CC	N	The Mimo22CC control module is a module that can handle 2 inputs and 2 outputs in both forward direction and partially in backward direction. The Mimo22CC control module can be run under time critical condition. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
Mimo41CC	N	The Mimo41CC control module is a module that can handle 4 inputs and 1 output in both forward direction and in backward direction. The Mimo41CC control module can be run under time critical condition. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
Mimo44CC	N	The Mimo44CC control module is a module that can handle 4 inputs and 4 outputs in both forward direction and partially in backward direction. The Mimo44CC control module can be run under time critical condition. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*

ControlSolutionLib

All the control module types in Control Solution library (ControlSolutionLib) provide a complete control solution, intended to be used directly in an application. The user requires only to connect the control module to I/Os and in some cases set some configuration parameters.

The control module types are ready-to-use solutions for frequently occurring control processes found at customers. They consists of a control solution with basic control module types, alarm handling, process graphics, preconfigured trend displays with logging, group displays, and an overview display.

The users may use the solutions directly as they are, or create own types by making copies and change these to fit an intended usage, which may be level control, flow control, etc. These new types can then be preconfigured with specific default values for controller tuning, alarm limits, data collection settings etc.

Table 81. Control Solution control module types

Control Module Type	TC	Description
CascadeLoop	N	This control module provides a complete cascade loop control solution for connection to I/O. The control module provides master and slave PID control with signal supervision, velocity limiter, alarm handling, trending, and operator graphics.
FeedforwardLoop	N	This control module provides a complete feedforward loop control solution for connection to I/O. The control module provides PID feedback control and dynamic feedforward control. The loop comes with signal supervision, velocity limiter, alarm handling, trending, and operator graphics.

Table 81. Control Solution control module types (Continued)

Control Module Type	**TC**	**Description**
MidrangeLoop	N	This control module provides a complete midrange loop control solution for connection to I/O. The control module provides single PID control of two outputs in parallel. The loop comes with signal supervision, velocity limiter, stiction compensation, alarm handling, trending, and operator graphics.
OverrideLoop	N	This control module provides a complete override loop control solution for connection to I/O. The control module provides a minimum selector with four controllers; one master PID controller and three override controllers. The loop comes with signal supervision, velocity limiter, alarm handling, trending, and operator graphics.
SingleLoop	N	This control module provides a complete single loop control solution for connection to I/O. The control module provides PID control with signal supervision, velocity limiter, stiction compensation, alarm handling, trending, and operator graphics.

ControlExtendedLib

The library ControlExtendedLib contains control modules for arithmetic and signal processing for continuous control; for example, PID loops. The control functions are available as control modules, and they also have associated engineering and operator graphics.

The control modules can be connected to other control modules in ControlStandardLib, ControlAdvancedLib or ControlFuzzyLib in order to construct more advanced control loops. The control modules are connected via graphical connections. Information is sent forward as well as backward in the control loop. The automatic code sorting is used to obtain good performance related to bumpless transfer and integrator wind-up in the entire control loop.

Using the control modules in ControlExtendedLib together with those in ControlStandardLib and ControlAdvancedLib, it is possible to construct control loops with high functionality. Arithmetic operations can be performed on the control signals. The control signals can also be processed in several ways; for example, filtered or integrated.

Arithmetic Control Modules

Table 82. Arithmetic control modules

Control Module Type	TC	Description
AddCC	Y	Addition, two inputs
SubCC	Y	Subtraction, two inputs
MultCC	N	Multiplication, two inputs
DivCC	N	Division, two inputs
BiasCC	N	Addition, two inputs
RatioCC	N	Multiplication, two inputs
SqrtCC	Y	Square root, one input
XRaisedToYCC	Y	Calculates the value In1 raised to In2.

Signal Handling Control Modules

Table 83. Signal handling control modules

Control Module Type	TC	Description
DerivativeCC	Y	Derivation.
IntegratorCC	Y	Integration with reset and hold function.
FlowCC	Y	This control module calculates the mass flow from a differential pressure (orifice plate) compensated by actual temperature and pressure.
Mean4ExcludeBadCC	Y	This control module calculates the mean value of the inputs where extreme values are excluded.

Table 83. Signal handling control modules (Continued)

Control Module Type	TC	Description
Mean8ExcludeBadCC	Y	This control module has the same functionality as Mean4ExcludeBadCC above. The only difference is the number of inputs.
Mean12ExcludeBadCC	Y	This control module has the same functionality as Mean4ExcludeBadCC above. The only difference is the number of inputs.
FilterCC	Y	This is a first-order, low-pass filter.
Filter2PCC	Y	This control module is a low-pass filter with one zero and two complex poles.
LeadLagCC	Y	This control module is used either as a Lead or Lag function, derivative or integrating limiter. The actual function (Lead/Lag) is determined by the relation between two input time constants.
DelayCC	N	This is a delay control module for the ControlConnection structure.
StateCC	Y	This control module delays the forward and backward components by one cycle to avoid program loops.
TimeAverageCC	Y	This control module calculates the time average value of the input over a specified number of samples.

Signal Conditioning Control Module

Table 84. Signal conditioning control module

Control Module Type	TC	Description
PiecewiseLinearCC	Y	Piece-wise linear transformation of the input signal.
PiecewiseLinear2DCC	N	This control module is a look-up table with a number of predefined input-output data pairs. It takes two inputs, which means that a nonlinear surface, z = f(x,y) can be specified. A maximum of 21 x-values and 11 y-values can be specified, 231 data points. An Interaction window can be used to edit the data.
PiecewiseLinear-Extension	Y	This control module is used as an add-in module to the PiecewiseLinearCC module. The latter is only able to handle 21 data points. This control module makes it possible to add another 20 data points. It is also possible to connect a control module to an existing control module of the same type. Thus, the number of points is unlimited.

ControlAdvancedLib

The library ControlAdvancedLib contains control module types for advanced, continuous control, PID loops. The control functions are available as control modules, and they also have associated engineering and operator graphics.

The control modules can be connected to other control modules in ControlStandardLib, ControlExtendedLib or ControlFuzzyLib in order to construct more advanced control loops. The control modules are connected via graphical connections. Information is sent forward as well as backward in the control loop. The automatic code sorting is used to obtain good performance related to bumpless transfer and integrator wind-up in the entire control loop.

The advanced PID controller has all the functionality of the previously described PID controllers. In addition, it can be configured for continuous adaptation of the controller parameters. It can also be configured as a predictive PI; that is, as a PPI, controller and it has a gain scheduler (also called a parameter scheduler).

PID Control Modules

Table 85. PID control modules

Control Module Type	TC	Description
PidAdvancedCC	N	In addition to the standard functions, this control module type contains a gain scheduler and an adaptive controller. It has a PPI (predictive PI) controller for processes with long dead times. The loop assessment tools can detect oscillatory or sluggish behavior of the control loop. The Autotuner is more advanced than that in the other PID controllers.See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.

Additional Control Modules

Table 86. Additional control modules

Control Module Type	TC	Description
StictionCompensator	Y	An optional extension to the AnalogOutCC control module to add pulses to the output of the AO to avoid the effects of sticky pneumatic valves.
DecoupleFilterCC	N	A filter introduced to decouple the process dynamics so that the total system behaves as two independent systems.

ControlFuzzyLib

The library ControlFuzzyLib contains control module types, which are building blocks for fuzzy controllers. A fuzzy controller is constructed by connecting control modules from ControlFuzzyLib. No programming is necessary. ControlFuzzyLib also has three templates (not protected), which consist of three different fuzzy controllers. The control module types can be copied to your own library and then modified.

Table 87. Control module types in ControlFuzzyLib

Control Module Type	TC	Description
FuzzyController1CC	N	This control module contains a very small configuration of a fuzzy controller.
FuzzyController2CC	N	This control module has the same structure as that of FuzzyController1CC, but contains a much larger configuration.
FuzzyController3CC	N	This control module is the same as FuzzyController1CC, but with no setpoint.
FuzzySpPvIn	N	This control module computes the control deviation EOut (Setpoint Process value) and its first and second derivatives. These signals are inputs to the InputMembership control modules. This control module can switch between an external and an internal setpoint. The process value can be filtered in a low-pass filter. Three Pv alarm levels and one deviation alarm can be displayed in the history and bar graphs. The control module also has an optional facility for Process value tracking. The outputs from the control module, the control deviation and its two first derivatives can be simulated by the operator. This facility can be used to test the fuzzy logic part of the controller.

Table 87. Control module types in ControlFuzzyLib (Continued)

Control Module Type	TC	Description
FuzzyPvIn	N	This control module makes the process value and its derivative available to the InputMembership control modules. The process value can be filtered in a low-pass filter. Three absolute alarm levels can be displayed in the history and bar graphs. The outputs from the control module, the process value, and its derivative, can be simulated by the operator. This facility can be used to test the fuzzy logic part of the controller.
FuzzyOut	N	This control module takes a defuzzyfied output from the fuzzy logic part of the controller and computes the output to the process. The signal can be integrated or not. It can also be set in Manual or Automatic mode or it can track an external value. A feedforward signal can be added, and an anti-reset wind-up function is provided.
InputMembership	N	This control module defines an input membership function for the fuzzy logic part of the controller. For every value of the input it computes the degree of membership to the corresponding fuzzy set. The control module is also used, together with the FuzzyCondition control module, to define the fuzzy conditions.

Table 87. Control module types in ControlFuzzyLib (Continued)

Control Module Type	TC	Description
OutputMembership	N	This control module defines an output membership function for a fuzzy rule. It computes the current membership function, which is equal to the defined membership function, multiplied by the degree of satisfaction of the rule. The current membership functions for a number of rules can then be combined into a membership function for the output of the controller. This is done by computing the envelope, the maximum of all the current membership functions at every point. This is done in the Defuzzyfication control module.
Defuzzyfication	N	This control module computes the envelope of all the connected output membership functions. It also computes the center of gravity of the envelope curve. The center of gravity is regarded as the defuzzyfied output from the fuzzy logic part of the controller.

Table 87. Control module types in ControlFuzzyLib (Continued)

Control Module Type	TC	Description
FuzzyCondition6, -12 and -18	N	The control modules FuzzyCondition6, FuzzyCondition12 and FuzzyCondition18 define and evaluate a fuzzy condition. The condition is defined as a fuzzy AND condition between a number of selected input membership functions. The input membership functions may, or may not, be inverted before the condition is formed. Together with other fuzzy conditions, the defined fuzzy condition is used in one or more fuzzy rules. The fuzzy AND condition is defined as the minimum value of the fuzzy variables included.
FuzzyRule5, -10, -15, -20, -25 and -30	N	The control modules FuzzyRule5, FuzzyRule10, FuzzyRule15, FuzzyRule20, FuzzyRule25, and FuzzyRule30 define and evaluate a fuzzy rule. The condition of the rule is defined as a fuzzy OR expression between a number of fuzzy conditions defined in FuzzyCondition control modules. The conditions from the FuzzyCondition control modules may, or may not, be inverted before the condition of the FuzzyRule control module is formed. The result of the rule is the degree of satisfaction of the rule. The degree of satisfaction is used to compute the output membership function for the rule. The fuzzy OR condition is defined as the maximum value of the fuzzy variables included.

Table 87. Control module types in ControlFuzzyLib (Continued)

Control Module Type	TC	Description
FuzzyProgramControl	N	This control module is used to toggle the Edit mode of the fuzzy logic part of the controller on and off. The fuzzy controller is fully operational in both modes.
FuzzyPres	N	This control module is an icon for the controller. It is intended to be built into the controller and displayed in the Control Module diagram via a control module selector.
FuzzyControlIcon	N	This control module type is the symbol for a controller that contains fuzzy logics.
FuzzyIcon	N	This control module type is the symbol for fuzzy logics.

Batch Library

The library BatchLib contains control module types for batch control and for control of other discontinuous processes. It can be used together with any batch system which communicates via OPC Data Access and which supports the S88 state model for procedural elements.

The control modules described here are used for the interaction between the control application for an Equipment Procedure Element (for example, a phase or an operation) and the Batch Manager.

Table 88. Equipment procedure element control module types

Control Module Type	TC	Description
EquipProcedure Template	N	A template control module for designing Equipment Procedure Elements, the control logic for phases, operations, and so on. It handles the interaction with the Batch Manager.
EquipProcedureCore	N	Handles the standard ISA/S88-based states and modes of an Equipment Procedure Element.
EquipProcedureIcon	N	This control module type is an icon for EquipProcedureTemplate and EquipProcedureCore.
InfoEquipProcedure Template	N	This control module type defines the pop-up interaction window for the procedural element.

Process Object Libraries

The libraries ProcessObjBasicLib, ProcessObjExtLib, ProcessObjDriveLib and ProcessObjInsumLib contain function block types and control module types for controlling motors, valves, ABB Drives and Insum Devices in a process application.

ProcessObjBasicLib

The library ProcessObjBasicLib contains basic core function block types for valve and motor control functions. They are to be used when designing your own function block types. The core function block types should be encapsulated in your own function block type. These core function block types are protected and cannot be changed.

Table 89. ProcessObjBasicLib function block types

Function Block Type	TC	Description
BiCore	N	Bi-directional object: Basic function block with two or three outputs and 0, 2 or 3 feedback signals. This function block can be used to represent a two-speed motor or a forward-backward motor. Parameters are available for the connection of an external panel for manual operation of the object, and interlock and force signals for connection of different safety interlocking. It is also possible to connect an external fault signal.
BiDelayOfCmd	Y	This function block type is used to avoid spurious commands in auto mode. For example, if a level detector the object to start, a disturbance pulse will not be able to start the object.
BiSimple	N	This function block type is suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) process object. For greater flexibility, the extensions to the basic BiCore type.
DetectOverrideBi	Y	Detects override commands like Priority and Ilock.
DetectOverrideUni	Y	Detects override commands like Priority and Ilock.
DetectOverrideVoteBi	Y	Detects override commands like Priority and Ilock.

Table 89. ProcessObjBasicLib function block types (Continued)

Function Block Type	TC	Description
DetectOverrideVoteUni	Y	Detects override commands like Priority and Ilock.
DriveCommandSend	N	This function block type sends command data to the connected drive. It can be used as a base for control of ABB Drives ACS800, ACS600 and ACS400 and their corresponding DC drives. The function block can easily be used together with other function blocks to create more complex objects that handle functionality such as modes and HSI. See also the UniCore and BiCore function block descriptions.
DriveStatusReceive	N	This function block type can be used as a base for control of ABB Drives ACS600 and ACS400, and their corresponding DC drives. The function block can easily be used together with other function blocks to create more complex objects that handle functionality such as modes and HSI. This function block provides the user with the ability to start and stop a drive with a chosen setpoint according to the local state matching in the drive.
Jog	Y	This function block handles the Jog functionality implemented in motor objects. Jog is a functionality that starts the motor object in a specified direction during a specified period of time. The Jog function only is applicable in manual mode of the motor object.

Table 89. ProcessObjBasicLib function block types (Continued)

Function Block Type	TC	Description
PrioritySup	Y	This function block type supervises the priority commands and sets the mode to PriorityMode if any of the inputs are active. Alarm situations are also supervised and, if active, an automatic priority to zero is performed.
UniCore	N	Uni-directional object: Basic function block with one or two outputs and 0, 1, or 2 feedback signals. This function block can be used to represent a valve. Parameters are available for the connection of an external panel for manual operation of the object, and interlock and force signals for connection of different safety interlocking. It is also possible to connect an external fault signal.
UniDelayOfCmd	Y	This function block type is used to avoid spurious commands in auto mode. For example, if a level detector the object to start, a disturbance pulse will not be able to start the object.
UniSimple	N	This function block type is suitable for graphical control and supervision of a unidirectional (one activated and one deactivated position) process object. An extension to the basic UniCore type is the graphics functionality.

Table 90. ProcessObjBasicLib control module types

Control Module Type	TC	Description
BiSimpleM	N	This control module type is suitable for graphical control and supervision of a bidirectional (two activated and one deactivated position) process object. The extensions to the basic BiCore type include control module graphics and Interaction windows.
UniSimpleM	N	This control module type is suitable for graphical control and supervision of a unidirectional (one activated position and one deactivated position) process object. The extensions to the basic UniCore type include control module graphics and Interaction windows.

ProcessObjExtendedLib

The library ProcessObjExtLib contains types that are based on the protected core functions available in the ProcessObjBasicLib. Unprotected code is then added to the core.

Table 91. ProcessObjExtLib function block types

Function Block Type	TC	Description
Bi	N	Bi-directional object with alarm and graphics: This function block type is based on the BiCore object. Extensions include alarm handling and a faceplate.
LevelDetection	N	Supervises the level of an input signal to an object.
MotorBi	N	Motor bi-directional object with alarm and graphics: This function block type is based on the BiCore object. Extensions include alarm handling and a faceplate. This function block has additional interlocks, safety commands, and output delay timers.
MotorUni	N	Motor uni-directional object with alarm and graphics: This function block type is based on the UniCore object. Extensions include alarm handling and a faceplate. This function block has additional interlocks, safety commands, and output delay timers.
OETextBi	Y	This function block concatenates text strings for the alarm printouts for Bi type objects.

Table 91. ProcessObjExtLib function block types (Continued)

Function Block Type	TC	Description
OETextUni	Y	This function block generates error-text strings. Calls of the respective types are included in the open code of the corresponding extended process objects.
OETextValveUni	Y	This function block concatenates text strings for the alarm printouts for ValveUni type objects.
OETextValveBi	Y	This function block concatenates text strings for the alarm printouts for Bi type objects.
Uni	N	Uni-directional object with alarm and graphics: This function block type is based on the UniCore object. Extensions include alarm handling and a faceplate.
ValveUni	N	Valve uni-directional object with alarm and graphics: This function block type is based on the UniCore object. Extensions include alarm handling and a faceplate. Some parameters from the UniCore object are not used, and this block serves as a more basic example for a normal valve.
MotorValve	Y	This function block is suitable for graphical control and supervision of a bidirectional (two activated position) motor valve. This function block type is based on the BiCore object. The extensions include alarm handling and faceplate. This function block has additional interlock parameters and priority commands.

Table 92. ProcessObjExtLib control module types

Control Module Type	TC	Description
BiM	N	Bi-directional object with alarm and graphics: This control module type is based on the BiCore alarm handling and a faceplate. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.
MotorBiM	N	Motor bi-directional object with alarm and graphics: This control module is based on the BiCore alarm handling and a faceplate. This control module has additional interlocks, safety commands, and output delay timers.See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.
MotorUniM	N	Motor uni-directional object with alarm and graphics: This control module is based on the UniCore alarm handling and a faceplate. This control module has additional interlocks, safety commands, and output delay timers.See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.

Table 92. ProcessObjExtLib control module types (Continued)

Control Module Type	TC	Description
UniM	N	Uni-directional object with alarm and graphics: This control module type is based on the UniCore alarm handling and a faceplate. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367*.*
ValveUniM	N	Valve uni-directional object with alarm and graphics: This control module type is based on the UniCore alarm handling and a faceplate. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367*.*

Table 92. ProcessObjExtLib control module types (Continued)

Control Module Type	TC	Description
MotorValveM	Y	Bidirectional (two activated position) motor valve object with alarm and graphics. This control module is based on the BiCore object. Extensions include alarm handling and a faceplate. This control module has additional interlocks and output delay timers. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.
MotorValveCC	Y	Motor controlled valve object of open/close type, with alarm and graphics. This control module is based on the BiCore object. Extensions include alarm handling and a faceplate. This control module has additional interlocks and output delay timers. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.

ProcessObjDriveLib

The library ProcessObjDriveLib contains function block types and control module types which can be used to control and supervise ABB Standard and Engineered Drives.

Table 93. ProcessObjDriveLib function block types

Function Block Type	TC	Description
ACStdDrive	N	This function block type can be used to control and supervise ABB Standard AC Drives. This function block type includes Interaction windows implemented in Control Builder M.
DCStdDrive	N	This function block type can be used to control and supervise ABB Standard DC Drives. This function block type includes Interaction windows implemented in Control Builder M.
EngDrive	N	This function block type can be used to control and supervise ABB Engineered AC and DC Drives. This function block type includes Interaction windows implemented in Control Builder M.

Table 94. ProcessObjDriveLib control module types

Control Module Type	TC	Description
ACStdDriveM	N	This control module type can be used to control and supervise ABB Standard AC Drives. This control module type includes Interaction windows implemented in Control Builder M. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367*.*
DCStdDriveM	N	This control module type can be used to control and supervise ABB Standard DC Drives. This control module type includes Interaction windows implemented in Control Builder M. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367*.*
EngDriveM	N	This control module type can be used to control and supervise ABB Engineered AC and DC Drives. This control module type includes Interaction windows implemented in Control Builder M. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367*.*

ProcessObjInsumLib

The library ProcessObjInsumLib library contains function block types and control module types to control and supervise the standard INSUM (INtegrated System for User optimized Motor management) devices MCU (Motor Control Unit) and trip unit for Circuit Breakers. The INSUM devices are connected via an INSUM Gateway and a CI857 Interface module to the AC 800M.

Table 95. ProcessObjInsumLib function block types

Function Block Type	TC	Description
InsumBreaker	N	This function block type is used to control and supervise an INSUM trip unit for circuit breakers. It is based on the UniCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errors communication errors and feedback errors from the device.

Table 95. ProcessObjInsumLib function block types (Continued)

Function Block Type	TC	Description
McuBasic	N	This function block type is used to control and supervise an INSUM MCU. It is based on the BiCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errors communication errors and feedback errors from the device.
McuExtended	N	This function block type is used to control and supervise an INSUM MCU. It is based on the BiCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errors communication errors and feedback errors from the device.

Table 96. ProcessObjInsumLib control module types

Control Module Type	TC	Description
InsumBreakerM	N	This control module type is used to control and supervise an INSUM trip unit for circuit breakers. It is based on the UniCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errorscommunication errors and feedback errors from the device. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also* Signal and Vote Loop Concept on page 367*.*

Table 96. ProcessObjInsumLib control module types (Continued)

Control Module Type	TC	Description
McuBasicM	N	This control module type is used to control and supervise an INSUM MCU. It is based on the BiCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errorscommunication errors and feedback errors from the device. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.
McuExtendedM	N	This control module type is used to control and supervise an INSUM MCU. It is based on the BiCore object for process logic, and INSUMRead and INSUMWrite blocks for the communication with the device. Function blocks for alarm handling are also included for the display of trips, warnings and other errorscommunication errors and feedback errors from the device. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending and receiving object). See also Signal and Vote Loop Concept* on page 367.

Signal Libraries

SignalLib

The library SignalLib contains function block types for analog and digital inputs and outputs. They extend the functionality of I/O signals and application variables with alarm and event handling. The function blocks also provide filtering and error handling. In the faceplates it is possible to force the objects, view trim curves, configure and enable/disable alarms and events, and view/modify parameters.

Table 97. SignalLib function block types

Function Block Type	TC	Description
SignalInReal	Y	SignalInReal extends the functionality of an analog input signal of data type *RealIO* with alarm/event handling for three high levels, three low levels and error handling.
SignalOutReal	Y	SignalOutReal extends the functionality of an analog output signal of data type *RealIO* with alarm/event handling for three high levels, three low levels and error handling.
SignalReal	Y	SignalReal is used to achieve alarm/event handling for an application variable of data type real with up to three high and three low levels.
SignalInBool	Y	SignalInBool extends the functionality for a Digital Input signal of data type *BoolIO* with alarm/event handling, when the input value is different from Normal value. In corresponding faceplates it is possible to force the object, view trim curves, and configure and enable/disable alarms and events.

Table 97. SignalLib function block types (Continued)

Function Block Type	TC	Description
SignalOutBool	Y	SignalOutBool extends the functionality of a digital output signal of data type *BoolIO* with alarm/event handling when the input value is different from Normal value. In corresponding faceplates it is possible to force the object, view trim curves, and configure and enable/disable alarms and events.
SignalBool	Y	SignalBool extends the functionality of an application variable of data type *bool* with alarm/event handling when the input value is different from Normal value. In corresponding faceplates it is possible to force the object, view trim curves, and configure and enable/disable alarms and events.
SignalSimpleInReal	N	An analog input signal, of RealIO data type, extended with alarm and event handling for errors. Filtering and error handling is also provided.
SignalSimpleOutReal	Y	An analog output signal, of RealIO data type, extended with alarm and event handling for errors.

Table 98. SignalLib control module types

Control Module Type	TC	Description
SignalBoolCalcInM	Y	Monitors an application variable, of bool data type. To be used when it requires connections to a Vote control module type. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
SignalBoolCalcOutM	Y	Monitors an application variable, of bool data type. To be used when it requires input connections from a Vote control module type. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
SignalInBoolM	Y	Monitors a digital input signal, of BoolIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
SignalInRealM	Y	Monitors up to three high levels and up to three low levels and errors for an analog input signal, of RealIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*

Table 98. SignalLib control module types (Continued)

Control Module Type	TC	Description
SignalOutBoolM	Y	Monitors a digital output signal, of BoolIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
SignalOutRealM	Y	Monitors up to three high levels and up to three low levels and errors for an analog output signal, of RealIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*
SignalRealCalcInM	Y	Monitors up to three high levels and up to three low levels and errors for an application variable, of real data type. To be used when it requires connections to a Vote control module type. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
SignalRealCalcOutM	Y	Monitors up to three high levels and up to three low levels and errors for an application variable, of real data type. To be used when it requires input connections from a Vote control module type. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*

Table 98. SignalLib control module types (Continued)

Control Module Type	TC	Description
SignalSimpleInRealM	Y	Monitors an analog input signal, of RealIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a sending object). See also Signal and Vote Loop Concept* on page 367*.*
SignalSimpleOutRealM	Y	Monitors an analog output signal, of RealIO data type. See Table 41 on page 378. *This object is a member of the voting logic concept (a receiving object). See also Signal and Vote Loop Concept* on page 367*.*

SignalBasicLib

The SignalBasicLib library contains the function blocks listed in Table 99. Control Builder graphics are included in the objects. The objects in Table 99 have no alarm and event recording, no detection of abnormal input status, and no Inhibit, Enable or override control functions.

Table 99. SignalBasicLib Function Block Types

Function Block Type	TC	Description
SignalBasicInBool	Y	Overview and forcing of boolean input of data type BoolIO.
SignalBasicBool	Y	Overview and maneuvering of boolean variable of data type bool.
SignalBasicOutBool	Y	Overview and forcing of boolean output of data type BoolIO.

Table 99. SignalBasicLib Function Block Types (Continued)

Function Block Type	TC	Description
SignalBasicInReal	Y	Overview and forcing of analog input of data type RealIO.
SignalBasicReal	Y	Overview and maneuvering of analog variable of data type real.
SignalBasicOutReal	Y	Overview and forcing of analog output of data type RealIO.

Supervision Library

SupervisionBasicLib

The SupervisionBasicLib is divided into two libraries.

- SignalSupportLib for support functionality
- SupervisionBasicLib for simple functionality

Table 100. SupervisionBasicLib Function Block Types

Function Block Type	TC	Description
SDInBool	Y	DiffNormal detection for Digital Input signal (BoolIO).
SDBool	Y	DiffNormal detection for Bool variable.
SDOutBool	Y	Control of Digital Safety Output (BoolIO).
SDInReal	Y	Level detection for Analog Input signal (RealIO).
SDReal	Y	Level detection for Real variable.

Table 100. SupervisionBasicLib Function Block Types (Continued)

Function Block Type	TC	Description
SDValve	Y	Control of Safety Valve output (BoolIO).
StatusRead	Y	Reading status output from other FB's in the library.
SDLevel	Y	Shutdown Level object.

Synchronized Control Library

GroupStartLib

The GroupStartLib (Group Start Library) contains control module types used to the control and supervise of process objects in a controller.

Table 101. GroupStartLib control module types

Control Module Type	TC	Description
GroupStartAnd	N	This control module type executes a logical AND between the connected input signals of type GroupStartStepConnection
GroupStartHead	N	This control module type supervises the entire group, keeps track of the alarms generated in the group and detects the connected objects not ready for start in group start mode. The behavior of the group at alarms is also a task for this control module. A process object may be connected to the bottom output node.

Table 101. GroupStartLib control module types (Continued)

Control Module Type	TC	Description
GroupStartObject Template	N	This control module type encapsulates a standard control loop to be connected into the group start environment and is equipped with simulated feedback signals.
GroupStartOr	N	This control module type executes a logical OR between the connected input signals of type GroupStartStepConnection.
GroupStartStep	N	This control module type has four in- and four output nodes to configure the group start preferably by using graphical connections. A process object may be connected to the bottom output node.
GroupStartTestObject	N	This control module type is used to connect in the group start to test the start up configuration and to simulate alarm situations.
GroupStartStandby4, 8, 12	Y	This control module shall be used for standby purposes and to be able to activate a desired number (maximum the number of connected objects on the output terminals) of objects all working together in the process.
InfoParGroupStart ObjectTemplate	N	This control module type contains the graphics of the interaction window of the GroupStartObjectTemplate control module type.

INDEX

Contact us

ABB AB
Control Technologies
Västerås, Sweden
Phone: +46 (0) 21 32 50 00
e-mail: processautomation@se.abb.com
www.abb.com/controlsystems

ABB Automation GmbH
Control Technologies
Mannheim, Germany
Phone: +49 1805 26 67 76
e-mail: marketing.control-products@de.abb.com
www.abb.de/controlsystems

ABB S.P.A.
Control Technologies
Sesto San Giovanni (MI), Italy
Phone: +39 02 24147 555
e-mail: controlsystems@it.abb.com
www.abb.it/controlsystems

ABB Inc.
Control Technologies
Wickliffe, Ohio, USA
Phone: +1 440 585 8500
e-mail: industrialitsolutions@us.abb.com
www.abb.com/controlsystems

ABB Pte Ltd
Control Technologies
Singapore
Phone: +65 6776 5711
e-mail: processautomation@sg.abb.com
www.abb.com/controlsystems

ABB Automation LLC
Control Technologies
Abu Dhabi, United Arab Emirates
Phone: +971 (0) 2 4938 000
e-mail: processautomation@ae.abb.com
www.abb.com/controlsystems

ABB China Ltd
Control Technologies
Beijing, China
Phone: +86 (0) 10 84566688-2193
www.abb.com/controlsystems

Power and productivity
for a better world™