System 800xA Control

AC 800M
Planning

Power and productivity
for a better world™

ABB

System 800xA Control

AC 800M
Planning

System Version 5.1 Feature Pack

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: February 2013
Document number: 3BSE043732-511

TABLE OF CONTENTS

About This User manual

Section 1 - Design Issues

Section 2 - Programming Languages

Section 3 - Programming in Practice

Appendix A - IEC 61131-3 Standard

Appendix B - Naming Conventions and Tools

INDEX

About This User manual

⚠ Any security measures described in this User Manual, for example, for user access, password security, network security, firewalls, virus protection, etc., represent possible steps that a user of an 800xA System may want to consider based on a risk assessment for a particular application and installation. This risk assessment, as well as the proper implementation, configuration, installation, operation, administration, and maintenance of all relevant security related equipment, software, and procedures, are the responsibility of the user of the 800xA System.

This manual provides some guidelines of what to consider when designing an automation solution using Control Software for AC 800M, such as memory consumption, CPU load, and task execution. The manual also contains advice to programmers regarding optimization of code.

The libraries described in this manual conform to the IEC 61131-3 Programming Languages standard, except for control modules and diagrams, which are not supported by this standard.

- Section 1, Design Issues, helps in identifying the issues to consider when planning the automation system. This section also gives advice on how to design the automation system.

- Section 2, Programming Languages, helps to decide which programming language to use. This section is a description of the supported programming languages.

- Section 3, Programming in Practice, gives practical advice on programming. It discusses a number of areas and gives practical advice on how to solve common problems.

In addition, the appendixes describe support information:

- Appendix A, IEC 61131-3 Standard gives a short introduction to the standard.

- Appendix B, Naming Conventions and Tools contains rules and recommendations for naming variables, parameters, types and instances (objects). It also describes the Name Uploader.

- Appendix C, AC 800M Aspects lists specific AC 800M aspects, with a short description of each aspect.

If SIL certified applications are to be run in a High Integrity controller, refer to the *System 800xA, Control, AC 800M, Getting Started (3BSE041880*)* manual.

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

Electrical Warning icon indicates the presence of a hazard 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

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

Related Documentation

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

Section 1 Design Issues

Introduction

Consider the following while planning an automation solution:

- The application layer protocol to be used – MMS or Inter Application Communication (IAC). MMS uses control modules and function blocks whereas IAC uses cyclic communication using communication variables.

- A number of conceptual choices have to be made, for example regarding which programming strategy to use, see Conceptual Issues on page 14.

- The right hardware and network solutions have to be selected, so that the performance is satisfactory and there is sufficient margin for further development. See Performance Issues on page 38.

- There are certain limitations to the number of controllers, applications, diagrams, programs, and tasks that co-exist in an 800xA system and in a single controller. See Limitations on page 46.

In addition to the information in this manual, refer to the following manuals:

- The *800xA System Guide, Performance and Capacity (3BSE041434*)* contains performance data and information on how to make important calculations. Performance Issues on page 40 contains an overview of the information that can be found in this manual.

- The *System 800xA, System Configuration (3BDS011222*)* contains information on how to configure the engineering environment and how to use high-level tools such as Bulk Data Manager to create a basis for the automation system.

Conceptual Issues

Consider the strategical (conceptual) issues provided below, before creating an automation solution:

- The type of programming to be used: traditional or object oriented programming. See Traditional Programming and Object-Oriented Programming on page 14.

- The type of execution to be used: list-driven or data flow driven execution. See List-Driven Execution and Data Flow Driven Execution on page 16.

- The type of libraries to be used to create re-usable type solutions, and the system parts suited for creating type solutions. See Libraries on page 18.

- The coding process to be used: control modules, programs or diagrams. See Code Organization on page 19.

- The programming language to be used. See Programming Languages on page 36.

- The structured data type to be used, and the instance to use structured data types. See Structured Data Types on page 37.

Traditional Programming and Object-Oriented Programming

This section discusses about the traditional programming and object-oriented programming concepts. The benefits of the object-based method as compared to the traditional method is also discussed, in connection with the data-flow-driven execution, which results in control modules or diagrams– two very powerful programming concepts.

Traditional Programming

In traditional programming, the design and execution is determined by the sequential appearance of the code, starting from the top to the bottom. As the program design proceeds, the code has to be re-written if any major changes are made. Also, the programmer must understand the type of an action and when the action should occur during an execution cycle. Hence, the code design is time-consuming, excessive, and difficult to maintain and update. It also has a poor structural design.

Object-Oriented Programming

Object-oriented programming is quite different from traditional programing in design and execution. While traditional programming focuses on the overall task (for example "Manufacture cement"), the object-oriented method reduces a complex or large problem, to isolated, self-sustained procedures. Such procedures, while considering the "Manufacture cement" example, could be "Regulate motor speed", "Open valve for water", etc. Each of these *objects* is an isolated procedure.

For example, "Regulate motor speed" does not require any specific information about the motor itself, and the same holds for the valve. Each object has an input and an output, and between these, the object analyses the situation and makes the appropriate decisions, resulting in an action, or output. While doing this analysis, the object does not require any input from the outside world.

When using this method of program design, the dependencies on other objects must be placed outside the current object.

For example, consider the command to start a motor. The "Regulator" object cannot start the motor independently; however, it can stop the motor if the speed becomes too high. Even the normal stopping order for the motor is placed outside the "Regulator" object, since this object is only concerned with regulating the motor speed and stopping the motor in cases of faults or emergency.

The normal behavior is placed outside the object, while actions, such as security measures, are placed within the object.

Using object-oriented design, the modelling environment is made simple. A number of reusable tools are created while the objects are being developed. These tools help in creating larger and more complex projects easily.

When the number of re-usable tools has reached a stage such that they become an acceptable and defined standard in the daily project development routine, they can be made into a *library,* for even more simplified project development.

In object-oriented programming, the initial effort and time is high. Each object must be analyzed thoroughly, and then designed, such that it can remain as an isolated, self-sustained, and re-usable object. Once the toolbox (or library) is ready with re-usable objects, large-scale projects can be developed with ease.

The object-oriented design applies to function blocks, control modules, and diagrams. However, technical considerations may lead to one technique being more suitable than the other in some situations.

The difference between control modules and function blocks is the automatic code sorting of code blocks in control modules, which gives better execution performance since the code blocks are then executed in the most optimized way (see Data Flow Driven Execution on page 17).

If diagrams are used, it is possible to include the entire object-oriented design in a single diagram as it allows mixing control modules and function blocks by 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.

Summary

The object-oriented programming/design:

- Simplifies daily project development.

- Provides a toolbox of reusable building blocks.

- Makes it easier to design large, complex projects using the building blocks.

- Makes it possible to create full-scale libraries that can be shared among project participants.

- Makes it easier to survey, maintain, and update your code.

List-Driven Execution and Data Flow Driven Execution

The actual control and interaction between the building blocks in a program can be through list-driven execution or data flow driven execution.

List-Driven Execution

The list-driven execution is a cyclic process. The code must be programmed and executed in the correct order. Otherwise, it leads to time delays.

Data Flow Driven Execution

In Data flow-driven execution, the coding procedure is easy. All the objects communicate with each other, and can determine when individual objects can send and receive information.
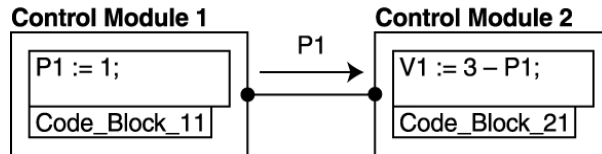
Figure 1. The code block in control module 1 must be executed before the code block in control module 2 – not the other way around

A data flow driven design can avoid errors caused in the execution order. In data flow-driven execution, *code sorting* exists.

Sometimes, Control Builder may not analyze the code design, resulting in *code sorting loop errors*. For example, two *IF* statements cannot possibly be executed at the same time. However, the code sorting routine may consider it as a loop, though it is not a loop. For such instances, there is a *nosort* attribute that can be issued to suppress such errors.

Summary

The data flow driven execution:

* Eliminates dangerous situations caused by poor, unclear design.

* Optimizes the code so that execution is carried out efficiently.

* Simplifies the code design allowing the user to concentrate on small parts rather than the entire project.

* Maintains the execution cycle and performs the actions contained in the cycle.

Libraries

There are two types of libraries, *Standard* and *Self-Defined*. The set of Standard libraries is installed with the system and is always present. They may not all be present in the current project, but will be loaded on demand (that is, if you connect them to your library or application).

To re-use the function blocks, control modules and diagrams, save them as types in either of following ways:

• Save them in the application. You can then re-use types within that particular application only.

• Save them in a self-defined library and then re-use them in any application or project.

Types saved locally in the application will not be available for the other applications or project.

As mentioned earlier, you can save your building blocks in your own, self-defined library. As is the case with any of the other libraries, this library will also be available to other projects, and you can also share the library with other project members.

The structure of a library does not need to be flat, you can connect other libraries (*Standard* or *Self-Defined*) to a new library, and by doing so, you create a library hierarchy. Assume you want to set up a project library called Cement Factory Library, which connects to a large number of Standard and Self-Defined libraries (perhaps in a deep hierarchical structure). By doing so, other participants do not have to load all the libraries required, just the "Cement Factory Library" in order to access all the necessary data types, function block types, control module types, and diagram types.
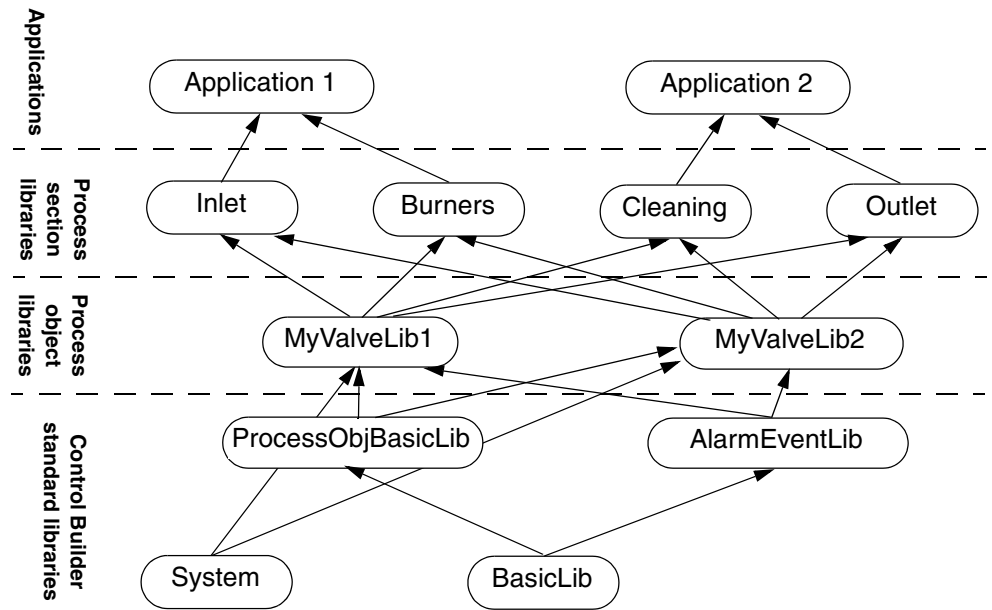
Figure 2. Create (deep) hierarchical structures using both Standard and Self-Defined libraries

Code Organization

This part of the manual will help you understand how function blocks, control modules, and diagrams interact and important differences that might affect which programming strategy you choose to use.

When organizing code, there are three basic methods of doing this:

- Using control modules, see Using Control Modules on page 20.

- Using function blocks inside programs, see Using Programs on page 21.

- Using diagrams, see Using Diagrams on page 26.

It is also possible to mix control modules and function blocks in diagrams (see Using Diagrams on page 26; and it is also possible to mix control modules and function blocks inside each other (see Mixing Control Modules in Function Blocks and Vice Versa on page 24).

For more detailed information on control module type, function block type, diagram type, and objects, refer to the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Using Control Modules

You can organize your code using control modules only. The benefit is clear: Control Builder will arrange (sort) the code[1] so that optimal data flow is achieved during execution. Note that this sorting only can be carried out for control modules, see List-Driven Execution and Data Flow Driven Execution on page 16 for more details.
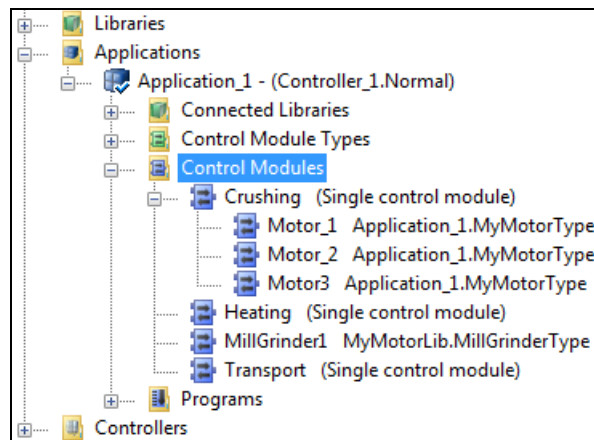
Figure 3. Design using control modules only

Figure 3 shows an example of extensive use of control modules. All code is sorted for optimal data flow during execution.

---

1. Since only control modules are used, all code will be sorted.

Using Programs

As well as using control modules only, it is also possible to use function blocks only. These are then placed in a program.

When organizing your code, you will sooner or later face the question of how many programs, code blocks, function blocks, etc., are needed and how the code should be distributed on these objects. The below list provides some recommendations and hints regarding this.

- **A Few Large Programs or Several Smaller Programs?**

    The smallest unit that is compiled and downloaded to the controller is a Program Organization Unit (POU). As discussed in other sections in this manual, a program is also a POU. It is tempting to put all the code in *one* program, but this is not recommended, since this will lead to *all code* being compiled every time, and *all code* being downloaded, even if the changes made are minimal. The results of such a design would be that downloading would take a long time, and that much more memory would be allocated in the controller during download. The latter would result in a dramatic reduction in the possible size of an application.

    An obvious solution to this problem is to divide the program into several smaller ones, and then make use of the code block features within the programs (or POUs). Although this may seem to be an ideal solution, there are a number of drawbacks with this design, compared to fewer but larger, programs. Finding the right balance can be difficult.

- **Let Programs Reflect Reality on a Large Scale!**

    Programs should be organized according to functional areas, such as "Intake", "Production", "Outlet", etc. Putting the code for these three functional areas in one program would be poor design. Furthermore, programs (and code) should also be organized with regard to execution demands (interval time, priority, etc.), that is, task properties.

Several programs can share the same task, and they should if the programs all have the same requirements regarding interval time, priority etc. There is seldom need for more than 3–5 tasks per application. Use Task Analysis to verify that the tasks are properly configured with respect to priority and offset.

- **Let Code Blocks Reflect Reality on a Small Scale!**

  Apart from dividing the code into several programs, you can also use several code blocks within each program (or POU). It is then necessary to decide how many code blocks to use, and how much code should be allowed in each code block.

  (Up to 100 code blocks can be used in a POU, but it is seldom appropriate to have so many code blocks.)

  As programs reflect the process on a large scale, code blocks can be used to reflect the process on a smaller scale. For example, the control of a motor or valve (within a functional area) can be put in a single code block with a suitable name. It is also possible to define the execution order within a POU using code blocks, since code blocks are executed from left to right.

  Obtaining a good balance between the number of programs and the number of code blocks within the programs may be difficult. The problem can be illustrated as a two-dimensional chart.

Figure 4. Programs vs. Code blocks

Having several programs and few code blocks (A), or few programs and several code blocks (B) is poor design. A compromise like (C) is the best solution. We can also add a third dimension – function blocks that are declared in the programs.
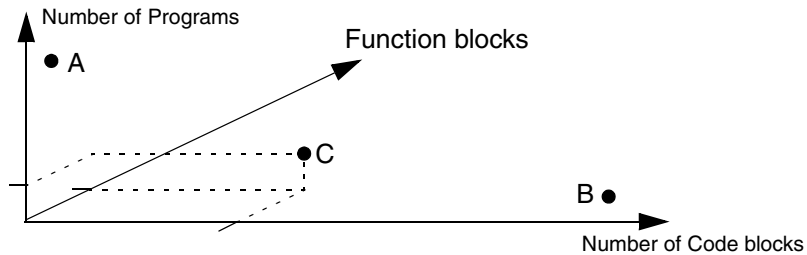
Figure 5. Including function blocks adds a third dimension to the problem

This third dimension is unlimited since you can have as many function blocks as you like, and more function blocks, in each function block, etc. You can also add control modules within a function block type, in order to use control modules in a program. (This provides a way of using regulatory functions that are only available as control modules, although you are using programs.)

It is important to note that the two-dimensional design (see Figure 4) is more important than the three-dimensional design (see Figure 5), when it comes to download time and memory consumption.

When using the programming languages FBD and LD it is easy to assume that the pages in a code block correspond to the "function block axis" in Figure 5. This is not the case. Code blocks in FBD and LD can contain many pages but they still belong to the same code block in the POU.

Mixing Control Modules in Function Blocks and Vice Versa

It is also possible to mix control modules and function blocks, that is, to place function blocks inside control modules, and to place control modules inside function blocks.

•   **Using Function Blocks in Control Modules**

It is not always possible (or suitable) to use control modules throughout the design. Simple tasks are best implemented using function blocks. Control modules should be used at higher levels of complexity.

When you need to implement simple tasks, function blocks can be included in a control module (or several control modules).
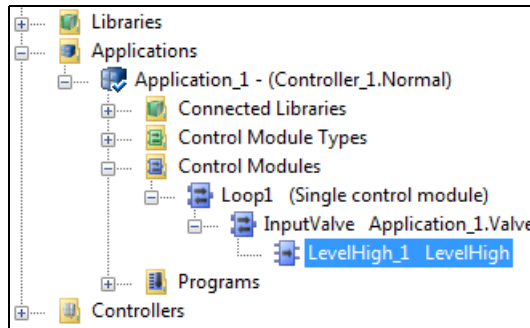
Figure 6. Function blocks can be used within control modules

In Figure 6, a function block (LevelHigh_1) of type LevelHigh (taken from BasicLib) has been included within the InputValve control module (of Valve type).

When control modules contain function blocks, the code blocks inside the function blocks are not sorted.

Control modules are only executed once per scan, whereas function blocks may be executed several times per scan (or not at all).

• **Using Control Modules in Function Blocks**

It is not always possible to use function blocks throughout the entire design, but you can include control modules in a function block (or several function blocks). A good reason to do this would be to include functionality that is only available as control module types in your design, while you continue working with function blocks and programs.
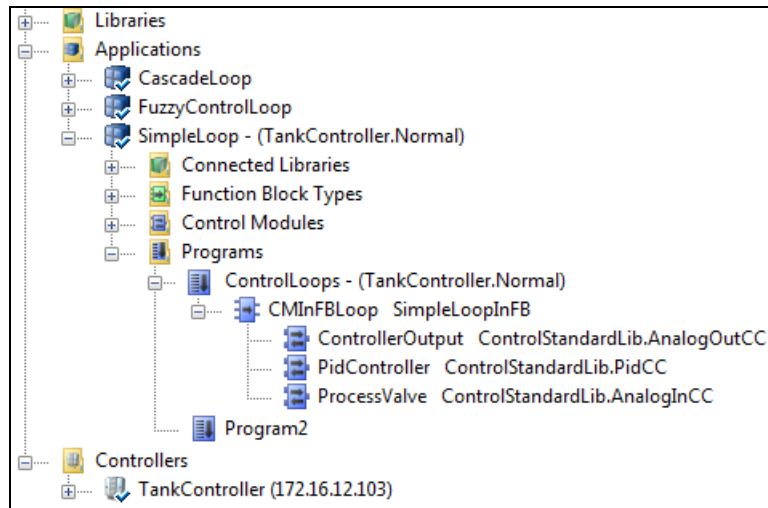
Figure 7. Control modules can be used within function blocks

In Figure 7, several control modules (ProcessValue, PidController, ControllerOutput) have been inserted into the function block CMInFBLoop. As mentioned previously, the code for function blocks is not sorted for optimal data flow during execution. However, the code for the control modules in the function block is sorted (locally) according to data flow.

Let us assume that we have another function block in parallel with this one. The control modules in that function block are also sorted locally. But, these two isolated groups of control modules are not sorted relative to each other. If there is an exchange of variables between these two control modules, there will be time delays.

In Figure 8 Control module 1 and 2 will be sorted together as will Control module 3 and 4. However, these two individual groups (grouped by Function Block 1 and 2) will not the sorted together.
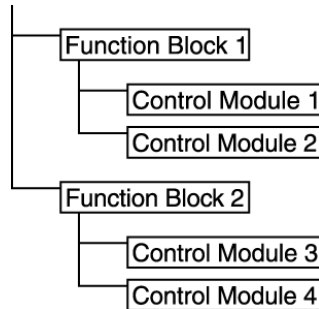
Figure 8. The boundaries of the function blocks limit the sorting of control modules

Using Diagrams

Diagrams allow you to configure the control logic in the project in a comprehensive graphical language called FD (Function Diagram). The diagram allows 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 FD code block, the diagram also supports SFC and ST code blocks, which are invoked from FD code block or sorted separately.

The main elements in FD are:

- Pages
- Blocks (objects)
- Connection Ports

The diagram is defined under an application, and it is available for download to the controller.

Additionally, it is possible to:

- Connect the diagram and its objects to controller tasks.
- Control the execution order of the diagrams, based on the controller tasks.

Figure 9 shows an example of a diagram.
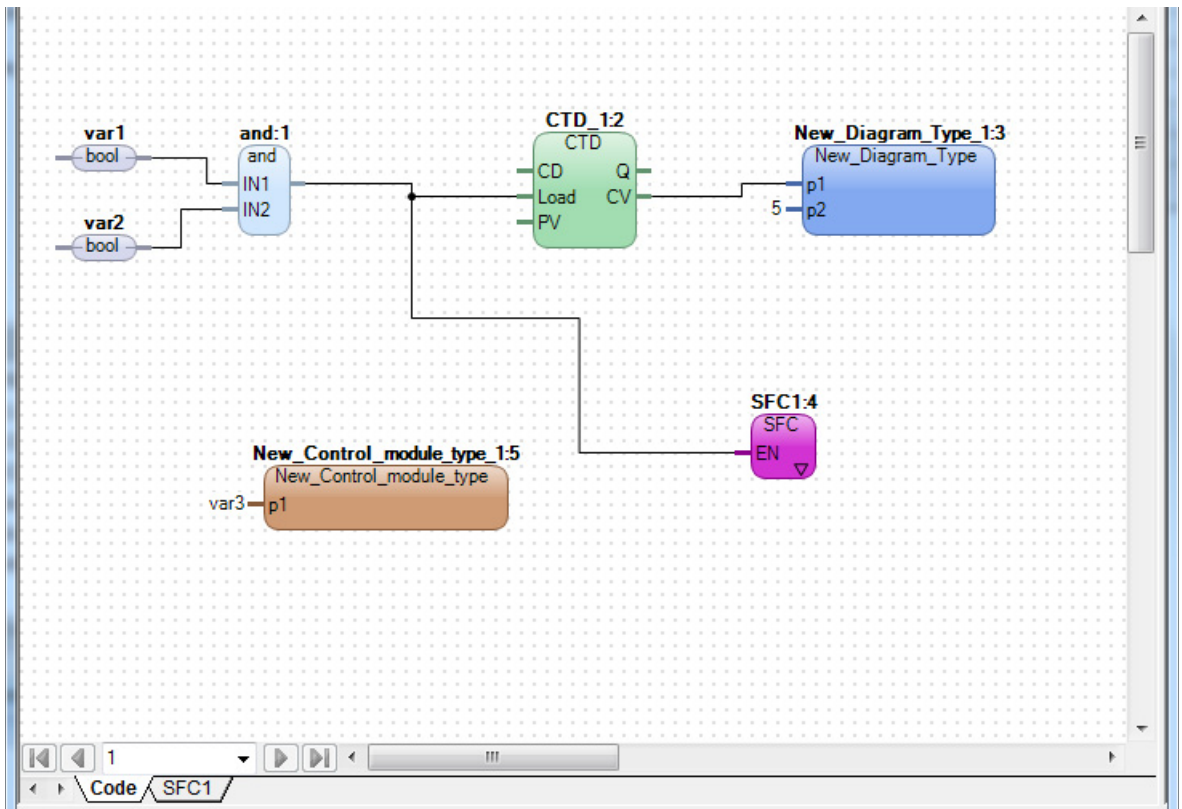
Figure 9. Example of a diagram

The graphical user interface makes complex control strategies easy to engineer and even easier to maintain.

The following factors affect the usage of diagrams:

- **Few Large Diagrams or Many Smaller Diagrams**

  A diagram is also a POU. It is tempting to put all the code in *one* diagram, but this is not recommended, since this will lead to *all code* being compiled every time, and *all code* being downloaded, even if the changes made are minimal. The results of such a design would be that downloading would take a long time, and that much more memory would be allocated in the controller during download. The latter would result in a dramatic reduction in the possible size of an application.

  An obvious solution to this problem is to divide the diagram into several smaller ones, and then make use of the pages and code blocks within the diagrams.

- **Let Diagrams Reflect Reality on a Large Scale**

  Diagrams should be organized according to functional areas, such as "Intake", "Production", "Outlet", etc. Putting the code for these three functional areas in the single FD code block in one diagram would be poor design. Furthermore, diagrams should also be organized with regard to execution demands (interval time, priority, etc.), that is, task properties.

Several diagrams can share the same task, and they should, if the diagrams all have the same requirements regarding interval time, priority etc. There is seldom need for more than 3–5 tasks per application. Use Task Analysis tool to verify that the tasks are properly configured with respect to priority and offset.

- **Let Pages and Data Flow Order Reflect Reality on a Small Scale**

  Apart from dividing the code into several diagrams, you can also use several pages within each diagram, and arrange objects in each page according to the desired data flow. Each object in a page has a unique Data Flow Order number, which determines the forward data flow.

  It is then necessary to decide how many pages to use, and how much objects should be inserted in each page.

  (Up to 200 pages can be used in a diagram, and up to 50 objects can be placed in one page. But, it is seldom appropriate to have so many pages and objects.)

As diagrams reflect the process on a large scale, pages can be used to reflect the process on a smaller scale. For example, the control of a motor or valve (within a functional area) can be put in a single page with a suitable name.

Obtaining a good balance between the number of diagrams and the number of pages within the diagrams may be difficult. The problem can be illustrated as a two-dimensional chart.

Figure 10. Diagrams vs. Pages

Having many diagrams and few pages (A), or few diagrams and many pages (B) is a poor design. A compromise like (C) is the best solution.

- **Reusable Diagram Types Used in the Diagram**

We can also add a third dimension – reusable diagram types – that are created outside the diagram and instantiated in the diagram. The diagram type also allows mixing of functions, function blocks, control modules. A diagram type can also contain its own parameters that can be connected to variables while creating the instances.

A diagram can contain multiple diagram type instances. A diagram type itself can contain other diagram type instances, which results in nested functionality of diagrams.
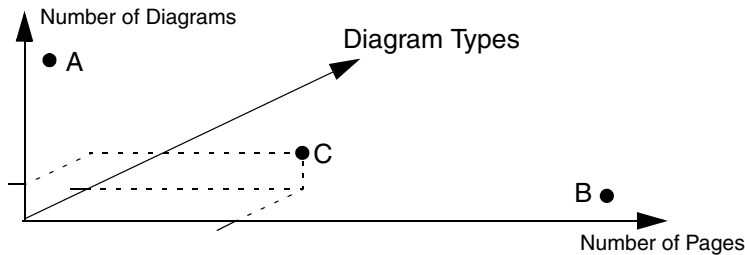
Figure 11. Including reusable diagram types adds a third dimension to the problem

> This third dimension is unlimited since you can have as many diagram types as you like, and many objects in each diagram type.

- **Lower SIL Signal Used in a Higher SIL Diagram**

> Another advantage of using diagram for the design with SIL applications is that the FD code block in a diagram of higher SIL can accept a signal of lower SIL, if required, using communication variables. The FD code block also displays such a lower SIL signal with a distinct color.

Differences Among Control Modules, Function Blocks and Diagrams

Control module solutions may be more efficient than function block solutions, particularly if complex applications with deep hierarchical structures are to be used. This is in particularly true for parameters of structured type, which are distributed as In and/or Out parameters. Function block parameters are copied at each call, while control module parameters are set up *once* during compilation.

Suppose we have the same code in a function block and in a control module, and that the blocks do not have any parameters. In this case, the function block will execute faster than the control module, since control modules have an extra overhead of 4-5 microseconds for each code block (in PM860), function blocks do not have this overhead. The memory requirement will be the same for both.

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

Differences arise when adding parameters to the block. Each parameter in a function block adds to memory requirements, but parameters in control modules do

not. Furthermore, connected parameters (In or Out) in function blocks are copied from one variable to another in order to establish data consistency within the block. Parameters in control modules, on the other hand, are simply references that are resolved during compilation.

The difference in performance between function blocks and control modules is caused by the number of parameters used. Connecting parameters in function blocks and/or having parameters of structured type in function blocks will result in poorer performance for the function blocks. Neither of these affect control module performance.

Function block parameters are copied at each call, which means that the CPU will spend considerable time passing parameter values to and from function blocks, if there are deep structures in the program. The parameter connections of a control module, on the other hand, are defined prior to compilation. This gives superior performance using control modules when parameters are transferred through deep hierarchies.
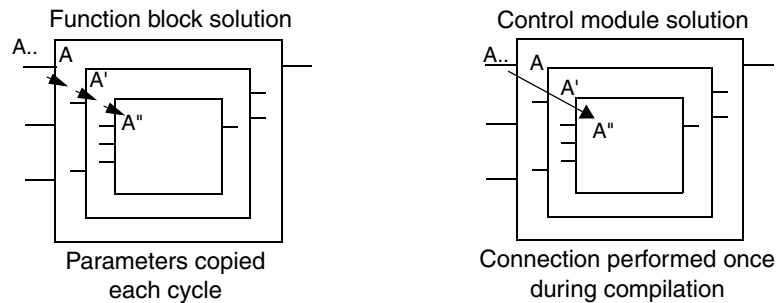
Figure 12. Parameter values passing through deep hierarchies

In other words, you should consider using control modules for complex applications with deep hierarchical structures, and in applications with many parameters. In such cases, the use of control modules may decrease the risk of deterioration in performance.

If diagrams are used in the design, the FD code blocks in the diagrams allow mixing function block instances, functions, control module instances, and diagram instances in the same diagram,
The following are the special rules that apply to diagrams:

- A Diagram POU (under an application) can contain only one FD code block, but the FD code block itself can contain multiple diagram type instances, which can also be nested. If structuring of a diagram is needed, the diagram is separated in several pages in the FD code block.

- Non-FD code blocks (ST and SFC) in the diagram can be invoked within the FD code block. Code blocks with no invocation in the FD code block are sorted with code blocks in the invoked control modules.

- Each diagram constitutes a code block sorting group. This means that all control module instances under a diagram are sorted together, but they are not sorted together with any control module instances outside the scope of the diagram.

The most important differences among control modules, function blocks, and diagrams are summarized in Table 1.

Table 1. Differences among function blocks, control modules, and diagrams

Property	Control modules	Function Blocks	Diagrams or Diagram Types
Execution order	Automatic; compiler-determined via code sorting/data flow optimization.	Manual; code statements, based on program flow as implemented by the programmer.	Code blocks from control modules are sorted into the diagram's execution order according to control module sorting rules (writing into a variable is sorted before a read). For other objects, the execution order is based on the Data Flow Order number that is automatically assigned.
Execution per scan	Code blocks are always executed once per scan according to data flow analysis.	A function block can be called, and executed, zero, one, or several times per scan.	The code blocks in the diagram types inside the diagram are executed once per scan. For control modules and function blocks inside the diagram, the respective rules apply.

Table 1. Differences among function blocks, control modules, and diagrams (Continued)

Property	Control modules	Function Blocks	Diagrams or Diagram Types
Static parameter connections	Yes. This is an important feature. A static parameter connection does not change during execution; it can only be changed via code changes and recompilation. Static connections are set at compilation, yielding efficient code generation, powerful compiler-based program analysis, and better performance.	No. Parameters are copied each time the function block is executed according to the IEC 61131-3 standard. For deep and complex hierarchies, the parameter copying of function blocks demands significant CPU time. By using the *By-ref* attribute for function blocks, the issue related to CPU time can be avoided.	Yes, the diagram types allow static parameter connections (same benefit as control modules).
Graphics	Yes, in three different ways: free-layout programming, supervision, and interaction.	Indirectly via inclusion of sub control modules.	Yes. In the same editor, free-layout programming, supervision, and interaction are possible.

Table 1. Differences among function blocks, control modules, and diagrams (Continued)

Property	Control modules	Function Blocks	Diagrams or Diagram Types
Parameters	Parameters of the type *in_out* can be connected to graphics. Connections cannot be inverted. ("NOT current parameter name" is invalid.)	Parameters of the type *in, out*, or *in_out* can be connected to sub control modules. However, in_out parameters must be connected statically if they are connected to a control module parameter.	In diagram type, all parameters (*in, out,* and *in_out*) can be connected to graphics. Additionally, an *out* parameter can be connected to an *in_out* parameter, and an *in_out* parameter can be connected to *in* parameter.
Task connections	Can be connected freely to any task.	Cannot be connected to a task if it contains in_out parameters (see further Task Connection and Parameters of Type in_out).	Can be connected freely to any task. The diagram type instance in a diagram inherits the task of the diagram. The control module instances and function block instances in a diagram can be connected to a different task, if required. Otherwise, they inherit the same task as the diagram.

Summary

When choosing which way to organize your code, you have a number of choices to make:

- If you prefer a task-oriented and graphics-oriented approach, mixing different objects and providing graphical connections between them, you have to choose diagrams and diagram types.

- If you prefer a task-oriented programming approach using programs, you have to choose between control modules or function blocks organized in programs (but in the first case, function blocks can be used inside control modules, and in the latter case, control modules can also be used in the program and inside function blocks).

Which approach to choose depends on things such as how your organization collects data and converts it to an automation solution, as well as on the development and maintenance strategy chosen for the automation solution in question.

Programming Languages

When it comes to selecting which one of the available programming languages to use, there is no clear rule. The languages available are Structured Text, Instruction List, Function Block Diagram, and Ladder Diagram. These are all "basic" programming languages and can be used in any code block. You can even use more than one language in a single program, function block, or control module.

There are two additional programming languages, Sequential Flow Chart (SFC) and Function Diagram (FD).

SFC is not, in a sense, a "real" programming language, but rather a structural programming tool. Using SFC, you can easily design more complex sequences than would be possible (with the same effort) in any of the other languages. As the name *Sequential* Flow Chart implies, you have to plan sequentially when using this language.
For each step in the sequence you can connect three code blocks, one that executes at the start of the step, one that executes during the step and one that executes at the end of the step. Additionally, there is also a Boolean variable for each step that can be used as an indicator in another code block in any of the four languages listed above.

The Function Diagram (FD) language allows mixing of functions, function blocks, control modules, and diagrams in the same editor.

For a more complete overview of the available programming languages, see Section 2, Programming Languages.

Structured Data Types

An important part of creating re-usable solutions is to use structured data types. These serve as communicators (tunnels) through the application, and can in many cases be used instead of writing code directly in function blocks and control modules. If we implement object-specific code in every place, instead of using variables of structured data types, we would soon narrow down the dynamics and flexibility of the system.

Structured data types can be seen as a thick cable, with each component as a cord inside the cable. The strength of structured data types shows when they are used in connections between types. You can connect control module types (through several layers), without declaring any variables on each module level. One component inside a valve can be directly accessed from a parameter on the outside. If you, later on, need a new component, it will pass through all control module types and control modules with no additional interconnection.

Figure 13 illustrates the use of structured data types inside a ValveLib library, in Project Explorer.
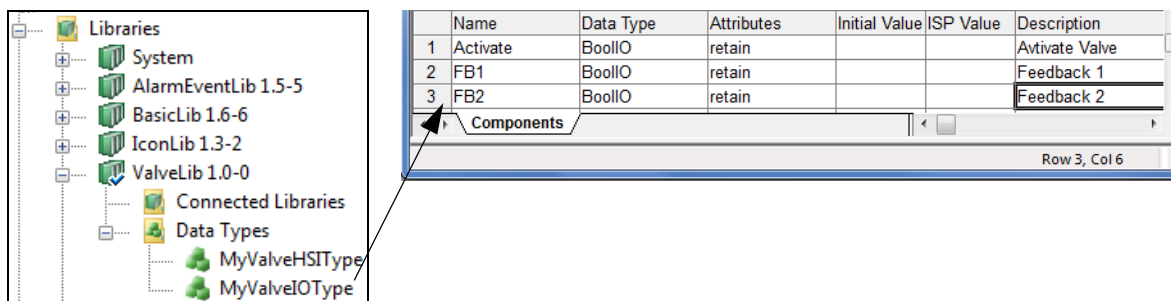
Figure 13. (Left) MyValveIOType created in ValveLib. (Right) MyValveIOType with its components

For examples of the use of structured data types for I/O communication, see Structured Data Type Examples on page 116.

Performance Issues

Before deciding on which hardware and communication protocol(s) to use, it is necessary to make a number of calculations and estimate a number of things. Once you have an idea of what you need, you can go on to choose your hardware. In order to help you prepare for this choice, this subsection has been split into the following parts:

- Memory Consumption on page 38, discusses things related to memory consumption that should be considered during the planning phase.

- Calculations and Performance Data on page 39, gives an overview of the Industrial IT, 800xA System Guide, Performance and Capacity (3BSE041434*), which is of great help when discussing performance issues.

- Choosing Controller Hardware on page 40, lists a number of things that should be considered when choosing controller hardware.

- Distribution on Applications and Controllers on page 42, discusses advantages and disadvantages of splitting your code on several applications and/or controllers.

When planning your control network communication, you should also read the *AC 800M, Communication Protocols (3BSE035982*)*. This manual gives a conceptual overview of all supported protocols and contains useful performance data for a number of key protocols, such as MMS.

Memory Consumption

Memory is reserved for each defined function block type, control module type, and diagram type. When another function block or control module or diagram is created from a type, the amount of memory reserved is very small, in relation to the memory reserved for the type. You should therefore create as many instances of control modules, function blocks, and diagrams, as possible from a few types, instead of implementing your control solution with many different memory-consuming types.

Calculations and Performance Data

Two very important concepts when discussing the performance of you automation solutions is CPU load and memory consumption. In order for you to understand how to calculate those in different situations, you should study the *System Guide, Performance and Capacity (3BSE041434*)*, which contains information on things such as:

- How much memory is needed in a controller before downloading an application? (Known as "available memory".)

- How much free memory is needed to update a running application?

- How does performance differ between different AC 800M processor units?

- How does performance differ between redundant and single controller configurations?

- How does the execution interval affect CPU load?

- How do I calculate the CPU load percentage?

- How much memory does the first instance of a type consume and how much do additional instances consume? Numbers are given for a number of common library types.

- What is the execution time for instances of a number of common library types?

- How do I calculate the ModuleBus scan time?

- What is the I/O response time for a number of common protocols?

- What effects MMS communication speed?

Performance data for all supported protocols is also found in the *AC 800M, Communication Protocols (3BSE035982*)*.

- What is the accuracy of the different types of clock synchronization?

In addition to the above manual, you should also study Section 3, Programming in Practice.

Choosing Controller Hardware

Another important consideration is the choice of controller hardware. At least two very important issues should be considered when choosing controller hardware:

- CPU capacity
- CPU priority

CPU Capacity

If your application puts high demands on the controller regarding application execution or communication performance, the following points should be considered.

- How many I/Os are to be connected? As the number of I/Os increases, so do the requirements on the control system.

- Do you have I/Os that require short interval times? You might want to add external cards that contain separate CPUs for I/O processing, for example, PROFIBUS-DP.

- Which protocol should be used, and hence, which control hardware?

CPU Priority

1.   AC 800M controller:

In AC 800M, servicing the S800 I/O via ModuleBus has the highest priority (interrupt), and may cause a significant load on the CPU. Note that this interrupt load is not accounted for separately, it will be evenly distributed over other tasks. This means that the cyclic load presented for IEC 61131-3 task execution includes the extra load caused by ModuleBus interrupts during task execution.

The default setting for the ModuleBus scan cycle time is 100 ms.

Calculate the minimum scan time possible for the I/O configuration that is used, using the formula in section Modulebus Scanning of Digital/Analog modules in the *System Guide, Performance and Capacity (3BSE041434*).*

Set the modulebus scan time to a value as high as possible, but higher than the calculated minimum value and lower than the interval time of the fastest task using I/O signals. This decreases CPU load, in favor of user application(s), and communication handling. This is needed in order to avoid sampling problems due to the fact that the I/O systems and the IEC 61131-3 tasks execute asynchronously.

A Supervision function implemented in the controller generates a system alarm if the time taken to scan all modules exceeds the configured value +10 ms. If the configured value is set to 0, then the Supervision is disabled.

The ModuleBus scan cycle time should be defined at an early stage of the project, since many other parameters depend on it. Based on the ModuleBus I/O system configuration, the complier indicates a recommendation or a warning on scan time while downloading to the controller.

Decreasing the ModuleBus scan cycle time in a running plant reduces CPU capacity available for application(s) and communication handling.

2.   IEC 61131-3 Code:

Execution of IEC 61131-3 applications has the second highest priority. Depending on the amount of code and requested task interval times, applications may demand up to 70% of CPU capacity (never more)[1]; the execution of IEC 61131-3 code is called *cyclic load*.

Should an application require more than 70% of CPU capacity, the task scheduler automatically increases the task interval times to re-establish a 70% load. This tuning is performed at 10-second intervals. You can read more about task handling in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*).*

Load balancing can be disabled. For SIL certified applications, load balancing is always replaced by overrun and latency supervision.

Since IEC 61131-3 has higher priority than communication, it is recommended not to configure for consecutive execution of more than 200ms.

3.   Cyclic Diagnostics in an AC 800M HI controller:

It is important to consider the relevant FDRT setting in perspective of the required diagnostic interval time for the system background diagnostics. Do not set the FDRT setting lower than necessary, since this has a significant impact on the load from diagnostic measures in the controller.

The system background diagnostics in the Controller must be able to execute to completion with a cycle time of half the FDRT. The cyclic diagnostics executed within FDRT has the priority next to 1131 application execution, which is, the sum of all 1131 application task execution time should not exceed 80% of ½ the configured FDRT setting. For example, with an FDRT setting of 3 sec, the 1131 execution during any 1.5 sec period must never exceed 1,2 sec (80% of ½ FDRT).

4.   Communication Handling (lowest priority):

It is important to consider CPU load if communication handling is vital to the application. Running at the maximum cyclic load will result in poor capacity and response times for peer-to-peer and OPC Server communication.

Communication handling has the lowest priority in a controller. It is therefore important to consider controller CPU load if the communication handling is vital to the application. Running close to 100% total load will result in poor capacity and response times for peer-to-peer and (OPC Server for AC 800M) communication. It is recommended that peak total load will be kept below 100%.

---

1.   This is **not** true if load balancing is set to false or if you run with an AC 800M HI. The controller will run until it is forced to stop.

Inter Application Communication (IAC) has higher priority than normal MMS communication.

Distribution on Applications and Controllers

The following sub-section discusses the pros and cons of dividing a project into applications for one or several controllers.

> Communication between applications in controllers takes place using *Access variables* or *Communication Variables*, independent of where the applications are located. See the *System 800xA, Control, AC 800M Configuration (3BSE035980*)*.

One Application to One Controller

In general, it is recommended to assign *one* application to *one* controller.
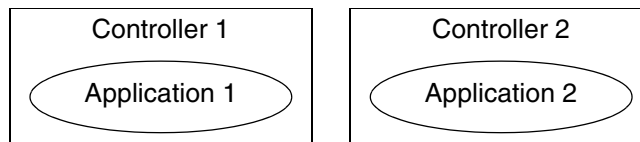
Figure 14. One application to one controller

There are some advantages to doing this, such as:

• There is only one application to keep track of.

• I/O variables can be freely connected (if several applications are connected to the same controller, then variables must be sent between the applications via communication), and it will be easy to search amongst them. You will not have to transfer I/O values between applications.

• *Communication variables*, which use controller-IP based resolution, are used for cyclic communication between diagrams, programs, and top level single control modules (communication in the entire control network).
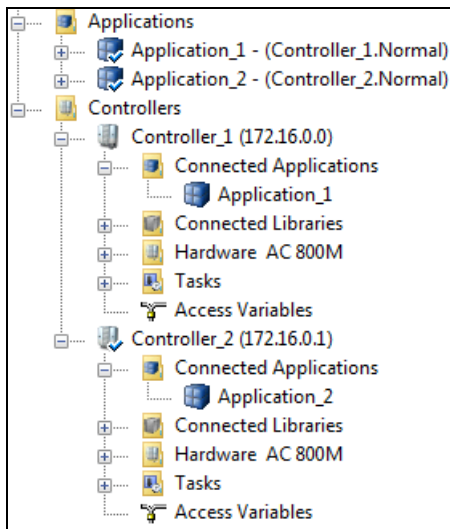
Figure 15. Project Explorer configuration with two controllers, each with one application

However, there are disadvantages. An application requires twice the memory size in the controller when downloading changes, which means that more memory must be allocated in the controller (to be able to handle future modifications).

Dividing one Application on Several Controllers

If you must divide one application on several controllers, you have to consider this when you plan your project. Although automatically generated variables handle communication between the various parts of the application, unacceptable delays in the variable communication may occur, leading to problems.

Communication variables are not supported in distributed applications.
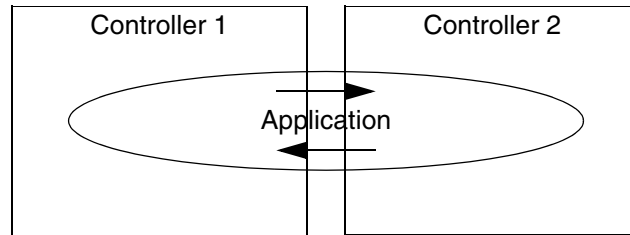
Figure 16. One application, divided on several controllers. There is automatic Application Internal Communication between the controllers

The automatic communication that results from distributing an application between several controllers is called **Application Internal Communication**. The transfer of exported variables takes place at a lower priority than the execution of tasks. The controller (Client) reading a certain set of variables asks the owner (Server) for the exported variables cyclically. There is no synchronization between communication and execution.

Note that the communication load resulting from Application Internal Communication can be high, if the structure of the application is such that a large amount of data needs to be transferred. Communication is divided into telegrams that can contain 1 kB. All telegrams required are fetched within a certain interval, but spread over that interval to increase throughput. Setting a low interval time can cause deterioration in communication performance. Task execution is, however, unaffected.

Distributed applications are not possible in High Integrity controllers. When distributing an application on several controllers you must consider the following.

- Tasks in different controllers execute asynchronously.

- Automatic cyclic transfer of values between the controller (Application Internal Communication), is asynchronous and the shortest possible interval time depends on the controller and the network load. Recommended minimum interval is 500 ms.

- The function *WriteVar* can be used to obtain non-cyclic Application Internal Communication.

- The values of strings are normally not transferred (default setting) as this lowers communication performance. If you want string values to be transferred, set the system variable *EnableStringTransfer* to True.

- Data consistency is not guaranteed with Application Internal Communication. Task execution can interrupt this communication service so it can not be guaranteed that all exported variables are sampled from the same execution, or incorporated into the client at the same time.

There is more information in the Control Builder online help. Search for "Distributed Execution".

- There is no automatic supervision of communication. This must be performed manually, by using appropriate function blocks.

- I/O signals must be read/written by a task residing in the controller to which the I/O signal is physically connected.

- You cannot run Compact Flash with distributed applications.

Several Applications in One Controller

Loading a controller with several applications can provide an excellent way of reducing controller stop time during a program change, and of gaining more space for code in the same controller.
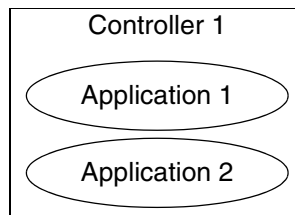

Figure 17. Several applications in one controller

A small application in a controller has the following advantages.

- The stop time during program modification will be reduced.

- The application will be easier to survey.

- There will be more memory available for future modifications.

However, there are a few disadvantages:

•   It will be more complicated to exchange data between applications, for example, if several applications must read from the same analog input.

•   The number of tasks increases, which means that the overhead (the controller load) will increase. A task can only execute code from one application, which makes it difficult to specify the order of programs in different applications.

Limitations

When designing an automation solution, there are certain limitations to the system and to hardware that must be considered. Limitations can be found within the following fields:

•   OPC server limitations, see OPC Server Limitations on page 47.

•   The size of applications, see Application Size Limit on page 47.

•   Number of controllers, applications, diagrams, programs and tasks, see Maximum Number of Controllers, Applications, Diagrams, Programs and Tasks on page 48.

•   Number of Program Organization Units (POUs) and variables, see Maximum Number of POUs and Variables on page 49.

•   INSUM communication and number of MCUs, see INSUM Limitations on page 52.

OPC Server Limitations

There are a number of limitations to OPC Server for AC 800M setup and configuration, of which the most important are:

•   A single OPC Server for AC 800M may subscribe to data from several controllers, or the equivalent number of variables from any other controller. For more details refer to *System Guide Technical Data and Configuration (3BSE041434*).*

•   A maximum of three OPC servers may subscribe to Data Access and/or Alarm and Event messages from one controller.

If third-party OPC clients are used, there will be further limitations.

For a complete list of basic rules and limitations when configuring OPC Server for AC 800M, see the *AC 800M, OPC Server (3BSE035983*).*

Application Size Limit

The total amount of memory allocated for a complete application in the controller is a critical factor. This section provides general guidance regarding the size of an application.

Using simple calculations of the amount of memory allocated for a function block or control module, it is possible to obtain a fairly good estimate of the actual size of an application.

This raises the important question: How large an application can I download and still maintain safe operation? A rule of thumb is: *never download an application, so large that it cannot be downloaded to the controller twice*.

When the template and added hardware have been downloaded, available memory in the controller should be at least twice the application size. The reason for this is that when changes are downloaded, the application running in the controller will be duplicated, along with the new downloaded changes.

Hence, at a certain point there will be 'two' applications, plus the new changes, in controller memory at the same time, thus the need for at least twice the memory size. When the new application has been fully built, the old application will be removed from memory.

For a further discussion of application size and memory consumption, see Calculations and Performance Data on page 39.

Maximum Number of Controllers, Applications, Diagrams, Programs and Tasks

The following limitations apply to the number of controllers and applications that can be handled by one 800xA system, and to the number of applications, diagrams, programs, and tasks that can be handled by each controller.

Table 2. Maximum number of controllers, applications, diagrams, programs, and tasks

Item	Maximum Number
AC 800M controllers	32 per control project
Applications	1024 per control project
Applications	32 per controller
Applications	One per task
Programs	64 per application
Diagrams	128 per application
Tasks	32 per controller

Maximum Number of POUs and Variables

The number of variables, function blocks, and control modules which can be used in an application, a diagram, a program, a control module, or a function block, may be very high, but it is not unlimited.

The total sum of diagrams, programs, control modules and function blocks in an application is 65534. The maximum number of variables in an application, or a control module type, or a program, or a diagram, or a function block type is 65535.

If you reach this limit, you should split your application (or type) into several. For details on other options, refer to *System 800xA, Control, AC 800M Configuration (3BSE035980*)*.

Applications

The maximum number of variables in one application is comprised of:

- Global variables with their total number of components,

- Local variables with their total number of components,

- Each program corresponds to one variable,

- Each diagram corresponds to one variable.

Control Module Type

The maximum number of variables in one control module type is made up from:

- Parameters (two variables per parameter),

- Local variables with their total number of components,

- Variables representing each graphical connection in FBD between sub-function blocks, with their total number of components,

- Variables representing each graphical connection in CMD between sub-control modules, with their total number of components,

- Variables representing each unconnected parameter on sub-control modules, with their total number of components,

- Variables representing each unique literal used in connections to sub-control modules (one per literal),

- Variables representing each SFC code block (two per code block),

- Variables representing each SFC state (three or four per SFC state),

- Project constants used in the control module type,

- Each function block corresponds to one variable.

Program

The maximum number of variables in one program is made up from:

- Local variables and Communication variables (total number of all components),

- Variables representing each graphical connection in FBD between sub-function blocks (total number of all components),

- Variables representing each SFC code block (two per code block),

- Variables representing each SFC state (three or four per SFC state),

- Project constants used in the program,

- Each function block corresponds to one variable.

Diagram

The maximum number of variables in one diagram is made up from:

- Local variables and Communication variables (total number of all components),

- Variables representing each graphical connection in the FD code block,

- Variables representing each optional SFC code block (two per code block),

- Variables representing each SFC state in the SFC code block (three or four per SFC state).

Function Block Type

The maximum number of variables in one function block type is made up from:

• External variables (one per external),

• IN_OUT parameters (one per parameter),

• IN parameters with their total number of components,

• OUT parameters with their total number of components,

• Local function block variables with their total number of components,

• Variables representing each graphical connection in FBD between sub-function blocks with their total number of components,

• Variables representing each unconnected parameter on sub-control modules with their total number of components,

• Variables representing each unique literal used in connections to sub-control modules (one per literal),

• Variables representing each SFC code block (two per code block),

• Variables representing each SFC state (three or four per SFC state),

• Project constants used in the function block type,

• Each function block corresponds to one variable.

Diagram Type

The maximum number of variables in one diagram type is made up from:

• Parameters (two variables per parameter),

• Local variables with their total number of components,

• Variables representing each graphical connection in the FD code block,

• Variables representing each optional SFC code block (two per code block),

• Variables representing each SFC state in the SFC code block (three or four per SFC state)

Structured Variable and Project Constants

•   The length of the name of each component in a structured variable or project constant is limited to 32 characters,

•   The total length of the name of a project constant (that is, the sum of the number of characters in each component name) is limited to 140 characters.

INSUM Limitations

For information on INSUM-specific limitations, see INSUM performance data in the *System Guide, Performance and Capacity (3BSE041434*)*.

Section 2 Programming Languages

This section gives an overview of the six programming languages in Control Builder, and provides guidelines on selecting the language that suits your application best.

Among the six languages, five languages are defined in the IEC 61131-3 standard, see also Appendix A, IEC 61131-3 Standard. However, there are some differences compared to the implementation in Control Builder. The sixth language, FD, is another graphical programming language.

Control Builder online help provides detailed information on the use of instructions, functions, expressions, and so on, in the different languages.

General

Depending on previous experience, programmers often have their own personal preference for a certain language. All the languages have advantages and disadvantages, and no single one of them is suitable for all control tasks. We start with three basic statements and then proceed to some important characteristics of each language.

- In small applications with relatively few logical conditions, the demand for good structure and re-use of code is not as great as in larger systems.

- ST and IL are textual languages, while FBD, LD, SFC, and FD are based on graphical metaphors.

- LD and IL are not as powerful as ST or FBD or FD.

- FBD is not as powerful as FD.

Note that the definition of function block is allowed in all the six languages. A function block is a method of encapsulating the code in a "black box" with inputs and outputs.
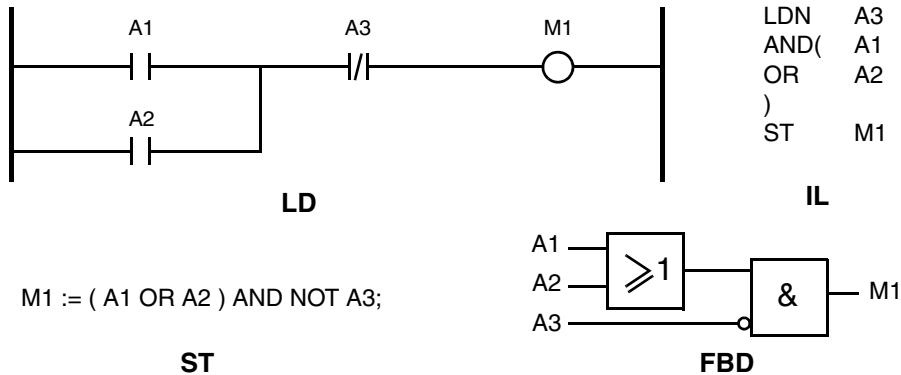
Figure 18. A Boolean condition programmed with four of the five IEC 61131-3 programming languages. SFC is normally only used for sequences

Some important characteristics of the languages are listed in the Table 3.

Table 3. Control Builder programming languages

Language	Function	SIL
Function Diagram (FD) on page 55	Function Diagram (FD) is a graphical language that allows mixing of functions, function blocks, control modules, and diagrams in one code block and create graphical connections between them.	Non-SIL, SIL1-2, and SIL3
Function Block Diagram, FBD on page 63	A graphical language for depicting signal and data flows through function blocks and re-usable software elements. Function blocks and variables are interconnected graphically, which makes the resulting control diagrams easy to read.	Non-SIL, SIL1-2, and SIL3
Structured Text, ST on page 61	A high-level programming language. ST is highly structured and has a comprehensive range of constructs for assignments, function/function block calls, expressions, conditional statements, iterations, etc. It is easy to write advanced, compact, but clear ST code, due to its logical and structured layout.	Non-SIL, SIL1-2, and SIL3

Table 3. Control Builder programming languages (Continued)

Language	Function	SIL
Instruction List, IL on page 78	A traditional PLC language. It has a structure similar to simple machine assembler code.	Non-SIL
Ladder Diagram, LD on page 71	Ladder diagram (LD) is a graphical language based on relay ladder logic.	Non-SIL
Sequential Function Chart, SFC on page 82	Sequential function chart (SFC) is a graphical language for depicting the sequential behavior of a control program.	SIL1-2 and Non-SIL

Table 4.

Function Diagram (FD) on page 55	Function Diagram (FD) is a graphical language that allows mixing of functions, function blocks, control modules, and diagrams in one code block and create graphical connections between them.

Function Diagram (FD)

The Function Diagram (FD) language is a graphical code block language that allows mixing of functions, function block instances, diagram instances and control module instances in the same editor.

The FD language is used in:
• Diagram editor
• Diagram Type editor
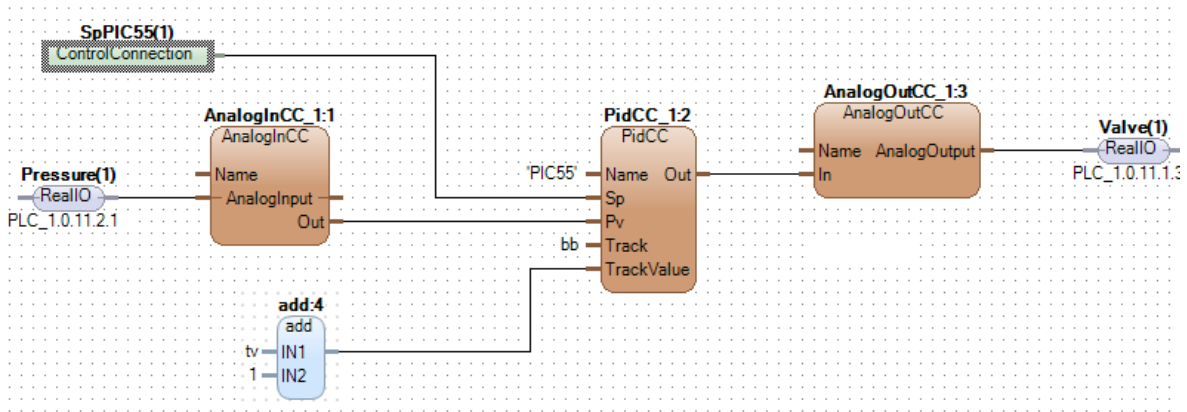
Figure 19. Example of control logic created in FD

The FD language has three main elements that help in creation of the logic:
- Pages
- Invocation blocks (objects)
- Data connections

Pages in FD

The Page element in FD represents one page in a diagram or diagram type.
It contains objects and data connection elements. Pages are numbered starting at 1.
Page numbers are unique within one diagram or diagram type.

Invocation Blocks (Objects) in FD

An Invocation Block (object) represents an invocation of an executable unit such as a POU instance or a code block.

The name of the block is followed by a colon':' and the data flow order number. This is placed in a label just above the block.

It is also possible to add a block description to the block. It is placed above the name label. The description is anchored to the block.
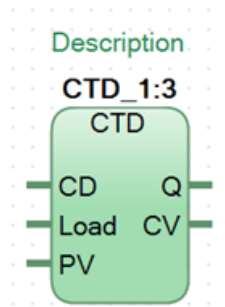
Figure 20. Example of an object in FD

An object can be freely placed on a diagram page, as long as it is completely inside the page boundary.

Data Connection through Connection Ports

Blocks display connection ports which are used as endpoints for data connections. The type of block determines the connection ports that are displayed. For example, a block representing an invocation of a system function displays connection ports for the parameters to the function and also a connection port representing the return value of the function.

A connection port consists of:
- Name of the corresponding parameter
- Data type of the corresponding parameter (not shown in the diagram, but displayed as tool tip at the port)
- Direction (In, Out, In_Out, or Unspecified), which determines the position (left or right, or on both sides)
- Negated - Boolean attribute, which depends on the Inverted command at the port.
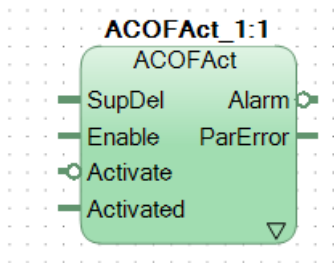
Figure 21. Example of connection ports with negation (Activate and Alarm)

Some blocks display built-in connection ports. One example is invocation blocks, invoking SFC code blocks. These blocks display the built-in control variables such as Hold, Reset, and so on, as connection ports.
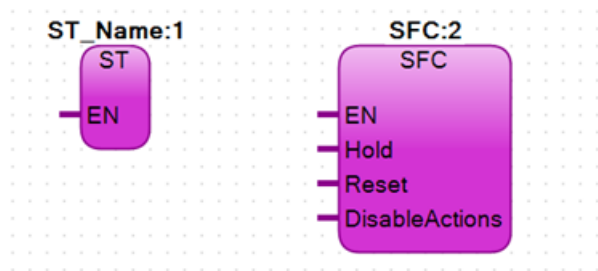
Figure 22. Examples of code block invocations in FD

Representation of in_out Parameter

Depending on block type, *in_out* parameters have ports on both sides.
Function blocks, Control Modules and Diagram instances are displayed with ports on both sides when the *FD Port* property is set to Yes or No.

Graphical Data Connections

A graphical data connection is drawn from an output port (the source, placed on the right side of an object) to an input port (the destination, placed on the left side of an object). The graphical data connections are also allowed across pages (source and destination ports in different pages), and in this case, a page connector is displayed.
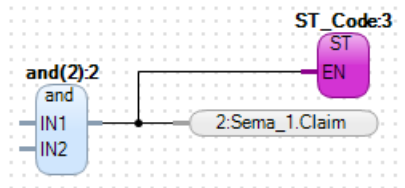
Figure 23. Example of graphical connection between ports

Textual Data Connections

A textual data connection is displayed as a label directly on the port of an object.

It is possible to make the following textual data connections to a port:
- Literal value. For example, *true*, *8.5*, *'My string value'*, and so on
- Project constant
- Variable name
- Communication variable name
- Parameter name (only in diagram type)
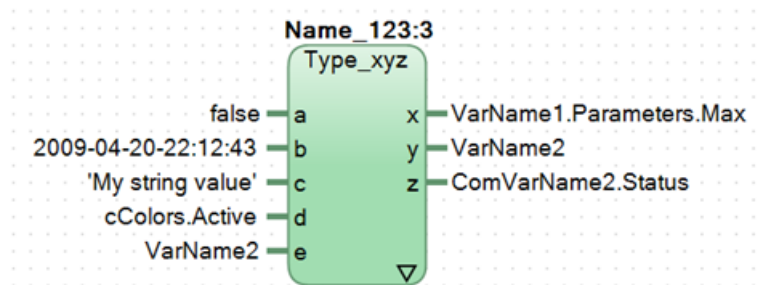
Figure 24. Example of textual data connections

Page Connector

A Page Connector block is a reference to a connection port on a block located on another page than the page connector itself. The page connector allows connections between ports on blocks on different pages in a diagram.

Execution of Diagram and Diagram Type

The execution of the content in a diagram or diagram type is mainly configured using Data Flow Order of different invocations within the FD (Function Diagram) code block. The Data Flow Order is a number that specifies the intended order of execution. Each diagram must have exactly one FD code block. In the FD code block, the Data Flow Order is given to invoke functions, function blocks, diagram instances, control modules, code blocks, split blocks and join blocks.

Control modules in a diagram are sorted based on access of variables to enable both forward and backward calculations and data flows to be executed in the same task scan. Therefore code blocks in invoked control modules will not always be executed in the order specified by the Data Flow Order.

The Structured Text and SFC code blocks can be defined without invoking them from the FD code block. These code blocks are then sorted together with code blocks of invoked control modules.

Asynchronous execution

The diagram supports asynchronous execution of both function block instances and control module instances. If an invocation block referring to an asynchronous function block is placed in a diagram, the invocation of this block results in passing parameter to the function block instance. This works similarly as invocations to asynchronous function block instances in Structured Text.

The code block sorting of function diagram is affected, if invocation block refers to an asynchronous control module instance. Asynchronous control module instances are not sorted together with the sorting group defined by the function diagram. Instead, they are sorted with the sorting group defined by the task to which they are connected.

Structured Text, ST

Structured Text (ST) is a high-level programming language, similar to *Pascal* and *C*, that has been specifically designed for use in programmable controllers. It is compact, highly structured and contains a comprehensive range of constructs for assignments, function/function block calls, expressions, conditional statements, iterations and more. The code is simple to write and easy to read, because of its logical and structured layout. The compactness of the language allows a clear overview of the code and less scrolling in the editor. Tabs and spaces are used to structure the code for easy reading.

ST code can be written using any text editor, for example Microsoft Word, and then copied and pasted into the Structured Text editor code pane in Control Builder. Note however, that you only have access to online help (use the F1 key) in the editor of Control Builder.

Suitable for Complex Calculations and Looping

The ST language has an extensive range of constructs for assigning values to variables, calling function blocks and creating conditional expressions. This is very useful for evaluating complex mathematical algorithms, commonly used in analog control applications.

No other IEC language can match the power of ST when iterations are needed, that is, when certain parts of the program code are to be repeated a fixed or a conditional number of times.

High Threshold for Programmers

Of the five IEC languages, Structured Text is often the natural choice for people with former experience in computer programming. Control engineers without computer knowledge sometimes consider ST to be more complex with a higher learning threshold than the LD or IL languages.

On the whole, ST is fairly easy to learn and a very effective tool for developing control applications. The language is a good general purpose tool for expressing different types of behavior with all kind of structured variables.
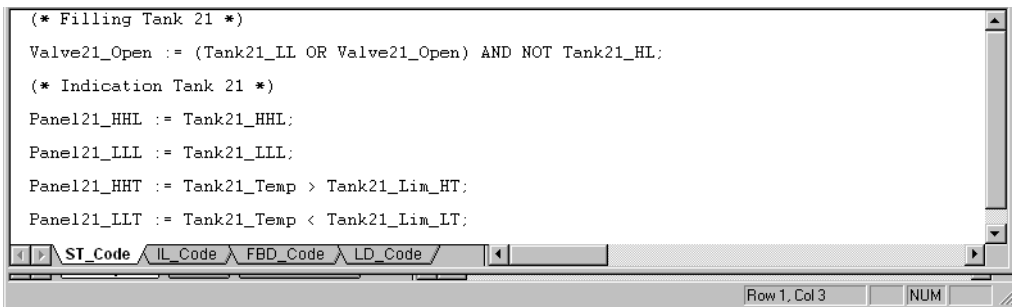
Most programmable controllers supporting the SFC language use ST as the default programming language to describe the step actions in sequences.

Functions in ST

Statements

The ST language contains a list of statements, such as assignment statements (variable:= expression), conditional statements (*if, then, else, case*), iteration statements (*for, while, repeat*) and control statements (*exit, return*). Statements contain *expressions* which, when evaluated, result in a value of a variable having any kind of data type.

Statements should be written in a structured way, similarly to when programming in Pascal or C.

```
(* Filling Tank 21 *)

Valve21_Open := (Tank21_LL OR Valve21_Open) AND NOT Tank21_HL;

(* Indication Tank 21 *)

Panel21_HHL := Tank21_HHL;

Panel21_LLL := Tank21_LLL;

Panel21_HHT := Tank21_Temp > Tank21_Lim_HT;

Panel21_LLT := Tank21_Temp < Tank21_Lim_LT;
```
ST_Code IL_Code FBD_Code LD_Code

Row 1, Col 3   NUM

Figure 25. Example of ST code

Expressions

ST uses Boolean expressions (*and, or, not, xor*), arithmetic expressions (+, -, *, **, *mod*), and relational expressions (=, >=, >, <=, <, <>). An expression using these operators always results in a single value. An expression contains operators, functions and operands. Operators may be +, -, /. Functions may be, for example, sin(x) or cos(x). The operand can be a value, a variable, a function or another expression.

When you run your code in Test mode, it is possible to view the code in Ladder or Function Block Diagram. Select **Tools> Setup** in the menu of the code block where the code is written (you must be in Test or Online mode when performing this).

Function Blocks

Function blocks are called by a statement consisting of the function block name followed by a list of named inputs and output parameter value assignments. The programmer selects any available function block from a list and enters the values.

```
Timer( IN := switch3,
  PT := delay1,
  Q => lamp;
```

The code above shows a function block in ST with input and output parameters.

Execution Rules

The priority of operators determines the order of evaluation of an expression. An expression in parentheses has the highest priority and the OR expression has the lowest priority.

Code blocks are executed from left to right, see Figure 25.

Function Block Diagram, FBD

Function Block Diagram (FBD) is a high-level graphical programming language in which the control function is divided into a number of *function blocks* or *functions* connected by flow signals. A function block may contain simple logical conditions, timers or counters, but can also provide a complex control function to a subprocess in a machine or even an industrial plant.

FBD describe the POUs in terms of processing elements and displays the signal flow between them, similarly to electronic circuit diagrams. It represents the function block and functions by graphical symbols (boxes), their input and output parameters by pins on the boxes and the assignment of parameters by lines between the pins. A comprehensive range of basic function blocks and functions is available.

A code block may contain an unlimited number of pages.
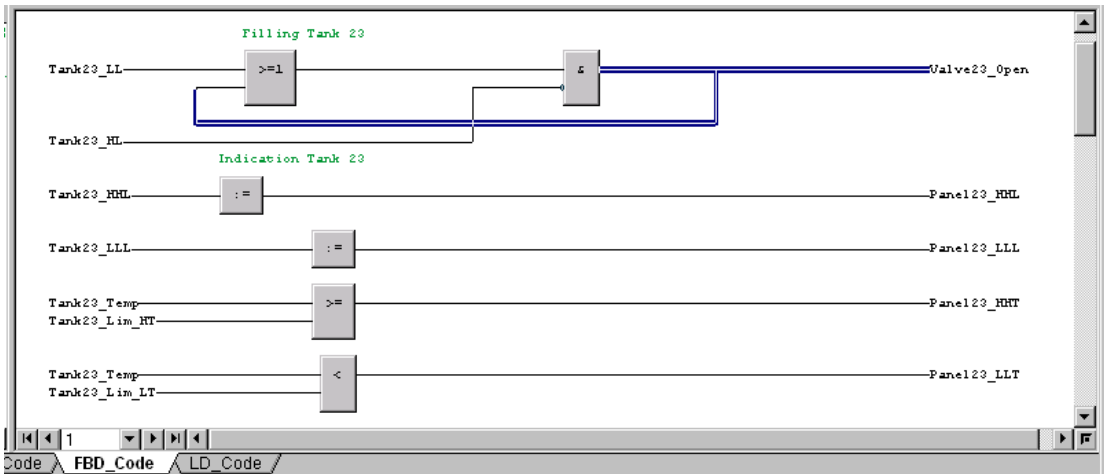
Figure 26. Example of FBD code
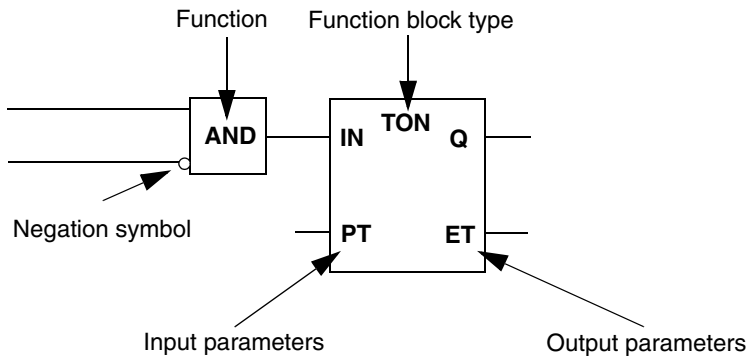
Figure 27. Some fundamental rules for drawing function block diagrams

Similar to Electrical Diagrams

In many ways, a function block can be compared to an *integrated circuit (IC)*, the building block of today's computers and other electronic devices. Like ICs, function blocks can provide standard solutions to common control functions. The connection

lines between blocks symbolize signal flow in the system. Electrical engineers who have experience in designing and analyzing circuit diagrams often have a preference for programming with FBD.

Boolean Functions and Feedback are Easy to Implement

FBD is very suitable for describing Boolean logic with associated timers, counters and bistables. Most programmable controllers have such function blocks predefined in *standard libraries* for direct use by the programmer. There is no other programming language where timers and counters are so easy to implement as in FBD.

Many analog control systems, for example PID controllers, use closed-loop control where some output signals are fed back and used as inputs in the control algorithm. The FBD program gives a good overview of signal flow in systems with feedback.

Not Suitable for Conditional Statements

FBD programs have very weak support for conditional statements when one or more actions are to be repeated for a specified number of times, or only as long as a certain condition is fulfilled.

This kind of construct is much easier to accomplish in the ST language with one of the statements FOR, WHILE, REPEAT, CASE or IF.

Functions in FBD

When graphically connecting two functions that have parameters of data type string, the system will create an intermediate string variable limited to 40 characters (default length of string data type). This means that strings may be truncated to 40 characters.

Basic Functions

The following basic FBD language functions correspond to the operators of textual programming languages.

- Assignment functions (*move, :=*)

- Boolean functions (*not, and, &, xor, or, >=*)

- Arithmetic functions (*expt, mul, div, add, sub, mod, abs*)

- Relational functions (*<, >, <=, >=, =, <>*).

Connections

In the Function Block Diagram editor, the parameters of functions and function blocks are shown as pins on the boxes. The assignment of values (variables and constants) to parameters is shown by lines connecting the pins.

If a parameter is assigned to another parameter, one of them must be an output parameter and the other an input parameter (an output parameter can be assigned to any number of input parameters but never to another output parameter).

All function blocks have a built-in algorithm for calculating output values based on the status of the inputs.

When working with Boolean signals, negated inputs or outputs can be shown using a small circle placed at the corresponding line, close to the block symbol. Some systems use a NOT function block instead of the circle.

Execution Rules

The evaluation of parameter values corresponds to the execution order of the function blocks and functions within the POU. The execution order is represented by the order of the graphic symbols (boxes) in FBD, from left to right, and from top to bottom. You can change the execution order later, by moving the selected function blocks and functions.

Standard Function Block Types in FBD

The IEC 61131-3 standard defines a small repertoire of rudimentary standard function block types. These are predefined in most of today's programmable controllers. Standard function blocks are often used to construct user-defined function blocks. The most commonly used blocks are:

- Boolean conditions like AND, OR, XOR and NOT

- Bistables

- Edge detectors

- Timers

- Counters

Bistables

Two types of bistables are available, SR and RS. Both of them have two Boolean inputs and one output. The output is set (SR) or reset (RS) as a memory when the triggering input (S1 or R1) momentarily becomes true. When the other input becomes true the output returns to its initial state. If both inputs are true the SR will be set while the RS will be reset.
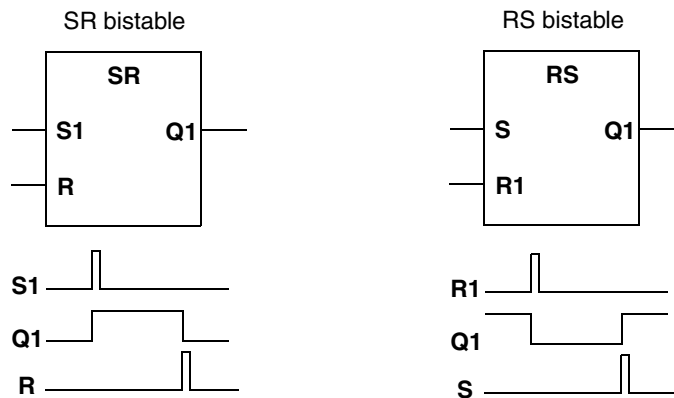
Figure 28. SR and RS bistable symbols with their corresponding functions below

Edge Detectors

There are two edge-detecting function blocks, Rising edge trigger (R_TRIG) and Falling edge trigger (F_TRIG), which are used to detect the changing state of a Boolean input. The output of the blocks produces a single pulse when a transition edge is detected.

When the input changes state, according to the type of edge detector, the output is true during one function block execution. After that the output remains false until a new edge is detected.
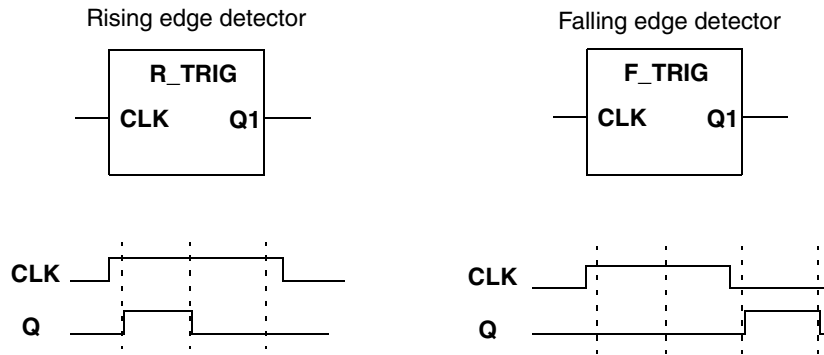
Figure 29. Edge detectors create a single pulse with the same duration as the execution time of the function block

Timers

Timers are among the most used function blocks in a control application. Whenever there is a need for a time delay between a change of state and the corresponding action a timer can be used. In most programmable control systems the timing is based on the CPU system clock, which means that the specified time intervals are very precise.

There are three different types of timer function blocks, pulse timers (TP), on-delay timers (TON) and off-delay timers (TOF). All of them have a Boolean input called IN, a Boolean output called Q, an input of type time called PT and an output of type time called ET.

The required delay (or pulse width) is specified on input PT (Preset Time) while the actual elapsed time is shown on output ET (Elapsed Time).

A pulse timer is normally used to generate output pulses of a specified duration. When input IN changes to the true state the output Q follows and remains true for a duration specified by input PT. The elapsed time ET is increased linearly as long as the pulse output is true. When the pulse terminates, the elapsed time is held until the input changes to false. Note that the output Q will remain true until the pulse time has elapsed, even if the input changes to false.

Both delay timers are used to delay an output action by the specified time PT when a certain condition becomes true.

The on-delay timer delays the activation of an output. When the input IN becomes true the elapsed time at output ET starts to increase. If the elapsed time reaches the value specified in PT, the output Q becomes true and the elapsed time is held. The output Q remains true until input IN becomes false. If input IN is not true longer than the specified delay in PT, the output remains false.

The off-delay timer delays the deactivation of an output. When the input IN becomes false, the elapsed time starts to increase and continues until it reaches the specified delay given by PT. The output Q is then set to false and the elapsed time is frozen. When input IN becomes true the output Q follows and the elapsed time is reset to zero.
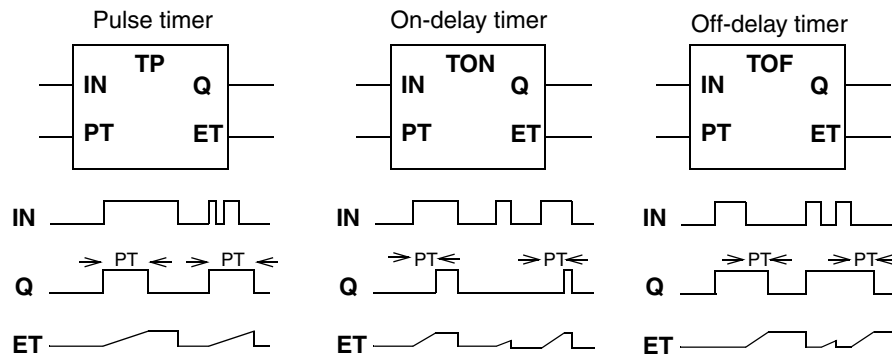
Figure 30. Timing diagrams for the three different types of timer function blocks

Counters

Counters are another commonly used type of function block. These are designed to be used in a wide range of applications, for example counting pulses, revolutions, completed production batches, etc.

There are three types of counter blocks, up-counters (CTUs), down-counters (CTDs) and up-down counters (CTUDs). CTUs are used to indicate when the counter has reached a specified maximum value. CTDs indicate when the counter reaches zero, on counting down from a specified value. CTUDs can be used to both count up and count down and have two outputs indicating both maximum value and zero.

A CTU has three inputs and two outputs. A CTU block counts the number of pulses (rising edges) detected at the Boolean input CU. The input PV (Preset Value) of data type integer defines the maximum value of the counter. Each time a new rising edge occurs on CU the output CV (Counter Value) of type integer is incremented by one. When the counter reaches the value specified in PV, the Boolean output Q becomes true and counting stops.

If necessary, the Boolean input R (reset) can be used to set the output Q to false and to clear CV to zero.
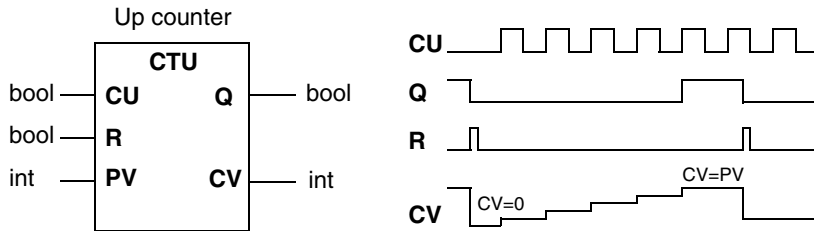
Figure 31. Example of a CTU counter block with preset value PV=5

The CTD is very similar to CTU with three inputs and two outputs. A CTD counts down the number of pulses detected at the Boolean input CD. The input PV is used to specify the starting (integer) value of the counter. Each time a new rising edge occurs on CD the output CV is incremented by one. When the counter reaches zero, the output Q becomes true and counting stops.

If necessary, the Boolean input LD (load) can be used to clear the output Q to false and to load the output CV with the value specified in PV.
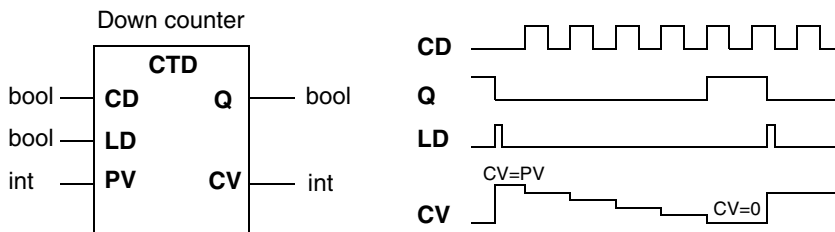
Figure 32. Example of a CTD counter block with preset value PV=5

The CTUD is a combination of the other two counter blocks. It has two Boolean inputs, CU and CD, used for counting up and counting down the value in output CV. Similarly to the two other counters, the integer input PV defines the counter's

maximum value. When the counter reaches the value specified in PV the output QU is set to true and counting stops. In a similar way, the output QD is set to true and counting stops when the counter reaches zero.

If necessary, the input LD can be used to load the value from PV to the output CV while the input R can be used to clear the output CV to zero.
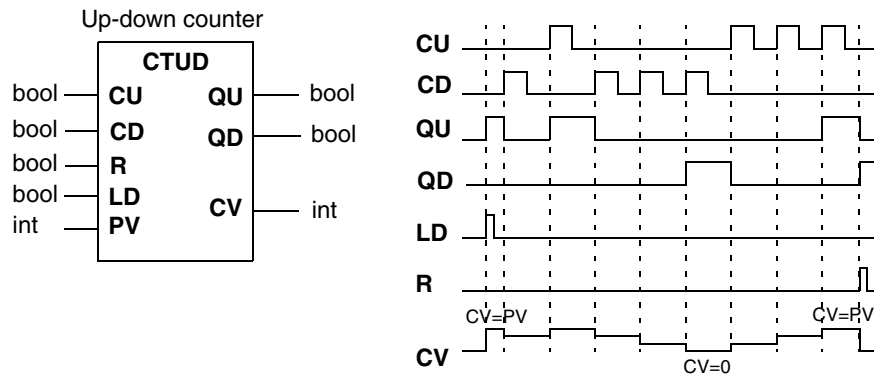
Figure 33. Example of a CTUD counter block with preset value PV=3

The CTUD is often used in applications where there is a need to monitor the actual number of items in a process. It could, for example, be used to count the number of products placed on and taken off a store shelf.

Ladder Diagram, LD

Ladder Diagram (LD) is a graphical language. LD describes the POUs in a way similar to relay logic. In LD, you can implement complex AND/OR logic based on the idea of power flow from a power rail through relay contacts and coils, to the other power rail. You can also add function blocks and functions to the power rails and LD presents them similarly to a Function Block Diagram (FBD). The use of the LD editor is especially advantageous with small systems, and if you are familiar with electrical wiring diagrams and relay control.
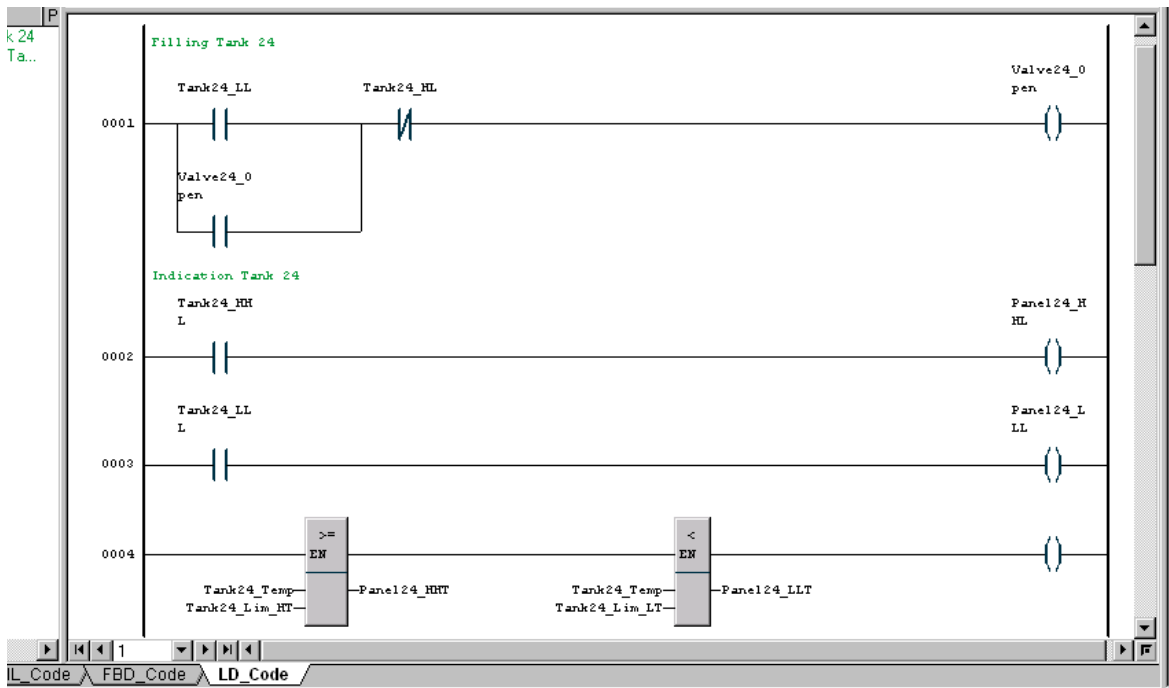
Figure 34. Example of LD code

Contacts represent inputs from the process and *coils* outputs. An LD diagram is limited on both sides by vertical lines, called *power rails*. The power rails serve as a symbolic electrical power supply for all the contacts and coils that are spread out along horizontal rungs.

Each contact represents the state of a Boolean variable, normally a transducer, but sometimes also an internal variable in the control system. When all contacts in a horizontal rung are made, i.e. in the true state, power can flow along the rail and operate the coil on the right of the rung. The coil normally represents physical objects like a motor or a lamp, but may also be an internal variable in the control system.

There are two types of contacts, normally open and normally closed. Contacts which are normally open present a true state (Boolean variable is 1) when they are

closed. Normally closed contacts present a false state (Boolean variable is 0) when they are closed.

In analogy with electrical circuits, contacts connected horizontally in series represent logical AND operations. Parallel contacts represent logical OR operations.

It is possible to create LD programs that contain *feedback loops*, where the variable from an output coil is used as an input contact, either in the same or in other logical conditions. In a real-world relay circuit this is equivalent to using one of the relay's physical switches as an input contact. A person with experience in computing would probably call this a *memory* bit.
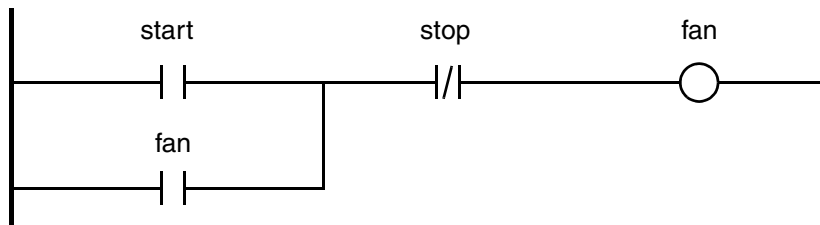
Figure 35. Feedback loop in an LD program. The fan starts with an impulse on contact start and continues to run until the contact stop is opened

Easy to Understand

Programming with LD can be learnt relatively quickly and the graphical presentation is easy to follow. The method is particularly easy to understand by people who are familiar with simple electrical or electronic circuits.
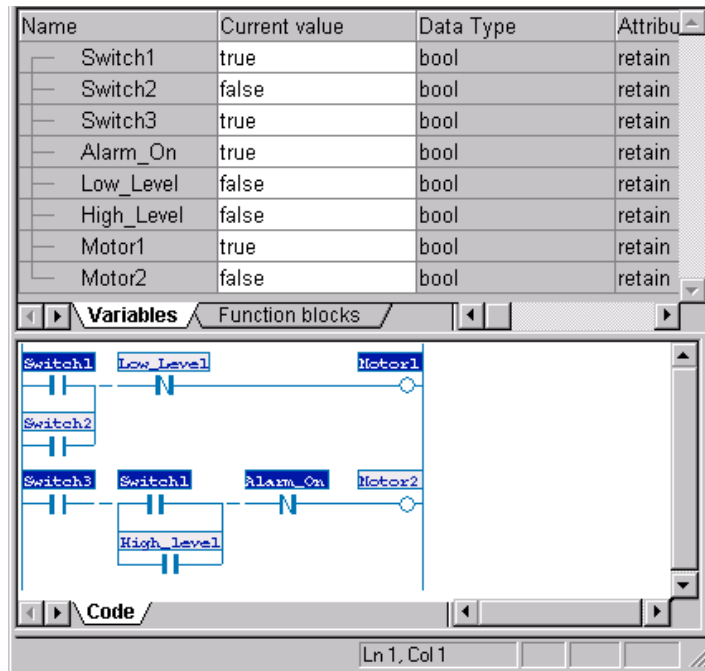
Figure 36. Status indication of an executing LD program

LD programs are very popular among maintenance engineers since faults can easily be traced. Most programming stations generally provide an animated display showing the live state of transducers while the programmable controller is running. This provides a very powerful online diagnostics facility for locating incorrect logic paths or faulty equipment.

Weak Software Structure

Ladder programming is a very effective method for designing small control applications. With increasing processing power and memory size with today's programmable controllers, the method can also be used to construct large control systems. Unfortunately, large ladder programs have several serious drawbacks.

Since most programmable controllers have limited support for program blocks, or *subroutines*, it is difficult to break down a complex program hierarchically.

The lack of features for passing parameters between program blocks makes it difficult to break down a large program into smaller parts that have a clear interface with each other. Usually, it is possible for one part of a Ladder Diagram to read and set contacts and outputs in any other part of the program, which makes it almost impossible to have truly *encapsulated data*.

This lack of data encapsulation is a serious problem when large programs are written by several different programmers. There is always a danger that internal data in one block can be modified by faulty code in other program blocks. Each programmer, therefore, has to be very careful when accessing data from other program blocks.

There are also problems in using structured data with ladder programs since data are normally stored and addressed in single memory bits. Many control applications often have a need to group data together as a structure. Some *sensors* provide more than one variable that has to be recorded by the control system. Apart from the physical value measured by the sensor, the application sometimes needs to disable the sensor, place it in test mode, record the time when the sensor is active and also raise an alarm if the sensor is activated longer than a certain prescribed period.

All of this information from the sensor should ideally be handled as a single structure that can be addressed using a common name. In most ladder programs such data is often spread out among different ladder rungs. Without a data structure the programmable controller has no provision for warning the programmer when incorrect data are accessed.

Limited Support for Sequences

Most control applications have a need to divide the function into a *sequence* of *states*. Each state represents a unique condition in the plant being controlled. Normally, only one state is active at a time.

When sequences are constructed with ladder programming the normal method is to assign one internal memory bit to each state and to use contact conditions from the transducers to trigger transitions between the states. Each state consists of a feedback loop using the memory bit as an alternative condition for remaining in the state. The feedback loop of a state is normally broken by the memory bit of a succeeding state. To get real-world actions the memory bits are used as conditions in separate rungs to control the outputs.
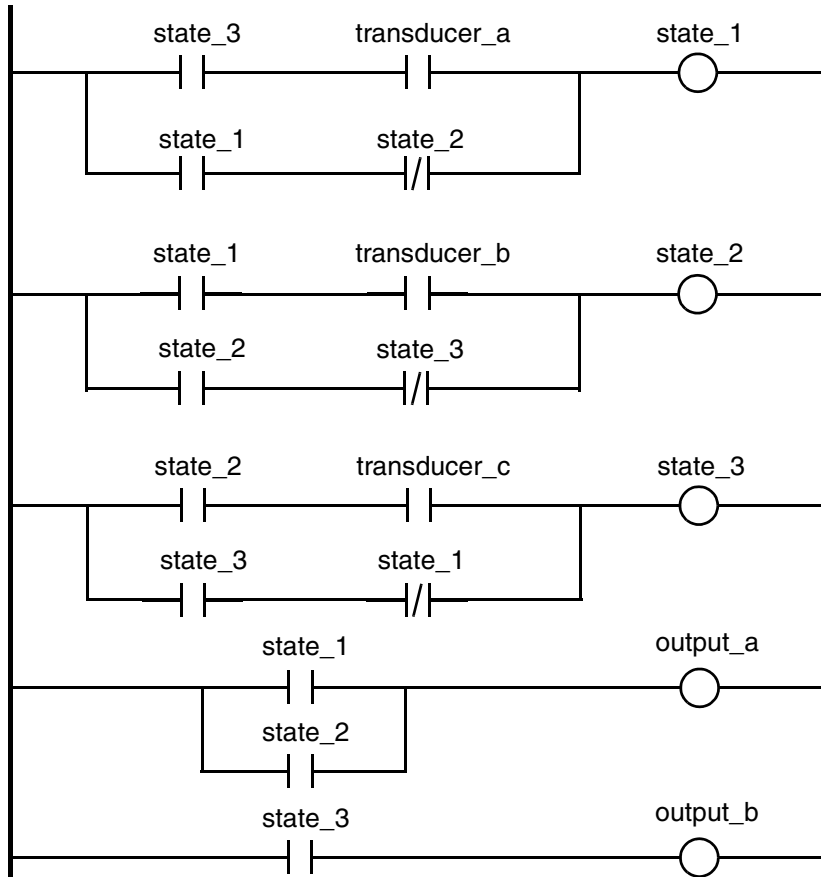
Figure 37. Sequence program with three states controlling two outputs

From the above example it is obvious that ladder programs with sequences can become very large and difficult to maintain. The most obvious problem is that control of the memory-based sequence model is mixed with the application logic so the behavior of the complete program is difficult to understand and follow.

Difficult to Reuse Code

In many large control systems similar logic strategies and algorithms are used over and over again. A common application is to detect fire by using two or more transducers with a comparison algorithm to eliminate false alarms. Such systems consist of a large number of similar ladder rungs with only minor modifications to read different contacts and to set different outputs. This can result in very large, unstructured programs.

Functions in LD

Basic Functions

The basic functions in the LD language are the same as the basic functions in FBD, see Basic Functions on page 65.

Connections

You can assign variables to coils of rungs and output parameters of function blocks and functions. The variables assume the values of the corresponding coils and output parameters. You can assign values to contacts of rungs and input parameters of function blocks and functions. The value can either be a variable, such as one with the value of an output parameter, or a constant. The assignment of parameters is shown by variable names, constant names and lines between the pins on boxes symbolizing the function blocks and functions.

Execution Rules

The evaluation of parameter values corresponds to the execution order of the rungs, function blocks and functions within the POU. The execution order is represented by the order of the rungs in LD from the top to the bottom. You can change the execution order later by moving the selected rungs up or down within the POU, for example, by cutting and pasting, or by moving them in the Structure pane.

The execution order of function blocks and functions within a rung is defined by their position. They are executed from left to right, as the current flows from the left power rail to the right one.

Instruction List, IL

Instruction List (IL) is a low-level language in which the instructions are listed in a column, one instruction on each line. It has a structure similar to simple machine assembler code.

IL has been chosen as the preferred language by a number of PLC manufacturers for their small to medium-sized systems. The lack of structured variables and weak debugging tools make the language less suitable for larger systems.

IL code can be written in Microsoft Excel, and then copied and pasted into the Instruction List editor code pane in Control Builder. Note, however, that you can only have access to online help (use the F1 key) in the editor of Control Builder.

	Label	Instruction	Operand	Description
1	Filling_Tank_22			
2		LD	Tank22_LL	
3		OR	Valve22_Open	
4		ANDN	Tank22_HL	
5		ST	Valve22_Open	
6	Indication_Tank_22			
7		LD	Tank22_HHL	
8		ST	Panel22_HHL	
9				
10		LD	Tank22_LLL	
11		ST	Panel22_LLL	
12				
13		LD	Tank22_Temp	
14		GT	Tank22_Lim_HT	
15		ST	Panel22_HHT	
16				
17		LD	Tank22_Temp	
18		GT	Tank22_Lim_LT	
19		ST	Panel22_LLT	
20				

\ ST_Code \ **IL_Code** \ FBD_Code \ LD_Code /

Row 1, Col 1   NUM

Figure 38. Example of IL code

Best System Performance

IL is ideal for solving small straightforward problems. In the hands of an experienced programmer it produces very effective code resulting in applications that are optimized for fast execution.

There is also another reason for using IL in order to optimize system performance. During a period of several years a huge amount of software has been written and thoroughly tested. Such software can be modularized into libraries and reused even by programmers with no detailed knowledge of the internal behavior.

Weak Software Structure

Since IL is a low-level language, great care should be taken in structuring the code so that it is easy to understand and maintain. It is very important that IL programs are well documented since conditional jumps will otherwise be very difficult to follow.

The behavior of the result register, with only one value available at a time, makes it difficult to work with structured data variables. Most compilers have no automatic function for checking whether the RR contains correct data for the actual instruction code. Therefore, it is up to the programmer to ensure that each instruction is given correct variable data.

Machine-dependent Behavior

Of all the five IEC languages, IL has been found to be the most controversial. Unfortunately, the semantics, i.e. the way in which the instructions operate, are not fully defined in the standard. For example, it is unclear how the result register stores values of different data types. Normally, the RR is not intended for storing structured data, which means that it is very difficult to obtain consistent behavior when working with arrays or strings.

Another problem is that the control system behavior for error conditions is not defined. This means that different system types may respond differently if the programmer uses inappropriate data types. Errors can normally only be detected when the system is running the application.

Functions in IL

Instructions

The following instructions are available in the IL language.

- Load and store (*ld, ldn, st, s, r*)

- Return (*ret, retc, retcn*)

- Jump (*jmp, jmpc, jmpcn*)

- Function block call (*cal, calc, calcn*)

Expressions

Expressions available in IL are boolean expressions (*and, andn, or, not, xor, xorn*), arithmetic expressions (*add, sub, mul, div)*, and relational and equality expressions (*gt, ge, eq, lt, le, ne*). An expression using these operators always results in a single value. An expression contains operators, functions and operands. The operand can be a value, a variable, a function or another expression.

Execution Rules

The instruction list is executed line by line, regardless of what is on the next line, as long as there are no parentheses.

Example

IL programs are often written on a spreadsheet-like form with one column for instructions and another for operands. *Labels*, used to identifying entry points for jump instructions, are placed in their own column to the left of the instruction. The instructions only need to have labels if the program contain jumps. *Descriptions* are placed in a fourth column to the right of the operand. It is strongly advisable to add descriptions to all instructions during programming. Large IL programs without descriptions are very difficult to follow.

Table 5. Example of an IL program for controlling the speed of a motor

Label	Instruction	Operand	Description
	LD	temp1	Load temp1 and
	GT	temp2	Test if temp1 > temp2
	JMPCN	Greater	Jump if not true to Greater
	LD	speed1	Load speed1
	ADD	200	Add constant 200
	JMP	End	Jump unconditional to End
Greater	LD	speed2	Load speed2

To improve readability, IL instructions are normally structured so that labels, instructions, operands and descriptions are put in fixed tabulated positions.

Result Register

The result register (RR) is of central importance in IL. This register is also called the *IL register* or *accumulator*. Current data and the results of calculations, comparisons, loading of variables, etc., are stored in this register.

In the Instruction List (IL) language, literals are assigned the shortest data type that can hold this literal. This might cause unwanted truncations during calculations. To avoid this, use variables with the attribute constant.

Most operations consist of calculation between the result register and the operand. The result of an instruction is always stored in the result register. Most programs start with the instruction LD, which loads the accumulator with a variable. The result register changes its data type automatically during program execution in order to fit the value that needs to be stored.

Programmable controllers normally only have one result register. This must naturally be taken into consideration by the programmer when writing code. The program example in Table 5 first loads the RR with a real variable. The second instruction compares RR with another variable which results in a Boolean TRUE or FALSE result in RR.

The conditional jump instruction JMPCN uses the Boolean value in RR as a condition for either continuing with the next instruction (RR false) or jumping to the label Greater. In both cases, the next instruction loads RR with a new real value. The final instruction stores the RR in a real variable called motor controlling the speed of the motor.

Sequential Function Chart, SFC

The Sequential Function Chart (SFC) programming language allows the user to describe the sequential behavior of the control program graphically. This concept enables all control actions for a process to be described in one compound sequence structure, even if it involves several parallel action chains. Furthermore, sequences can be hierarchical, that is, action chains can be grouped to give a clear, high-level presentation of the process control unit.

A sequence is a unit with a complete sequence, surrounded by an unconditional closed loop; the first step is re-activated when the sequence is complete. A sequence can be divided into separate types of structures. There are two types of structures, sequence selection and simultaneous sequence. It is possible to structure the sequence view into several hierarchical levels with the subsequence function.

SFC Language is not allowed in SIL3 applications.
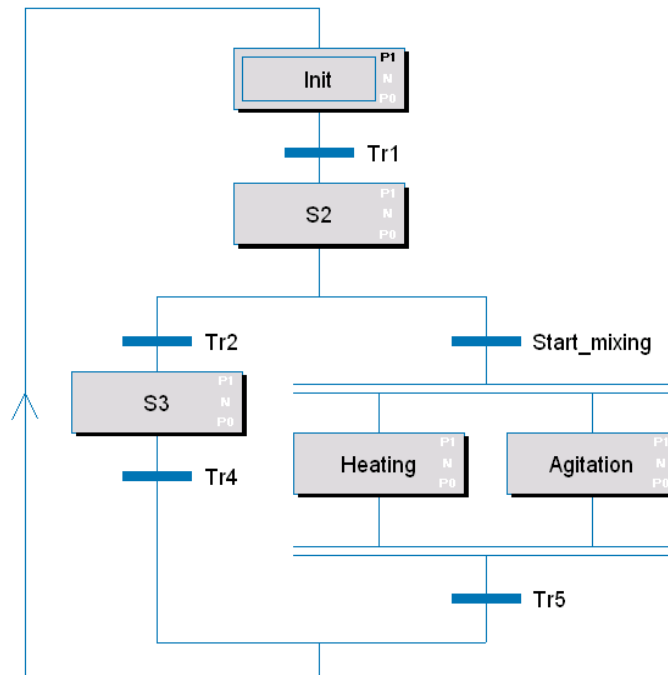
Figure 39. Example of a sequence structure. In the box Init P1 contains code, N and P0 are empty

Powerful Tool for Design and Structuring

SFC is a very suitable top-level design tool in the early phase of a project, but can also be used to describe the more detailed behavior of the plant objects being controlled.

The SFC's graphical metaphor can be used from the beginning, to give an initial representation of the overall behavior of the system. Since the description is very easy to follow SFC is a very suitable means of communication between the customer and the programmer.

In the early phases of a project, there are normally many aspects of the system behavior that have not been defined. By using an easy-to-follow tool for the preliminary specification the number of misunderstandings between customer, system designer and programmer can be reduced to a minimum.

The SFC schemes produced in the first phase of a project can be further specified and refined as new information becomes available. Actions associated with overall steps can then be described via other nested SFC schemes.

The good continuity of SFC from the initial phase to the refining design phases makes it very popular among system designers and programmers.

Other Programming Languages are Needed

Even though SFC has many advantages as a design and structuring tool it is not a complete programming language. Therefore, the transition conditions and action descriptions have to be programmed with one or more of the other four IEC programming languages.

Most experienced programmers prefer the ST language as a complement to SFC. Therefore, the vast majority of programmable controllers use ST as the default language for detailed descriptions in SFC schemes.

Functions in SFC

Basic Functions

The SFC editor contains a number of commands for creating steps, transitions, sequence selections, new branches, jumps, make subsequence etc. Basic elements in a sequence are steps and transitions. Each transition has an associated boolean transition condition.

The actions in a step are written in structured text, ST. To see whether the actions P1, N or P0 contain any code or not, there is an indication for each action type (P1, N and P0) on the right-hand side part of the step-box (see Figure 39). The text color indicates the following:

- White text means that the block exists, but is empty.

- Black text means that the block exists and contains code.

- No color means that the action block does not exist.

Double-clicking on a box expands the information, as shown in Figure 40.
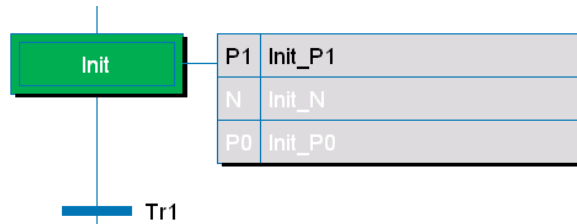
Figure 40. Expanded mode of a box

Sequential Rules

The sequence loop is always closed. The last transition is always connected to the first step. Execution continues from the last step to the first step when the last transition condition becomes true.

Transitions

The transition from one step to another is controlled by transition conditions, which are boolean expressions including process signals.

Automatically Generated Variables

Examples of variables that are automatically generated for a sequence, when the program is compiled: *SequenceName.Reset, SequenceName.Hold, SequenceName.DisableActions* and *SequenceName.Stepname*

Online Functions

In online mode, code and the variable values are displayed in the program editor. Online commands in the menu bar and tool bar buttons for the code are the same as in the other language program editors. Live values for global variables declared in the application root can be shown in SFC Viewer.

Some functions are only available in the online mode, for example:

•   Disable Actions

•   Show Actions

•   Block Transitions

Chart Structure

SFC is a method of dividing the control function into a series of *steps* represented by rectangular boxes and connected by vertical lines. Each step represents a physical state of the system being controlled. On each connecting line there is a horizontal bar representing a *transition*. The transition is associated with a *transition condition* which, when true, deactivates the step before the transition and activates the step after the transition. The execution flow is normally down the page, but SFC can also branch backwards in the chart.

Each step is normally associated with one or more *actions*. These actions describe the actual physical behavior in the plant, e.g. open valve, start motor, and so on. An action can, in some editors, be described directly in the associated step rectangle. However, in most editors the actions are described as separate program statements (normally in ST language) in other code blocks or in a separate editor window associated with the step. An important consideration in SFC programs is that only the code in active steps is executed.

All SFC sequences must have an *initial step* identifying where program execution starts after system initialization. This step is drawn as a rectangular box with double border lines. The initial step remains active until the following transition enables flow to the next step.

Some editors allow the programmer to describe short transition conditions directly on the SFC, close to the corresponding bar. However with more complex conditions it is better to put the code in a separate window.

When the sequence has finished, the flow can be terminated by a step with no associated action. If necessary, the sequence can also repeat the same behavior cyclically. Cyclic execution is enabled by a conditional branch backwards to the first step in the flow. To avoid cluttering the SFC with crossing lines, branches are drawn with a starting arrow where the branch begins and a concluding arrow at the step where the branch ends up. In order to clarify the flow the transition name is written at both places.
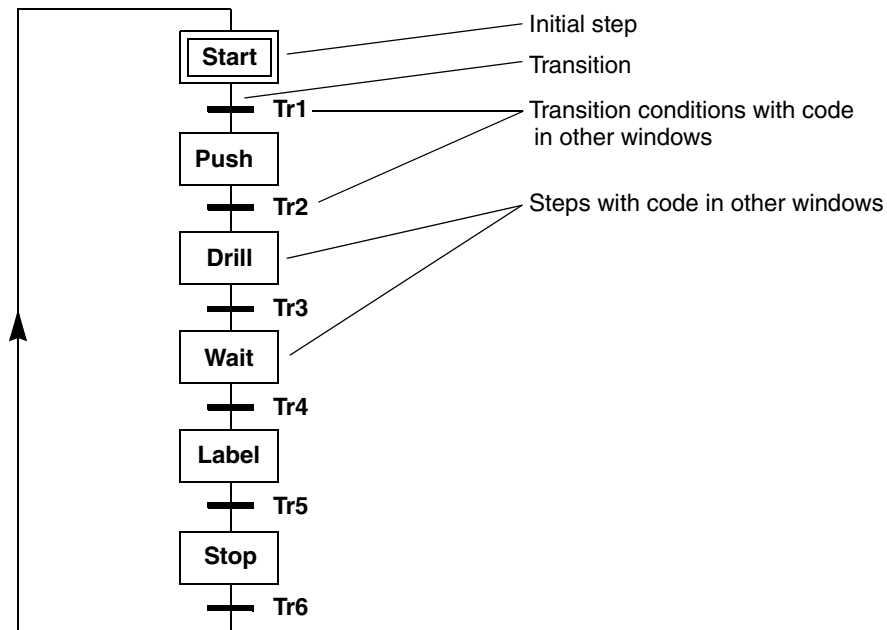
Figure 41. Example of an SFC program for an automatic drilling machine. Note the cyclic execution being enabled by the Tr6 transition condition

Steps and Transitions

All steps within an SFC must have unique names and may only appear once in each flow. Every step has an automatically defined Boolean *Step active flag* variable that is true while the corresponding step is active. The Step active flag is given the same name as the step plus the suffix X, e.g. Drill.X. It can be used within the current SFC to control the logical flow.

Two adjacent steps must always be separated by a transition condition which produces a Boolean result. A transition that always occurs can be expressed by the Boolean literal TRUE. The transition conditions may contain any kind or complexity of statements, variables and parameters, as long as the result can be expressed as a Boolean variable.

Action Descriptions

Steps in an SFC are used to describe the states of a controlled plant or machine. When the programmable controller executes an SFC program the state model only works as an internal memory representation of the control function. In order to get real-world actions each state has one or more *action descriptions* containing program code controlling the physical objects. Any of the four IEC languages can be used to describe the behavior of an action.

Action descriptions are normally placed in rectangular boxes that are attached to the step with a connection line. To avoid overloading the SFC with too much detailed information the boxes can be folded in or out. Most editors use a separate window or another code block for specifying the actions.
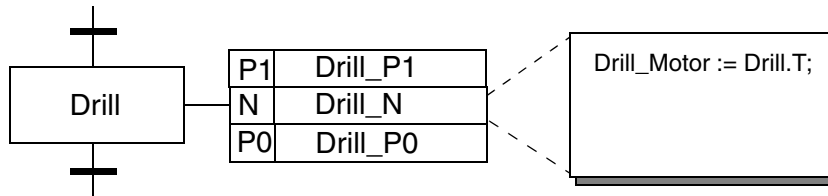
Figure 42. Example of a step with the associated actions folded out and one of them described in a separate editor window

Each action can have one or more action qualifiers that determine when and how the action is executed. Most editors support the following three action qualifiers.

- The N action qualifier (Non-stored) causes the action code to be executed continuously as long as the step is active.

- The P1 (Pulse rising edge) action qualifier causes the action code to be executed once when the step becomes active.

- The P0 (Pulse falling edge) action qualifier causes the action code to be executed once when the step becomes inactive.

To use one or more of the action qualifiers the programmer writes the code statements in the associated editor window. It is not necessary to use all three action qualifiers. Most sequences use the N action qualifier, but it is possible to leave all three qualifiers empty resulting in a step without any actions.

Sequence Selection and Simultaneous Sequences

In its simplest form, an SFC program consists of a series of steps in a closed-loop executed continuously. This type of system (see example in Figure 41) has only one main flow path.

In many systems there is a need for two or more branches in the sequence flow, often referred to as *sequence selection*. This is required in many batch process applications. In the example below with *divergent paths*, each branch starts and ends with a transition. When either of the transition conditions Tr2 or Tr3 becomes true, the corresponding branch is selected and execution continues along that path. Note that only one branch can be executed at a time. If more than one transition condition is true the left-most branch has the highest execution priority. When the last transition in the selected branch becomes true the flow converges back to the main flow.
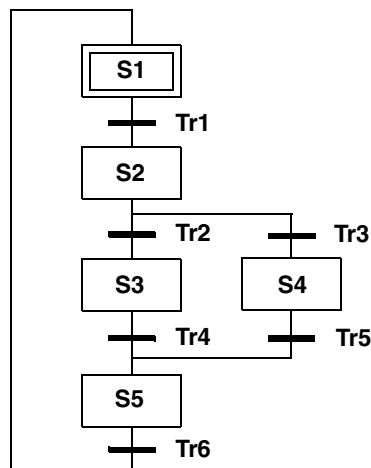
Figure 43. Example of a sequence selection with two branches

We have earlier seen how divergent paths can be used to execute alternative paths in sequences. An important characteristic of such parallel branches is however, that only one step in one of the branches may be active at any time.

However, in many batch process applications there is a need for *simultaneous sequence* structure with several branches. The main sequence is used for primary process control, while secondary parallel sequences are used to monitor that the process is running normally. Such parallel sequences can e.g. check that plant temperatures and pressures are within required limits, otherwise the control system may shut down the process.

In the example below, all three divergent branches start with a common transition condition. Execution then continues in parallel and independently along all three paths until convergence is reached. Both the divergent and the convergent flow in simultaneous sequences are drawn with a pair of lines to distinguish the construct from a sequence selection. The transition condition that succeeds the simultaneous sequence structure will not be tested until all the branches have finished execution, that is when the last step of each branch is active.
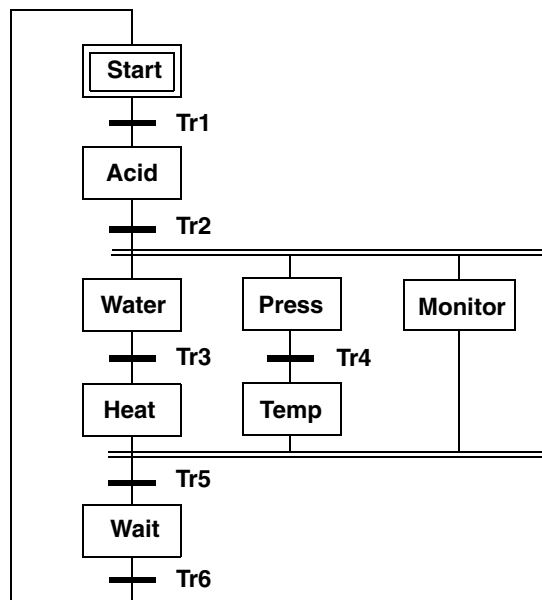
Figure 44. Example of a simultaneous sequence with three continuous branches

Subsequences

One of the main uses of SFC is as a tool for developing the *top down* design of the control function in a complex plant. Most processes can be described by a relatively small number of main states, each representing a subprocess with a number of *minor states*.

Some editors provide a method for dividing large SFC programs into a number of *subsequences*, each represented by a general symbol. A subsequence may in turn contain other subsequences which provides a powerful tool for structuring the overall control function into any number of hierarchical levels. This allows attention to be focused on either the overall behavior of the entire plant or on the detailed operation of the controlled process objects.

A subsequence usually contains sequence parts that perform a set of logically related actions. Steps and actions from different hierarchical levels are never visible at the same time. To study the inside of a subsequence the programmer has to step into the subsequence which changes the SFC view, so that only the contents of the selected subsequence are displayed.

SFC Viewer Recommendations

The SFC Viewer aspect in Process Portal is used to represent the Sequential Function Chart (SFC) programs created in Control Builder. Control Builder allows to combine sequence selection, simultaneous sequences and subsequences to create complex configuration of SFCs. However, SFC Viewer has certain restrictions in showing SFC configurations having subsequences, which is created in Control Builder.

The recommendations of SFC Viewer are:

- Do not use multiple levels of sequence selection or simultaneous sequences containing subsequences for showing in the SFC Viewer. Subsequences should be limited to the first level in a sequence selection or a simultaneous sequence. Nested subsequences are not allowed in a sequence selection or a simultaneous sequence. A jump is considered to be one level in nested subsequence.

- Do not use more than one subsequence in any branch of a sequence selection or simultaneous sequence. But, the steps and transitions before or after the subsequence can be used. Sequence selections or simultaneous sequences having subsequences can have a maximum of two branches.

- More than one step and one transition can be used in each subsequence. Subsequences should not contain more than two levels of sequence selections or simultaneous sequences.

- Sequences with a maximum of up to 8 steps and/or up to 16 transitions can be executed parallely.

- Use Boolean variables for display in the transition display of SFC Viewer. Keep analog supervision separated.

- Keep the variable name to be exposed in the SFC Viewer as the left most part in the structured variable used in the SFC Editor. This has to be notified in order to secure the essential part of the very long structured variable name to get exposed in the SFC Viewer. In SFC Viewer the tool tip function shows the complete structure variable name.

- Do not refer function block sub object parameter names directly in SFC Viewer transition conditions, instead connect a local variable to the sub object variable and use it. Example, if there is a sub object called "Motor1" with a parameter called "Running", one can not write "Motor1.Running" in the SFC, because that property does not exist on the SFC object. Instead a variable can be connected to "Motor1.Running", and used in SFC.

- The function block or control module in control builder with the SFC must be an aspect object in order to create the SFC Viewer. In order to show function blocks and control modules as aspect objects in PPA the option "Instantiate as aspect object" must be checked in control builder. The option is located in context menu for function block and control module tree items in the project explorer.

The Transition window (Function plan or List view) has a limitation on the number of variable entries it can immediately display when clicked. This limit is found to be 150 variable entries. For variable entries more than 150, the performance is negatively affected, and the transition window takes a longer time to display the function plan or list views.
For example, for a transition window showing 200 variable instances, it takes around 15 seconds to display the function plan or list view contents.

Advice on Good Programming Style

Names of steps, transitions and actions should be unique within each program organization unit (For example function block). It as also wise to use meaningful names whenever possible.

Try to keep all SFCs as small as possible and focused on the overall behavior. It is better to put detailed behavior in the action blocks or in other SFCs at a lower hierarchical level.

It is good practice to reduce the interaction between simultaneous sequences to a minimum.

Never allow step actions from different simultaneous sequences to change the same variables.

Avoid using constructs in which a divergent path branches out of a simultaneous sequence since this may lead to a sequence that never completes or behaves unpredictably.

Section 3 Programming in Practice

Introduction

This section contains examples and practical advice on:

- How to organize your code, see Organizing Code on page 93.

- How to use the code sorting function to optimize execution and how to solve code loops, see Code Sorting on page 121.

- How to optimize your code, see Code Optimization on page 135.

- How to tune your tasks to optimize execution, see Task Tuning on page 143.

Organizing Code

This subsection contains advice on how to implement the methods for organizing code, as well as more detailed information about data flow and execution designed to help you understand how to solve various programming problems:

- For advice on how to program using function blocks, see Programming with Function Blocks on page 94.

- For information on function block calls, see Function Block Calls on page 97.

- For information on function block execution, see Function Block Execution on page 98.

- For information on function block code sorting, see Function Block Code Sorting on page 100.

- For advice on how to create your own function block type, control module type and diagram type, see Self-Defined Types on page 110.

- For an example of how to use structured data types, see Structured Data Type Examples on page 116.

For a discussion of which method (control modules, programs or Function Designer) to use when organizing your code, see Code Organization on page 19 in Section 1, Design Issues.

Programming with Function Blocks

This subsection provides advice when programming with function blocks. For an introduction to function blocks, refer to the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Functions and Function Blocks

There are a great number of predefined functions and function block types available in the Control Builder standard libraries. Note that library functions and function blocks are available in all six programming languages. The main differences between functions and function blocks are described below.

Functions
- always return a (single) value at the time they are executed,
- can be used in expressions,
- do not retain their old values from one scan to the next,
- always give the same value when the input parameters have the same value,
- has a declaration and a definition,
- do not have a type-instance relationship
- cannot be customized.

Function Blocks
- have both input and output parameters,
- can provide several output values using local variables, external variables, parameters, and extensible parameters,
- retain their values, from the last call, when called again, and can give different output values even if the input values are the same,

- have to be used as function blocks of a function block type definition,
- support type-instance relationship
- can be customized.

Function Block Parameters

Function blocks have parameters, *In*, *Out*, *In_out,* and *by_ref*. *In* and *Out* parameters are passed by value, which means an *In* parameter makes a copy of the actual variable connected to the parameter, to a local representation inside the function block, and an *Out* parameter makes a copy of a local representation inside the function block to the actual variable outside the function block. *In_out* parameters are passed by reference, which means only a reference to the actual variable outside the function block is passed inside the function block, that is, no local representation of the parameter exists inside the function block. Performing operations on an *In_out* parameter inside a function block, thus means performing the operations directly on the actual variable connected to the function block, while operations on *In* and *Out* parameters act on local copies inside the function block.

By_ref is used for controlling the passed value. For *in* and *out* parameters the value is usually copied into the called instance at the invocation. But for non simple data types and strings it is time consuming. In that case, a reference to the data instance is passed in the function block call. This is achieved by setting the attribute of the parameter to *by_ref*.

> The *In_out* parameters are direct references and are not copied as *In* or *Out* parameters. *In_out* parameters can be compared with parameters of control modules, they are all direct references.
>
> For more information on passing data, see the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Some characteristics of the different parameter types are listed below.

- All parameter types (*In*, *Out* and *In_out*) occupy memory when a function block of the function block type is created. An *In_out* parameter always occupies 4 bytes, while an *In* or *Out* parameter occupies the same amount of memory as the corresponding parameter data type.

- When a function block is called, all connected parameters are copied to/from the internal representation inside the function block.

- An *In* or *Out* parameter can be left unconnected in a call, while an *In_out* parameter requires an actual parameter being connected in each call. An unconnected *In* or *Out* parameter requires no data copying.

- *In* and *Out* parameters can be supplied with initial values, but not *In_out* parameters. The initial value will be the value of unconnected *In* or *Out* parameters inside the function block.

- The extensible parameters (multi-parameters) have a general limit of 128 parameters in each function block. However, a few function blocks have a lower limit, described in the online help.

Conclusions

Using an *In_out* parameter instead of an *In* or *Out* parameter, will result in better execution time, and memory performance, *if* the data type size of the parameter is greater than 4 bytes (for example, for the string data type, and structured data type). For simple data types (for example *dint*, *real*, *bool*, *dword*) no improvement in performance is gained by using the *In_out* parameter type, and it is then better to select either the *In* or *Out* parameter type depending on how the parameter is to be used.

In and *Out* parameters can be assigned initial values. These can be used together with the possibility of leaving *In* and *Out* parameters unconnected. When writing the function block type, you can, for example add an *In* parameter that can be optionally connected when calling a function block of the function block type. The parameter can then have a suitable initial value, being the default value of the parameter that is used inside the function block, if the parameter is left unconnected. Note, however, that the parameter will not be re-initialized if it is unconnected in one call, but has been connected (or assigned a value in another way) earlier. The parameter of the function block will only be initialized once when the controller is started (at warm restart, or at cold restart) depending on the parameter attribute (*retain* or *coldretain*).

Function Block Calls

When a function block is called, the *In* parameters will be copied (and references to the *In_out* parameters) before the function block is executed, and the *Out* parameters will be copied after the function block has been executed. Since it is possible to omit *In* and *Out* parameters in a call, this is a way of increasing execution speed. For example, if an *In* parameter only changes its value occasionally, it can be omitted in the call and be replaced by a direct assign to the *In* parameter before the call, see example below.

```
If UpdateIndata then
   MyFB.In1 := Indata1;
   MyFB.In2 := Indata2;
end_if;

MyFB(InOut1 := InOutdata1,
   InOut2 := InOutdata2);

If ReadOutdata then
   Outdata1 := MyFB.Out1;
   Outdata2 := MyFB.Out2;
end_if;
```

It is possible to omit *In* and *Out* parameters in a function block call. The *In* parameters of MyFB will only be changed occasionally in this example.

The method of directly accessing *In* and *Out* parameters by dot notation (for example *MyFB.In1* or *MyFB.Out*1) in the code can be used to save a local variable when connecting two function blocks, see below.

```
MyFirstFB(In1 := Indata1,
Out1 => LocalVar);

MySecondFB(In1 := LocalVar,
Out => Outdata1);
The code can be written without using the intermediate
variable LocalVar:

MyFirstFB (In1 := Indata1);
```

```
MySecondFB (In1 := MyFirstFB.Out1,
Out => Outdata1);
```

Function Block Execution

When a function block is executed, the code blocks are executed from left to right. When the last code block has been executed, function block execution is complete, and the execution is returned to the calling unit.

Since a controller executing an application, often has its execution time divided between several tasks, it is important to remember that task switching may occur. Task switching occurs when a task with higher priority is ready to execute while a lower priority task is executing. However, the low priority task can not be interrupted anywhere in the code, it must reach a scheduling point. It is therefore important to know where scheduling points occur when programming, for example, a function block type, since function block execution only can be interrupted by other tasks at these points.

The following are defined as scheduling points during the execution of an application:

- at the beginning of each code block (in function blocks, control modules and programs),

- at backward jumps in the code (that is, loops such as For, While and Repeat statements).

The main issue concerning task switching is data consistency. It must be assured that the data required in a function block, do not change during function block execution. Such a problem may arise if *In_out* parameters are used and the data referred to the *In_out* parameter are also manipulated by another task. This problem does not occur for *In* and *Out* parameters, since they make use of local copies within the function block, which remain unchanged during the execution of the whole function block.

It is important to allow scheduling points, bearing data consistency in mind, at a sufficient number of points in the application code, otherwise tasks with higher priority will be delayed by a low priority task leading to what is called *task latency*.

Another related problem concerns the ability of the controller to handle a power failure. If a power failure occurs, the controller must go into a safe state and maintain this safe state until the power supply is re-established. To do this, the controller must thus be able to reach a scheduling point within a certain limited time (a few milliseconds).

Conclusions

It is important to bear in mind that scheduling points at inappropriate places in the code can give rise to data inconsistency (especially when using *In_out* parameters).

A sufficient number of scheduling points should be included in the code so as not to cause task latency.

The frequency of scheduling points should be such that one can be reached within a few milliseconds to allow the controller to reach a safe state in the event of a power failure

In practise, this means that you should avoid writing long code blocks and avoid data access from several code blocks which may involve changes in data.

You can read more about *tasks* in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Function Blocks in ST and IL

In the Structured Text (ST) and Instruction List (IL) languages, function blocks are called explicitly and parameter connections are expressed in the call. The same function block can be called several times in the same code block and it will then be executed several times during the same scan.

Function Blocks in FBD, LD and FD

In the Function Block Diagram (FBD), Ladder Diagram (LD), and Function Diagram (FD) languages, function blocks are called implicitly from the graphical block representation. Parameter connections are expressed by graphical connections. The execution order of the function blocks in LD is from left to right, while for FBD, it is from top to bottom. In FBD, a function block can only have one block representation and can thus only be called once in an FBD code block.

In FD, the execution order of a function block is based on its Data Flow Order number. Each function block can be used once in a diagram and can be placed anywhere in the FD code block.

Enable and Disable Inputs

In the FBD, LD, and FD languages it is possible to Enable/Disable the execution of a function block with the *EN* input of the block (in LD this input is always connected to the rung, but in FBD and FD, it is an optional input that can be connected as any other input).

When a function block is disabled (that is, *EN* = False), parameter copying cannot be performed and no internal code in the function block can be executed. This means that you can gain speed by using the *EN* input, instead of having an internal Enable/Disable If-statement inside the function block controlled by a parameter.

The *EN* functionality is only available in FBD and LD, but it can be compared to a similar construction in ST, see below.

```
If EN then
  MyFB(In1 := Indata1,
  Out1 => Outdata2);
end_if;
```

The *EN* functionality of a function block is similar to the function of the ST code above.

Function Block Code Sorting

As already mentioned, the code for the control modules is sorted for optimal data flow during execution, see Correcting Sorting Problems on page 133. However, the code *inside* the function block is *not* sorted. The function block as a unit is sorted according to data flow. At this stage, it is important to note that:

- the read/write status of parameters in the function block plays an important role in the code sorting routine,

- for any function blocks called from control modules, only the parameter interface (*IN*, *OUT*, or *IN_OUT*) affects code block sorting.

That is, no analysis is made of how the function blocks actually use parameters. In addition, function block references to external variables do not affect the execution order of control module code blocks.

The table below shows the parameters that affect the code block sorting.

Table 5. Parameters that affect code block sorting

Parameter	Status in the function block
IN	read
OUT	write
IN_OUT	read/write Compilation does not involve analyzing the code inside the function block to more thoroughly determine whether the parameter is of read or write status.

Hence, it is important to note that *IN_OUT* parameters may result in a code block loop error, since the analysis cannot determine if the parameter has write or read status. For further information, see Correcting Sorting Problems on page 133.

Control Modules in Function Blocks

Function blocks may contain control modules. For a discussion of when this is suitable, see Using Programs on page 21.

⚠ When control modules are created in function blocks, an explicit call to the ExecuteControlModules system function must be made, to execute all the control modules in such function blocks. The function can be called at any point in the code in any of the function block code blocks.

Control Module Groups

All control modules in the function block form a group of control modules. All direct sub control modules in the function block will be added to the group. Also all sub control modules to the direct sub control modules will be part of this group.

The rule is that the call cannot be done if the function block has an empty local group of control modules.

The group is considered empty if all control modules are either of the following.

- asynchronous (a sub control module has a task connection other than its parent's task connection)

- without code

- only contain start code

Compile errors are generated if the rule does not apply.

It is possible to make several calls to ExecuteControlModules from the function block code. At each call, all control modules in the group are executed.

Static Function Block in_out Parameter

If a control module parameter is connected to a function block in_out parameter, the function block in_out parameter must be static in all calls. This means that if the function block is called more than once (in a scan), it must have the same actual variable connected to the in_out parameter in all calls. It also means that at least one call to the function block must be done.

If the function block in_out parameter is connected to a parent function block in_out parameter, this parent in_out parameter must also be static in all calls.

Examples of Control Modules in Function Blocks

The following two examples of control modules located in function blocks show how co-sorted groups are formed.
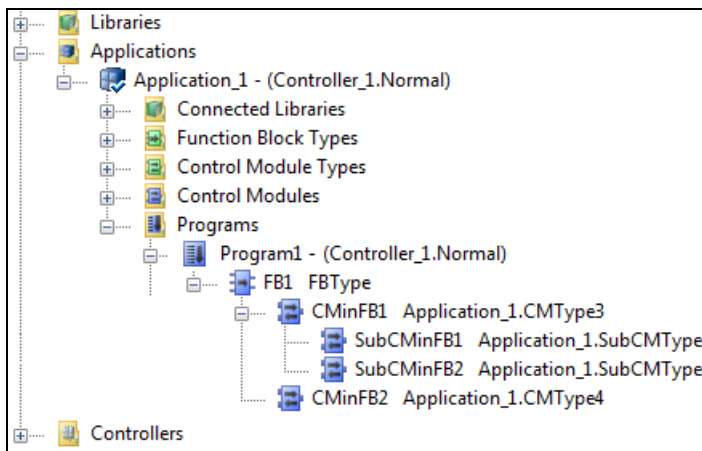
Figure 45. Control modules in function blocks example

In the figure above, in Program1, the FB1 function block contains a group of control modules. This local group contains the CMinFB1 control module with its SubCMinFB1 and SubCMinFB2 sub control modules and the CMinFB2 control module. All execute when the ExecuteControlModules function is called from the FB1 function block.

In this group of control modules, the execution order of all code blocks is determined by the code sorting.

As a comparison in the figure above, CM1, SubCM1, SubCM2 and CM2 form a co-sorted group of control module code blocks executing in the Slow task.

Figure 46. Example of task connection to a sub control module

It is also possible to make a task connection to a sub control module other than its parent's task connection. In Figure 46 above, CMinFB2 in FB1 has such a task connection and is therefore removed from the local group in FB1. The CMinFB2 control module is instead added to the group of co-sorted control modules in the Slow task.

Continuous and Event-Driven Execution of Function Blocks
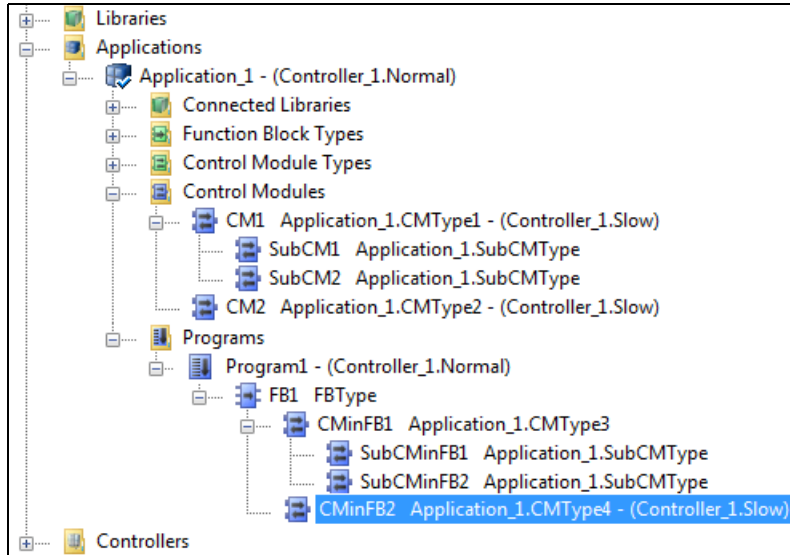
The following subsection explains the differences between continuous and event-driven execution of function blocks and how to use function blocks when using one or the other method.

Function block types with parameters listed in Table 6 are intended for continuous execution. They should be executed once per scan.

Table 7 shows parameters for event-driven execution.

Do not call function blocks of these types in an If or Case statement, or in an SFC step, since this might cause errors in the state machine that is included in the function block. Use the *Enable* and *Request* parameters to control how they are executed.

Continuous Execution

The table below lists the general parameters used in continuously executed function blocks. Note that all parameters do not have to be connected.

If an error and a warning take place at the same time, the error has precedence over the warning and *Status* is set to the error code. *Error* and *Warning* are only activated upon the call of the function block.

The duration of the *Error* and *Warning* parameters is a pulse during one scan only. Therefore latching in the application is required to detect these signals.

Table 6. Parameters used for continuous execution of a function block

Parameter Name	Data Type	Direction	Description
Enable	bool	In	Activates/deactivates continuous functionality
Valid	bool	Out	Indicates that there is no error and that the function is active. Warning status does not affect Valid
Enabled	bool	Out	Indicates that the function is active. Is not affected by error status or warnings status.

Table 6. Parameters used for continuous execution of a function block (Continued)

Parameter Name	Data Type	Direction	Description
Error	bool	Out	Indicates an error (Status < 0)
Warning	bool	Out	Indicates a warning (Status > 1)
Status	dint	Out	Indicates Status code

A function block can be activated and deactivated using the *Enable* parameter. Figure 47 shows an example of the parameter interaction in this case.

If a function block is invoked exactly once every scan, the distance in time between two invocations is the interval time of the task.
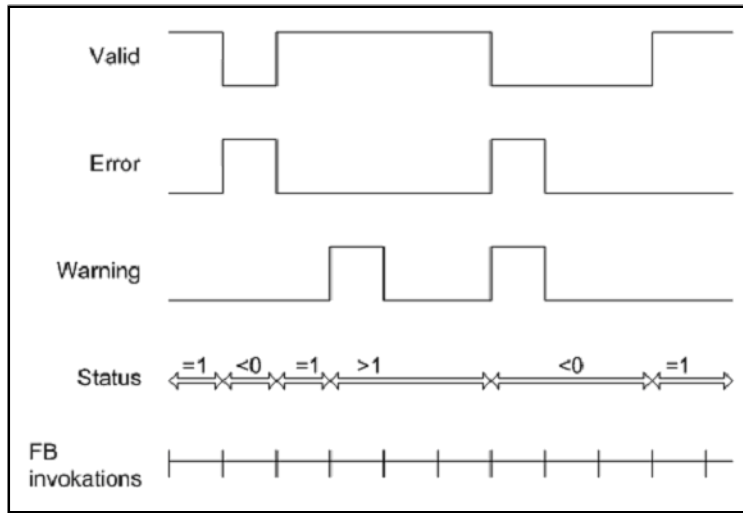
Figure 47. Enable always true

Some function blocks have a direct response on an activation/deactivation. The response is basically, the same as that to "Enable Always True", with the exception that at deactivation, the parameter *Status* is reset to 1. The figure shows an example of the parameter interaction in this case.

This is for functions that can be started and stopped directly. The *Enable* parameter is used to activate/deactivate the function block.
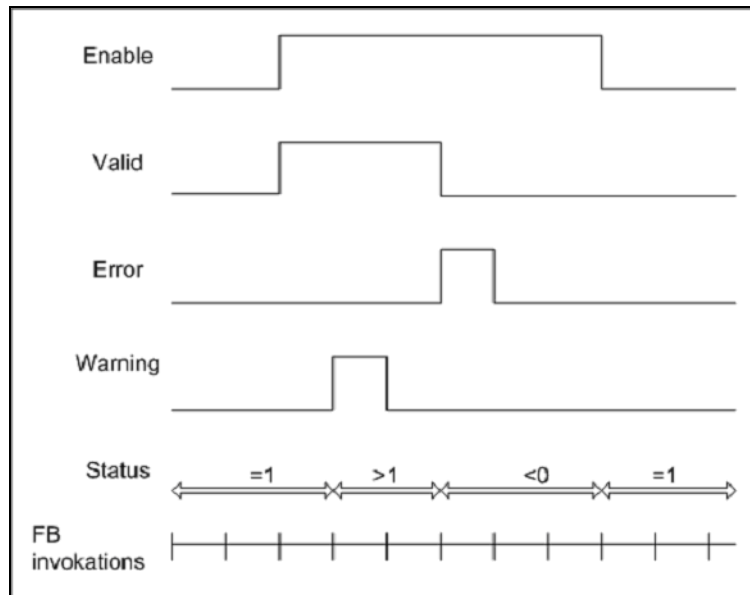
Figure 48. Direct response on activation/deactivation

Some function blocks have a delayed response on activation/deactivation. This means that the function block is started/stopped with a delay when the Enable parameter is activated/deactivated. In order to "read" whether or not the function block has been activated a parameter called *Enabled* is used. The figure shows an example of the parameter interaction in this case. Note that the warning is active during two calls before it disappears.

The function is started/stopped with delay when the *Enable* parameter is activated/deactivated. To be able to read this an *Enabled* parameter can be used.
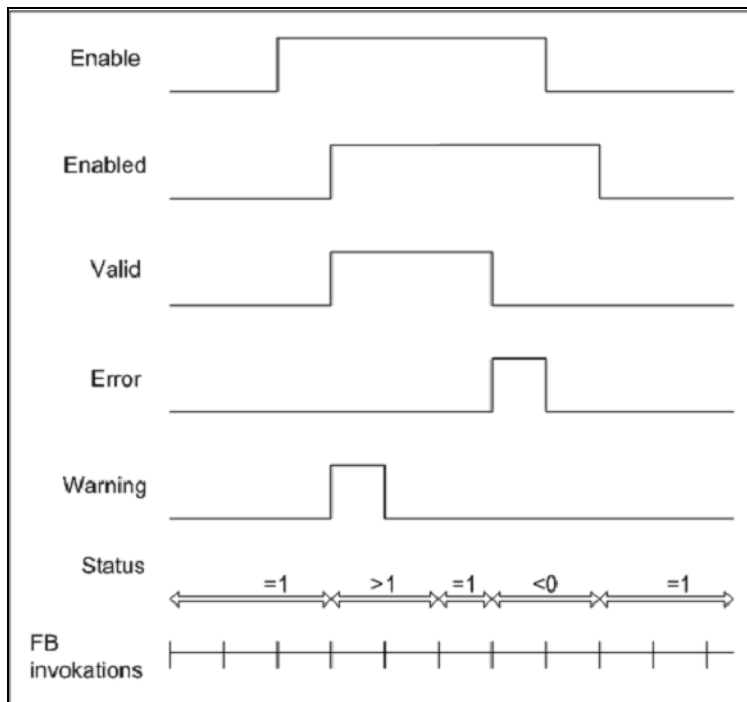
Figure 49. Delayed response on activation/deactivation

Event-Executed Function Block Types (Standard Libraries)

The hand-shaking signals that are required when the functionality is (event-driven) asynchronous and performed by commands, for example, values that are read from another controller, are described by the parameters below.

The duration of the *Error*, *Warning* and *Done* or *Ndr* parameters is a pulse during one scan only. Therefore latching in the application is required to detect these signals.

Table 7. Parameters used for event-driven execution of a function block

Parameter Name	Data Type	Direction	Description
Req or *Request*	bool	In	Activates the function block on a positive edge. Must be reset by the user. When the Req parameter is set for the first time, it is advisable to wait until the execution of the operation is completed. That is, wait for the result derived via the Done (Ndr) parameter or alternatively the Error parameter before triggering again. The Status parameter can be used for this validation instead of the above mentioned parameters. A "pending operation" is indicated by setting the Status parameter to "0". An important example of this usage is when communicating in a multi drop configuration where it is important to achieve a distributed access to the slaves. That is, do not ever trigger them in a stochastic way without using the handshaking.
Done or *Ndr* (New Data Received)	bool	Out	Indicates that a command has been executed but there were errors.
Error	bool	Out	Indicates that a command has been executed but there were errors.
Status	dint	Out	Status code
Warning	bool	Out	Optional parameter. Indicates that the command has been executed and that there were no errors, but a warning.

Examples of Function Blocks with Event-Driven Functionality
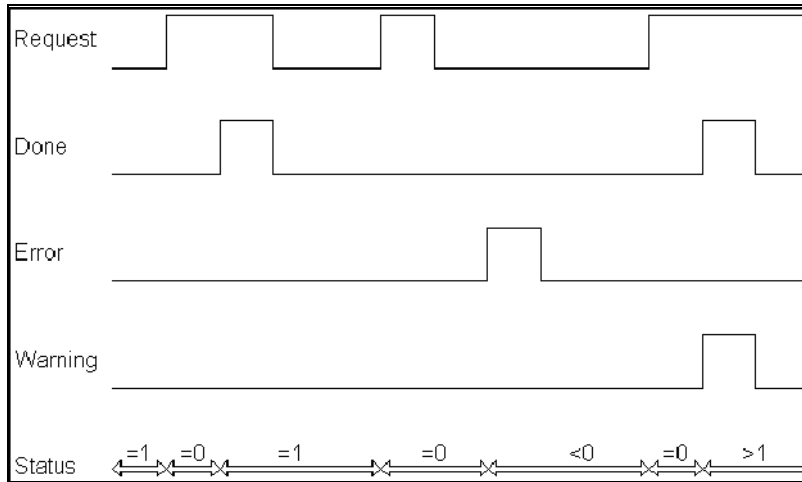
Figure 50. Examples of event-driven execution

Figure 50 shows examples of event-driven execution. Parameters are used according to Table 7. The output signal *Done* only lasts for one call, whereas *Status* is stable until next change. Figure 50 shows an example of the parameter interaction in this case.

Self-Defined Types

This section gives some good advice on increasing the performance of your own function block types, control module types, and diagram types. The advice concerns solutions where an efficient application code is the primary goal. Other advice would be given if readability of the application code was of more interest.

It is important to apply foresight when creating your own function block, control module or diagram types. The choice of suitable types saves memory and reduces execution time, which means increased control system performance. Saving 1 kilobyte of memory in a type may mean that megabytes of memory will be saved in the whole application.

> Examples showing how to create your own function block types, control module types, and diagram types are given in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

It is recommended that you create your own libraries when working on a large project, as this will give the project a better structure. Another major advantage of creating your own libraries is that it is possible to re-use data types, function block types, control module types, and diagram types in other projects. You can create online help for own your libraries, see the *System 800xA, Control, AC 800M, Binary and Analog Handling (3BSE035981)*.

Each library has to be connected to the application where objects from the library are used. See the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Function Blocks and Control Modules in General

Basically, a function block or control module is a POU which consists of parameters, variables and code. Both the function block and the control module are based on the object[1] type design, which means that they are described by their types. From the object types, you create instances that behave exactly as the type.

Each instance (object) has its own memory representation of variables and parameters, and when an object is called, it is the object that is called, not the underlying type. The executable code, however, is shared between all objects and belongs to the object type. To reduce memory usage it is therefore a good idea to try to have object types that can be used as a source for several objects. If different behavior is required for an object, this can be expressed by type parameters, see Flexible Types on page 115.

When you create your own, self-defined, types, consider the following:

- **Language Selection**

  In order to get optimal performance when programming IEC 61131-3 code, do not choose a programming language that generates intermediate variables (for example, FBD).

---

1. In this section we write function block or control module as object.

- **Code Arrangement**

  Considering that types are instantiated, it may be better to accept a less well-arranged code, in order to achieve increased type performance. Comments in source code (comments are not included in the compiled code), may compensate for the reduced clarity of the code.

- **Code Tab Start**

  Note that code tabs starting with *Start_* in control modules, are only executed once, after each warm restart, before all other code. This is a feature that reduces code size and memory consumption.

- **Overhead Time**

  Each code tab requires an extra overhead time in control modules. No application execution will take place during this overhead time. The overhead time will be 4–5 microseconds in a PM860 CPU[1], so use a minimum of code tabs in a control module. For example, removing five code tabs in a function block type with 1000 function blocks based on that type, will save 25 ms of execution time.

- **Simple Function Blocks**

  Avoid the use of simple function blocks such as *SR* and *R_Trig*. Write the equivalent code instead. The overhead time for simple function blocks will be long due to parameter copying. For the same reason, avoid creating simple types of your own.

  Use timer functions instead of timer function blocks in your type. This will save memory and reduce the execution time.

- **Project Constants**

  Do not use variables intended to be literals in a type, even if they have the attribute *constant*. Use project constants instead.

  Project constants are easy to change in a single place, the same value is always used in the whole control project, and they facilitate easier use of logical names, instead of values. Objects access project constants by pointers. They can be located in your library, thus they are easy to find and modify.

---

1. Code tabs used in SIL application has much greater influence on the overhead time compared to non SIL applications.

i If a project constant connected to a retain parameter (or variable) is changed online, then the change does not effect on existing instances until a cold restart is performed.

i Project constants may be defined in two locations: in the project or in a library. Project constants to be used in types should be defined in the library where the type is defined.

i More information about project constants can be found in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

- **Unused Variables and Parameters**

  Variables and parameters require memory space, irrespective of whether they are used or not. When the type has been created, clean up, and delete all variables and parameters that are not used.

- **HSI Communication**

  You should take into consideration any possible communication with HSI (operator stations, etc.), when creating a type. Consider which variables are required for communication, and which name convention is to be used. Variables that do not have to be visible in the HSI should have the attribute Hidden.

  Remember that a type requires two parameters; *Name* and *Description*, if the Name Uploader aspect in the Plant Explorer is to function properly.

- **Alarm and Event Objects**

  Generally speaking, you only have to use a single SimpleEventDetector function block in a type. The function of SimpleEventDetector is like that of a printer, logging an event according to its parameter values when the condition is changed. Using a single event detector and then changing its parameter values for different purposes will save a considerable amount of memory.

Control Module Interaction Windows in Function Blocks

The concept of placing an interaction window inside a control module can also be extended to function blocks. Any function block type can be equipped with, for example, an interaction window for testing or maintenance purposes. After the interaction window has been designed, right-click the function block type in question and choose **New Control Module…**. Select the library and the name of the control module type.

There can be only one control module in the function block, and during online mode, the control module is displayed by right-clicking the function block and selecting **Interaction Window**.

Diagram Types

A diagram type is a POU that consists of parameters, variables, and code that supports FD language. It is based on the object[1] type design, which means that they are described by their types. From the object types, you create instances that behave exactly as the type.

Each instance (object) has its own memory representation of variables and parameters, and when an object is called, it is the object that is called, not the underlying type. The executable code, however, is shared between all objects and belongs to the object type. To reduce memory usage, it is therefore a good idea to try to have object types that can be used as a source for several objects. If different behavior is required for an object, this can be expressed by type parameters, see Flexible Types on page 115.

When you create your own, self-defined, diagram types, consider the following:

*   **Language Selection**

    The FD (Function Diagram) language is a mandatory programming language for diagram types. Additionally, two IEC 61131-3 programming languages, ST and SFC, can be used optionally in a diagram type.

---

1.   In this section, we write function block, control module, or diagram as object.

- **Code Arrangement in FD**

    The FD code pane supports graphical connections between the inserted objects. It is recommended to divide the code in many pages in the FD code pane, for better understanding and analysis. Each page and each object can have optional description that is displayed together with the objects.

    Each object has a unique Data Flow Order number that is assigned automatically and displayed, based on the position of the object in the page. This number is assigned from left to right when new objects are added. The order of execution follows the Data Flow Order.
    However, if the position of an object is changed after its graphical connections are made, the connections are analyzed and the source will have the lowest Data Flow Order number.

    Therefore, it is recommended to place the objects that need to be executed first on the extreme left of the FD code pane.

    If the code becomes too complex to have graphical connections, the optional ST and SFC code blocks can be used, which are either invoked from the FD code block or sorted separately.

Flexible Types

A flexible (adaptable) function block, control module type or diagram type can be used for creating several objects (that is, instances of a type), without having to make too many variants of them. This can be done in two ways, you can either set specific initial values (Init_Val) on an object in Plant Explorer (see *System 800xA, Control, AC 800M, Configuration, 3BSE035980**), or create parameters (for the flexible type) for the control of certain behavior of the type. The parameters should be set up (initiated) during the application start procedure.

To distinguish such parameters from other parameters, it is recommended that the phrase *Init* be included at the end of the parameter name. You can see an example of this by examining the library control module *PidCC*, where the parameters *SpExternalInit, SpManValueInit*, etc., have the suffix *Init*.

- **Programming Example with Flexible Types:**

   Suppose we have created a control module type named *MyPIDLoop*. It consists of an *AnalogInCC*, a *PidCC,* and an *AnalogOutCC*. The purpose of the type is to generate a general PID loop, which can be used for several purposes (to regulate temperature, level, pressure, etc.).

   Most parameters of the included objects must be connected to parameters of the new type, in order to make the type general.

   We also want to be able to set the gain for each PID object during the engineering phase (not during runtime). We must then declare a parameter called *GainInit* [real]. The following piece of code in the new type sets the PID gain for each object after the first download:

```
(* The variable 'init' is of type bool, has the initial value
false, and has the attribute ColdRetain set *)
If Not Init Then
  Init := True;
  PidCC_InteractionPar.Main.Gain := GainInit;
end_if;
```

Structured Data Type Examples

You can connect application variables directly to the I/O, which is the easiest method. The drawback of this strategy is that it might not be possible to read the I/O signals from other parts of the program, and also not from other applications, in other controllers. It will also be more difficult to identify signals belonging to a certain process object.

It is possible to group your I/O signals in structured data types, or use a single structured I/O variable for communication between the application and the controller.

This subsection contains two examples of how to use structured data types to create flexible automation solutions:

- The Valve Configuration Example on page 117 shows how to set up I/O communication using structured data types when configuring a valve.

- The Structured Data Types in Structured Data Types – an Example on page 119 shows how to create a structured data type within a structured data type.

Valve Configuration Example

This example intends to show how to use structured data types when configuring a valve. To start with, the Table 8 asks some fundamental questions that you might ask yourself before programming.

Table 8. Questions to ask before deciding on what data types to use

Question	Answer
What type of valve object do I need and what additional functions/modules do I need to achieve this?	A control module with activate and two feedbacks.
How shall the valve be configured?	Both in online and offline mode, that is, from an interaction window (during engineering) and/or from a faceplate (Operators workplace). Also during offline: in programming mode, set a value on a connection that sets a default (opened/closed) value for the valve in the editor.
What kind of interfaces do we need?	Four kinds of interfaces (HSI, IO, Configuration and Application).
How shall the object operate, that is, shall it operate independently, or be an integrated part of a larger object type, for example a tank line?	Our valve must be able to operate both independently, as an object type, and as an object.
What kind of property permissions do we need?	HSI and Configure Variables are the only ones that need to be exposed via the OPC server. Thus, these variables need property permission settings and, eventually, security definition settings as aspect objects.
Does the valve need alarm handling, and if so, which object level shall contain the alarm owner?	In our case we will make the valve template the alarm owner (highest level).

In Figure 51, one data type is used to send data to the application (POType) and another data type is used to send data to the IO (IOType). Read steps 1 – 5 below Figure 51.
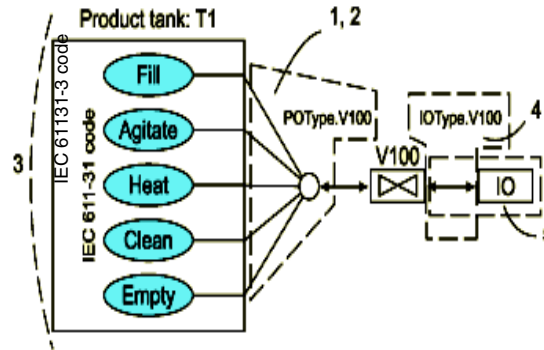

Figure 51. Using structured Data Types to communicate between IO and the application

1. The Application code (IEC 61131-3) is connected to the valve module via the PO Data Type, forming a star connection.

2. The software creates an automatic OR function.

3. Each individual function (Fill, Clean, Empty etc) that activates the valve object can be written as if it was the only function using the valve. This makes the design easier, and improves the re-usability of the software.

4. The IO data type is the connection between the valve and the IO module.

5. If the type of IO connection should change from, for example, local IO to fieldbus, you only need to change the object. You do not have to make any changes in your application module (see item 3), since it has already been tested and validated!

When we know what kind of object we are going to build, and how it should be configured, we need to decide on what interfaces the valve needs. We could be content with a single interface for all IO, HSI, configuration etc., but it would not be a very good design.

Earlier on, we identified at least four separate interfaces (HSI, IO, Configuration and Application). These interfaces can in turn be divided or translated into data types.

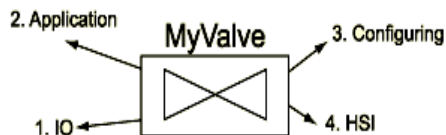MyValve can be put in the center of a design map, surrounded by interfaces that are linked to the valve.

Figure 52. Interfaces linked to a valve object

The four interfaces IO, Application, Configuration and HSI can then each use a structured data type to communicate with HSI, IO etc.

Structured Data Types in Structured Data Types – an Example

This example shows how to create a structured data type inside another structured data type. This makes it possible to group signals according to which part of the process they belong to.

The example starts with a one-way valve for acid liquids. The valve has a total of four signals: *OrderOpen*, *OrderClose*, *AnswerOpen*, and *AnswerClosed*.

The valve is programmed as a type in a library, and the name of the valve type is ValveOnewayAcidType. One of the parameters of this valve is the I/O parameter of structured data type, *IO_ValveOnewayAcidType*. It consists of four components.

	Name	Data Type	Attributes	Initial Value	ISP Value	Description	
1	OrderOpen	BoolIO					
2	OrderClose	BoolIO					
3	AnswerOpen	BoolIO					
4	AnswerClosed	BoolIO					

Components

Row 4, Col 2

Figure 53. A structured data type

The components are of the predefined structured data type *BoolIO*. Further, each component of the IO_ValveOnewayAcidType will be connected to an I/O channel of the digital I/O modules.

Another valve in our process is a single actuator, one-way valve for non-acid liquids. It is programmed as ValveOnewayType and has an I/O parameter of *IO_ValveOnewayType* with the following components.

	Name	Data Type	Attributes	Initial Value	ISP Value	Description	
1	OrderOpen	BoolIO					
2	AnswerOpen	BoolIO					

Components   Row 2, Col 2

Figure 54. Programmed as ValveOnewayType

There is a process cell called "Mixing", which mixes acid and water, where there are two valves of ValveOnewayAcidType type, and one of the type ValveOnewayType. The I/O signals for these valves are collected in one structured data type IO_MixingType.

	Name	Data Type	Attributes	Initial Value	ISP Value	Description	
1	InletAcid	IO_ValveOneway					
2	InletWater	IO_ValveOneway					
3	OutletMix	IO_ValveOneway					

Components   Row 3, Col 2

Figure 55. I/O signals collected in one structured data type, (IO_MixingType)

We have now grouped the signals from and to the process (from the point of view of the control system) into different parts of the process. The degree to which signals are grouped is up to you. The I/O data types should preferably be placed as variables or global variables in the application editor, so that they can be read and written from the application.

Code Sorting

For control modules, the compiler analyzes each code block separately, with respect to which variables are read and written by each block. ST, IL, FBD, and LD – SFC are treated somewhat differently, see remark below. The compiler then determines the optimal execution order for the code block. A block that assigns a value to a variable has to be executed before the block that reads the variable value.
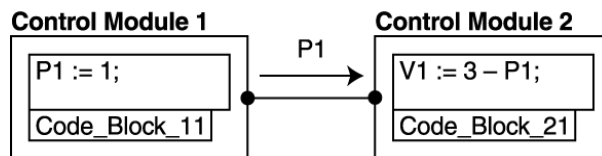
Control Module 1
```
P1 := 1;
Code_Block_11
```
P1 →

Control Module 2
```
V1 := 3 – P1;
Code_Block_21
```

Figure 56. The code block in control module 1 must be executed before the code block in control module 2

The technique for ordering the blocks is called *code sorting,* and means that the optimal execution order will be based on *data flow,* instead of the *program flow* (as is the case for function blocks). Code sorting guarantees that the code will be executed in the correct order. Code sorting reduces time delays and results in a new control module automatically being correctly placed in the process.

> If a function block is called from a control module, only the parameter interface (*In*, *Out*, or *In_out*) affects the code block sorting. That is, no analysis is carried out of the actual use of parameters within the function block. In addition, function block references to external variables do not affect the execution order of control module code blocks.

> Code sorting has no effect on control modules connected to different tasks.
>
> Within an SFC code block, only the *N* action parts (not *P0* or *P1* actions) are sorted.

> The execution order of function blocks follows the program flow, as implemented by the programmer according to IEC 61131-3.

Sorting of Diagrams

The FD language allows function block instances and control module instances to be used in the same diagram even though they have very different execution models. Function block instances are explicitly invoked whereas control module instances are implicitly invoked by the system as a result of their declaration.

The code block sorting determines the order in which the code blocks are executed in control module instances. The two mechanisms working together on function diagrams are:

*   Grouping of function block instances, functions calls, and code blocks calls into invocation groups.

*   Code block sorting performed on function diagrams. A function diagram constitutes a sorting group which covers both the control module instances and invocation groups in the function diagram.

The initial order presented to the sorting operation is based on the data flow order specified on control modules in the diagram. Only data dependencies between the control modules can alter this order.

If the diagram contains ST or SFC code blocks without invocations in the FD code block, these code blocks do not have a data flow order. Instead, they are sorted with the code blocks of invoked control modules.

Sorting of Diagram Type Instances

Diagram type instances executes at the top level context of a single diagram. Since a diagram type instance can hold code blocks from other module types, its behavior is similar to that of a control module, but has special characteristics.

Figure 57 shows a representation of a diagram with two control modules (CM1, CM2) and a Diagram Type instance (DT1). Inside DT1, there is a Function Block (FB), a Control Module (CM3) and a structure text code block (ST). 'A' and 'B' are code blocks.
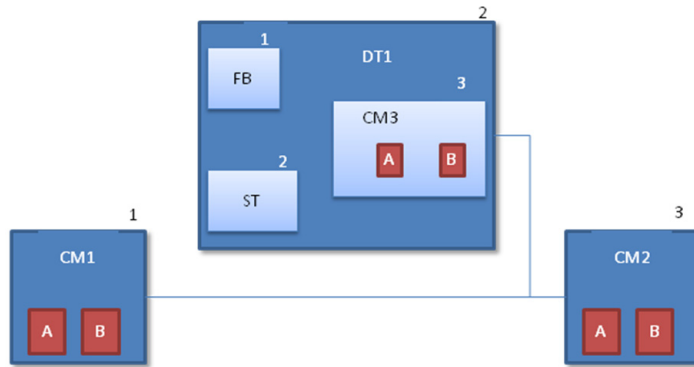
Figure 57. Diagram instance with other module types

When sorting code blocks (and when running with the same task), the DT1 boundary is ignored. Instead, all code blocks both within control modules and other types are sorted together with the code blocks outside DT1. The control module code blocks are sorted first, and then, the data flow order (which is already set) determines the sorting order of the rest of the code blocks.

If the data flow order of a code block is smaller than the sort order, it is inserted before. For example, in Figure 58, the sort order of code blocks can be 1A 2 3 4A 5A 5B 4B 1B.
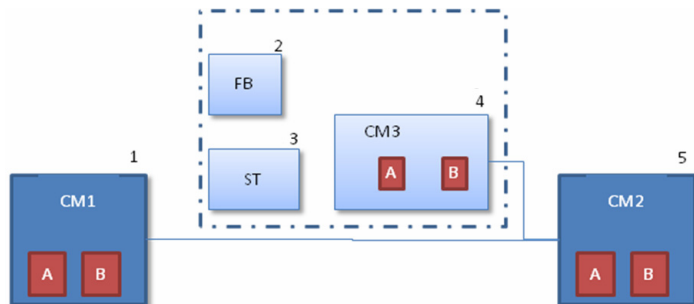

Figure 58. Sorting code block in diagram type instance

It is also possible to have code blocks in diagrams without specifying the data flow order. The code block will then be sorted like code blocks in a Control Module.

Code Loops

If more than one control module code block uses the same variable for both reading and writing, the compiler gives a warning message that a code loop has been found, which means that the execution order cannot be resolved:
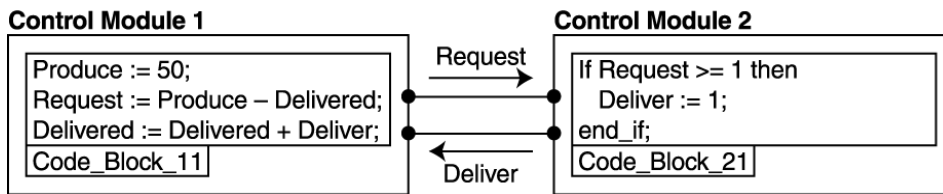

Figure 59. Control module 2 reads Request and writes Deliver, and control module 1 reads Deliver and writes Request. This execution order cannot be resolved

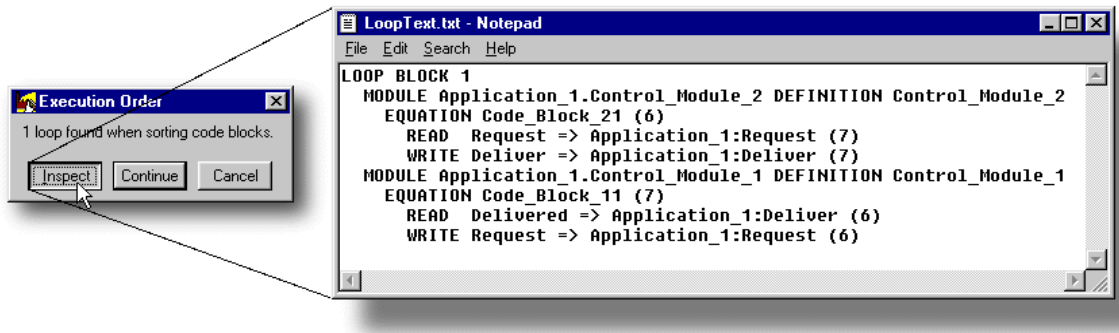This case yields the following error information:

Figure 60. An error message is generated, indicating a code loop problem

Code loops can generate an error during compilation and interrupt the download. Set the compiler switch **Loops in Control Modules** to **Warning** to bypass the interruption for download.

However, it still might lead to unexpected execution behavior. It is recommended that you solve code loop problems when they occur.

To bypass interruption for download

1. Right-click project and select **Settings > Compiler switches** from the context menu.

2. Select **Loops in Control Modules** in the Switch list.

3. Change **Error** to **Warning** in the Global drop-down menu.

4. **OK**.

In the example (Figure 59), the *Request* value determines the *Deliver* value, which, in turn, determines the *Request* value. This condition is shown in the automatically generated text file, where the figures within parentheses refer to the code block each parameter depends on. Provided that circular dependence actually exists (and is not merely the result of a programming error), the problem can be solved by assigning a State qualifier to the *Delivered* variable and introducing a new code block in control module 1, Code_Block_12:

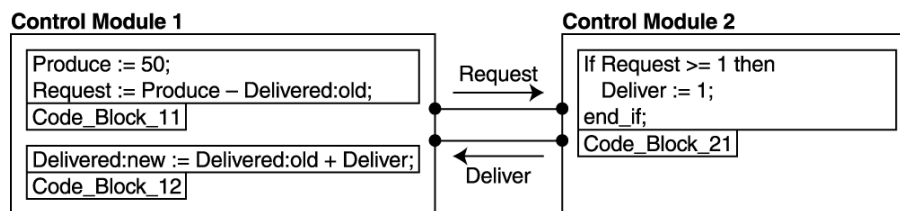
Figure 61. How to eliminate code loop dependencies

The code loop dependency has now been eliminated; *Delivered:old* stores the value from the previous scan and *Delivered:new* contains the updated value from the current scan. Hence, the execution order becomes code blocks 11 – 21 – 12. This approach is particularly valuable for complex applications, which are difficult to monitor manually.

Variable State

State can only be specified for local variables of types *bool*, *int*, *uint*, *dint*, and *real*. If, for some reason, you wish to override sorting and avoid the *State* implications, you can assign the *NoSort* attribute to the variable.

NoSort Attribute

Incorrectly used, the *NoSort* attribute may cause execution errors and application failure. *NoSort* should typically be used only when the code block connections themselves unambiguously determine the execution order.

Use *NoSort* only if you know the data flow characteristics in detail.

Interpret and Correct Code Loop Errors

Code sorting means that Control Builder sorts the code blocks in control modules in an execution order based on optimal data flow, determined by the system. Code blocks/tabs are sorted so that variables receive a value (Read in) before it is used by another variable (Write out). However, if the system is unable to accomplish this in one scan, that is, a variable value is passed on before it has been updated, a code loop has been identified. The system will display an Error Log output that presents the actual execution order, indicating one or several code loop occurrence(s).

This subsection will present analysis procedures for interpreting Error Log output and give some advice on how to deal with code loops.

See also Code Loops on page 124.

Error Log Output

A typical sorting error log output is shown in Figure 62.

Figure 62. A typical sorting error log output

The five parts of the log labeled A – E, in the figure are explained below.

- A: Code loop block identifier/delimiter
  Each code sorting error results in a block, numbered 1, 2, …

- B: Control module name
  The name of the control module in question.

- C: Control module type
  The name of the control module type in question (see B above).

- D: Code block name
  The name of the code block in question.

- E: Code block identifier
  The number given by the compiler to the code block.

Code Loop Block

As can be seen, the error log output is a structured list, where the top level is represented by code loop blocks, numbered 1, 2, 3, etc. Each code loop block corresponds to a code sorting error. If you correct the code for the error in code loop block 1, it disappears when you re-compile the code, and code loop block 2 will take its place, and all code loop blocks will be renumbered accordingly.

If there is a code sorting error within a type, you will receive a code loop block for each control module that uses the type. This means that if you use a type in 600 control modules, for example, you will see 600 code loop blocks in the error log. Correcting the error in the type will make all 600 code loop blocks disappear in the next run. Hence, a long error log may not indicate many errors, it may be a single error in a type that is used in many places.

Control Module Name and Control Module Type

Inside each code loop block you will find, on separate rows, the names of control modules that are related to the error. In Figure 63, code loop block 1, we have four control modules; *MyApplication.Pump1.V1*, *MyApplication.Pump1.V2*, *MyApplication.Pump1.M2*, and *MyApplication.Pump1.M1*.

Following the control module name, the type is displayed, for example, *Valve*, *Valve*, *Motor*, and *Motor*.

Code Block Name and Identifier

Within each control module section, the code block in which the error is contained is given. Each code block within an application is given a unique number (identifier) by the compiler.

Visualizing the Error Log Output

The next step is to visualize the error log output (in Figure 63). We will take code loop block 1 as an example.

1.   Concentrate only on the code loop block in question. Code loop blocks are not connected to each other, so if you correct the error(s) within a code loop block this will not result in errors being automatically corrected in other code loop blocks.

2.   Note the CONTROL MODULE lines and the succeeding CODE BLOCK line. For each code block line, draw a circle on a piece of paper and write "1", "2", etc, representing the code block identifiers.
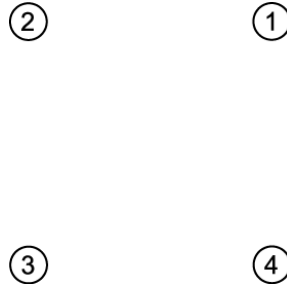
Figure 63. The four code block representatives in the Loop Block 1

3.   Return to the first control module in the code loop block and proceed to the CODE BLOCK section. Inspect the lines below the code block line (1). (Do not continue reading into the next CONTROL MODULE section.)

At the end of the first line we note that the variable *M1_Fwd* is read from "4". Draw an arrow from "4" to "1" (it is read from "4" by "1"). Label the arrow "M1_Fwd".

Continue to the next line. We note that the variable *V1_Open* is written to "3". Draw an arrow from "1" to "3" (it is written by "1" to "3"). Label the arrow "V1_Open".
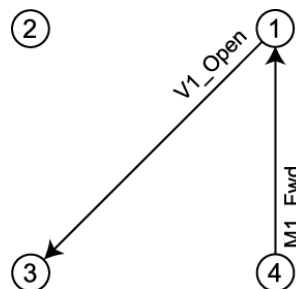
Figure 64. Analyze of the Code Block 1 dependency

We have now finished the analysis for code block 1.

4.   Continue to the next code block, code block 2

At the end of the first line the variable *M1_Fwd* is read from "4". Draw an arrow from "4" to "2" (it is read from "4" by "2"). Label the arrow "M1_Fwd".

Continue to the next line. The variable *V2_Open* is written to "3". Draw an arrow from "2" to "3" (it is written by "2" to "3"). Label the arrow "V2_Open"
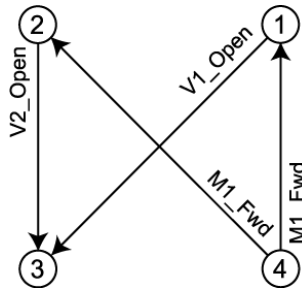
Figure 65. Analyze of the Code Block 2 dependency

We have now finished the analysis of code block 2.

5.   Continue to the next code block, code block 3

At the end of the first line the variable *V1_Open* is read from "1". Draw an arrow from "1" to "3" (it is read from "3" by "1"). Now you find that this has already been done. As you can see, there will be occasions when an operation has already been analyzed.

Continue to the next line. Here we have a similar situation, "3" reads "2". This has already been dealt with ("2" writes to "3"). Go to the next line.

The last line says: "3" writes to "4" using variable *M2_Stop*. Draw an arrow from "3" to "4" and label it "M2_Stop".
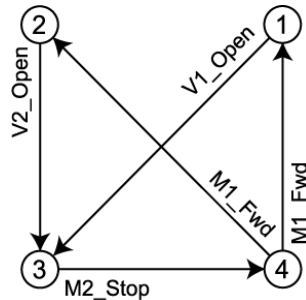
Figure 66. Analyze of the Code Block 3 dependency

We have now finished the analysis of code block 3.

Proceed in this way with the remaining code block 4. You will find that the flows and arrows have already been dealt with above. When finished, you should have a diagram like that in Figure 67.

6.   You may want to rotate the drawing to get a clearer picture of the flow, as shown below
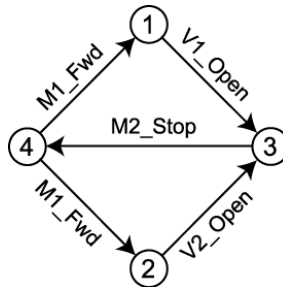
Figure 67. The Loop Block 1 visualized from the Error Log output

The analysis shows that the code contains two loops, and that the variable *M2_Stop* is part of both loops. Therefore, programming adjustments should, in this case, be concentrated to the variable M2_stop. These are the facts, we can either accept that code block 4 reads *M2_Stop* with a delay of one scan, or start over and re-design the code. See Correcting Sorting Problems on page 133.

When you have a more trained eye you will be able to identify the flows in the error log more quickly. Think as follows.

- Concentrate on one code loop block at a time.

- Reduce the control module lines by deleting the "CONTROL MODULE" lines.

```
CODE BLOCK ValveCode (1)
   READ      In => MyApplication.Pump1:M1_Fwd (4)
   WRITE     Out => MyApplication.Pump1:V1_Open (3)
CODE BLOCK ValveCode (2)
   READ      In => MyApplication.Pump1:M1_Fwd (4)
   WRITE     Out => MyApplication.Pump1:V2_Open (3)
CODE BLOCK MotorCode (3)
   READ      In => MyApplication.Pump1:V1_Open (1)
             In2 => MyApplication.Pump1:V2_Open (2)
   WRITE     Out => MyApplication.Pump1:M2_Stop (4)
CODE BLOCK MotorCode (4)
   READ      In => MyApplication.Pump1:M2_Stop (3)
   WRITE     Out => MyApplication.Pump1:M1_Fwd (1)
             Out => MyApplication.Pump1:M1_Fwd (2)
```

- Delete the code block name for simplicity.

```
CODE BLOCK (1)
   READ      In => MyApplication.Pump1:M1_Fwd (4)
   WRITE     Out => MyApplication.Pump1:V1_Open (3)
CODE BLOCK (2)
   READ      In => MyApplication.Pump1:M1_Fwd (4)
   WRITE     Out => MyApplication.Pump1:V2_Open (3)
CODE BLOCK (3)
   READ      In => MyApplication.Pump1:V1_Open (1)
             In2 => MyApplication.Pump1:V2_Open (2)
   WRITE     Out => MyApplication.Pump1:M2_Stop (4)
CODE BLOCK (4)
   READ      In => MyApplication.Pump1:M2_Stop (3)
   WRITE     Out => MyApplication.Pump1:M1_Fwd (1)
             Out => MyApplication.Pump1:M1_Fwd (2)
```

- Replace "Read In =>" and the application name, up to the variable name by "Read:", change the places of the variable name and the identifier.

```
CODE BLOCK (1)
   READ      (4) M1_Fwd
   WRITE     Out => MyApplication.Pump1:V1_Open (3)
CODE BLOCK (2)
```

```
    READ     (4) M1_Fwd
    WRITE    Out => MyApplication.Pump1:V2_Open (3)
CODE BLOCK (3)
    Read:    (1) V1_Open
             (2) V2_Open
    WRITE    Out => MyApplication.Pump1:M2_Stop (4)
CODE BLOCK (4)
    Read:    (3) M2_Stop
    WRITE    Out => MyApplication.Pump1:M1_Fwd (1)
             Out => MyApplication.Pump1:M1_Fwd (2)
```

•   Replace "Write Out =>" and the application name, up to the variable name
    with "Write:", change the places of the variable name and the identifier.

```
CODE BLOCK (1)
    READ     (4)  M1_Fwd
    WRITE    (3)  V1_Open

CODE BLOCK (2)
    READ     (4)  M1_Fwd
    WRITE    (3)  V2_Open

CODE BLOCK (3)
    Read:    (1)  V1_Open
             (2)  V2_Open
    Write:   (4)  M2_Stop

CODE BLOCK (4)
    Read:    (3)  M2_Stop
    Write:   (1)  M1_Fwd
             (2)  M1_Fwd
```

•   The flow has now been greatly simplified, and is easier to read. Code Block 3
    write out the value from variable M2_Stop, before the value has been read in
    (see Code Block 4), thus one scan delay.

Correcting Sorting Problems

You should always design your type solutions (motors, valves etc.), so that the user
of the objects never has to correct sorting problems caused by your types. A control
module type normally requires at least three code blocks: one that receives signals

from the outside, another that performs the actual control and handles any state machines, and a third that transfers signals to the outside.

There are a number of ways of correcting a sorting problem. The solutions below represent an ascending scale of responsibility for the programmer. Complex code loops may require combinations of these methods, in order to solve problems efficiently.

1. Change code blocks contents.
    This is the best way to correct a sorting problem. The compiler determines the execution order of the code blocks in each control module. It is sometimes possible to correct a code loop simply by splitting, or merging, code blocks. Note that all "pages" in an FBD or LD code block belong to the same code block.

    a. Split one or more of the affected code blocks into two or more code blocks.

    b. Merge two or more of the affected code blocks. Note that this is only possible if they are in the same control module.

2. Instruct the compiler what action should be taken when a sorting problem arises. This is necessary if it is not possible to solve the code loop.

    a. Use the "State" attribute on the variable/variables that form the code loop.

        Use ":New" when you want the value from the current scan, and ":Old" when you want the value from the previous scan. Use ":Old" in the code block that is the least affected by data one scan old.

        This alternative is used when the programmer wants to decide the sorting order. Note, however, that not all data types have the "State" attribute; only simple data types, excluding the string type.

The State attribute can only be specified for local variables of type bool, int, uint, dint, and real. If you, for some reason, want to override sorting and thereby avoid the State implications, you can assign the NoSort attribute to the variable.

3. Use "NoSort" attribute on the variable/variables that form the code loop. This alternative is used when it is not important which code block is executed first. In this case, the programmer take full responsibility and allows the compiler to randomly choose one code block before the other.

> Incorrectly used, the NoSort attribute may cause execution errors and application failure. Use *NoSort* only if you know the data flow characteristics in detail.

Code Blocks in SFC

Code loops in code blocks called by N actions are often confusing to people who are new to control modules. Pieces of code placed in N action steps can never be in conflict with each other, since this is prevented by the transitions involved. However, the compiler does not consider that, but sorts the N actions just like other code blocks, thus indicating code loops. If possible to bypass these loop-warning messages, move the affected code from N actions to the outer action steps P1 or P0.

Code Optimization

Code optimization involves many different activities and can be seen from a number of angles. This section does not aim to tell you all about code optimization, it merely intends to provide you with hints and good advice within a number of important fields:

- Basic Rules and Guidelines Regarding Tasks and Execution on page 136 gives you some basic rules and guidelines regarding tasks and execution.

- Function Block, Operation, and Function Calls on page 137 discusses the best way to call function blocks and functions, and gives some tips on how to improve communication with the operator interface.

- Excessive Conditional Statements on page 140 shows you how to reduce the number of conditional statements (these consume a lot of time and memory).

- 16- or 32-Bit Data Variables on page 141 discusses in which cases it is advisable to use 16-bit and 32-bit variables, respectively.

- Variables and Parameters on page 141 contains recommendations and advice regarding string handling, data types, and the use of retain and cold retain attributes.

- Code Optimization Example on page 142 contains an alarm and event handling code optimization example.

Basic Rules and Guidelines Regarding Tasks and Execution

There are some basic rules and guidelines regarding tasks and execution:

- A maximum of 70% of the system capacity can be used for the execution of application code (This is not valid for AC 800M HI. For details, refer to *System Guide, Technical Data and Configuration, (3BSE041434*)*. In the Controller Settings dialog box select Enable overload compensation under Load Balancing.

- Code should not be executed more often than necessary. Example: A regulator for pressure normally needs to be executed several times faster than a regulator for temperature (since pressure changes more rapidly than temperature). Code that takes 25 ms to execute and is executed every 100 ms (interval time) loads the system with 25% (load = execution time / interval time). If the same code is run every 500 ms, the load decreases to 5%. Pieces of code with different execution frequency are best connected to separate tasks.

- Code should not be written in such a way that it takes an unnecessarily long time to execute. On the other hand, code should not be written in a cryptic way, just to speed up execution time. The code must be clear, so that anyone can understand it.

- Task interval times should be multiples of each other.

Recommendations

- Code items that are closely connected or exchange large amount of data with each other should be connected to the same task.

- There should be no more than five tasks in each controller. The controller can handle more than five tasks, but experience shows that most users find it difficult to maintain an overview if there are too many tasks.

- Execution of code should not load the system by more than 55–65%.

- Excessive use of code tabs lowers the performance, use only as many code tabs as the design requires without loosing readability or creating code loops.

- Use attribute *by_ref* for *In* or *Out* whenever possible. When parameters are passed to control modules and function blocks, especially for strings and large data types it is time consuming. In that case, a reference to the data instance is passed in the function block call. This is achieved by setting the attribute of the parameter to *by_ref*.

- Use the Task Analysis tool to analyze the task scheduling in the application, view the graphical representation of how the tasks will execute, understand possible overload situations, and prepare remedial actions by changing the execution time in the analysis.

Function Block, Operation, and Function Calls

String Function Calls

String handling is especially demanding for the CPU. By minimizing the number of string operations, significant savings in CPU capacity can be made. Especially string concatenations increase CPU load, and should be avoided, if possible.

Reducing String Execution Time

In some cases, string values can be sent to HSI system software, and concatenations etc. can be performed in the workstation instead. Producing a report using string concatenations and a function block for printing to a local printer (for example *PrintLines*) is a heavy task for the CPU. The report can instead be produced in the HSI system.

In some cases, strings are more or less static, that is, they are never or seldom changed. In this case the code should be designed so that string operations are performed only when it is absolutely necessary. The IP address, when using MMS communication, is an example of this. Instead of passing the IP address to the *MMSConnect* function block every cycle, it can be done in the first cycle of execution only, by using start code blocks.

Another alternative is to use project constants for static strings, or to encapsulate the string procedure in a conditional statement, so that it executes only on demand.

Using Start Code Blocks

As we have seen before, it is sometimes useful to prevent parts of the code from executing every cycle. A typical example is the passing of static string values (for example IP addresses) to function blocks. Another example is when strings are to be concatenated to produce alarm messages, etc. Adding two strings, such as "Tank_114" and "_High_Level", increases CPU load considerably if the operation is performed in each cycle. Since the string value in this case is not meant to be changed during execution, it is much more efficient to execute the string operation only once, when the code starts to execute.

When control modules are used, this is achieved by using *Start Code* blocks. By naming the code block *Start_name*, the code inside the code block will only be executed in the first cycle after a cold start, or a change in the code. The code inside a start code block is only executed once after a download.

If the project consists of programs and function blocks instead, the function *FirstScanAfterApplicationStart* can be used. This function only returns a true value for the first cycle after downloading a change in the code, or after a cold start. This function can be used in conditional statements, for example, to set the value of a function block parameter:

```
If FirstScanAfterApplicationStart() then
  MMSConnect_1.Partner := IPAddress_1;
end_if;
```

 where IPAddress_1 is a string variable with the actual address as initial value.

Using Project Constants

Project constants are especially useful for passing constant values into function blocks in different programs. The benefits are that the values are always the same for all function blocks that use a particular constant.

Project constants are suitable to use for library items that the user wants to change. Examples are, date and time formats, logical colors and logical names. Do not use project constants to change the functionality of an object, for example, initial values and comparisons in code.

One example of this is the time-out for an ACOF function block. Let us say that there are ACOFs in ten different programs in your project. They should all have a time-out setting of 3 s. Instead of declaring 10 variables with the initial value of 3 s and the attribute *constant*, one project constant with this value can be created and used for all ACOF function blocks.

Firmware Functions for Arrays and Struct

Firmware Functions for Arrays and Struct might be time consuming.

Carefully consider the impact on 61131-3 execution when using the firmware functions for arrays and structs, if they contain string variables. Execution time can go up to 100-150 µs per string in a PM864/865 (e.g., creating an array with 100 string components takes roughly 10 ms).

The 1131 scheduler is NOT executed during the function call, causing latency of the same duration as the function execution time, even to higher-priority tasks!

Exceeding the configured latency for SIL tasks will cause a controller halt.

Arrays and structs without strings are much less time-consuming, and should not cause any problems for array/struct sizes <1000 elements. Creating an integer array of maximum size (65535 elements) takes 25 ms.

Excessive Conditional Statements

If you use conditional statements, care must be taken to avoid unnecessary execution of code. Let us look at the following examples.

Example 1

```
If Reset then
 Count := 0;
elsif Stop then
 Count := Count;
elsif Start then
 Count := Count +1;
end_if;
```

Example 2

```
If Reset then
 Count := 0;
end_if;
If Stop then
 Count := Count;
end_if;
If Start then
 Count := Count +1;
end_if;
```

The code examples above have the same function, but the code in example 1 is executed much faster than that in example 2.

In the first example only the first condition is evaluated as long as Reset is true. In the second example all conditions must be evaluated in each cycle, even though the three variables are mutually exclusive. If there is a large amount of code inside the conditions, valuable CPU power will be wasted.

16- or 32-Bit Data Variables

In many cases, the choice of data type for a variable or parameter is obvious, but sometimes, making the wrong choice will actually lower the performance of the CPU. In most cases, working with variables of data type *int* (16 bits) gives worse performance than using *dint* (32 bits) as data type. The reason for this is that the CPUs are designed for 32 bit operations. If 16-bit data are used, the CPU has to transform these values to 32 bits before the operation can be made, and then back to 16 bits after the calculation has been completed.

Variables and Parameters

Strings

Below is some good advice when handling strings.

- The handling and copying of strings creates considerable CPU load. Variables of type *string* require a great deal of memory, and the execution time is long during copying and concatenation.

- Always use square brackets, [ ] around strings, to limit string variables or string parameters.

- Copy strings every scan only when required. Remember that a connection to a function block actually involves copying variables from one to the other, see the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

- Concatenation of strings should only be performed when absolutely necessary.

- Use the attribute *by_ref* for *in* and *out* function block parameters whenever possible.

Variable Attributes

The attribute *retain*[1] should only be set for variables if they really require such an attribute, as this attribute increases the stop time during download of program changes to the controller.

---

1.   All function block parameters are set to *retain* by default.

However, *In* parameters should normally have the attribute *retain* to obtain bumpless transfer of signals, after a warm restart. *Out* parameters that are always written (before reading) in each scan, do not require a *retain* attribute.

More information on attributes can be found in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Code Optimization Example

To facilitate readability, the following conventions are used in the examples below.

fb	Function Block
p	Parameter
v	Variable
c	Project constant

The code below (before optimization) will generate an alarm, and an output will be activated if the alarm value exceeds a preset value.

```
(* Comment placed here *)
fbSR( S1 := pLevel > pHighLevel,
   Reset := pLevel < pNormalLevel,
   Q1 => vSignal );
fbAlarmCond( Signal := vSignal,
   SrcName := pName,
   Message := pDescription + vTextSpace + vTextHighLevel,
   Severity := pSeverity,
   Class := pClass,
   FilterTime := pFilterTime,
   EnDetection := pEnDetection,
   CondState => vCondState );
pAlarm := vCondState > 2;
```

The code below has been optimized for the best performance.

```
(* The variable vFirstScan is default true *)
if vFirstScan then
```

```
   fbAlarmCond.SrcName := pName;
   fbAlarmCond.Message := pDesciption + cTextSpace +
cTextHighLevel;
   fbAlarmCond.Severity := pSeverity;
   fbAlarmCond.Class := pClass;
   fbAlarmCond.FilterTime := pFilterTime;
   vFirstScan := False;
end_if;
if pLevel > pHighLevel then
   fbAlarmCond.Signal := True;
elsif pLevel < pNormalLevel then
   fbAlarmCond.Signal := False;
end_if;
fbAlarmCond( EnDetection := pEnDetection );
pAlarm := fbAlarmCond.CondState > 2;
```

The following modifications have been made in the optimized code example:

- The function block *fbSR* has been replaced by equivalent code.

- Local variables have been replaced by project constants.

- The alarm function block is called using connected static values during start-up only.

- The alarm function block will be called continuously with its parameters, which can be changed dynamically.

- Writing and reading the inputs and outputs of the alarm function block is performed without using any intermediate variables.

Task Tuning

To make all the tasks work together, without deterioration in performance, you may have to tune task settings, such as offset. In the example below, we show a typical situation and some actions that can be taken in order to tune task execution.

This topic contains:

Example of Task Tuning Using Manual Analysis

Assume there are four tasks in a controller, see Table 9.

Table 9. Four tasks with interval and execution times

Task	Interval Time (ms)	Execution Time (ms)
A	50	10
B	150	20
C	300	30
D	600	20

Follow the steps below to tune these tasks for optimum performance.

1. Compile information

   Gather information about existing tasks. Tasks that are defined, but not used, should be deleted. Note the interval time and execution time of all remaining tasks.

2. Analyze

   Analyze the tasks regarding the interval time, that is, are they reasonable?

   It is recommended that all interval times must be multiples of each other. A slower task should preferably have an interval time that is *n* times the interval time of the closest faster task. In this example, the interval times are optimal, B has 150 = 3 x 50
   (= A), C has 300 = 2 x 150 (= B), and D has 600 = 2 x 300 (= C)ms.

   The smallest common denominator is 50 ms.

3. Draw a time diagram

   Draw a time diagram of a complete cycle, in our case, 600 ms. Mark the 12 time slots (600/50 = 12).
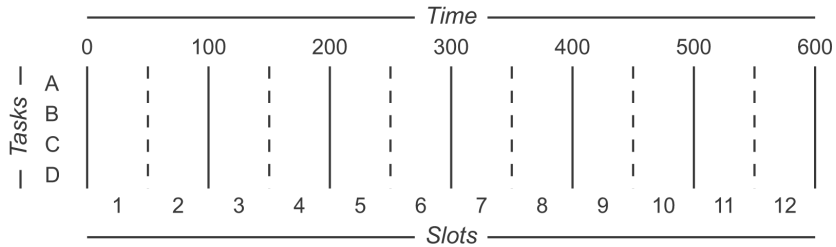
4. Insert the tasks into the time diagram

   a. Start by inserting the task (or tasks) that must be executed in every time slot (in our case, task A).
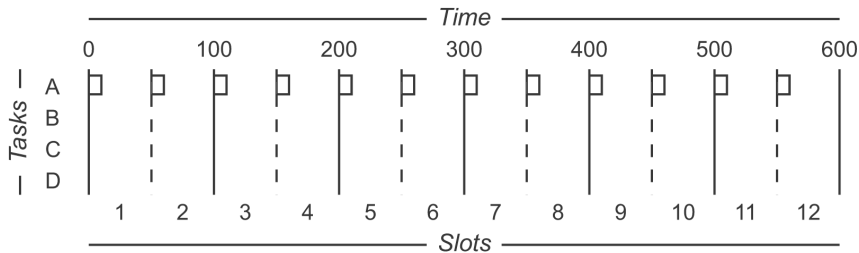
   b. Continue with the second shortest interval time (in our case, task B). This task should be executed in every third slot (interval time = 150 ms = 3 slots). Since the execution time for A + B = 10 + 20 ms < 50 ms we can start in slot 1 and then use slots 4, 7, and 10.
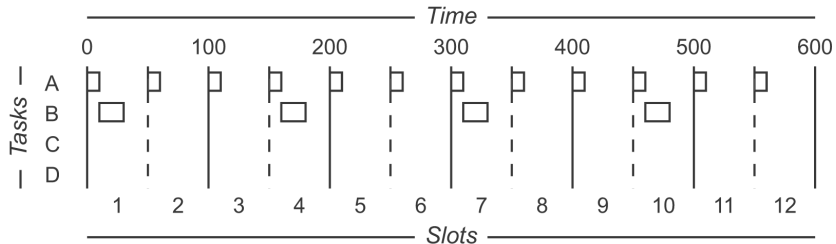
   c. Continue with the third shortest interval time (in our case, task C). This task should be executed in every 6th slot (interval time = 300 ms = 6 slots). Since the execution time for A + B + C = 10 + 20 + 30 ms > 50 ms

we cannot start in slot 1. We start in slot 2, and then again after 6 slots, that is, slot 8.

d. Finally, consider the task with the longest interval time (in our case, task D). This task should be executed in every 12th slot (interval time = 600 ms = 12 slots), that is, once every cycle. Task D could be started in slot 1 since the execution time for the three tasks then would become 10 + 20 + 20 = 50 ms. But then the entire slot would be filled, which is not recommended. Task D cannot be placed in slot 2 for the same reason. We choose slot 3.
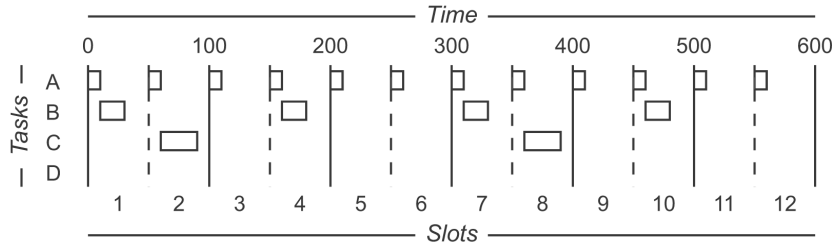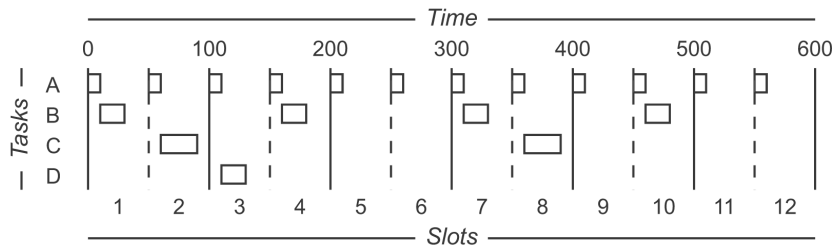

What should you do if a task cannot be placed in a time slot?

There are three ways of dealing with this problem.

1. Use the priority settings, to specify which task should be prioritized.

2. Divide a task into two tasks, with the same interval time, but with shorter execution times.

3. Examine the possibility of increasing the interval time for the task that has the shortest interval time. This will increase the width of the time slots.

The first alternative means longer execution time for task execution, thus other functions are locked-out for a longer time, before they can be executed. Alternative 2 and 3 are the most suitable ones, since they use offset. With offset, a time gap is maintained between task executions for other functions, for example communication. However, to accomplish offset for tasks, you need to re-design the entire time diagram.

Calculate Offset for Tasks

Calculate the appropriate offset for the tasks. Task A should be executed first in every slot, so we set the offset for task A to 0 ms.

Task B will be executed together with task A in slots 1, 4, 7, and 10. Task A has the execution time 10 ms, and task B 20 ms. In these slots there will be $50 - 10 - 20 = 20$ ms of free execution time. This allows us to place these 20 ms in equal parts on each side of task B, that is, set the offset for task B to 20 ms (the execution time for task A + 10 ms).

The modified time diagram for task B is shown below.
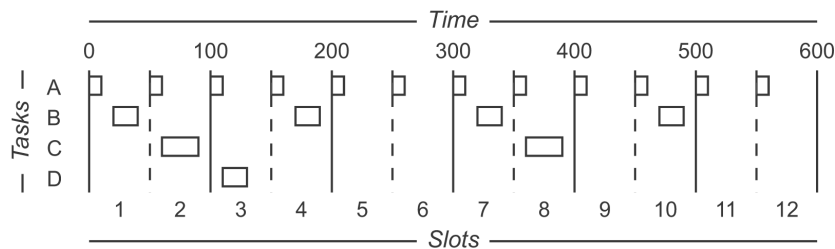
Task C is modified in the same way as follows.

Free time in slot = slot time (50 ms) – execution time for task C (30 ms) – execution time for task A (10 ms) = 10 ms. Divide this time to give a 5 ms interval on each side of task C.

The offset for task C is then: 50 ms (slot time for slot 1) + 10 ms (task A) + interval (5 ms) = 65 ms.

The modified time diagram for task C is shown below.
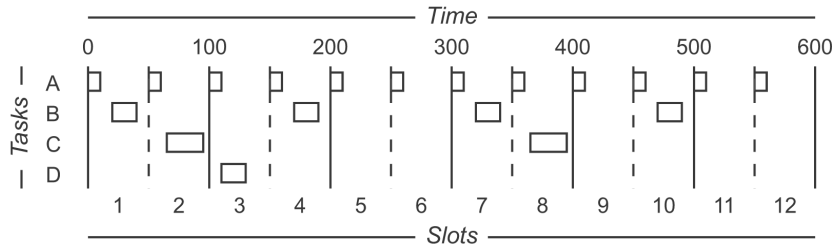
Task D is also modified as follows.

Free time in slot = slot time (50 ms) – execution time for task D (20 ms) – execution time for task A (10 ms) = 20 ms. Divide by 2 to give 10 ms time intervals before and after task D.

The offset for task D is then: 100 ms (slot time for slots 1+2) + 10 ms (task A) + interval (10 ms) = 120 ms.

The modified time diagram for task D is shown below.
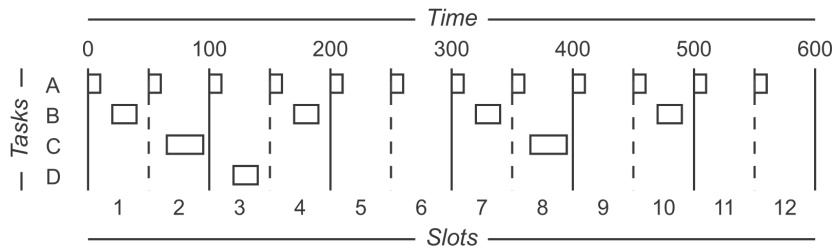
We have now tuned our tasks without having to use priorities. All the tasks in the example above have the same priority. The only reason for using priorities is when you can not find a slot to contain a task as mentioned above.

Example of Task Tuning Using the Task Analysis Tool

The Task Analysis tool in Control Builder can be used to tune the tasks for analysis. The tool also displays warnings or errors if the tasks are not well tuned.

Let's assume the following tasks exist in a controller:

Table 10. Four tasks with interval and execution times

Task	Interval Time (ms)	Offset	Execution Time (ms)
A	50	0	10
B	150	65	20
C	300	115	30
D	600	265	20

This is a well tuned task configuration (for normal execution) that will allow all tasks to execute on time and within the overrun and latency restrictions. The load from the tasks should be (10/50 + 20/150 + 30/300 + 20/600) * 100 = 46.7%. The Task Analysis tool displays the task execution as shown in Figure 68.

Figure 68. Well tuned tasks displayed in Task Analysis tool

Let us then assume that an application only running in Task B is to be downloaded. This will create a total load of 60% (if the execution time is the same), still leaving some time for other system jobs to execute, e.g. for communication, but it will result in a warning that the cyclic load is high since it exceeds the limit of 50%.

The execution will result in a task layout without margins and in fact Task B will be interrupted by Task A, see Figure 69. This should not be a problem in itself but still, it is not fully acceptable, so a warning for interrupted task execution appears with the information about the consequences. It is possible with the Task Analysis tool to abort or continue the download.

Figure 69. Tasks with warning displayed in Task Analysis tool.

It can also be seen that the risk for task C to get latency has increased significantly. If task B suddenly executes just slightly longer than before, task C will not be able to start at the correct time. This will also give a warning, since the margin is less than 5% of the interval time of task C.

In the Task Analysis tool it is possible to modify the execution time of tasks for analysis, and view the updated analysis in the graph. For details, refer to *System 800xA Control, AC 800M, Configuration (3BSE035980*) manual.

Appendix A IEC 61131-3 Standard

Main Objectives

The main objectives of the IEC 61131-3 standard are as follows.

- The standard encourages well-structured program development. All application programs should be broken down into functional elements, referred to as program organization units or POUs. A POU may contain functions, function blocks or programs.

- It should be possible to execute different parts of the application program at different rates. This means that the system must support individual interval times for different POUs.

- Complex sequential behavior can easily be broken down into events using a concise graphical language.

- The system must support data structures so that associated data can be transferred between different parts of a program as if they were a single entity.

- The system should have parallel support for the five most used languages, Ladder Diagram (LD), Instruction List (IL), Function Block Diagram (FBD), Structured Text (ST) and Sequential Function Chart (SFC).

- The programming syntax should be vendor independent, resulting in more or less portable code that can easily be transferred between programmable controllers from different vendors.

Benefits Offered by the Standard

Well-structured Software

The main purpose of the IEC 61131-3 standard is to improve overall software quality in industrial automation systems. The standard encourages the development of well-structured software that can be designed either as *top down* or *bottom* up software. One of the most important tools in achieving this is function blocks.

A *function block* is part of a control program that has been packaged and named so that it can be reused in other parts of the same program, or even in another program or project. Function blocks can provide any kind of software solution from simple logical conditions, timers or counters, to advanced control functions for a machine or part of a plant. Since the definition of input and output data has to be very precise, a function block can easily be used, even by other programmers than those who developed it.

By packaging software into function blocks the internal structure may be hidden so that well-tested parts of an application can be reused without risk of data conflict or malfunction.

Five Languages for Different Needs

The IEC 61131-3 standard supports five of the most commonly used programming languages on the market. Depending on previous experience, programmers often have their personal preferences for a certain language.

Since most older programmable controllers use Ladder Diagram or Instruction List programming, there are often many such programs available. These programs can relatively easily be reused in new systems supporting the standard.

Today's programmable controllers can handle both logical conditions for digital signals and arithmetic operations on analogue signals. Arithmetic operations are much easier to program with Structured Text than with Ladder diagrams.

The initial structuring of a control application is normally best done with the graphical language Sequential Function Chart. This method is ideal for describing processes that can be separated into a sequential flow of steps.

An optimal software application often contains parts written in more than one of the five programming languages. The standard allows the definition of function block types using all the languages.

Software Exchange between Different Systems

Before the IEC 61131-3 standard was established it was not possible to port control programs from one vendor's programmable controller to a competing system. This has been a major obstacle to a free market, where the customer selects a system based on the suitability of the hardware and price, rather than by the type of programming languages supported by the controller.

With programmable controllers that are IEC compliant the potential for porting software is much better. Software developed for one manufacturer's system should, at least theoretically, be possible to execute on any other IEC- compliant system. This would open up the market dramatically resulting in better standardization, lower prices and also improved software quality.

Unfortunately such a high level of software portability may be difficult to achieve in practice. The IEC 61131-3 standard defines many features and only requires that vendors of programmable controllers specify a list of which features their system supports. This means that a system can be compliant with the standard without supporting all features. In practice, portability will therefore be limited, since systems from two different vendors often have different feature lists.

Appendix B Naming Conventions and Tools

> For more information about naming conventions etc, when creating control
> module types, see also the manual *AC 800M, Library Objects Style guide
> (3BSE042835*)*.

Introduction

In order for operators and maintenance personnel to be able to quickly find
information about the status of your process and recent events, it is important that a
consistent naming strategy is used. This appendix gives advice on how to avoid
naming problems:

- Naming Conventions on page 155 gives an overview of naming conventions
  that are widely used throughout the field of automation. It is recommended that
  these conventions are considered when creating a naming standard.

- Suggested I/O Signal Extensions on page 168 suggest names for types,
  parameters and I/O extensions. Read this section if you need a basis for
  developing naming rules.

- Name Handling in an 800xA System on page 178 describes the name handling
  functions of the 800xA system, including name aspects and how to address
  sub-objects using type qualification. This part also describes how to use the
  Name Uploader to upload names to the system from the controllers.

> For information on NLS handling of alarms and events, see *System 800xA,
> Control, AC 800M, Configuration (3BSE035980*)*. For information on NLS
> handling of graphics, see operator workplace information.

Naming Conventions

The importance of a good naming standard cannot be over-emphasized. The standard should be general enough to cover all possible applications and all possible extensions or changes to the plant installation. Needless to say, the name of each item must be unique, with no room for misinterpretation. The names used should follow rules that are easy to learn and have precise definitions. It should be possible for personnel at all skill levels to comprehend and use these rules.

Object names have been in use for a long time and certain national and international standards have been defined. In addition, different plants and companies have defined their own standards. These should, of course, be used as a base, which, with small changes and/or additions, will let you take full advantage of the possibilities offered by the 800xA system.

General Guidelines

When addressing a type, variable, parameter, function block, etc., excessive length is not recommended. Bear in mind that dot-notation is used to access hierarchical (structured) variables and if each "level" has a lengthy name, the total length of the path will be considerable.

Object Type Names

The length of object type names should not exceed twelve characters, but if this is not possible, up to 32 characters are allowed. If names contain an abbreviation, for example, MMS, it is recommended that these abbreviations be written in capital letters.

Variable Names

The identifier, the name of the variable, function block type or function, may have a maximum length of 32 characters, and may contain letters (a–z, A–Z), digits (0–9) and the underscore character (_). The first character must be a letter (or underscore), and spaces are not allowed. See also Table 12 on page 159.

More information is given in Control Builder online help. Search the index for "naming conventions".

Compound Words

If the parameter name is a compound word, it is recommended to write the verb/action first, and then the noun/object on which the action should be taken. The description field of the parameter should use the long name, for example, *AlarmCond*.

Other Naming Issues

• Several suffixes are allowed.

• Do not use _ (underscore) to separate abbreviations (occupies one position and is unnecessary).

• Acronyms should be written in capital letters, for example, *FBD* (Function Block Diagram).

• If several information windows are required, they should be separated by suffixes, for example *NameInfoBar*, *NameInfoHist*, *NameInfoPar.*

• Project constants should begin with a lower-case "c", followed by an uppercase letter. For structured data types, this rule applies to the main name only, not to individual components.

• All object names share the same name space. Therefore, it is important to avoid names that are too general. It is recommended to name libraries and types with a prefix, to avoid ending up with several names of the type "Motor1", originating from different libraries or type definitions.

Recommended Object Type Names

The following object type names are used in the standard libraries, and should also be used when creating your own libraries.

Table 11. Recommended object type names

Object Type	Description
*Name*Template[1]	An object type, which the user makes a self-defined type from, and then modifies it according to the needs. For example, EquipProcedureTemplate.

Table 11. Recommended object type names (Continued)

*Name*Core	Protected core functionality, typically re-used in several different object types. Functionality that may be subject to changes is put outside the core. For example, EquipProcedureCore, UniCore.
*Name*HSI	Contains calculations needed in the graphics (only to avoid performing of the calculations in the graphics), for example, UniHSI.
*Name*Icon	If the icon is used for a particular object type, Name should be equal to the name of that type, for example PidIcon. If, on the other hand, the icon is used for a certain function, *Name* should be equal to that function, for example ExecuteIcon*.*
*Name*CC	Used as suffix to all control modules that have a parameter of the data type ControlConnection. For example: PidCC.
*Name*M	A control module type that has the same functionality as an existing function block type should have the same name as that function block type added with the suffix "M" (for Control module type). For example: AlarmCond **–** function block type AlarmCondM **–** control module type
*Name*D	Naming convention for diagram types
*Name*Faceplate	Naming convention for object types that represent a faceplate popup window in Control Builder graphics, in which control, mode, forcing, etc., can be set. *Name* corresponds to the name of the main object type.
*Name*Info	Naming convention for object types that represent a popup window in Control Builder graphics, in which information is presented or parameter values can be set. *Name* corresponds to the name of the main object type.

(1) Object types intended to be templates – types that the user has to make a new definition of and rename before actual usage – should always have the suffix *Template* in the object type name. For example, the object type name *EquipProcedureTemplate.*

ℹ️ Note that some words are reserved, which means that they cannot be used. Such keywords, which are recognized by the IEC 61131-3 standard are, for example, *IF*, *THEN*, *ELSE*.

Non-Valid Characters in Object Names

Valid characters are all ASCII characters except the ones given in Table 12. Apart from the backslash character (\\), the following combinations, which constitute "control sequences" are not-valid in object names.

• \t – the Tab command,

• \n – the Newline command,

• \r – the Return command.

Furthermore, an object name may not begin with a digit (0-9).

Table 12. Non-valid characters

½	Half sign	§	Paragraph sign
"	Double quotation marks	@	At sign
#	Number sign	£	British pound sign
¤	Currency sign	$	Dollar sign
%	Percentage sign	&	Ampersand
/	Slash	{	Left curly brace
(	Left parenthesis	[	Left bracket
)	Right parenthesis	]	Right bracket
=	Equals sign	}	Right curly brace
+	Plus sign	?	Question mark
\\	Backslash	'	Right and left single quotation marks
^	Circumflex	¨	Diaeresis

Table 12. Non-valid characters (Continued)

~	Tilde	*	Asterisk
'	Apostrophes	<	The less than sign
>	The greater than sign	|	The vertical bar
,	Comma	;	Semicolon
.	Period	:	Colon
-	Hyphen		

Project Constants

It is important that you follow certain naming conventions when naming project constants. Apart from normal naming conventions, it is recommended that all project constants begin with a "c". Furthermore, it is not permitted to call parameters and variables in POUs by the same name as a project constant. If you do so, you might face some extremely serious situations. For example, a project constant with the name "Max" is not a good idea. If you also define a local variable with the name "Max", this will take precedence over the project constant, which could lead to severe problems. Use the name "cMax" instead or, even better, a more descriptive name such as "cMaxTempAllowedInTank".

Variables

Naming of variables is especially important in templates (compared to hidden or protected code), since the user should be able to easily understand the design and function of any given template. Some general recommendations on variable names are given below:

- Variable names should be descriptive.

  To help distinguish between different types of variables, add a suffix "v" to the variable name for local variables, "e" for external variables, and "cv" for communication variables.

- Underscore (_) should not be used. To separate parts of a name, use uppercase letters instead.

- Avoid global/external variables, whenever possible.

- Avoid access variables, whenever possible.

- Avoid very long names, especially in FBD.

- Add the suffix *Old* to the variable name, to create a variable that keeps the old value.

- Add the suffix *Loc* to the variable name, to create a variable that is a working copy of, for example, an *EDIT* parameter.

> For information on project constants, see *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Types and Parameters

Data Types

The naming rules for data types are essentially the same as for object types. That is, it is recommended that the length of data type names does not exceed 12 characters. When this is not possible, up to 32 characters are allowed.

You may find it helpful to add a suffix "Type" to your function block, control module, diagram, and data type to explicitly show that it is a type.

Parameters

The need for short names is equally important for function block types, control module types, and diagram types. The names in Suggested I/O Signal Extensions on page 168 and Table 11 on page 157 are reserved for special types of parameters. *FB, M*, and *D* denote that the name applies to function block types, control module types and diagram types, respectively. However, a parameter used in function block type, control module type, and diagram type should have the same name throughout all types.

Short names for types and parameters are important considering the following:

• How many function block symbols will fit on a screen or printed page in the Function Block Diagram (FBD) language.

• How many function blocks, control modules, and diagrams will fit on a page in the FD code block of a diagram or diagram type

In FBD and FD, the possibility to simultaneously see many symbols/objects and their connections is essential to readable logic. Unnecessary paging (both on screen and in printout) has a negative effect on readability, and requires more space for page references.

Also, note that using upper- and lower-case letters improves the readability of names, for example, *ManMode* is better than MANMODE. POU editors allow the use of upper- and lower-case letters for declaration of parameter names, and allows the user to refer to the name in any form, as long as the letters are the same (for example, entering *ManMode* is the same as entering MANMODE).

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

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

The length of parameter names should not exceed eight characters (however, the system allows longer names for parameters, up to the system limit of 32 characters). These restrictions also apply to graphically connected parameters in control module types and diagram types. Other parameter names in control module types and diagram types should be as short as possible, and easy to understand.

Full Names and Short Counterparts for Type Names and Parameter Names

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

- Use only part of a whole word.
  Example: Request = Req

- Remove all vowels (and some consonants).
  Example: Print = Prt

- Use a new word.
  Example: Communication Link = COMLI

Use the description field in Control Builder editors to provide a short name with its full name. Example: The description field for the parameter *PrtAckAlarms* may contain "Prints acknowledged alarms…" etc.

A list of names and recommended abbreviations of names for types and parameters is given in the *AC 800M, Library Object Style Guide (3BSE042835*)*.

Diagrams

It is recommended that you change the standard names for diagram POUs, from "Diagram1" etc. to what the diagrams actually do in the application, for example, "PIDControl".

Alarms and Events

Source names and condition names must follow the alarm and event syntax. There is also an NLS syntax if alarm texts are to be translated. See the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*.

Tasks

It is recommended that you change the standard names for the tasks, from for example "Slow", to what the tasks actually do control. However, do not include task-specific information such as interval time, offset, etc. in the name.

> You can read more about tasks in the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)*. If a High Integrity controller is used, there is also an additional task, called VMT (Virtual Machine Test). Never change the name of this task.

Libraries

Standard Libraries

The name of a library should include upper- and lower-case letters, forming the structure "*LibName***Lib**". That is, the different parts of the name should be separated with upper-case letters and the name string should always end with "Lib". The maximum string length may not exceed 20 characters (letters or figures), and some names are reserved.

Split Libraries

Split libraries are libraries that belong to a certain functionality family. Control libraries, in which the total functionality is split into several libraries, are examples of this. They all have control functionality in common, but are divided into categories, for example, *ControlStandardLib*, *ControlAdvancedLib*, and so on. The naming rules for split libraries should follow this example, that is, first the function family (Control), then the sub-category (for example "Advanced"), and, finally, the suffix "Lib".

Another example of this principle is a hidden-contents library that contains basic control functions; such a library could be named *ControlSupportLib*. That is, in a hidden-contents library, the word "Support" should always be included, between the main part of the name and the "Lib" suffix.

An object type name should be unique, not only within a library, but throughout all libraries. Remember that an object type is referenced by its name only, in other words, not by the name of the library.

As for library names, the object type name should start with an upper-case letter, and different parts of the name should be separated by capital letters, such as *PidCascadeLoop* and *FFToCC*

⚠️ A library may not have the same name as an application. If this happens, the handling of coldretain values will fail in certain situations.

Aspect Objects should be categorized in such a way that, for example, the names of all motors contain the string *Motor*, all valves contain the string *Valve*, all PID controllers the string *PID*, and so on.

I/O Naming

An object name should locate the object in the project, state the function of the object, and uniquely define the individual object. This can be achieved with a designation according to the following example:

Figure 70. Object name syntax

Location

The 3-character location "block" is a concatenation of a letter and two digits, describing the Area "block". The Area "block" consists of the concatenation of two digits, the first describing the process area, and the second the process equipment.

Function Code

Function codes for instrumentation functions have been in use for a long time and can be found in several national and international standards (for example ISA at www.isa.org).

Function codes should be reduced to two characters that indicate the main function. An *LIC*, would for example, be written *LC*.

There is no widespread standard for function codes for electrical equipment, and the simplest way of designating them is to use the equipment number or the code *M*, for a motor. A more elaborate standard is to define codes such as *PU* for pumps, *FA* for fans, etc. Ensure that you do not use any function code already used for instrument functions.

Individual Object Number

An individual object number is a sequential number within the part of the process addressing a single piece of equipment.

The individual object is defined by a three- or four- (electrical) digit number. These numbers can be divided into series used for instrument and electric functions, for convenience. If a process area has parallel lines or equipment, the same object number should be used for the same function on the parallel lines, using the location definition to create unique object names. This will help operators and others to remember object names.

Extension

The extension is defined as a 2−5 character code to identify single signals related to a main object.

A list of examples of different I/O and calculated signals, for which extensions should be defined, can be found in Suggested I/O Signal Extensions on page 168.

Example (analog control)

| AI
A21FT121A | PID
A21FIC121A | AO
A21FV121A |
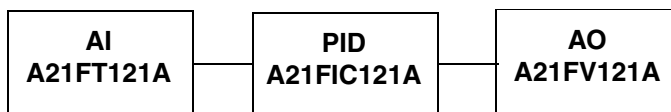
Figure 71. Analog control

Example (digital or Boolean control)

| Limit Switch Close A21HS155ZSC | Valve A21HS155 | Command Open A21HS155OPN |
| Limit Switch Open A21HS155ZSO | | Command Close A21HS155CLS |
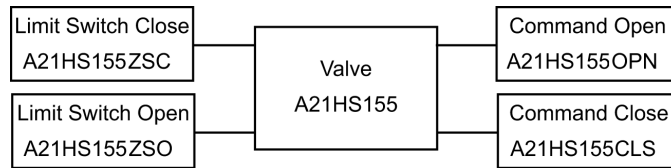
Figure 72. Digital or Boolean control

Collect I/O

You may collect all your I/O in a structured type. For such design, it is recommended that you add the prefix "IO_" to this structured type. For example, if the type is named "MotorType", its I/O data type could be called "IO_MotorType". The parameter of the type could be named "IO" (of "IO_MotorType").

> For examples on how to use structured data types, see Structured Data Type Examples on page 116.

Parameters

All objects with an Alarm Condition must have the *Name* and a *Description* parameter connected, if the Name Uploader aspect in Plant Explorer is to work properly. See Name Uploader on page 183.

It is never a problem to have a *Name* parameter in control module types. In function block types, however, care must be taken, since the parameter is copied to a local copy in the function block, which consumes memory and degrades performance.

Descriptions

A description should be provided, whenever possible. This means that all object types and data types should have a brief (three or four short lines) but clear description which is shown under the *Description* tab in the lower pane of Project Explorer, when the object is selected.

A structured data type component should have a line of text briefly describing its purpose/function. Use the column specifically designated for this in the POU/object type editor. Parameters in object types should be described in a similar way.

Suggested I/O Signal Extensions

Table 13. Extensions for digital inputs

Function	Suggested extension or function	Function	Suggested extension or function
Alarm	ALM	Local forward	LFWD
Alignment	ZSA	Local reverse	LREV
Automatic control (order)	AUTO	Local start	LSTR
Bypass	BYP	Local stop	LSTP
Clear	CLR	Main contactor acknowledge	M
Close	CLS	Main contactor acknowledge, forward	MF
Control voltage	CTRLV	Main contactor acknowledge, high / fast	MH
Cycle	CYC	Main contactor acknowledge, low / slow	ML
Decrease (Local)	LDEC	Main contactor acknowledge, reverse	MR
Emergency pull cord switch	SAFE	Main contactor position	MPOS
Emergency stop	ESTOP	Manual control (order)	MAN
Fast	FST	Off	OFF
Fault	FLT	On	ON
Flow high	FSH	Operating time	OT
Flow low	FSL	Overload relay	OL

Table 13. Extensions for digital inputs (Continued)

Function	Suggested extension or function	Function	Suggested extension or function
Forward	FWD	Pressure high	PSH
Hand	HND	Pressure low	PSL
High	HI	Ready	RDY
Increase (Local)	LINC	Remote control (order)	REM
Interlock	INT	Reverse	REV
Jog (Inch)	JOG	Run	RUN
Jog forward	JFWD	Selector switch	S1
Jog reverse	JREV	Speed switch (Monitor)	SS
Level high	LSH	Speed switch high (Monitor high)	SSH
Level low	LSL	Speed switch low (Monitor low)	SSL
Limit switch	ZS	Start	STRT
Limit switch close	ZSC	Stop	STOP
Limit switch down	ZSD	Temperature high	TSH
Limit switch forward	ZSF	Temperature low	TSL
Limit switch in	ZSI	Test	TST
Limit switch open (or out)	ZSO	Torque monitor close	WSC
Limit switch reverse	ZSR	Torque monitor open	WSO
Limit switch up	ZSU	Torque switch	WS

Table 14. Extensions for digital outputs

Function	Suggested extension or function	Function	Suggested extension or function
Blocking	BLK	Main contactor forward	MFWD
Close valve	CLS	Main contactor high / fast	MHGH
Decrease	DEC	Main contactor low / slow, Crawl	MLOW
Electrically operated valve	EV	Main contactor open	MOPN
Electrically operated valve close	EVC	Main contactor reverse	MREV
Electrically operated valve open	EVO	Main contactor selector	MSEL
Increase	INC	Main contactor start	MSTR
Lamp (pilot)	H1	Main contactor stop	MSTP
Local/Remote	LR	Open valve	OPN
Main contactor close	MCLS	Solenoid valve	XV
		Trip or shutdown of external equipment	TRIP

Table 15. Extensions for analog inputs

Function	Suggested extension or functions	Function	Suggested extension or functions
Actuator position	ZT,POS	Power	JT
Analysis	AT	Power (reactive)	QT
Consistency	NT	Power factor (capacitive)	PFC
Flow	FT	Power factor (inductive)	PFI
Frequency	HZ	Pressure	PT
Level	LT	Speed	ST
Load	WT	Temperature	TT
Measured Value (General)	MV	Transmitter (general)	T
Motor current (Alt. 1)	IT	Vibration	XT
Motor current (Alt. 2)	CURR	Viscosity	VT
Operating time	OT	Voltage	UT

Table 16. Extensions for analog outputs

Function	Suggested extension or functions	Function	Suggested extension or functions
Control valve (general)	V	Output (General)	OUT
Flow control valve	FV	Pressure control valve	PV
Indication	IND	Set-point	SETP
Level control valve	LV	Temperature control valve	TV

The list shows examples of I/O signals. More types can be defined as desired.

Name Handling in an 800xA System

This section contains a description of the different name handling functions in the 800xA system:

- Name Aspects in an 800xA System on page 178 describes the two name handling aspects in an 800xA system and how they interact.

- Avoid Name Conflicts for Types–Type Qualification on page 181 describes how to use type qualification to address sub-objects that have identical names but belong to different main objects.

- Name Uploader on page 183 describes how to use the Name Uploader tool to upload object names.

It is not recommended to change the engineering workstation language during development. This might cause OPC and NLS handling problems.

Name Aspects in an 800xA System

There are two different name aspects in Plant Explorer for representing a control object name. The first is the Control Builder Name aspect, which always is synchronized with the present name in Project Explorer, and therefore always follows IEC 61131-3 naming conventions. The second is the Name aspect, which contains the corresponding aspect object name in Plant Explorer.

Provided that a control object is created in Project Explorer, these two aspects start with the same object name. If the control object is created in Plant Explorer and the name is in conflict with the IEC 61131-3 name restrictions, the aspect system automatically converts the name so that it follows the rules. During such a name conversion, the Control Builder Name aspect will be disconnected from the Name aspect, with consequences described in Consequences of a Rename on page 179.

There is also a third name aspect only for hardware units. For a hardware unit the name set in the OPC Source Name aspect will be a combination of the controller name and the unit path e.g. Controller_1-0.4.0. See the *System 800xA, Control, AC 800M, Configuration (3BSE035980*)* manual, Section 2 Alarm and Event Handling.

Name Conversion

If we create a control object in Plant Explorer and give it a name that is not allowed according to IEC 61131-3 naming conventions, it would still be possible to keep the new name without any name conflicts. The reason for this is that the aspect system handles multiple object names.

If a name is entered in Plant Explorer that does not follow IEC 61131-3 naming conventions, the Project Explorer name will be changed to fit the IEC 61131-3 naming conventions. If we, in Plant Explorer, name a tank object "12-1456%tank", the Control Builder conversion would look like "x12_1456_tank". The automatic name conversion function always adds an "x" to the beginning of the name and replaces all non-alphanumeric characters with underscore. If the name is longer than 32 characters, it will be truncated.

Name Server

The Name Server keeps track of names from both the Control Builder Name aspect and the Name aspect. This means that references to an object may refer to either name.

When you create a new object, the system proposes a valid IEC 61131-3 name for the object. Objects on the same level as the newly created object are checked, so that name conflicts are avoided.

You should create object descriptions in Project Explorer. These descriptions will then be transferred to the aspect system and finally be visible in the Name aspect.

Consequences of a Rename

When an object is renamed, the consequences depend on if the Name and Control Builder Name aspects are synchronized or not. There are two cases of interest:

*   Rename when the Name aspect has never been changed, and therefore has the same control object name as the Control Builder Name aspect:

    –   If you rename the object from the Control Builder Name aspect, the name will change in the Name aspect as well.

–   If you rename the object from the Name aspect, this will only change the name in Plant Explorer. The Control Builder Name aspect will not change, that is, the two aspects are no longer synchronized.

• Rename when the Control Builder Name aspect and the Name aspect already have separate names.

–   Changes in one aspect will not affect the other.

–   To synchronize the two aspects, simply rename the object in the Name aspect, so that it has the same name in both aspects.

Objects always have the same name in Project Explorer and in the Control Builder Name aspect.

The advantage of having a synchronized Name aspect is that the aspect object name will be updated every time you change the corresponding control object name.

Once the Name aspect has been used, which could be the case if you need to choose names from other industry standards that conflict with IEC 61131-3 naming conventions, synchronization is lost. Control object names in the Project Explorer **must** follow IEC 61131-3 naming conventions, but the corresponding aspect objects do not.

Name uploading can be used to secure unique names for control objects that are embedded in type solutions. See Name Uploader on page 183.

Never rename an object inside a POU editor. If you give an object a new name in an editor, the old object, with all its added aspects, will be deleted, and a new object, without those aspects, will be created.

Change the Name in the Control Builder Name Aspect

All objects in the Project Explorer must follow IEC 61131-3 naming conventions, which means that a name must start with a letter or underscore, and be followed by only letters, digits, or underscores. The maximum length is 32 characters.

You can change a control object name (as seen in Project Explorer) from two locations, provided that you follow the IEC 61131-3 name convention.

To change an object name from Project Explorer:

1.   Right-click the object and select **Rename**. A Rename window opens.

2.   Enter a new name and click **OK**.

To change an object name from Plant Explorer:

1.   In the structure that displays the object, select the control object. The Aspect pane opens.

2.   Click the **Control Builder Name** aspect. The Control Builder Name aspect preview pane opens.

3.   Enter the new name and click **Apply**.

Avoid Name Conflicts for Types–Type Qualification

Type qualification increases the freedom to name types after their logical context, even if the same type name exists in different libraries and/or applications. To address a specific type, use the following syntax:

```
LibraryOrApplicationName.TypeName
```

Type Qualification Example

A control project contains two libraries (MyLib1 and MyLib2), and both libraries contain a function block type MyFBtype. MyApplication has both libraries connected and needs to call MyFBtype, see Figure 73.

There is a need to separate between the two libraries, when referencing this type. Normally, this would generate an error message from the compiler, since the function block type 'MyFBtype' is defined in both 'MyLib1' and 'MyLib2'.
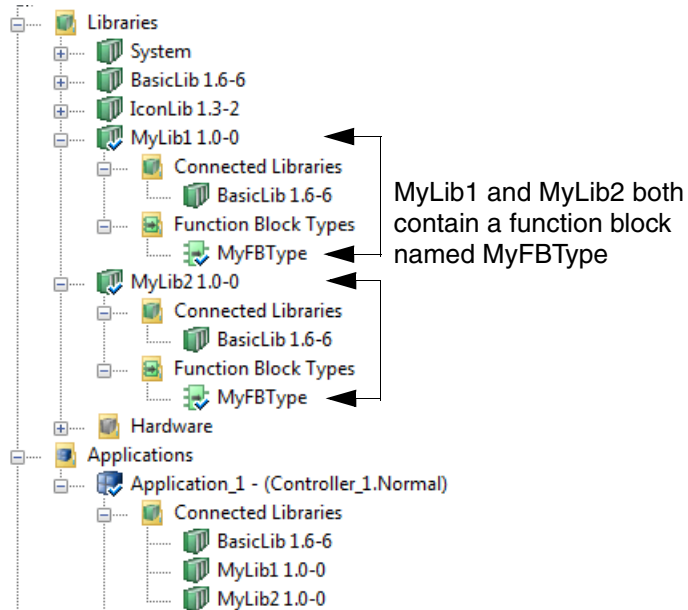
Figure 73. Application_1 has two libraries connected that contain the function block MyFBtype. A function block call to MyFBtype will cause a name conflict

This is avoided by declaring which library or application the type belongs to, before addressing it, see Figure 74.

Figure 74. The MyFBtype name conflict is avoided by first pointing to the library in the Program editor, and then referencing the function block type

Name Uploader

When a function block type, control module type, or diagram type is used to create an object, only the top-level of the type is assigned a new and unique name. The Name Uploader tool solves this problem by going through all objects further down in the Control Structure, looking for Name properties. The Name Uploader then fetches the names via the OPC server so that each object has a system-unique name. Each object calculates the name of the objects on the next level.

> ⚠ For the Name Uploader to work, all objects must have a Name and a Description parameter.

The Name Uploader aspect is by default placed on all control networks and on all applications. You can add the Name Uploader aspect to other objects if you want to upload a specific branch.

The Name Uploader must be run in order for suffixes to be added to all sub-objects.

To start a Name Upload for an application:

1. In the Control Structure, select your application and click the **Name Uploader** 🔧Name Uploader icon.The aspect preview pane opens.

2. Select the **Name Uploader** tab and click **Start Upload**.

3. Verify the name changes in the Control Structure.
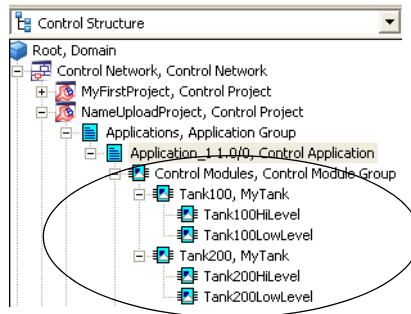
Figure 75. Tank100 and Tank200 after a name upload. The two objects HiLevel and LowLevel receives unique names in the Plant Explorer

Name Upload Example

The following example shows how to use the Name Uploader to fetch unique names for all objects.

In the TankLib example (Figure 75), a Tank100 object was created, which contains Motor10 and Motor20. They in turn have FeedbackError alarms.

But, there is also a Tank200 with a Motor10 and Motor20, and additional objects Tank300, Tank400, ... This means that we only have unique tank names, while there are several Motor10 and Motor 20 objects in the application. If the alarm FeedbackError activates inside Motor10, we cannot determine which Motor10 triggered the alarm. Did Motor10FeedbackError occur in Tank100, Tank200, ... or Tank800?

If the Name Uploader is used on the application, it will go through all objects and fetch the *Name* parameter of the object on the level above as a suffix. For all matches, it requests an OPC read of the corresponding OPC item.

The Motor10 object will be given the unique name, **Tank100Motor100**, provided that the Name parameter of the object Tank100 contains the name 'Tank100'.

For a more detailed description on how this works, study the code example in Figure 76 and the step-by-step description that follows after the figure.

MyTank type
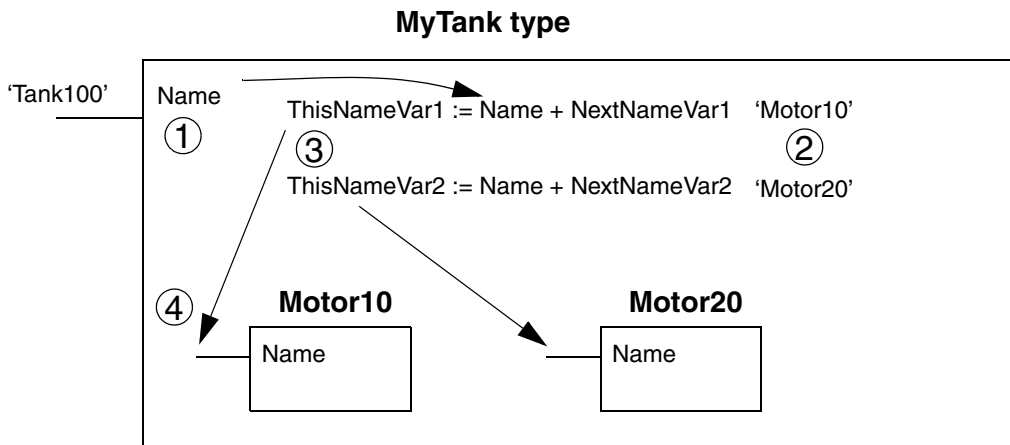
Figure 76. Code example showing how to create unique object names below the first level. Note the numbers and corresponding descriptions below (1 - 5)

Study the below steps to get a better understanding of how name uploading should be set up:

1. A *Name* parameter and four local variables *ThisNameVar1, ThisNameVar2, NextNameVar1* and *NextNameVar2* (all of data type string) have been declared in the MyTank type.

2. The initial values for *NextNameVar1* and *NextNameVar2* have been set to "Motor10", and "Motor20". The *Name* parameter gets its initial value, once a Tank object is created in the application.

3. In the Start Code block, the start value for the variables `ThisNameVar1` and `ThisNameVar2` will be the concatenation of the *Name* (Tank100) parameter of the tank object and the initial value of the variable `NextNameVar1` (Motor10), that is, 'Tank100Motor10'.

    (The same type of calculation is of course valid for `ThisNameVar2.`)

4. The *ThisNameVar1* variable is connected to the *Name* parameter in Motor10, and *ThisNameVar2* is connected to the *Name* parameter in Motor20.

5. The *Name* parameter of Motor10 now holds the following value:

        Tank100Motor10

Appendix C AC 800M Aspects

This Appendix presents the most common aspects you will meet during engineering in the Plant Explorer. The description below contains some branch-specific words that are explained under Terminology on page 13.

Table 18. AC 800M aspects

Aspect categories	Description
Access Variables	Holds XML source code for the access variables. Has a link to the CB editor.
Application	Holds XML source code for an application. Exposes runtime properties for application variables. Has a link to CB editors.
Confirmed Write Support	This aspect is added to SIL-classed objects to which it should be possible to write from an operator workstation. The aspect can also be used to add descriptive texts to a variable.
Control Alarm Event	Shows alarm conditions associated with object.
Control Module	Holds XML source code for a control module. Exposes runtime properties for the control module variables/parameters. Has a link to the CB editor.
Control Name Hook	Used to calculate proposed object names when creating new objects and copy/paste of objects.
Control Properties	Exposes engineering properties such as object specific initial values.
Controller	Holds XML source code for controllers.

Table 18. AC 800M aspects (Continued)

Aspect categories	Description
Data Type	Holds XML source code for a data type. Has a link to the CB editor.
Diagram	Holds XML source code for a diagram in the application. Exposes runtime properties for the diagram variables. Has a link to the CB editor.
Diagram Block	Holds XML source code for diagram type and instances. Exposes runtime properties for the diagram type variables/parameters. Has a link to the CB editor.
Function Block	Holds XML source code for a function block. Exposes runtime properties for function block variables/ parameters. Has a link to the CB editor.
Generic Folder	This aspect is automatically placed on AC800M folder objects in Plant Explorer (for example, the Applications folder). Its purpose is to protect folders from being renamed, moved or deleted.
Hardware Unit	Holds XML source code for a hardware unit. Exposes runtime properties for the hardware unit. Has a link to the CB editor.
Hidden	Holds XML source code for types with the attribute hidden.
Library	Represents a CB library. Holds XML source code for libraries.
Name Uploader	Used to upload names on objects from the controllers.
Program	Holds XML source code for a program. Exposes runtime properties for the program variables. Has a link to the CB editor.

Table 18. AC 800M aspects (Continued)

Aspect categories	Description
Project	Represents and holds XML source code for a CB project. GUI for insertion/removal of libraries to the project.
Project Constants	Holds XML source code for the project constants.
Property Attribute Override	Used to allow users to override property permission and authentication settings on properties.
Single Control Module	Holds XML source code for a single control module. Exposes runtime properties for the single control module variables/parameters. Has links to CB editors.
Task	Holds XML source code for a task. Has a link to the CB editor.

INDEX

W

WriteVar 45

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

3BSE043732-511

Power and productivity
for a better world™

ABB