SIEMENS

Ingenuity for life

Programming style guide
for SIMATIC S7-1200/ S7-1500

TIA Portal

Siemens
Industry
Online
Support

Legal information

Use of application examples

Application examples illustrate the solution of automation tasks through an interaction of several components in the form of text, graphics and/or software modules. The application examples are a free service by Siemens AG and/or a subsidiary of Siemens AG ("Siemens"). They are non-binding and make no claim to completeness or functionality regarding configuration and equipment. The application examples merely offer help with typical tasks; they do not constitute customer-specific solutions. You yourself are responsible for the proper and safe operation of the products in accordance with applicable regulations and must also check the function of the respective application example and customize it for your system.

Siemens grants you the non-exclusive, non-sublicensable and non-transferable right to have the application examples used by technically trained personnel. Any change to the application examples is your responsibility. Sharing the application examples with third parties or copying the application examples or excerpts thereof is permitted only in combination with your own products. The application examples are not required to undergo the customary tests and quality inspections of a chargeable product; they may have functional and performance defects as well as errors. It is your responsibility to use them in such a manner that any malfunctions that may occur do not result in property damage or injury to persons.

Disclaimer of liability

Siemens shall not assume any liability, for any legal reason whatsoever, including, without limitation, liability for the usability, availability, completeness and freedom from defects of the application examples as well as for related information, configuration and performance data and any damage caused thereby. This shall not apply in cases of mandatory liability, for example under the German Product Liability Act, or in cases of intent, gross negligence, or culpable loss of life, bodily injury or damage to health, non-compliance with a guarantee, fraudulent non-disclosure of a defect, or culpable breach of material contractual obligations. Claims for damages arising from a breach of material contractual obligations shall however be limited to the foreseeable damage typical of the type of agreement, unless liability arises from intent or gross negligence or is based on loss of life, bodily injury or damage to health. The foregoing provisions do not imply any change in the burden of proof to your detriment. You shall indemnify Siemens against existing or future claims of third parties in this connection except where Siemens is mandatorily liable.

By using the application examples you acknowledge that Siemens cannot be held liable for any damage beyond the liability provisions described.

Other information

Siemens reserves the right to make changes to the application examples at any time without notice. In case of discrepancies between the suggestions in the application examples and other Siemens publications such as catalogs, the content of the other documentation shall have precedence.

The Siemens terms of use (https://support.industry.siemens.com) shall also apply.

Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, systems, machines and networks.

In order to protect plants, systems, machines and networks against cyber threats, it is necessary to implement – and continuously maintain – a holistic, state-of-the-art industrial security concept. Siemens' products and solutions constitute one element of such a concept.

Customers are responsible for preventing unauthorized access to their plants, systems, machines and networks. Such systems, machines and components should only be connected to an enterprise network or the Internet if and to the extent such a connection is necessary and only when appropriate security measures (e.g. firewalls and/or network segmentation) are in place. For additional information on industrial security measures that may be implemented, please visit https://www.siemens.com/industrialsecurity.

Siemens' products and solutions undergo continuous development to make them more secure. Siemens strongly recommends that product updates are applied as soon as they are available and that the latest product versions are used. Use of product versions that are no longer supported, and failure to apply the latest updates may increase customer's exposure to cyber threats.

To stay informed about product updates, subscribe to the Siemens Industrial Security RSS Feed at: https://www.siemens.com/industrialsecurity.

Table of content

1 Introduction

Programming a SIMATIC Controller a programmer has the task to develop the application program as readable and structured as possible. Each developer applies their own strategy to realizing this task, e.g. naming of variables, blocks or the way the programs is commented. Different developers use different philosophies, therefore very different application programs exist, which often can only be interpreted by the respective creator.

Note	The basis for this document is the programming guide for SIMATIC S7-1200/ S7-1500, which describes the system properties of the controllers S7-1200 and S7-1500 how they are programmed in an optimal way: https://support.industry.siemens.com/cs/ww/en/view/81318674

1.1 Goal

The rules and recommendations described in the following chapters [are supposed to/ shall] help you create a uniform program code which is maintainable and reusable. In case of multiple developers working on the same application program; it is recommended to apply a project wide terminology as well as an agreed upon programming style. This way you can detect and avoid errors at an early stage.

For the sake of maintainability and readability it is required, to follow a certain format. Optical effects have only a limited impact on the quality of software. It is more important to define rules, which support the developer as follows:

- Avoiding typos and inadvertent mistakes, which the compiler then misinterprets
  Objective: The compiler shall recognize as many errors as possible.

- Supporting the developer diagnosing programming errors, e.g. reuse of temporary variables beyond one cycle.
  Objective: The Identifier indicates problems early.

- Standardization of applications and libraries
  Objective: The training shall be made easy and the reusability of the program code shall be increased.

- Easy maintenance and simplification of further developments
  Objective: Changes made in individual modules of the program code, should have minimal effects whole program. Changes may be performed by different programmers.

Note	The described rules and recommendations in this document are consistent and do not interfere with each other.

1.2 Advantages of a uniform programming

- Uniform and consistent style
- Easy to read and understand
- Easy maintenance and increased reusability
- Easy and fast error recognition and correction
- Efficient cooperation of multiple programmers

1.3 Applicability

This document is applicable for projects and libraries in the TIA Portal, which are programmed in the programming languages according to EC 61131-3 (DIN EN 61131-3), which are Structured Text (SCL/ ST), Ladder Logic (LAD/ KOP) and Function Block Diagram (FBD/ FUP).

This document is also applicable for Software Units, folders, groups, Organization Blocks (OBs), Functions (FCs), Function Blocks (FBs), technological Objects (TOs), Data Blocks (DB), PLC data types (UDTs), variables, constants, PLC message text lists, Watch tables and Force tables as well as for external sources.

1.4 Scope

This document doesn't contain descriptions of:

- STEP 7-programming with TIA Portal
- Commissioning of SIMATIC-controllers

Sufficient knowledge and experience in the mentioned topics above are the prerequisite to correctly interpreting and applying the given rules and recommendations.

This document serves as a reference and does not replace proper knowledge in the field of software development.

1.5 Rule violations and other regulations

In customer projects the applicable regulations, customer or branch specific standards as well as technological regulations (e.g. Safety, Motion, …) are to be followed and take precedence over the style guide or parts thereof.

When combining both, customer regulations with regulations within this style guide, special care must be taken to maintain integrity and consistency of the rules.

A violation of any of the regulations must be justified and documented appropriately in the user program.

The customer provided rules and regulations must be documented appropriately.

2 Definitions

2.1 Rules/ Recommendations

The regulations in this document are either recommendations or rules:

- **Rules** are binding definitions and must be followed. They are essential for a reusable and performant programming. In exceptional cases rules may be violated. This must be justified and documented.

- **Recommendations** are regulations, which support the uniformity of the program code and serve as support and documentation. Recommendations should be followed in general. However, there are exceptions when such a recommendation may not be followed. Reasons for this may be a better efficiency or better readability.

2.2 Enumerating rules

For a unique rule identification, within categories rules and recommendations are identified with a prefix (2 characters) and are enumerated (3 digits).

In case a regulation is canceled its number will not be reassigned. In case more regulations become necessary, you may use the numbers between 901 and 999.

Table 2-1

Prefix	Category
ES	Engineering System: programming environment
GL	Globalization:
NF	Nomenclature and formatting:
RU	Reusability:
AL	Allocation: Referencing of objects
SE	Security:
DA	Design and architecture:
PE	Performance:

2.3 Performance

The performance of an automation system is defined by the execution time of the program.

When mentioning a performance penalty, this means that it would be possible to reduce the execution time and therefore increase the user programs cycle time, by applying programming rules and an efficient way of programming.

2.4 Identifier/ Naming

It is important to differentiate an identifier and a name. The name is part of the identifier, which describes the meaning of an identifier.

The identifier is assembled out of:

- prefix
- name
- suffix

2.5 Abbreviations

The following abbreviations are being used throughout this document:

Table 2-2

Abbreviation	Type
OB	Organization block
FB	Function block
FC	Function
DB	Data block
TO	Technology object
UDT	PLC data type

2.6 Terms used with variables and parameters

There are many terms when it comes to variables, functions and function blocks. These terms are being used differently and even wrongly. The following figure shall explain the terms. This is necessary to make sure that a uniform understanding about the terms within this document is achieved.

Figure 2-1
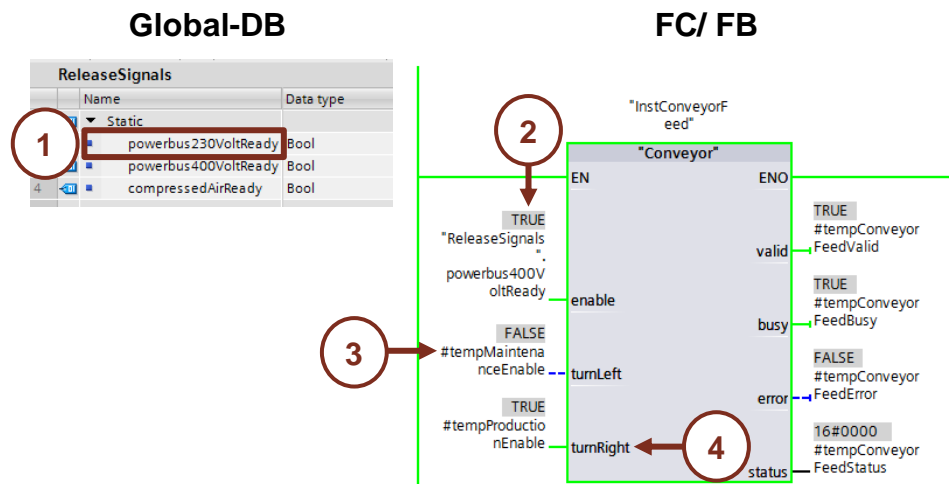
Table 2-3

	Term	Description
1.	variable	Variables are being declared by an identifier and allocate memory at a specific address within the controller. Variables are always defined with a specific data type (Boolean, Integer, etc.): • PLC variables or user constants • Variables or constants in blocks • Variables of Structures ("STRUCT") and PLC data types • Data blocks/ Instance data blocks • Technology objects
2.	current values actual values	Current values are the values, which are stored within a variable (e.g. 15 as value of an Integer variable)
3.	actual parameter	Actual parameter are the variables, which are connected to the formal parameters of a block.
4.	formal parameter	Formal parameters are the variables which are declared in the interface of a block. They are used for calls within a program. Formal parameters are often referred to as "interface parameters" or "transfer parameters". However these terms should be avoided.

2.6 Terms used with variables and parameters

The block interface consists of two parts, the formal parameters and the local data.

Formal parameter

Table 2-4

Type	Section	Function
Input parameter	Input	Parameter, which the block reads values from.
Output parameter	Output	Parameter, which the blocks writes values to.
Input/ Output parameter	InOut	Parameter, which the block reads values from, processes the values and writes the processed values back into the same parameter.
Return value	Return	Value, which the block returns to its caller.

Local data

Table 2-5

Type	Section	Function
Temporary variables	Temp	Variables to store intermediate values.
Static variables	Static	Variables to store persistent/ static intermediate values into the instance data block.
Constants	Constant	Constant values with a symbolic identifier to be used within a single block.

3 Settings in TIA Portal

In this chapter rules and recommendations for the initial setup of the programming environment are described.

Note The rules and recommendations for the settings in TIA Portal listed here are stored in the TIA Portal Settings File (tps file). You can find the tps file as a separate download in this entry. To apply the settings, you can import the tps file into TIA Portal.

ES001 Rule: User Interface Language "English"

The User Interface Language is to be set to "English". In this way all newly created projects have the editing and reference language as well as all the system constants are set to English.

Justification: To have all system constants available in the same language the user interface language must be set to a common uniform language.

ES002 Rule: Mnemonic "International"

The mnemonic (language setting for the programming language) shall be set to "International".

Justification: All the system languages and system parameter are set system independent. This enables a seamless cooperation between the programmers in the team.

Figure 3-1

ES003 Recommendation: Non-proportional font in editors

For the editors it is recommended to use a non-proportional font (monospace font). All characters have the same width and the presentation of codes, words and indentation is uniform.

The recommended setting is "Consolas" with a font size of 10 pt.

Justification: In contrast to "Courier New" with "Consolas" the focus is set to the differentiation between similar characters. It was specifically designed for the programming environments.

Figure 3-2
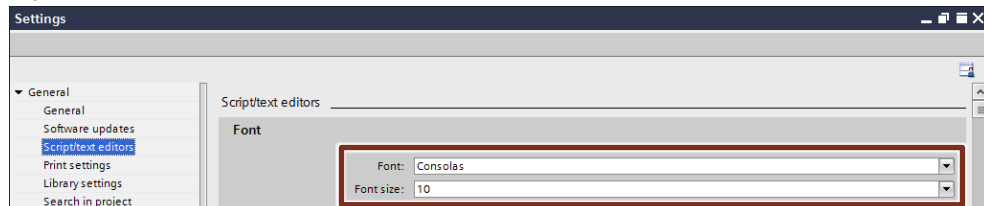
2.6 Terms used with variables and parameters

Figure 3-3
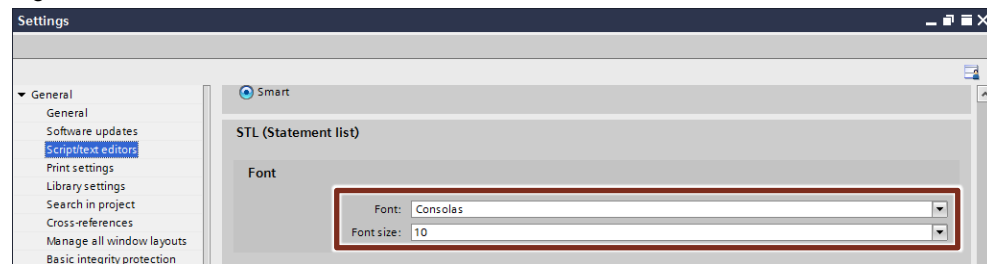
Figure 3-4
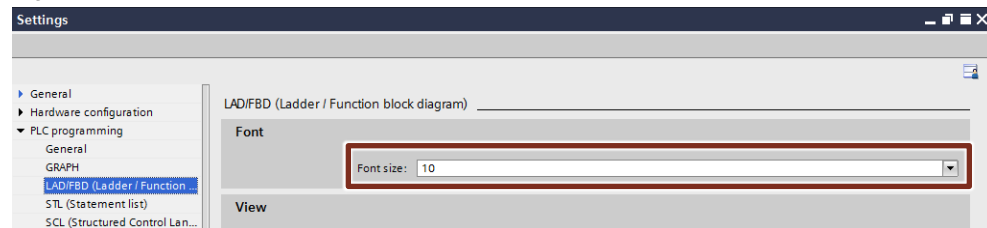
ES004 Rule: Smart Indentation with two whitespaces

For the indentation of instructions two whitespaces are used. The option "Indent" is to set to "Smart". Tabulators are no permitted in text-based editors, as their width is interpreted and displayed differently.

Justification: With this setting a uniform presentation is provided, even with different editors.

Figure 3-5
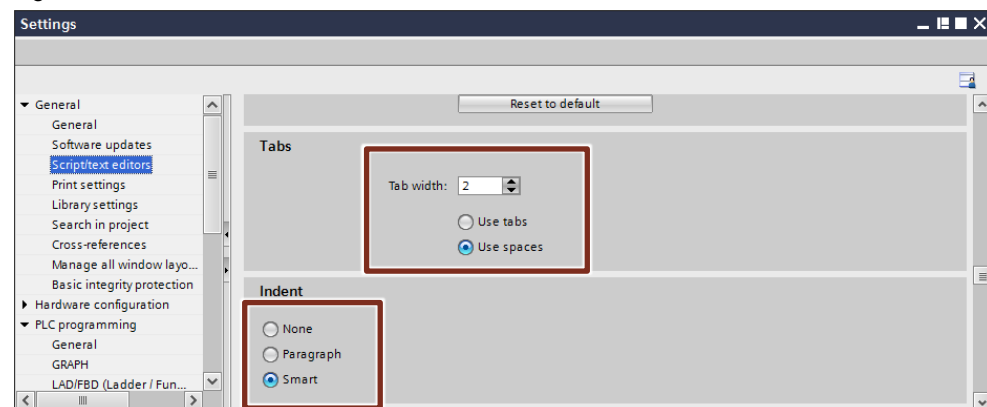
ES005 Rule: Symbolic representation of operands

The representation of operands is set to "Symbolic"

Justification: The programming is fully symbolic.

Figure 3-6

2.6 Terms used with variables and parameters
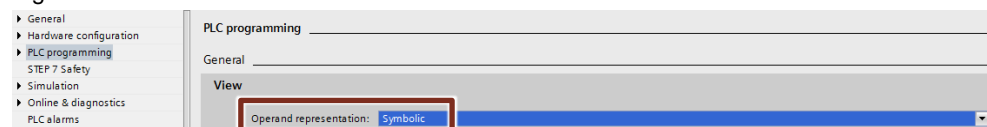
ES006 Rule: IEC conformant programming

To comply with IEC programming, the IEC check is turned on by default for every new block.

Justification: With this setting turned on every new block has the IEC check turned on. This in turn ensures a type safe usage of variables.

Figure 3-7
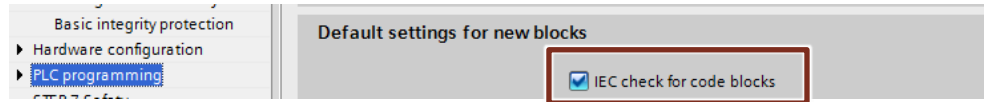
ES007 Rule: Explicit data access via HMI/ OPC UA/ Web API

Disabling accessibility and writability from HMI/ OPC UA/ Web API for interfaces limits the access to internal data for external applications.

Justification: External applications should only be able to access internal data, when explicitly enabled.

Figure 3-8
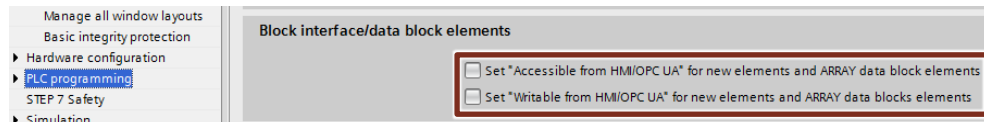
Figure 3-9
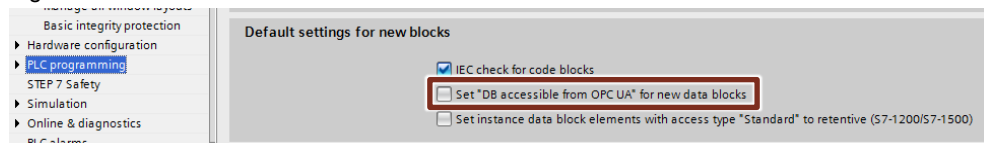
ES008 Rule: Automatic value evaluation (ENO) enabled

For the automatic evaluation of type defined value boundaries and their operations the EN/ ENO mechanism is responsible. This mechanism is turned on by default

Justification: With this setting to be active the evaluations are being executed by the system, refer also to "SE003 Rule: Handle ENO".

ES009 Rule: Automatic evaluation of Array boundaries

The automatic evaluation of Array boundaries must be turned on.

Justification: Staying within the boundaries of an Array is already evaluated at compile time. An out of bounds access can be avoided this way.

4 Globalization

This chapter describes the rules and recommendations for a global cooperation.

GL001 Rule: Use consistent language

The language used must be consistent in the PLC as well as in the HMI programming. This means, that English texts can only be found in the English language setting.

GL002 Rule: Set editing and reference language to "English (US)"

If not otherwise demanded by the customer, the language must be set to "English (United States)" for both the Editing and Reference language. The complete program including all comments must be created in English.

Figure 4-1

2.6 Terms used with variables and parameters
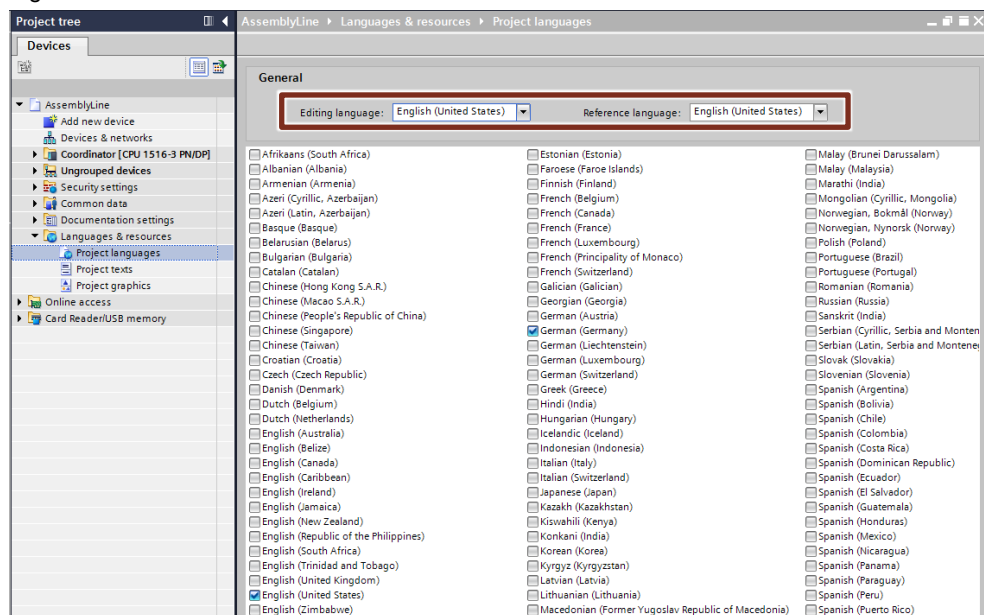
GL003 Rule: Supply texts in all project languages

All project texts must be provided at least in English as well as in all other used project languages.

Note	In a block editor the texts and their translations can be easily managed in the tab "Texts".

Figure 4-2
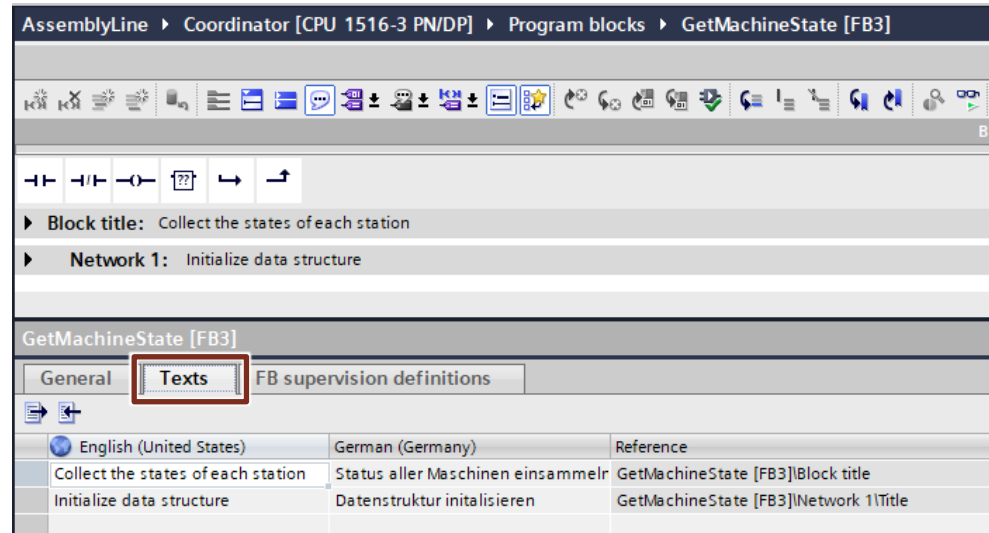
5 Nomenclature and Formatting

This chapter describes rules and recommendations for naming and writing of programs and comments.

NF001 Rule: Unique and consistent English identifiers

The name of an identifier (Blocks, variable, etc.) must be in English language (English – United States). The name describes the meaning of the identifier in the context of the source code and therefore promotes an understanding of the functionality and usage of the identifier.

- The chosen spelling of an identifier must be maintained in all blocks and PLC data types and shall be as short as possible.

- The same functional meaning of an identifier causes the same naming for the identifier. This applies to capitalization as well.

- Identifier names can be assembled out of multiple words; the order of the words has to be the same as in the spoken language.

- Functions and Function block identifiers shall start with a verb, e.g. "Get", "Set", "Put", "Find", "Search", "Calc".

- Is the identifier a name for an Array, then the name uses the plural. Non-countable nouns remain in their singular form ("data", "information", "content", "management").

- Static and temporary Boolean variables are often state indicating variables. In such cases names starting with "is", "can" or "has" can be understood the easiest.

Justification: A quick overview about the program and its inputs and outputs will be provided.

| Note | The names assigned by TIA Portal are place holders and need to be replaced by yours. |

Table 5-1

	Correct naming	**Incorrect naming**
for Arrays	data beltConveyors	datas beltConveyor
For static and temporary Boolean variables, which indicate a state	isConnected canScan	connected scan
For other Boolean variables	enable	setEnable
For Functions/ Function blocks	GetMachineState SearchDevices	MachineStateFC FB_Device

2.6 Terms used with variables and parameters

NF002 Rule: Use meaningful comments and properties

Comment and property fields shall be used and filled with meaningful comments and information. This includes

- Block title and block comments (refer also to "NF003 Rule: Document developer information")

- Block interfaces

- Network title and network comments

- Blocks and their variables and constants

- PLC data types and their variables

- PLC tag tables, PLC tags and user constants

- PLC alarm text lists

- PLC supervision & alarms

- Library properties

Justification: Using this the user gets the most information and guidance in using the components, e.g. through tooltips.

Figure 5-1

Note	Additionally, the block description and documentation can be provided to TIA Portal as document (e.g. *.pdf, *.html) The user can open this documentation by pressing <Shift> <F1> as part of the Online help.
	Further information is contained in the Online help using the keyword "providing user-defined documentation":
	https://support.industry.siemens.com/cs/ww/en/view/109773506/119453612171

2.6 Terms used with variables and parameters

NF003 Rule: Document developer information

Each block contains a block header in the program code (SCL/ ST) or in the block comment (LAD, FBD). Herein the most important information for the block development must be documented. Due to the placement inside the program, the development relevant information will be hidden in knowhow protected blocks.

User relevant information must be provided in the block properties. This information is available to the user even in knowhow protected blocks.

The following template for such a block header contains the elements from the block properties as well as the development relevant information, which don't need to be copied into the properties.

The description contains the following items:

- (Optional) Company name / (C) Copyright (Year). All rights reserved.

- Title/ Block description

- Description of the functionality

- (Optional) Name of the library

- Department/ Author/ Contact

- Target system – PLCs with firmware version (z. B. 1516-3 PN/DP V2.6)

- Engineering – TIA Portal with version at time of creation/ modification

- Limitations for usage (e.g. certain OB types)

- Requirements (e.g. additional hardware)

- (Optional) additional information

- (Optional) change log with version, date, author and change description (with Safety blocks incl. Safety signature)

Template for a block header in LAD/ KOP and FBD/ FUP:

```
(company) / (C) Copyright (year)
------------------------------------------------------------------------------
Title:          (Title of this block)
Comment/Function: (that is implemented in the block)
Library/Family:  (that the source is dedicated to)
Author:          (department / person in charge / contact)
Target System:   (test system with FW version)
Engineering:     TIA Portal (SW version)
Restrictions:    (OB types, etc.)
Requirements:    (hardware, technological package, etc.)
------------------------------------------------------------------------------
Change log table:
Version     | Date       | Signature  | Expert in charge | Changes applied
------------|------------|------------|------------------|----------------------
001.000.000 | yyyy-mm-dd | 0x47F78CC1 | (name of expert) | First released version
```

2.6 Terms used with variables and parameters

Template for a block header in SCL:

```
REGION Description header
  //================================================================================
  // (company) / (C) Copyright (year)
  //--------------------------------------------------------------------------------
  // Title:           (Title of this block)
  // Comment/Function: (that is implemented in the block)
  // Library/Family:   (that the source is dedicated to)
  // Author:           (department / person in charge / contact)
  // Target System:    (test system with FW version)
  // Engineering:      TIA Portal (SW version)
  // Restrictions:     (OB types, etc.)
  // Requirements:     (hardware, technological package, etc.)
  //--------------------------------------------------------------------------------
  // Change log table:
  // Version    | Date       | Expert in charge | Changes applied
  //------------|------------|------------------|--------------------------------
  // 001.000.000 | yyyy-mm-dd | (name of expert) | First released version
  //================================================================================
END_REGION
```

NF004 Rule: Comply with prefixes and structure for libraries

The identifier of a library has the prefix "L" and does not exceed a maximum length of eight characters

The identifier of a library starts with the prefix "L" and is followed by a maximum of seven characters as the name (e.g. LGF, LCom). "L" stand for the word Library. After the library identifier an underscore (_) is used as a separator (e.g. LGF_).

The maximum length of an identifier for libraries (incl. prefix) is limited to eight characters.

Justification: This limitation serves the purpose of assigning compact and short names.

Every element in the library carries the prefix.

All types and master copies contained in the library get the identifier of the library.

An element, which only demonstrates the use of the library, is not a library element in the sense of a standardized library, it is rather an example and therefore doesn't necessarily carry the library prefix.

Justification: With the prefix included in the identifier, naming collisions are being avoided.

Table 5-2

Type	Identifier according to style guide
Library, main folder of the library	LExample
PLC data type	LExample_type<Name>
Function block	LExample_<Name>
Function	LExample_<Name>
global data block	LExample_<Name>
Organization block	LExample_<Name>
PLC symbol table	LExample_<Name>
Global constant	LEXAMPLE_<NAME>

2.6 Terms used with variables and parameters

Type	Identifier according to style guide
Global constant for errors	LEXAMPLE_ERROR_<NAME>
	LEXAMPLE_ERR_<NAME>
Global constant for warnings	LEXAMPLE_WARNING_<NAME>
	LEXAMPLE_WARN_<NAME>
PLC alarm message text list	LExample_<Name>

Grouping within the library

All master copies and types shall be placed in a subfolder inside the library, which carries the library identifier as its folder name.

Justification: The subfolder supports the project harmonization efforts and allows grouping of multiple libraries within a project.

Figure 5-2
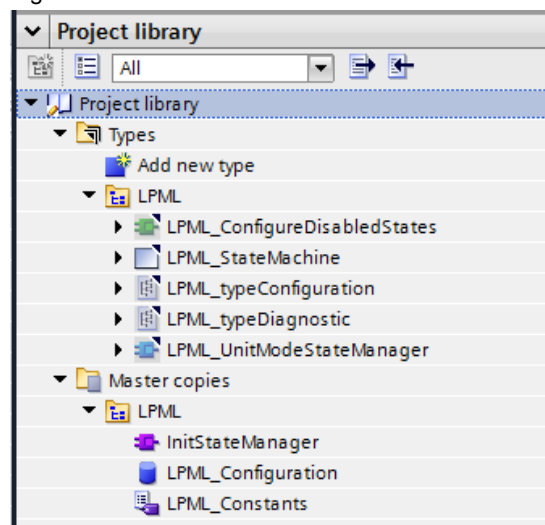
Note	This rule contains only information for the nomenclature of library elements. Additionally, it is recommend to follow the recommendations given and explained in detail in the library guideline available:
	https://support.industry.siemens.com/cs/ww/en/view/109747503

NF005 Rule: Use PascalCasing for objects

Identifiers for TIA Portal objects, such as:

- Blocks
- Software Units, technology objects, libraries, projects
- PLC tag tables
- PLC alarm text lists
- Watch and force tables
- Traces and measurements

are written using PascalCasing.

2.6 Terms used with variables and parameters

The following rules apply for PascalCasing:

- The first character is a capital letter
- If an identifier is assembled out of multiple words, then the first character of each word is a capital letter.
- There are no separators (e.g. hyphen or underscore) used for the optical separation of the identifier. For structuring and specialization purposes the sparingly use of the underscore (not more than three) is permitted.

Table 5-3

Sparingly	Excessive
GetAxisData_PosAxis GetAxisData_SpeedAxis GetAxisData_SyncAxis	Get_Axis_Data_Pos_Axis

NF006 Rule: Use camelCasing for code elements

Identifiers for code elements, such as

- Variables
- PLC data types
- Structures ("STRUCT")
- PLC tags
- Parameters

are written using camelCasing.

The following rules apply for the camelCasing:

- The first character is a non-capitalized (small) letter
- If an identifier is assembled out of multiple words, then the first character of the following word is capitalized.
- The use of separators (e.g. hyphen or underscores) for the optical separation is not permitted.

Figure 5-3

2.6 Terms used with variables and parameters

NF007 Rule: Use prefixes

No prefix for formal parameters

Formal parameters of blocks are used without prefixes. When passing PLC data types, the identifiers also do not carry a prefix.

Temporary and static variables carry a prefix "temp" or "stat"

To distinguish temporary variables from formal parameters in the code, the prefixes defined in Table 5-4 shall be used.

Justification: this measure makes it easier for the programmer to distinguish between formal parameters and local data. With this prefixing in place the access to a variable can be easily defined and recognized.

Note	The static variables inside the global DBs and Array DBs do not have the prefix "stat".

Instance data with prefix "inst" or "Inst"

Single instances as well as multi-instances and parameter instances get a prefix. Single instances get the prefix "Inst", whereas multi-instances and parameter instances get the prefix "inst".

Justification: With the prefixes in place it can be easier recognized, whether an (invalid) access to instance data has been made.

PLC data type with prefix "type"

A PLC data type gets the prefix "type". The individual elements inside the PLC data type do not get a prefix.

2.6 Terms used with variables and parameters

Table 5-4

Prefix	Type
No prefix	**Input and Output parameter** Access possible from the outside ➔ enable ➔ error
No prefix	**InOut parameter** Modifications of the assigned data possible by the user as well as by the block at any time. ➔ conveyorAxis
No prefix	**PLC tags and user constants** ➔ lightBarrierLeft (identifier for a PLC variable) ➔ MAX_BELTS (identifier for a user constant)
No prefix	**Global data blocks** Neither global DBs nor the contained elements get a prefix ➔ ReleaseSignals (identifier of a global DB) ➔ powerBusReady (identifier of a variable in a global DB)
temp	**Temporary Variables** No access to local data from the outside possible ➔ tempIndex
stat	**Static variables** No access to local data from the outside permitted ➔ statState
inst	**Variables of multi-instances and parameter instances** ➔ instWatchdogTimer ➔ instWatchdogTimer**s** (with Arrays of instances)
Inst	**Single instance data blocks** ➔ InstConveyorFeed
type	**PLC data type** Only the data type gets the prefix, the elements do not get a prefix ➔ typeDiagnostic (identifier for PLC data type) ➔ stateNumber (identifier for a variable inside the PLC data type)

NF008 Rule: Write identifier of constants in CAPITALS

The names of constants (global and local constants) are written completely with capital letters (UPPER_CASING). For separation and recognition purposes of individual words or abbreviations an underscore between the words or abbreviations shall be used.

Figure 5-4: constants in a FB

		Constant			
		MAX_VELOCITIY	Real	10.0	MAximum Velocity of conveyor
		MAX_NO_OF_AXES	DInt	3	Maximum number of axes

Note	"TRUE" and "FALSE" are also constants.

2.6 Terms used with variables and parameters

NF009 Rule: Limit the character set for identifiers

For all object and code identifiers the Latin alphabet (a-z, A-Z) and the Arabic numerals (0-9) as well as the underscore (_) are to be used exclusively.

Table 5-5

Correct naming	Incorrect naming
tempMaxLength	temporary Variable 1

NF010 Recommendation: Limit the length of identifiers

The overall length of an identifier incl. prefix, suffix or library identifier shall not exceed 24 characters.

Justification: Since variable names in structures are assembled out of many identifiers, the length of the identifier at the code location will become excessively long anyway.

Example:

```
instFeedConveyor024.releaseTransportSect1.gappingTimeLeft
```

NF011 Recommendation: Use one abbreviation per identifier only

Multiple abbreviations shall not be used directly one after the other to realize the best possible readability. To reduce the amount of used characters in an identifier recommended abbreviations are listed in Table 5-6.

This table only contains the most commonly used abbreviations. The spelling of the abbreviations must follow the rules for the particular use and needs to be adopted accordingly (capitalization).

Table 5-6

Abbrev.	Type
Min	Minimum
Max	Maximum
Act	Actual, Current
Next	Next value
Prev	Previous value
Avg	Average
Sum	Total sum
Diff	Difference
Cnt	Count
Len	Length
Pos	Position
Ris	Rising edge
Fal	Falling edge
Old	Old value (e. g. for edge detection)
Sim	Simulated
Dir	Direction
Err	Error
Warn	Warning
Cmd	Command
Addr	Address

2.6 Terms used with variables and parameters

NF012 Rule: Initialize in the appropriate format

The initialization (assignment of constant data) of variables shall be done in the appropriate format of the data type. This means that a WORD typed variable shall be initialized with 16#0001 instead of 16#01.

Initialization done in code shall use local symbolic constants, refer also to "RU005 Rule: Use local symbolic constants".

Table 5-7

Correct initialization			Incorrect initialization		
statTriggerOld	Bool	FALSE	statTriggerOld	Bool	false
statStatus	Word	16#7000	statStatus	Word	123
statStep	DInt	101	statStep	DInt	16#0
statVelocity	LReal	0.0	statVelocity	LReal	16#000
statCommand	Byte	16#01	statCommand	Byte	16#1
statFlags	Byte	2#1010_0101	statFlags	Byte	25

NF013 Recommendation: Hide optional formal parameters

Hide formal parameters, which are optional.

Justification: This way the block call can be reduced to the necessary minimum by collapse the optional formal parameters.

Figure 5-5

2.6 Terms used with variables and parameters
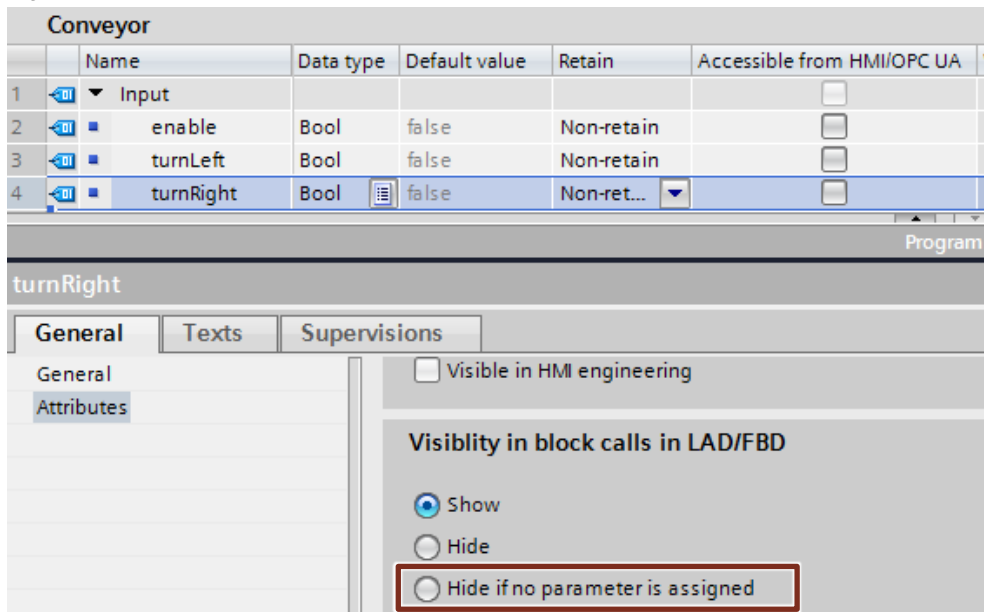
NF014 Rule: Format SCL code meaningfully

It is recommended to use the Auto Format function of TIA Portal. The advantage of this is, that all users work with the same formatting. Indentation is done automatically.

Figure 5-6
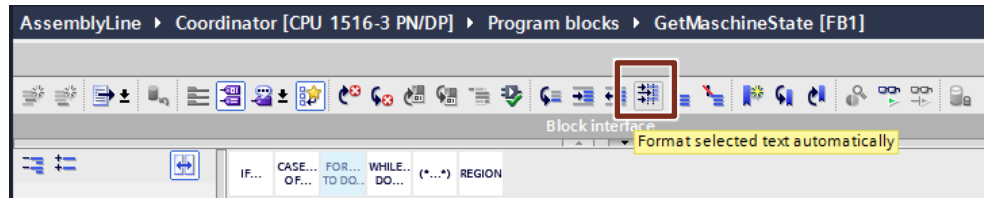
Use of line comment "//" only

There are two different types of comments:

- Block comments "(*…*)" or multilanguage block comments "(/*…*/)" can span over multiple lines. It describes a function or a code fragment.

- A line comment "//" describes a single line of code and is located at the end of the code line or in front of it.

To allow for easy disabling of code fragments for debugging purposes only line comments "//" are permitted.

A comment provides information to the reader, why something has been done at this point in the code. The comment must not contain the code in redundant clear text – this means, it should not describe what is done – this is being described already by the code itself – instead the reason why something has been done should be described.

| Note | You may use the button "comment". This way you don't have to type in the comment signs manually.

The engineering system supports this via menus, to comment selected blocks of texts or the remove the comment signs. Additionally, you can avoid syntax problems with nested comment blocks due to using "(*...*)" or "(/*...*/)". |

Figure 5-7
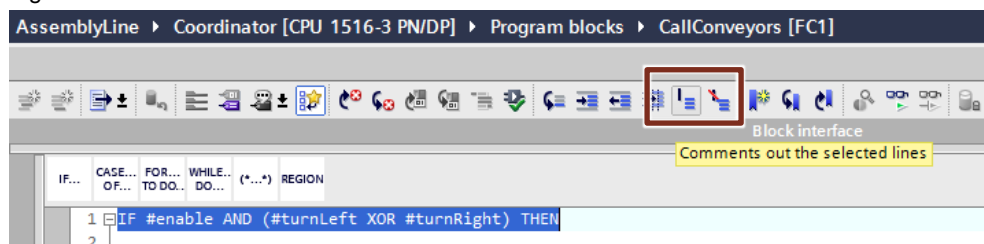
Whitespaces in front of and behind operators

In front of and behind an operator a whitespace has to be used.

2.6 Terms used with variables and parameters

Expressions are always placed into parenthesis

To support the order of interpretation expressions are put into parenthesis in the desired interpretation order.

Example:

```
#tempSetFlag := (#tempPositionAct < #MIN_POS)
                OR (#tempPositionAct > #MAX_POS);
```

Condition and instructions are separated with a line break

A clear separation must be created between condition and instruction. This means, that after a condition (e.g. THEN) or after an alternative branch (e.g. ELSE) a line break must be used before an instruction is programmed. This rule applies in a similar way to the conditions of the other constructs (e.g. CASE, FOR, WHILE, REPEAT).

Example:

```
IF #isConnected THEN // Comment
  ; // Statement section IF
ELSE
  ; // Statement section ELSE
END_IF;
```

Line breaks in partial conditions

In more complex conditions it is helpful to put each partial condition into its own line. Operators are being put in front of the new partial condition.

Example:

```
#tempResult := (#enable AND #tempEnableOld)
               OR (#enable AND #isValid
                 AND NOT (#hasError OR #hasWarning)
               );
```

2.6 Terms used with variables and parameters

Proper indentation of conditions and instructions

Each instruction in the body of a control structure must be indented. If a single line is not enough, Boolean expressions will be continued on the next line.

Multiline conditions in IF statements are indented by two whitespaces. The THEN follows on a new line at the same indentation level as the IF. When the IF condition fits onto a single line, the THEN can be put at the end of the same line. In case the nesting depth is deeper, the THEN instruction will be put on its own line. A single closing bracket indicates the end of a nested condition. Operands are always at the beginning of the line.

These rules apply in a similar way for the conditions of the other control structures (e.g. CASE, FOR, WHILE, REPEAT).

Examples:

```
IF #enable // Comment
  AND (
  (#turnLeft XOR #turnRight)
    OR (#statIsMaintenance AND #statIsManualMode)
  ) // Comment
  AND #tempIsConnected
THEN
  ; // Statement
ELSE
  ; // Statement
END_IF;
```

```
IF #enable THEN
  ; // Statement
  IF #tempIsReleased THEN
    ; // Statement
  END_IF;
ELSE
  ; // Statement
END_IF;
```

6 Reusability

This chapter describes the rules and recommendations applicable to ensure the multiple use of program elements.

RU001 Rule: Provide blocks which can be simulated

Activate the simulation capabilities via the project properties.

Justification: With this the blocks can be used completely in a simulated environment.

Note	Please be aware that the knowhow protection in a simulated environment could be weakened.

Figure 6-1
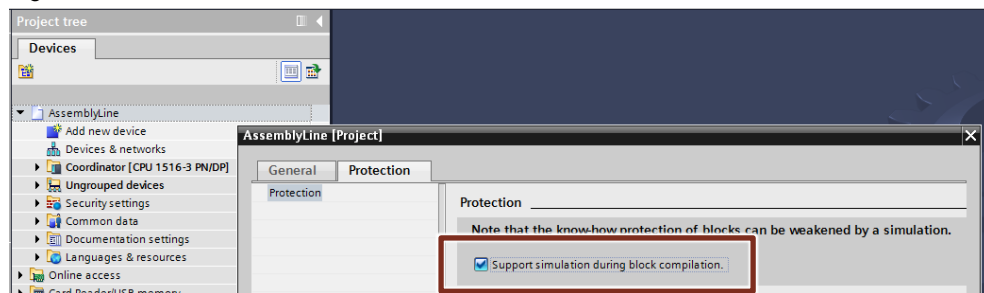
RU002 Rule: Version entirely with libraries

Assign versions entirely. This means, that every change in a block must be documented and the assigned version must be maintained. Every change in the assigned version must be documented in their respective locations, such as the block header of a LAD block.

When using a library and block types the block version is managed by TIA Portal. In this case it is not necessary to manually maintain the version in the block properties. The change log remains untouched by this fact.

An upgrade of the library to the latest TIA Portal version does not require a change in the block and is therefore not a new version.

| Note | Before a block can be inserted into a library, all necessary settings, such as auto numbering, knowhow protection, simulation capabilities (via project properties) need to be done. Once the block is part of the library, the mentioned settings above are difficult to change afterwards.

The property "published" in the context of software units may be adjusted later without modifying the type. |
|------|------|

2.6 Terms used with variables and parameters

Version numbers and their use

The first released version always starts with 1.0.0 (refer to Table 6-1).

The first digit describes the left most number.

The third digit in software versioning indicates changes, which have no effect on function or documentation, such as pure bug fixing.

Extension to the functionality the second digit will be incremented, and the third digit reset.

With a new major release, containing new functionality and incompatible changes to the previous version, increase the first digit and resets the second and third digit.

Each digit has a valid range between 0 and 999.

Table 6-1

Library	FB1	FB2	FC1	FC2	Comment
1.0.0	1.0.0	1.0.0	1.0.0		Released version
1.0.1	1.0.1	1.0.0	1.0.0		Bug fix in FB1
1.0.2	1.0.1	1.0.1	1.0.0		Optimization of FB2
1.1.0	1.1.0	1.0.1	1.0.0		Extension to FB1
1.2.0	1.2.0	1.0.1	1.0.0		Extension to FB1
2.0.0	2.0.0	1.0.1	2.0.0		New and possibly incompatible function in FB1 and FC1
2.0.1	2.0.0	1.0.2	2.0.0		Bug fix in FB2
2.1.0 / 3.0.0	2.0.0	1.0.2	2.0.0	1.0.0	New function in FC2/ possibly larger new release

Use of type concept for FC, FB and PLC data types

Reusable functions, function blocks and PLC data types, which cannot be changed by the user are provided as types in a library.

Justification: Only this way it is possible to fully benefit from the type concept of TIA Portal.

Note	This rule contains only generic information about versioning. A detailed explanation about the automatic versioning of library elements is provided in the library guideline:
	https://support.industry.siemens.com/cs/ww/en/view/109747503

RU003 Rule: Keep only released types in released projects

Finalized projects contain only typified library elements, which are not in status "in test":

- Blocks (only functions and function blocks)
- PLC data types

2.6 Terms used with variables and parameters

RU004 Rule: Use only local variables

Within a reusable block only local variables may be used. Accessing global data from within the FC or FB is not permitted. Global data may be passed in via the formal parameters of the block interface.

Passing global data into a FC is possible for:

- Accesses to global DBs and the use of single instance DBs
- Use of global timers and counters
- Use of global constants
- Access to PLC tags

When the mentioned requirements above are fulfilled, TIA Portal indicates this automatically in the block header with a status "The object is library-conformant". This status can therefore be easily used to verify the compliance with this rule.

Figure 6-2
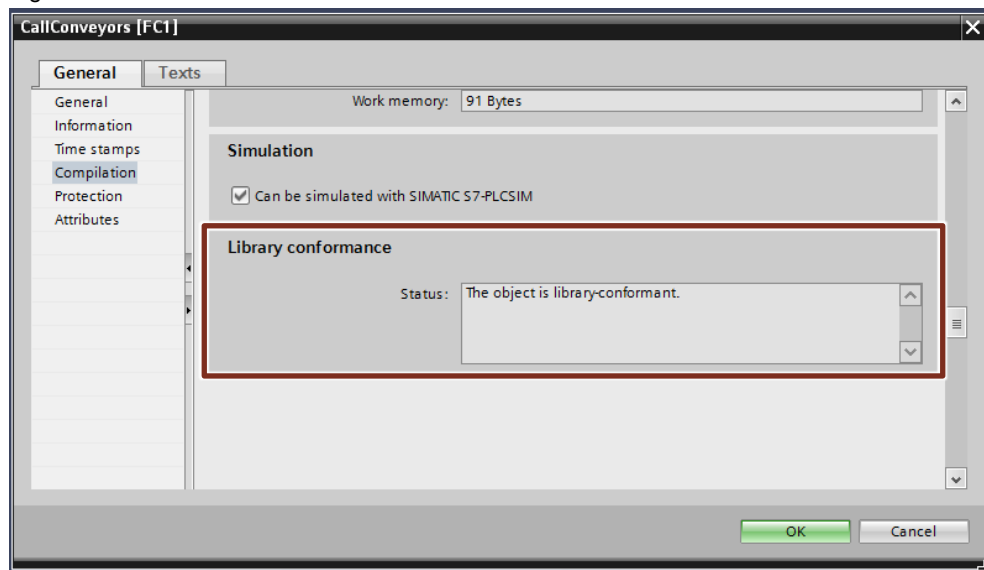
RU005 Rule: Use local symbolic constants

To further encapsulate a block local constant shall be used. When global constants need to be used, they must be passed into the block via the formal parameters of the block interface. Global constants shall be defined in their own PLC tag table.

Note	When using a global constant in a block a change of its value requires a recompile of that block. With knowhow protected blocks this requires the knowledge of the assigned password.

No "magic numbers"

When a variable in the code is being compared to or is being assigned a value different from 0 (Integer), 0.0 (Real/ LReal), TRUE or FALSE a symbolic constant shall be used for this.

Justification: An adjustment of the value is much easier this way as this is centrally in the block header instead of multiple places in the code.

2.6 Terms used with variables and parameters

Note	Constants are textual replacements for numerical values, which are replaced by the preprocessor. Therefore, the use has no negative impacts on performance or memory consumption in the PLC.

It is possible to assign different symbolic constants to the same value (incl. equal to 0).

Example:

```
STATUS_DONE             WORD    16#0000

STANDSTILL_SPEED        LREAL   0.0

FREEZING_TEMPERATURE    LREAL   0.0
```

Furthermore, the readability is increased as a symbolic identifier is much easier to understand than a number.

Example:

Figure 6-3

		Name	Data type	Default value	Retain
1	▼	Input			
2	■	velocity	Real	0.0	Non-retain
3	▶	Output			
4	▶	InOut			
5	▼	Static			
6	■	statVelocity	Real	0.0	Non-retain
7	▶	Temp			
8	▼	Constant			
9	■	MAX_VELOCITY	Real	10.0	

```
IF (#velocity < #MAX_VELOCITY) THEN
  #statVelocity := #velocity;
ELSE
  #statVelocity := #MAX_VELOCITY;
END_IF;
```

RU006 Rule: Program fully symbolic

The programming is done fully symbolic. This means there are no physical addresses, such as with ANY pointer, used in the program.

Justification: This increases the readability and maintainability due to the symbols used.

Note	The alternative to ANY pointer is the VARIANT data type, optionally a REF_TO reference. The data type VARIANT detects type errors early on and offers a symbolic addressing.

2.6 Terms used with variables and parameters

RU007 Recommendation: Program independently from hardware

To guarantee compatibility between the different systems it is recommended to use exclusively hardware independent data types.

The use of global memory flags and system memory flags is not permitted to support reusability and hardware independency.

This includes the system provided timers and counters, such as the S5-Timer. Instead of these data types the use of the IEC conformant types, e.g. IEC_Timer is encouraged, which can also be used in multi-instances.

The storage of data needed in the whole user program shall be done in a global data block.

Note	A comparison table of the system functions for the hardware independent programming is available in the Siemens Industry Online Support at the following entry: SIMATIC S7-1200/ S7-1500 comparison table for programming languages https://support.industry.siemens.com/cs/ww/en/view/86630375

RU008 Recommendation: Use templates

Using templates, you can achieve a uniform basis for all programmers. The functionality provided by the templates can be considered validated and reduce the development times dramatically.

Another positive aspect is the easier usability and the improved perception due to the uniform block basis as the blocks provide the same base functionality. As an example, refer to the PLCopen standard.

Note	The referred to templates can be found as master copies in the library of general functions (LGF): https://support.industry.siemens.com/cs/ww/en/view/109479728

7 Referencing objects (Allocation)

This chapter describes rules and recommendations for the memory management and the access.

AL001 Rule: Use multi-instances instead of single instances

In the program multi-instances shall be preferably used instead of single instances.

Justification: With this method the creation of encapsulated modules in the form of a function block becomes possible. No additional instances in upper level structures or global structures become necessary, thus reducing the number of objects.

AL002 Recommendation: Define array boundary from 0 to a constant value

Array boundaries start at 0 and end with a symbolic constant as its upper boundaries of the array.

- For arrays inside a block the constant must be defined in the local data of the block interface.

- For an array in global DBs and in PLC data types the constant used as the upper limit must be defined in a PLC tag table

- As the data type of the constant as well as for the index used to access the array elements DINT shall be used for performance reasons.

Example:

```
BUFFER_UPPER_LIMIT    DINT   10


diagnostics           Array[0..BUFFER_UPPER_LIMIT] of typeDiagnostics
```

Justification: Beginning the array index at 0 has several benefits as some system instructions and mathematical operations work zero-based, e.g. modulo function. This way the index can directly be used in such functions without any adjustments.

Another benefit is, that WinCC (Comfort, Advanced, Professional and Unified) can handle zero based Arrays, e.g. in their scripting.

In the case that the Array boundaries cannot be zero based, then a symbolic constant should be used for both the upper and the lower limit.

AL003 Recommendation: Declare array parameter as ARRAY[*]

If an Array must be passed in as a formal parameter, it is recommended to pass it in as an Array of an unspecified size.

The size and the limits can be determined with the system functions "UPPER_BOUND" and "LOWER_BOUND".

Example:

```
diagnostics           Array[*] of typeDiagnostics
```

Justification: Doing this enables the creation of generic program structures. Especially in knowhow protected blocks a recompile is not necessary as the size is not declared explicitly in the block interface.

2.6 Terms used with variables and parameters

AL004 Recommendation: Specify the required string length

"String" and "WString" always reserve the memory required to store 254 characters. A "String" can contain up to 254 characters, a "WString" can contain up to 16382 characters. It is recommended to limit all strings to the necessary length provided as symbolic constant.

Justification: This procedure prevents the system from allocating excessive memory. Besides that, it provides performance benefits, when passing in the strings per formal parameter assignment.

Example:

```
MAX_MESSAGE_LENGTH      DINT    24


errorMessage            String[#MAX_MESSAGE_LENGTH]
```

8 Security

This chapter describes the rules and recommendations applicable to create an as robust and secure program as possible.

SE001 Rule: Validate actual values

All actual values that are passed in shall be validated to avoid uncontrolled and unexpected program execution and states.

In case of implausible or invalid actual values an indication must be provided to the user, refer also to "DA013 Rule: Report status/ errors via "status"/ "error"".

SE002 Rule: Initialize temporary variables

Every temporary variable used in the block must be initialized before its first use. The initialization is a direct assignment of either an operation result or a constant in its usual presentation for the data type (literal).

Refer also to "NF012 Rule: Initialize in the appropriate format"

Justification: As only elementary data types are initialized by the system, all others have an undefined value, which can cause unexpected program behavior.

Note	Using variables with technological blocks, values less than 0.0 have a special meaning. Depending on the formal parameter instead of the variables value the default value preconfigured in the technological object will be used.
	That is why an appropriate initial value shall be chosen.

SE003 Rule: Handle ENO

With the help of the enable output ENO selected runtime errors can be detected. The execution of the following instructions depends on the signal state of the ENO.

Justification: The use of the EN/ ENO mechanism avoids unexpected program interruptions. The block status is passed on in the format of a Boolean variable.

To increase the execution performance of the PLC the automatic EN/ ENO mechanism can be deactivated. The result of this is, that there is no possibility to respond to runtime errors using the ENO value anymore. Enabling the following instructions must be realized manually.

Therefore, under all circumstances an assignment to ENO shall be present in the program. In its simplest form:

```
ENO := TRUE;
```

SE004 Rule: Enable data access via HMI/ OPC UA/ Web API selectively

Per default the access to variables per HMI/ OPC UA/ Web API shall be disabled. The access to static variables of a FB is not permitted. Variables for read or write access must be created.

In general, the accessibility via HMI/ OPC UA/ Web API must at least be activated in the editor for PLC data types to allow accesses. When using the PLC data type the access to it must be adjusted in the block interface accordingly.

2.6 Terms used with variables and parameters

SE005 Rule: Evaluate error codes

In case used FCs, FBs or system functions provide error codes to the program, they must be evaluated.

If the user program cannot handle a reported error, then a unique error must be provided to the user to allow error localization.

Note	Further information about the topic error handling are provided in "[DA013 Rule: Report status/ errors via "status"/ "error"]".

SE006 Rule: Write Error OB with evaluation logic

If organization blocks are used for error handling, then they fulfill a certain task.

The minimum requirement is to evaluate the error, the reporting, and handling of it in the user program.

9 Design guidelines/ architecture

This chapter describes the applicable rules and recommendations for program design and program architecture.

DA001 Rule: Structure and group a project/ library

Split your program into logical units. The system provides several different means for this matter.

- Group blocks belonging together into a group or folder

- Structure technological machine parts into Software Units

- Structure your program into logical functional units – FC/ FB

- Collate data belonging together into PLC data types

- Structure the program with networks or regions

Note	A REGION in SCL is comparable to a network rung in LAD/ FBD.
	The name of a region is comparable to the network title and shall be written as such.
	Regions provide several benefits:
	• An overview of all regions inside the editor on the left-hand side
	• Quick navigation through the code with the help of the overview the linking inside it.
	• The possibility of folding of code fragments
	• Quick collapse and expand with the help of the navigation through synchronization of overview and code

DA002 Recommendation: Use appropriate programming language

Use a programming language suitable for the programming task.

Standard blocks – structured text (SCL/ ST)

The preferred programming language for standard blocks is SCL. It provides the most compact form and best readability of all programming languages. It supports the programmer by auto marking all occurrences of a selected code elements.

Call environments – graphical/ block oriented (LAD, FBD)

In case several blocks shall be interconnected, e.g. in an OB as call environment, then the programming languages LAD or FBD serve best. Also, in the case that mostly binary logic is contained in the block, LAD or FBD should be used. In these cases, LAD and FBD allow an easier diagnosis and provide a faster overview for the service personnel.

Sequential control – flow oriented (GRAPH)

GRAPH is the preferred language, when it comes to programming of sequences. Using GRAPH sequential steps can be programmed fast and the execution can be easily followed. Additionally, "Interlocks" and "Supervisions" are already provided by the system.

2.6 Terms used with variables and parameters

DA003 Rule: Set/ evaluate block properties

The following settings must be activated in the block properties:

- **Auto numbering:** Blocks (OB, FC, FB, DB, TO) shall only be delivered with auto numbering turned on. Be aware that the execution of an Organization blocks depends on its number/ priority.

- **IEC Check:** To ensure the IEC conformant programming, the IEC check must be turned on. Only this way a type conformant and type safe programming can be ensured.

- **Optimized access:** For full symbolic programming and the maximum performance the optimized access to blocks must be activated.

In the block properties the following attributes shall be checked:

- **ENO:** Refer to "SE003 Rule: Handle ENO".

- **Multi-instance capability:** The use of a block as a multi-instance capable block is guaranteed, if this block internally uses multi-instances instead of global single instances.

- **Library conformance:** Refer to "RU004 Rule: Use only local variables"

Figure 9-1: Auto numbering

2.6 Terms used with variables and parameters
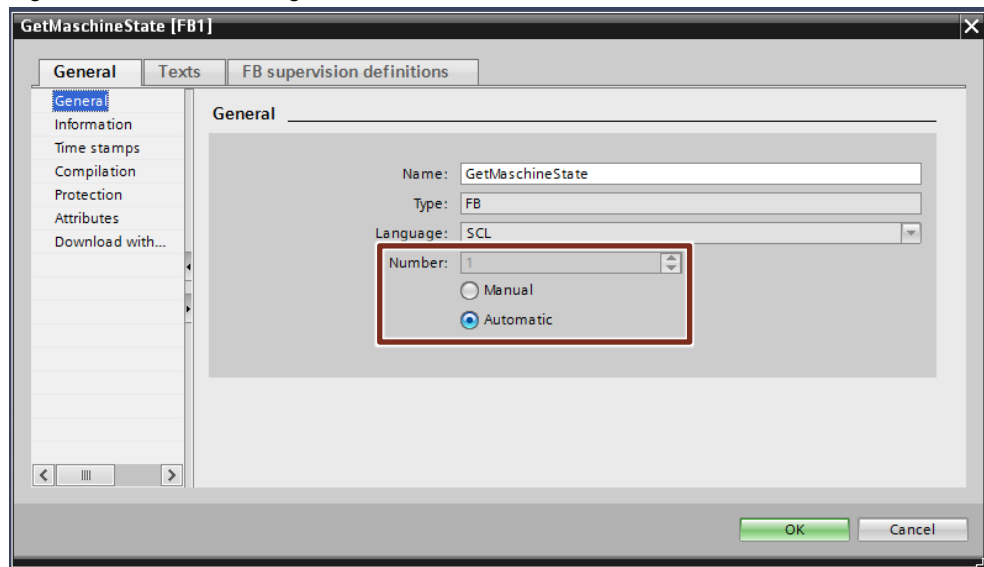
Figure 9-2: IEC Check, ENO, Optimized access, Multiple instance capability
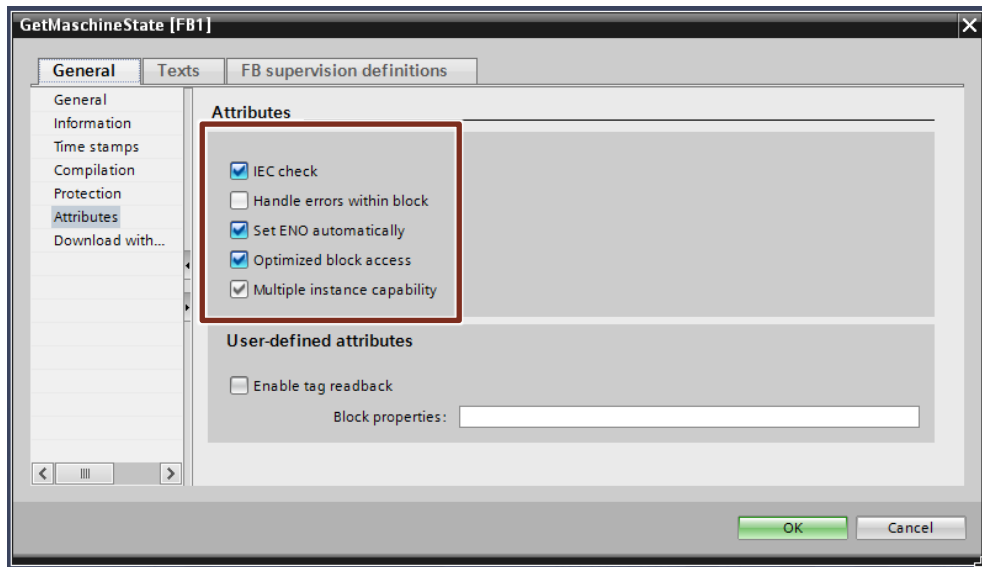
Figure 9-3: Library conformity
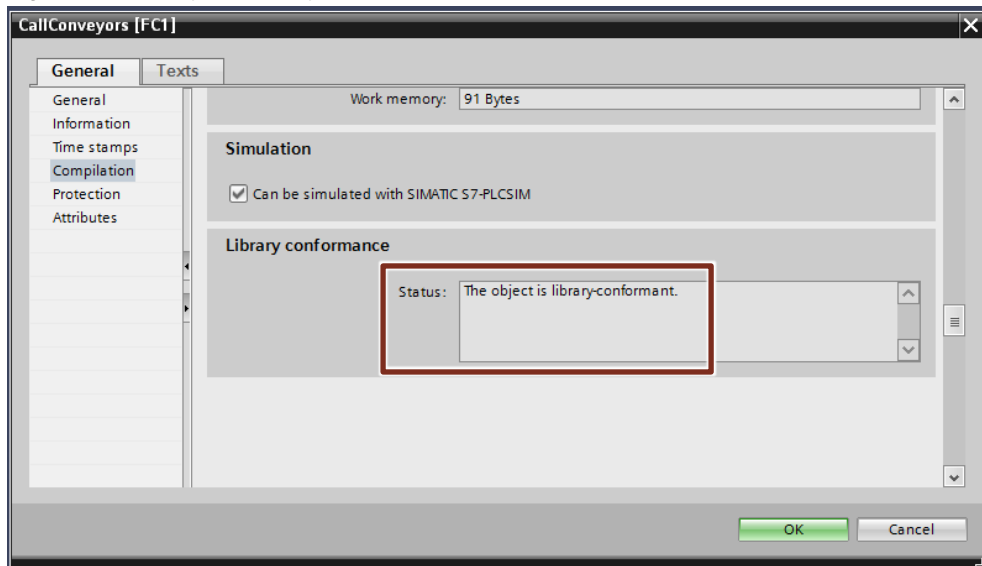
DA004 Rule: Use PLC data types

> PLC data types shall be used for structuring in the user program. In the local data PLC data types are used as well, when the variables are transferred in a single unit.

> Structures (STRUCT) are only declared in the local data of a block, to group variables in an easier to understand manner, but not to exchange them.

> **Justification:** A change in the PLC data type is automatically updated in all locations, which eases the data exchange between multiple blocks via formal parameter.

2.6 Terms used with variables and parameters

DA005 Rule: Exchange data only via formal parameters

If data is required in multiple FBs or FCs, then the data exchange is done only via the Input, Output or InOut parameters.

Justification: In the sense of encapsulated blocks data are disconnected from such reusable blocks and such dependencies are solved. The block does not need to be modified as the data is passed in via formal parameters. The caller remains in control, which data is being used where. The data consistency in case of multiple accesses (possibly in different places within the program) is guaranteed.

DA006 Rule: Access static variables from within the block only

The static data of a function block shall only be used within the block in which they have been declared.

Justification: With direct access to static variables of an instance the compatibility cannot be guaranteed, because there is no influence on future updates. Additionally, it is unclear, which influence the modification of static variable has on the execution of the FB.

DA007 Recommendation: Group formal parameters

When there are many (e. g. more than ten) parameters to pass, then these parameters shall be grouped into a PLC data type. This parameter shall be declared as InOut parameter and will be passed as "Call by reference".

Examples for such parameters are configuration data, actual values, setpoints or the output of diagnostic data of a function block.

Refer to "PE003 Recommendation: Pass structured parameters as reference"

Note	In case of often changing control and status variables it may be beneficial to make these directly available for an easy monitoring in LAD/ FBD and declare them as elementary input or output parameter.

DA008 Rule: Write output parameters only once

The output variables and return values are written once per execution cycle. This shall take place, when possible, collectively towards the end of the block.

It is not permitted to read the own output parameter or return value. Instead of that a temporary or static variable must be used.

Justification: This makes sure, that all output values are consistent.

2.6 Terms used with variables and parameters

DA009 Rule: Keep used code only

In the released program only code shall be contained, which is being executed in the PLC.

Examples for violations:

- Never called blocks or technological objects
- Never used variables or constants
- Never used parameter
- Never executed program code
- Commented out code
- Never accessed PLC variables
- Never used user constants
- External source files

Note	Productive code, which may be used at a later point in time depending on an option, is not affected by this.

DA010 Rule: Develop asynchronous blocks according to PLCopen

The PLCopen organization has defined a standard for Motion Control blocks. This standard can be generalized in that way, that it can be applied to all asynchronous blocks. Asynchronous means in this context, that the execution of the function inside the block extends over multiple (more than one) execution cycles of the PLC, e.g. for communication, closed loop control or motion control blocks.

Justification: Applying this standard a simplification for the programming and the use can be achieved.

DA011 Rule: Continuous asynchronous execution with "enable"

Blocks which are started and initialized only once and afterwards remain in operation to respond to inputs have an "enable" input parameter.

Example: A communication block (acting as server) waits after initialization for incoming connection requests from a client. After a successful data exchange the server waits for other incoming connection requests.

Note	The block template with enable can be found as master copy in the library of general functions (LGF): https://support.industry.siemens.com/cs/ww/en/view/109479728

Setting the parameter "enable" starts the execution of an asynchronous task. If "enable" remains set, the task execution remains active and new values are being accepted and processed.

Resetting the parameter "enable" the task will be finished.

Diagnostic information (diagnostics) will be cleared with a new rising edge on "enable".

If the block is implemented according to PLCopen and the "enable" input para-meter is used, then at least the output parameter "valid", "error" and "busy" must be provided.
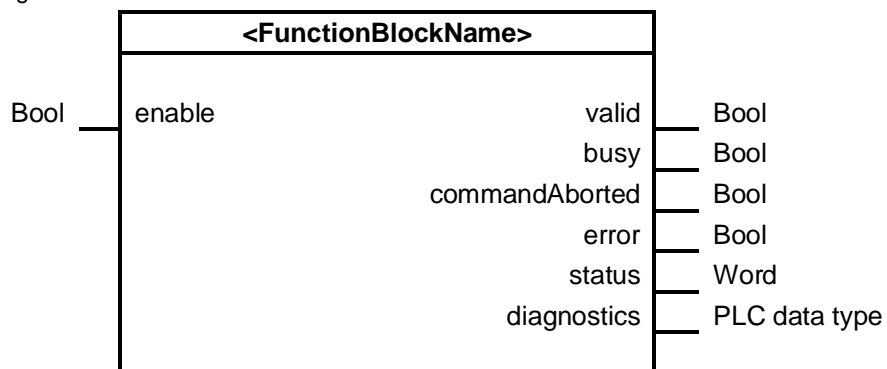
2.6 Terms used with variables and parameters

Table 9-1

Identifier	Data type	Description
Input parameters		
enable	Bool	All parameters are being activated with a rising edge on the input "enable" and may be modified continuously. The function is level triggered activated (with TRUE) and deactivated (with FALSE).
Output parameters		
valid	Bool	The outputs "valid" and "error" are **mutual exclusive**. The output is set if the output values are valid and the enable input is set. As soon as an "error" is detected the output "valid" is reset.
busy	Bool	The FB is executing a command. New output values can be expected. The output "busy" is set with a rising edge on enable and remains set, as long as the FB executes a command.
error	Bool	The outputs "valid" and "error" are **mutual exclusive**. A rising edge of the output indicates that an error occurred during the execution of the FB.
commandAborted	Bool	**Optional** output, which indicates that the currently executed task of the FB has been canceled by another function or another task for the same object. Example: An axis is being positioned while another function block stops the same axis. The positioning function block sets the "commandAborted" output to indicate, that the command has been aborted by the Halt command.
status	Word	**Optional** output: error and status information of the block: This parameter has got its name from the system functions. ("errorID" according to PLCopen)
diagnostics	PLC data type	**Optional** output: detailed error information Here all error messages and warnings are being stored. The structure of the diagnostic information is described in the recommendation "passing underlying status information"

Example:

Figure 9-4

2.6 Terms used with variables and parameters

Signal diagram of a block with "enable":

Figure 9-5
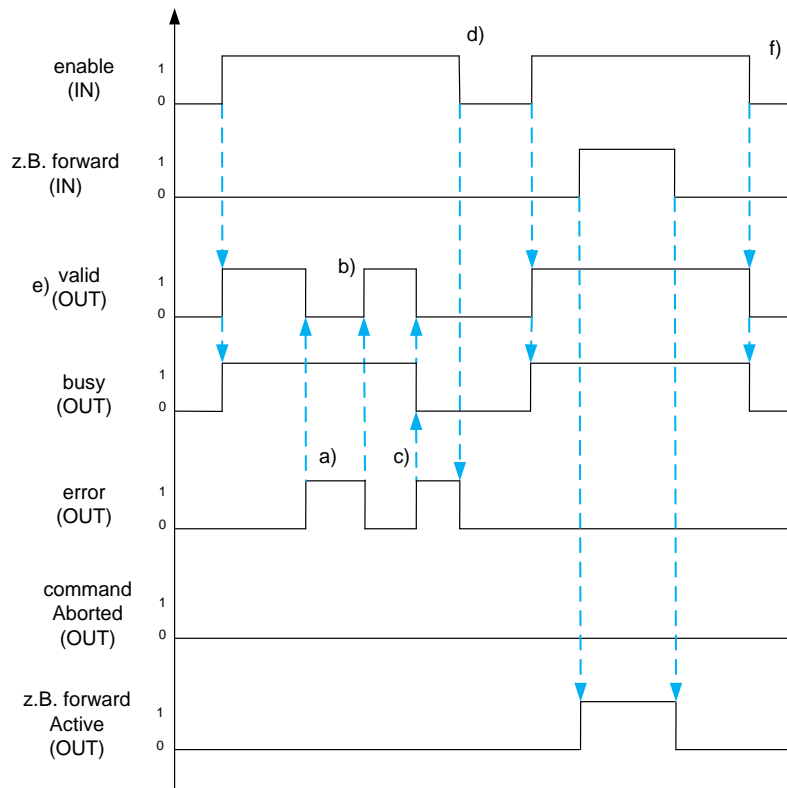
a) With "error" at TRUE the "valid" is being reset and all functions within the block are being stopped. Because the error can be handled by the block itself the "busy" flag remains active.

b) After clearing the cause of the error (e.g. reestablishing a connection) "valid" becomes active again.

c) An error occurs, which can only be cleared by the user, then "error" must be set and "valid" and "busy" must be reset.

d) Only a falling edge at "enable" clears the current error, which can only be cleared by the user.

e) "valid" set to TRUE means, the block is active, no errors occurred, and the output signals of the FB are valid.

f) If "enable" is reset to FALSE, then "valid" and "busy" must be reset as well.

"commandAborted", "error" and "done" must be set for as long as the signal enable is set, at least for one execution cycle.

2.6 Terms used with variables and parameters

DA012 Rule: Single asynchronous execution with "execute"

Blocks, which get executed only once have an input parameter "execute".

Example: A communication block (client) requests data of a server only once. This is triggered by an edge at the input signal "execute". After the processing the reply the execution is done. A new request is placed by another edge on "execute".

Note

The block template with "execute" can be found as master copy in the library of general functions (LGF):

https://support.industry.siemens.com/cs/ww/en/view/109479728

A rising edge on "execute" starts the task and the values at the input parameters are applied.

Any changes to the values after the start of the task only take effect after a start of a new task, unless "continuousUpdate" is being used.

The reset of the parameter "execute" does no stop the execution of the current task but has an influence on the display duration of the execution status. If "execute" gets reset before the current task has finished, then the parameter "done", "error" and "commandAborted" will be set for only one cycle.

Diagnostic information ("diagnostics") will be cleared only with a new rising edge on "execute".

After finishing the task, a new rising edge on "execute" is necessary to start a new task. This ensures the block is its initial state and the task can be executed independent of previous tasks.

If the block is implemented according the PLCopen standard and the "execute" input parameter is used, then the output parameter "busy", "done" and "error" must be used.
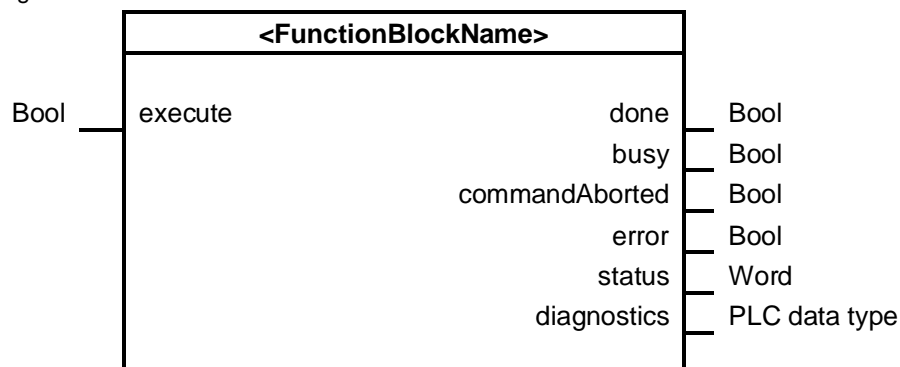
Table 9-2

Identifier	Data type	Description
Input parameters		
execute	Bool	**"execute" without "continuousUpdate":** All parameters are taken over with a rising edge on "execute" and the implemented function is started. When changes on the input parameter become necessary, a new rising edge on "execute" is required. **"execute" with "continuousUpdate":** All parameters are taken over with a rising edge on "execute". Their values can be adjusted if the input "continuousUpdate" is set.
continuousUpdate	Bool	**Optional** input: Refer to "execute" input
Output parameters		
done	Bool	The outputs "done", "busy", "commandAborted" and "error" are **mutual exclusive**. The output "done" is being set, if the command was executed successfully.
busy	Bool	The outputs *done*, *busy*, *commandAborted* and *error* are **mutual exclusive**. The FB is not done with the execution of the command, which means that new output values can be expected. *busy* is set with a rising edge on *execute* and reset, if one of the outputs *done*, commandA*borted* or *error* is being set.

2.6 Terms used with variables and parameters

| Identifier	Data type	Description
error	Bool	The outputs "done", "busy", "commandAborted" and "error" are **mutual exclusive**. A rising edge of the output indicates that an error occurred during the execution of the FB.
commandAborted	Bool	The outputs "done", "busy", "commandAborted" and "error" are **mutual exclusive**. **Optional** output, which indicates that the currently executed command has been aborted by another function or by another command to the same object. Example: An axis is being positioned while another function block stops the same axis. The positioning function block sets the "commandAborted" output to indicate, that the command has been aborted by the Halt command.
status	Word	**Optional** output: Error and status information of the block: This parameter has got its name from the system functions. ("errorID" according to PLCopen)
diagnostics	PLC data type	**Optional** output: Detailed error information. Here all error messages and warnings are being stored. The structure of the diagnostic information is described in the recommendation "passing underlying status information"

Example:

Figure 9-6

2.6 Terms used with variables and parameters

Signal diagram of a block with "execute":

Note	If the input parameter "execute" is being reset before the output parameter "done" or "error" are being set, then "done" or "error" must be set for one cycle only.

Figure 9-7
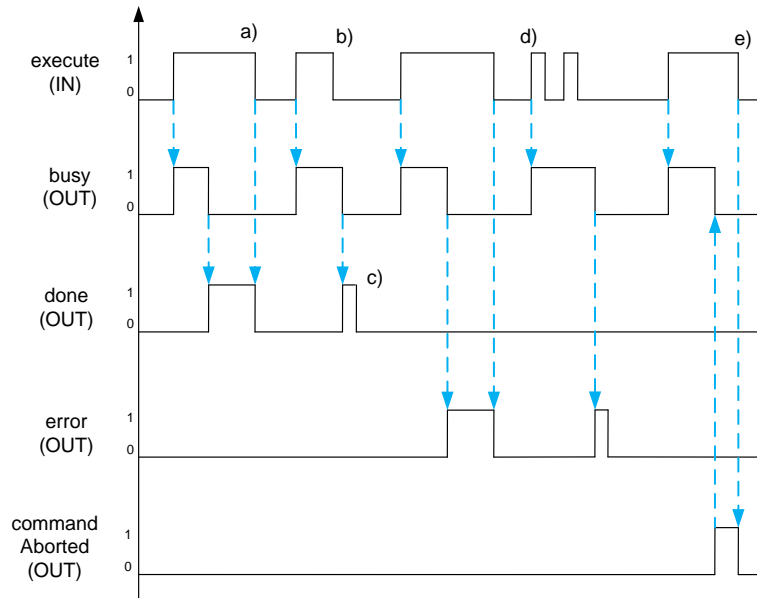
a) "done", "error" and "commandAborted" are being reset with a falling edge on "execute".

b) The functionality of the FB is not being stopped by a falling edge on "execute".

c) If "execute" is already FALSE, then "done", "error" and "commandAborted" are being set only for one cycle.

d) A new command is requested with a rising edge on "execute" while to previous command is still in execution ("busy" = TRUE). The previous command shall be either finished with the previously used parameters or the previous command shall be aborted and restarted with the new parameters. The behavior depends on the use case scenario and must be documented.

e) In case the execution of a command is interrupted by another command of the same or higher priority (from another block/ instance), then the block sets the "commandAborted" output parameter. The block stops any remaining execution of the command. This scenario can occur, when an emergency stop is issued, while an axis executes a positioning command.

2.6 Terms used with variables and parameters

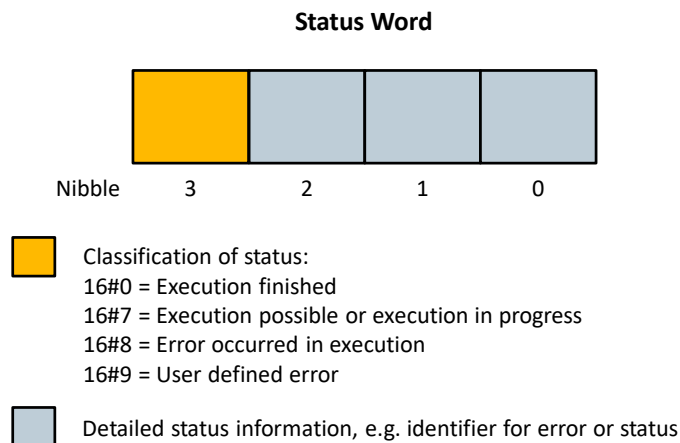## DA013 Rule: Report status/ errors via "status"/ "error"

The block reports a unique status at its output parameter "status", which provides information about the internal status of the block. The values must be defined as local symbolic constants in the block interface to avoid double usage and increase the readability.

Reports the block an error the output parameter "status" and "error" shall be used. Following the below described status concept, the parameter of "error" is the MSB of the "status" (bit 15). The remaining bits are being utilized for the error code, which allows an identification of the cause of the error.

For compatibility reasons to the SIMATIC system blocks the output "errorID", which is mandatory in the PLCopen standard, is replaced by the output "status".

Justification: This allows to pass on further detailed information about the block's status via the output "status", which do not contain error information.

Figure 9-8

Status Word

Nibble   3   2   1   0

Classification of status:
16#0 = Execution finished
16#7 = Execution possible or execution in progress
16#8 = Error occurred in execution
16#9 = User defined error

Detailed status information, e.g. identifier for error or status

DA014 Rule: Use standardized value ranges for "status"

For a standardization of the output parameter "status" the below defined value ranges for information and errors shall be used.

Table 9-3

Information	Value range
Command finished, no warnings and no further details	16#0000
Command finished, further details	16#0001 ... 16#0FFF
No command in execution (initial value)	16#7000
First call after receiving a new command (rising edge on *execute*)	16#7001
Follow up call during active execution of a command without further details	16#7002
Follow up call during active execution of a command with further details. Warnings without effect of further processing.	16#7003 ... 16#7FFF

2.6 Terms used with variables and parameters

Table 9-4

Error	Value range
Wrong operation of the function block	16#8001 ... 16#81FF
Wrong parameterization	16#8200 ... 16#83FF
Errors during execution from outside (e.g. wrong I/O signals, axis not homed)	16#8400 ... 16#85FF
Internal error during execution (e.g. during a system call)	16#8600 ...16#87FF
Reserved	16#8800 ... 16#8FFF
User defined error classes	16#9000 ... 16#FFFF

DA015 Recommendation: Pass underlying information

If a block calls other subfunctions, which report detailed status and possibly diagnostic information, then they must be copied into a diagnostic structure at the output parameter "diagnostics". Further this diagnostic structure may contain additional values for diagnostic purposes, such as runtime information.

Note The diagnostic structure may be stored persistent to allow a diagnosis even after a power failure.

Example for a simple diagnostic structure:

Figure 9-9

The simple diagnostic structure contains three parameters:

Table 9-5

Identifier	Data type	Description
status	Word	Status of the current block
subfunctionStatus	Word/ DWord	Status of the underlying subfunction
stateNumber	DInt	Number if the internal execution state/ execution step, where the error occurred.

2.6 Terms used with variables and parameters

Example for an extended diagnostic structure:

Figure 9-10

The variable "timestamp" contains the point in time when the error occurred.

In "stateNumber" the current internal state of the internal state machine is stored.

If there is an error of a system function or a called FB/ FC, its status shall be stored in the variable "subfunctionStatus".

The unique error code of the output parameter shall be copied to the variable "status" of the diagnostic structure.

Additional variables amending an error (also underlying variables) can be added to the diagnostic structure with an appropriate data type, e.g. with the help of "additionalValueX", where "X" is replaced by an increasing number starting by 1.

DA016 Recommendation: Use CASE instruction instead of ELSIF branches

When possible use the CASE instruction instead of the IF instruction with multiple ELSIF branches.

Justification: The program becomes more readable.

DA017 Rule: Create ELSE branch in CASE instructions

A CASE instruction must always have an ELSE branch.

Justification: This serves the purpose to report errors, which may occur at runtime.

Example:

```
CASE #stateSelect OF
  CMD_INIT: // Comment
    ; // Statement
  CMD_READ: // Comment
    ; // Statement
ELSE
  // default statement
  ; // Generate error message
END_CASE;
```

DA018 Recommendation: Avoid Jump and Label

Avoid jumps within the program whenever possible. Jumps are only permissible on an exceptional basis, if there is no other method possible to realize the program.

Justification: Jumps lead to programs, which are difficult to follow as these instructions may jump from one place to another inside the program.

10 Performance

In this chapter the rules and recommendations are described, which support the development of performant user programs.

PE001 Recommendation: Deactivate "Create extended status info"

Deactivating the "Create extended status information" may lead in the productive operation to a better performance. During the development process and for debugging purposes of the user program it may be beneficial to activate this setting.

Figure 10-1
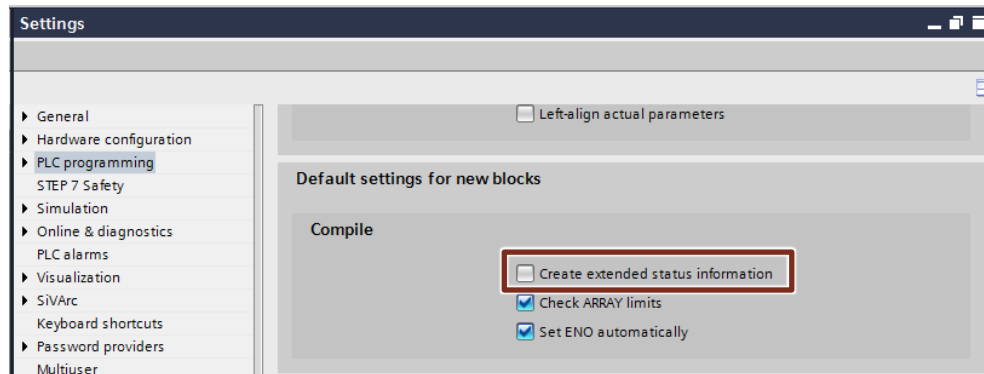
PE002 Recommendation: Avoid "Set in IDB"

To allow block optimization and for the sake of full symbolic programming the functionality "Set in IDB" in the block interface shall be avoided.

Justification: The use of "Set in IDB" causes the system to create a hybrid DB made up of optimized and non-optimized data areas. Accessing these data causes the system to copy these data into the other data format.

Note "Set in IDB" is being used mostly in conjunction with the "AT construct". Instead Slice accesses or the system functions SCATTER and GATHER may be used.

PE003 Recommendation: Pass structured parameters as reference

To pass data as performant (memory and runtime optimized) as possible into the formal parameters of the block interface, it is recommended to use the "Call by reference" schema.

Justification: Calling a block a reference to the actual parameters are being passed. For this the actual parameters are not copied.

Note Using this method, the original data may be modified.

If optimized data is passed to a block with the deactivated property "Optimized block access" (or vice versa) when the block is called, the data is always passed as a copy. If the block contains many structured parameters, this can lead to the temporary memory area of the block overflowing. You can avoid this by setting the "optimized" access type for both blocks.

2.6 Terms used with variables and parameters

The following table provides an overview of how formal parameters with elementary and structured data are passed in a SIMATIC S7-1200/ S7-1500 PLC.

Table 10-1

Block type/ formal parameter		Elementary data type	Structured data type
FC	Input	Copy	Reference
	Output	Copy	Reference
	InOut	Copy	Reference
FB	Input	Copy	Copy
	Output	Copy	Copy
	InOut	Copy	Reference

PE004 Recommendation: Avoid formal parameter with Variant

To avoid performance losses due to the use of Variant, it is recommended to keep separate blocks for different data types.

You only need to use Variant if, for example, you need to pass data to a block for communication in order to pass it on to the internal system communication modules or for serialization.

PE005 Recommendation: Avoid formal parameter "mode"

Avoid developing blocks that operate differently depending on a input parameter, e.g. "mode".

Justification: This prevents code fragments that are not are needed ("dead code"), since the mode parameter is usually connected statically.

Instead, you should distribute the functionalities to individual modules:

• This reduces memory consumption and increases performance through code reduction.

• It increases readability through better differentiation and better naming.

• It increases maintainability through smaller code fragments, which are independent of each other.

PE006 Recommendation: Prefer temporary variables

Variables should be declared as temporary variables if they are only needed in the current cycle. Temporary variables offer the best performance in the block.

If input or in/out parameters are accessed frequently, a temporary variable should be used as a cache to improve the runtime.

Note	Temporary variables cannot be monitored or forced in watch tables or force tables.

PE007 Recommendation: Declare important test variables as static

Important test variables should be declared statically. They must provide enough information about the state of the functions.

Justification: The value of a static test variable remains even after the execution of a block is finished. This way it can be used for diagnostic purposes.

2.6 Terms used with variables and parameters

PE008 Recommendation: Declare control/ index variables as "DInt"

It is recommended to use the data type "DInt" for control and index variables that are used for loops, iterations and array access.

Justification: The data type "DInt" can be processed with the best performance, since no type conversion is necessary. Accordingly, the definitions for the array sizes and loop boundaries should also be created as constants of the data type "DInt".

PE009 Recommendation: Avoid multiple access using the same index

Avoid repeated access to the same index of an array. A temporary variable should be used as cache.

Justification: This method reduces the internal system checks of the array boundaries and the check of exceeding them to a minimum.

Example:

```
FOR #tempIndex := 0 TO #MAX_ARRAY_ELEMENTS DO
  // Copy to temporary variable
  #tempCurrentData := #statArray[#tempIndex];
  // Reset all member variables
  #tempCurrentData.element1 := FALSE;
  #tempCurrentData.element2 := FALSE;
  #tempCurrentData.element3 := FALSE;
  #tempCurrentData.element4 := FALSE;
  #tempCurrentData.element5 := FALSE;
  // Write back the changes made
  #statArray[#tempIndex] := #tempCurrentData;
END_FOR;
```

PE010 Recommendation: Use slice access instead of masking

Instead of masking for a few individual bit accesses, the slice access can be used to access individual bits.

Justification: This method increases performance and readability of the source code.

Example: Evaluating Bit1 = TRUE and Bit0 = FALSE using slice access

```
#tempIsTriggered := (#trigger.%X1 AND NOT #trigger.%X0);
```

Masking is recommended whenever bit patterns are to be compared with variables.

Example: Evaluating Bit1 = TRUE and Bit0 = FALSE using masking

```
PATTERN_MASK    BYTE    2#00000011
PATTERN         BYTE    2#00000010


#tempIsTriggered := ((#trigger AND #PATTERN_MASK) = #PATTERN);
```

2.6 Terms used with variables and parameters

PE011 Recommendation: Simplify IF/ ELSE instructions

Simplifying IF/ ELSE instructions to simple binary operations improves performance and reduces memory consumption.

Negative example demonstrating edge detection:

```
// Check for rising edge
IF #trigger AND NOT #statTriggerOld THEN
  #tempIsTrigger := TRUE;
ELSE
  #tempIsTrigger := FALSE;
END_IF;
// Store trigger for next cycle
#statTriggerOld := #trigger;
```

correct example:

```
// Check for rising edge
#tempIsTrigger := #trigger AND NOT #statTriggerOld;
// Store trigger for next cycle
#statTriggerOld := #trigger;
```

PE012 Recommendation: Sort IF/ ELSIF branches according to expectation

IF/ ELSIF statements, should be ordered by decreasing likelihood so that the most likely case should come first and so forth.

Justification: This avoids evaluations for less likely conditions and therefore improves performance.

Example: Assuming the program flow has been implemented error free and the ideal situation is known, then the likeliest condition is evaluated first.

```
// Check if connection is established
IF #instConnect.done = TRUE THEN
  // Connection is established - set next state
  ;
// Check if TCON throws an error
ELSIF #instConnect.error = TRUE THEN
  // TCON throws an error - do error handling
  ;
END_IF;
```

PE013 Recommendation: Avoid memory intense instructions

The use of memory intense instructions, like:

- "GetSymbolName"
- "GetSymbolPath"
- "GetInstanceName"
- "GetInstancePath"

shall be avoided.

Justification: The use of the mentioned instructions above results in increased working memory usage. The amount being used depends on the number of instruction calls and the length of the symbolic identifiers.

PE014 Recommendation: Avoid runtime intense instructions

The use of the runtime intense instructions shall be reduced to a minimum.

Runtime-intense instructions are instructions that process large amounts of data, such as "Serialize" and "Deserialize" or those who access the memory card.

The system functions "WRITE_DBL", "READ_DBL", "FileRead" and "FileWrite" access comparatively slow SIMATIC Memory Card. This is typically done asynchronously and may take several cycles. However, since larger data quantities are transferred, the use of these system functions can have a negative impact on the overall program duration.

Further examples for accesses to the SIMATIC Memory Card are DataLog and Recipe functions.

PE015 Recommendation: Use of SCL/ LAD/ FBD for time critical applications

For time-critical programs/ program parts and algorithms it is recommended to use one of the three programming languages SCL, LAD or FBD.

Justification: GRAPH as programming language generates additional diagnostic information, which require additional runtime. GRAPH is recommended to program sequential machine flows.

PE016 Recommendation: Check the setting for minimum cycle time

For time critical application without a high communication load the "Minimum cycle time" can be turned off, to allow for a fast response time.

High communication loads can be counteracted by enabling and raising the "minimum cycle time".

Figure 10-2
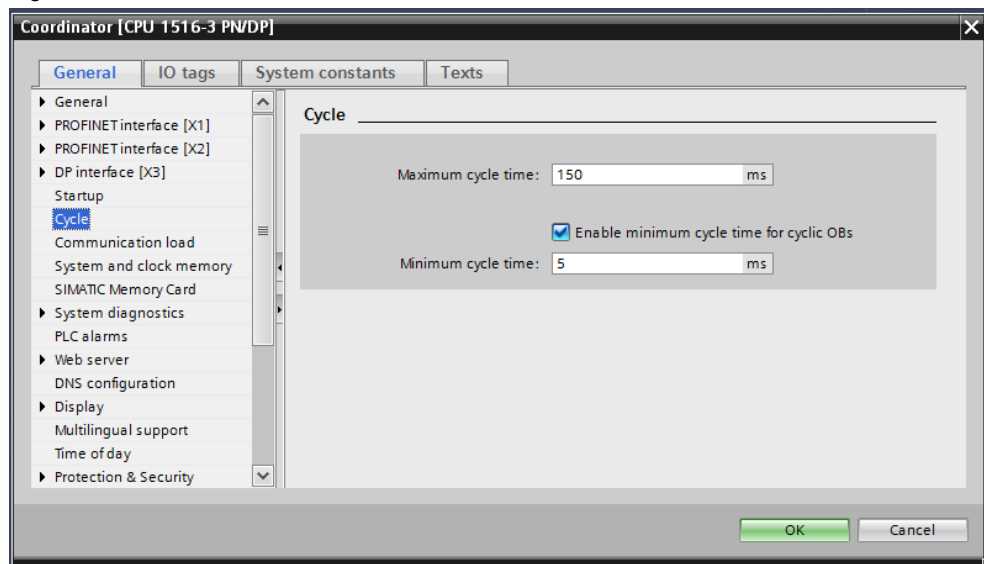
11 Cheat sheet

Block interface	In	Out	InOut	Stat	Temp	Const
	enable	done	conveyorAxes instTimer	statState instTimer powerBusReady	tempIndex	MAX_VELOCITY
Prefix	--	--	-- (default) "inst" (param-instance)	"stat" (default) "inst" (multi-instance) -- (in global data block)	"temp"	--
Casing	camelCasing	camelCasing	camelCasing	camelCasing	camelCasing	UPPER_CASING

Tag table	PLC tag	User constant
	lightBarrierLeft	MAX_BELTS
Prefix	--	--
Casing	camelCasing	UPPER_CASING

Programming styleguide for SIMATIC S7-1200/ S7-1500 in TIA Portal

- Unique, meaningful identifiers in English
- Only the characters a-z, A-Z, 0-9 and _
- Maximum 24 characters per identifier
- Array: axesData [0..#MAX] of type…
- Library: Name max. 8 chars; prefix "LExample_"

Object		Prefix	Casing
Project	AssemblyLine	--	PascalCasing
Library	LCom	"L"	PascalCasing
Organization block	Main	--	PascalCasing
Function block	HeatTank	--	PascalCasing
Function	CalculateTime	--	PascalCasing
Global data block	MachineData	--	PascalCasing
Single instance data block	InstHeater	"Inst"	PascalCasing
Technological object	HeatingAxis	--	PascalCasing
PLC tag table	Sensors	--	PascalCasing
Watch/Force table	MachineState	--	PascalCasing
Trace	ConveyorSpeed	--	PascalCasing
Measurement	Temperature	--	PascalCasing
PLC alarm text list	ConveyorAlarms	--	PascalCasing
Software unit	Magazine	--	PascalCasing
PLC datatype	typeDiagnostics	"type"	camelCasing
Element in a PLC datatype	stateNumber	--	camelCasing

Usual abbreviations (maximum one per identifier)

Min / Max	Minimum / Maximum
Act	Actual, Current
Next / Prev	Next / Previous value
Avg	Average
Sum	Total sum
Diff	Difference
Cnt	Count
Len	Length
Pos	Position
Ris / Fal	Rising / falling edge
Old	Old value
Sim	Simulated
Dir	Direction
Err / Warn	Error / Warning
Cmd	Command
Addr	Address

12 Annex

12.1 Service and Support

Industry Online Support

You have question or need support?

Via the Industry Online Support, you have 24/7 access to the complete Service and Support Know-how as well as to our service offerings.

The Industry Online Support is the central address for information about our products, solutions and services.

Product information, manuals, downloads, FAQs and application examples – all information are only a few mouse clicks away:
https://support.industry.siemens.com

Technical Support

The Technical Support of Siemens Industry supports you fast and competent with all technical requests with a wide variety of tailored offerings
- starting with the basic support up to individual support contracts.

Requests to the Technical Support can be send via the WebForm:
www.siemens.de/industry/supportrequest

SITRAIN – Training for Industry

With our worldwide available trainings for our products and solutions we support you, with innovative learning methods and a customized concept.

More about the offered trainings and courses as well as our locations and schedules you can find at:
www.siemens.de/sitrain

Service

Our service offering contains the following:

- Plant Data services
- Spare Part services
- Repair services
- On Site and Maintenance services
- Retrofit- and Modernization services
- Service programs and contracts

Detailed information to our service offering can be found in our service catalogue:
https://support.industry.siemens.com/cs/sc

Industry Online Support App

With the App "Siemens Industry Online Support" you will get the optimum support even on the road. The App is available on Apple iOS, Android and Windows Phone:
https://support.industry.siemens.com/cs/ww/en/sc/2067

12.2 Links and Literature

Table 12-1

	Topic
\1\	Siemens Industry Online Support http://support.industry.siemens.com/
\2\	Download page of this entry https://support.industry.siemens.com/cs/ww/en/view/81318674
\3\	Standardization guideline https://support.industry.siemens.com/cs/ww/en/view/109756737
\4\	Library guideline https://support.industry.siemens.com/cs/ww/en/view/109747503
\5\	Provide user defined documentation https://support.industry.siemens.com/cs/ww/en/view/109755202/114872699275
\6\	SIMATIC S7-1200/ S7-1500 Compare list for programming languages https://support.industry.siemens.com/cs/ww/en/view/86630375
\7\	Library of General Functions (LGF) for SIMATIC STEP 7 (TIA Portal) and SIMATIC S7-1200/ S7-1500 https://support.industry.siemens.com/cs/ww/en/view/109479728

12.3 History

Table 12-2

Version	Date	Changes
V1.0	10/2014	First Release after internal review
V1.1	06/2015	Adaptations and Corrections
V1.2	10/2016	Adaptations and Corrections New: Cheat sheet
V2.0	05/2020	Categorization and Modernizing • Categorization according to Workflow • Extend topics for Performance and Design-/ Architecture • Extend Programming guidelines • Amendment of Justifications • Review of Cheat sheet