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

TIA Portal

https://support.industry.siemens.com/cs/ww/en/view/81318674

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. **The application examples are not subject to standard tests and quality inspections of a chargeable product and may contain functional and performance defects or other faults and security vulnerabilities. You are responsible for the proper and safe operation of the products in accordance with all applicable regulations, including checking and customizing the application example for your system, and ensuring that only trained personnel use it in a way that prevents property damage or injury to persons. You are solely responsible for any productive use.**

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. Any further use of the application examples is explicitly not permitted and further rights are not granted. You are not allowed to use application examples in any other way, including, without limitation, for any direct or indirect training or enhancements of AI models.

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 and to terminate your use of the application examples at any time. 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://www.siemens.com/global/en/general/terms-of-use) shall also apply.

Cybersecurity information

Siemens provides products and solutions with industrial cybersecurity 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 cybersecurity 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 cybersecurity measures that may be implemented, please visit https://www.siemens.com/cybersecurity-industry.

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 Cybersecurity RSS Feed under https://www.siemens.com/cert.

Table of contents

Legal information **2**
1 Introduction **4**
1.1 Goal 4
1.2 Advantages of uniform programming 5
1.3 Applicability 5
1.4 Scope 5
1.5 Rule violations and other regulations 5
2 Definitions **6**
2.1 Rules/Recommendations 6
2.2 Enumeration of rules 6
2.3 Performance 6
2.4 Identifier/Naming 7
2.5 Abbreviations 7
2.6 Terms used with variables and parameters 8
3 Settings in TIA Portal **10**
4 Globalization **14**
5 Nomenclature and Formatting **16**
6 Reusability **29**
7 Referencing objects (Allocation) **34**
8 Security **36**
9 Design guidelines/architecture **39**
10 Performance **54**
11 CheatSheet **60**
12 Appendix **61**
12.1 Service and support 61
12.2 Industry Mall 62
12.3 Links and Literature 63
12.4 Changelog / History 64

1 Introduction

When programming a SIMATIC controller, a developer has the task of making the application program as readable and structured as possible. Each developer uses his own strategy to accomplish this task, e.g. the naming of variables, blocks or the way in which the program is commented. The developers use different philosophies, so that there are very different application programs that can often only be understood 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 as well as 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 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 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 only have 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 of new employees 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 on the 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 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 TIA Portal, which are programmed in the programming languages according to IEC 61131-3 (DIN EN 61131-3). Those are Structured Text (SCL/ST), Ladder Logic (LAD/KOP), Function Block Diagram (FBD/FUP) and Sequential Function Chart (S7-GRAPH / SFC), as well as Continuous Function Chart (CFC) and Cause Effect Matrix (CEM).

This document is also applicable for Software Units, folders/groups, Organization Blocks (OB), Functions (FC), Function Blocks (FB), Technology Objects (TO), Data Blocks (DB), PLC data types (UDT), Named value data types (NVT), variables, constants, PLC alarm 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 interpret and apply 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 general, the names and identifiers used must always be unique in relation to the functionality and interface type used. This means that the name used also allows conclusions to be drawn about the underlying functionality. It is advisable to define project wide terminology.

In customer projects the applicable regulations, customer or branch specific standards as well as technological regulations (e.g. Safety, Motion, Communication, …) 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 Enumeration of 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 you want to establish your own numbered rules, you may use the numbers between 901 and 999.

Table 2-1

Prefix	Category
ES	Engineering System: programming environment
GL	Globalizatio
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 composed of:

- prefix
- name
- suffix

2.5 Abbreviations

The following abbreviations are 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
NVT	Named value data type

2.6 Terms used with variables and parameters

There are many terms when it comes to variables, functions and function blocks. 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

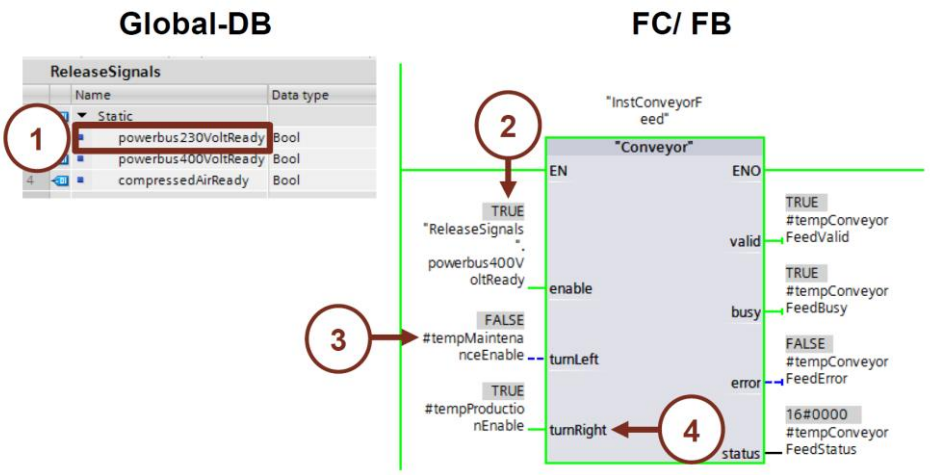


Table 2-3

	Term	Description
1	Variable	Variables are 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 - Variables in blocks - Variables of Structures (“STRUCT”), PLC data types and Named value data types - Data blocks and Instance data blocks - Technology Objects
2	Actual values Current values	Current values are the values, which are stored within a variable or constant (e.g. 15 as value of an Integer variable)
3	Actual parameter	Actual parameter are the variables or constants, 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 to pass data for calls within a program. Formal parameters are often referred to as “interface parameters” or “transfer 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, the block reads values from.
Output parameter	Output	Parameter, the block writes values to.
Input/Output parameter	InOut	Parameter, the block reads values from and writes the processed values back into the same parameter.
Return value	Return	Value, the block returns to its caller.

Local data

Table 2-5

Typ	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.

Where you find the settings: `TIA Portal` - Menu `Options` - Entry `Settings`

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.

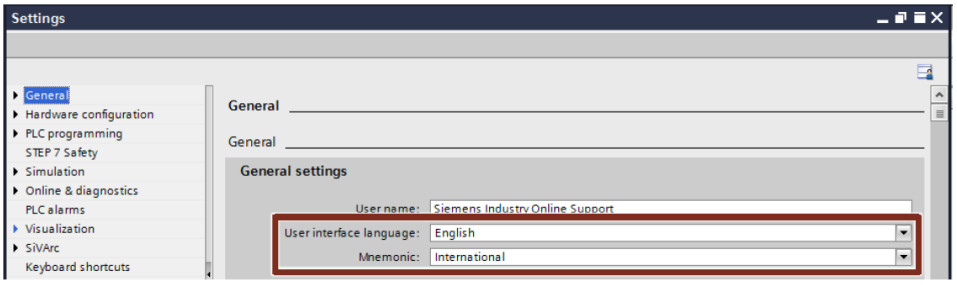
ES001 Rule: User Interface Language “English”

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 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.

Effect on: TIA Portal, TIA Portal Project, Create new objects (e.g. HW constant names set on UI language)

Figure 3-1



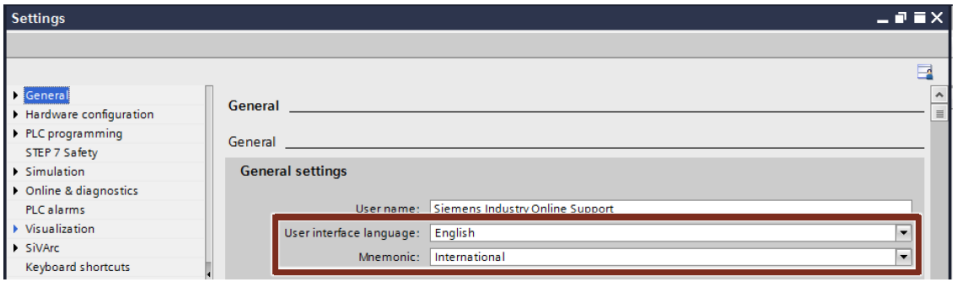
ES002 Rule: Mnemonic “International”

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.

Effect on: TIA Portal, TIA Portal Project, Creating new objects

Figure 3-2



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 10pt.

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

Effect on: TIA Portal

Figure 3-3

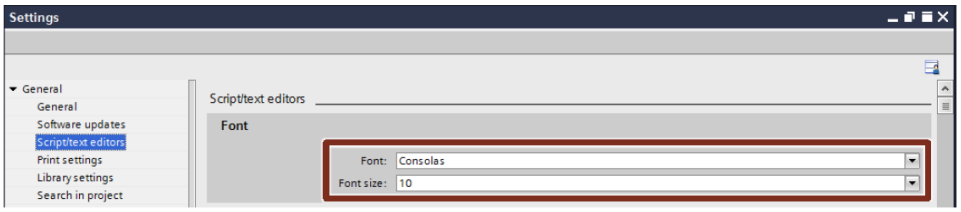


Figure 3-4

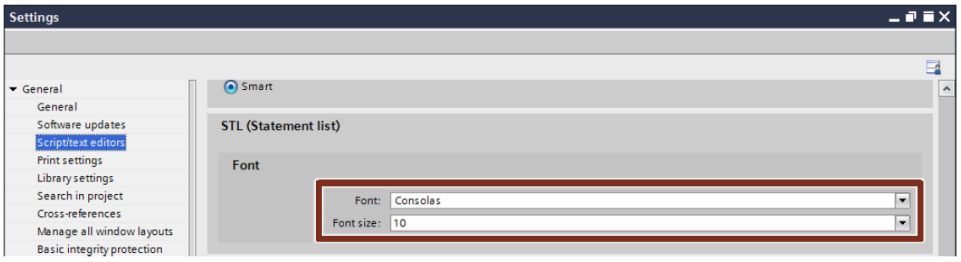
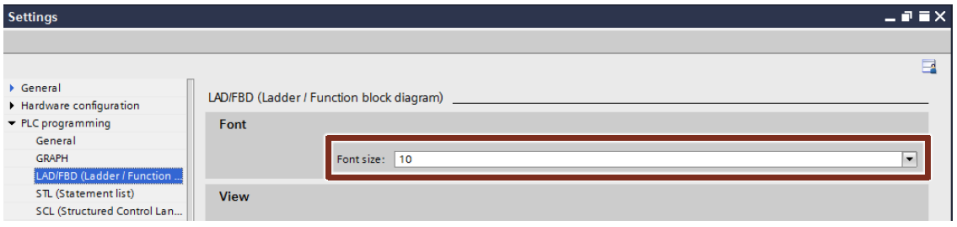


Figure 3-5



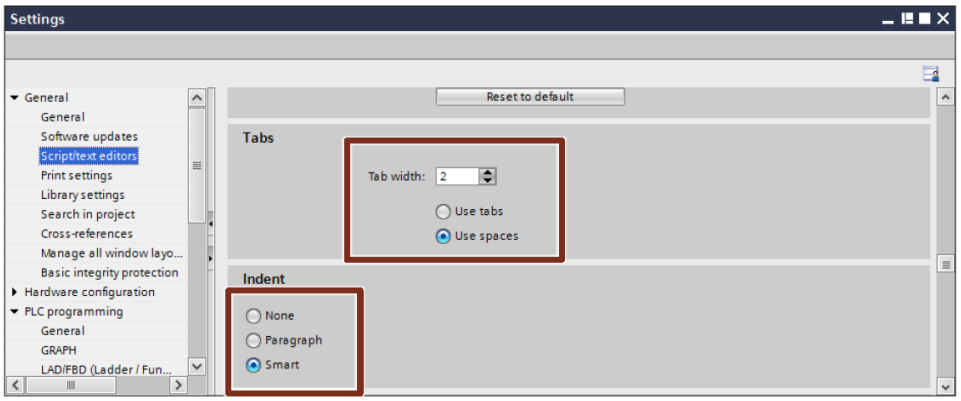
ES004 Rule: Smart indentation with two whitespaces

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.

Effect on: TIA Portal, TIA Portal Project, Automatic indentation

Figure 3-6



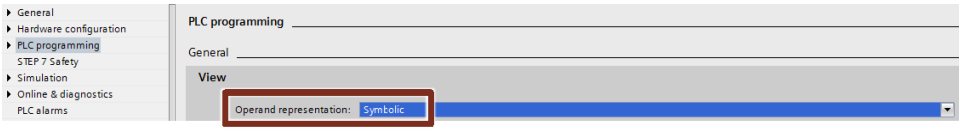
ES005 Rule: Symbolic representation of operands

The representation of operands is set to “Symbolic”

Justification: The programming is fully symbolic.

Effect on: TIA Portal (Adjustable in editor)

Figure 3-7



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.
For example, implicit type conversions, where there is a risk of data loss (`USint` 0..255 <–> `Uint` -128..127), or slice accesses to numerical data types are marked as errors by the compiler.

Effect on: Create new objects (Adjustable in object properties)

Figure 3-8

ES007 Rule: Explicit data access via HMI/OPC UA/Web API

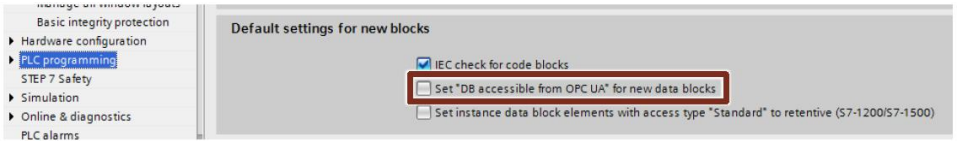
In order to grant access to a PLC via external applications such as HMI/OPC UA/Web API as secure as possible, enabling access is deactivated in the standard settings.

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

Effect on: Create new objects (Adjustable in object properties)

Figure 3-9

Figure 3-10



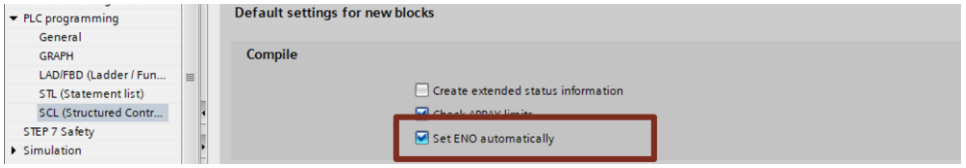
ES008 Rule: Automatic value evaluation (ENO) enabled

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.
The EN/ENO mechanism can be deactivated later at the module itself.

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

Effect on: Create new objects (Adjustable in object properties), Runtime environment of PLC

Figure 3-10



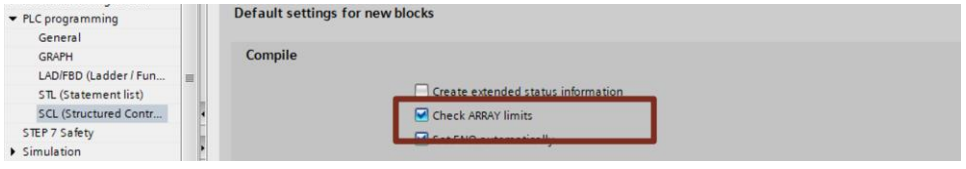
ES009 Rule: Automatic evaluation of Array boundaries

The automatic evaluation of Array boundaries must be turned on.

Justification: Checks at runtime whether array indices are within the declared range for an Array. If an array index exceeds the permissible range, the enable output `ENO` of the block is set to `FALSE`.

Effect on: TIA Portal, TIA Portal Project, Runtime environment of PLC

Figure 3-11



4 Globalization

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

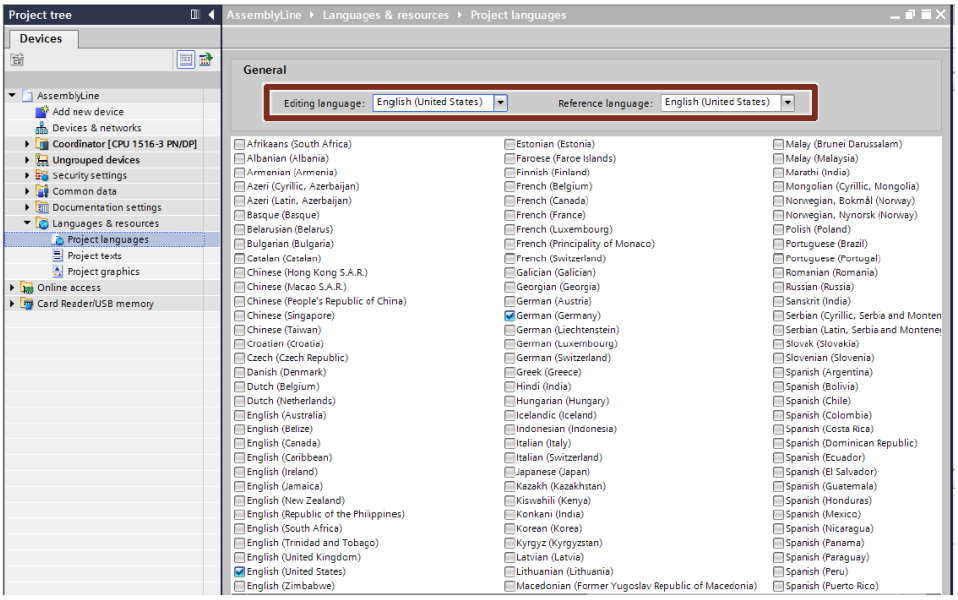
GL001 Rule: Use consistent language

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



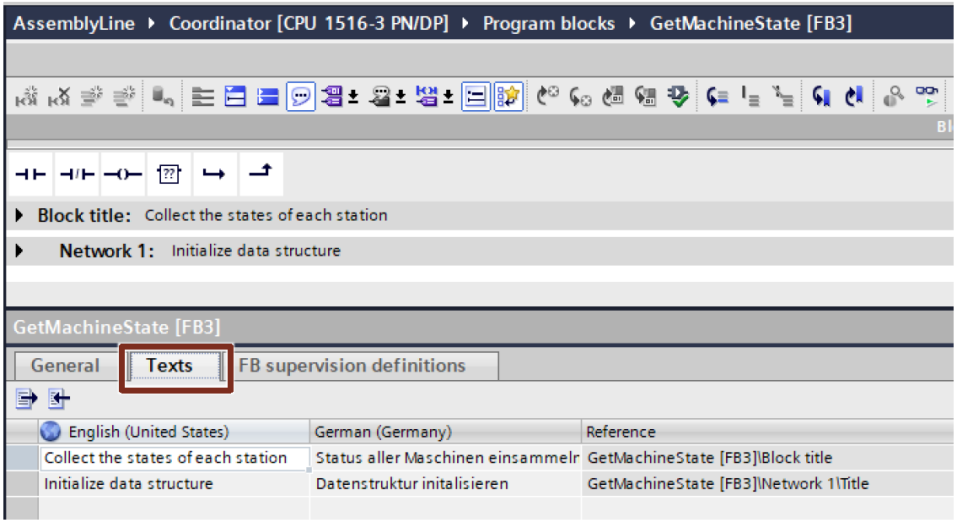
GL003 Rule: Supply texts in all active project languages

Project texts must always be translated into all active project languages.

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

It is furthermore possible to Ex- and Import the text for external handling.

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, variables, 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 objects 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. z. B. Get, Set, Put, Find, Search, Calc etc.
- 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).
- Boolean variables are often state indicating variables. In such cases names starting with “is”, “can” or “has” can be understood the easiest.
- Namespaces are used to structure and delimit the software elements and are named in the same way as the project folder structure.

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: Example table

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

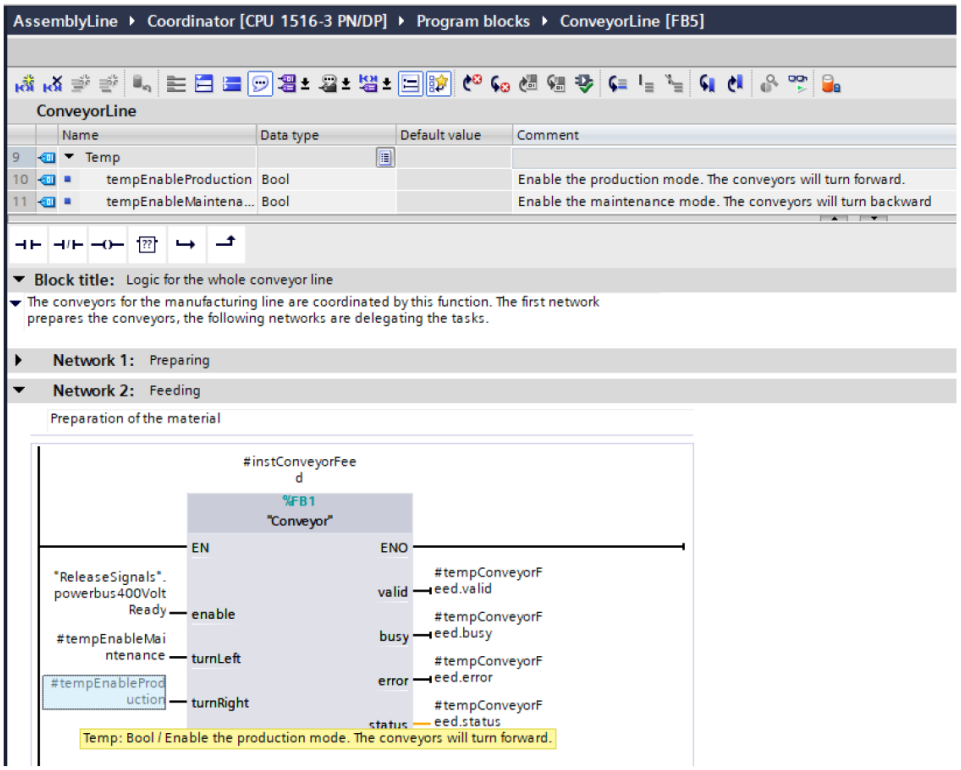
NF002 Rule: Use meaningful comments and properties

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

- Block title and block comments (refer also to “NF003 Rule: Document developer information“)
- Block interfaces (variables and constants)
- Network title and network comments
- PLC data types, Named value 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. *.html / *.pdf). 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/109755202/114872699275

Note

With the *Code2Docu* Add-In, it is possible to generate documentation from the objects in TIA Portal.

https://support.industry.siemens.com/cs/ww/en/view/109809007

NF003 Rule: Document developer information

Each block contains a block header in the program code (SCL/ST) or in the block comment (LAD/KOP, FBD/FUP). 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 know-how protected blocks.

User relevant information must be provided in the block properties. This information is available to the user even in know-how 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) (C)Copyright (Company Name) (Year of first release) - (Year of last change)
- (Optional) Title/Block description
- (Optional) Description of the functionality
- (Optional) Name of the library
- (Optional) Department/Author/Contact
- Tested on system – PLC(s) with firmware version (z. B. 1516-3 PN/DP V4.0)
- 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:

```
(C)Copyright (company) (Year of first release) - (Year of last change)
-------------------------------------------------------------------------
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)
Tested on 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 release
```

Template for a block header in SCL:

```
REGION Description header
  //=============================================================================
  // (C)Copyright (company) (Year of first release) - (Year of last change)
  //-----------------------------------------------------------------------------
  // 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)
  // Tested on 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 Description header
```

NF004 Rule: Comply with prefixes and structure for libraries

The identifier of a library has the prefix L and 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 name (e.g. LGF, LCom). "**L**" stands for the word "**L**ibrary". 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 avoided.

Table 5-2: Example table

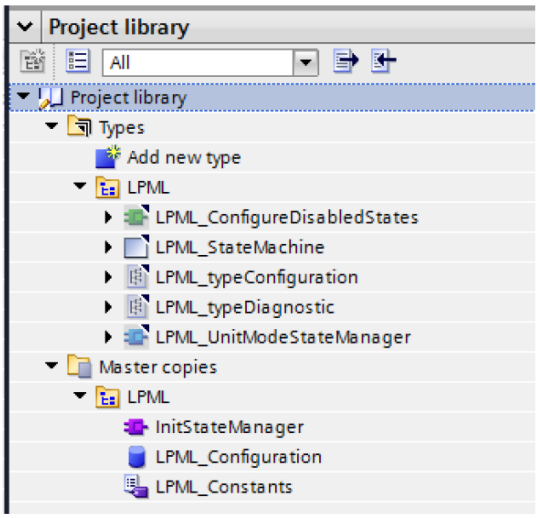
Type	identifier according to style guide
Library, main folder of the library	LExample
PLC data type	LExample_type<Name>
Named value data type	LExample_nvt<Name>
Function/Function block	LExample_<Name>
Organization block	LExample_<Name>
Global data block	LExample_<Name>
PLC symbol table	LExample_<Name>
Global constant	LEXAMPLE_<NAME>
Global constant for status/error/…	LEXAMPLE_STATUS_<NAME>
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 UpperCamelCase for objects

Identifiers for TIA Portal objects, such as:

- Blocks, Software Units, Technology Objects, Libraries, Projects, Namespaces
- PLC tag tables, PLC alarm text lists
- Watch and force tables
- Traces and measurements

are written using UpperCamelCase (PascalCasing).

The following rules apply for UpperCamelCase:

- 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_Axi

NF006 Rule: Use lowerCamelCase for code elements

Identifiers for code elements, such as

- Variables
- PLC data types
- Named value data types
- Structures (“STRUCT”)
- Parameters

are written using lowerCamelCase (camelCasing).

The following rules apply for the lowerCamelCase:

- 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

Conveyor

	Name	Datentyp
1	Input	
2	enable	Bool
3	turnLeft	Bool
4	turnRight	Bool
5	Output	
6	valid	Bool
7	busy	Bool
8	error	Bool
9	status	Word

NF007 Rule: Use prefixes

No prefix for formal parameters

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

Note Formal parameters of blocks do not get a prefix (e. g. `in` / `out` / `inOut`).

Temporary and static variables carry a prefix “temp” or “stat”/“ext”

To distinguish temporary and static internal variables clearly from formal parameters in the code, the prefixes `temp` and `stat` are used.
To declare static interface parameters to PLC external systems (e.g. HMI, MES,...) in the static area of an FB, these are marked with the prefix `ext`.

Justification: this measure makes it easier for the developer 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 Static variables inside global DBs do not have the prefix `stat` or `ext`.

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.

Note Data blocks derived from a PLC Data Type do not get a prefix `Inst`.

PLC data type with prefix “type”

A PLC data type declaration gets the prefix `type`. The definition of a variable of this PLC data type and the individual elements inside the PLC data type do not get a prefix.

Justification: This makes it easier for the developer of a function block to differentiate between PLC data types, Named value data types, function blocks and basic / system data types.

Note Data blocks derived from a PLC data type do not get a prefix `Type` or `type`.

Named value data type with prefix “nvt”

A Named value data type declaration gets the prefix `nvt`. The definition of a variable of this Named value data type and the individual elements inside the Named value data type do not get a prefix.

Justification: This makes it easier for the developer of a function block to differentiate between Named value data types, PLC data types, function blocks and basic / system data types.

Note Variables derived from a Named value data type do not get a prefix `nvt`.

For user irrelevant blocks and PLC data types with prefix “unpub”

If blocks and PLC data types of internal library elements are to be “hidden” for the user in the Intellisense list, this can be implemented by prefixing them with `unpub`.

Justification: By using the prefix, the objects in the Intellisense list are moved to the end due to the alphabetical order.

Examples:
`Lib_UnpubObject` / `Lib_unpubTypeDataType` / `Lib_unpubNvtNamedValue`

Table 5-4: Prefix overview

No.	Prefix	Type
1	No prefix	Input and Output parameter Access possible from outside | | ➔ enable ➔ error
2	No prefix	InOut parameter Modifications of the assigned data possible by the user as well as by the block at any time. ➔ conveyorAxis
3	No prefix	PLC tags and user constants ➔ lightBarrierLeft (identifier for a PLC variable) ➔ MAX_BELTS (identifier for a user constant)
4	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)
5	temp	Temporary Variables No access to local data from outside possible ➔ tempIndex
6	stat	Static internal variables No access to local data from outside permitted ➔ statState
7	ext	Static interface parameters Access to local data from PLC external systems (e.g. HMI, MES,...) permitted ➔ extInterface
8	inst	Variables of multi instances and parameter instances ➔ instWatchdogTimer ➔ instWatchdogTimers (with Arrays of instances)
9	Inst	Single instance data blocks ➔ InstConveyorFeed
10	type	PLC data type Only the data type gets the prefix, the elements do not get a prefix ➔ typeDiagnostics (identifier for PLC data type) ➔ stateNumber (identifier for variable inside the PLC data type)
11	nvt	Named value data type Only the Named value data type gets the prefix, the elements do not get a prefix ➔ nvtStates (identifier for NVT) ➔ PROCESSING (identifier for constant in NVT)
12	unpub	PLC block / PLC data type / Named value data type The prefix is placed before the name, for library elements after the library prefix ➔ Lib_UnpubObject (identifier for PLC block) ➔ Lib_unpubTypeDataType (identifier for PLC data type) ➔ Lib_unpubNvtNamedValue (identifier for Named value data type)

NF008 Rule: Write identifier of constants in UPPER_CASE

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

The constants include the following elements:

- Global constants
- Local constants
- Elements in Named value data types

The following rules apply for constants in UPPER_CASING:

- The first character is a capital letter.
- All following characters are either capitals or numerals.
- If an identifier is assembled out of multiple words, the words are split by an underscore.

Figure 5-4: Constants in a FB

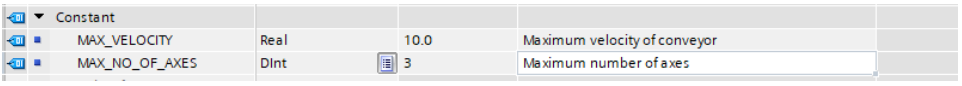


Figure 5-5: Constants in a Named value data type

```
1  TYPE
2      nvtStatus : Word (
3          STATUS_NO_ERROR := 16#0000,
4          ERR_UNDEF := 16#8600
5      );
6  END_TYPE
```

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

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 a single 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 elsewise.

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 number 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: Examples for abbreviations

Abbrev.	Type
Min	Minimum
Max	Maximum
Lim	Limit
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

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 5-6

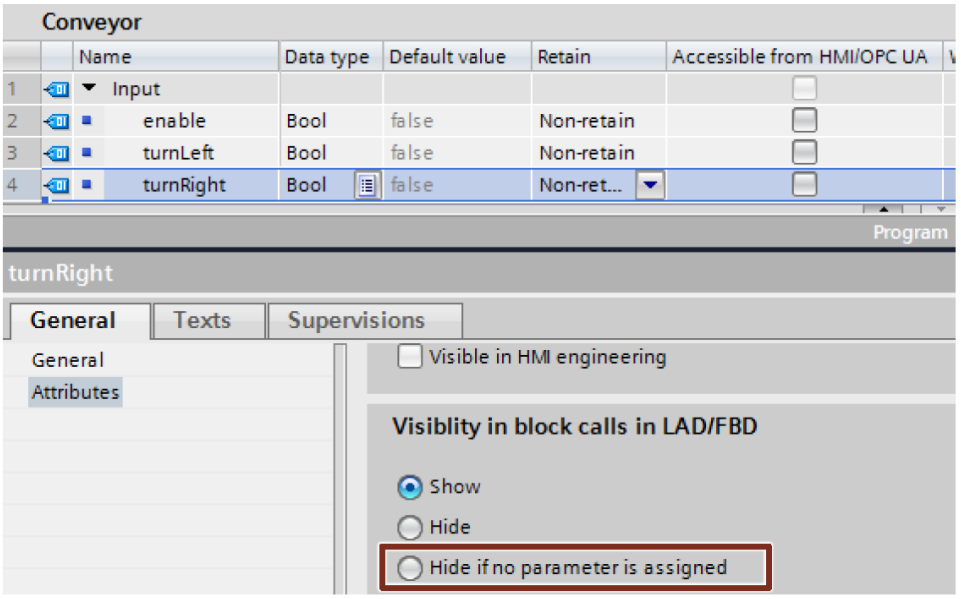
Correct initialization			Incorrect initialization		
statTriggerOld	Bool	FALSE	statTriggerOld	Bool	false
statStatus	Word	16#7000	statStatus	Word	123
statStep	DInt	101	statStep	DInt	16#0
statVelocity	LReal	0.0	statVelocity	LReal	16#000
statCommand	Byte	16#01	statCommand	Byte	16#1
statFlags	Byte	2#1010_0101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

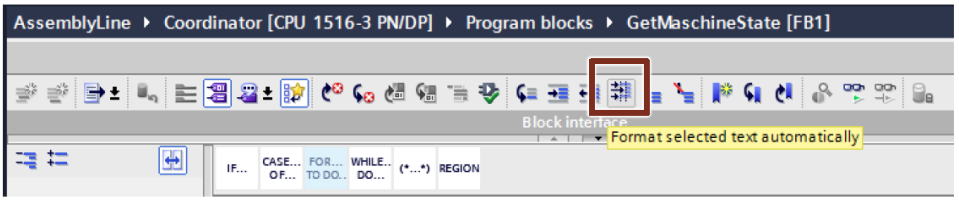
Figure 5-7



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 5-8



1. Preferable use of line comment `//`

There are two different types of comments:

- Comment sections `(*…*)` or multilingual comment sections `(/*…*/)` can span over multiple lines. It is used to describe 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.

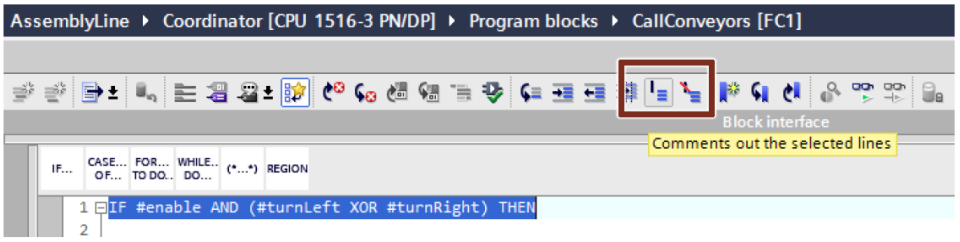
To allow easy disabling of code fragments for debugging purposes only line comments `//` are permitted.

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 described already by the code itself. Instead the reason why something has been done should be described.

Justification: This avoids syntax problems caused by inserting and possibly nesting comment sections with `(*...*)` or `(/*...*/)`.

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 with // or to the remove the comment signs.

Figure 5-9



2. Whitespaces in front of and behind operators

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

3. Expressions are always placed into parenthesis

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

Example:

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

4. 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;
```

5. Line breaks in partial conditions

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

Poorly readable example:

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

Better readable example:

```
#tempResult := FALSE
OR (#enable AND #tempEnableOld)
OR (#enable AND #isValid AND NOT (#hasError OR #hasWarning))
; // semicolon in seperate line, so each line can be commented out
```

6. 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`).

Example A1:

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

Example A2:

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

Example B:

```
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 Recommendation: Enable support for different target systems

Activate the simulation capabilities and usability for virtual PLCs via the project properties.

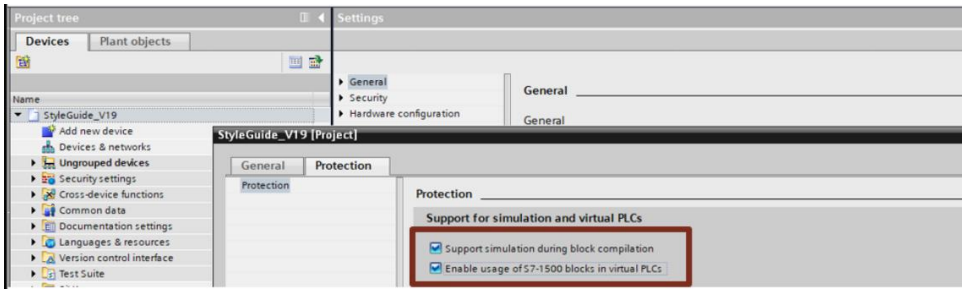
Justification: This ensures complete and full usability of the blocks during simulation or in virtual PLCs.

Note

Adding these options for know-how protected program blocks potentially reduces the level of protection, as know-how protection depends on the executing target system, among other things.

In runtime environments that can be operated on a wide variety of PC systems, the level of protection cannot be as high as in a dedicated hardware CPU, for example.

Picture 6-1



RU002 Rule: Version with libraries

Reusable Elements (e.g. FC, FB, PLC data types, Named value data types) which should not be changed by the user are provided as types in a library.

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

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 is inserted into a library, all necessary settings, such as auto numbering, know-how 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.

Version numbers and their use

- The first released version always starts with 1.0.0 (refer to Table 6-1).
- The first digit (Major) describes the left most number.
- Compatible extension to the functionality the second digit (Minor) will be incremented, and the third digit reset.
- The third digit (Patch) in the software versioning indicates changes that do not include any functional enhancements and do not require any new documentation, e.g. purely bug fixes.
- With a new major release, containing new functionality and incompatible changes to the previous version, increase the first digit and reset the second and third digit.
- Each digit has a valid range between 0 and 999.

Table 6-1: Version numbers and their use

Library	FB1	FB2	FC1	FC2	Comment
1.0.0	1.0.0	1.0.0	1.0.0	-	Released
`1.0.1`	`1.0.1`	1.0.0	1.0.0	-	Troubleshooting of FB1
`1.0.2`	1.0.1	`1.0.1`	1.0.0	-	Optimization of FB2
`1.0.3`	1.0.1	`1.0.2`	1.0.0	-	Adjusting Documentation of FB2
`1.1.0`	`1.1.0`	1.0.1	1.0.0	-	Compatible functional extension to FB1
`1.2.0`	`1.2.0`	1.0.1	1.0.0	-	Compatible functional extension to FB1
`2.0.0`	`2.0.0`	1.0.1	`2.0.0`	-	Incompatible functional extension on FB1 and FC1
`2.0.1`	2.0.0	`1.0.2`	2.0.0	-	Troubleshooting FB2
(`3.0.0`) `3.0.0`	2.0.0	1.0.2	2.0.0	`1.0.0`	New function FC2, can be labeled as a major or minor release
`3.0.1`	2.0.0	1.0.2	`2.0.1`	`1.0.1`	Troubleshooting

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
- Named value data types

RU004 Rule: Use only local variables

Within a reusable object (e. g. FCs, FBs, PLC Data types)

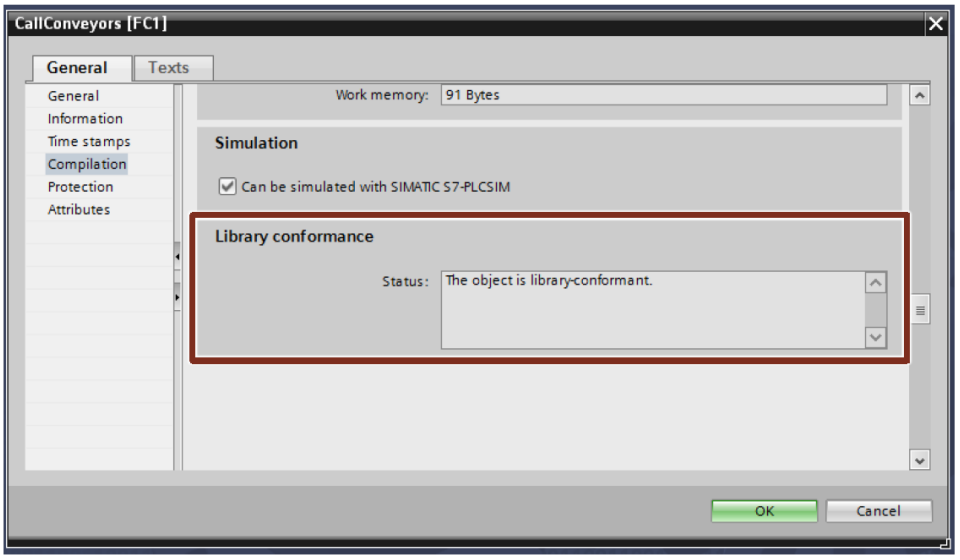
1. only local variables may be used. Global data must be passed in via the formal parameters of the block interface.
2. access to global constants is permitted for the definition of array limits for scaling different quantity structures.
   Refer also to “AL002 Recommendation: Define array boundary from 0 to a constant value”.

The library conformity of a block or PLC data type indicates that none of the following access types are used:

- 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 properties 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: Library-conformant object



RU005 Rule: Use symbolic constants

Local constants or Named value data types shall be used to further encapsulate a block. 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 know-how protected blocks this requires the knowledge of the assigned password.

No “magic numbers”

When a variable in the code is compared to or is 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.

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.

Note Named value data types are only available within Software Units.

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

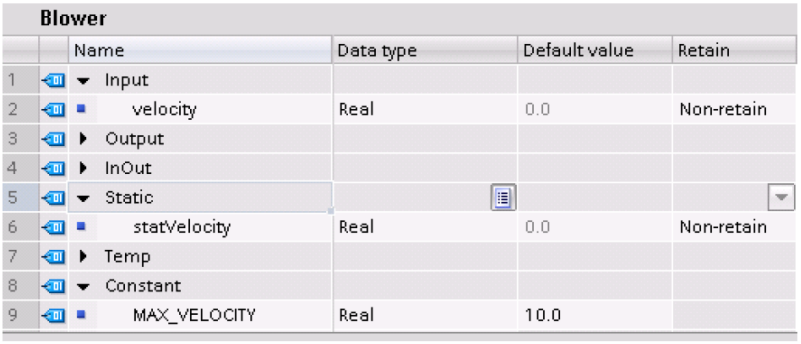
Example:

```
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.

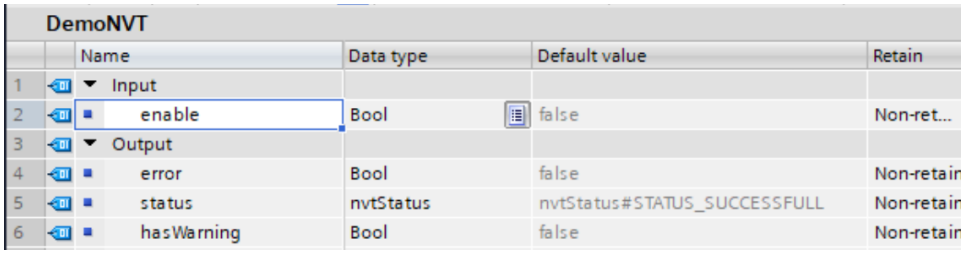
Example:

Figure 6-3




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

Picture 6-4




```
IF #tempError THEN
  #status := nvtStatus#ERR_UNDEF;
ELSE
  #status := nvtStatus#STATUS_SUCCESSFUL;
END_IF;

#error := #tempStatus.%X15;
```

RU006 Rule: Program fully symbolic

The programming is done fully symbolic. This means there are no physical addresses, such as with `Pointer` or `ANY-Pointer`, used in the program.
Furthermore, the numbers of system objects (e.g. HW-ID, OB, DB, ...) are not used directly (no “Magic Numbers”). Instead system constants are used.

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.

RU007 Recommendation: Program independently from hardware

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

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 global data blocks.

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 reduces 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 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 superimposed structures or global structures become necessary, thus reducing the number of objects.

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

The following apply to array boundaries:

1. Start index is 0 (lower boundary)
2. End with a symbolic constant (upper boundary)

Consider the following rules and recommendations:

- For arrays only used in a block, the constant must be defined as local constant.
  Access to global constants is only permitted for the definition of array limits for scaling different quantity structures.
  Refer to “RU004 Rule: Use only local variables”
- We recommend selecting DInt as the data type for the constant, to align to the data type for the variable of the loop index / array access.
  Refer to "PE008 Recommendation: Declare control/ index variables as “DInt”

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 of unknown size must be passed as a formal parameter, it is recommended to declare it 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 know-how protected blocks a recompile is not necessary as the size is not declared explicitly in the block interface.

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…

1. limit all strings to the necessary length.
2. use symbolic constants to specify the length.

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 transferred actual values must be checked for validity in the module in order to prevent uncontrolled program sequences and states.

In the event of invalid or implausible values, a corresponding message must be provided for the user.

Note Further information about the topic error handling are provided in "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 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.

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 via HMI/OPC UA/Web API shall be disabled.
Refer to “ES007 Rule: Explicit data access via HMI/OPC UA/Web API”.

Below is an overview of the permitted access settings:
(Reference: ✓ = Active / ✗ = Inactive / ★ = optional)

No.	Type	Accessible	Writable	Visible
1	Input	✓	✓	★
2	Output	✓	✗	★
3	InOut	✓	✓	★
4	Static interface	✓	✓	★
5	Static internal	✗	✗	✗
6	PLC data type	✓	✓	★
7	Global DB interface	✓	✓	★
8	Global DB internal	✗	✗	✗

1. Input parameter: Read and write access
2. Output parameter: Read access
3. InOut parameter: Read and write access
4. Static interface variables: Read and write access (Prefix `ext`)
5. Static internal variables: No access at all (Prefix `inst`, `stat`)
6. PLC data type definition: Read and write access
7. Global DB interface variables: Read and write access
8. Global DB internal variables: No access

For the distinction between interface/internal variables, also refer to rules:

- “DA006 Rule: Access static variables from within the block only”
- “NF007 Rule: Use prefixes”.

Note In the PLC data type editor, it can be useful to activate accessibility via HMI/OPC UA/Web API in order to enable access in principle.
When a PLC data type is used, it is determined whether access should actually be permitted.

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 is provided in "DA013 Rule: Report status/errors via `status/error`".

SE006 Rule: Write Error OB with evaluation logic

Unexpected PLC stops can be avoided by using error OBs.
If organizational blocks are used for error handling, these must also be evaluated.

Examples of possible error OBs:

- Time error OB (CYCL_FLT [OB 80])
- Diagnostic interrupt OB (I/O_FLT1 [OB82])
- Pull/plug interrupt OB (I/O_FLT2 [OB83])
- Rack failure OB (RACK_FLT [OB86])
- Programming error OB (PROG_ERR [OB121])
- I/O access error OB (MOD_ERR [OB122])
- ProDiag OB

Examples of possible minimum reactions:

- Bit Alarm
- ProDiag Alarm
- Program Alarm `Program_Alarm`
- Diagnostic buffer entry `Gen_UsrMsg` / `GEN_DIAG`
- ...

Justification: To ensure that errors are not permanently undetected and can be reacted to, the minimum requirement is the evaluation of an error and a message to the person responsible.

SE007 Rule: Use value ranges for Real/LReal comparisons

For comparing Real/LReal values only use compare operators like `<`, `>`, `<=`, `>=`, TIA Portal provides as well `IN_RANGE` and `OUT_RANGE` for several programming languages.

Justification: As Real/LReal are stored in the IEEE754 format, they have a limited accuracy. By using tolerance based comparisons, you avoid unexpected results due to potential rounding errors and representation discrepancies. Therefore `=` should not be used.

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 related blocks into a folder/group
- 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

"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 to show and hide 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.

Signalflow oriented (CFC - Continuous Function Chart)

CFC is used in particular for process engineering or structured automation solutions. For example, you link self-created blocks or instructions supplied by CFC with the operands of your target system. Even a complex user program remains clear with the help of a CFC chart. You can monitor process values as well as input and output parameters of the instructions and blocks online for testing purposes.

Cause Effect Matrix (CEM)

CEM is a programming language with which you can quickly and clearly define direct cause-effect relationships. You describe specific process events and define possible process reactions. You assign these to one another in a two-dimensional matrix.

DA003 Rule: Set/evaluate block properties

The following settings must be activated in the block properties:

1. **Auto numbering:** Blocks (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.
2. **IEC-Check:** To ensure the IEC conformant programming, the IEC-Check must be turned on. The compatibility of operands for comparison operations and arithmetic operations is checked in accordance with the IEC rules. This ensures type-compliant and type-safe programming. Incompatible operands must be converted explicitly.
   For example, implicit type conversions, where there is a risk of data loss (`USint` 0..255 <–> `Uint` -128..127), or slice accesses to numerical data types are marked as errors by the compiler.
3. **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. Refer to “AL001 Rule: Use multi instances instead of single instances”
- **Library conformance:** Refer to “RU004 Rule: Use only local variables“.

Figure 9-1: Auto numbering

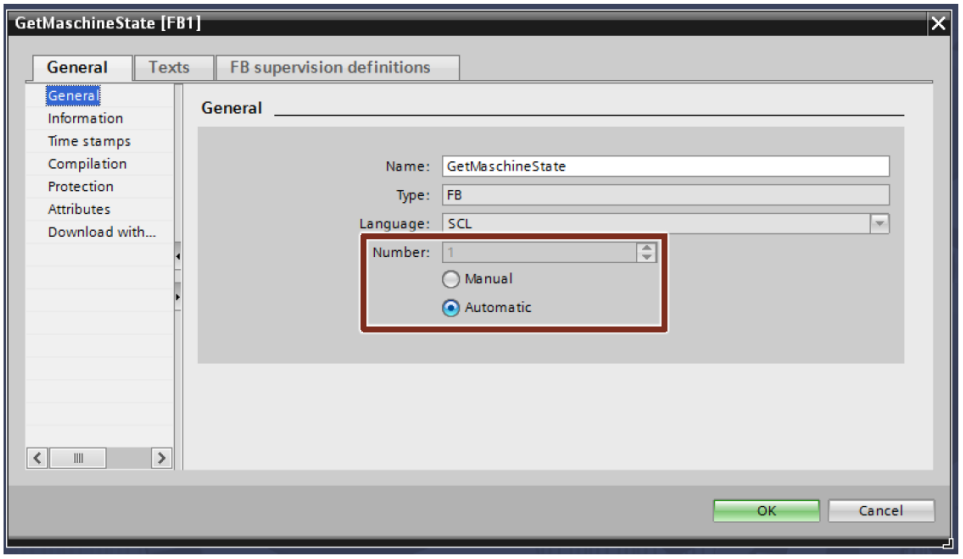


Figure 9-2: IEC-Check, ENO, Optimized access, Multiple instance capability

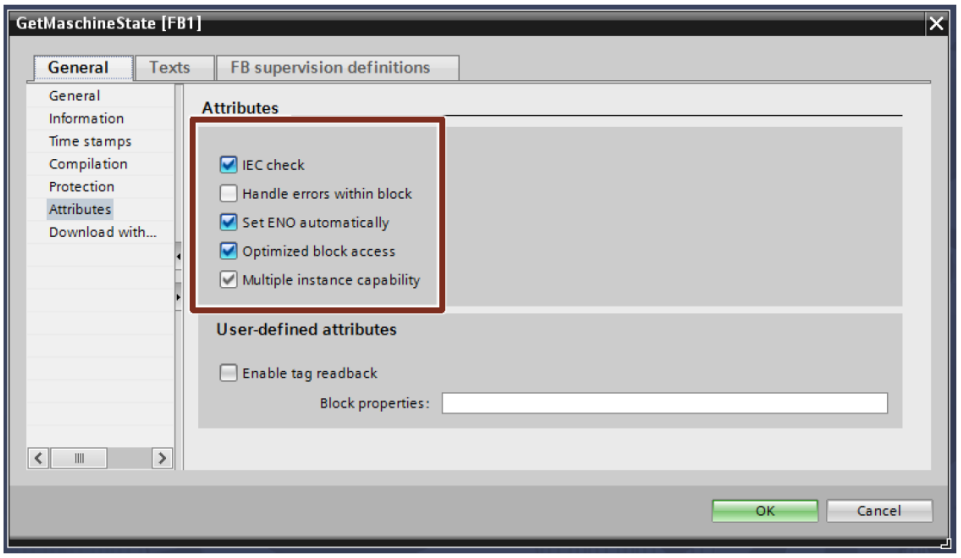
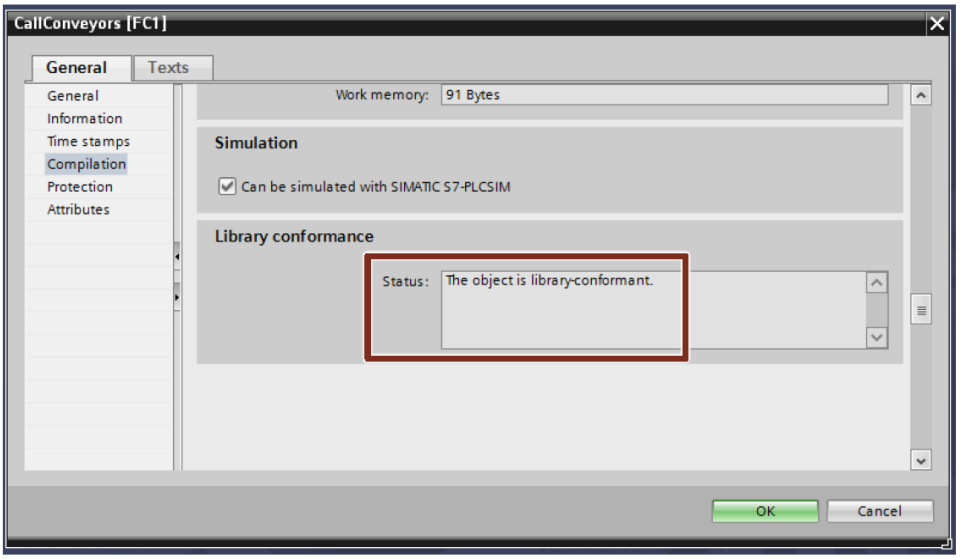


Figure 9-3: Library conformity



DA004 Rule: Use PLC data types

PLC data types shall be used for structuring 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 defined in the local data of a block or within a PLC data type in order to display variables within the block or PLC data type more clearly by grouping them if required.

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.

Note

Variables of PLC data types can be easily initialized with an assignment by providing an empty/unused variable of the same type and pre-assigning the corresponding start values. If the data type is changed, no further reworking is required for initialization.

DA005 Rule: Exchange data only via formal parameters

Data exchange within the PLC with FBs or FCs is always carried out via the block interface (input, output or in/out parameters).
If FBs or FCs are used as a call environment or for structuring the software and do not require an interface for reuse, formal parameters can be omitted. In this case, data is exchanged exclusively via direct access to global data. Mixing both types of access in one block is not permitted.

Justification: In terms of encapsulated modules, data is decoupled via the interface and dependencies are resolved. This ensures data consistency in the event of multiple calls (possibly at different program locations).
If a call environment/structural element does not require an interface for reuse, the effort can be reduced by not using formal parameters.

Note

It is acceptable to write directly to input variables of subordinate instances and to read from their output variables. No additional variables are required for this and unnecessary copying processes are avoided.

```
instCall.execute := TRUE;

instCall();

IF instCall.done THEN
  ; // next step

ELSIF instCall.error THEN
  ; // do something....
END_IF;
```


DA006 Rule: Access static variables from within the block only

The internal static data of a function block shall only be used within the block in which they have been declared.
If static variables are required as an interface to PLC-external systems, for example to the HMI faceplate associated with the module, these must be explicitly labeled.

Refer to:

- “NF007 Rule: Use prefixes”
- “SE004 Rule: Enable data access via HMI/OPC UA/Web API selectively”

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

1. 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.
2. 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 ensures, that all output values are consistent.

DA009 Rule: Keep used code only

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

Examples for violations:

- Never called blocks or technological objects
- Never used parameter
- Never executed program code
- Commented out code
- Never used PLC variables
- Never used user constants
- Never used PLC data type
- Never used Named value data types (NVT)
- 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 usage 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 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 parameter is used, then at least the output parameters `valid`, `error` and `busy` must be provided.

If a block that uses the enable-behavior contains other subordinate commands, these must also be implemented following the `Enable`- or `Execute`-behavior. Subordinate commands also need to provide a feedback, which can be used to determine the status of the command(s), for example:

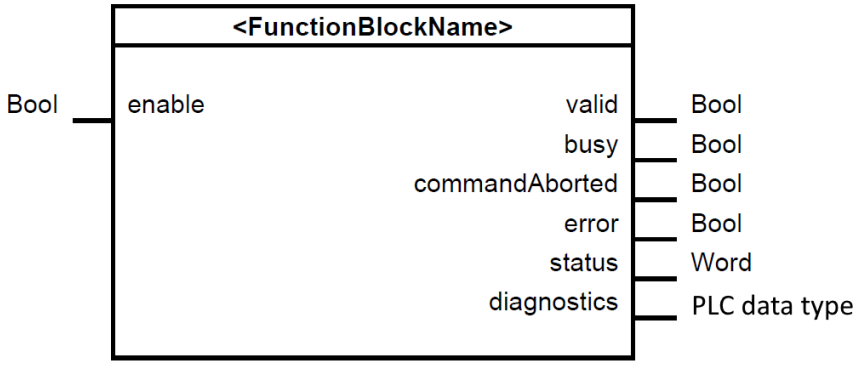
- Enable command:
  `commandA` → `commandBusy`
- Execute command:
  `commandA` → `commandBusy` & `commandDone`
- Multiple commands - additional parameter to distinguish the active command
  `commandA` → `commandASelected`
  `commandB` → `commandASelected`
  `commandAborted`

Table 9-1

Identifier	Data type	Description
Input parameters		
enable	Bool	All parameters are 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`).
commandA	Bool	**Optional** input: (name function-dependent) is implemented following `Enable`- or `Execute`-behavior for a subordinate command that is activated with a rising edge.

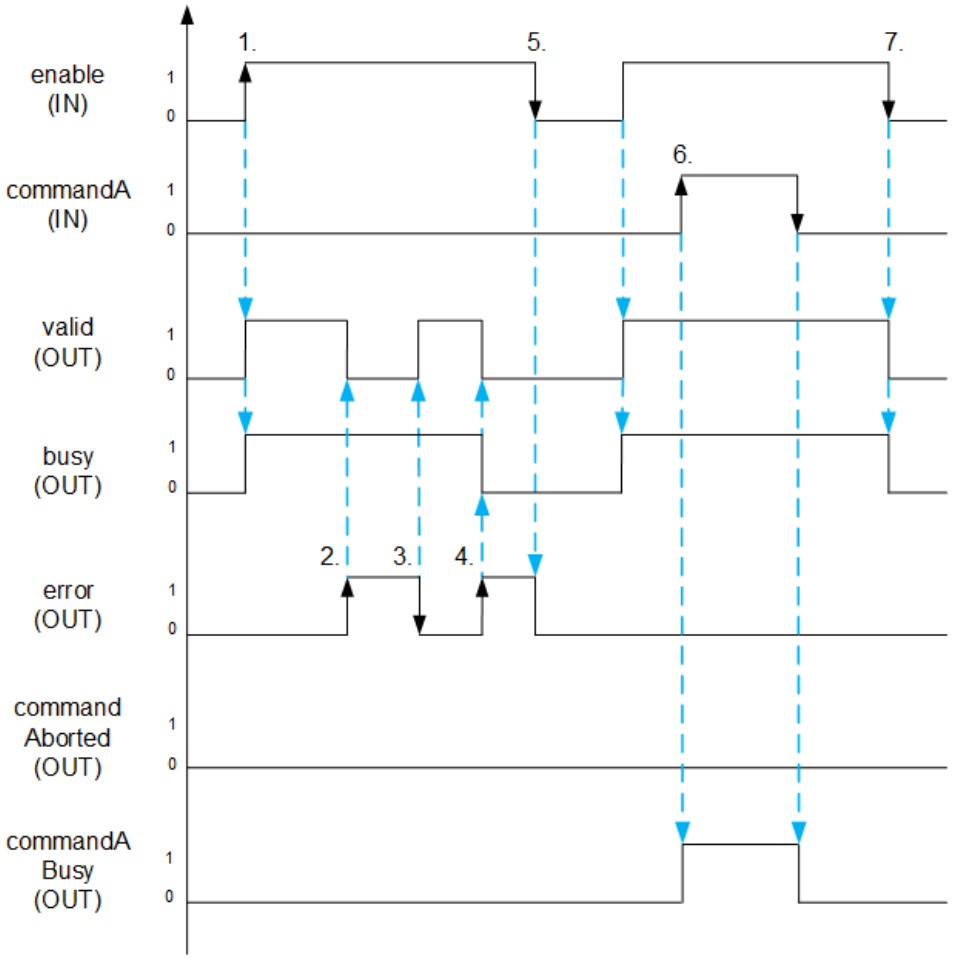
Identifier	Data type	Description
Output parameters		
valid or enabled	Bool	The outputs `valid`/`error` are mutual **exclusive**. The output is set if the output values are valid and the `enable` input is set. As soon as an error is detected the output `valid` is reset. According to PLCopen, `enabled` can also be used instead of `valid`.
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`/`error` are mutual **exclusive**. A rising edge of the output indicates that an error occurred during the execution of the FB.
status	Word	**Optional** output: error and status information of the block: This parameter has 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 stored. The structure of the diagnostic information is described in the recommendation “DA015 Recommendation: Pass underlying information”.
commandBusy	Bool	**Optional** output: indicates that a command of the FB is processed.
commandDone	Bool	**Optional** output: indicates that a command of the FB was processed (for `Execute` commands)
commandAborted	Bool	**Optional** output: indicates that the currently executed task of the FB was 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 was aborted by another command.
commandASelected	Bool	**Optional** output: (name command-dependent) displays the last active command of the FB and is reset with another command or `commandDone`. This parameter is required to distinguish between more than one subordinate commands.

Figure 9-4: Example block interface with enable



Signal diagram of a block with enable:

Figure 9-5



1. With a rising edge at `enable` the block is activated, `valid` and `busy` set to `TRUE` means, the block is active, no errors occurred, and the output signals of the FB are valid.
2. With `error` at `TRUE` the output `valid` is reset and all functions within the block are stopped. Because the error can be handled by the block itself the `busy` flag remains active.
3. After clearing the cause of the error (e.g. reestablishing a connection) `valid` becomes active again.
4. An error occurs, which can only be cleared by the user, then `error` must be set and `valid` and `busy` must be reset.
5. Only a falling edge at `enable` clears the current error, which can only be cleared by the user.
6. Subordinate commands such as `commandA` with enable behavior in the signal diagram have a feedback output about the status of the command - here `commandABusy`.
7. 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.

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 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 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 that the block is in its initial state every time a job is started and is executed independently of previous jobs.

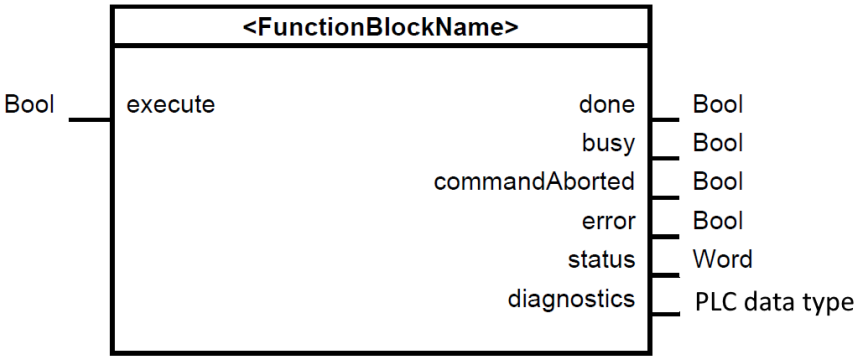
If the block is implemented according the PLCopen standard and the `execute` input parameter is used, then the output parameter `busy`, `done` and `error` must be used.

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 `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 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. The output `busy` is set with a rising edge on `execute`. The output is reset, if one of the outputs `done`, `commandAborted` or `error` is set.

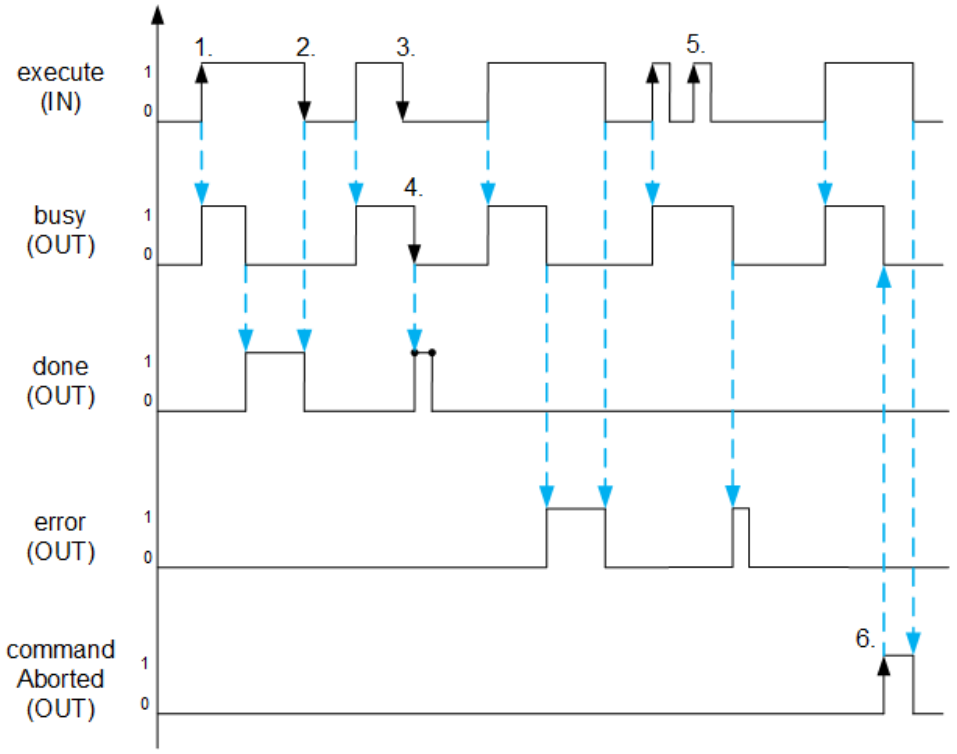
Identifier	Data type	Description
Output parameters		
error	Bool	The outputs `done`, `busy`, `commandAborted` and `error` are **mutual exclusive**. A rising edge of the output indicates that an error occurred during the execution of the FB.
commandAborted	Bool	The outputs `done`, `busy`, `commandAborted` and `error` are **mutual exclusive**. **Optional** output: indicates that the currently executed command was 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 was aborted by another command.
status	Bool	**Optional** output: error and status information of the block: This parameter has 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 stored. The structure of the diagnostic information is described in the recommendation “DA015 Recommendation: Pass underlying information”.

Figure 9-6: Example Block interface with execute



Signal diagram of a block with execute:

Figure 9-7



1. With a rising edge at `execute` the Block is activated, `busy` set to `TRUE` means, the block is active, no errors occurred, and the output signals of the FB are valid.
2. The outputs `done`, `error` and `commandAborted` are reset with a falling edge on `execute`.
3. The functionality of the FB is not stopped by a falling edge on `execute`.
4. If `execute` is already FALSE, then `done`, `error` and `commandAborted` are set only for one cycle.
5. 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.
6. 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.

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

DA013 Rule: Report status/errors via “status”/“error”

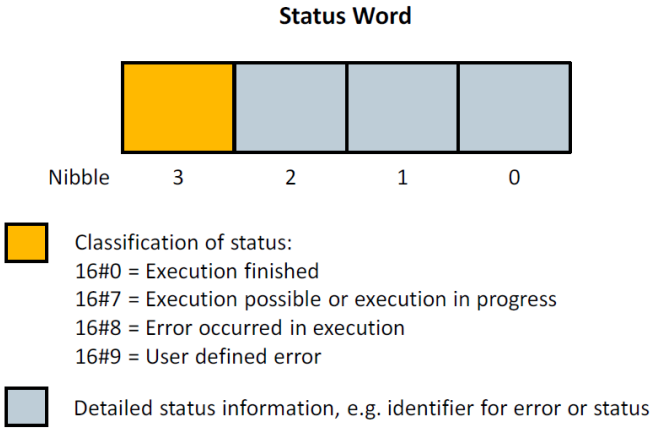
The block reports a unique status at its output parameter `status`, which provides information about the internal status of the block. The values must be defined as local symbolic constants in the block interface to avoid double usage and increase the readability.

If the block reports an error, the output parameter `status` and `error` shall be used. Following the below described status concept, the parameter of `error` is the most significant bit (MSB) of the `status`-word (bit 15). The remaining bits are 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 does not contain error information.

Figure 9-8



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: Status Codes

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

Table 9-4: Error Codes

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

Note

It is recommended to use Named value data types (NVT) or constants within the block to define the status/error code and name them correctly.
A prefix should therefore be used for the names, e.g.:

- Status codes: `STATUS_`, `INFO_`, `WARN_`, `WARNING_`
- Error codes: `ERR_`, `ERROR_`

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`. Furthermore, this diagnostic structure may contain additional values for diagnostic purposes, such as runtime information.

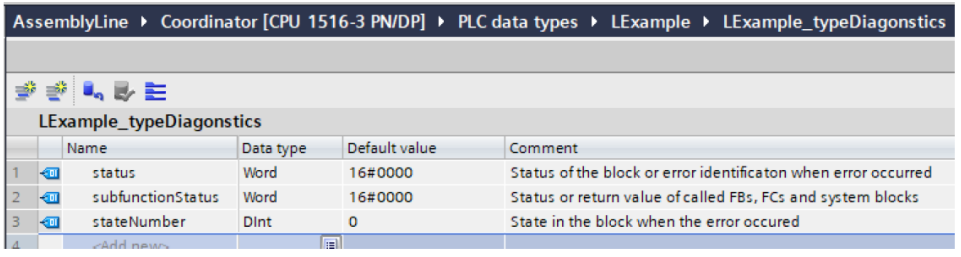
Note

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

System-based diagnostics (`ProDiag` or `Program_Alarm`) can also be used to output and log diagnostic information.

Example for a simple diagnostic structure:

Figure 9-9



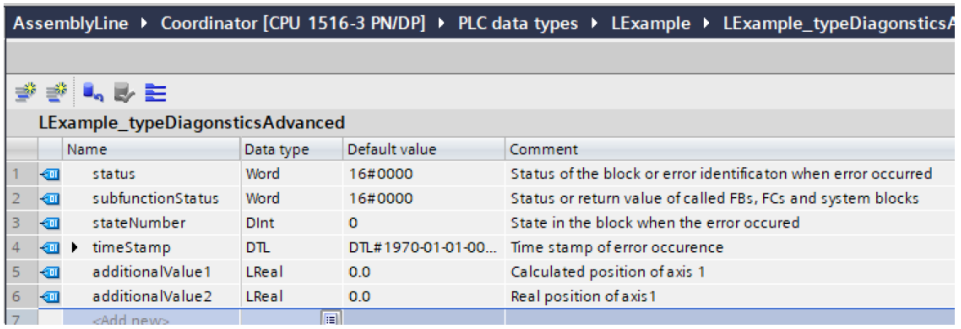
The simple diagnostic structure contains three parameters:

Table 9-5

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

Example for an extended diagnostic structure:

Figure 9-10



The variable `timestamp` used in the screenshot 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 `status` 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.

Note

The block templates with `enable` and `execute` show the use of a diagnostic structure in combination with a state machine. The block templates can be found in the copy templates in the library with general functions (LGF):

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

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.

DA019 Recommendation: Use of Named value data types

Use of Named value data types whenever possible values, e. g. for:

1. CASE instructions, e.g. in state machines.
2. Configuration parameters instead of integer
   example: `direction := nvtDirection#FORWARD` / `direction := nvtDirection#BACKWARD`
3. Status and diagnostic data

Refer to “RU005 Rule: Use symbolic constants”

Justification: This will ensure readability and traceability of the software.

Note Named value data types are only available within Software Units.

Example:

```
CASE #statMainState OF
  nvtStates#NO_OPERATION:
    REGION NO_OPERATION
      ; // No operational state
    END_REGION NO_OPERATION

  nvtStates#ENABLING:
    REGION ENABLING
      ; // Enabling state
      #statMainState := nvtStates#PROCESSING;
    END_REGION ENABLING

  nvtStates#PROCESSING:
    REGION PROCESSING
      ; // Processing state
      #statMainState := nvtStates#DISABLING;
    END_REGION PROCESSING

  nvtStates#DISABLING:
    REGION DISABLING
      ; // Disabling state
      #statMainState := nvtStates#NO_OPERATION;
    END_REGION DISABLING

  ELSE  // Statement section ELSE
    // Error, undefined state reached
    #status := nvtStates#ERR_UNDEF;
END_CASE;
```

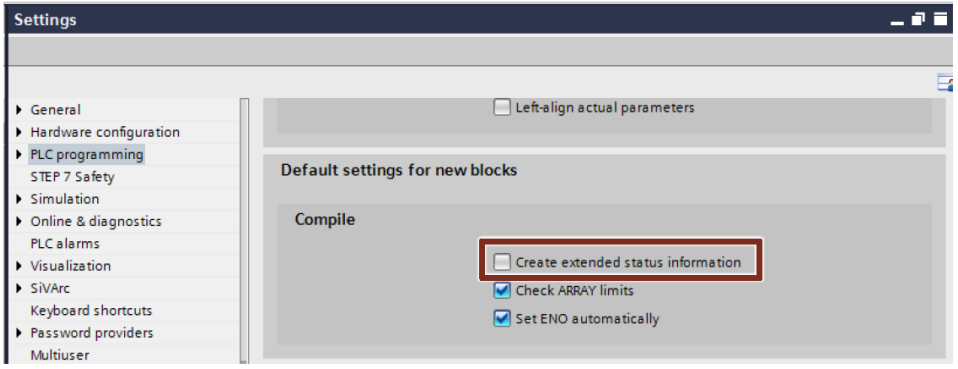
10 Performance

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 option “Create extended status information” may lead to a better performance in the productive operation. 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 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 passed. For this the actual parameters are not copied.

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

Note 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.

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 as formal parameter, 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 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 an input parameter, e.g. “mode”.

Justification: This prevents code fragments that are not 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.

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. For compatibility reasons, 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);
```

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 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 as the use of these system functions can have a negative impact on the overall program duration.

Runtime-intense instructions are instructions that

1. process large amounts of data, such as
   - “Serialize” / “Deserialize”
2. access the memory card, such as
   - “CREATE_DB” / “WRITE_DBL” / “READ_DBL” / “ATTR_DB” / “DELETE_DB”
   - “FileRead” / “FileWrite” / “FileDelete”
   - “RecipeExport” / “RecipeImport”
   - “DataLogCreate” / “DataLogOpen” / “DataLogClear” / “DataLogWrite” / “DataLogClose” / “DataLogDelete” / “DataLogNewFile”
   - Flash storage like the SIMATIC Memory Card is comparatively slow. The access is typically done asynchronously and may take several cycles, since larger data quantities are transferred.
3. read / resolve symbol data from load memory, such as
   - “GetSymbolName” / “GetSymbolPath” / “GetInstanceName” / “GetInstancePath”
   - “ResolveSymbols” / “MoveToResolvedSymbol” / “MoveFromResolvedSymbol” / “MoveResolvedSymbolsToBuffer” / “MoveResolvedSymbolsFromBuffer”

Justification: The use of these system functions can have a negative impact on the overall program duration.

Possible solutions:

- Do not call them cyclically
- Call them only if needed
- If needed regularly, call them every n'th-cycle
- If having high amount of calls, split them in several groups and call them in different cycles
- If needed only once, call them in the startup OB

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.

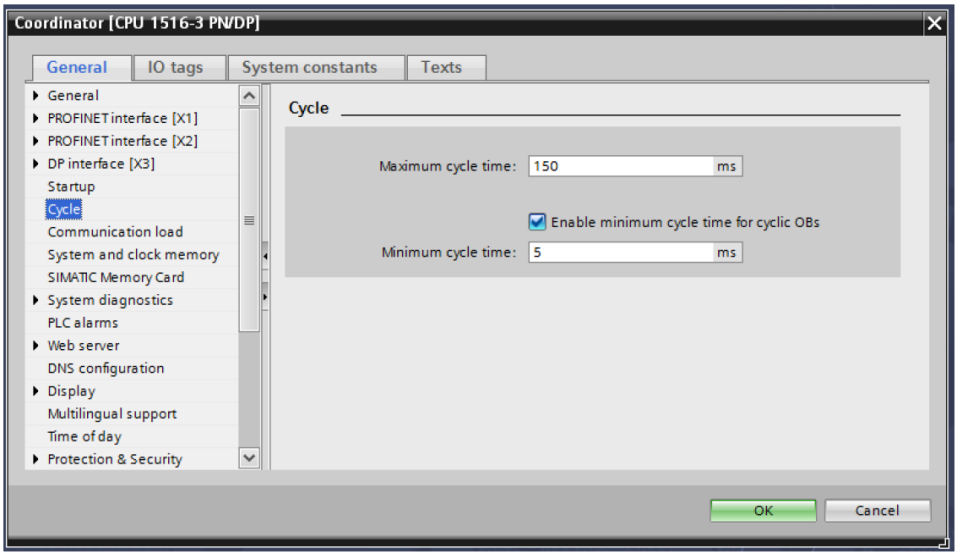
Justification: GRAPH and CFC as programming languages generate additional machine code and diagnostic information that require additional runtime. GRAPH is suitable for creating sequential machine sequences, while CFC is used for signal flow-oriented sequence programming.

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



PE017 Recommendation: Avoid multiple Access for IO / TO variables

In the case of multiple variable access to I/O areas or Technology Objects, it is recommended to copy the values once to internal temporary or static variables and work with these.

It should also be noted that:

- Data is not read more frequently than its actual update rate (e.g. in the motion cycle -> pre/post servo),
- Data is only read if it is required for program processing.
- Static data is only read on specific request (e.g. configuration values).

Justification: Accessing I/O data area and especially Technology Objects is significantly slower than accessing optimized data. Therefore, in the case of multiple accesses, copying once and working with the copy is more efficient.
In addition, working with the copy creates a process image and the module works with consistent data. If the function block is accessed several times, it could happen that the data of a technology object, such as the actual position, originates from different interpolator cycles and the application does not deliver the expected result.

11 CheatSheet

Block interface	In	Out	InOut	Stat	Temp	Const
	enable	**done**	**conveyorAxes** **instTimer**	**statState** **instTimer** **extInterface**	**tempIndex**	**MAX_VELOCITY**
Prefix	--	--	-- (default) "inst" (parameter-instance)	"stat" (default) "inst" (multi-instance) "ext" (external accessible interf.)	"temp"	--
Casing	camelCasing	camelCasing	camelCasing	camelCasing	camelCasing	UPPER_CASING

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

Programming style guide for SIMATIC S7 in TIA Portal S7-1200 / S7-1500

- 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**	--	UpperCamelCase
Library	**LCom**	"L_"	UpperCamelCase
Organization block	**Main**	--	UpperCamelCase
Function block	**HeatTank**	--	UpperCamelCase
Function	**CalculateTime**	--	UpperCamelCase
Global data block	**MachineData**	--	UpperCamelCase
Single instance data block	**InstHeater**	"Inst"	UpperCamelCase
Technology Object	**HeatingAxis**	--	UpperCamelCase
PLC tag table	**Sensors**	--	UpperCamelCase
Watch/Force table	**MachineState**	--	UpperCamelCase
Trace	**ConveyorSpeed**	--	UpperCamelCase
Measurement	**Temperature**	--	UpperCamelCase
PLC alarm text list	**ConveyorAlarms**	--	UpperCamelCase
Software unit	**Magazine**	--	UpperCamelCase
PLC data type	**typeDiagnostics**	"type"	lowerCamelCasing
Element in a PLC datatype	**stateNumber**	--	lowerCamelCasing
Named value data type	**nvtStatus**	"nvt"	lowerCamelCasing
Element in NVT	**STATUS_NO_ERROR**	--	UPPER_CASING
User irrelevant PLC Object	**Lib_UnpubObject**	"Unpub" / "unpub"	lower-/UpperCamelCasing

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

different Casings	Synonyms
UpperCamelCase	PascalCasing
lowerCamleCase	camelCasing
UPPER_CASING	CAPITALS / UPPER_SNAKE_CASING / ALL_CAPS / SCREAMING_SNAKE_CASE

12 Appendix

12.1 Service and support

SiePortal

The integrated platform for product selection, purchasing and support - and connection of Industry Mall and Online support. The SiePortal home page replaces the previous home pages of the Industry Mall and the Online Support Portal (SIOS) and combines them.

- Products & Services
  In Products & Services, you can find all our offerings as previously available in Mall Catalog.
- Support
  In Support, you can find all information helpful for resolving technical issues with our products.
- mySieportal
  mySiePortal collects all your personal data and processes, from your account to current orders, service requests and more. You can only see the full range of functions here after you have logged in.

You can access SiePortal via this address: sieportal.siemens.com

Technical Support

The Technical Support of Siemens Industry provides you fast and competent support regarding all technical queries with numerous tailor-made offers - ranging from basic support to individual support contracts.

Please send queries to Technical Support via Web form:
support.industry.siemens.com/cs/my/src

SITRAIN - Digital Industry Academy

We support you with our globally available training courses for industry with practical experience, innovative learning methods and a concept that's tailored to the customer's specific needs.
For more information on our offered trainings and courses, as well as their locations and dates, refer to our web page:

siemens.com/sitrain

Industry Online Support app

You will receive optimum support wherever you are with the “Siemens Industry Online Support” app. The app is available for iOS and Android:

12.2 Industry Mall

The Siemens Industry Mall is the platform on which the entire Siemens Industry product portfolio is accessible. From the selection of products to the order and the delivery tracking, the Industry Mall enables the complete purchasing processing – directly and independently of time and location:
mall.industry.siemens.com

12.3 Links and Literature

Table 12-1: Links and Literature

No.	Topic
\1\	Siemens Industry Online Support https://support.industry.siemens.com
\2\	Link to the entry page of the application example https://support.industry.siemens.com/cs/ww/en/view/81318674
\3\	Programming Style guide for S7-1200/S7-1500 https://support.industry.siemens.com/cs/ww/en/view/109478084
\4\	Programming Guidelines for S7-1200/S7-1500 https://support.industry.siemens.com/cs/ww/en/view/90885040
\5\	Programming Guidelines Safety S7-1200/S7-1500 https://support.industry.siemens.com/cs/ww/en/view/109750255
\5\	TIA Portal Settings for Style Guide (Login required) 81318674_TIAPortalSettingsForStyleGuide.zip
\5\	Test Suite Advanced: Example for checking TIA Portal projects https://support.industry.siemens.com/cs/ww/en/view/109779806
\7\	Project Check for TIA Portal: Check against programming style guides https://support.industry.siemens.com/cs/ww/en/view/109741418
\7\	SIMATIC Project Insight https://support.industry.siemens.com/cs/ww/en/view/109818320
\6\	SIMATIC S7-1200/ S7-1500 Compare list for programming languages https://support.industry.siemens.com/cs/ww/en/view/86630375
\7\	Standardization guideline https://support.industry.siemens.com/cs/ww/en/view/109756737
\8\	Multiuser Engineering with TIA Project Server https://support.industry.siemens.com/cs/ww/en/view/109740141
\9\	Guideline on Library Handling in Tia Portal https://support.industry.siemens.com/cs/ww/en/view/109747503
\10\	Provide user defined documentation https://support.industry.siemens.com/cs/ww/en/view/109755202/114872699275
\11\	TIA Portal Add-In Code2Docu for generating documentation https://support.industry.siemens.com/cs/ww/en/view/109809007
\12\	Library of General Functions (LGF) for TIA Portal and SIMATIC S7-1200/ S7-1500 https://support.industry.siemens.com/cs/ww/en/view/109479728

12.4 Changelog / History

Table 12-2: Changelog

Version	Date	Changes
V2.1	04/2025	**General** - Wording alignments - Add programming languages SFC, CFC and CEM to style guide - Add Named value data type (NVT) to style guide - Update CheatSheet **NEW RULES** SE007: - Use value ranges for Real/LReal **NEW RECOMMENDATIONS** DA019: - Use of Named value data type PE017: - Avoid multiple Access for IO / TO variables **UPDATED RULES/RECOMMENDATIONS** ES*: - Add where the rules effects TIA Portal or the Project ES007: - Adjust wording to be more precise and clearer about the rule GL003: - Change from “all project languages” to “all active project languages” NF001 / NF002: - Wording adjustments to be more general NF001 / NF005: - Add namespace to rule description NF003: - Document developer information - remove / set to optional for redundant infos NF004: - Extend example table with… - Global constants table for status/error/… - Named value data type NF005/NF006/NF008: - Rename rules to more understandable names - PascalCasing –> UpperCamelCase - camelCasing –> lowerCamelCase - CAPITALS –> UPPER_CASING - Extend texts with multiple names for the different namings NF006: - Add Named value data type NF007: - Add prefix `ext` for static interface variables - Add prefix `nvt` for Named value data types - Add prefix `unpub` for unpublished / internal PLC objects - Derived data blocks from PLC data type do not get a prefix `Inst` - PLC data type prefix is just needed for its declaration - Numbering the example table - Extend with notes for a. Formal parameters b. Internal static variables c. Interface statics (`ext`) d. Named value data type (`nvt`)

Version	Date	Changes
		e. Unpublished / internal PLC objects (`unpub`) NF008: - Add Named value data type members NF010: - Adjust wording to be more precise and clearer about the rule - “overall length of a **single** identifier” NF011: - Add `Lim` for Lmit to abbreviation table NF014: - Change from “Use of line comment // only” to “Preferable use of line comment //” - Insert justification and rework note box - Rework “Line breaks in partial conditions” - Number the topics and rework the examples RU001: - Rename to “Enable support for different target systems” because inserting virtual PLC support - Change from `Rule` to `Recommendation` because it potentially reduces the level of know-how protection if used RU002: - Change wording to precise the meaning for versioning with libraries - Improve list and table for “Version numbers and their use” - Add `Named value data type` RU003: - Add `Named value data type` RU004: - Add PLC data types as they belong as well to the rule - Split topics into ordered and unordered list RU005: - Remove `Local` and add `Named value data type` for symbolic constants - Add example for NVT RU006: - Extend examples with `Pointer` - Extend with system constants AL002: - Split topics into ordered and unordered list - Insert references to other rules AL003: - Adjust wording to be more precise and clearer about the rule - Applies for “Array of unknown size” AL004: - Split topics into numbered list SE001: - Adjust wording to be more precise and clearer about the rule SE004: - Adjust wording to be more precise and clearer about the rule - Insert table with maximum permitted settings and description - Insert differentiation for Interfaces within static area (e.g. PLC external systems) - Insert reference to dependent rules SE006: - Adjust wording to be more precise and clearer about the rule - Insert examples for Error OBs and Error handling DA002: - Add CEM and CFC as programming language to recommendation DA003:

Version	Date	Changes
		- Remove OB from auto numbering list - OB number need to be adjusted for call sequence - First part to numbered list to separate rules and user actions - Extend description regarding IEC Check setting - Insert Reference to dependent rules DA004: - Add `struct` for the use in PLC data types - Add note for initialization with empty PLC data types DA005: - Adjust wording to be more precise and clearer about the rule - Insert differentiation for interfaces within static area (e.g. PLC external systems) - Insert note and example for multi-instance access DA006: - Insert differentiation for Interfaces within static area (e.g. PLC external systems) DA008: - Split topics into ordered list DA011/DA012: - Refine description, timing diagrams and description listing - Insert description for subordinated commands DA014: - Extend Error and Status ranges with recommendation for constant prefixes DA015: - Extend note for system-based diagnostics (`ProDiag` or `Program_Alarm`) PE014: - Rework to list - Extend examples - Extend with resolve symbolic names PE015: - Add CFC to rule
V2.0	05/2020	- Categorize according to Workflow - Extend topics for Performance and Design-/ Architecture - Extend Programming guidelines - Amendment of Justifications - Review of Cheat sheet
V1.2	10/2016	Adaptations and Corrections