SIEMENS

Programming Styleguide•06/2015

Programming Styleguide for S7-1200/S7-1500

TIA Portal

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

Warranty and Liability

Note

The programming guidelines do not purport to cover all details or variations in equipment, nor do they provide for every possible contingency. The programming guidelines do not represent customer-specific solutions. They are only intended to provide support for typical applications. You are responsible for the correct operation of the described products. These programming guidelines do not relieve you of the responsibility of safely and professionally using, installing, operating and servicing equipment. When using these programming guidelines, you recognize that we cannot be made liable for any damage/claims beyond the liability clause described. We reserve the right to make changes to these programming guidelines at any time and without prior notice. If there are any deviations between the recommendations provided in this programming guideline and other Siemens publications – e.g. catalogs – the contents of the other documents have priority.

We do not accept any liability for the information contained in this document.

Any claims against us – based on whatever legal reason – resulting from the use of the examples, information, programs, engineering and performance data etc., described in this programming guideline will be excluded. Such an exclusion will not apply in the case of mandatory liability, e.g. under the German Product Liability Act ("Produkthaftungsgesetz"), in case of intent, gross negligence, or injury of life, body or health, guarantee for the quality of a product, fraudulent concealment of a deficiency or breach of a condition which goes to the root of the contract ("wesentliche Vertragspflichten"). The compensation for damages due to a breach of a fundamental contractual obligation is, however, limited to the foreseeable damage, typical for the type of contract, except in the event of intent or gross negligence or injury to life, body or health. The above provisions do not imply a change of the burden of proof to your detriment.

Any form of duplication or distribution of these programming guidelines or excerpts hereof is prohibited without the expressed consent of Siemens.

Security information

Siemens provides products and solutions with industrial security functions that support the secure operation of plants, solutions, machines, equipment and/or networks. They are important components in a holistic industrial security concept. With this in mind, Siemens' products and solutions undergo continuous development. Siemens recommends strongly that you regularly check for product updates.

For the secure operation of Siemens products and solutions, it is necessary to take suitable preventive action (e.g. cell protection concept) and integrate each component into a holistic, state-of-the-art industrial security concept. Third-party products that may be in use should also be considered. For more information about industrial security, visit http://www.siemens.com/industrialsecurity.

To stay informed about product updates as they occur, sign up for a product-specific newsletter. For more information, visit https://support.industry.siemens.com.

Table of Contents

1 Introduction

When programming SIMATIC controllers, the task of the programmer is to create as clear and readable a user program as possible. Each user uses its own strategy, for example, how to name tags or blocks or the way of commenting. The different philosophies of the programmers create very different user programs that can only be interpreted by the respective programmer.

Advantage of a consistent programming style

If several programmers work on the same program, it is recommended to stick to a joint and coordinated programming style. This offers the following advantages:

- consistent continuous style
- easily readable and comprehensible
- simple maintenance and reusability
- easy and quick troubleshooting and error correction
- efficient working at the same project with several programmers

Objective of the programming styleguide

Note The programming guidelines described here are a mere suggestion for sticking to a consistent programming style. It is up to you which rules and recommendations you consider sensible and which ones you use or not. However, please note that the rules and recommendations described here are adapted to one another and do not interfere with each other.

The programming guidelines described here help you create a consistent program code which can be better maintained and reused. That is, errors can be detected early on (e.g. by means of a compiler) or avoided.

The source code must have the following properties:

- consistent continuous style
- easily readable and comprehensible

For maintenance and clarity of the source code it is initially necessary to stick to a certain external format. However, optical effects - for example, a consistent number of space characters before each comma - contribute only little to the quality of the software. It is much more important to find rues, for example, that support the developer in the following way:

- Avoiding typing errors and slips which the compiler then interprets wrongly.
  Objective: the compiler shall detect as many errors as possible.
- Support from the program code for diagnosing program errors, such as reusing a temp tag beyond one cycle, for example.
  Objective: the code points at problems early on.
- Consistent standard applications and libraries
  Objective: familiarization shall be easy and reusability of program code increased.
- Simple maintenance and simplification of further development
  Objective: modification of program code in the individual modules that may include functions (FCs), function blocks (FBs), data blocks (DBs), organization blocks (OBs) in libraries or in the project shall have minimal effects on the overall program/overall library. Modifications of program code in the individual modules shall be performed by different programmers.

Validity

This document applies for (customer) application examples, such as libraries created in the IEC 1131-3 (DIN EN 61131-3) programming languages Structured Text (ST), Ladder Diagram (LAD), Function Block Diagram (FBD) and Statement List (STL).

Topics not covered in this application

This document does not contain a description of:

- STEP 7 programming
- commissioning SIMATIC controllers

Basic knowledge of these topics is assumed.

2 Explanation of Terms

Rules / recommendations

Specifications are divided into recommendations and rules.

Rules are binding specifications and must be met. They are mandatory for reusable and performant programming. In exceptional cases, rules may also be violated. However, this must be documented accordingly.

Recommendations are specifications which, on the one hand, serve for a consistent code and, on the other hand, are intended as support and information. Recommendations should principally be followed; however, there may be cases where the recommendation is not followed. This may be due to efficiency, but also because the code is more readable otherwise.

Performance

Performance of an automation system specifies the processing time (cycle time) of a program.

A performance loss refers to the possibility of reducing processing time, hence the cycle time of a program run by means of applying the programming rules and skillful programming of the user program.

Identifier / name

It is important to differentiate between name and identifier. The name is part of an identifier that describes the respective meaning.

The identifier is composed of ...

- prefix
- name
- suffix

Abbreviations

The following abbreviations are used within the text:

Table 2-1: Abbreviations for blocks

Abbr.	Type
OB	Organization block
FB	Function block
FC	Function
DB	Data block
TO	Technology object
SFB	System function block
SFC	System function

Terms for tags and parameters

When it is about tags, functions, and function blocks, many terms are repeatedly used differently or even incorrectly. The following figure clarifies these terms.

Figure 2-1: Terms for tags and parameters

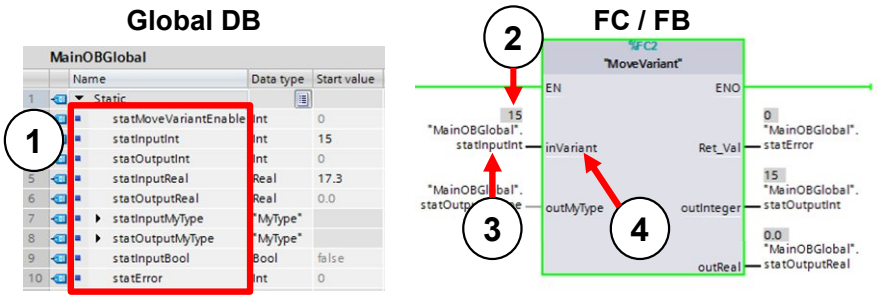


Table 2-2: Terms associated with tags and parameters

	Term	Description
1.	Tag	Tags are labeled by a name/identifier and assign an address in the memory of the controller. Tags are always defined with a certain data type (Bool, Integer, etc.): • PLC tags • Single tags in data blocks • Complete data blocks
2.	Tag value	Tag values are values stored in a tag (e.g., 15 as value of an Integer tag).
3.	Actual parameter	Actual parameters are tags interconnected at the interfaces of instructions, functions, and function blocks.
4.	Formal parameter (transfer parameter, block parameter)	Formal parameters are the interface parameters of instructions, functions, and function blocks (Input, Output, InOut, and Ret_Val).

3 General Specifications

Generally, it should be noted that the names used are always unique regarding functionality and used interface type.

That is, if the same name is used, the functionality behind it should also always be the same.

The basis for this document is the programming guideline for S7-1200/1500. It describes the system properties of the S7-1200 and S7-1500 controllers and how to program them in an ideal way.

Note Programming Guideline for S7-1200/S7-1500

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

3.1 Specifications and customer requirement

Rule: documenting rule violations

Each time a rule is violated, it becomes necessary to document this at the respective location in the program code.

For customer projects, the requests of the end customer take priority. If the customer asks for changes or deviations from these programming guidelines, this takes priority. The rules defined by the customer must be documented in suitable format in the source text.

3.2 Settings in TIA Portal

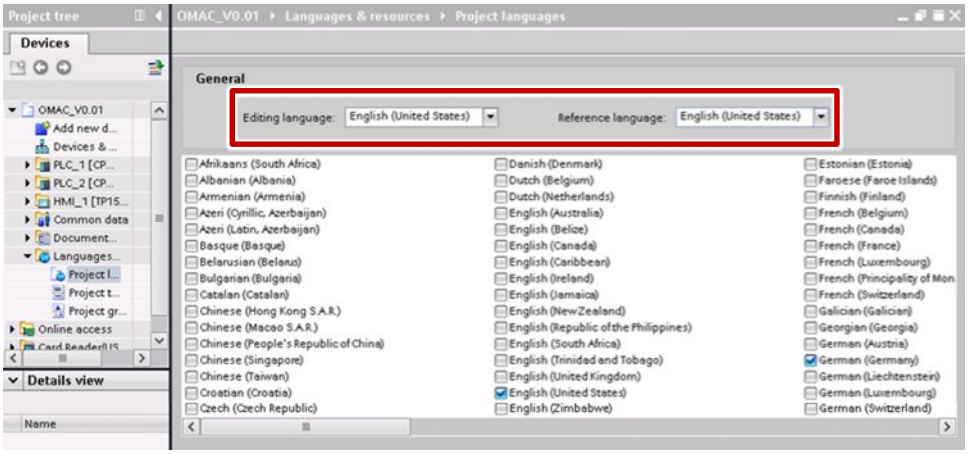
Rule: consistent language

The language must always be consistent in PLC programming as well as the HMI. That is, languages must not be mixed within a project at all (e.g. English as editing language and German comments in the blocks, or French texts in the English language area of the HMI).

Rule: editing and reference language: English (United States)

Unless explicitly asked for by the customer, “English (United States)” must be used as editing and reference language. Program code and all comments hence also come in English.

Figure 3-1: Editing and reference language



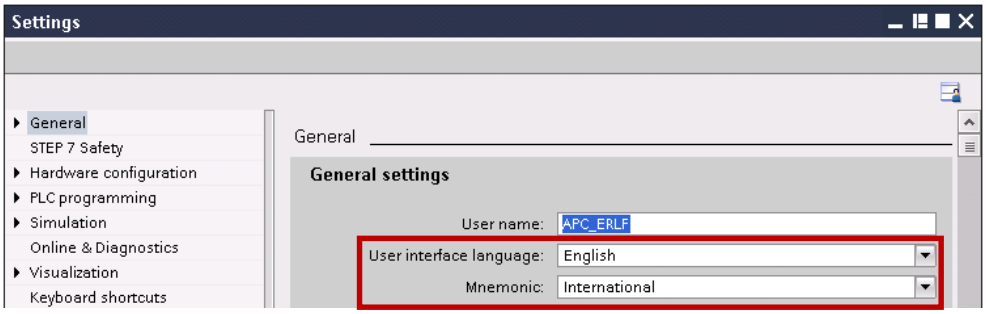
Recommendation: HMI language: English (United States)

The user interface in the TIA Portal should be set to English. All newly created projects then automatically come in the editing and reference language *English (United States)*. If in contrast, *German* has been selected as the HMI language, all projects are crated in the editing and reference language German.

Rule: Mnemonic: International

Mnemonic (language setting for programming languages) must be set to *International*.

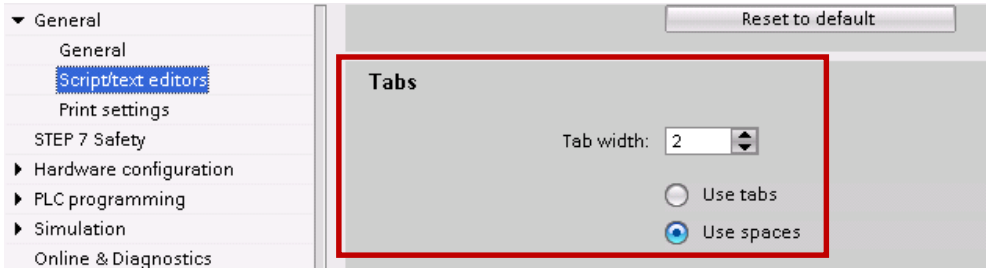
Figure 3-2: Language setting and Mnemonic TIA Portal



Rule: Tabs: 2 space characters

Tabs are not permitted in the text editors. Indentations must be created with two space characters. The respective setting must be made in TIA Portal.

Figure 3-3: TIA Portal Tab settings



3.3 Identifier

3.3.1 Formatting

Rule: English identifier

The name in identifiers (blocks, tags, etc.) must be entered in English language (*English – United States*)). The name reflects meaning and purpose of the identifier in the context of the source code.

Rule: unique identifiers

It is not allowed to use several identifiers with the same name that only differ in upper and lower case. The notation of an identifier is maintained in all blocks and sources.

Rule: identifier in camelCasing notation

If no other rule for the notation of an identifier is noted in the programming styleguide, the respective identifier is written in camelCasing.

The following rules apply for camelCasing:

- initial letter is written in small print.
- no separators (such as hyphen or underscore) are used.
- if an identifier consists of several words, the initial letter of each word is written as a capital letter.

Recommendation: Identifier: max. 24 characters

The identifier of tags, constants or blocks should not exceed 24 characters.

Rule: No special characters

No language-specific special characters, such as ä, ö, ü, à, etc., and no space characters are used.

Special characters between prefix and identifier are not allowed.

Recommendation: no separator

Separator (underscore) should not be used in identifiers; the length of the identifier should be kept short.

Example

temporary tag: tempMaxLength

Rule: meaningful identifier

For identifiers consisting of several words, the sequence of the words should be selected like that in spoken language.

3.3.2 Abbreviations

Recommendation: permitted abbreviations

To save characters for a tag name and increase the readability of the program, the uniform abbreviations in Table 3-1 should be used.

Table 3-1: Standardized abbreviations

Abbr.	Type
Min	Minimum
Max	Maximum
Act	Actual, current
Next	Next
Prev	Previous
Avg	Average
Diff	Difference
Pos	Position
Ris	Rising edge
Fal	Falling edge
Sim	Simulated
Sum	Sum
Old	Old value (e.g. for edge detection)

Recommendation: only one abbreviation per identifier

Several abbreviations should not be used in direct sequence.

4 PLC Programming

4.1 Program blocks and sources

4.1.1 Block name and numbers

Recommendation: short, functional block name

The name of the block is kept as short as possible and does not contain any information regarding its functionality.

Rule: identifier of blocks start with a capital letter

Identifiers of blocks (OBs, FB, FCs, DB, instance DB, TOs, etc.) start with a capital letter to achieve a consistent representation of the names in TIA Portal.

Rule: prefix 'inst' / 'Inst' for instances

Instances (single-instance, multi-instance) received 'inst' / 'Inst' as prefix.

Example

Single-instance: InstPidHeater (capital letter → own block)
Multi-instance: instTimerMotor (lower case → within one instance)

Recommendation: blocks with auto numbering

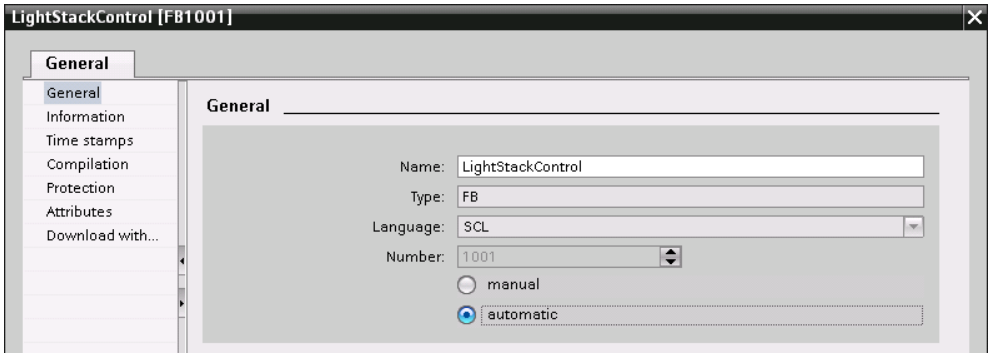
Blocks are only delivered with active automatic number assignment.

The following procedure is recommended when a certain block number shall be used at a block:

Table 4-1: Procedure number assignment

No.	Action
1.	Setting the number assignment to *manual*
2.	Assigning the desired block number (e.g. 1001)
3.	Adopting the properties with *OK*
4.	Open the block properties again
5.	Setting the number assignment to *automatic*
6.	Adopting the properties with *OK*

Figure 4-1: Block properties number assignment



4.1.2 Formatting

Recommendation: line length in the program editor: max. 80 characters

The line length in the source text must be limited to 80 characters in print format for better readability.

4.1.3 Programming

Rule: not using sources

To be able to use the full functionality of the TIA Portal with auto-complete and guarantee easy debugging, only blocks are used. The detour via source editing in an external editor and later source import must not be used.

Recommendation: preferential use of SCL

SCL should be selected as programming languages for blocks. SCL offers the best readability among the programming languages and, at the same time, has no performance disadvantages as opposed to other SIMATIC PLC programming languages.

If individual blocks shall be interconnected, for example in an OB, programming language LAD or FBD should be chosen. LAD or FBD should also be chosen even if a block mostly consists of binary logic operations. In these cases, selecting the programming language LAD or FBD provides the service staff with easier diagnostics and quicker overview.

Rule: Multi-instances instead of single-instances

Multi-instances are preferably used instead of single-instances. Closed functions can be created, such as an FB with integrated timer for time monitoring.

Recommendation: DBs in the load memory only in exceptional cases

Data blocks are always stored in the work memory of the CPU. Using the load memory for storing the data blocks is only permitted in exceptional cases. Exceptions are, for example, storing large amounts of measured data or a recipe management.

Rule: within a block, only local tags are used

Tags are only used locally. Access to global data is not permitted within FCs and FBs. This includes the following:

- Access to global DBs and use of individual instance DBs
- Access to tags (tag table)
- Access to global constants

Rule: do not define important test tags of blocks as temp

To facilitate the testability of FCs and FBs, particular attention must be placed on observability of tags in the watch and force tables.

This requires defining internal tags, inputs and outputs in suitable form (no temp tag). They need to give information about the state and processes of the functions. This includes, for example, the last editing state, or the current step number.

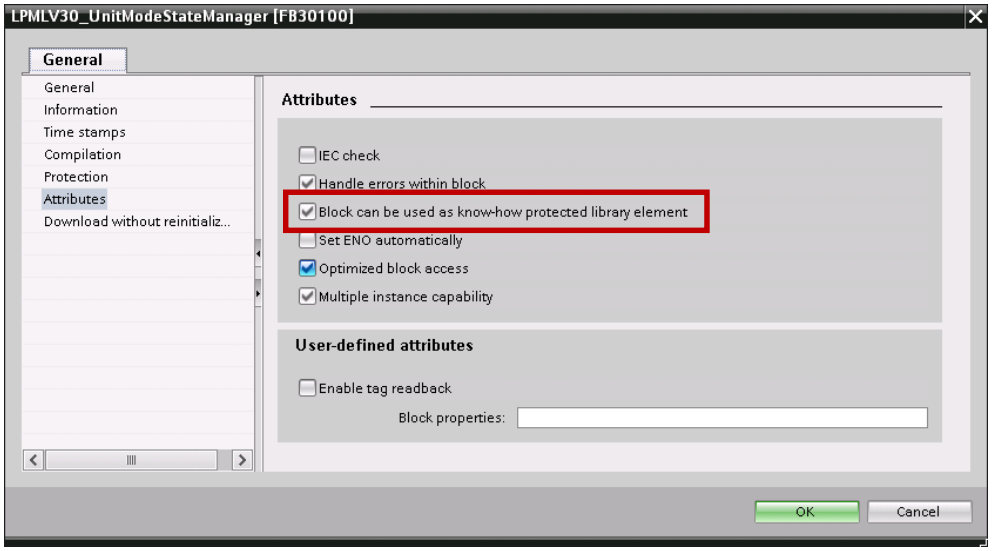
Temp tags cannot be monitored in force and watch tables or HMI.

Rule: 'Block can be used as know-how protected library element'

For an FB or FC it must be ensured that in the properties of the block, the "Block can be used as know-how protected library" check box in Attributes has been activated by the system (automatically when compiling).

This requires that the block has been programmed modular and must not use any global constants or tags.

Figure 4-2: Block properties - Attributes



4.1.4 Comments

Two types of comments must be differentiated:

- block comment (describes what a function or a code section does)
- line comment (describes the code of an individual line

Recommendation: Comments

A block comment must be set in front of the respective code section in one or several rows.

Recommendation: Line comments

A line comment must be placed at the end of the code line if possible, otherwise in front of the respective code line.

Recommendation: only use // comments

To facilitate commenting out code sections during debugging, only // comments should be used in the PLC code.

Rule: use template for block description

Each block is described in a description header in program code (SCL) or in the block comment (LAD, FBD). The description contains the following points:

- name of the library
- Tested PLCs with firmware version (e.g. S7-1511 V1.6)
- TIA Portal version when created
- Restrictions for usage (e.g. certain OB types)
- Requirements (e.g. additional hardware)
- Description of the functionality
- Version of the block with author and date
- Changes made

Template for block header

```
//===========================================================================
// Company
// (c)Copyright (year) All Rights Reserved
//---------------------------------------------------------------------------
// Library:       (that the source is dedicated to)
// Tested with:   (test system with FW version)
// Engineering:   TIA Portal (SW version)
// Restrictions:  (OB types, etc.)
// Requirements: (hardware, technological package, memory needed, etc.)
// Functionality: (that is implemented in the block)
//---------------------------------------------------------------------------
// Change log table:
// Version  Date        Expert in charge Changes applied
// 01.00.00 dd.mm.yyyy (Name of expert)  First released version
//===========================================================================
```


4.1.5 Formal parameters: Input, Output und InOut

Rule: No prefix for formal parameters

No prefix is used for formal parameters (*Input, Output* and *InOut)* of FCs/ FBs. If structures for transfer and output tags are used, the individual elements also have no prefixes.

Rule: data exchange via block interfaces

If data is required in several FB or FCs, data exchange is handled via the block interfaces (*INPUT*, *OUTPUT* and *InOut* interface).

Direct access to *Static* tags outside the FB is prohibited.

Recommendation: using elementary data type as In / Out or InOut

For elementary data types (e.g. type WORD, DWORD, REAL, INT, TIME) the *Input* or the *Output* interface type should be used.

For elementary data types, the *InOut* interface type is only used if a value is edited in writing outside as well as inside of a block.

Recommendation: transfer many tags as structured tags

If many parameters are transferred, it should be attempted to encapsulate these in a PLC data type. This PLC data type should then be declared as *InOut* tag. Examples for such PLC data types are configuration data, actual values, setpoint values, outputs of the current state of the function block, etc.

For frequently changing control or status tags, it may make sense to declare these directly as elementary *Input* or *Output* tags for simple access in LAD / FBD outside such a PLC data type.

Recommendation: Transfer structured tags as InOut

For structured tags (e.g. of type ARRAY, STRUCT, STRING, …) and PLC data types, the *InOut* interface type should generally be used.

For identical optimization setting of the interconnected data and the called block, this saves copying within the CPU unlike, for example, for input tags. Instead of copying, it is worked directly with the pointer reference to the data. Furthermore, using a reference saves storage space in the load memory.

Note

For *InOut* interfaces, setting the optimization at the block and at the interconnected data must be paid attention to.

Only if the optimization settings match (data optimized and block optimized, or data not optimized and block not optimized) no local copy of the data is created by the system.

If the optimization settings differ, from arrays at least one element is always copied into the stack, from other data types always the complete data. This makes the performance advantage of InOut interfaces void.

Note

Further information is available in the “Programming Guideline for S7-1200/1500” in chapter “Block interfaces”.

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

Recommendation: writing output tags only once per cycle

Per editing cycle, an *Output* tag should only be assigned a new value once at the end of the block. This ensures that all outputs can be kept as consistent as possible.

To achieve this, a tag can be created in the *Temp* or *Static* area which then represents a representative of the output within the block. At the end of the block, this representative tag is then assigned to the real *Output* tag of the block.

4.2 Tag declaration

4.2.1 Static and temp

Rule: Static tags are only called locally

The static data (*static*) of a function block is not accessed outside of this block. This applies in particular when calling and respectively interconnecting the instance data of the block.

Rule: prefix static tags: stat; temp tags: temp;

To be able to separate static and temp tags clearly from the transfer and output parameters in the code, the prefixes in Table 4-2 are used. This makes it easier for the user of a block to differentiate between interface tags and internal tags. The access rights for the user can hence be immediately defined and detected.

The prefixes do not apply for global DBs and PLC data types, but only for blocks which contain a complete interface.

Table 4-2: prefixes for tags

Prefix	Type
stat	*Static* tags → No outside access permitted in the instance data
temp	Temp tags → No outside access permitted in the instance data
	Input and *Output* tags (no prefix) → Access in instance data possible from outside
	Output tags (no prefix) → Changing the interconnected data possible at any time for the user as well as through the block

4.2.2 Constants

Rule: Identifier of constants always in CAPITAL LETTERS and underscore

The names of the constants are always in upper case. To detect individual words or abbreviations, underscore should be used between individual words or abbreviations.

Rule: use only local constants

To guarantee later usage of the blocks in a library, only local constants are used in the blocks.

This guarantees that errors cannot occur during compilation in the user program due to missing program parts.

If local constants shall be provided to the user of the block, these must also be created as global constants. The name of the global constants should also contain a reference to the block or the library.

This applies in particular to constants that identify defined values at block outputs, such as error numbers.

Global user constants can be created as *PLC tag* in the copy templates of the library. However, these global constants are not automatically adopted into the controller as well when using the typed block in the project.

Example

Figure 4-3: Constants in an FB

Constant			
MAX_VELOCITY	Real	10.0	Maximum velocity of conveyor
MAX_NO_OF_AXES	UInt	3	Maximum number of axes

Recommendation: using constants for polling values unequal to 0

If a tag shall be assigned in the code with a numerical value unequal to 0, constants need to be used.

This clearly simplifies a change of the numerical value since it is not changed at several locations in the code, but centrally in the constant.

Example

Figure 4-4: Using constants

Blower

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

```
#statVelocity := 0.0; //Correct, cause assignment with
                      //default value of data type 0.0

//Correct --> Working with constants
IF (ABS(#velocity) < #MAX_VELOCITY) THEN
  #statVelocity := #velocity;
ELSIF (#velocity < 0) THEN
  #statVelocity := -1.0 * #MAX_VELOCITY;
ELSE
  #statVelocity := #MAX_VELOCITY;
END_IF;

//Wrong --> Working with numerical values
IF (ABS(#velocity) < 10.0) THEN
  #statVelocity := #velocity;
ELSIF (#velocity < 0) THEN
  #statVelocity := -10.0;
ELSE
  #statVelocity := 10.0;
END_IF;
```

Note Constants are text replacements for numerical values exchanged by the preprocessor. Using constants in the CPU neither causes a performance loss, nor does the memory consumption increase in the data memory.

Only the memory consumption in the load memory of the CPU increases due to the increasing number of characters in the block sources.

4.2.3 Arrays

Recommendation: array name is always plural

The name of an array is always plural.

Example

Array of a structure of axes
axisData → not okay

axesData → okay

Recommendation: array index starts with 0 and ends with a constant

Array limits start with 0 and end with a constant for the upper limit of the array (e.g. DIAG_BUFFER_UPPER_LIM).

An array from 0 on makes sense, since certain system commands, such as MOVE_BLK_VARIANT work zero-based. This enables entering the desired index directly into the system function without the need for recalculation.

Another advantage is that WinCC (Comfort, Advanced and Professional) only work with zero-based arrays, for example for scripts.

However, if still not working with zero-based arrays, an array should be supplied with one constant each for the lower and upper limit.

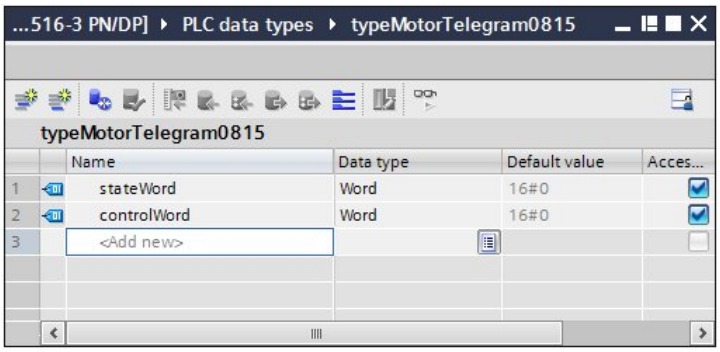
4.2.4 PLC data types

Rule: Prefix ‘type’

The identifier of a user-defined data type is preceded with the prefix ‘type’.

Example

Figure 4-5: Example - PLC data type



Rule: PLC data types instead of structures

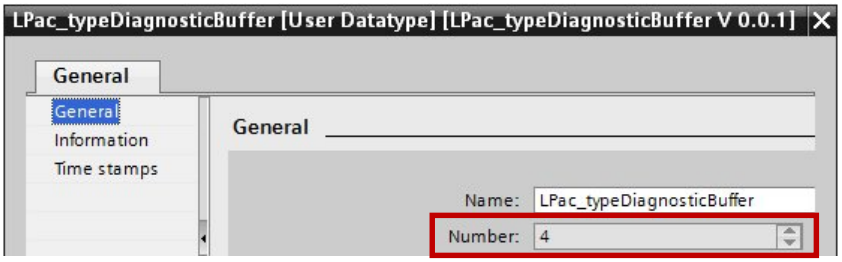
In the PLC program, structures (as common in STEP 7 Classic) must not be used anymore, but only PLC data types.

Note

An exception are know-how protected blocks. In this case, the application of PLC data types should be carefully considered. The reason for this is that in the background, a number is assigned for each PLC data type. If this PLC data type is then used in a know-how protected block, this number must stay the same when copying into another project. If this is not the case, the new project, can only be compiled with the password for know-how protection.

If a block shall be know-how protected, PLC data types are only used where type-safe copying processes or interconnections with the respective structured data type are performed.

Figure 4-6: Numbers for PLC data types



4.2.5 Initialization

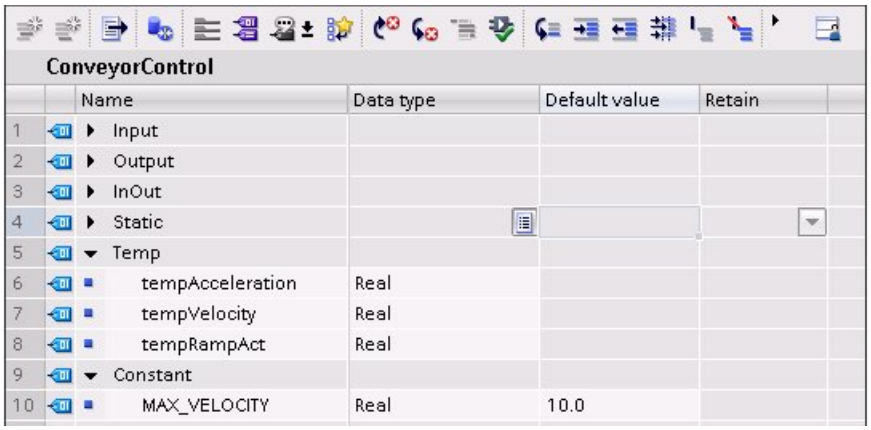
Rule: initializing temp tags in the program

Tags of the L stack (*Temp*) must be initialized at the start of the program by the user.

Generally, it must be ensured, that tags are always written first before they are read.

Example

Figure 4-7: Initialization of temporary data




```
#tempAcceleration := 0.0;
#tempVelocity := #MAX_VELOCITY;
#tempRampAct := 0.0;
```

Rule: initialization is performed in the usual representation

The initialization (assigning constant data) is performed in the usual representation of your data type (literal). A WORD-type tag, for example, is initialized with 16#0001 and not with 16#01.

Example

Figure 4-8: Initialization of static data

Static			
statMask1	Word	16#01	not okay
statMask2	Word	16#0001	okay
statMask3	Byte	2#0000_1010	okay
statMask4	DWord	5	not okay
statCounter1	Int	16#00	not okay
statCounter2	Int	10	okay
statVelocity1	Real	16#0000	not okay
statVelocity2	Real	40.0	okay

Recommendation: parameter initialization of TOs: -1.0

User-defined parameter structures, in which values of a TO are also to be accessed (e.g. speed, acceleration, jerk), are initialized with the value -1.0. This helps differentiate whether a value is transferred for the parameter. If there is no assignment by the user, the default settings of the TO are adopted.

4.3 Instructions

4.3.1 Operators and expressions

Recommendation: Before and after operators, there is a space character

Before and after binary operators and the assignment operator, there is a space character.

Example

```
//Okay
#statSetValue := #statSetValue1 + #statSetValue2;

//Not okay
#statSetValue:=#statSetValue1+#statSetValue2;
```

Recommendation: expressions always in brackets

Expressions must always be put in brackets to make the sequence of the interpretation unique.

Example

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

4.3.2 Program control instructions

Recommendation: line breaks for partial conditions

For more complex expressions, it makes sense to emphasize each “partial condition” by means of a line break. This also allows for clear comments.

Rule: condition and instruction part are separated by a line break

A clear separation of condition and instruction part must be followed.

That is, after a condition (after THEN, for example), there must always be a line break before an instruction is programmed.

Recommendation: correct indentation for IF instructions

Boolean operations are written at the beginning of the line if a line is not sufficient for the entire condition.

For conditions reaching over several lines in IF instructions, these are indented by two space characters. THEN is placed on a separate line at the same level as IF.

If the conditions of an IF instruction fit in one row, THEN can be written at the end of the line.

When cascading, the operand is placed in a separate line alone. A single bracket indicates the end of the cascaded condition. Operands are always placed at the beginning of the line.

Analog, these recommendations also apply to handling other instructions (e.g. WHILE, etc.)

Example

```
IF (DriveStatus() = #OK) //Comment
  AND
  ((#statOldDrive XOR #tempActDrive)
  OR (#statOldPower XOR #tempActPower)
  ) //Comment
  AND (#start = True)
THEN
  ; //Statement
ELSE
  ; //Statement
END_IF;
```

Rule: always create ELSE branches for CASE instructions

A CASE instruction must always have an ELSE branch to be able to report the errors occurring during runtime.

```
CASE #tempSelect OF
  1: //Comment
    ; //Statement
  4: //Comment
    ; //Statement
  2..5: //Comment
    ; //Statement
ELSE
  ; //Generate error message
END_CASE;
```

Recommendation: CASE instruction instead of several ELSIF branches

If possible, a CASE instruction shall be used instead of an IF instruction with several ELSIF branches. This makes the program clearer.

Rule: indenting instructions

Each instruction in the trunk of a control structure is indented.

Example

IF instruction

```
//---------------
IF #tempCondition THEN
  ; //Statement
  IF #tempCondition2 THEN
    ; //Statement
  END_IF;
ELSE
  ; //Statement
END_IF;
```

Example

CASE instruction

```
//-------------------
CASE #statSelect OF
  CMD_INIT: //Comment
    ; //Statement
  CMD_READ: //Comment
    ; //Statement
  CMD_WRITE: //Comment
    ; //Statement
ELSE
  ; //Generate error message
END_CASE;
```

Example

FOR instruction

```
//---------------
FOR #tempIndex := 0 TO #MAX_NUMBER - 1 DO
  ; //Statement
END_FOR;
```

Example

FOR instruction with width of the jump

```
//---------------
FOR #tempIndex := 0 TO #MAX_NUMBER - 1 BY 2 DO
  ; //Statement
END_FOR;
```

Example

Conditional termination of a loop with EXIT

```
//---------------
FOR #tempIndex := 0 TO #MAX_NUMBER - 1 BY 2 DO
  IF #tempCondition THEN
    EXIT; //EXIT Loop
  END_IF;
END_FOR;
```

Example

Recheck loop condition with CONTINUE

```
//---------------
FOR #tempIndex := 0 TO #MAX_NUMBER - 1 BY 2 DO
  IF #tempCondition THEN
    CONTINUE; //loop condition
  END_IF;
END_FOR;
```

Example

WHILE instruction

```
//---------------
WHILE #tempCondition DO
  ; //Statement
END_WHILE;
```

Example

REPEAT instruction

```
//---------------
REPEAT
  ; //Statement
UNTIL #tempCondition END_REPEAT;
```

4.3.3 Error handling

Rule: always evaluate error codes

If FCs, FBs or system functions called in the program provide error codes, these always need to be evaluated.

Further information about error handling can be found in chapter 4.4.3 Error return and diagnostics of function blocks.

4.4 Programming according to PLCopen

The PLCopen organization has defined a standard for Motion Control blocks. This standard is generalized here and can hence be applied to all asynchronous function blocks. It is described here, what the interface of a function block looks like and how the signals of this interface behave.

This standardization enables achieving a simplification of the programming and application of function blocks.

Rule: using standard identifiers for PLCopen

If parameters with standard meaning are required regarding functionality according to *PLCopen Function Blocks for Motion Control V2.0*, the respective standard identifiers must be used.

The following parameters are standard parameters:

Table 4-3: Standard parameters according to PLCopen

Standard function signals PLCopen-conform	Meaning
***Input* parameters**	
execute	**execute without 'continuousUpdate':** All parameters are started with a rising edge at the *execute* input and the functionality is started. If changes of the parameters become necessary, the *execute* input must be retriggered.
or	**execute with 'continuousUpdate':** All parameters are adopted with a rising edge at the *execute* input. These can be adjusted as long as the *continuousUpdate* input is set.
enable	**enable:** All parameters are adopted with a rising edge at the *enable* input and can be continuously changed. The function is activated level-controlled (for TRUE) and deactivated (for FALSE).
***Output* parameter**	
Exclusiveness: done busy valid commandAborted error	**With *execute*:** The outputs *done*, *busy*, *commandAborted* and *error* are mutually exclusive, that is, only one of the outputs can be set at any given time. If *execute* is set, one of these outputs must be set. **With *enable*:** The outputs *valid* and *error* are mutually exclusive.
done	Output *done* is set, if the command was processed successfully.
busy	**With *execute*:** The FB has not yet completed processing the command and therefore, new output values can be expected. *busy* is set and reset at a rising edge of *execute*, if one of the outputs *done*, *commandAborted* or *error* is set. **With *enable*:** The FB is currently busy processing a command. New output values can be expected. *busy* is set with a rising edge of *enable* and remains set for

Standard function signals PLCopen-conform	Meaning
	as long as the FB executes actions.
active	**Optional output** to produce compatibility with PLCopen (*buffered mode* of function blocks). The output is set, as soon as the FB takes control over the axis. If no *buffered mode* has been selected, *active* and *busy* can be identical.
commandAborted	**Optional output**, which indicates that the running job of the function block was cancelled by another function or another job for the same object. Example: An axis is positioned straight via the function block, while the same axis is stopped by another function block. At the positioning function block, the *commandAborted* output is then set, since this job was cancelled by the stop command.
valid	The output is only used in conjunction with *enable*. The output is set, as long as valid output values are available and the *enable* input has been set. As long as an error is pending, the *valid* output is reset.
error	Rising edge of the output signals that an error has occurred while processing the FB.
status (state errorID)	Error information or status of the block In contrast to the PLCopen standard, the identifier *errorID* is not used for reasons of compatibility with existing SIMATIC system functions and blocks; instead, *status* is used.
diagnostics	**Optional output**: Detailed error buffer Any errors, warnings and information of the block is stored in a ring buffer. The size (number of array elements) is oriented along the available memory of the PLCs supported by the application. The diagnostic structure is described in Chapter 4.4.3 Error return and diagnostics of function blocks.

4.4.1 Blocks with execute

The job is started with a rising edge at the *execute* parameter and the values pending at the input parameters are adopted.

Subsequently changed parameter values are only adopted at the next job start, if no *continousUpdate* is used.

Resetting the e*xecute* parameter does not terminate processing the job, however, it affects the display duration of the job status. If e*xecute* is reset before a job has been completed, the parameters *done*, e*rror* and *commandAborted* are respectively only set for one call cycle.

After the job has been completed, a new rising edge is required at *execute* to start a new job.

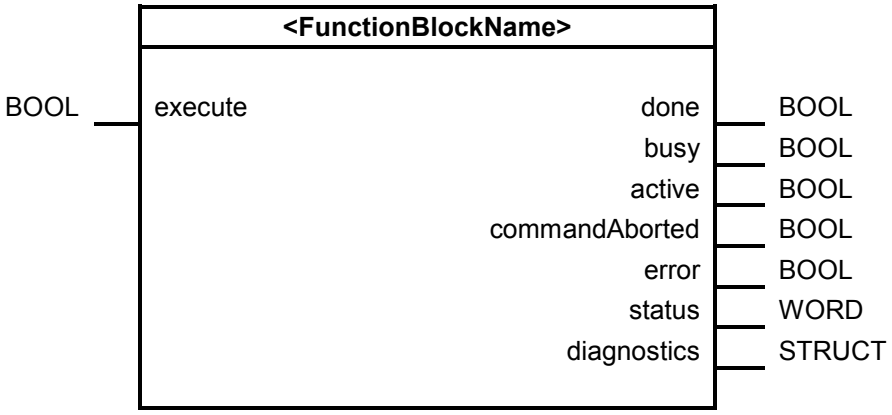
This ensures that for each start of a job, the block is in the initial state and the function is processed independently of the previous jobs.

Rule: Parameter: execute requires busy and done

If the programmer uses input parameter *execute*, the output parameters *busy* and *done* must be used.

Example

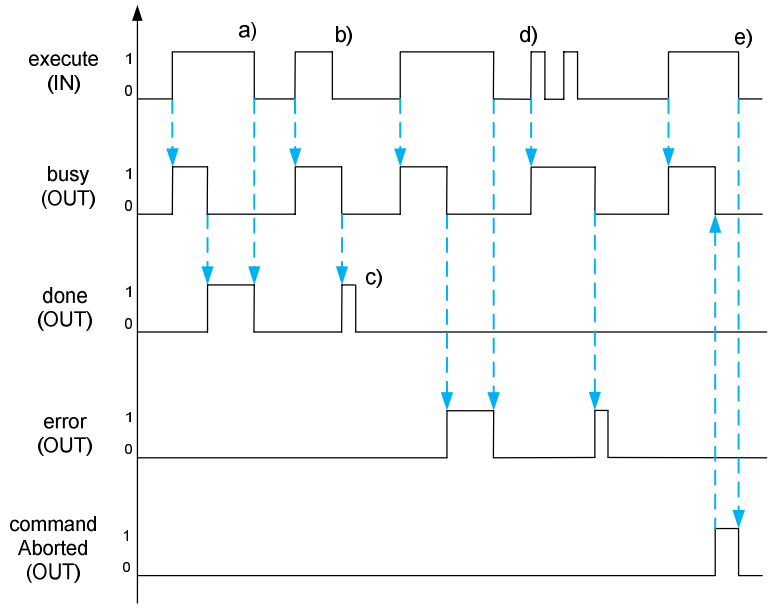
Figure 4-9: LAD representation



Signal flow diagram - Block with execute

NOTICE	If input *execute* is reset before output *done* is set, output *done* must only be set for one cycle.

Figure 4-10: Signal flow diagram of a function block with execute input



a) *done*, *error* and *commandAborted* are reset with falling edge at *execute*.

b) The functionality of the FB is not stopped with falling edge at *execute*.

c) If *execute* is already FALSE, then *done*, *error* and *commandAborted* are only pending for one cycle.

d) A new job is requested with a rising edge at *execute* while the block is still processed (*busy* = TRUE). The old job is either terminated with the parameters pending at the start of the job, or the old job is cancelled and restarted with the new parameters. The behavior follows the application case and must be documented accordingly.

e) If processing a job is interrupted by a higher or equal priority job (of another block/instance), *commandAborted* is set by the block. It immediately interrupts the remaining job processing. This job occurs if an emergency stop at an axis shall be executed, while another block executes a move job at the axis.

4.4.2 Block with enable

Setting the *enable* parameter starts the job. As long as *enable* remains set, job processing is active and new values can be adopted processed continuously.

Resetting the *enable* parameter terminates the job.

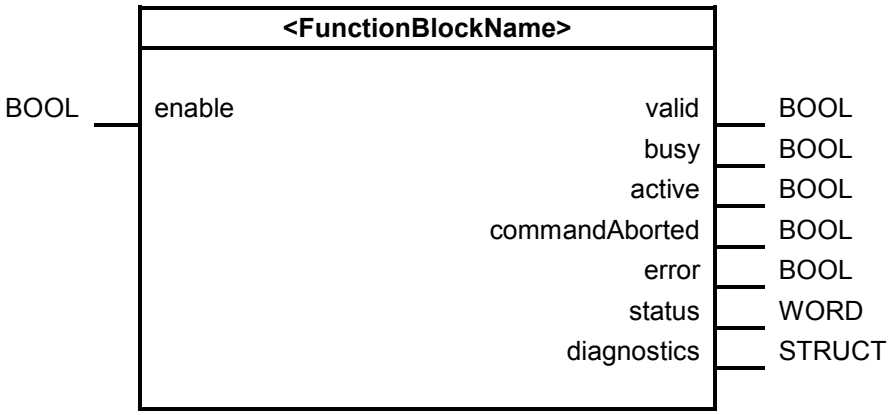
If a new job is started, the block is put to its initial state and can be switched and configured completely new.

Rule: Parameter: enable requires valid

If the programmer uses the *enable* input, at least the output parameter *valid* must be used.

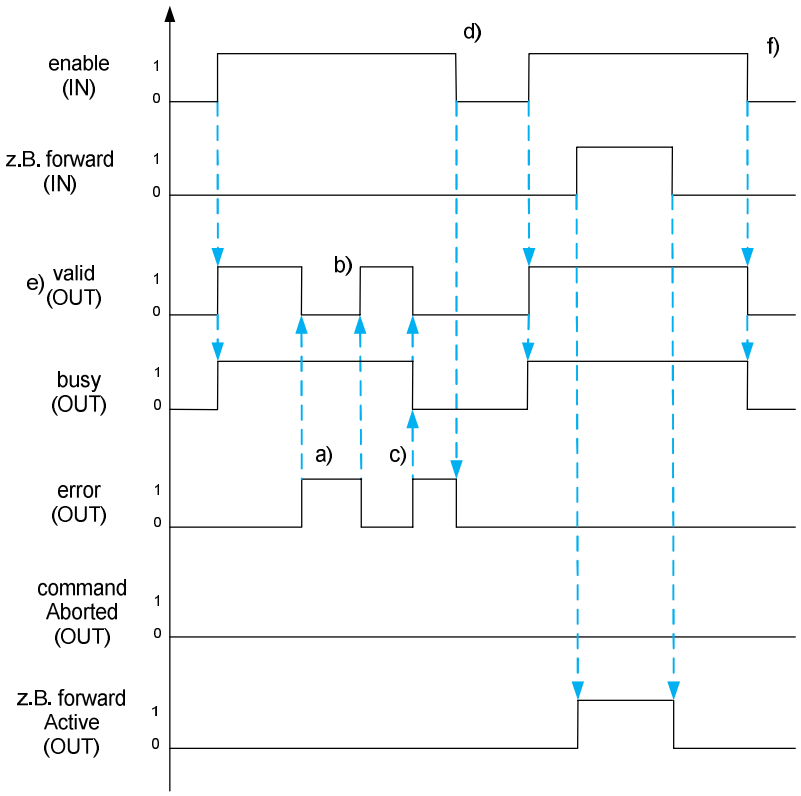
Example

Figure 4-11: LAD representation



Signal flow diagram - Block with enable

Figure 4-12: Signal flow diagram of a function block with enable input



a) *error* on TRUE is used to reset *valid* and stop all functionalities of the FB. Since it is an error that can be removed by the block itself, *busy* remains set.

b) After removing the error cause (e.g. restoring communication) *valid* is set again.

c) An error occurs that can only be removed by the user. Here, *error* must be set, *busy* and *valid* be reset.

d) The pending error that has to be removed by the user can only be acknowledged by a falling edge at *enable*.

e) *valid* on TRUE means, that the block is activated, no errors are pending, and hence, the outputs of the FB are valid.

f) If *enable* is reset to FALSE, *valid* and *busy* are also reset.

CommandAborted, error and done are always set for as long as the execute signal is pending, at least, however, for one cycle.

4.4.3 Error return and diagnostics of function blocks

Note This chapter is no longer part of the *PLCopen Function Blocks for Motion Control V2.0* standard.

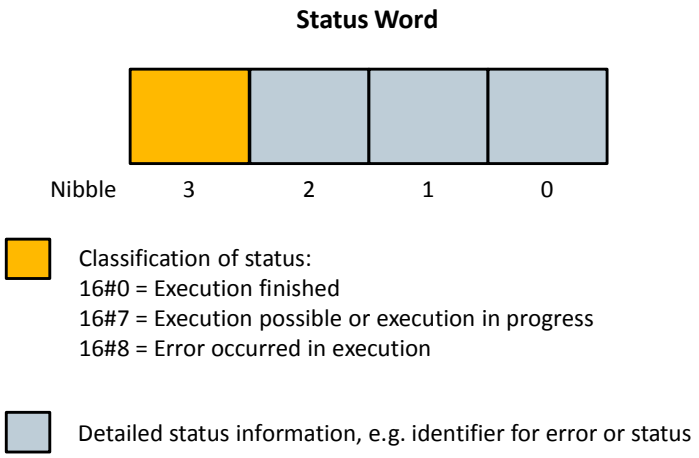
The following additional rules and recommendations describe further requirements for consistent programming of error return and diagnostics in function blocks.

Rule: Formal parameter status: general error return

An error is indicated by setting the Boolean tag *error*. At the same time, an error is displayed by setting the highest-order bit in the *status* output. The remaining bits are used for an error code that uniquely points to the cause. For compatibility reasons with the previous SIMATIC system blocks, the *status* output is used instead of the *errorID* output that is required according to the PLCopen standard.

As an alternative, the connection to an error concept (e.g. message handling) can be realized. The tags must then be realized according to the concept. For example, error numbers with several associated values, diagnostic structure, ...

Figure 4-13: Structure of the *status* output



Recommodation: Parameter status: standardized error numbers

For a standardization of the errors, the number bands shown in the following table must be followed.

Table 4-4: Number bands for errors

Error cause	Number band *status*
Job completed, no warning or further details	16#0000
Job completed, further details	16#0001 ... 16#0FFF
No job currently processed (**also initial value**)	16#7000
First call after incoming new job (rising edge	16#7001

Error cause	Number band *status*
execute)	
Subsequent call during active processing without further details	16#7002
Subsequent call during active processing with further details. Occurred warnings that do not affect the operation.	16#7003 .. 16#7FFF
Wrong operation of the function block	16#8001 .. 16#81FF
Error during parameterization	16#8200 .. 16#83FF
Error when processing from outside (e.g. wrong I/O signals, axis not referenced)	16#8400 .. 16#85FF
Error when processing internally (e.g. when calling a system function)	16#8600 ..16#87FF
Reserved	16#8800..16#8FFF
User-defined error classes	16#9000…16#FFFF

Recommodation: Error keeps pending until acknowledgement

If an error is detected when processing a function block, the current job and hence the motion, for example, is stopped. The error code for the first error remains pending until it is acknowledged (negative edge of *execute;* falling edge also necessary for *enable,* depending on error type).

Recommodation: Use of output *status* for state and error codes of instruction

State codes of instructions (system blocks) are looped through unchanged to output *status*. Therefore the documentation of the blocks and its error and state codes can refer to the TIA Portal help.

Recommodation: Use of output *statusID* for identification of error source

To identify error sources precisely it's recommended to use the additional output *statusID* with the following properties:

- Its value shows which block or sub block (sub instance) reports an error. It's recommended to assign the *statusID* "1" to the calling block, and "2" or higher to all other sub block numbers.
- If no error/warning is reported, then the value "0" is returned.
- The output data type is "UINT".

All instances are assigned a unique *statusID* within the calling block

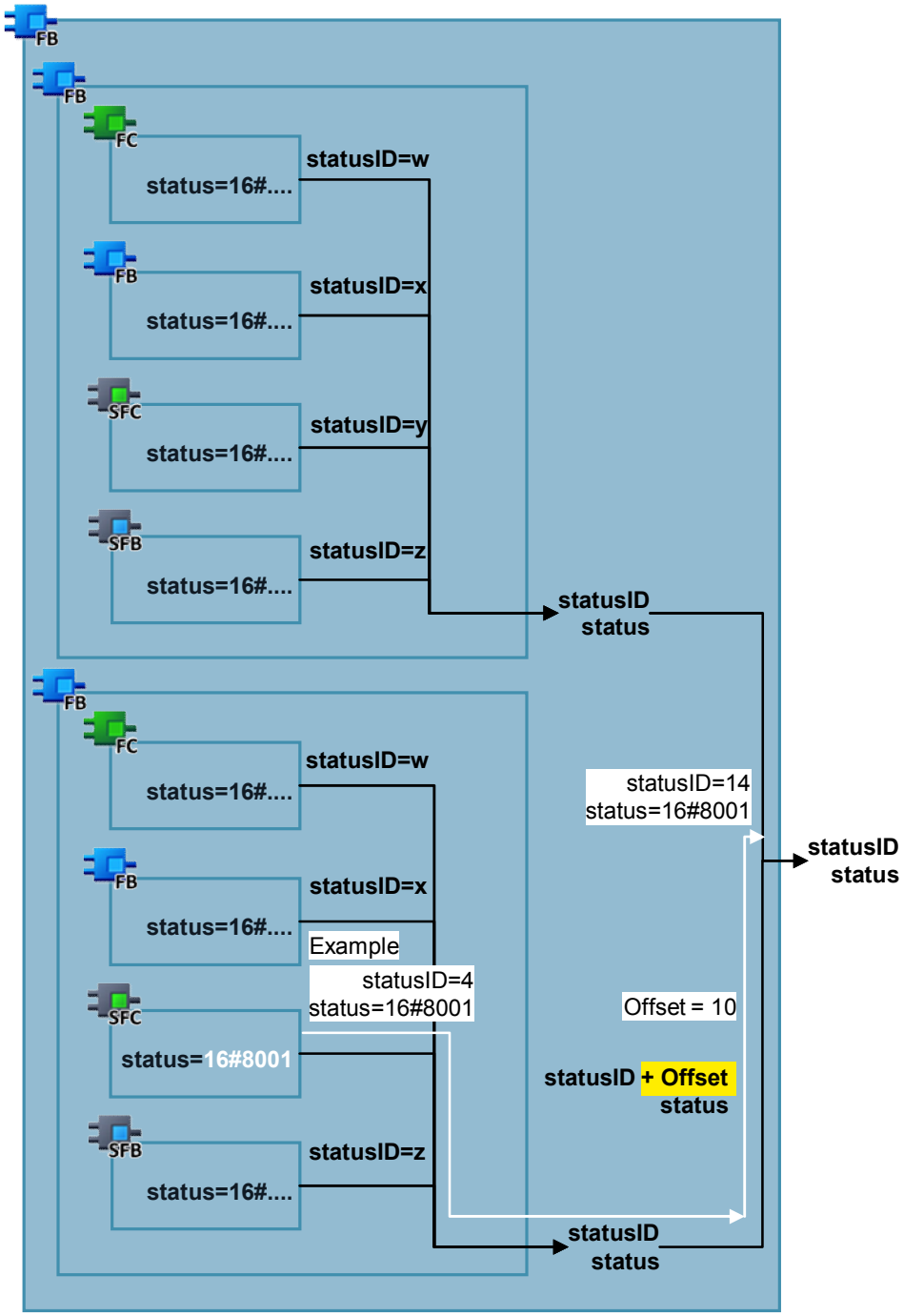
Recommodation: Output *statusID* and offset with nested blocks

When using nested blocks it's recommended to assign a unique value of the error source (identifying the called instance) to the output *statusID* of the calling block.

To do so, add a defined offset to the value of *statusID* of the called blocks, in the case of multiple instances of nested blocks. That way a program-wide unique *statusID* value is assigned to each instance of a block.

Example

Figure 4-14: *status* and *statusID* by nested blocks



Recommendation: Parameter diagnostics: diagnostic structure

In a diagnostic structure, all further information on the occurred error must be stored at the *diagnostics* output. Furthermore, values for diagnosing the current block behavior, such as runtime information, can also be stored there.

Figure 4-15: Setting up the diagnostic structure

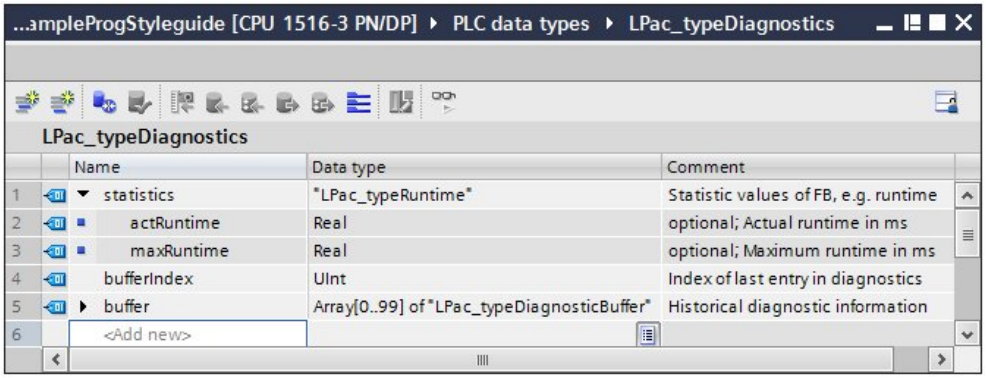


Table 4-5: Elements of a *diagnosticBuffer* structure

Name	Data type	Optional	Comment
timestamp	DATE_AND_TIME		Time stamp of when the error occurred
stateNumber	DINT		State of the internal state machine when the error occurs
modeNumber	DINT	x	Mode of the internal mode-state machine when the error occurs
subfunctionStatus	DWORD	x	Return value for errors of called FBs, FCs and system blocks
status	WORD		Status that uniquely identifies the error
additionalValueX	Any	x	Additional value (X = number), to be able to save error-specific diagnostic information (e.g. axis position).

In parameter *timestamp*, the time at which the error has occurred is stored.

In *stateNumber*, the current state of the state machine is stored. For a function block with different operating modes, the operating mode in which the error has occurred is stored in the *modeNumber* tag.

If an error of a system function or a called FB / FC was detected, the return code is stored in the *subfunctionStatus* element.

The unique error code of the *status* output is additionally stored in the *status* element of the diagnostic structure.

Additional parameters for an error are stored in the a*dditionalValueX* tags. The neutral designation of the *additionalValueX* values should be kept to enable saving different values on a memory area.

If further elements become necessary, they can be added.

Recommendation: retentive diagnostic structure

The diagnostic structure should be created retentive to enable diagnostics after a power failure at the PLC.

4.5 Tables, traces, measurements

Rule: PascalCase notation for tables and traces

PascalCase notation (first letter in upper case) is used for:

- PLC tag tables
- Watch tables
- Traces
- Measurements

4.6 Libraries

In this chapter, rules and recommendations for programming libraries are specified. The rules for source code and tag names introduced in the previous chapters are binding for creating libraries.

Recommendation: documentation for libraries

In general, it is recommended to describe each library sufficiently in a documentation:

- Blocks and their functions
- Version system
- Change history
- etc.

4.6.1 Name assignment

Recommendation: Library name: prefix L and length max. 8 characters

The name of a library receives the prefix L (e.g. LPac). L stands for the English word library. Furthermore, no underscores are used, except between the prefix of the library and the block/constant name. The maximal character length for a library name, and hence for the prefix, is limited to 8 characters.

This restriction is used for compact name assignment.

Rule: all elements contain the name of the library as a prefix

All elements existing in a library (PLC and HMI elements) received the name of the library. This ensures that there are no collisions in the block names.

NOTICE	If a block is inserted into a library, all properties of the block, such as block number and know-how protection, must have been set already. Once the block is in a library, its properties cannot be changed anymore.

Example

Table 4-6: Example for name assignment for library LExample

Type	name according to styleguide
Library	LExample
PLC data type	LExample_type<Name>
Function block	LExample_<Name>
Function	LExample_<Name>
Organization block	LExample_<Name>
PLC tags	LExample_<Name>
general constant	LEXAMPLE_<NAME>
Constant for error code	LEXAMPLE_ERR_<NAME>

Example

Identifier: LCom_CommToClient
Library: LCom
Functionality: communication via TCP/IP between different devices

4.6.2 Setup

Rule: Types: FC, FB, PLC data types

Functions, function blocks, and PLC data types are added into a library as types. Everything else is added into a library as a copy template, especially organization blocks and tag tables

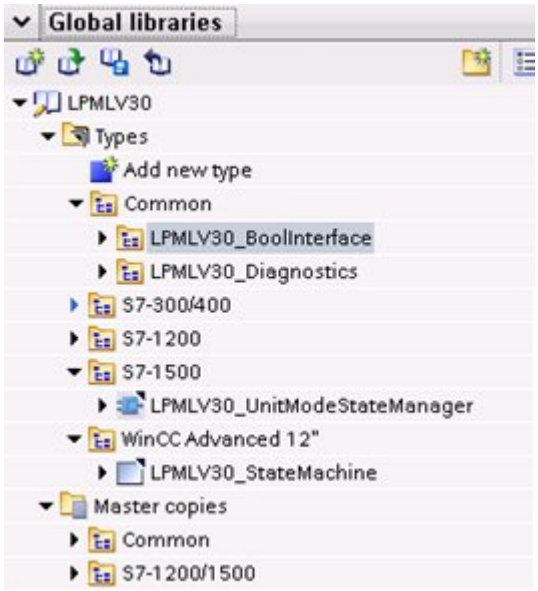
NOTICE	**Know-how protection binds the block to the controller type and firmware of the last compilation.** That is, if the block has been compiled on an S7-1500 in the development phase, it cannot be used on an S7-1200 despite S7-1200 compatible programming. It must also be noted here, that the block is also bound to the firmware as well as the controller type. A know-how protected block cannot be compiled again without password. If PLC data types are used, the user must ensure not to change these. PLC data types cannot be know-how protected.

Recommendation: grouping in the library

The PLC blocks and HMI screens of a library are assigned to a group named after the controller or HMI type. If a block (e.g. PLC data type) or HMI screen applies for all types of controllers (S7-300, S7-400, S7-1200 and S7-1500) or HMIs (…), the Common folder must be used.

Example

Figure 4-16: Structure of a library



4.6.3 Version system

Rule: definition of the version system

The official version system (first released version) starts with version V1.0.0 (see Table 4-7). Versions smaller than 1.0 refer to development versions.

The third digit in the software version system marks the changes that do not affect the documentation, such as mere error removal with no new functions.

When expanding the existing functionality, the second digit is counted up.

For a new main version, incompatible, with new functionalities, the first digit is incremented.

Rule: continuous version system

The version system of the library is continuous. In the case of changes, the version number of the library is always counted up. Additionally, the versions of the blocks are assigned according to the above version procedure. It is possible here, that none of the blocks carries the library version, since the versions of blocks and library are assigned independently of one another (see example below).

Example

Table 4-7: Example for changing the version

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		Error handling of FB1
1.0.2	1.0.1	1.0.1	1.0.0		Optimization of FB2
1.1.0	1.1.0	1.0.1	1.0.0		Expansion at FB1
1.2.0	1.2.0	1.0.1	1.0.0		Expansion at FB1
2.0.0	2.0.0	1.0.1	2.0.0		New functionality at FB1 and FC1
2.0.1	2.0.0	1.0.2	2.0.0		Error handling of FB2
3.0.0	2.0.0	1.0.2	2.0.0	1.0.0	New function FC2

Rule: updating changes and version in the block header

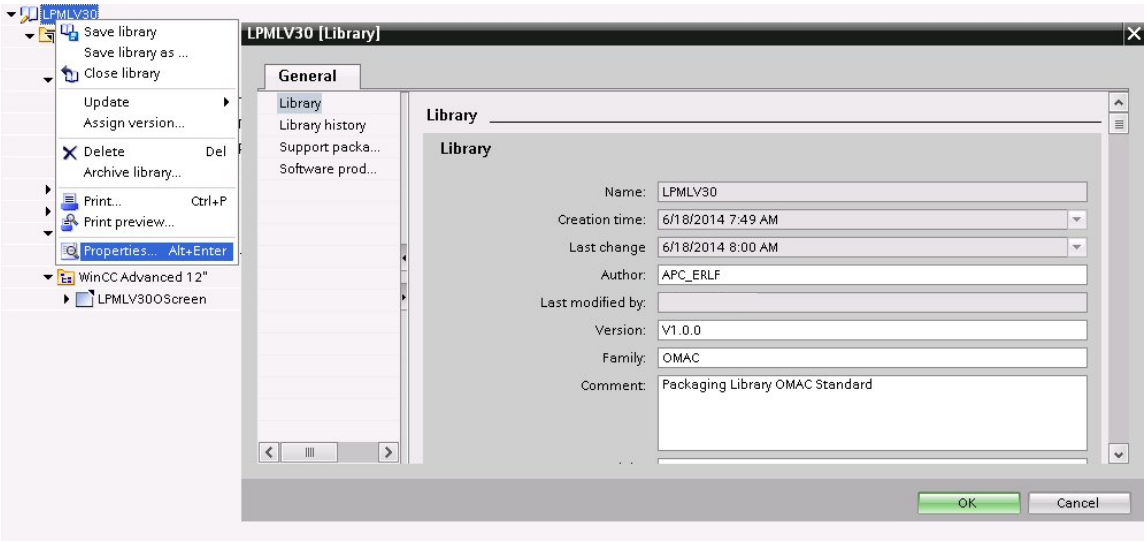
For every change of the version, the adjustments at the respective locations, for example, in the block header of the function, are described.

Rule: updating the properties dialog: version, department short

The current version of the library is entered in the properties dialog of the library. For standard libraries, a unique short for the respective department is stored in the properties window.

Example

Figure 4-17: Properties of a library



4.6.4 Performance test

Recommendation: performance test

Before a library is delivered, the function blocks shall be tested on a CPU in the average performance range with small-medium quantity framework (e.g. CPU 1212 or CPU 1511-1 PN), in order to detect problems in performance and storage early on.

4.6.5 Delivery

Recommendation: delivery as a zip-file

A library should not be delivered as folder structure, but as archive (file format .zip or TIA Portal archive).

4.6.6 Example project

Recommendation: Example project for complex library elements

If a block is more complex, or if the application has HMI pages as well as PLC code, an example project should also be created in addition to a library.

The example project needs to be runnable on the standard controller for this application without adjustments. The HMI should also be usable in the example project immediately without adjustments.

Recommendation: HMI screen only in example projects if possible

If HMI pages are not necessarily part of a library or an application, they should only occur in the example project. This elegantly by-passes the problem of having to offer HMI pages for the different panel types (WinCC Runtime, Comfort and Basic Panels) and panel sizes.

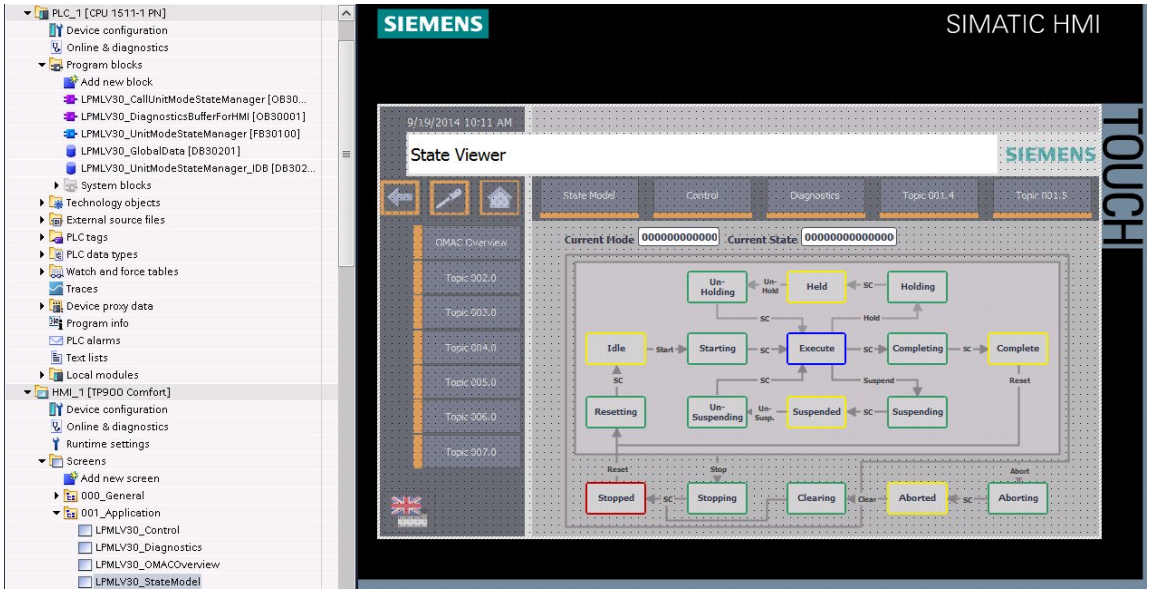
Recommendation: using HMI OS templates

The HMI OS template \6\ should be used for example projects.

https://support.industry.siemens.com/cs/ww/en/view/96003274

Example

Figure 4-18



Recommendation: using standard devices, if possible

Unless there are other specifications regarding the hardware, the following devices should be used:

Table 4-8

Type	Device
S7-1500	CPU 1513-1PN
S7-1200	CPU 1214
S7-300	CPU 315-2 PN/DP
Comfort Panel	TP900

5 Links & Literature

Table 5-1

	Topic	Title
\1\	Siemens Industry Online Support	https://support.industry.siemens.com
\2\	Download page of the entry	https://support.industry.siemens.com/cs/ww/en/view/81318674
\3\	Totally Integrated Automation	http://www.industry.siemens.com/topics/global/en/tia/Pages/default.aspx
\4\	Basic functions for modular machines	https://support.industry.siemens.com/cs/ww/en/view/61056223
\5\	Programming Guideline for S7-1200/1500	https://support.industry.siemens.com/cs/document/81318674
\6\	Know-how in the Online Support	https://support.industry.siemens.com/cs/ww/en/view/96003274
\7	Norm	Further information you can find in the norm IEC 61131-8.

6 History

Table 6-1

Version	Date	Modifications
V1.0	10/2014	First version after internal release
V1.1	06/2015	Adjustments and corrections