Application Style Guide

applications & TOOLS

Programming Guidelines: SIMOTION Applications

SIEMENS

Application Number: A4027118-A0054

We reserve the right to make technical changes to this product.

Copyright

General information

Note

The Style Guide is not binding and does not claim to be complete regarding the configuration as well as possible eventualities. The Style Guide does not represent customer-specific solutions. It is only intended to provide support for typical applications. You are responsible in ensuring that the described products are correctly used. This Style Guide does not relieve you of the responsibility of safely and professionally using, installing, operating and servicing equipment. When using the Style Guide, you recognize that Siemens cannot be made liable for any damage/claims beyond the liability clause describe. We reserve the right to make changes to Style Guide at any time without prior notice. If there are any deviations between the recommendations provided in this Style Guide and other Siemens publications - e.g. Catalogs, then the contents of the other documents have priority.

Warranty, liability and support

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

Claims against us - irrespective of the legal grounds - resulting from the use of the examples, information, programs, engineering and performance data etc., described in this Style Guide are excluded. Such an exclusion shall not apply where liability is mandatory e.g. under the German Product Liability Act involving intent, gross negligence, or injury of life, body or health, guarantee for the quality of a product, fraudulent concealment of a deficiency or non-performance. Claims of the purchaser for compensation relating to non-performance of essential contract obligations shall be limited to foreseeable damages typically covered by a contract unless intent, willful misconduct or gross negligence is involved or injury of life, body or health. The above stipulations shall not change the burden of proof to your detriment.

If you have any questions relating to this document then please send them to us at the following e-mail address:

mailto:applications.erlf.aud@siemens.com

Qualified personnel

In the sense of this documentation qualified personnel are those who are knowledgeable and qualified to mount/install, commission, operate and service/maintain the products which are being used. He or she must have the appropriate qualifications to carry-out these activities
e.g.:

- Trained and authorized to energize and de-energize, ground and tag circuits and equipment according to applicable safety standards.
- Trained or instructed according to the latest safety standards in the care and use of the appropriate safety equipment.
- Trained in rendering first aid.

There is no explicit warning information in this documentation. However, reference is made to warning information and instructions in the Operating Manual for the particular product.

Information regarding export codes

AL: N

ECCN: N

Table of Contents

1 Preliminary comment 7
1.1 Terminology 7
1.2 Scope of the document 8

2 Programming guidelines 9
2.1 Using programming guidelines for customer applications 10
2.2 ST source 10
2.2.1 Source 10
2.2.2 Formatting 10
2.2.3 Comments 10
2.2.4 Unit names 11
2.3 Name syntax for programs, variables and data types 11
2.3.1 General name syntax 11
2.3.2 Prefixes for derived/user-defined data types 15
2.3.3 Prefixes for functions, function blocks and FB instances 17
2.4 Variable definitions 18
2.4.1 Constants / enumeration data types 18
2.4.2 Variables 18
2.4.3 Initialization 19
2.4.4 IO variable 19
2.5 Programs 20
2.5.1 Operators 20
2.5.2 Expressions 20
2.5.3 Program control instructions 21
2.5.4 Error handling 24
2.6 Functions and function blocks 24
2.6.1 FC/FB parameters 25
2.6.2 Signal timing diagram of standardized parameters 29
2.7 Libraries 31
2.7.1 Assigning names 31
2.7.2 Structure 32
2.7.3 Programming 33
2.7.4 Structure of a library unit 34
2.7.5 Programming function blocks for libraries 34
2.7.6 Error return and diagnostics of function blocks 44
2.7.7 Versions 47
2.7.8 Performance test 49
2.7.9 Delivery 49
2.7.10 ST code template for an aggregate function block 50

3 Template 51

4 Documentation 52

Appendix 53

5 Revisions/author 53

6 Literature 53

7 Contact partners 54

1 Preliminary comment

This document applies to (customer) applications as well as libraries that have been written in the programming languages of IEC 1131-3 (DIN EN 61131-3) Structured Text (ST), Ladder Diagram (LAD) and Function Block Diagram (FBD) – as well as the SIMOTION programming languages Motion Control Chart (MCC) and Drive Control Chart (DCC).

The rules and recommendations described here are binding and are mandatory when generating standard applications and libraries.

The rules and recommendations for assigning names are valid for all programming languages in SIMOTION. Rules and recommendations for the source code structure, programming function blocks and libraries are intended for programming engineers using ST.

All personnel in the various APCs (Application Centers) should use this document.

It should always be observed that the names referring to the functionality and data type are always clear and unique; i.e. if the same name is used then the functionality it refers to should also be the same.

When transferring parameters, the various possibilities should be taken into consideration – with their associated advantages and disadvantages.

- Access operations to individual structural elements of FB or FC outputs provide better testability and improved transparency – however, they reduce the performance.
- Transferring complete structures improves the performance but has a negative impact on the testability.

The procedure which is the best should be decided on a case for case basis.

1.1 Terminology

Recommendations/rules

Specifications are sub-divided into recommendations and rules. Recommendations are intended to keep the code standard (uniform) and are also intended to provide support and information. Recommendations should in principle always be followed; however, there are certainly situations where a recommendation is not followed – whether it be relating to efficiency, or because the code would be able to be more easily read. Contrary to recommendations, rules should always be followed (they are binding).

Customer application

A customer application is a specific application for a certain user and is not provided by SIEMENS AG as product.

1.2 Scope of the document

In addition to applications, users can also access SIMOTION system functions that are used within the framework of applications.

Note The rules and recommendations described in this document do not apply to these system functions.

2 Programming guidelines

Programming guidelines are used to obtain a standard/uniform code that can be more easily serviced and re-used. Not only this, errors (bugs) can be identified at an early stage (e.g. by the compiler) and avoided.

The source code should have the following properties:

- A standard and unified style
- Should be able to be easily read and understood

A certain appearance should initially be maintained regarding serviceability and transparency of the source code. Optical effects – e.g. a standard number of blanks before the comma – only play an insignificant role regarding the software quality. It is far more important for example, to find and select rules that support development engineers in the following way:

- Avoid typing errors and careless mistakes that the compiler then incorrectly interprets.

  Objective: The compiler identifies as many errors as possible.

- Support the code for identifying and resolving program errors and bugs; for instance, by using prefixes to more simply identify type incompatibilities

  Objective: The code indicates problems at an early stage.

- Standardize standard applications and libraries

  Objective: The program code is more easily learned and the reusability of program code is increased

- Modularization

  Objective: Increasing the level of transparency

  Selectively use of sub-functions and simple combination of different modules by encapsulating and clearly separating sub-functions

  Define clear and unique interfaces

- Increase the serviceability and ongoing development

  Objective: Changes to the program code in the individual modules, that can involve functions/function blocks/programs or units in libraries or in the projects – should have a minimum impact on the total application/total library.
  Different programming engineers should be able to make changes to program code in the individual modules.

Rule

Each time a rule is violated it must be documented!

2.1 Using programming guidelines for customer applications

Rule

Customer's requirements have priority for customer applications. If the customer requests changes or deviations from these programming guidelines, then this has first priority. This must also be documented in writing. Rules defined by customers must be documented in the source text in a suitable form.

2.2 ST source

2.2.1 Source

Rule

Every ST source must be documented. The template from /4/ should be used.

Recommendation

No language-specific special characters should be used, e.g. ä, ö, ü, à, etc.

2.2.2 Formatting

Rule

Tab characters are not permissible in the source text. Indentations should be made using four blanks. When using the internal ST Editor, this is done automatically.

Recommendation

For an improved readability, the line length of the source text in printed form should be limited to 80 characters.

Recommendation

Various ST code sections with associated functionality should be optically separated using a line break.

2.2.3 Comments

A distinction should be made between two types of comments:

- Strategic comments (these describe what a function or a code section does)
- Tactical comments (these describe the code of an individual line)

Recommendation

A strategic comment should be located at the beginning of the corresponding code section.
A tactical comment, if possible, should be located at the end of the code line – otherwise in front of the associated code line.

Rule

Comments in ST start with //. The comment is started after the slash symbols without any blanks.
Reason: For test purposes, complete blocks can be simply removed using (*..*).

2.2.4 Unit names

Rule

Each unit contains a prefix. This prefix is assigned after the POUs (Program Organization Units) or declared data used in the unit.

Table 2-1: Prefixes for unit names

Prefix	Significance	Example:
a	Version unit in libraries, name for the version unit is always aVersion (use of the letter a so that this Unit is located at the uppermost position in the project navigator	aVersion
c	Unit for global constants	cConst
d	Unit that is only used to declare global type definitions and variables	dGlobal
f	Unit that is only used for functions and function blocks	fWinder
p	Unit that is only used for programs	pCrossX
x	Mixed unit with data (type definitions, global variables, constants), functions/ FBs and/or programs	xMisc

Rule

The use of the prefix x and therefore the use of data, functions/ FBs and programs – in a unit is permitted to ensure extremely compact Units with a low code scope and/or a low number of POUs.

2.3 Name syntax for programs, variables and data types

2.3.1 General name syntax

The syntax applies to:

- Identifiers of variables and constants

- Identifiers of derived data types (array, struct, enum) and their associated elements
- Identifiers of programs, functions and function blocks
- Identifiers of parameters of functions and function blocks

It is important to distinguish between names and identifiers. The name is part of an identifier that describes the particular meaning.

The identifier comprises the following

- Prefixes (list as shown in Table 2-2 and Table 2-3
- The name

Rule

The leading underscore character in the identifiers of functions and function blocks is reserved for SIEMENS system functions.
Leading underscore characters should not be used in the application itself.

Rule

Prefixes are specified and must be compliant with the specifications as listed in Table 2-2 and Table 2-3. The reason for this is to ensure that the code has a standard appearance.

Rule

Identifiers where the only difference is either upper or lower case may not be used. Once the notation of an identifier has been selected, then it is kept in all of the sources.

Recommendation

The name in the identifiers should be in English. The name indicates the meaning and purpose of the particular identifier in the context of the source code.

Rule

Prefixes are not used for input and output variables of FCs/ FBs. If structures are used for input and output variables, then the individual elements have prefixes.

Rule

Constants do not contain a prefix.

Rule

The name in the identifiers starts with a lower case letter – unless prefixes are being used.

Rule

Names, that comprise several words, are written together and each word that has a word in front of it or a prefix, starts with an upper case letter; the rest of the word is written in lower case letters.
Exception: Constants and enums (enumerations)

Example:

Local variable: rMaxLength

Rule

Terms (e.g. names of variables and functions) that are defined in the system are not permissible.

Recommendation

The maximum length of a name is 32 characters.

Rule

The identifier of functions, function blocks and programs should be structured according to the following schematic [Operation] Object [Attribute].

Example:

Identifier: FCSwapWordBigEndian
Name: SwapWordBigEndian
Operation: Swap
Object: Word
Attribute: BigEndian

Rule

Separators (underscore characters) in the prefix (memory location/data type) and between the prefix and identifier are not permitted.

Rule

For prefixes, the sequence is the memory location followed by the data type.

Table 2-2: Save location

Prefix	Save location/Visibility
without	**Local variable**
g	Global variable defined as global device variable in the INTERFACE part of a UNIT (can be exported) in the IMPLEMENTATION part of a UNIT
i	In the I/O symbol browser (peripheral access, inputs) or declared using an absolute identifier (VAR AT)
q	In the I/O symbol browser (peripheral access, outputs) or

Prefix	Save location/Visibility
	declared using an absolute identifier (VAR AT)
r	retentive data

The prefixes in [ ] brackets are an alternative approach. However, within any one project, the selected procedure must be kept uniform.

Table 2-3: Data type

Prefix	[Alternative]	Data types (identifiers)		Value range
bo		BOOL	Bit (1)	TRUE, FALSE
b8	[**b**]	BYTE	Byte (8)	16#00...16#FF
b16	[**b**]	WORD	Word (16)	16#0000...16#FFFF
b32	[**b**]	DWORD	Double word (32)	16#0000 0000...16#FFFF FFFF
i8	[**i**]	SINT	Short integer number (8)	-128...127
u8	[**u**]	USINT	Unsigned short integer number (8)	0...255
i16	[**i**]	INT	Integer number (16)	-2**15...2**15 -1
u16	[**u**]	UINT	Unsigned integer number (16)	0...2**16 -1
i32	[**i**]	DINT	Double integer number (32)	-2**31...2**31 -1
u32	[**u**]	UDINT	Unsigned double integer number (32)	0...2**32 -1
r32	[**r**]	REAL	Floating-point number (32)	Refer to IEC 559
r64	[**r**]	LREAL	Long floating point number (64)	Refer to IEC 559
a		ARRAY	Array	
e		ENUM	Enumeration	
s		STRUCT	Structure	
sg		STRING	String	
to			TO reference	

Example:

gasFeeder Global array of a feeder structure

Table 2-4: Pre-defined data types

Prefix	Pre-defined data types (identifiers)		Value range
t	TIME	Duration	Refer to SIMOTION
d	DATE	Date	Refer to SIMOTION

Prefix	Pre-defined data types (identifiers)		Value range
tod	TIME_OF_DAY	Time	Refer to SIMOTION
dt	DATE_AND_TIME	Date and time	Refer to SIMOTION

2.3.2 Prefixes for derived/user-defined data types

Recommendation

The prefix e, s, a is set in front of the identifier of a user-defined data type and the 'type' suffix attached.

Table 2-5: User-defined data types

User def. data types (identifier)		
a< Name > Type	TYPE	Array
e< Name > Type	TYPE	Enumeration type
s< Name > Type	TYPE	Structure

Example:

```
TYPE
    //array
    ar32PositionType : ARRAY[0..MAX_NUM_ELEM - 1] OF REAL;
    //enum
    eColorType       : (RED, GREEN, LIGHT_BLUE);
    //structure
    sMixtureType     : STRUCT
                           i16Elem1 : INT := 1;
                           i16Elem2 : INT := 2;
                       END_STRUCT;
END_TYPE
VAR_GLOBAL
    gar32Position : ar32PositionType;
    geColor       : eColorType;
    gsMixture     : sMixtureType;
END_VAR
```


Rule

The elements of enumerations (enums) are written in upper case letters. If they comprise individual words, then they are separated by an underscore character (such as constants).

Rule

Prefix und identifier are not separated by an underscore character.

Recommendation

Array limits start with 0 and end with "constant – 1".

Recommendation

Type definitions should be made in the interface section in order to ensure unique and clear definitions within the particular device.

Recommendation

Type definitions are not applied to elementary data types.

Example:

```
TYPE
    r64PositionType : LREAL; //not ok
END_TYPE
TYPE
    //array
    ar32PositionType : ARRAY[0..MAX_NUM_ELEM - 1] OF REAL;
    eColorType   : (RED,GREEN,LIGHT_BLUE); //enum
    sMixtureType : STRUCT //struct
        i16Elem1 : INT := 1;
        i16Elem2 : INT := 2;
    END_STRUCT;
END_TYPE
```


2.3.3 Prefixes for functions, function blocks and FB instances

Table 2-6: Prefixes for functions and function blocks

Prefix	Type
FC	**Function**
FB	Function block

Recommendation

The name of an FB instance should include a reference to the FB as well as the actual use

```
VAR
    FBBottleCheckOutlet : FBBottleCheck;
    FBRTC1              : RTC;
END_VAR
```

2.4 Variable definitions

2.4.1 Constants / enumeration data types

Rule

The names of constants are always written using upper case letters. In order to be able to identify individual words or abbreviations, underscore characters should be inserted between the individual words or abbreviations.

Recommendation

Direct numerical values in the code should be avoided. If at all possible, use constants.

Examples

```
VAR CONSTANT
    MAX_INDEX : INT := 99; // max. index of array
END_VAR

TYPE
    eModeType : (BUSY,FREE,RUN);   //user defined
END_TYPE

VAR
    eMode : eModeType;             //user defined
END_VAR

IF (eMode = BUSY) THEN             //use
```


2.4.2 Variables

Rule

When declaring variables, the variables are indented and are then separated by a line break.

```
//global variables--------------------------------
VAR_GLOBAL
    gtSetTime  : TIME := T#10d_5m_3s_200ms;
    //user defined
    gsMixture1 : sMixtureType;    //struct
END_VAR


//local variables--------------------------------
VAR_TEMP
    //elementary types
    u32RetValue    : UDINT;
    r64MaxVelocity : LREAL := 1.1234E002;
```

```
        //references of TO-types
        toTransportAxis : DriveAxis;
        toCrossAxis     : FollowingAxis;
        //derived types
        atoAllAxes      : ARRAY[0..MAX_NUM_AXES - 1] OF
                          DriveAxis;
    END_VAR
```

2.4.3 Initialization

Recommendation

Variables are only initialized if a default value that differs from the type-specific default value is required. If at all possible, the initialization should be made when defining the variable for runtime reasons.

Recommendation

The initialization (assigning constant data) is realized in the usual representation of its data type (literal).

Example:

```
b16Mask1    : WORD  := 16#01;       //not ok
b16Mask2    : WORD  := 16#0001;     //ok
b8Mask3     : BYTE  := 2#0000_1010; //ok
b32Mask4    : DWORD := 5;           //not ok
b32Mask5    : DWORD := 16#0000_0005;//ok
r32Temp1    : REAL  := 40;          //not ok
r32Temp2    : REAL  := 40.0;        //ok
i16Counter1 : INT   := 16#00;       //not ok
i16Counter2 : INT   := 10;          //ok
```

2.4.4 IO variable

Recommendation

Access operations to the process image of the SIMOTION device using absolute identifiers should be kept to a minimum in order to remain portable. The assignment should be made at a central location.

Example:

```
//declaration

boJogPos   AT %IX10.1 : BOOL;
b8Port1    AT %IB1    : BYTE;

i16ToolKey AT %IW10   : INT;   //cast possible
i32ProgNum AT %QD10   : DINT;

//access to I/O-Image
b8Image := b8Port1;             //ok
b8Image := %IB1;                //not ok
```

2.5 Programs

Rule

Each program is described in a descriptive header in the program code. The description contains the following points:

- Description of the functionality
- Description of the task assignment
- Requirements of the target system
- Program version together with the author and date

The template for the descriptive header is provided in Chapter 3.

2.5.1 Operators

Recommendation

A blank is located before and after binary operators and the assignment operator - as long as this does not have a negative impact on the understanding the sense.

Example:

```
i8SetValue := i8SetValue1 + i8SetValue2;  //ok
i8SetValue:=i8SetValue1+i8SetValue2;      //not ok
```

2.5.2 Expressions

Recommendation

Expressions should always be set in brackets in order to clearly show the sequence of interpretation.

Example:

```
boSetFlag := (r32ActualPosition < 100.0) OR
             (r32ActualPosition > 150.0);
```

2.5.3 Program control instructions

Recommendation

For complex expressions it makes sense to "highlight" each "sub-condition" using a line break. This means that also transparent and clear comments can be made.

Recommendation

A strict demarcation should be maintained between the condition part and the instruction part.

Rule

If one line is not sufficient for the complete condition, then Boolean logic operations are written to the right at the end of the line.

Conditions in IF instructions are indented by four blanks, THEN is located in its own dedicated line at the same height as IF.

For IF conditions in a single line THEN is written at the end of the line.

For each new structure level, the bracket character is offset by a blank so that brackets of a particular structure level are indented by the same amount.

Example:

```
IF  (FCDriveStatus() = OK) AND      //comment ...
    ((boOldDrive XOR boActDrive) OR //comment ...
     (boOldPower XOR boActPower))   //comment ...
THEN
    ; //statement
ELSE
    ; //statement
END_IF;
```

Rule

A CASE instruction must always show an ELSE branch in order to be able to signal errors that occur during the runtime.

```
CASE i16Select OF

    1:    //comment
         ;//statement
    4:    //comment
         ;//statement

ELSE
     ;// generate error message
END_CASE;
```

Rule

Each instruction in the main body of a control structure is indented.

Example:

IF statement

```
//---------------
IF  boCondition THEN
    ;//statement
    IF  boCondition THEN
        ;//statement
    END_IF;
ELSE
    ;//statement
END_IF;
```

Example:

CASE statement

```
 //-------------------
CASE i16Select OF
    1:  //comment
        ;//statement
    2:  //comment
        ;//statement
ELSE
     ;//statement
END_CASE;
```

Example:

FOR statement

```
 //---------------
FOR i16Index := 0 TO MAX_NUMBER - 1 DO
    ;//statement
END_FOR;
```

Example:

WHILE statement

```
//---------------
WHILE boCondition DO
    ;//statement
END_WHILE;
```

Example:

REPEAT statement

```
//---------------
REPEAT
    ;//statement
UNTIL boCondition END_REPEAT;
```

2.5.4 Error handling

Rule

If functions or function blocks provide error codes, then these must always be evaluated.

2.6 Functions and function blocks

Recommendation

The return type of a function FC should always be declared.

Example:

```
FUNCTION FCSwapWord : WORD   //description
```

Recommendation

Only a RETURN instruction should be used if the processing of function/function block is to be prematurely ended. There is only one return instruction for each function/function block in order to have a controlled exit point. If this rule is consciously not followed (e.g. as a result of complex control structures) then this must be documented in detail.

Recommendation

The parameters of a function/block must be specified when called in the sequence of the declaration. If feasible - default values should be specified in the blocks because it may be possible to make the call shorter and more transparent.

Recommendation

For a call in ST, a new line should be started for each parameter.

Recommendation

A lot of input parameters (call-by-value) should - as far as possible - be encapsulated in a structure (for more efficient copying). For simpler use in LAD / FBD, individual variables that frequently change their value – e.g. control variables – can be set-up as individual variables.

Note

An argument transfer as in-out parameter (VAR_IN_OUT) is efficient from the runtime perspective ("call-by-name").

e.g. to transfer higher data quantities

Example:

```
//structure within VAR_IN_OUT
FBBottleCheckOutlet(
     formulaInOut    := sFormula4711
    ,vectorInOut     := sMatrix2011[8]
    ,...
    );
```

2.6.1 FC/FB parameters

Rule

If parameters with a standard significance in regard to identifier and function are required according to PLCopen V1.1, then the appropriate standard identifiers should be used (Table 2-7). In addition, for functions/function blocks that can abort themselves the reset input to abort and the resetActive output to display an active abort (abort) are defined. These two variables are optional and are not defined in PLCopen V1.1.

Recommendation

If the programmer uses the VAR_INPUT parameter reset, then the VAR_OUTPUT parameter resetActive can be used. This parameter signals that the reset response is still present (e.g. the axis is presently being stopped and has still not come to a standstill. If the functionality can be stopped in one clock cycle (e.g. to withdraw a communication relationship), then resetActive is only present for one cycle.

Rule

Contrary to PLCopen, instead of the *inVelocity*, *inGear* and *inSync* term, *done* is always used.

Recommendation

If the *inGear* and *inSync* names are used for Boolean output variables, then these should have the same behavior as the outputs with these identifiers of the SIMOTION system function, _MC_GearIn and _MC_CamIn (set to TRUE as long as the axis runs in synchronism with the master axis).

Rule

If the programmer uses the VAR_INPUT parameter execute, the VAR_OUTPUT parameter done must be used.

Rule

Prefixes are not used for the FC and FB parameters. The reason for this is to remain in conformance with the SIMOTION parameter names and the PLCopen specifications.

Rule

Prefixes should be used for structure elements within FC and FB parameters.

Rule

For parameter names that comprise several words, the sequence of the words should be selected the same as the spoken word. The names of parameters start with lower-case letters. Names, that comprise several words, are written together and every word that has a word in front of it starts with an upper case letter.

The following parameters involve standard parameters:

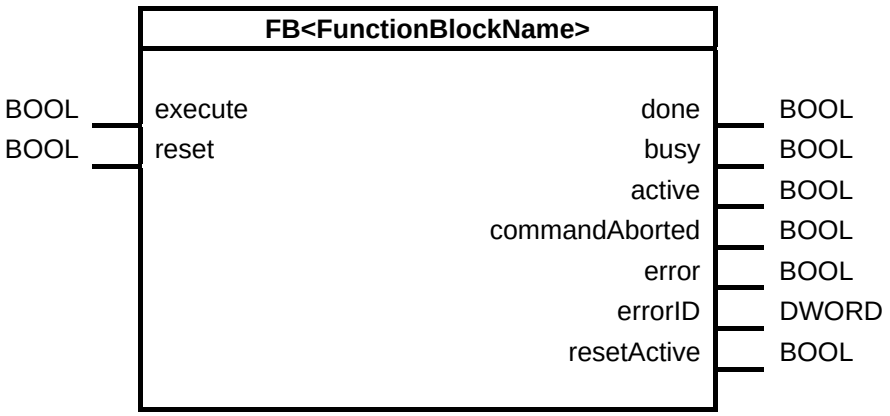
Table 2-7: Standard parameters

Signal standard function in conformance with SIMOTION	Significance
INPUT parameter	
execute or enable	The function is started with a signal edge The function is started with a signal level
reset	**Optional** to initialize or reset internal data, premature interruption of the function/request Signal level active, processing blocked
OUTPUT parameter	
done	Function or request was successfully executed
busy	Function or request is being processed/issued
valid	**optional** compatibility to PLCopen (buffered mode of function blocks), same behavior of signals like busy output
commandAborted	Function or task was aborted from outside of the FB Example: Function is still positioning the axis - but the axis is stopped at another position in the user program
valid	Only used for enable input Set if enable = TRUE and there is no fault
resetActive	**Optional** and only when using the reset input: After selecting reset, function stop/request still not completed
error	Error has occurred
errorID	Error type (error ID)

Note If simulataneous use of output active and the enumeration enumActiveInactive the compiler will report an error. For avoidance e.g. when checking the state of a TO use the following notation: <stateOfTO> = EnumActiveInactive#ACTIVE.

Example:

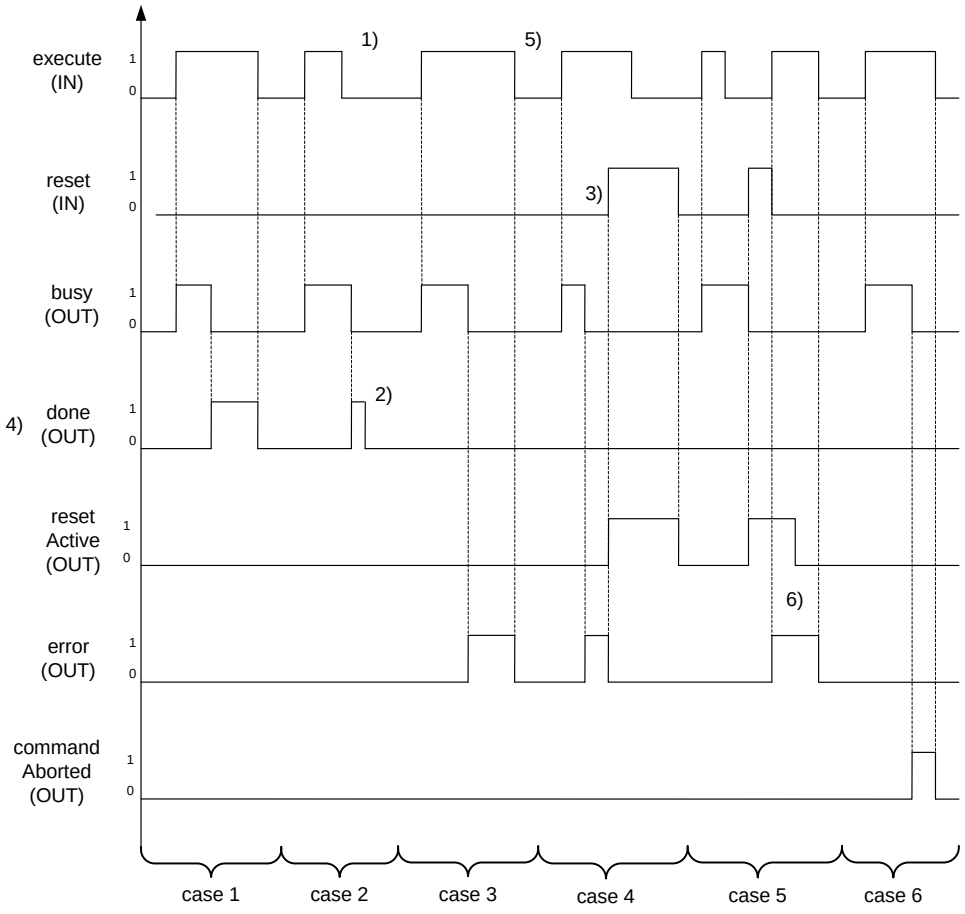
Fig. 2-1: LAD representation



2.6.2 Signal timing diagram of standardized parameters

Note If the execute parameter is withdrawn before the done bit, then the done bit should only be set for one cycle.

Fig. 2-2: Signal timing diagram of a function block with execute input



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

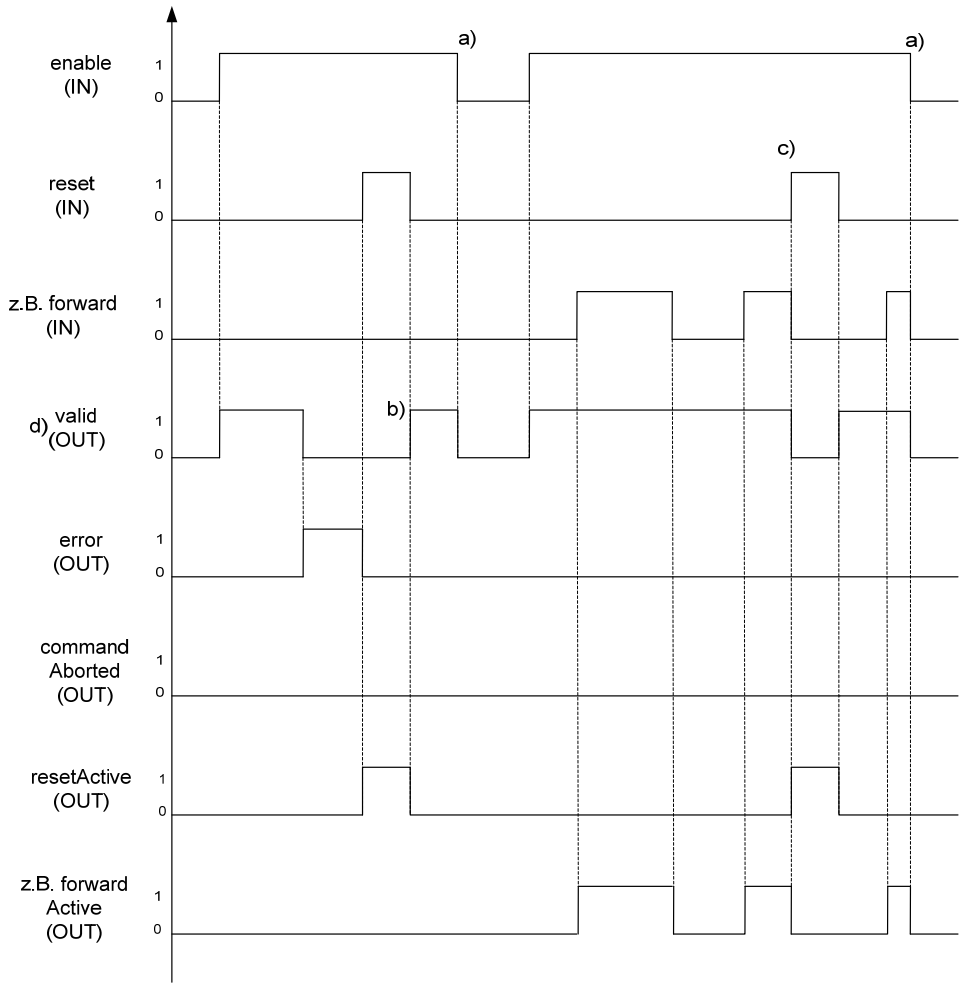
b) If *execute* is already 0, the *done*, *error* and *commandAborted* are only present for 1 cycle.

c) With a rising signal edge at *reset*, the FB function is stopped and *resetActive* is set - *busy*, *done*, *error* and *commandAborted* – if active – are reset. resetActive remains set as long as the stop response is present (e.g. axis stop, aborting the communication relationship)

d) *Done*, *error* and *commandAborted* are reset with a falling signal edge at *execute*.

e) *reset* is set to TRUE for one clock cycle. *resetActive* remains set until the function/task of the function blocked was stopped. If a rising signal edge is identified at *execute* - and *resetActive* is active - then *error* is set. The *error* bit is withdrawn again for a falling signal edge of *execute*.

Fig. 2-3: Signal timing diagram of a function block with enable input



a) With a falling signal edge at *enable* or *error* to TRUE, *valid* is reset and all FB functions are stopped.

b) *valid* is set again after the cause of the error has been removed and acknowledged.

c) With a rising signal edge at *reset*, the FB functions are stopped and *resetActive* is set, *valid* is FALSE for *resetActive*.

d) *Valid* at TRUE means that the block has been activated, there is no error and therefore the FB outputs are valid.

2.7 Libraries

The various rules and recommendations when programming libraries are specified in this Chapter. The rules relating to source code and variable names, listed in the previous chapters, are binding for standard libraries.

Functions for standard applications should be encapsulated in functions and function blocks and saved in a separate dedicated library.

2.7.1 Assigning names

Recommendation

The name of a library includes the prefix L (e.g. LCarton). No underscore characters are used. The maximum character length for a library name is limited to 8 characters.

Background:

This restriction applies to keep names small.

Rule

All of the functions/function blocks/type definitions exported from a library - as well as constants - have, as prefix, the name of the library. Type definitions that have been exported initially include the prefixes defined in Table 2-2 and Table 2-3 followed by the name of the library.

This prevents the same names being assigned as for other libraries. The use of name spaces is not permitted as the longer names would make handling with LAD / FBD / MCC more difficult.

Example:

Table 2-8: Example for assigning a name for library LExample

Type	Name according to the style guide
Library	LExample
Date type, enumeration	eLExample<Name>Type
Data type, array	aLExample<Name>Type
Data type, structure	sLExample<Name>Type
Data type, array of a structure	asLExample<Name>Type
Exported function block	FBLExample<Name>
Exported function	FCLExample<Name>
General constant	LEXAMPLE_<NAME>
Constant for the error code	LEXAMPLE_ERR_<NAME>

Rule

Each unit of a library contains one prefix. The rules from Chapter 2.2.4 apply.

Rule

Each library contains one unit for the version history. This unit has the name aVersion.

Rule

Extensive type definitions and constants are declared in separate, dedicated units.

Rule

There are a maximum of two units to define type definitions in a library that can be exported from the library. A know-how protected unit contains the dProtected name and an open unit has the dPublic name.

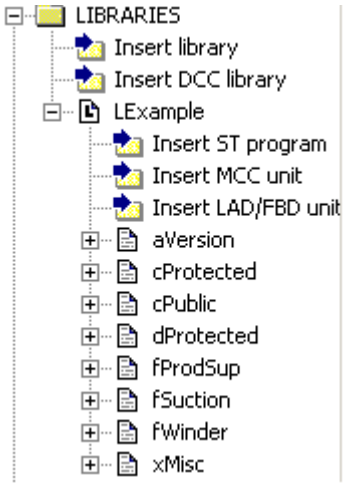
Rule

There are a maximum of two units to define global constants in a library. One know-how protected unit contains the cProtected name and an open unit, the cPublic name.

Note The name for units for data and constant definitions are only defined for libraries. A name for units with data or constant definitions in the application is not specified with the exception of the prefix (refer to Chapter 2.2.4).

Example:

Fig. 2-4: Example for the prefixes of library units



2.7.2 Structure

Rule

A library is created for each application (e.g. DPV1 services) or machine type (tubular bagging machine) or machine types that are very similar from

a technological perspective (e.g. vertical and horizontal tubular bagging machines).

Rule

Functions/function blocks that are used more than once within the library are general function/function blocks. If these functions/function blocks are only used within the library, then the library name is not included in the identifier. General functions/function blocks are saved as follows, depending on the scope:

- Small functions/function blocks are linked into each unit of the library. This therefore increases the update time & costs. The reason for this is that each unit must be updated when it is changed. However, documentation does not have to be generated as these functions/function blocks are not exported.
- If general functions/function blocks are too extensive both regarding the number (quantity) or code, then they are saved in the library of a separate, dedicated unit. Every exported function/function block of this general unit must be documented. However, a reference to the internal use in the documentation is sufficient.
- Central and higher-level functions/function blocks should be saved in their own separate libraries to make it simpler to reuse them.

Rule

The number of units of a library should be kept to a minimum.

Rule

Know-how protection is not applied to a complete library, only to the individual units. This means that it is possible to subsequently modify the device version.

Background:

For a know-how protected library, the platform as well as the SIMOTION-version cannot be changed without a password.

Rule

A library can contain both know-how protected as well as also non know-how protected units.

This therefore allows e.g. data definitions to be adapted (size of ARRAYs) that are processed by functions where the know-how is protected, without obtaining access to the source code.

2.7.3 Programming

Rule

Libraries are preferably created in the ST programming language. This allows more complex algorithms to be efficiently programmed.

If a library also contains simpler functions or logic components, then these can be programmed in graphic languages.

Rule

The functions/function blocks that have been generated must be suitable for use in LAD/FBD charts. The handling of the function block should be kept as simple as possible for users – even if this requires a somewhat higher level of programming time & resources.

Recommendation

The function blocks should be programmed for cyclic operation. It is preferable that the background task is used for the task to be processed. This is also the reason that the use of an endless loop (e.g. WHILE TRUE) should be avoided for cyclic functionalities in a motion task.

Background: When using motion tasks with endless loops it is not possible to download when the CPU is in the RUN mode.

2.7.4 Structure of a library unit

Rule

Only the structures, functions and function blocks that are required for export are declared in the INTERFACE section of a unit. This therefore allows data to be encapsulated.

Local data is preferably declared in the IMPLEMENTATION part of a unit (unit global) or in the POUs.

Background: The documentation costs can be lowered by reducing the codes and/or constants and types that can be exported.

Rule

Global definitions (data, block instances) that prevent or restrict the instantiating of the library functions or the general use of functions/function blocks may not be used within libraries.

Exception:

Global definitions are permitted in libraries and/or functions/function blocks that as a result of their definition or function exclude instantiating.

2.7.5 Programming function blocks for libraries

Rule

For function blocks whose parameters must remain consistent when processing across several calls it must be guaranteed that the values that are used remain constant (e.g. by saving the input parameter in a help variable and inhibiting these help variables from being written to as long as a value change is not permitted).

Rule

Due to the fact that there is no debug support for libraries, special emphasis must be placed on being able to test functions by monitoring variables in the symbol browser. In this case, internal variables must be defined in a suitable form so that they provide adequate information about the state and sequences of the functions. For instance, this could include the last processing state or the actual step number.

Recommendation

If a lot of parameters are transferred, then a type VAR_IN_OUT variable is used. A structure for example, configuration data, actual values, setpoints, TO references, output of the actual state of the function block etc. is generated for this variable. For control and/or status variables that often change, it may make sense to declare these as VAR_INPUT or VAR_OUTPUT to ensure simple access in LAD/FBD.

Recommendation

Several VAR_IN_OUT type variables can be used if parameter structures are to be saved at various memory locations (setpoints in the retain area, values from/to the HMI in the global area).

Recommendation

Numerical parameters for which there are default values in SIMOTION, (e.g. velocity, acceleration, jerk) are initialized with a value of -1.0. This is used to distinguish whether a value is transferred for the parameter. The default values of the TO are accepted if the user does not make an appropriate assignment.

Rule

For performance reasons, actual values of technology objects, which are not used in the function block, are not written into a VAR_IN_OUT parameter structure or into an OUTPUT variable.

Rule

Structures of arrays should be set-up instead of arrays of structures.

Background: This simplifies handling in the HMI and also results in faster data transfer to the HMI.

Recommendation

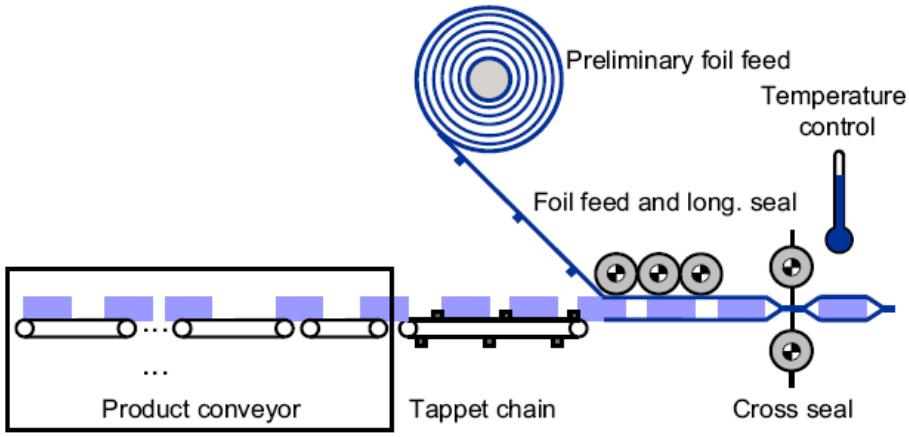
The PLCopen function blocks contained in the SIMOTION system functions are used to program standard functions such as jogging, homing,

Programming function blocks for machine units

Functional units of a machine are combined to form aggregates in a machine library. An aggregate comprises e.g. axes as well as sensors and

actuators. Modular library use is supported by sub-dividing the machine into aggregates.

Fig. 2-5: Aggregates for a tubular bagging machine



The aggregates of a horizontal tubular bagging machine are shown in Fig. 2-5. The following can be combined to form individual aggregates

- Product conveyor
- tappet chain
- Preliminary foil feed
- Foil feed
- Cross Seal

Rule

A machine aggregate is only controlled from the aggregate function block. When programming, it is also possible to take into account various technical versions of an aggregate and be able to select these versions using configuration variables.

If the differences between the technical versions are very extensive, then separate function blocks can be created for the different versions.

Rule

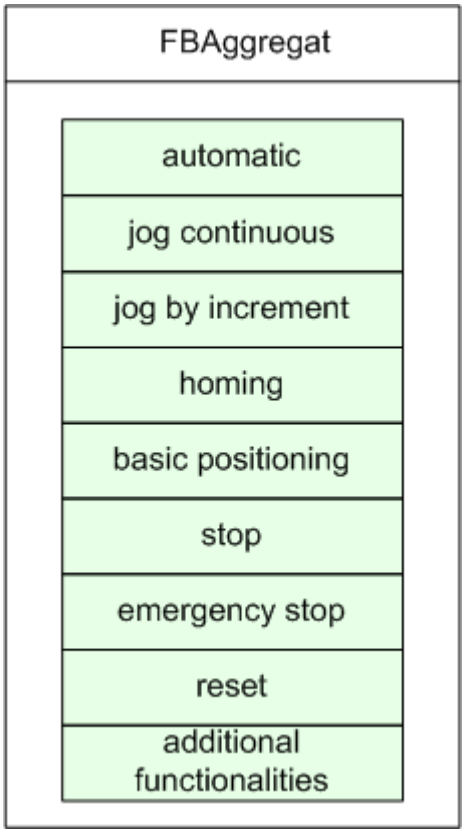
The aggregate function block provides all of the functions required to operate the machine aggregate. The functions of an aggregate are listed in Fig. 2-6. Less functions are also possible for simple aggregates (e.g. a fan).

All of the functions refer to the total aggregate with its associated axes as well as additional technology objects (e.g. cams, measuring inputs). For

and measuring inputs will also be deactivated..

The functions are controlled depending on the actual machine mode.

Fig. 2-6: Functions of a function block for an aggregate



Rule

Depending on the complexity and scope of an aggregate - or also depending on the degree of standardization - it may make sense to shift out individual functions of an aggregate into subordinate functions and/function blocks.

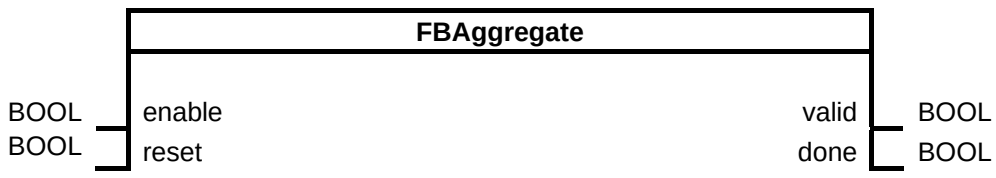
In this particular case, the aggregate function block only handles the coordination. The actual function itself is processed by the subordinate blocks. All of the parameters required for this are transferred to the subordinate block.

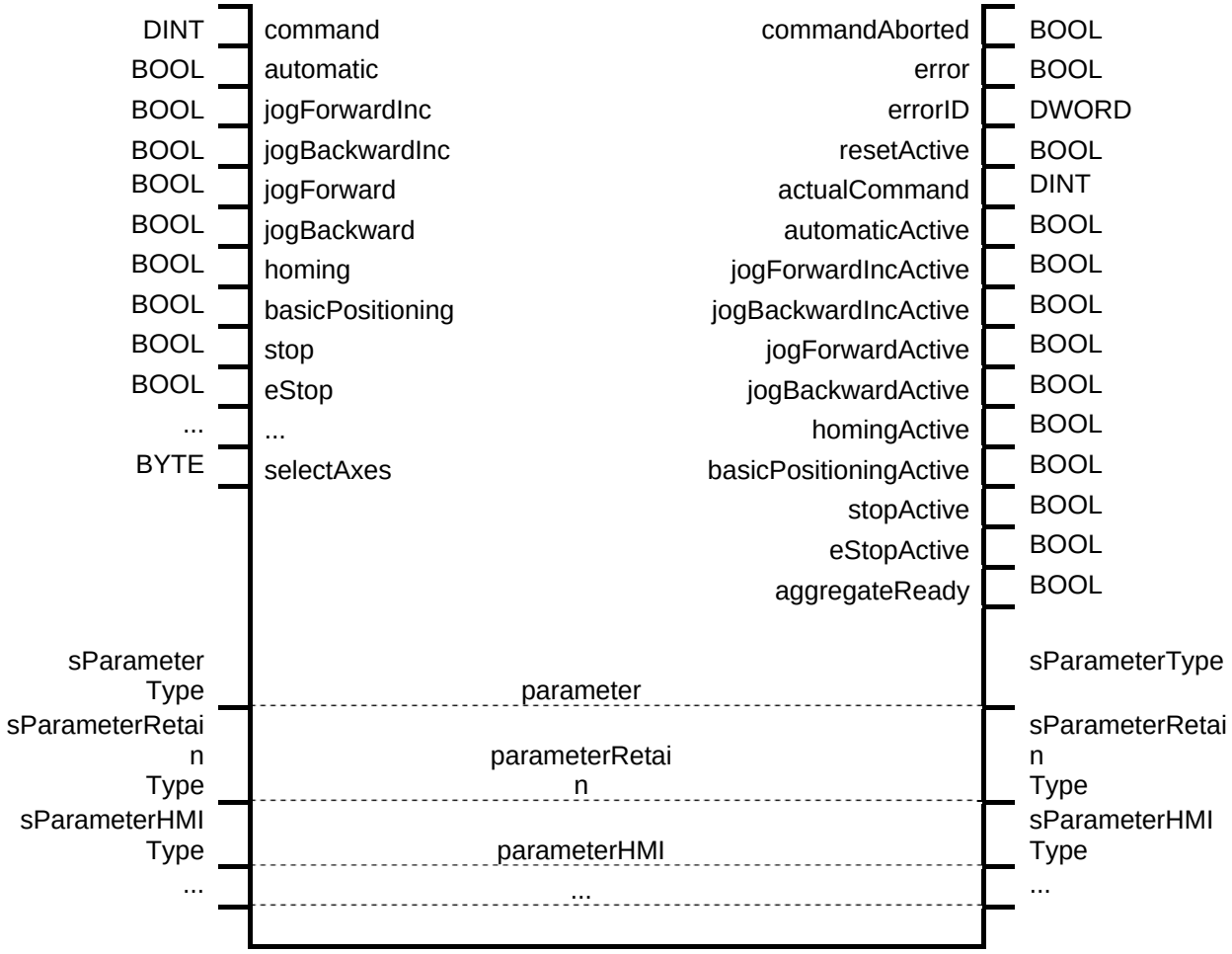
Functions with a low complexity can be directly implemented in the aggregate function block by calling system functions or PLCopen blocks.

Recommendation

The interface for an aggregate function block is shown in the following diagram.

Fig. 2-7: LAD representation for an aggregate function block





State control of the FB in ST as well as LAD / FBD

The control of the functions of an aggregate can be realized in two ways depending on the programming language being used and the opinion of the programming engineer.

In ST, the handling of a numerical value offers the highest degree of transparency.

On the other hand, for state control from logic programs (LAD/FBD) the handling of individual Boolean inputs is simpler and more transparent.

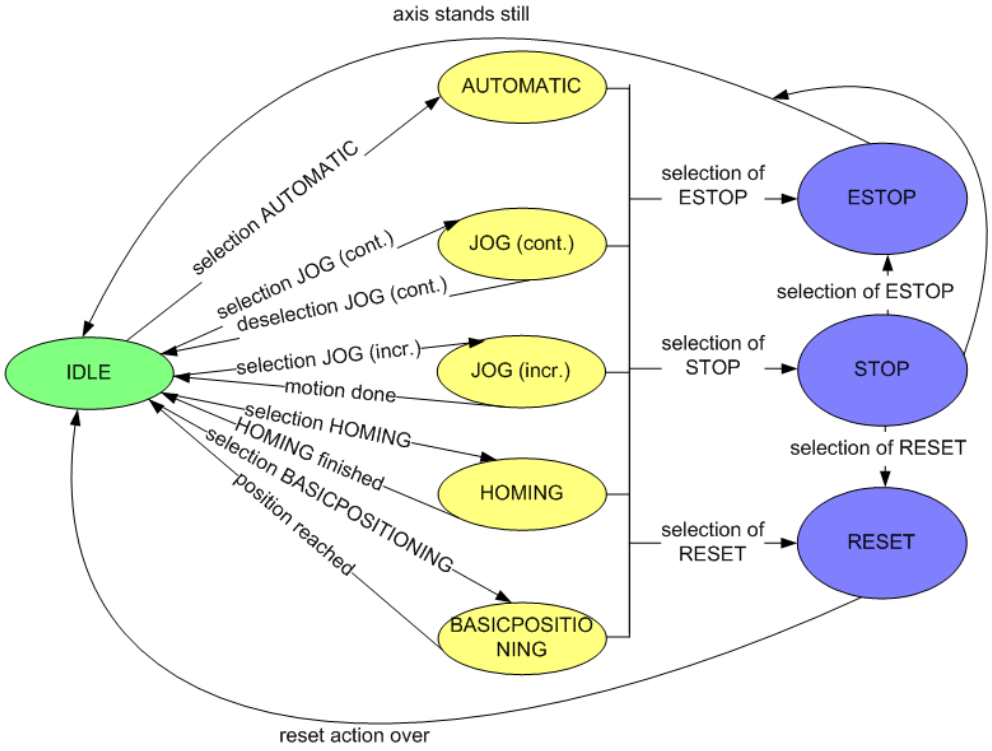
The AggregateFB offers both interfaces. The numerical interface (DINT input *command*) has a higher priority. The Boolean interface is used if the *command* input is not interconnected with a variable.

The selection of functions with numerical value and/or Boolean input is shown in Table 2-9. A positive signal edge is used to start processing a particular function.

The actual state is represented in the same way using a numerical interface (DINT output *actualCommand*) as well as also using Boolean outputs. All of the outputs are always updated.

Behavior of the operating states

Fig. 2-8: Transitions between the functions of the AggregateFB



AUTOMATIC, JOG, HOMING (REFERENCING) and BASICPOSITIONING can only be started from the IDLE state.

The processing of AUTOMATIC, JOG, HOMING and BASICPOSITIONING can be aborted by selecting STOP and ESTOP.

When de-selecting the JOG continuously state (command = 0 or a negative signal edge at the corresponding Boolean input) then a transition is made into the IDLE state.

In the JOG incremental state, a transition is made into the IDLE state after an increment has been traveled through. A new positive signal edge is required for an additional increment.

After homing has been completed, the system transitions from the HOMING state into the IDLE state.

After positioning has been completed, the system transitions from the BASICPOSITIONING state into the IDLE state.

After a stop operation has been completed for STOP, ESTOP, a transition is made into the IDLE state.

The *done* output bit is set after successfully executing the states JOG, HOMING, BASICPOSITIONING, STOP, and ESTOP.

The AUTOMATIC state can only be stopped by selecting STOP and ESTOP.

If both inputs of a jog mode are set, then jogging is not executed or if jogging is active, then it is stopped.

The operating states are assigned priorities as shown in Table 2-9. Higher priority operating states can replace low-priority states. A value of 1 has the highest priority and a value of 5 the lowest priority.

Table 2-9: Numbering the operating states and assigning priorities

Required operating state	Number	Priority	Description
idle	0		Idle state
eStop	10	1	Emergency stop of the aggregate
jogForward (incr)	20	5	Position forward by one increment (jog)
jogBackward (incr.)	30	5	Position backward by one increment (jog)
jogForward (cont.)	40	5	Constant travel forwards as long as selected (jog forward)
jogBackward (cont.)	50	5	Constant travel backwards as long as selected (jog backward)
homing	60	5	Referencing (homing) the axes of the aggregate
manual (*)	70	5	
basicPositioning	80	5	Basic positioning of the axes
automatic	90	5	Automatic mode of the aggregate
stop	100	3	Axis is immediately stopped
holdAtEndOfCycle (*)	110	4	The axis is stopped after the actual product has been completed/finished
singleStep (*)	120	5	A clock cycle/step is executed
userDefined (*)	> 1000		

*) Is not programmed in the aggregate (template).

The currently active mode is displayed using Boolean outputs as well as a DINT output.

Selecting certain axes of the aggregate

The *selectAxes* input is used, if jogging, homing and basic positioning should not be executed for all axes of an aggregate. The bits of this byte are used to code for which axes the functionality should be executed. Depending on the assignment of the byte, several axes can be simultaneously moved.

Parameter structure of an aggregate function block

The configuration as well as setpoints and actual values are transferred to the function block via structures and are VAR_IN_OUT data type. A parameter structure is shown in Fig. 2-9 and an explanation is given in Table 2-10.

The programming engineer should define the sub-structures.

Depending on the memory location (e.g. Retain, Global, ...) as well as use (e.g. HMI), then more than one IN_OUT structure can also be created.

Fig. 2-9: Parameter structure of an aggregate function block

```
TYPE
    s<LibName>ParameterType : STRUCT
        //for technology objects
        sTOs                : s<LibName>TOType;
        //for config data
        sConfigData         : s<LibName>ConfigDataType;
        //for command values
        sCommandValues      : s<LibName>CommandValuesType;
        //for dynamic parameter
        sDynamics           : s<LibName>DynamicType;
        //for actual values
        sActualValues       : s<LibName>ActualValuesType;
        //for diagnostics
        sDiagnostics        : s<LibName>DiagnosticsType;
    END_STRUCT;
END_TYPE
```

Table 2-10: Description of the elements of the sParameterType structure

Structure element	Description
sTOs	Required technology objects, array for each TO type used
sConfigData	Configuration data, static data that do not change in operation (e.g. machine configuration)
sCommandValues	Setpoints (e.g. velocities, positions)
sDynamics	Dynamic parameters, if necessary with sub-structures for dynamic parameters of the individual functionalities
sActualValues	Actual values
sDiagnostics	Diagnostics structure, e.g. current step

Recommendation

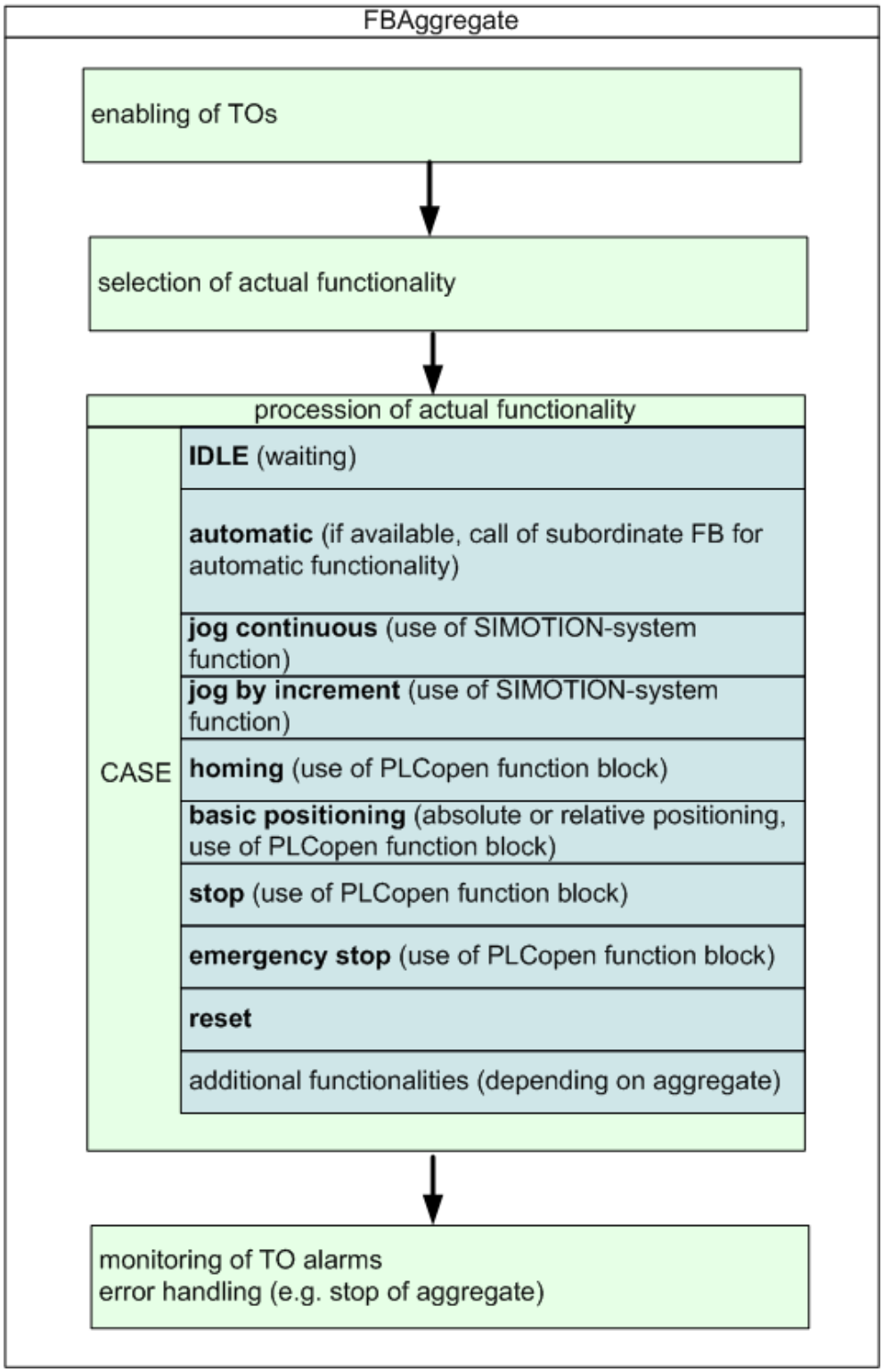
If several aggregates are described in a library, then a parameter structure should be used for all aggregates with as few sub-structures as possible. It must be clearly documented if structure variables are not used.

Rule

The sequence of an aggregate block is shown in

Fig. 2-10: .

Fig. 2-10: Sequence of the aggregate function block



Cyclic:

- Enable the axes and additional TOs if still not enabled
- Set aggregateReady if all TOs are enabled
- Select the actual functionality (interrogate the inputs for the new state, check for changes to the current state, set the new state)
- Check the parameters only when selecting (rising signal edge) the functionality
- Execute functions in a case structure, use the PLCopen blocks, use a subordinate function block for automatic operation.
- Cyclically interrogate for alarms at the active technology objects, for alarms, set the error bits
- Initiate an error response when an error occurs (e.g. stopping of axes, de-activate output cams)
- When enable is de-selected, de-activate the TOs

2.7.6 Error return and diagnostics of function blocks

Rule

An error is displayed by setting the Boolean variable *error*. An error code that refers to the cause is output by the *errorID* variable, DWORD data type.

Rule

If the error involves a technology object (e.g. TO alarm, no TO object transferred to a system function, incorrect parameterization), then the technology object is coded in the first word (high word) of *errorID*.

The error number, which defines why the error occurred, is output in the second word (low word) of the *errorID* variable.

If the error is not caused by a technology object, the first word of *errorID* is assigned *16#0000*.

Rule

The technology object type is coded using a numerical value, refer to Table 2-11 – and is saved in the first byte (high byte) of the first word of the *errorID* variable.

Table 2-11: Technology object type coding

Technology object	TO type coding
No TO	16#00
Speed controlled axis (driveAxis)	16#01
Positioning axis (posAxis)	16#02
Synchronous axis (followingAxis)	16#03
Synchronous object (followingObjectType)	16#04

Technology object	TO type coding
Cam (camType)	16#05
Measuring input (measuringInputType)	16#06
Output cam (outputCamType)	16#07
External encoder (externalEncoderType)	16#08
Temperature controller (temperatureControllerType)	16#09
Fixed gear (_fixedGearType)	16#0A
Addition object (_additionObjectType)	16#0B
Formula object (_formulaObjectType)	16#0C
Sensor (_sensorType)	16#0D
Controller object (_controllerType)	16#0E
Cam track (_camTrackType)	16#0F
Path axis (_pathAxis)	16#10
Path object (_pathObjectType)	16#11
Additional objects	16#12 ..16#FF

Technology objects of the same TO type, as described in Chapter 0, are transferred in an array in the sub-structure sTOs. The index of the arrays is written to the second byte of *errorID* in order to be able to make a clear differentiation between technology objects of the same type.

Fig. 2-11: Structure of the error ID for errors caused by a technology object

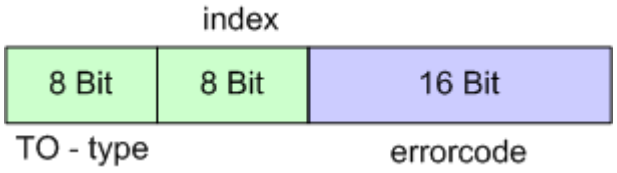
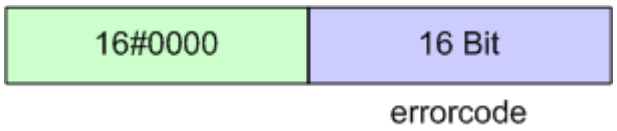


Fig. 2-12: Structure of the error ID for errors that are not caused by a technology object



Rule

Return codes from a system function are not directly output at output *errorID*. The reason for this is that they are often difficult for users to understand. The return codes from system functions are output in their own range of numbers.

Before calling the system function, an interrogation is made for any incorrect parameters assignments or illegal states of the axis (the axis has not been released, axis has not been homed if necessary).

Rule

In order to standardize the errors, the range of numbers indicating the reason for the error shown in the following table, should be used.

Table 2-12: Ranges of numbers for errors

Error reason	Number range for FB return value
No error	0
Function block incorrectly handled/operated	16#1000 .. 16#1FFF
A function was selected with reset active	16#1001
Invalid operating state selection	16#1002
Error when parameterizing	16#2000 .. 16#2FFF
Error when executing from external (e.g. incorrect I/O signals, axis not homed)	16#3000 .. 16#3FFF
Error when executing from internal (e.g. when calling a SIMOTION system function)	16#4000 ..16#4FFF

Rule

If an error is identified when processing a function block, then the actual request/motion is stopped. The error code associated with the first error remains until it has been acknowledged (select *reset*, negative signal edge from *execute* or *enable*).

Rule

All of the additional information about the errors that have occurred while processing a function block should be saved in a diagnostics structure.

Fig. 2-13: Diagnostics structure

```
TYPE
    sDiagnosticsType : STRUCT
        r64AdditionalValue1 : LREAL;
        r64AdditionalValue2 : LREAL;
        r64AdditionalValue3 : LREAL;
        i32ModeNumber       : DINT;
        i32StateNumber      : DINT;
        b32ReturnCode       : DWORD;
        b16ErrorId          : WORD;
        b8TOType            : BYTE;
        b8TONumber          : BYTE;
    END_STRUCT;
END_TYPE
```

The TO type is coded in element *b8TOType* (refer to Table 2-11); the index of the technology object from the corresponding array is coded in element *b8TONumber*.

If a SIMOTION system function error is identified, then the return code of this system function is saved in the element *b32ReturnCode*.

The error code from *errorId* is additionally saved in element *b16ErrorId*.

Additional parameters associated with an error are saved in the *r64AdditionalValue* variables.

For a function block with various modes, e.g. the aggregate function block, the mode is saved in the variable *i32ModeNumber*, in which the error actually occurred.

Rule

If a step chain is used for programming, then the actual step number is entered into the element *i32StateNumber*.

Background: The user then receives information regarding the condition that the function block is waiting for before continuing to process.

Rule

Each diagnostics structure of a function block includes, as a minimum, the elements from Fig. 2-13. If additional elements are required, then these can be appropriately added.

2.7.7 Versions

Rule

The official version (first release status) starts with version V 1.0. Version releases less than 1.0 indicate that it involves a development release.

The document version has two digits and the software version has three digits. The first two digits match one another.

The third digit in the software version designates revisions that have no effect on the documentation; for example pure debugs that do not involve any new functions.

When expanding the existing functionality, the second digit is incremented.

For a new main version, that has new functions, then the first digit is incremented.

Rule

Functions/function blocks are not changed so that they become incompatible to a previous version. If such a change is required, then a new function block should be created and the old function block should be kept in the library.

Rule

The library versions are without any gaps. When changes are made, the version number of the library is incremented. The modified function block/function as well as the unit, in which the changes were made, includes the actual number of the library version.

Example:

In the example shown below, function block FB1 and function FC1 are in UnitA, function block FB2 is in UnitB. Both units are included in the library.

Table 2-13: Example of changing the version

Library	UnitA	UnitB	FB1	FB2	FC1	Comment
1.0.0	1.0.0	1.0.0	1.0.0	1.0.0	1.0.0	Released
1.0.1	1.0.1		1.0.1			FB1 has been debugged
1.0.2		1.0.2		1.0.2		FB2 has been optimized
1.1.0	1.1.0		1.1.0			FB1 has been expanded
1.2.0	1.2.0		1.2.0			FB1 has been expanded
2.0.0	2.0.0		2.0.0		2.0.0	New functions in FB1 and FC1
2.0.1		2.0.1		2.0.1		FB2 has been debugged

Rule

For each change of the version, the adaptations are described at the library / unit / function block / function.

Rule

The actual library version is additionally entered into the properties dialog box of the library. For standard libraries, APC is saved as author in the properties window.

Template for versions unit aVersion

```
//SIEMENS AG
//(c)Copyright 2007 All Rights Reserved
//-----------------------------------------------------------------------------
// unit for version history of a library
// file name: aVersion
// library: (that the source is dedicated to)
// system: (target system)
// version: (SIMOTION / SCOUT version)
// restrictions:
// requirements: (hardware, technological package, memory needed, etc.)
// functionality: (that is implemented in the library)
//-----------------------------------------------------------------------------
// change log table:
```

```
// version  date       expert in charge changes applied
// 01.00.00 dd.mm.yyyy (name of expert) only refer to changed unit and
//                                      function block/ function
//=============================================================================

//--this part is only necessary because of use of a unit for version history-
INTERFACE
END_INTERFACE

IMPLEMENTATION
END_IMPLEMENTATION
//-----------------------------------------------------------------------------
```

2.7.8 Performance test

Recommendation

Before a library is supplied, the function blocks should be tested on a slow CPU (e.g. D425) in order to better identify performance problems.

2.7.9 Delivery

Rule

To program the source code, all compiler warning classes are activated in the properties dialog box of the library. When compiling the library with the generated version, no warnings are to be displayed that the programmer would consider as being non-relevant. These warnings should be suppressed using a pre-processor instruction. Warning suppression should only be activated for the code section that generates the warning.

The pre-processor instruction to suppress a warning has the following syntax:

Fig. 2-14: Suppressing compiler warnings

```
//suppression of compiler warning <warningNumber>
{_U7_PoeBld_CompilerOption:= warning:<warningNumber>:off;}
```

To suppress warnings, "off" is used as option and to activate warning output, "on" is used. A comment should indicate which warning is suppressed.

Rule

A library is not supplied as project, but as XML file. This is inserted using a script in a project.

Recommendation

A script can be supplied to generate a code example.

Recommendation

A script can be supplied to parameterize specific aggregates - e.g. axes. From an HTML interface, settings can be made at the axis, e.g. the modulo length or a calibration/adjustment of the reference torque from the drive and axis.

2.7.10 ST code template for an aggregate function block

A template for the aggregate function block is provided in the Intranet through Application Support, production machines under Standard Applications, topic "Style guide“ or on the Utilities & Applications CD. Type definitions for the parameter structure, constants as well as runnable code to select the current function and a CASE instruction in which the sequences of the individual functions are programmed are contained in this.

3 Template

From the perspective of the development engineer, a description is made within the program (development engineer documentation). The description uses templates and comments. The development engineer defines the source codes and documentation at one location.

The documentation is sub-divided into:

1. Documentation of the functionality
2. Documentation of the changes/debugs

Depending on the customer's requirement and the possibilities open to the programming engineer, the documentation language is either in the mother tongue or English.

STcode templates for the formal structure of a unit as well as for a function block according to PLCopen are provided in the Intranet through Application Support, product machines under Standard Applications with the topic "Styleguide" or on the Utilities & Applications CD.

4 Documentation

The documentation for the project or the application is drawn-up specifically to comply with the project requirements. The existing templates should be used if individual blocks have to be documented.

Rule

An example to call the function block is supplied in the ST and LAD programming languages.

Rule

Every exported function / function block / type definition is documented.

A template to document units / functions / function blocks and/or programs is available on SIMOTION Utilities & Applications, section tools.

Appendix

5 Revisions/author

Table 5-1: Revisions/author

Version	Date/revision	Author
Origin	03.01.2008 / based on ST Style guide 3.5, expanded by specifications for libraries, changeover to PLCopen V1.1	
1.01	19.03.2008	
1.02	17.12.2009	

6 Literature

Literature

This list is in no way complete and only reflects a selection of suitable literature.

Table 6-1

	Subject	Title
/1/	SIMOTION	System Manuals
/2/	PLCopen	Function blocks for motion control Version 1.1

7 Contact partners

Application Center

SIEMENS

Siemens AG
Automation & Drives
A&D MC PM APC
Frauenauracher Str. 80
D - 91056 Erlangen
Fax: +49 9131-98-1297
mailto: applications.erlf.aud@siemens.com