

Guidelines for Library Design

Document Version 1.52

Last Update: 18.05.2011

CONTENTS

1	GENERAL	3
1.1	Project Information	3
1.2	Library Categories	6
1.3	Template Libraries and Placeholder Concept	6
1.4	External Libraries and Placeholder Concept	7
1.5	Placeholder Concept and Device Description	8
1.6	Library Reference Summary	8
1.7	Version History	8
2	REQUIREMENT SPECIFICATION	9
3	INTERFACES	9
3.1	General	9
3.2	Variable names	9
3.2.1	Variable names in CoDeSys V3.x libraries	9
3.3	User defined data types (DUT)	9
3.3.1	DUTs in CoDeSys V3.x libraries	9
3.4	Functions, function blocks, programs (POU)	9
3.4.1	POUs in CoDeSys V3.x libraries	9
3.5	Visibility of project items in CoDeSys V3.x libraries	9
3.6	Version POU	9
3.7	CoDeSys Project	9
3.8	Behaviour- and interface model of function POUs	9
3.9	Error handling	9
3.9.1	Function Blocks	9
3.9.2	Functions	9
3.10	Lists and arrays	9
3.11	GetProperty function	9
4	FUNCTION DESCRIPTION AND EXAMPLES	9

5	HELP	9
6	TEST- AND CERTIFICATION PROJECTS	9
7	RELEASE	9
	APPENDIX A: SYMBOL CONFIGURATION	9
A.1	Filtering of symbols inside the Symbol Configuration	9
A.2	Limit the number of symbols by proper library design	9
	CHANGE HISTORY	9

1 General

One area of operation of the CoDeSys Automation Alliance Technical Workgroup is the creation of non-ambiguous library specifications. Libraries only specified by their interfaces might show different behaviour in different implementations (Example: Reading from a serial interface: what happens if there are less characters than have been requested). It might happen that CoDeSys applications basing on such libraries do not work on each target system.

In order to avoid these problems it is necessary to specify libraries non-ambiguously and completely, namely by the definition of:

- Interfaces
- Function / behaviour
- Help texts
- Examples
- Test and certification procedures

Simple hardware-independent internal libraries additionally can be implemented open accessibly.

The specification of the libraries must be **downward compatible** with previous version. **Extensions** of specifications or **clarifications** of non-ambiguous definitions are **possible**.

Ambiguities, detected by the participants of the work group during the implementation, should be reported to and gathered by the group and presented during the next workshop, and – if necessary – they should get reworked.

It is planned to established a **Newsgroup** forum, subdivided into areas of operation and documents, where each member of the workshop can make comments.

1.1 Project Information

A **short description** of the library is to be entered in the library project information.

For **CoDeSys V3.x libraries** the following entries in the **Project Information** are obligatory:

- **Company:** For templates and interface conventions as an author identifier should be entered: „**CAA<space>Technical<space>Workgroup**“. In case of target-dependent implementations the respective company name must be entered (e.g. „3S<space>Smart<space>Software<space>Solutions<space>GmbH“).
- **Title:** Library name (e.g. “CAA Callback”). In case of a target-dependent library the target name (e.g. “CoDeSys SP Win V3”) should be appended to the library name.
- **Version:** 3.<specification version>.<release version>.<implementation version> (see also 3.5).
- **Categories:** At least one of the subcategories described below should be entered.
- **Namespace:** The library prefix should entered as Default namespace.
- **Author:** Company name of the author (see „Company“).
- **Description:** Short description of the library.

Project Information

File Summary Properties Statistics

Company: CAA Technical Workgroup

Title: CAA TickUtil

Version: 3.0.0.1

Library Categories: Intern|CAA ...

Default namespace: TICKU

Author: 3S - Smart Software Solutions GmbH

Description: Simple helper functions for calculation and converting with CAA.TICK datatype.

The fields in bold letters are used to identify a managed library.

☐ Automatically generate POUs for property access

OK Cancel

Project Information

File Summary Properties Statistics

Company: 3S - Smart Software Solutions GmbH

Title: CAA Tick (PLCWinNT V3.x)

Version: 3.0.0.0

Library Categories: Target|3S - Smart Software Solutions GmbH ...

Default namespace: TICKS

Author: 3S - Smart Software Solutions GmbH

Description: Simple target dependant abstraction of a system tick

The fields in bold letters are used to identify a managed library.

☐ Automatically generate POUs for property access

OK Cancel

Project Information

File Summary Properties Statistics

Key: SpecVersion Add

Type: Version Modify

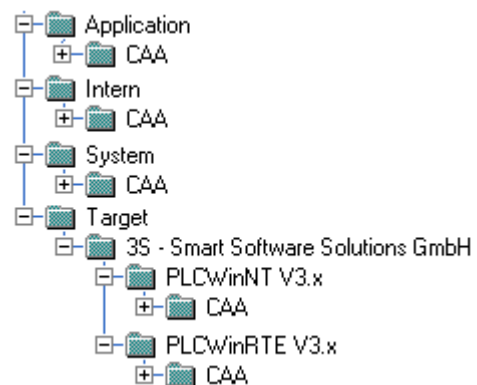
Value: 3.0.0.0 Remove

Properties:

Key	Value	Type
Company	3S - Smart Softwar...	Text
DefaultNamespace	TICKS	Text
Description	Simple target depen...	Text
SpecVersion	3.0.0.0	Version
Title	CAA Tick (PLCWinN...	Text
Version	3.0.0.0	Version

☐ Automatically generate POUs for property access

OK Cancel



Hierarchy of the library categories

- **Properties:** The following properties should be inserted

Property name (Key,Type)	Description (Value)
SpecVersion (version)	A reference on the specification version provided by the CoDeSys-Technical Workshop.
Depends on RTS (bool)	Describes the dependency on the runtime system.
Depends on PS (bool)	Describes the dependency on the programming system.
Licensed (bool)	The library should be licensed
Documentation (string)	The path and file name of the specification file
LanguageModelAttribute (string)	Only access over the namespace ('qualified-access-only')
ShowSmartCodingInfo (bool)	True False
ContainerLibrary (bool)	True False
InterfaceLibrary (bool)	True False

- The option for automatic generation of POU's for accessing the project properties should not be activated.

1.2 Library Categories

There are the following main categories for CAA libraries:

- **Application:** Libraries of this category are target-independent and are explicitly to be inserted in a project by the user. They are implemented matching the CAA rules and are sorted in subcategory „Application|CAA“.
- **Intern:** Libraries of this category get automatically inserted in a CoDeSys 3.x project by certain CoDeSys 3.x extensions (plugin) and in normal case should not be inserted or removed by the end user. They are implemented conforming the CAA rules and are sorted in subcategory „Intern|CAA“.
- **System:** Libraries of this category directly depend on the runtime system or implement parts of the runtime system. Usually they are not directly inserted in the CoDeSys 3.x project, but are referenced by other libraries. They are implemented conforming the CAA rules and are sorted in subcategory „System|CAA“.
- **Target:** Libraries of this category are not target-independent and are a vendor-specific implementation of CAA specifications. Below the main category „Target“ each vendor creates a subcategory named like his company. Below the vendor category a further subcategory should be created for each target, named like the target. All libraries for this target should get assigned the respective category in their project information. They are to be implemented conforming the CAA rules and are sorted in subcategory „Target|<vendorname>|<targetname>|CAA“.

Each library category is defined in a xml-file. An appropriate template (LibraryCategoryBase.libcat.xml) is provided with the current document.

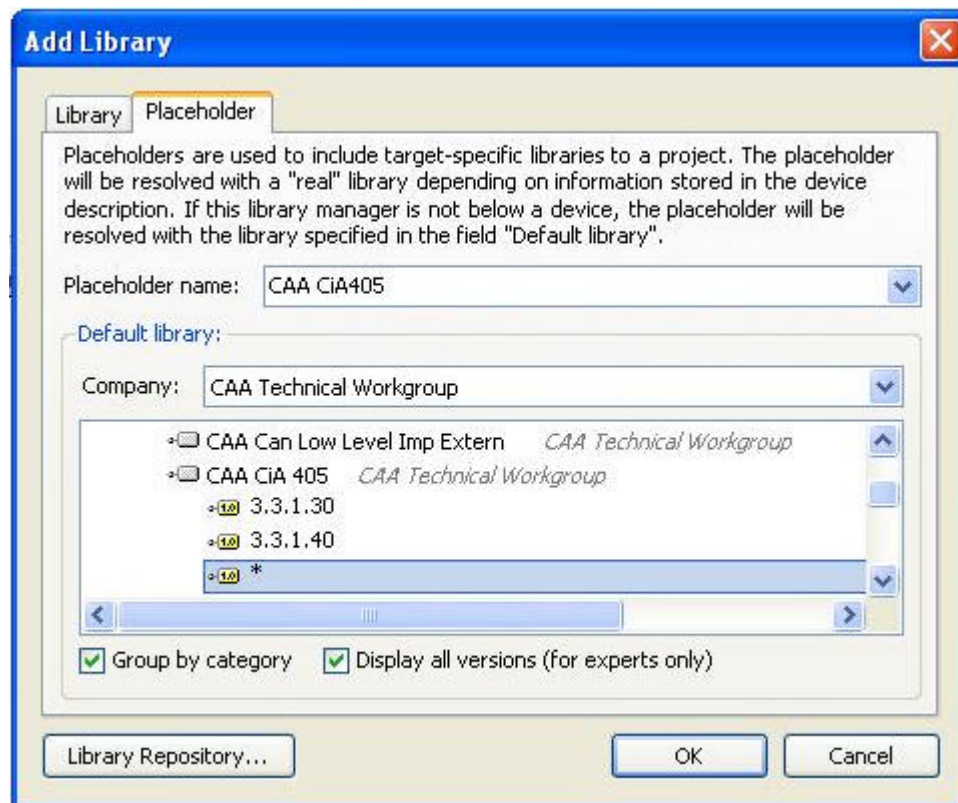
1.3 Template Libraries and Placeholder Concept

For each target-specific library a corresponding **template library** must be created, which only contains the interfaces and data types of the real library.

To enable that target-independent libraries can base on target-specific libraries without any problems, a **placeholder concept** has been established in CoDeSys V3.x. It guarantees a strict versioning of library scopes and thus – in contrast to CoDeSys V2.3 – a more trouble-free interaction of libraries of different derivations. Each project or library referencing a target-dependent library does not directly use this target-dependent library but a corresponding **template library** defining the same interfaces but not containing an implementation. This allows a syntactical check. At the same time a library placeholder is defined. When the new library is used for a definite target by the user, the resolution of the placeholder will be done according to an appropriate definition in the target description and thus a definite version of the underlying library will be found.

The name of such a kind of Library should extended by the term “Template”.

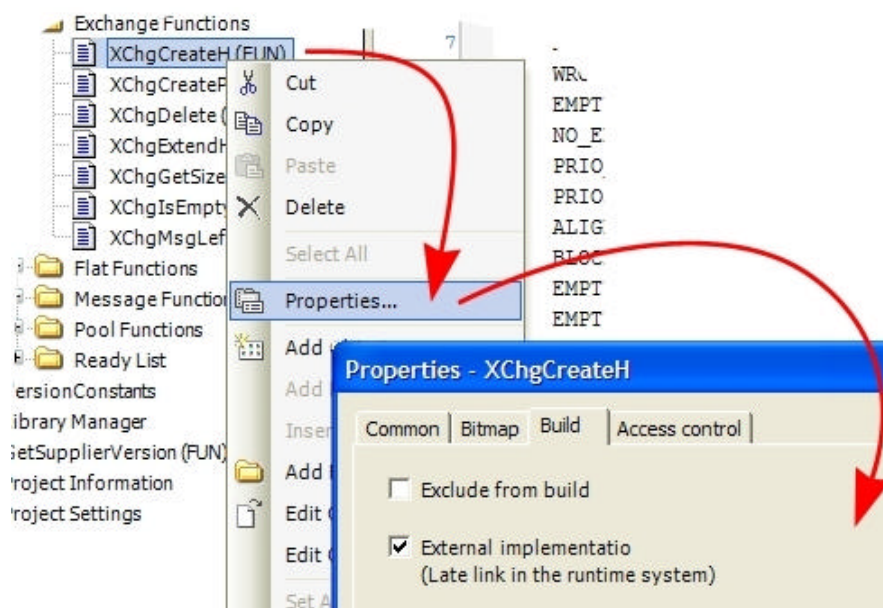
Procedure: When inserting a library use tabulator „Placeholder“ and define a placeholder. For correct placeholder names see document „Prefix.pdf“. Then in „Default Library“ select a **template library**.



1.4 External Libraries and Placeholder Concept

A library which is implemented as a part of the runtime system (in ANSI C or C++) is called an external library. So each external library is a target-specific library and the placeholder concept should be used to handle the target dependencies. Each project or library referencing an external library does not directly use this external library but a corresponding special **template library** defining the same interfaces but not containing an implementation. For all functions and function blocks the build property "External implementation" should be set.

The name of such a kind of Library should be extended by the term "External".



1.5 Placeholder Concept and Device Description

Each placeholder must be resolved in the target description: In section „placeholder-libraries“ there must be an entry „library-management“, where for each placeholder a description must be available.

```
<ts:section name="library-management">
  <ts:section name="placeholder-libraries">
    <ts:setting name="IoStandard" type="string" access="visible">
      <ts:value>IoStandard, 3.1.1.0 (System)</ts:value>
    </ts:setting>
    <ts:setting name="CAA Types" type="string" access="visible">
      <ts:value>CAA Types (PLCWinRTE V3.x), 3.0.0.4 (3S - Smart Software Solutions GmbH)</ts:value>
    </ts:setting>
    <ts:setting name="CAA Tick" type="string" access="visible">
      <ts:value>CAA Tick (PLCWinRTE V3.x), 3.0.0.1 (3S - Smart Software Solutions GmbH)</ts:value>
    </ts:setting>
    <ts:setting name="CAA CanL2I" type="string" access="visible">
      <ts:value>CAA Can Low Level Imp (PLCWinRTE V3.x), 3.0.0.2 (3S - Smart Software Solutions GmbH)</ts:value>
    </ts:setting>
    <ts:setting name="CAA CanL2" type="string" access="visible">
      <ts:value>CAA Can Low Level (PLCWinRTE V3.x), 3.0.0.2 (3S - Smart Software Solutions GmbH)</ts:value>
    </ts:setting>
  </ts:section>
</ts:section>
```

1.6 Library Reference Summary

The rules for referencing other libraries are described below:

- A library without any implementation is called an interface library. This kind of libraries can be extended in a code compatible way only.
- An interface library is referenced **without** a placeholder but with a **newest** constrain.
- All other libraries are referenced with a placeholder. The version constraint of the default lib is **newest**.



1.7 Version History

The comment in front of the declaration part of the function „GetSupplierVersion“ is the place to enter the version history.

2 Requirement Specification

The Requirement Specification which should serve as a guide for a workgroup when issuing a library specification, and which is to be created in the workshop, should define the following items:

- Target: Which **task** should be worked out.
- **Terms** and **Definitions**
- **Models** (state transitions etc.)
- List of desired **Functionality**
- **Constraints** (standards, patents, norm-specific notation, data types, word widths etc.)
- **Dependencies** (other libraries, runtime system classes, language scope (CoDeSys Version))
- **Library name** and **-prefix**

3 Interfaces

3.1 General

The **file name** of a library which has been defined in the CAA Technical Workshop should be “**CAA_**” followed by a short expressive name.

For each library name a prefix is to be used, consisting of a few capital letters, the **library prefix**. On the CAA-homepage a table is published, containing all library prefixes known to the CAA. CAA members can publish their proprietary library prefixes there. In CoDeSys V3.x libraries the library prefix is used as default namespace.

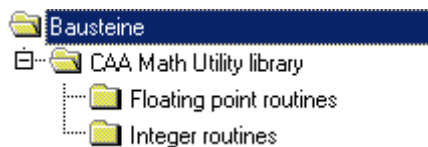
The prefix – without underscore – must not consist of a word reserved by the IEC standard. Thus for example the following would not be allowed: COS_, DT_, ABS_, etc.

The prefix – without underscore – must not consist of the name of an already defined data type (ALIAS, ENUM, STRUCT,...). Thus for example it is not allowed to use prefix „TICK_“ for library CAA_TICK.lib, because in library CAA_Types.lib already an alias “TICK” is defined. If this rule is not observed, compile errors will result in CoDeSys 3.x when applying the other prefix-/namespace rules.

All library objects are to be organized in folders. On top level always use the complete library name, subfolders should be established.

In each folder there might exist an additional folder with name „**internal**“, which might contain internal auxiliary functions and function blocks only used within the library.

Example:



Comments are to be done in **English**.

The specifications should comply with IEC61131-3. As few as possible CAA-specific constructs should be used.

When **using pointers**, take care that a pointer with value 0 never will cause a system crash but at most an error message.

3.2 Variable names

The naming of variables preferably should be unique in all libraries, following the **Hungarian notation**:

For each variable a meaningful, preferably short, English name should be used, the **base name**. Always the first letter of a word of the base name is to be written uppercase, the remaining letters lowercase. (Example: FileSize). If required, additionally a translation file for other languages can be created.

The base name is to be **prefixed** (lowercase) corresponding to the data type of the variable. See the following table:

Data type	Lower limit	Upper limit	Length	Prefix	Flag	Note
BOOL	FALSE	TRUE	1/8 Bit	x ¹	0x01	
				b		reserved
				n		reserved
				f		reserved
BYTE		Bitstring, not for arithm. operations	8 Bit	by	0x02	
WORD		Bitstring, not for arithm. operations	16 Bit	w	0x03	
DWORD		Bitstring, not for arithm. operations	32 Bit	dw	0x04	
LWORD		Bitstring, not for arithm. operations	64 Bit	lw	-	
SINT	-128	127	8 Bit	si	0x06	
USINT	0	255	8 Bit	usi	0x07	
INT	-32.768	32.767	16 Bit	i	0x08	
UINT	0	65.535	16 Bit	ui	0x09	
DINT	-2.147.483.648	2.147.483.647	32 Bit	di	0x0A	
UDINT	0	4.294.967.295	32 Bit	udi	0x0B	
LINT	-2 ⁶³	2 ⁶³ - 1	64 Bit	li	-	
ULINT	0	2 ⁶⁴ - 1	64 Bit	uli	-	
REAL		Single-precision IEEE 754	32 Bit	r	0x0C	
LREAL		Double-precision IEEE 754	64 Bit	lr	-	
STRING		8 Bit per character		s	-	
WSTRING		16 Bit per character		ws	-	
TIME	T#0	T#49d17h2m37s295ms	32 Bit	t	0x0D	ms
LTIME	T#0	T#213503d23h34m33s709ms551us615ns	64 Bit	lt	-	ns
TIME_OF_DAY	TOD#0:0:0.0	TOD#23:59:59.999	32 Bit	tod	0x0E	ms
DATETIME	DT#1970-1-1-0:0:0	DT#2106-2-7-6:28:15	32 Bit	dt	0x0F	s
DATE	D#1970-1-1	D#2106-2-7	32 Bit	dat	0x11	d
ENUM			16 Bit	e	0x12	
INTERFACE		I<Interface>		I		
Interface reference		itf<instance>		itf		
REFERENCE		r<Prefix><Variable>		r		

¹ For BOOLEAN variables deliberately „x“ is chosen as prefix in order to get a distinction against BYTE and to meet the viewpoint of an IEC programmer (compare addressing %IX0.0).

Examples:

```
bySubIndex: BYTE;
sFileName: STRING;
udiCounter: UDINT;
```

Please keep in mind that further base data types are defined in the library **CAA_Types**. These are data types depending on the platform properties (usually on the size of the natural processor word).

In case of **nested declarations** the prefixes are concatenated according to the sequence of their declarations.:

Type	Lower limit	Upper limit	Memory space	Prefix	Flag	Note
POINTER				p	0x13	
ARRAY				a	0x14	

Example:

```
pabyTelegramData: POINTER TO ARRAY [0..7] OF BYTE;
```

Function block instances and **variables of user-defined data types** as a prefix have to get a short identifier for the FB- resp. data type name (Example: sdo).

Example:

```
sdoReceivedTelegram: CAN_SDOTelegram;

TYPE CAN_SDOTelegram :      (* prefix: sdo *)
STRUCT
    wIndex:WORD;
    bySubIndex:BYTE;
    byLen:BYTE;
    aby: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE
```

Local Constants (c) have to start with the constant-prefix c and an additional underscore, followed by the type-prefix and the variable name.

Example:

```
VAR CONSTANT
    c_uiSyncID: UINT := 16#80;
END_VAR
```

For **Global Variables** (g) and **Global Constants** (gc) an additional prefix+underscore is to be appended:

Examples:

```
VAR_GLOBAL
```

```

    CAN_g_iTest: INT;

END_VAR

VAR_GLOBAL CONSTANT

    CAN_gc_dwExample: DWORD;

END_VAR

```

3.2.1 Variable names in CoDeSys V3.x libraries

The composition of variable names in CoDeSys V3.x corresponds to the description in chapter 3.2 except that global variables and constants do not need a library prefix, because those function will be replaced by the library namespace.

Example:

```

g_iTest: INT;           (declaration)

CAN.g_iTest             (usage, call in program)

```

3.3 User defined data types (DUT)

The name of each **structure data type** should consist of the library prefix (e.g. CAN), an underscore and a short meaningful description (e.g.: SDOTelegram) of the structure. The corresponding prefix for defined variables of this structure should follow as a comment directly behind the colon.

Example:

```

TYPE CAN_SDOTelegram :      (* prefix: sdo *)
STRUCT
    wIndex:WORD;
    bySubIndex:BYTE;
    byLen:BYTE;
    abyData: ARRAY [0..3] OF BYTE;
END_STRUCT
END_TYPE

```

Enumeration values should start with the library prefix (e.g.: CAL), followed by an underscore and the identifier in capital letters.

Regard that in previous CoDeSys versions ENUM values > 16#7FFF caused errors, because they did not automatically get converted into INT. For this reason ENUMs always should be defined with correct INT values.

Example:

```

TYPE CAL_Day : (
    CAL_MONDAY,
    CAL_TUESDAY,
    CAL_WEDNESDAY,
    CAL_THURSDAY,
    CAL_FRIDAY,
    CAL_SATURDAY,
    CAL_SUNDAY);

```

Declaration:

```

eToday: CAL_Day;

```

3.3.1 DUTs in CoDeSys V3.x libraries

In case of DUT names in CoDeSys V3.x libraries do not use a prefix because its function is taken by the library namespace. For the exact use of the namespace concept there is a need for a so called attribute. In front of the definition of a DUT the "{attribute 'qualified_only'}" phrase should be used.

Also the enumeration values are defined without library prefix:

Example (from a library with namespace CAL):

```
{attribute 'qualified_only'}
TYPE Day : (
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY);
```

Declaration:

```
eToday: CAL.Day;
```

Usage in application:

```
IF eToday = CAL.Day.MONDAY THEN
```

3.4 Functions, function blocks, programs (POU)

Identifiers of functions, function blocks and programs should be composed of the library prefix (e.g.: CAN), an underscore and a meaningful, preferably short name of the POU (e.g.: SendTelegram). Like for variables always the first letter of a word of the base name should be written uppercase, the remaining ones lowercase. It is recommended to compose the name of the POU of a verb and a substantive.

Example:

```
FUNCTION_BLOCK CAN_SendTelegram (* prefix: canst *)
```

In the declaration part a **short description** of the function block should be added as a comment. Also all **inputs** and **outputs** should be supplemented with **comments**. In case of function blocks the prefix for defined instances should be inserted as a comment directly behind the name.

Actions basically get no prefix; only the identifiers of actions which are only called internally, i.e. by the pou itself, start with „prv_“. Libraries which might be implemented as external libraries must not contain actions except for those identified by „prv_“. Actions cannot be implemented in ANSI-C for CoDeSys 2.3.

Each **function** – for the reason of compatibility with previous CoDeSys versions - must have at least one parameter. **External functions** must not use structures as return values.

3.4.1 POUs in CoDeSys V3.x libraries

For POU names in CoDeSys V3.x libraries the library prefix is not needed because its function is taken by the library namespace.

For the naming of **methods** the same rules apply as for actions. Possible inputs of methods should be provided with English comments. Also a short description of the method should be contained added to the declaration. For methods there are no restrictions concerning the ability to get implemented in external or internal libraries.

Interface names should start with letter „I“, e.g.:

```
ICANDevice
```

3.5 Visibility of project items in CoDeSys V3.x libraries

Sometimes there is a need for some helper functions or variables. These items should be not members of the input assistant lists and the intellisense lists of CoDeSys V3.x. Regarding this requirement there is a need for another so called attribute. In front of the definition of such items the "{attribute 'hide'}" phrase should be used.

3.6 Version POU

Each library should have a function with exactly one BOOLEAN dummy input parameter xDummy with name **<libraryprefix>_GetSupplierVersion**, returning a WORD describing the **implementation version**. This version information is defined by the vendor (recommendation: higher-value byte: release version²; lower-value byte: implementation version³).

Example:

Function return value: 522 = 16#020A

Release version number: 2

Implementation version number: 10

The **specification version number**⁴ is set up in the library as a global constant of type BYTE with name⁵

```
<libraryprefix>_gc_byVersion: BYTE;
```

Further on a global constant of type BYTE with name⁵

```
<libraryprefix>_gc_Version_<specification version>: BYTE;
```

is to be defined. This constant can be used for creating compile errors in case a library with wrong version is used. For CoDeSys V 3.x the library prefix is not needed because the namespace takes its function.

Explanation:

The interface will change only if the specification version gets changed. Thus it can be defined as a global constant. The implementation version however usually changes without any change of the interface. Due to the fact that the implementation can be done as an external library in the runtime system, the implementation version must be retrieved via a function call.

3.7 CoDeSys Project

Together with each library specification a CoDeSys library file is to be published, containing all POUs (as pure interface definitions, empty implementation part), data structures, enumerations and global variables.

Used libraries should be included in the interface project, so that they get bound automatically as soon as the library gets included in a project.

² Release version; a tested and released version should get a release-version number by the vendor.

³ Each source change should be followed by a change of the implementation version number

⁴ Version of the library specification; defined by the CAA-Workshop

⁵ In case of CoDeSys V3.x-libraries no library prefix + underscore is used

3.8 Behaviour- and interface model of function POU

Actions whose execution time depends on external devices (e.g. reading from the serial interface) or is strongly influenced by the size of parameters (e.g. 64kByte CRC), should be defined as function blocks (FB) with asynchronous processing, i.e. distributed on several cycles.

2 models for start behaviour:

1. **xExecute** edge controlled

Rising edge: action start

At a rising edge the POU takes action. In standard case the inputs get copied locally, whereby a modification of the inputs during action has no effect. Any different behaviour of the inputs must be documented.

Falling edge: reset of the outputs

If the falling edge occurs before the POU has terminated its action, the outputs will work as usual and will not be reset until either the action is terminated or aborted (xAbort), or an error occurs. In this case the appropriate values (xDone, xError, eError) must be available at the outputs for exactly one cycle.

Optional input **xAbort**: stops the action

If this input is TRUE, the action will be stopped immediately and all outputs will be set to the initial values. In case of abort the xAborted output is set to true after abort is done.

Optional input **udiTimeLimit**: time after which the active FB should return

This input shows how long (μ s) a FB (whose total processing takes a considerable time) may work per call before returning. Value 0 indicates that there is no time limit.

Optional input **udiTimeOut**: time after which the active FB should abort with an error

This input indicates after which time (μ s) – measured from action start - a FB (e.g. waiting for an external acknowledgement) will abort the execution with a timeout error.

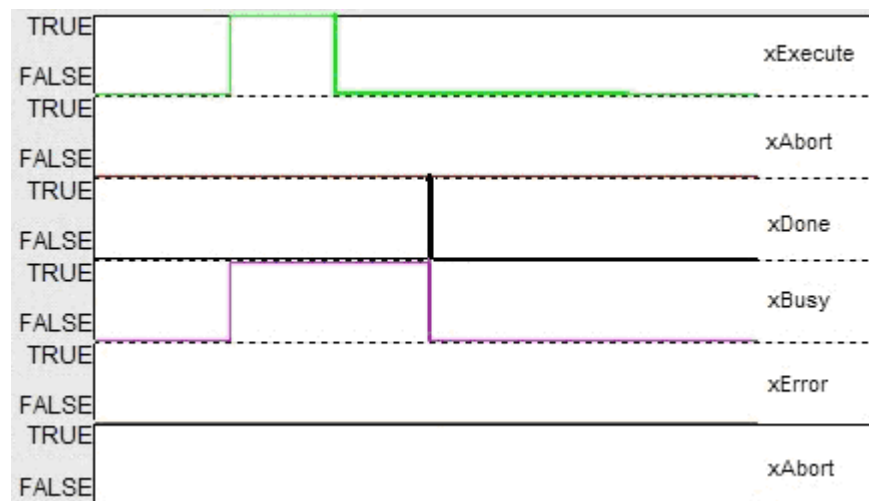
These inputs, if used, always are to be declared first, namely in the above shown order from up to down.

The following model cases more detailed specify the functionality of the inputs and outputs:

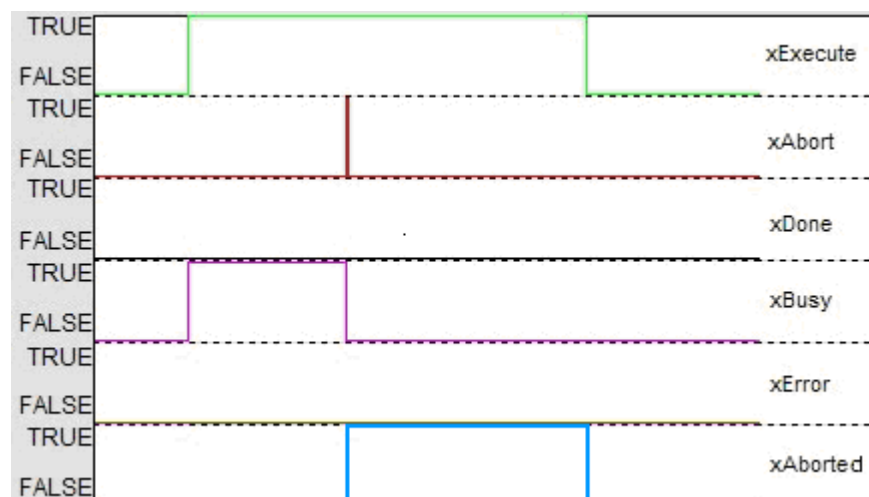
Error-/Abort-free execution:



Error-/Abort-free execution with falling edge of xExecute while busy:

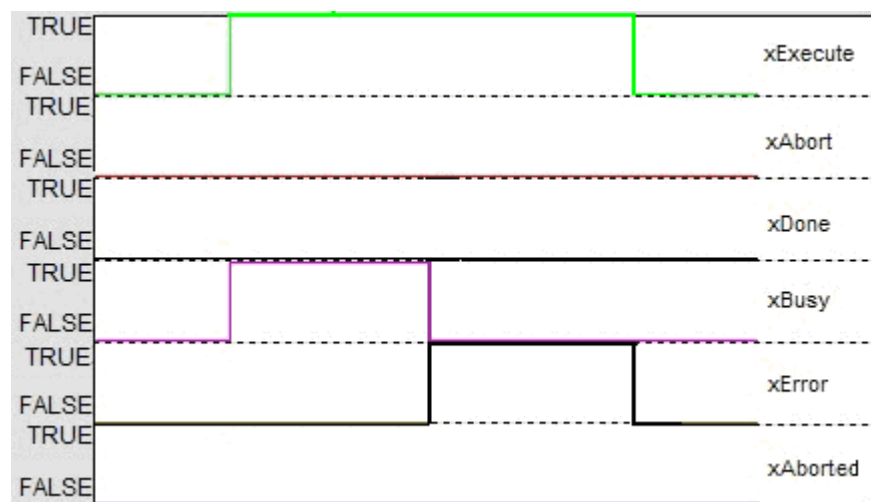


Abort:

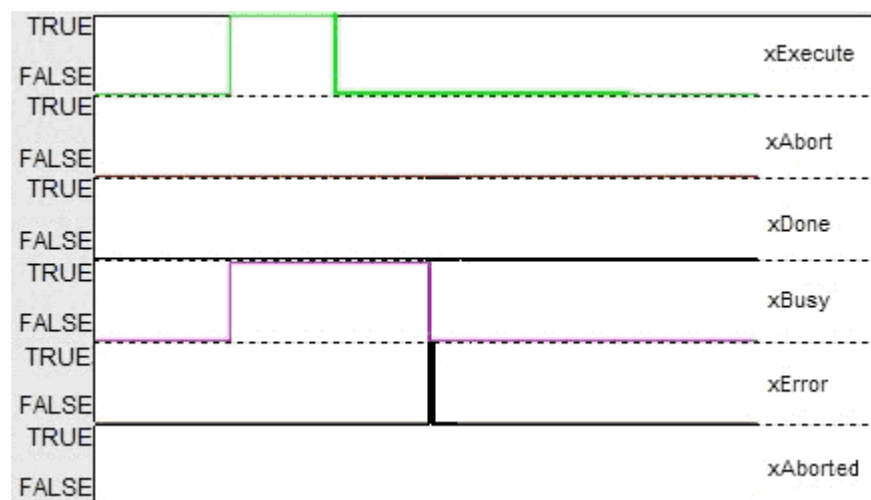




Error case:



Error case with falling edge of xExecute while busy:



2. **xEnable** level-controlled

TRUE: Action running

Per default any changes at the inputs effect the current action. Any different behaviour must be documented.

FALSE: Action stopped, Outputs xDone, xBusy, xError, eError are reset

Predefined outputs:

xDone: Action terminated successfully

xBusy: function block is active

xError: an Error occurred

These outputs, if used, always are to be declared first, namely xDone before xBusy, if applicable followed by the error outputs xError and eError.

POUs may use these inputs and outputs only in connection with the above described function.

In the **descriptions of the libraries** the functionality of the above mentioned default inputs and outputs is not to be described in order to avoid multiple specifications.

An **example project „BehaviourModel.pro“**, containing function blocks with the above described behaviour, will be published together with the current version of this guidelines.

For CoDeSys V3.x there is a library **CAA_BehaviourModel.library**, uniquely implementing the behaviour model. Also there is the project template **CAA_Template.library** and **CAA_Template_Extern.library**, extending the behaviour model for the particular usage. All three libraries are published with the current guideline document and are part of the CoDeSys V3.x setup.

CAA_BehaviourModel.library FBs:

ETrig: edge triggered

ETrigA: edge triggered with abort input

ETrigTo: edge triggered with time out

ETrigTI: edge triggered with time limit

ETrigToA: edge triggered with time out and abort input

ETrigTIA: edge triggered with time limit and abort input

ETrigToTI: edge triggered with time out and time limit

ETrigToTIA: edge triggered with time out, time limit and abort input

LTrig: level triggered

3.9 Error handling

For each library all error IDs created by the library modules – not by instantiated modules of referenced libraries – are to be specified and listed. For this purpose each library should define an enumeration **<libraryprefix>_ERROR**. Components of this enumeration are “<libraryprefix>_<error description>”. In case of CoDeSys V3.x libraries the prefix is not to be used, because its functionality is taken by the library namespace.

Via the library prefix also error ranges are to be assigned to a library. These error ranges are divided up into a default range (first part) and a vendor-specific range (second part). The ranges are limited by the following enumeration values defined in each library: **<libraryprefix>_FIRST_ERROR**, **<libraryprefix>_FIRST_MF** and **<libraryprefix>_LAST_ERROR**. These limit values do not describe errors. They are allocating values within the enumeration which thereby are not any longer available as error numbers. Between <libraryprefix>_FIRST_ERROR and <libraryprefix>_FIRST_MF the default errors are to be defined, between <libraryprefix>_FIRST_MF and <libraryprefix>_LAST_ERROR the vendor-specific errors.

Additionally each error enumeration should contain **<libraryprefix>_NO_ERROR** with value 0.

Each library using function blocks with input **udiTimeOut** in the error enumeration additionally should contain a symbol named **<libraryprefix>_TIME_OUT** with value **<libraryprefix>_FIRST_ERROR+1**. Other libraries must not allocate this value.

If required, manufacturer-specific errors and errors of modules of subjacent layers can be reported by a collective error number and be diagnosed in detail by another error module.

3.9.1 Function Blocks

Each function block should have the following outputs:

xError (BOOL): FALSE: no error; TRUE: error occurred, module aborts action.

eError (<libraryprefix>_ERROR): Library-local error-ID (0: no error).

If default inputs (xExecute/xEnable, xAbort, udiTimeLimit, udiTimeOut) are used, these must be positioned at top of the list of inputs, followed by specific inputs and VAR_IN_OUTs. Further on the default outputs are to be declared in the following sequence: xDone, xBusy, xError, eError. Subsequently the specific outputs are to be defined.

3.9.2 Functions

Not each function returns information on an error occurrence (e.g. min, sin etc.). If the return value of the function should contain information on an error, the following cases are to be distinguished:

1. only the error is returned:
0 means success, another number describes the error-ID.
2. the return value besides the error itself also provides further information:

The function has an input variable **peError** (POINTER TO <libraryprefix>_ERROR), which should contain a pointer on a defined error variable. In error case the called function allocates this variable with the corresponding error value. If the user is not interested in the error number, he will pass a CAA_gc_pNULL here.

3.10 Lists and arrays

It is recommended to define the first element of a list or of a array as "1". So any function returning an index can indicate an error by return value „0“ (according to 3.9).

3.11 GetProperty function

Mainly in case of external libraries describing an interface which gets implemented on multiple systems with different hardware components, it often happens that e.g. platform A supports a certain function, not however platform B. In order to be able to use the same interface for both implementations, the application resp. the layers basing on this interface must have the possibility to poll the properties/abilities of the implementation and to react correspondingly.

For this purpose the interface libraries should have a function **<libraryprefix>_GetProperty**, getting the number of the property to be scanned as a WORD input and returning the property as a DWORD. For CoDeSys V3.x libraries the library prefix is not to be used, because its functionality is taken by the library namespace.

The property numbers and the meaning of the return value are defined in the library specification.. If any property number is scanned which is not known to the implementation, 0 will be returned..

Property numbers as from 16#8001 can be allocated implementation-specifically; regard however that applications basing on those numbers might be not portable. In order nevertheless to enable the application to react appropriately, the property number 16#8000 is reserved for the identification of the implementation (e.g. "Vendor Platform name"), which is returned as POINTER TO STRING in the DWORD. Thus the application can check whether the implementation is the expected one and thus whether the implementation-specific property numbers are available and of meaning. Implementations not using specific property numbers return 0 on 16#8000.

No enumeration should be defined for the property numbers⁶. Instead of it is recommended to define global constants.

⁶ Explanation: Implementation-specific properties can be defined, which of course cannot be part of an enumeration available for all implementations.

4 **Function description and examples**

For each library a detailed **function description** should be issued. It should describe the **inputs** and **outputs** of each particular POU as well as its function (perhaps also via interesting sample cases).

When documenting POUs of type "Function", a reasonable exemplary variable name is to be used for the return variable, because the name of the return variable equates the function name.

Data structures, enumerations and global **variables** are to be documented as well.

All **hardware-** and **system resources**, which can be allocated by library modules, must be listed in the documentation.

Besides the description of the library the document also should contain the following:

- Participants in creating the library (working group)
- Dependencies (used libraries)
- Change history
- perhaps restrictions etc.

If possible, the library description should be rounded off by a comprehensive, documented **example**, using as many as possible library modules and illustrating the handling of the library to the user. Available examples will be published as CoDeSys projects together with library specification.

3S provides a MS Word **Template** (CAA_lib.dot) usable for the library documentation.

5 **Help**

For each library a help file in chm-format should be created, describing the interfaces and function of the library.

3S provides an example.

6 Test- and certification projects

Test- and certification projects are to be created as CoDeSys projects and to be documented. These can be used to check whether the libraries work properly.

The test- and certification projects will be published together with the library.

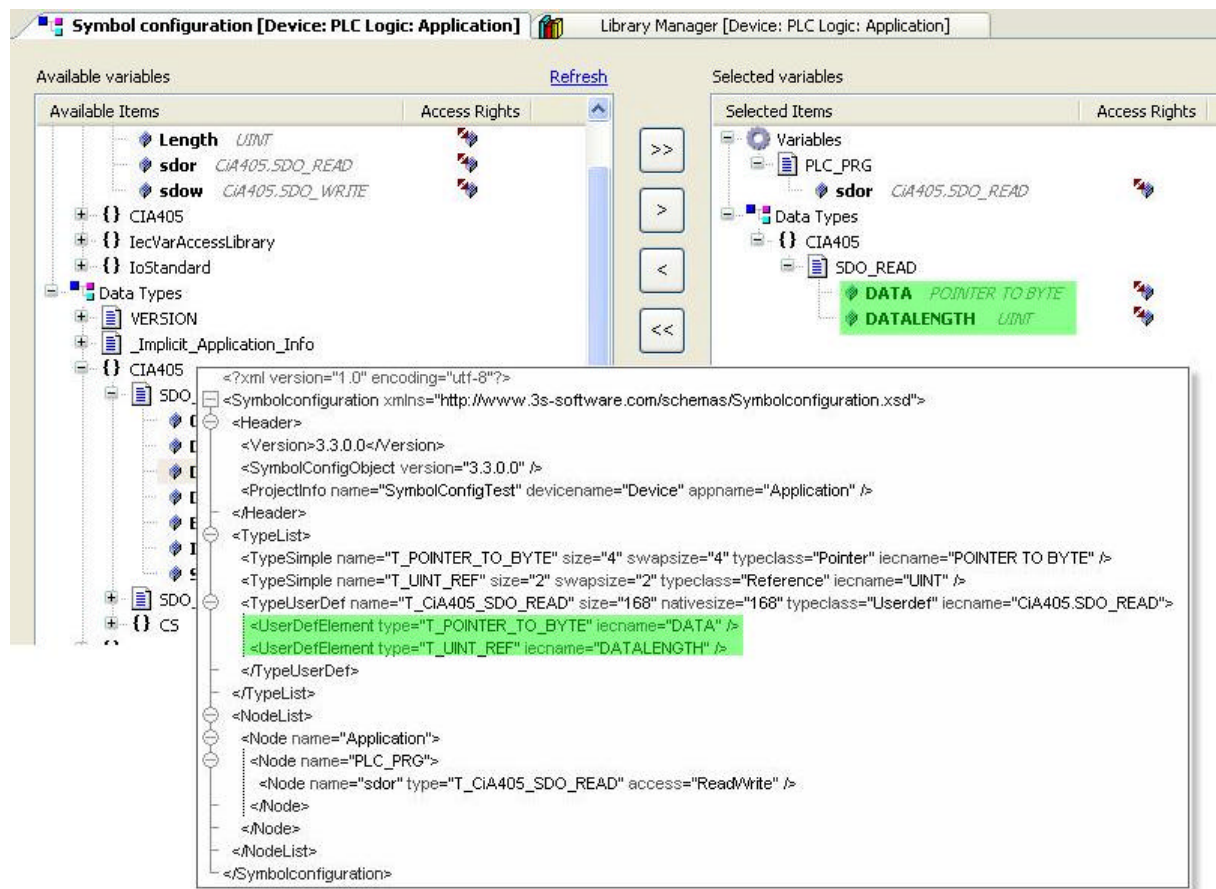
7 Release

A release of the library specification is not possible until a complete implementation is available.

Appendix A: Symbol Configuration

A.1 Filtering of symbols inside the Symbol Configuration

The symbol configuration editor provides two tools to define a set of symbols which will be exported. In a first step it is possible to define a set of variables / instances below the **Variables** node of the configurator. These Instances will be broken down to its member variables. Each single variable will be accessible from an external system. To optimize the size of this variable set it is often helpful to constrain the content of the exported variable set. This can be achieved by selecting some members of an data type below the **Data Types** node in the configurator. By such a selection a reduced data set will be generated. For all instances of the selected data type only the selected members will be exported.



Limit the number of exported items by using the Data Types section

A.2 Limit the number of symbols by proper library design

The number of symbols exported from a library has a major impact on the usability of the symbol configuration. It is in the responsibility of the developer of an library to limit the number of symbols which are exported by this library. By applying the following rules, it is possible to develop a library which only exports the really required symbols.

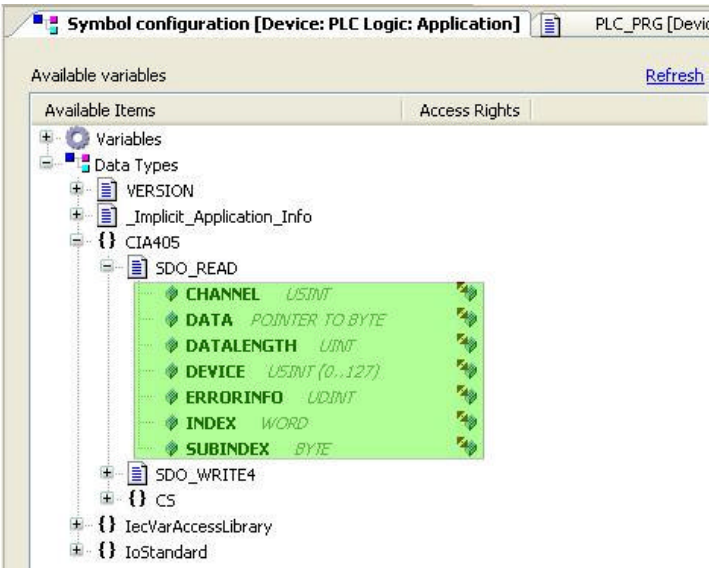
- Reduce the amount of visible symbols. Use the hide attributes where ever possible. For example: {attribute 'hide'} or {attribute 'hide_all_locals'}
- Constant values should be declared as `constant`. This type of symbols will never be exported.

VAR_GLOBAL CONSTANT

```
CANOPEN_KERNEL_NO_ERROR : WORD := 16#0;
CANOPEN_KERNEL_OTHER_ERROR : WORD := 16#1;
CANOPEN_KERNEL_DATA_OVERFLOW : WORD := 16#2;
CANOPEN_KERNEL_TIMEOUT : WORD := 16#3;
```

- Reduce the amount of parameters inside the public interface of a function block. Create lightweight interfaces.
- Use data structures for grouping related parameters. Stand to benefit from the power of structured data types like `struct`, `array` and `enum` and the composite of them.
- Create a hierarchical set of various well defined small data structures instead of using few monolithic huge ones.

```
{attribute 'hide_all_locals'}
FUNCTION_BLOCK SDO_READ
VAR_INPUT
    DEVICE : DEVICE;
    CHANNEL : USINT := 1;
    INDEX : WORD;
    SUBINDEX : BYTE;
    DATA : POINTER TO BYTE;
END_VAR
VAR_IN_OUT
    DATALENGTH : UINT;
END_VAR
VAR_OUTPUT
    ERRORINFO : SDO_ERROR;
END_VAR
VAR
    SDO : CS.SDO_READ;
    StateSDO : CS.SDO_READ;
    usiInternalState : USINT;
    eProcStateGetter : CS.PROC_STATE;
END_VAR
```



The screenshot shows the 'Symbol configuration' window for a PLC Logic Application. It displays a tree of available variables. Under the 'CIA405' section, the 'SDO_READ' function block is expanded, showing its parameters: CHANNEL (USINT), DATA (POINTER TO BYTE), DATALENGTH (UINT), DEVICE (USINT (0..127)), ERRORINFO (UDINT), INDEX (WORD), and SUBINDEX (BYTE). Other visible items include SDO_WRITE4, CS, IecVarAccessLibrary, and IoStandard.

Example for {attribute 'hide_all_locals'}

Change history

V 0.1	Hilmar Panzer, 3S	12.05.2004	Draft for presentation at 2. Technical Workshop
V 0.2	Hilmar Panzer, 3S	14.05.2004	Changes after 2. Technical Workshop
V 1.0	Hilmar Panzer, 3S	08.07.2004	Changes after 3. Technical Workshop; Ratification
V 1.1	Hilmar Panzer, 3S	09.09.2004	Changes after 4. Technical Workshop
V 1.2	Hilmar Panzer, 3S	25.01.2005	Changes after 5. Technical Workshop
V 1.30	Wolfgang Doll, 3S	17.02.2005	Changes after 6. Technical Workshop
V 1.31	Hilmar Panzer, 3S	24.02.2005	Supplement: local/global constants, global variables
V 1.32	Hilmar Panzer, 3S	20.05.2005	CAA_Type.lib; <prefix>_GetProperty function
V 1.33	Hilmar Panzer, 3S	30.06.2005	Spelling corrections
V 1.34	Wolfgang Doll, 3S	06.07.2005	Prefix data structure/FB without Lib-prefix; CoDeSys-Library instead of -Project; Formatting
V1.35	Hilmar Panzer, 3S	27.09.2005	Documentation of function return names; implementation samples behaviour model; ENUM values > 16#7FFF; at least one parameter for functions; error returns of functions
V1.36	Hilmar Panzer, 3S	30.09.2005	Behaviour model sample project; external functions must not return structures
V1.37	Hilmar Panzer, 3S	11.01.2006	Spelling corrections
V1.38	Hilmar Panzer, 3S	25.01.2006	Documentation of allocated resources
V1.40	Hilmar Panzer, 3S	10.04.2006	Changes after 12. Workshop: behaviour model, error numbers
V1.41	Hilmar Panzer, 3S	30.05.2006	Changes after 13. Workshop (see protocol)
V1.42	Wolfgang Doll, Hilmar Panzer, 3S	22.08.2006	Changes after 14. Workshop: Library prefix and namespaces
V1.43	Wolfgang Doll, 3S	07.03.2007	Changes after 15. Workshop: CAA_Template, CAA_CmdModel, ...
V1.44	Wolfgang Doll, 3S	04.06.2007	Changes after 16. Workshop: Library Placeholder
	Margit Neuss, Wolfgang Doll, 3S	11.06.2007	English version issued, basing to German V1.44.
V1.45	Wolfgang Doll, 3S	02.08.2007	Description of additional properties
V1.46	Wolfgang Doll, 3S	16.08.2007	Rename Commanding Model to Behaviour Model, Description "{attribute 'qualified_only'}" Description "{attribute 'hide'}"
V1.47	Alexandra Müller, 3S	21.04.2008	Behaviour Model Documentation
V1.48	Alexandra Müller, 3S	19.06.2008	Behaviour Model Documentation updated; added xAborted output and updated execution graphs
V1.49	Wolfgang Doll, 3S	14.05.2009	Prefixes for Interfaces, Instance of an Interfaces and References
V1.50	Wolfgang Doll, 3S	09.02.2010	Appendix A: Library design and symbol configuration
V1.51	Wolfgang Doll, 3S	19.03.2010	Correction of some typos
V1.52	Wolfgang Doll, 3S	18.05.2011	LanguageModelAttribute clarification

