Generating libraries in CoDeSys V3

Document Version 2.0

CONTENT

1 Introduction

The concept of libraries in CoDeSys V3 shows major differences to that of CoDeSys V2. Hence, creating and making use of libraries you have to pay regard to the following items being characteristic for libraries in V3:

Each library is identified uniquely by its name, its version and its manufacturer.

Each library has its own namespace by use of which the library elements (modules and variables) may be accessed.

The library version is composed by 4 digits, where each digit has its own specific meaning (see Chapter 2.3).

Within CoDeSys libraries are stored in a data base, the so called library repository.

For a clear arrangement each library may be assigned to a category (see Chapter 2.1).

A library being based on another library may include this one as so called child library. In this case loading the library will always invoke the loading of the child library as well.

The version of a library to be included can be specified in different ways (see Chapter 2.2).

Libraries may contain functions or modules, which have been implemented in ANSI-C/C++ within the runtime system (so called *„external libraries“*). See Chapter 2.5.

The so called implicit Libraries will be included automatically by CoDeSys in order to generate code (e.g. visualization, symbol configuration or IO driver in IEC). These are highlighted in gray within the library repository and may not be modified. See Chapter 2.6.

Each library released can be marked internally by a release flag. The consequence is, that trying to modify such a library will provoke a warning message. See Chapter 2.8.

Be careful in separating the libraries defining interfaces or defining their implementation. See Chapter 2.7.

2 Library concept in detail

Subsequently the main aspects in creating and employing libraries in CoDeSys V3 are discussed in detail.

2.1 Library categories

For a clear arrangement each library can be assigned to a specific *category*. Thereby each user may create its own categories.

Currently there are the following categories:

2.1.1 System

Libraries necessary for the general handling of the components of the runtime system; these libraries will be included automatically.

2.1.2 Internal

Libraries required by specific plug-ins or device descriptions, which will be included automatically.

2.1.3 Application

Libraries useful for the programmer, which may be included in his project according to his requirements.

2.1.4 Target

Category that groups specific manufacturer and target system dependent libraries.

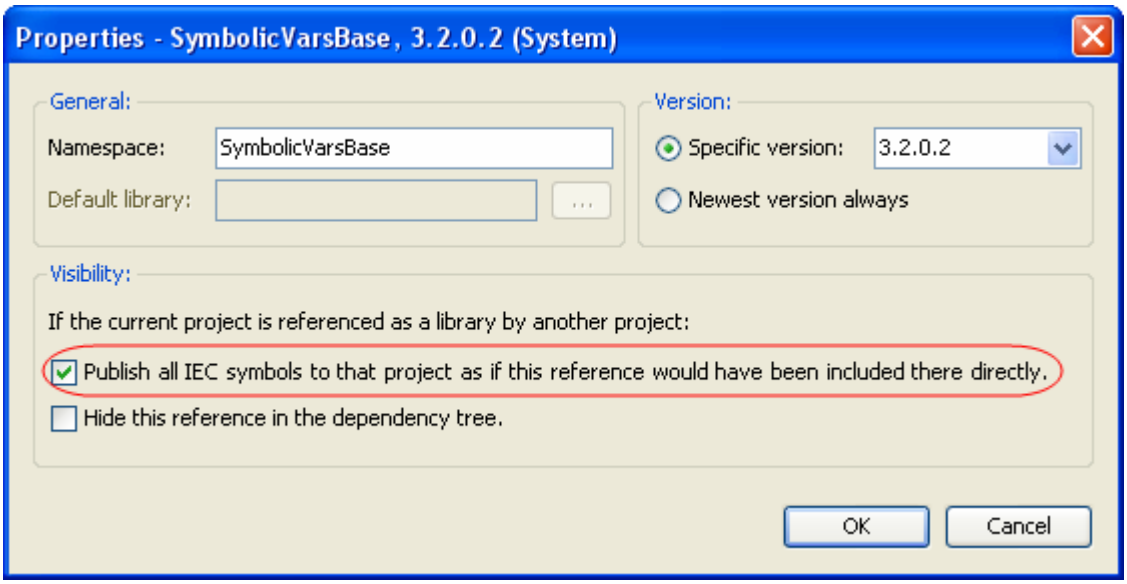
2.2 Namespaces

Within the library manager each top level library may be assigned to a namespace. The default namespace will be adopted from the property 'DefaultNamespace' of the library. In case that property has not been set, the default namespace will be derived from the name of the company and the library itself.

Namespaces must fulfill the standard for IEC identifier. Therefore all characters banned from being part of an IEC identifier will be replaced by an underscore '_' . Furthermore, an underscore will be prefixed to the namespace, if this one will begin with a number. In case the compiler cannot resolve a symbol uniquely, it will prompt an error. Thereby uniqueness is specified by uniqueness of the access path <Namespace>.<Symbol>. Library namespaces may be multi-level, i.e. in case of nested libraries the "lowest" library can be accessed by nesting the namespaces.

Example:
Assume that library Test with namespace Test refers to library Test2 with namespace Test2. Both of them contain the function aFUN. Within a project the use of the symbol aFUN will cause an error message only if the library Test makes also the symbols of Test2 public.

This is done by the following setting within the properties of the library reference of Test2:



If this setting has been activated, using the symbol aFUN without indicating the namespace will cause an error message, as the symbol cannot be resolved uniquely. The function aFUN of Test has to be called by Test.aFUN. The function aFUN of Test2 may be either by Test.Test2.aFUN or by Test2.aFUN, as the namespace Test2 is also part of the published IEC symbols.

Even if the library symbols are not published by the higher level library, the symbols of Test2 can be accessed by nesting the namespaces.

The namespaces of the referenced libraries may only be modified within the libraries making use of them, but not within the project. Within the project only the namespaces of libraries that are included on top level can be modified.

2.3 Versioning of libraries

Basically each library gets assigned to an unique version number during its release. The strategy for assigning the version numbers is the following:

A **compatible modification** in terms of a supplement or bug fix is reflected by incrementing the last number (patch version) of the version identification.

An **incompatible modification** in terms of an enlargement is reflected by incrementing the second last number (service pack version) of the version identification.

An incompatible modification in terms of modifying the interface will be reflected by incrementing the second number (sub version) of the version identification.

This results in:
V3. <sub version>. <service pack version>. <patch version>

2.4 Including a library version

A library is identified uniquely by its name, its version and an optional namespace. There are several possibilities to influence the library version to be included (so called „Version Constraints“).

The version of a library can be confined resp. extended by the following constraints:

2.4.1 Specific Version

Only the specified version of a library, e.g. 3.2.0.1, will be loaded, even if older or newer versions of this library are available. Therefore, the loaded project has not to be compiled again, as the libraries rest the same.

2.4.2 Newest Version always

The newest version available of a library will be loaded.

If, for example, the versions 3.1.3.1, 3.1.3.2, 3.2.0.0 and 3.2.0.1 are installed, the newest version 3.2.0.1 will be loaded, even if it has not yet been provided at the moment the library has been included to the project (and so e.g. version 3.1.3.1 has originally been included).

This procedure is obligatory for interface libraries! Hence, all of the included libraries implementing this interface agree upon the newest version and this is the only one that will be loaded. Otherwise, in case of mixed versions compiler errors may occur, as the same interface exists in different and therefore incompatible versions. Therefore interfaces and their implementation have to been divided in disparate libraries (see Chapter 2.7)!

2.4.3 Placeholder

Within the device description of a PLC a library may be defined as placeholder. When this placeholder gets included into a project or a library, the real version will be adopted from the device description. Therefore, only the specific version of a library being defined by the device description will be employed.

IMPORTANT:
All libraries containing external implementations (see Chapter 2.5) have to be included as placeholders, so that the library will always fit to the implementation within the runtime system!

2.5 External libraries

Libraries being implemented in ANSI-C within the runtime system are called external libraries. However, CoDeSys V3 will not differentiate any longer between internal and external libraries. (For further information OEMs may have a look to [1], Chapter 9.4.)

Instead, each object can be declared as internal or external within the object properties. In case of an external object for each function to be linked an entry will be created within the list of external references. This entry includes the name of the function without namespace, the address of the function pointer as well as a check code for the interface (CRC of signature). Functions implemented within the runtime system must have an unique name! Thereby each function or each module of a library may be implemented individually within the runtime system.

The name within the library typically equals the name within the runtime system, but it does not have to. For example, the name of an external function might clash wit an IEC identifier. In this case the external function may be decorated with the attribute „external_name“:

```
{attribute 'external_name' := 'name_on_rts'}
FUNCTION ExtFun : BOOL
VAR_INPUT
END_VAR
VAR
END_VAR
```

In this case the function will not be identified with the function „ExtFun“ of the runtime system, but with a function „name_on_rts“.

As there is only one implementation of a function or module within the runtime system, only the library version with exactly this implementation may be used in the project! This is typically realized by the placeholder concept (see 2.4.3). Therefore an external library is typically deposited in the device description.

This may be omitted, if the interface of a library is not modified from one version to the other.

2.5.1 Version check of libraries against the runtime system

Each library function and each library method of a module being implemented in the runtime system will be checked on version compatibility by the runtime system- Each function and each method of a module will thereby inherit the version of its library.

If the first two digits of this version number (of length 4 digits) do not correspond, the download will be denied and a related error message will be created in CoDeSys. Therefore the version of the (external) library and of its implementation within the runtime system have to agree in the first two digits.

2.5.2 Signature check of libraries against the runtime system

Each library function and each library method of a module being implemented in the runtime system has a defined interface. As only the name of the function or method will be checked by the runtime system, fatal errors culminating in system crashes may occur, if these interfaces do not agree with each other.

This is the reason why a checksum for each function or method is transferred via the interface to the runtime system. By this CRC the runtime system may check the particular implementation.

The checksum of the interface of function or method contains the data type of the return value as well as name and data type of the parameters.

As the check sum may only be calculated within CoDeSys, CoDeSys provides the possibility to automatically export all externally implemented functions or methods of a library to a file for the runtime system. You will find this described in detail in the following paragraph.

2.5.3 Export in m4 files

CoDeSys V3 provides the possibility to automatically export all externally implemented functions or methods of a library to a file for the runtime system, so that incompatibilities between the library and the implementation in the runtime system are almost precluded.

In the created file (called m4 file) all externally implemented functions and methods are represented by their names (as defined in CoDeSys or specified by the attribute „external_name“), their interfaces, the version of the library and the checksum of the interface. Via a m4 compiler this file has to be translated into an ANSI-C interface Header file. Furthermore CoDeSys provides the possibility that a C-Source frame for the implementation gets generated.

Within the runtime system the interface header file (<LibraryName>Itf.h) and the source file (<LibraryName>.c) may then be used as base for implementing a component containing the implementation of the external functions or modules. Afterwards this component can be included into the runtime system.

2.6 Implicitly loaded libraries

Internal components of CoDeSys creating IEC code (e.g. symbol configuration or visualization) often require library elements. Therefore these libraries are included automatically, i.e. implicitly into the project by CoDeSys. These libraries are displayed in gray within the library manager.

Libraries may also be implicitly loaded by a device description. For example this is used for IO devices bringing a library with them while being included. Mostly, also an instance of an IO driver will be created from this library at the same moment.

2.7 Separating interfaces from implementations

Creating libraries you should pay attention to dislocate interfaces being implemented by different libraries in a separate library.

Under this condition, all libraries implementing a common interface may include this library employing the "Newest" constraint (see Chapter 2.4.2). Otherwise compile errors may result, if two different versions of this interface library are included to the project, as the CoDeSys compiler will detect an incompatibility of an interface being available in two different versions.

2.8 Release of libraries

Within the project information of a library a release flag serves for identifying a released library. This information may also be read out automatically (e.g. Rep-Tool).

This flag having been set each attempt to modify the library will provoke a warning.

Bibliography

[1] CoDeSys SP 3.x Runtime System Overview.pdf (OEM documentation)

Change History

Version	Description	Date
0.1	Translation from german version 2.0	02.09.2008
2.0	Release after formal review (set to doc version 2.0 in order to match German version)	03.09.2008