



Creating Runtime System Components, Libraries and I/O Drivers

Tutorial

Version: 6.0

Template: templ_tecdoc_en_V1.0.docx

Dateiname: CODESYSControlV3_Create_Components_Libraries_IOdrivers.docx

CONTENT

	Page
1 Overview	3
2 Summary of basic concepts	3
2.1 Component based Architecture	3
2.2 Single Source, Macros and M4	4
2.3 Macros for Function Calls	5
3 Create and compile a Runtime System Component	6
3.1 Creating a Component	6
3.2 Compiling a Component	6
3.3 Runtime Configuration	6
4 Create a CODESYS Library with external functions	8
4.1 Basic Concepts	8
4.2 Create a CODESYS Library	8
4.3 Extend the Runtime System Component	10
5 Create an I/O Driver in C	12
5.1 Basic Concepts	12
5.2 Create an I/O Driver	12
5.2.1 Initialization	13
5.2.2 Download	13
5.2.3 Reading Inputs	13
5.2.4 Writing Outputs	14
5.2.5 Start Bus Cycle	14
5.2.6 Scan Modules	14
5.2.7 Diagnosis	14
Appendix A: Step by Step: Creating a new Component	15
Appendix B: Step by Step: Compiling a Component	19
Appendix C: Step by Step: Creating a new I/O Driver	21
Change History	23

1 Overview

The CODESYS runtime system is component based. With the CODESYS Runtime Toolkit, device manufacturers (OEM customers) can extend the CODESYS runtime system by creating additional OEM specific runtime components.

These additional components live inside the runtime system, like all the other runtime system components supplied by 3S. The additional components can use the same interfaces as the other runtime components, so this is a quite powerful feature.

The additional components can be grouped like this:

- Components that just use the functions of other runtime components, for example react on runtime system events.
- Components that implement a CODESYS library with external functions and function blocks. These components realize a library written in C code. You can also integrate your existing C code functions to such a component.
- Components that implement the CODESYS IODriver interface. These components realize an IODriver written in C code.

Components can also be a mix of the above types.

Writing components must follow the basic concepts of the CODESYS runtime system. The basic concepts are

- Component based architecture
- Single source and m4
- Macros for function calls

The component based architecture requires some functions that are used as frame code. These functions are part of every component.

Single source requires macros when calling functions from other components. These macros allow using the same code in different compile scenarios.

These concepts are described in more detail in the runtime manual. The implications are explained in the following chapters.

2 Summary of basic concepts

Please also look into CODESYSControlV3_Manual, chapter 2 Architecture.

2.1 Component based Architecture

The runtime system is component based. There is one specific component in the runtime system, which is called the component manager. It is in charge of initializing the components.

Every component must have some component frame code. This is a set of functions, called by the component manager. Most of these functions are used for initialization.

The functions are

- ComponentEntry
- ExportFunctions
- ImportFunctions
- CmpGetVersion
- HookFunction.

Most of these functions make use of predefined macros, so you do not need to care about the implementation. You can re-use the implementation from the template components that are part of the runtime toolkit delivery.

When starting to write a component, use the template components provided in the toolkit as a starting point.

Probably, you will want to add your own init code into HookFunction.

In the following, we call these functions “component frame code”.

2.2 Single Source, Macros and M4

As the runtime system is implemented in C language, the components consist of C source file(s) (*.c) and header files (*.h).

The runtime supports

- static linking: all C files are linked to one binary by the linker, and
- dynamic linking: every component is compiled to a separate library (e.g. .dll or .so)
- mixed linking: some components are statically linked, others are linked as separate libraries

To support single source for static, dynamic and mixed linking, macros are used for function calls, especially if functions from another component are called. These macros are defined in the header files.

Every component has two header files: the interface file and the dependency file.

- Interface .h file: Called Cmp*Itf.h, contains declarations for exported functions
- Dependency .h file: Called Cmp*Dep.h, contains import declarations

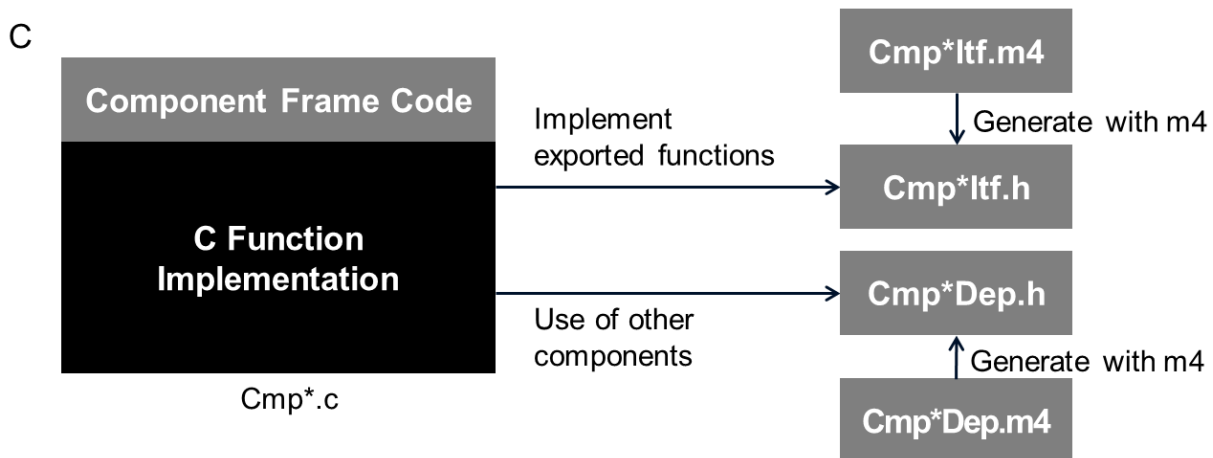
As many macros are needed for all the functions, we are using a tool to generate the header files with the macros. This tool is called the “m4” preprocessor. This tool is provided with the runtime delivery.

The source files for the header files are also called m4 files. As there are two header files for every component, there are also two m4 files:

- Interface .m4 file: Called Cmp*Itf.m4, contains declarations for exported functions
- Dependency .m4 file: Called Cmp*Dep.m4, contains import declarations

When writing a component, do not edit the header files directly. Only edit the .m4 files! Then, generate the .h files with the m4 tool.

The following drawing illustrates the structure of the source files of a component:



In the template components, some batch files are provided. They call the m4 preprocessor to generate the .h files from the .m4 files.

If your components has a new dependency on another component, e.g. you want to call a function from another component, you need to configure this new dependency. You can do that in the Dep.m4 file of your component. You need to add the component interface, that you want to use with “USE_ITF”, and you need to specify which function you want to use with “REQUIRED_IMPORTS” or “OPTIONAL_IMPORTS”.

Example:

If you want to use the functions EventOpen and EventClose from runtime component CmpEventMgr (which is implementing the interface “CmpEventMgrItf”, declared in CmpEventMgrItf.m4), add this new dependency to your Dep.m4 file:

```
USE_ITF(`CmpEventMgrItf.m4')
```

```
REQUIRED_IMPORTS(
EventOpen,
EventClose)
```

2.3 Macros for Function Calls

The C syntax of calling functions from other components depends on the linking method.

- Static linking (all components are compiled and linked to one binary): A function can be called just using its name.
Example: `MyFunction();`
- Dynamic/Mixed linking (every component is linked to one library). For example, this is a .dll in Windows or a .so in Linux: Function pointers are needed. The function pointer has to be
 - declared,
 - filled with the correct value, or
 - exported,
 - checked if valid and
 - used for the call.

Therefore, the following macros are used. They are generated by the m4 mechanism:

- Function pointers are declared in this macro:
`USE_STMT`
It must be placed at the beginning of the C source file.
- Functions are exposed to the rest of the runtime, and exported to the component manager, with this macro:
`EXPORT_STMT`
It must be placed inside `ExportFunctions()`.
- Function pointers from other interfaces/components are retrieved in this macro:
`IMPORT_STMT`
It must be placed inside `ImportFunctions()`.
- Function pointers are checked for “availability” with this macro:
`CHK_<function name>`
It must be used before calling a function from another component.
Example: Before calling function `EventOpen()` from component `CmpEventManager`, use this macro:
`if (CHK_EventOpen)`
...
- Functions are called with this macro:
`CAL_<function name>`
It must be used when calling a function from another interface/component.
Example: For calling function `EventOpen()` from component `CmpEventManager`, use this macro:
`s_hEvent = CAL_EventOpen(...);`

When writing a component, use macros like `CHK_` and `CAL_` to call functions from other components.

3 Create and compile a Runtime System Component

3.1 Creating a Component

Please also look into CODESYSControlV3_Manual, chapter 10.3 Implementing own components.

To create a new component, start with a template component provided in the runtime toolkit delivery. We recommend starting with one of these templates:

- **CmpTemplate**: Quite complex template with examples for various runtime mechanisms and common functionalities
- **CmpTemplateEmpty**: Minimal template component, without example code (only empty component frame)
- **CmpTemplateEvent**: Minimal template component, with example code how to react on runtime system events
- **CmpTemplateEventIEC**: Minimal template component, with example code how to create runtime system event, and how to react in IEC code.

Copy the template and adjust the file names, component name, vendor id and version. A Python script (gen_cmp_from_temp.py) is provided that can help you with this task.

A step by step instruction is given in chapter “Appendix A: Step by Step: Creating a new component”.

3.2 Compiling a Component

To compile the component you need to create a makefile or project for your C compiler. This is highly system/toolchain specific.

You also have to add compiler defines and include paths.

A step by step instruction for various operating systems can be found in chapter “Appendix B: Step by Step: Compiling a component”

3.3 Runtime Configuration

Please also look into CODESYSControlV3_Manual, chapter 10.2 Configuration.

Add your component to the configuration of the runtime system (“tell the component manager to load and initialize your component”).

The method depends on the linking method.

- Systems with dynamic/mixed loading of additional components (Windows, Linux, VxWorks, ...):
 - Just add the name of the component to the CODESYSControl.cfg runtime configuration file. It has to be appended to the section [ComponentManager], for example:


```
[ComponentManager]

Component.1=CmpTargetVisuStub
Component.2=CmpCodeMeter
Component.3=CmpWebServer
Component.4=CmpWebServerHandlerV3
Component.5=CmpTemplate
```


Please note that the numbering is subsequent (Component.4 followed by Component.6 would not be recognized).
- Systems with static linking of additional components (embedded systems)
 - Add the component C source file to the workspace or makefile of the runtime system
 - In your main runtime C header file, add the component to the list of components in #define COMPO_INIT_DECLS
 - In your main runtime C header file, add the component to the list of components in COMPO_INIT_LIST (or COMPO_DYNAMIC_INIT_LIST)

You can check if your component is loaded with the output of the logger in CODESYS device view, or in the console output of the runtime at startup.

For example, console output like this can be expected:

```
01441104355283: Cmp=CM, Class=1, Error=0, Info=10, pszInfo= Dynamic: <cmp>CmpTem  
plate</cmp>, <id>0x00012000</id> <ver>3.5.8.0</ver>
```

Or in the PlcLogger:

	25.05.2016 18:37:30.580	Dynamic: CmpTemplate, 0x00012000 3.5.9.0	CM
---	-------------------------	--	----

Alternative: you can check if the component was added correctly, by setting a breakpoint inside the ComponentEntry or the HookFunction of this component.

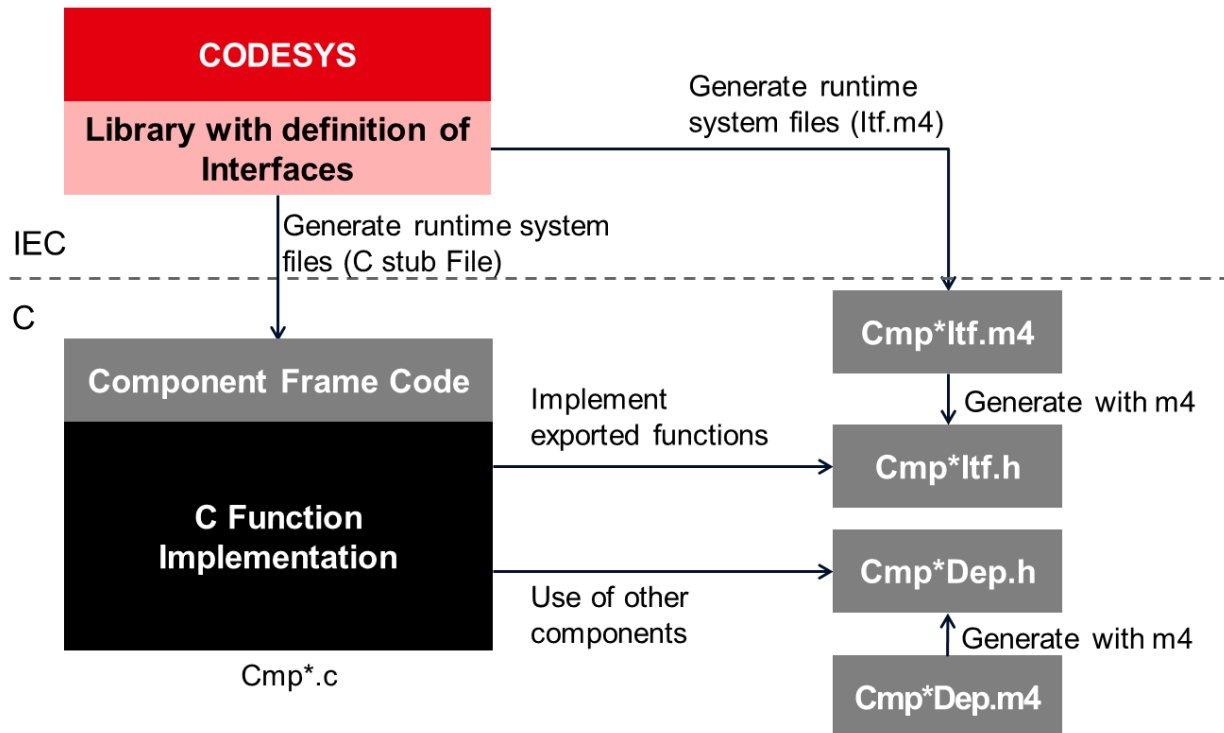
4 Create a CODESYS Library with external functions

4.1 Basic Concepts

Please also look into CODESYSControlV3_Manual, chapter 10.5 Libraries.

CODESYS libraries can contain “internal” and “external” functions. Internal functions are implemented in IEC code and do not require any extension of the runtime system. External functions are implemented in C code. They are implemented inside a runtime system component.

The following drawing illustrates the structure of a runtime component implementing some external library functions:



This is an extension of the concept explained in chapter 2.

The CODESYS library contains the interfaces of the external functions. Using the menu command “Generate runtime system files”, CODESYS will create

- definitions that will be part of the ltf.m4 file, and
- code that can be used as a template for the C implementation.

This helps to create components that contain the C functions exactly matching the interfaces defined in IEC..

4.2 Create a CODESYS Library

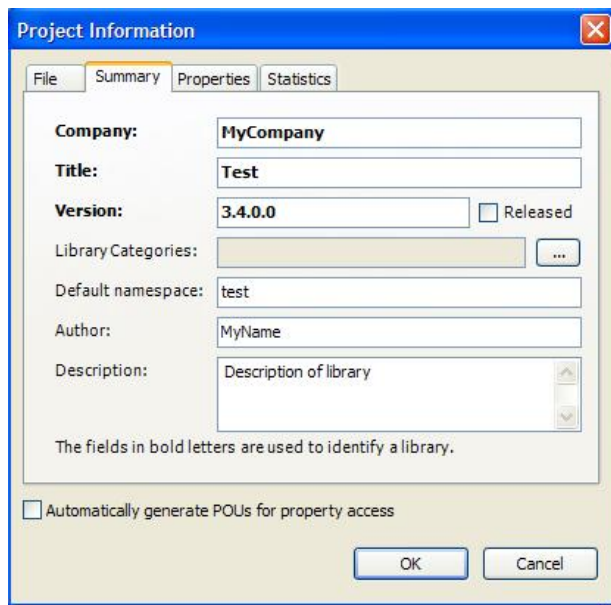
This chapter shows how to create a CODESYS library that declares external functions.

The code of this function has to be implemented inside a runtime component. In the next chapter, we will have a look on the runtime part and create a component that implements this function.

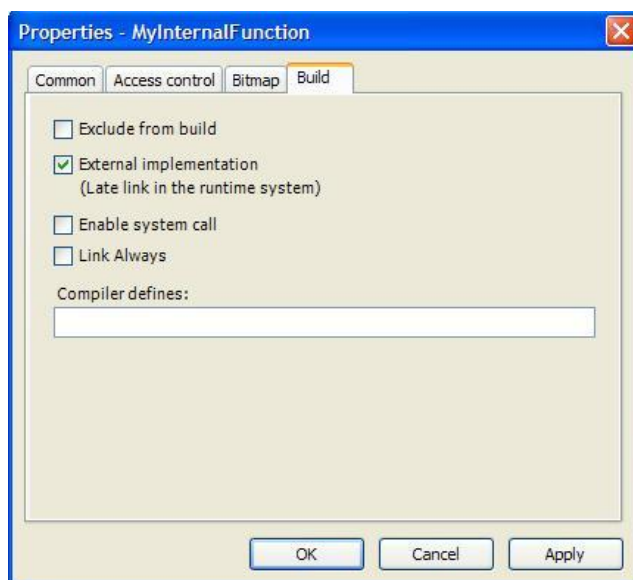
The CmpTemplate folder of the runtime delivery contains a library and the component. You can use this as a starting point or create a new library:

- Create a new empty CODESYS library from the Menu: *File -> New Project*

- Edit project information (*Project -> Project information...*)



- Add your function and function blocks to the POU pool. Every CODESYS V3 library can have internal (in IEC) and external (in C) POUs mixed:
 - a. For every internal POU, you have to implement the body in IEC.
 - b. For every external POU, you have to set the flag “external implementation” in the POUs properties and leave the body empty.



- If the “Check all Pool Objects” check does not show any compile errors, save and install the library into the library repository:



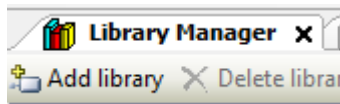
Note: During library development, it is recommendable that you start two instances of CODESYS: One with your test project and in the other instance, you edit the library. There you can use this button to save and install your modified library. The library is then automatically reloaded in the other instance.

- Generate and save the runtime system files (in the CODESYS instance with the library) with *Build -> Generate runtime system files*. In the following dialog, select “M4 interface file” and “C stub file”. You will need both for your runtime system component.

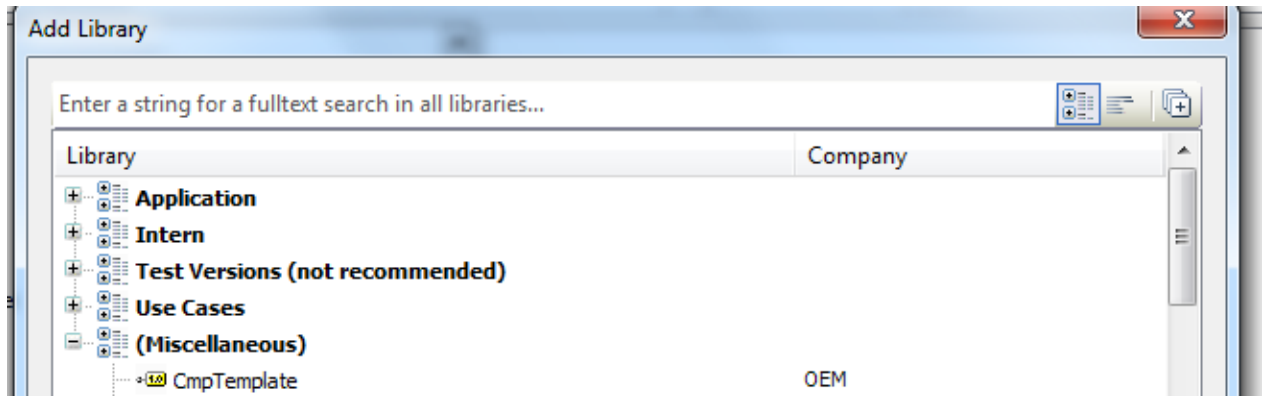
Note: If you change your interface later, you need to merge those files manually.

Now the library is ready to use in a project. For test, you can create a new project, add the new library to your application and call the internal and external functions and function blocks from the IEC code.

- To use the library in your project, open the library manager of your application, and choose the button "Add library":



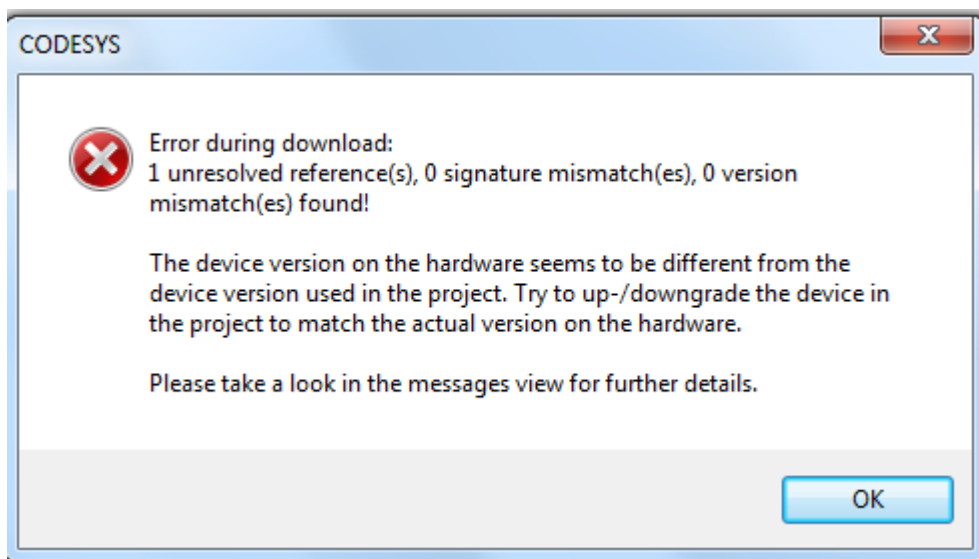
- The library will show up in the library list. The position depends on the library categories you did choose when creating the library. In this example, we did not define a library category, so the library appears under miscellaneous:



- You can now add the function call to your IEC application. Now, you can compile your application without errors.

If you download the application to a runtime system, you will receive errors, because of unresolved external references. Unresolved external references appear, when the project contains calls to external POU's that are not implemented in the runtime.

You will receive an error message like this:



And in the logger:

Severity	Time Stamp	Message	Component
Error	01.09.2015 12:59:19.971	Could not link external function MYEXTERNALFUNCTION	CmpApp

To resolve these errors, the library functions must be implemented in the runtime system component. This is explained in the next chapter.

4.3 Extend the Runtime System Component

Start with a runtime component as explained in chapter 3.

With the menu command “Generate runtime system files”, an interface file (Cmp*Ift.m4) is created. It will contain the interface of the library, mainly the list of exported functions.

This menu command also generates a .c file with empty function declarations, corresponding to the IEC function interfaces of the library.

You will edit this file by adding the function implementation in C code, and by adding the runtime component interface to your C file.

The runtime component frame code is the set of functions common for all components: These functions are ComponentEntry, CreateInstance, DeleteInstance, ExportFunctions, ImportFunctions, CmpGetVersion, HookFunction and QueryInterface. Examples are provided in the delivery, and you can copy these functions from the CmpTemplate (or other) component, and adapt the component names.

The menu command also creates a dependency (Cmp*Dep.m4) file. The dependency file contains the dependencies of your component. These are component interfaces and functions used from other components. The dependency file also contains the identification (name, id, and version) of your component.

With the m4 macro preprocessor, you generate C header files (Cmp*Dep.h and Cmp*Ift.h) from the interface and dependency files.

You will include these generated C header files in your C source file. Your C compiler will be used to compile the components C source file.

The compiled component is then added to the runtime system configuration. Now you can download a CODESYS application calling your C code functions.

5 Create an I/O Driver in C

5.1 Basic Concepts

A CODESYS I/O Driver in C is, in general, just like a normal runtime component. So we are using the component from the previous chapter as a starting point for our I/O driver. You can also use templates for I/O Drivers, like `IoDrvSimple` or `IoDrvVerySimple`.

Every I/O driver must have a corresponding device description file (*.devdesc.xml). This file defines connectors, parameters and I/O channels. Every element is identified by a unique id. The corresponding I/O driver must know these unique ids.

For more information about device description files, see also `CODESYS_DeviceDescriptions_en.pptx`.

For more information about device description files and I/O drivers, see also `CODESYSControlV3_Manual.pdf`, chapter 7 Device- / I/O Configuration and chapter 8 I/O Drivers.

The I/O Driver component will use additional interfaces. They will be added to the `USE_ITF` section of the `dep.m4` file:

- `CmpIoMgr`: We need to register our driver at the I/O Manager.
- `SysMem`: We need to allocate some memory dynamically for our driver instance.
- `SysCpuHandling`: We want to allow bit access and use `SysCpuTestAndSet()` for this.

Additionally to this, we also need to implement some predefined interfaces in our component. They will be added to the `IMPLEMENT_ITF` statement of the `dep.m4` file:

- `CmpIoDrvItf`: This is the I/O driver interface and defines functions like „`IoDrvReadInputs()`“, „`IoDrvWriteOutputs()`“ or „`IoDrvUpdateConfiguration()`“. This is the main interface between an IEC task and our I/O driver.
- `CmpIoDrvParameterItf`: This is just a small interface. It allows the I/O Manager to read parameters from the I/O Driver. Implementing this interface is optional.

These interfaces are already defined in `CmpIoDrvItf.m4` and `CmpIoDrvParameterItf.m4`. We don't need to define new interfaces, in contrary to libraries described above. We also don't need to create a new CODESYS library for an I/O driver written in C.

In the runtime system core, the component I/O Manager (`CmpIoMgr`) manages all I/O drivers. Calls to the I/O drivers are made through this component. They are not made directly from the IEC code to the IO driver.

5.2 Create an I/O Driver

Please also look into `CODESYSControlV3_Manual`, chapter 10.3 Implementing own components.

To create a new I/O driver, start with a template I/O driver component provided in the runtime toolkit delivery. You can use these templates:

- `IODrvTemplate`: I/O driver template component
- `IODrvSimple`: Quite simple I/O driver template component
- `IODrvVerySimple`: Minimal I/O driver template component
- `IoDrvMasterTemplateIEC`: Extensive example with
 - I/O driver library written in IEC
 - Device descriptions with comments and explanations
 - Examples for diagnosis
 - Several types of submodules
 - Documentation in the library
 - Example project
- `IoDrvSubModulesFix`: Device description example with fixed submodules
- `IoDrvSubModulesVar`: Device description example with var submodules
- `IoDrvSubModulesSlot`: Device description example with slot submodules

Copy the template and adjust the file names and component name. A Python script (`gen_cmp_from_temp.py`) is provided that can help you with this task. Adjust vendor id and version.

A step by step instruction is given in chapter “Appendix C: Step by Step: Creating a new I/O Driver”.

The most important steps in an IODriver are:

5.2.1 Initialization

At startup, an instance of the driver is created and registered at the I/O manager. This is done in the HookFunction:

```
case CH_INIT2:
{
    RTS_HANDLE hIoDrv = 0;
    void *pTmp;

    /* first instance */
    pTmp = CAL_IoDrvCreate(hIoDrv, CLASSID_CIoDrvVerySimple, 0, NULL);
    s_pIBase = (IBase *)pTmp;
    CAL_IoMgrRegisterInstance(s_pIBase, NULL);

    break;
}
```

I/O driver interface function IoDrvCreate is called in this case.

From now on, the IOManager knows that our driver instance is available. In this example we create one instance, but it is possible to create and register more than one instances. This can be useful, e.g. when more than one IO-device respectively fieldbuscard is available.

5.2.2 Download

At download, loading of a bootapplication or at reset of an application, the I/O driver interface function IoDrvUpdateConfiguration is called. You have to differentiate between two cases:

- The parameter pConnectorList is NULL: if an application was loaded before and is removed before the new download starts.
- The parameter pConnectorList is not NULL: then pConnectorList contains the complete resource/connector tree as defined by the user, including submodules, parameters and channels.

I/O Manager helper functions can be used to read information from pConnectorList, for example IoMgrConfigGetFirstConnector.

If a connector is found that our driver feels responsible for, it registers in that connector. This is done in this line:

```
pConnector->hIoDrv = (RTS_IEC_HANDLE)pIBase;
```

From now on, the I/O Manager knows that our driver is responsible for this connector.

The driver can read information about child connectors (with IoMgrConfigGetFirstChild) and read parameter values (with IoMgrConfigGetParameter).

The driver can also prepare for I/O update. The member dwDriverSpecific of the IoConfigParameter structure can be used to store any information. In our example, we store a pointer to global memory.

```
pParameter->dwDriverSpecific = (RTS_IEC_BYTE *)&s_ulyInputs[i];
```

In real drivers, this memory could be refreshed by a low level driver to update physical I/Os.

Our driver also write diagnostic information to the connector (with IoMgrConfigSetDiagnosis). This information will be used by CODESYS in online mode, to display the status of the modules in the device tree (indicated by green circles or red triangles).

I/O driver interface function IoDrvUpdateMapping can be used to update and optimize the mapping of channels.

5.2.3 Reading Inputs

I/O driver interface function IoDrvReadInputs is called at the beginning of every IEC task. A list of all channels that need to be updated in this task is passed, together with a list of used I/O modules. This function shall copy the values of physical inputs to the IEC input area. We are using I/O manager function IoMgrCopyInputLE to copy from the physical to the logical memory.

Use IoMgrCopyInputBE if the physical memory is big endian.

5.2.4 Writing Outputs

Very similar to reading inputs. I/O driver interface function `IoDrvWriteOutputs` is called at the end of every IEC task. A list of all channels that need to be updated in this task is passed, together with a list of used I/O modules. This function shall copy the values of the IEC output area to physical outputs. We are using I/O manager `IoMgrCopyOutputLE` to copy from the physical to the logical memory.

Use `IoMgrCopyOutputBE` if the physical memory is big endian.

5.2.5 Start Bus Cycle

I/O driver interface function `IoDrvStartBusCycle` is called in one specific task, called the bus cycle task. This task can be specified by the user. E.g.: Fieldbus drivers can send telegrams to the fieldbus slaves here.

5.2.6 Scan Modules

I/O driver interface function `IoDrvScanModules` is called when the user executes the command “Scan for devices”. The function has to fill the `ppConnectorList`. It has to allocate the memory for this connectorlist.

5.2.7 Diagnosis

I/O driver interface function `IoDrvGetModuleDiagnosis` is called to set diagnostic information. I/O manager function `IoMgrConfigSetDiagnosis` can be used to set the diagnostic information to a connector.

Appendix A: Step by Step: Creating a new Component

To create a new component, the CmpTemplateEmpty component can be used as a starting point. The CmpTemplateEmpty is included in the runtime system delivery.

To copy and rename the files, you have two possibilities:

1) (recommended method): you can use the Python script gen_cmp_from_temp.py. To run the script, you need Python installed. The script can be used like this:

```
gen_cmp_from_temp [-h] [-t TEMPLATE] [-c CMPNAME] [-a TRGARCH]
```

TEMPLATE is the template component name which is used for generating component/driver.

CMPNAME is the name for the target component

TRGARCH is the target architecture, options (only capital letter): X86 | PPC | ARM | CORTEX.

- Open a command prompt (DOS/shell) and go to the runtime Templates folder (e.g. D:\SVN\CodesysSpV3\trunk\CodesysSpV3\Templates)
- Run >gen_cmp_from_temp -t CmpTemplateEmpty -c CmpTest

This will create a new folder with the given name.

2) copy and rename the files by hand, please follow these steps:

- Copy the CmpTemplateEmpty folder and rename the folder to the new component name.
Delete all files from your component folder, except:
CmpTemplateEmpty.c
CmpTemplateEmptyDep.m4
CmpTemplateEmptyItf.m4
CmpTemplateEmptyItf_m4.bat
CmpTemplateEmptyItf_m4.bat
- Rename all files to the new component name (e.g. CmpTest):
CmpTemplateEmpty.c -> CmpTest.c
CmpTemplateEmptyDep.m4 -> CmpTestDep.m4
CmpTemplateEmptyItf.m4 -> CmpTestItf.m4
CmpTemplateEmptyDep_m4.bat -> CmpTestDep_m4.bat
CmpTemplateEmptyItf_m4.bat -> CmpTestItf_m4.bat
- Open the CmpTestItf_m4.bat file and replace the CmpTemplate component name with the new component name. If you have changed the directory structure, you have to adapt the path to the m4 build utils in the *.bat files, too.
Note: If you change the path to the "BuildUtils", make sure to be aware of the two different kind of slashes ('/' and '\') which are used. If you are using absolute paths, you can use syntax like this for the paths with the forward slashes: d:/CODESYSSpV3/BuildUtils/.../...
Note: The M4 tools need an empty directory named "etc" in "BuildUtils/msys/etc". If this doesn't exist, you need to create it.

Now, execute the modified CmpTestItf.bat file. This will generate a new, non-empty *Itf.h file. If there is now new *Itf.h file or the file is empty, the component name or the path to the m4 build utils is not correct.

- Do the same with the CmpTemplateDep_m4.bat file. If everything is correct, you will get a new *Dep.h and a new *.cpp file. Both files must not be empty.
- Change the Component Information in *Dep.m4:
 - a. Set the new component name in the SET_COMPONENT_NAME macro.
 - b. Set the name of the c file in the COMPONENT_SOURCES macro. List all c files separated with commas.
 - c. Set the component version in the COMPONENT_VERSION macro.
 - d. Set the component vendor ID.
Note: Every vendor gets an ID from 3S. The value 0x0000 is the 3S vendor Id and should not be used.
 - e. Rename and set the CMPID_ (component Id), CLASSID_ (class Id) and ITFID_ (interface Id) of your new component.

- f. Remove the category macro, it is not needed at the beginning.
- g. Set the implemented interfaces in the IMPLEMENT_ITF macro.
At the beginning the component will implement only its own interface.
- h. If your component uses the interface of another component, you have to add the component in the USE_ITF list. Remove all unused USE_ITF definitions. For more information, see the documentation of the m4 mechanism.
- i. In addition to the USE_ITF macro, you have to add every function, which is used by your new component, to the REQUIRED_IMPORTS macro. Remove all unused functions.

Our Example may now look like this:

```
/**
 * <name>Component Template</name>
 * <description>
 * An example on how to implement a component.
 * This component does no useful work and it exports no functions
 * which are intended to be used for anything. Use at your own risk.
 * </description>
 * <copyright>
 * (c) 2009 3S-Smart Software Solutions
 * </copyright>
 */
SET_COMPONENT_NAME(`CmpTest`)
COMPONENT_SOURCES(`CmpTest.c`)

COMPONENT_VERSION(`0x11223344`)

/* NOTE: REPLACE 0x0000 BY YOUR VENDORID */
COMPONENT_VENDORID(`0x5678`)

#define CMPID_CmpTest 0x2000
#define CLASSID_CCmpTest 0x2000
#define ITFID_ICmpTest 0x2000

CATEGORY(`Customer`)

IMPLEMENT_ITF(`CmpTestItf.m4`)

USE_ITF(`SysTimeItf.m4`)
```

```
REQUIRED_IMPORTS(
SysTimeGetMs)
```

Open the *Itf.m4 file and set the interface name in the SET_INTERFACE_NAME macro.
The *Itf.m4 file may look like this:

```
/**
 * <interfacename>test</interfacename>
 * <description></description>
 *
 * <copyright></copyright>
 */

SET_INTERFACE_NAME(`CmpTest`)

/** EXTERN LIB SECTION BEGIN **/

#ifdef __cplusplus
extern "C" {
#endif

/**
 * <description>myexternalfunction</description>
 */
typedef struct tagmyexternalfunction_struct
{
    RTS_IEC_INT MyExternalFunction;          /* VAR_OUTPUT */
} myexternalfunction_struct;

DEF_API(`void', `CDECL', `myexternalfunction',
` (myexternalfunction_struct *p)', 1, 0xE1C6D757, 0x3040000)

#ifdef __cplusplus
}
#endif

/** EXTERN LIB SECTION END **/
```

Execute the *Dep_m4.bat and the *Itf_m4.bat to generate the new header files. If you want to edit the header files afterwards, you have to edit the m4 file and then you have to create the header files using the bat files.

- a. Never change the header files directly!
- b. Always generate both header files! (even if you change only one of them)

If you want to implement an external function (defined in an own IEC Library), add it now to your C File (CmpTest.c). You can copy the declaration from the header file *Itf.h.

For example:

- a. If you have the following Declaration in your M4 file (generated from CODESYS):

```
DEF_API(`void', `CDECL', `myexternalfunction',  
  `(myexternalfunction_struct *p)', 1, 0xFFADC096, 0x1000000)
```
- b. Executing the corresponding batch file will create the following definition in your *Itf.h:

```
void CDECL CDECL_EXT myexternalfunction(myexternalfunction_struct *p);
```
- c. Then you can create the following declaration in your C-File (CmpTest.c):

```
void CDECL CDECL_EXT myexternalfunction(myexternalfunction_struct *p)  
{  
    /* function body */  
}
```
- d. When implementing the function, make sure you are using the CAL_ macros instead of calling functions of other components directly. Add the newly references functions to the list of REQUIRED_IMPORTS or OPTIONAL_IMPORTS, and add a section USE_ITF if a component is references newly.

Appendix B: Step by Step: Compiling a Component

How to compile your component exactly depends hardly on the Platform/toolchain you are using. We cannot describe every IDE or Makefile mechanism here, but we picked up a few examples.

- Create a new project to compile the source files of our new component:
 - a. VxWorks / Tornado:
 - i. *File -> New Project*
 - ii. Select „Create downloadable application modules for VxWorks“
 - iii. Finish the Wizard
 - iv. *Right click on new Project -> Add files ...*
 - v. Select your Source file (CmpTest.c)
 - b. VxWorks / Workbench:
 - i. *File -> New -> VxWorks Downloadable Kernel Module Project*
 - ii. Finish the Wizard
 - iii. Delete second build-target named <project-name>_partialImage
 - iv. Delete reference to <project-name>_partialImage from first build target
 - v. Copy your Component folder under your newly created project.
 - vi. *Right click on the project -> Refresh*
 - vii. *Right click on the build target -> Edit content*
 - viii. Select your source file (CmpTest.c) and press „Add“
 - ix. *Right click on the project -> Build Options -> Set Active Build Spec...*
 - x. Select the corresponding build specification for your platforms
(Note, that we only support the GNU compiler)
 - c. Visual Studio 2015:
 - i. *File -> New -> Project*
 - ii. Visual C++ -> Win32 and here select a “Win32 Project”
 - iii. Select a name and the path for the project and solution
 - iv. Select in the Wizard “Application Settings” and select here “DLL”
 - v. Check “Empty project” and uncheck “Precompiled header”
 - vi. Uncheck optional “Security Development Lifecycle (SDL) checks” if not necessary
 - vii. Finish the Wizard
 - viii. Change to „Solution Explorer“ Tab
 - ix. *Right click on „Source Files“ -> Add -> Existing Item...*
 - x. Select you source file (CmpTest.c)
 - xi. *Right click on „Header Files“ -> Add -> Existing Item...*
 - xii. Select your header files (*Itf.h and *Dep.h) and both M4 files
 - xiii. Right click on each m4 file and select *Properties*
 - xiv. Go to category *General* and select in “Excluded From Build” the option “Yes”
 - xv. Close the property dialog with saving the changes
 - d. Linux:
 - i. Use the supported makefile under Templates/CmpTemplate/Linux as a basis for your component

- Add Compiler Defines and Include Paths:
The most essential preprocessor defines are:
STATIC_LINK, DYNAMIC_LINK and for some RISC CPUs also NO_PRAGMA_PACK
 - a. VxWorks / Tornado:
 - i. Change to the „Builds“ Tab
 - ii. *Right click on the „Build Spec“ under your project -> Properties*
 - iii. Change to the „C/C++ Compiler“ Tab
 - iv. Add the following compiler flags:
 1. -DMIXED_LINK (that's always necessary in VxWorks)
 2. -I<path-to-runtime-interfaces>/Components
 3. -I<path-to-runtime-interfaces>/Platforms/VxWorks
 - b. VxWorks / Workbench:
 - i. *Right click on build target -> Properties*
 - ii. Change to „Build Macros“ Tab
 - iii. Add the following Define:
 1. -DMIXED_LINK (that's always necessary in VxWorks)
 - iv. Change to „Build Paths“ Tab
 - v. Add the following Build Paths to your configuration:
 1. -I<path-to-runtime-interfaces>/Components
 2. -I<path-to-runtime-interfaces>/Platforms/VxWorks
 - c. Visual Studio 2015:
 - i. *Project -> Properties*
 - ii. Change to „C/C++“ Category
 - iii. Select Category „General“
 - iv. Add the following paths, comma separated to „Additional Include Directories“:
 1. <path-to-runtime-interfaces>/Components
 2. <path-to-runtime-interfaces>/Platforms/Windows
 - d. Linux:
 - i. Additional hints:
 - The template makefile requires the tools “m4” and “to/fromdos” installed to work properly, usually they are included in your Linux derivate, or at least installable via the default package mechanism.
 - ii. Adopt and check the following variables in the makefile:
 - TARGET: should be the component name (without lib prefix and without “.so” extension): “CmpTemplate” is the default (this leads to an object called “libCmpTemplate.so”).
 - INCLUDE: should include all required include paths for compilation
 - CXXFLAGS
 - ➔ TRG_ define: at least one “TRG_” define is required: choose the correct one from “TRG_X86 | TRG_PPC | TRG_ARM | TRG_CORTEX”.

Appendix C: Step by Step: Creating a new I/O Driver

To create a new I/O driver, the `IoDrvVerySimple` component can be used as a starting point. The `IoDrvVerySimple` is included in the runtime system delivery.

To copy and rename the files, you can use the Python script `gen_cmp_from_temp.py` or copy and rename them manually. To run the script, you need Python installed. The steps are the same as for creating a new component. See description above.

Notes:

- An I/O driver implementation will need some functions from other components of the runtime system. Add those existing interfaces to the `*Dep.m4` file. For example:

```
IMPLEMENT_ITF( `CmpIoDrvItf.m4' )

USE_ITF( `CmpIoMgrItf.m4' )

REQUIRED_IMPORTS(
CMRegisterInstance,
IoMgrConfigGetParameter,
IoMgrConfigGetFirstConnector,
IoMgrConfigGetNextConnector,
IoMgrConfigGetFirstChild,
IoMgrConfigGetNextChild,
IoMgrConfigGetParameterValueWord,
IoMgrConfigSetDiagnosis,
IoMgrConfigResetDiagnosis,
IoMgrSetDriverProperties,
IoDrvCreate,
IoDrvDelete,
IoMgrCopyInputLE,
IoMgrCopyOutputLE)
```

- These functions, included in the template, are needed to implement the I/O Driver interface:
 - a. `IoDrvCreate`
 - b. `IoDrvDelete`
 - c. `IoDrvGetInfo`
 - d. `IoUpdateConfiguration`
 - e. `IoDrvUpdateMapping`
 - f. `IoDrvReadInputs`
 - g. `IoDrvWriteOutputs`
 - h. `IoDrvStartBusCycle`
 - i. `IoDrvScanModules`
 - j. `IoDrvGetModuleDiagnosis`
 - k. `IoDrvIdentify`
 - l. `IoDrvWatchdogTrigger`

Now, your component should compile without errors. In the current state it won't run, because we removed the assignment which made the link between the driver and the physical I/O area. The purpose of this was, that we want to support more than one I/O channel and therefore, we need to make a link between every parameter and the corresponding address in the I/O area.

In the next step, we want to create a device description, which describes the I/O layout for the CODESYS programming system. The templates contain examples of their device descriptions. What you configure here, will be visible to

- your driver, through the download of the CODESYS application, and
- the user when he opens the configuration editor for your device in CODESYS.

For our example we are using the following parameters:

- Vendor Name: 393218 (type: STRING)

- Device Name: 393219 (type: STRING)
- Inputs: 1000 - 1999 (type: DWORD)
- Outputs: 2000 - 2999 (type: DWORD)

As a starting point for the device description, it's possible to use the device description from the „IoDrvTemplate“, „IoDrvSimple“ or „IoDrvVerySimple“. But in the end it shouldn't contain more than two connectors, one with the hostparameter set, defined above.

After this, we need to adapt our device driver to handle those new parameters.

- Add a static array to simulate our I/O area:

```
#define MAX_CHANNELS 128
#define MAX_DRIVERS 1

static unsigned long s_ulyIO[MAX_DRIVERS][MAX_CHANNELS];
```

In the code of the template, only one input and one output channel are configured in IoDrvUpdateConfiguration(). Change this to configure all channels, which are defined in the device description. Save a direct pointer to the I/O area in the field dwDriverSpecific of the parameters, corresponding to the channels. For example:

```
/* inputs */
for(i=0; i < MAX_CHANNELS; i++) {
    pParameter = CAL_IoMgrConfigGetParameter(pConnector, 1000+i);
    if (pParameter != NULL) {
        pParameter->dwDriverSpecific =
            (unsigned long)&s_ulyIO[0][i];
        s_ulyIO[0][i] = i+1;
    }
    else
        break;
}
/* outputs */
for(/* continue */; i < MAX_CHANNELS; i++) {
    pParameter = CAL_IoMgrConfigGetParameter(pConnector, 2000+i);
    if (pParameter != NULL)
        pParameter->dwDriverSpecific =
            (unsigned long)&s_ulyIO[0][i];
    else
        break;
}
```

Change the „pbyDeviceAddress“ in IoDrvReadInputs() and IoDrvWriteOutputs() from „pInfo->hSpecific“ to „pConnectorMapList[i].pChannelMapList[j].pParameter->dwDriverSpecific“. Because this is the value, where we saved our I/O address in IoDrvUpdateConfiguration in the previous step.

Change History

Version	Description	Author	Date
0.1	Issued, review concerning contents	IH / Review:BS	08.12.2009
1.0	Release (after formal review)	MN	23.06.2010
1.1	CDS-24354 "-DMIXED_LINK" instead of "-DSTATIC_LINK"	MN	10.01.2011
1.1	Reviewed	IH	11.01.2012
1.2	CDS-29303 + spelling corrections	MN	17.09.2012
2.0	Release	MN	03.12.2012
3.0	Applying new template templ_tecdoc_en_V1.0.docx Release after formal review	MaH	03.02.2014
3.1	Reworked, added notes	TZ	01.09.2015
3.2	Reworked	TZ	22.03.2016
3.3	Added notes according review of StR	TZ	14.04.2016
3.4	Small corrections Appendix B Visual Studio 6 changed to Visual Studio 2015	AH	25.05.2016
4.0	Release	AH	30.05.2016
4.1	Added legal note	TZ	23.08.2016
5.0	Release after formal review	MaH	23.08.2016
5.1	Legal note removed	GeH	24.10.2016
6.0	Release after formal review	MN	28.11.2016