Tutorial:
Creating own Runtime System Components and I/O Drivers.

Document Version 2.0

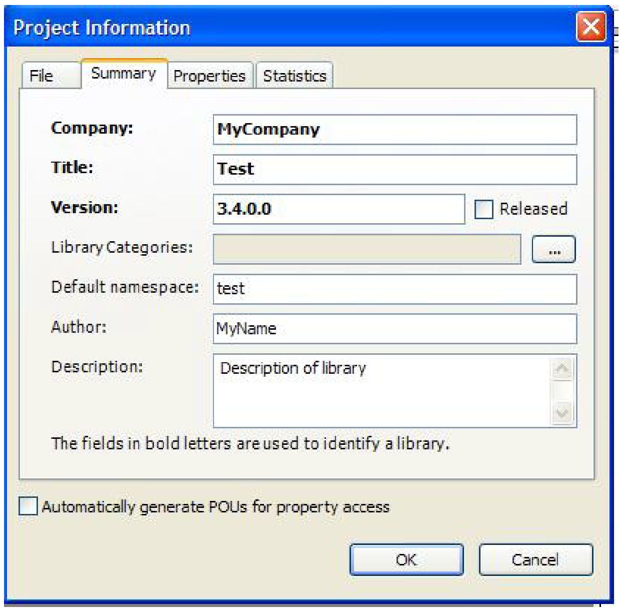
tech_doc_e.doc / V1.2

CONTENT

1 Create a CODESYS IEC Library

We are creating an IEC library, which declares an external function, that has to be implemented inside of the runtime. In the next chapter we will go to the runtime part and create a component that implements this function.

1. Create a new empty CODESYS library from the Menu: *File -> New Project*
2. Open from the Menu: *Project -> Project information...*, which will add a new object to your POUs. Fill out the Information:



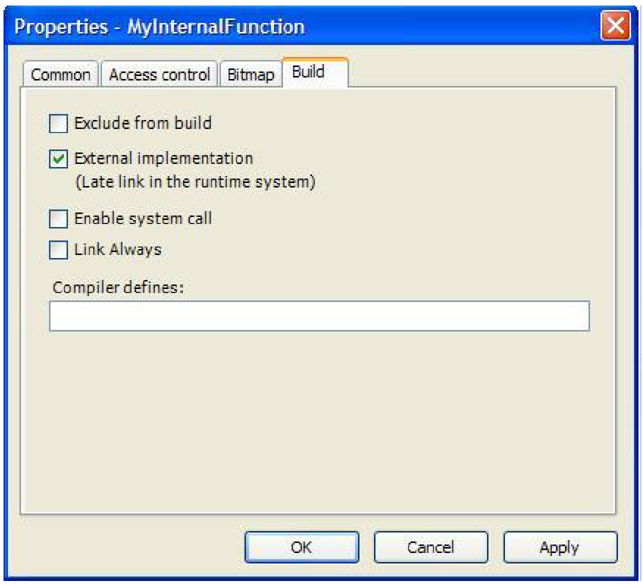
Project Information

Company:	**MyCompany**
Title:	**Test**
Version:	**3.4.0.0** ☐ Released
Library Categories:	
Default namespace:	test
Author:	MyName
Description:	Description of library

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

☐ Automatically generate POUs for property access

3. Add your function and function blocks to the POU pool. Every CODESYS V3 library can have internal and external 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.



Properties - MyInternalFunction

☐ Exclude from build
☑ External implementation
(Late link in the runtime system)
☐ Enable system call
☐ Link Always
Compiler defines:

4. If the "Check all Pool Objects" check does not have 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, you can use this button to save and install your modified library. The library is then automatically reloaded in the second instance.

5. Generate and save the runtime system files 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 you can create a new project, add the new library to your application and call the internal and external functions and function blocks form the IEC code. If you download the application to a runtime system, you will receive errors because of unresolved external references. Unresolved external references appear, when there are calls to external POUs (must be implemented in the runtime), which are not implemented in the runtime. To solve these errors, you have to implement the POUs in the runtime system in your own component.

2 Create a Runtime System Component

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

1. Copy the whole CmpTemplate folder and rename the folder to the new component name.
2. For our example, you should delete all files from your component folder, except:
   CmpTemplateEmpty.c
   CmpTemplateDep.m4
   CmpTemplateDep_m4.bat
   CmpTemplateItf_m4.bat.
3. Copy the *Itf.m4 file, which you have generated with CODESYS to the folder of your new component.
4. Rename all files to the new component name (e.g. CmpTest):
   CmpTemplateEmpty.c -> CmpTest.c
   CmpTemplateDep.m4 -> CmpTestDep.m4
   CmpTemplateDep_m4.bat -> CmpTestDep_m4.bat
   <LibName>Itf.m4 -> CmpTestItf.m4
   CmpTemplateItf_m4.bat -> CmpTestItf_m4.bat
5. 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 a 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.
6. 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.
7. 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.
8. 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 0x0000 is the 3S vendor Id.
   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 an other 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.
9. Our Example may 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)
```

10. 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 **/
```

11. 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)

12. Add an implementation for your external function 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:
    ```
    DEF_API(`void',`CDECL',`myexternalfunction',
    `(myexternalfunction_struct *p)',1,0xFFADC096,0x1000000)
    ```
    b. That 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 */
    }
    ```

3 Compile a new Runtime System Component

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

1. 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 you 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 6:
      i. File -> New
      ii. Change to „Projects“ Tab
      iii. Select „Win32 Dynamic-Link Library
      iv. Finish the Wizard
      v. Change to „FileView“ Tab
      vi. *Right click on „Source Files“ -> Add Files to Folder*
      vii. Select you source file (CmpTest.c)
      viii. *Right click on „Header Files“ -> Add Files to Folder*
      ix. Select your header files (*Itf.h and *Dep.h) and both M4 files
2. 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

tech_doc_e.doc / V1.2

            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 6:
      i. *Project -> Settings*
      ii. Change to „C/C++“ Tab
      iii. Select Category „Preprocessor“
      iv. Add the following paths, comma seperated to „Additional include directories“:
         1. <path-to-runtime-interfaces>/Components
         2. <path-to-runtime-interfaces>/Platforms/Windows

You should now be able to compile your runtime component. How you are loading it, hardly depends on your Platform. You should check if your component is loaded by checking the output of the Logger in CODESYS or the output of the Runtime at startup on the terminal (if possible).

If your component is correctly loaded, you should now be able to create a project, which is using your IEC library with the external function and call this function. The function call will be forwarded to your new runtime component.

4 Create an I/O Driver in C

A CODESYS I/O Driver in C is in general just another component. So we are using the component from the previous chapter as a starting point for our I/O driver. Interfaces which we need additionally for our I/O driver:

- 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 new Interfaces in our component:

- 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 only a small interface, which allows the I/O Manager to read parameters from us. This means that we can manipulate our actual parameters as needed, before they are passed to the I/O Manager.

You should add those definitions to the *Dep.m4 file. For example:

```
IMPLEMENT_ITF(`CmpTestItf.m4',`CmpIoDrvItf.m4',`CmpIoDrvParameterItf.m4')

USE_ITF(`CMItf.m4')
USE_ITF(`CmpIoMgrItf.m4')
USE_ITF(`SysMemItf.m4')
USE_ITF(`SysCpuHandlingItf.m4')

REQUIRED_IMPORTS(
CMRegisterInstance,
IoMgrConfigGetParameter,
IoMgrConfigGetFirstConnector,
IoMgrConfigGetNextConnector,
IoMgrConfigGetFirstChild,
IoMgrConfigGetNextChild,
IoMgrConfigGetParameterValueWord,
IoMgrConfigSetDiagnosis,
IoMgrConfigResetDiagnosis,
IoMgrSetDriverProperties,
IoDrvCreate,
IoDrvDelete,
SysMemAllocData,
SysMemFreeData,
SysCpuTestAndSet,
SysCpuTestAndReset)
```

For the following steps, we will need the component as it was created in the previous chapter as well as the IoDrvTemplate as it is inside of our runtime delivery.

1. Copy the following functions from the I/O Driver Template to your component „CmpTest.c“:
    a. IoDrvCreate
    b. IoDrvDelete
    c. IoDrvGetInfo
    d. IoUpdateConfiguration
    e. IoDrvUpdateMapping
    f. IoDrvReadInputs
    g. IoDrvWriteOutputs
    h. IoDrvStartBusCycle
    i. IoDrvScanModules
    j. IoDrvGetModuleDiagnosis

   k. IoDrvIdentify
   l. IoDrvWatchdogTrigger
   m. IoDrvReadParameter
   n. IoDrvWriteParameter

2. Copy the following definitions from I/O Driver Template to your component „CmpTest.c“:
```
typedef struct
{
#ifndef CPLUSPLUS
      IBase_C Base;
      ICmpIoDrv_C IoDrv;
      ICmpIoDrvParameter_C IoDrvParameter;
#endif
      IBase *pIBase;
      ICmpIoDrv *pIoDrv;
      ICmpIoDrvParameter *pIoDrvParameter;
      IoDrvInfo Info;
} IoDrvTemplate;

#ifndef CPLUSPLUS

DECLARE_ADDREF
DECLARE_RELEASE
DECLARE_QUERYINTERFACE

#endif

static IBase *s_pIBase, *s_pIBase2;
```
3. Copy the following code from CreateInstance() to your new component „CmpTest.c“:
```
#else
      if (cid == CLASSID_CIoDrvTemplate)
      {
            IoDrvTemplate *pIoDrvTemplate = (IoDrvTemplate
)CAL_SysMemAllocData(COMPONENT_NAME, sizeof(IoDrvTemplate), NULL);
            IBase *pI = &pIoDrvTemplate->Base;
            pIoDrvTemplate->pIBase = &pIoDrvTemplate->Base;
            pI->bIEC = 0;
            pI->AddRef = AddRef;
            pI->Release = Release;
            pI->QueryInterface = QueryInterface;
            pI->iRefCount++;
            return pI;
      }
```
4. Copy the following code from DeleteInstance() to your new component „CmpTest.c“:
```
#else
      pIBase->iRefCount--;
      CAL_SysMemFreeData(COMPONENT_NAME, pIBase);
```
5. Uncomment the call to pInitStruct->pfCMRegisterClass() in Component Entry
6. Copy the function QueryInterface(), including the preprocessor definitions IMPLEMENT_ADDREF and IMPLEMENT_RELEASE to your new component „CmpTest.c“
7. Add the following hooks to your HookFunction(), to register and unregister your driver:
```
case CH_INIT:
{
        RTS_HANDLE hIoDrv = 0;
        void *pTmp;
        /* first instance */
        pTmp = CAL_IoDrvCreate(hIoDrv, CLASSID_CCmpTest, 0, NULL);
        s_pIBase = (IBase *)pTmp;
        CAL_CMRegisterInstance(CLASSID_CCmpTest, 0, s_pIBase);
        break;
}
case CH_EXIT:
```

```
{
        /* Delete instance */
        ICmpIoDrv *pI;
        pI = (ICmpIoDrv *)s_pIBase->QueryInterface(
                s_pIBase, ITFID_ICmpIoDrv, NULL);
        s_pIBase->Release(s_pIBase);
        CAL_IoDrvDelete((RTS_HANDLE)pI, (RTS_HANDLE)pI);
        DeleteInstance(s_pIBase);
        break;
}
```

8. Replace all occurrences of „IoDrvTemplate“ with your component name „CmpTest“
9. Remove the assignment of s_byIO[] to hSpecific in IoDrvCreate().

Now, your component should compile again. 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. What you configure here, will (a) be visible to your driver (b) be visible to the user when he opens the configurator 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“. But in the end it shouldn't contain more than two connectors, one with the hostparameter set, defined above.

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

1. 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];
```
2. 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;
}
```

3. 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.

Note: In our deliveries, this driver may be included under the name „IoDrvSimple“.

Change History

Version	Description	Editor	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