CODESYS Control V3
Migration and Adaptation

Document Version 4.0

CONTENT

tech_doc_e.doc / V1.2

1   Preface

This document describes the basic approach to port the CODESYS Control runtime system to a bare hardware without an operating system, as well as to a platform with an unsupported operating system. This is a practical document and does not describe the concepts behind CODESYS Control or I/O Drivers. To learn more about the concepts and special implementation details, please refer to the document *CODESYSControlV3_Manual.pdf* which is delivered with every CODESYS Control SDK.

For systems with proprietary operating systems as well as for small embedded systems, we will start with a very small, stripped down configuration of CODESYS Control which is called "Compact Runtime". This term describes only a minimal set of components with a limited configuration which is necessary for communicating with CODESYS and to run an application on it.

The reason for this is just, that we have the least amount of adaptation work to do in the system layer to get this working. If you need more functionality, you can configure one component after another into the system. This way, the functionality which you need to implement will grow linearly and you can more easily implement and test one after another. This reduces the complexity of this porting task drastically.

1.1   Version Control System

We strongly recommend you to use some kind of Version Control System, like "Subversion", "Clear Case" or "SourceSafe" to maintain the changes to runtime system. It will help you to keep track of your own changes in the beginning of the project and it will essentially support you if you wan to update your runtime to a newer version at a later time.

Within this document, we will give you some hints on where you should check in the current state of the sources. If possible with your version control system you should even tag those versions, that it's easier to find those revisions later.

But if you follow our Coding Guidelines, you don't need to care too much about those tags. Because then, you just need to copy the new files from your new CODESYS Control starter package over the existing files and see the differences in the "SysTemplates" folder. Then you just need to make all those adaptations to your own platform files and you are fine.

1.2   Content of the Starter Package

The base for this porting is a Starter Package of CODESYS Control for an "embedded" or "customer specific" platform. This Starter Package contains all sources of the CODESYS Control runtime system. It contains no platform dependant adaptation layer for your or any other CPU or platform.

For some platforms, 3S – Smart Software Solutions provides reference implementations. For those it is possible to receive a package from your first level support. He can give you the most current files which should fit your environment and requirements best.

1.3   Reference Implementations

In some cases, you will receive a reference implementation for your CPU or for a specific Evaluation Board from your first level support of 3S – Smart Software Solutions. Even in those cases, we recommend, that you work yourself through this guide and just take this reference implementation really as what it is: Just a reference.

Especially you should not assume that those implementations are error-free or that they will work for your specific application. Their aim is just to help you in the initial porting.

2 Porting to a new Platform

This chapter describes the steps, starting from a plain delivery of a CODESYS Control Starter Package, to a Runtime, which you are able to login to with CODESYS. For simplicity, we start with a Compact Runtime here. That means, that we strip out some advanced functionality from the existing components by setting the pre-processor define „RTS_COMPACT" and if there is an „Embedded-Version" of a component, we favour this one ahead the full-featured version. This way, we have the smallest possible Runtime and we need to implement the smallest amount of system components for the beginning.

If you need more features for your runtime, please check chapter 3 of this document, to find out how to add some of the most essential components to your runtime system.

2.1 Step by Step

To follow this step-by-step tutorial, we assume that you have a specific project layout. For simplicity, we call the new Platform „MyPlat". So you should replace this name everywhere with the name of your specific CPU-, OS- or Boardname.

We don't want to focus on a specific compiler or a specific platform. So most of the descriptions are kept very neutral in respect to this. Depending on your specific hardware, there might be some more steps involved, which are not mentioned here.

The expected file layout looks like this:

- **Components**
  Here you will find all generic code of our runtime core. You should never change any of those files for your adaptation. If you really have the need to change something here, you should copy this specific component to your platform directory, rename it and adopt it for your project. **Note:** Renaming is essential, because otherwise the component will be faulty marked as a 3S component!

- **Platforms**
  - **Native**
    - **MyPlat**
      - **rtsconfig**
        This folder contains the configuration of the runtime which we create with our „RtsConfigurator".

      - **Sys**
        This folder will keep all source files, which need to be adopted for our platform.

      - **Projects**
        Here, you should place your project-/makefiles for your specific compiler.

2.1.1 Configure your Runtime, using the RtsConfigurator

The RtsConfigurator is used to manage a collection of Components that build up a CODESYS Control runtime system. In this process, the RtsConfigurator helps you in resolving the dependencies between the components dynamically. You can select the components in the top left field by categories or in the bottom left field from a sorted list. If the currently selected components have unresolved dependencies, they are displayed on the right. There you can decide, how you want to resolve them.

Note: Optional dependencies and implicit dependencies are not resolved. Such an implicit dependency might be: *CmpBlkDrvCan* needs a *CAN-Mini-Driver* to work. But because this Mini-Driver has registers itself at the CANL2 layer and not the other way, the dependency to the Mini-Driver is not known.

- Before we can start the RtsConfigurator, we have to create the file structure, like it is described in section 2.1:
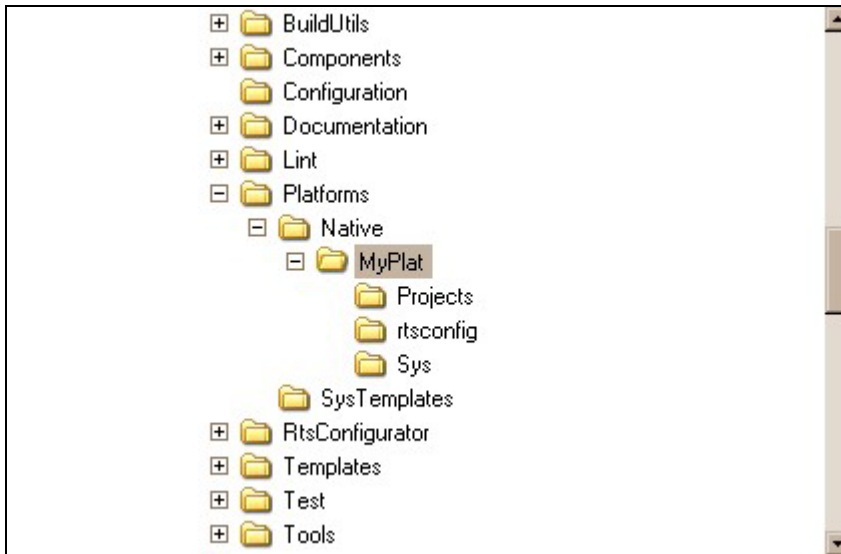
- Start the RtsConfigurator from *RtsConfigurator -> Bin*:

- Select Components Folder:

- Select the System Folder
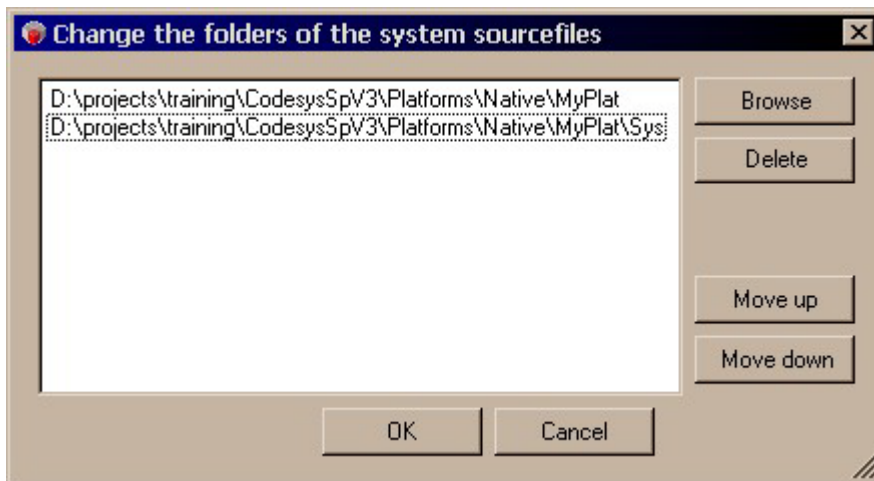
- Set OS Prefix

- Add the following components to your project by selecting them in the bottom left list:

    CmpAppEmbedded
    CmpBinTagUtil
    CmpBlkDrvCom
    CmpChannelMgrEmbedded
    CmpChannelServerEmbedded
    CmpChecksum
    CmpCommunicationLib
    CmpDevice
    CmpEventMgr
    CmpIecTask
    CmpIoMgrEmbedded
    CmpLogEmbedded
    CmpMemPool
    CmpMonitor
    CmpNameServiceServer
    CmpRetain
    CmpRouterEmbedded
    CmpScheduleEmbedded
    CmpSettingsEmbedded
    CmpSrv
    SysCom
    SysCpuHandling
    SysExcept
    SysFileFlash
    SysFlash
    SysInternalLib
    SysMem
    SysTarget
    SysTime

> **Note:** If your Runtime is newer than the version described in this document, you might encounter some new dependencies. Please resolve them manually if necessary.

- After that, you will see the following list of errors in the "Messages" window:
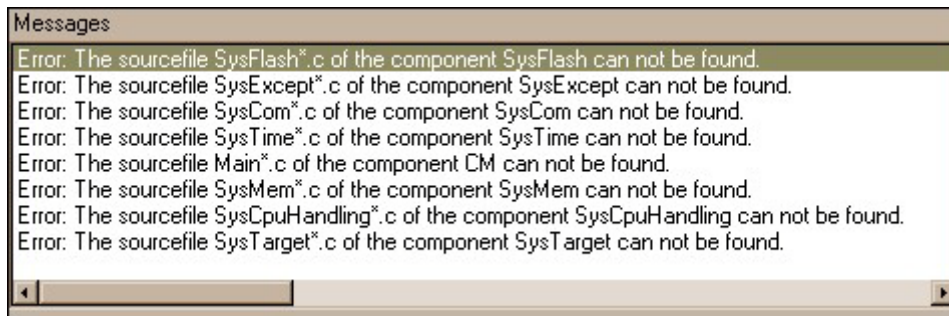
This gives you a list of files, that you need to implement, to adopt the CODESYS Control Runtime with the current settings to your new platform. The adaptation will be done in section 2.1.5 and 2.1.7. Now we will leave the RtsConfigurator open in the background and start copying the required files...

2.1.2 Get all necessary source files from SysTemplates

The Folder *Platform -> SysTemplates* of the StarterPackage contains some examples and scaffolds to build up your own adaptation files. Based on the list printed by the RtsConfigurator in section 2.1.1, you should copy and rename all necessary files from "*Platforms -> SysTemplates*" to "*Platforms -> Native -> MyPlat -> Sys*".

You might notice, that all files in the SysTemplates folder have the postfix "OS". This will be replaced by our own name. In our example this is "MyPlat", but you should use your own description here.

```
SysTemplates/SysFlashOS.c -> Native/MyPlat/Sys/SysFlashMyPlat.c
SysTemplates/SysExceptOS.c -> Native/MyPlat/Sys/SysExceptMyPlat.c
SysTemplates/SysComOS.c -> Native/MyPlat/Sys/SysComMyPlat.c
SysTemplates/SysTimeOS.c -> Native/MyPlat/Sys/SysTimeMyPlat.c
SysTemplates/MainOS.c -> Native/MyPlat/Sys/MainMyPlat.c
SysTemplates/SysMemOS.c -> Native/MyPlat/Sys/SysMemMyPlat.c
SysTemplates/SysCpuHandlingOS.c ->
Native/MyPlat/Sys/SysCpuHandlingMyPlat.c
SysTemplates/SysTargetOS.c -> Native/MyPlat/Sys/SysTargetMyPlat.c
```

Additionally to this, you should also copy some header files from the SysTemplates. We will place those files not to the "Sys"-Folder, but outside, so that they are clearly seperated from the sources. Those header files are containing the configuration of the runtime as well as some platformspecific definitions.

```
SysTemplates/sysspecific.h -> Native/MyPlat/sysspecific.h
SysTemplates/sysdefines.h -> Native/MyPlat/sysdefines.h
```

2.1.3 Finish Configuration with RtsConfigurator

Go back to the RtsConfigurator and open the menu *Options -> Folders of system components* just to close it again. This way, the RtsConfigurator is doing a new scan for the files and the error messages from the "Messages" window should disappear.

Note: If there are still some error messages, please check the that you selected the correct paths and that the "OS Prefix", selected in the *Options -> Project options* is the same as the postfix of the files, which you copied to your folder (postfix = the name with which you substituted the postfix "OS" from the default filename. In our example this is "MyPlat").

Now you can save the project file in the following path:
*Platforms -> Native -> MyPlat -> rtsconfig*

We recommend to name it **"compact.rcp"**, because this is the configuration of our compact runtime. This name will be used to generate some header files, which we are including later.

Now, that we saved everything, we can also change the paths in our RtsConfigurator Project from absolute to relative:

- *Options -> Component folders*

- *Options -> Folders for system components*
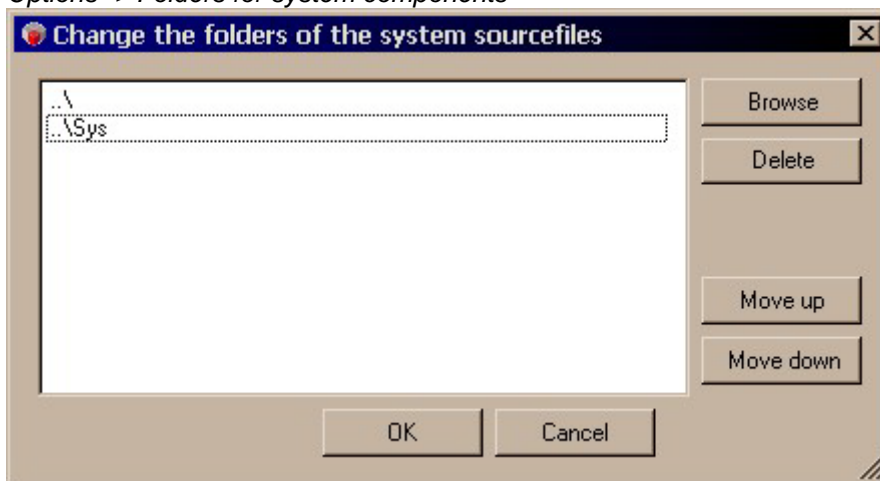
- Save again!

Note: This is a good time to check everything in into your version control system. Later on I will only refer to files, which need to be added explicitly, based on this version.

- After you saved the project, you can generate the output files:
  Output -> Generate all output files

- The most important files for us are the following:
  *compact.c_*
  compact.h
  compact_NotImpl.h

2.1.4 Create Project to Compile

Now you should create a new project, using your compiler and maybe the IDE which comes with it. How this works exactly depends heavily on your compiler and is not covered here. But there are a few settings, which will be necessary in any way:

- Save the Project under: *Platforms -> Native -> MyPlat -> Projects*

- Set the Include Paths to (if possible with relative paths):
  Components
  Platforms -> Native -> MyPlat

- On systems with less then 32Bit address width, you should set your memory model to s.th. like "huge" or "large" (please check your compiler manual). Because we might need to address buffers, which might be more far away as the segment size of your processor.

- Add all C-Files, which are listed in the file *compact.c_* to your project.

If you now try to compile, it depends on your toolchain if it works or not. But in general you should encounter only some small issues, like unnecessary include directives. Just remove them as necessary.

2.1.5 Adapting all files for communication

2.1.5.1 MainMyPlat.c

This file should contain your "main" entry point. On an embedded platform, you will usually have another processor entry in which you need to setup some of the basic peripherals or memory mappings. It's up to you if you integrate this to this file or not. We will assume that your entry begins in a C-Function, called "main()".

At the top of this file, there is an include statement for *"myPlatform.h"*. You need to change this to *"rtsconfig/compact.h"*. This file contains basically a list of all configured components, which we will pass to CMInit() to initialise our system.

For now we will not load the configuration settings from a file, but we will configure the system statically (see 2.1.5.5). So we will pass NULL as the name of the configuration file. The minimal version of our main() function will look like this:

```
int main() {
    RTS_RESULT Result;

    Result = CMInit(NULL, s_ComponentList);

    while(!s_bExitLoop) {
        CMCallHook( CH_COMM_CYCLE, 0, 0, FALSE);
    }
    return 0;
}
```

Note: MainLoadComponent() and MainUnloadComponent() can be used for dynamically linked systems, to load new components dynamically at system startup. This is not necessary for our small runtime, because we will link our system statically.

2.1.5.2 SysComMyPlat.c

The communication between CODESYS and CODESYS Control will be done over a serial link. The driver for the serial interface needs to be implemented in this component.

The general driver scheme which is necessary is as follows:

- SysComOpen()
  Just make a wrapper for SysComOpen2().

- SysComOpen2()
  Setup the serial interface with our configuration settings. Especially set the baudrate of the interface.

- SysComClose()
  Disable the serial interface. Especially you should disable the hardware interrupts.

- SysComSetSettings() / SysComGetSettings()
  These functions are not used in our communication layer, so you can leave them blank for now.

- InterruptHandler()
  Register your own interrupt handler for send- and receive interrupts. Use the buffers "sendBuf" and "recvBuf", which are already declared in this file for your send and receive packets.

  o On a receive interrupt: read byte(s) and write it to the "recvBuf".

- o On send interrupt: check if there is something left in the "sendBuf" and write it to the chip.

- SysComRead()
  Read as much as you can from the "recvBuf" and return it to the user. For simplicity it is also possible to return only one byte at a time, but this will produce more system overhead and is only recommended for the first tests.

- SysComWrite()

  - o If "sendBuf" is empty, write first character directly to the chip and fill the rest into the "sendBuf".
  - o If "sendBuf" is full, just append everything to "sendBuf".

Note: Remember to disable interrupts during manipulation of the buffers!

2.1.5.3 SysTargetMyPlat.c

This component returns the target identification, like: TargetID, VendorID, NodeName, Serial Number, ... Most of the parameters are already returned by the higher-level component, and are taken from the file "targetdefines.h", which is part of your delivery (see 2.1.5.5).

In this file, you just have to return the NodeName, which is displayed in the Device Scan Dialog. But for the first tests you can leave it as is.

2.1.5.4 SysTimeMyPlat.c

This component provides some basic functions for the runtime to measure relative timings. The time values returned by the functions from this file have no relations to a realworld time, so for example they can not be converted directly to a fix UTC time. Usually the time is measured since the system startup and that's enough, because the functions are just used to measure relative times.

- SysTimeGetMs()
  Returns a relative timestamp in milliseconds.

- SysTimeGetUs()
  Returns a relative timestamp in microseconds.

- SysTimeGetNs()
  Returns a relative timestamp in nanoseconds.

Depending on your hardware, you can use a free programmable system timer or a timestamp counter to determine this time. Examples for some sources in some architectures:

- X86: Timestamp Counter (TSC):
  With every processor cycle the x86 increments this counter by one. So you get the number of processor cycles since system startup.

- PPC: Timebase register (TB):
  This counter is incremented with the frequency of the bus. You need to check your hardware manual to determine the exact frequency.

- ARM and others:
  If no such counter is available, you can program a periodic timer to get a high-precision system time. Note, that you don't have to generate interrupts, but in most cases you can just read the current value of the timer. That's much more efficient then generating permanently interrupts and counting these.

2.1.5.5 sysdefines.h / sysspecific.h

For the Platformspecific configuration, we differentiate between those two include files. Both files are included in every source file, which is built in the CODESYS Control Runtime. But anyway, they are both playing different roles in the configuration:

- sysdefines.h
  This file contains and includes everything which belongs to the configuration of the runtime system. In the end it should really be a collection of defines and configuration entries, which are configuring the system.

- sysspecific.h
  In contrary to sysdefines.h this file should only define some compiler-, OS- or CPU-specific things which are necessary on this platform. For example: HUGE_PTR, CDECL, ...

Because the **sysdefines.h** contains the configuration of the system, we also include the header file, generated by the RtsConfigurator here. You should add an include statement for "rtsconfig/compact_NotImpl.h":

```
#include <rtsconfig/compact_NotImpl.h>
```

This file is important, because it defines which components are not contained in our system. Some of our runtime system components have optional dependencies. And to disable those dependencies in a statically linked application, we need to set those defines.

You should also add a include statement for the file "targetdefines.h", which was included in our SDK. This file contains your VendorID, TargetID, Target Signature, ... Just copy it to *Platforms -> Native -> MyPlat*:

```
#include <targetdefines.h>
```

Because we have no Filesystem, yet, we cannot load any configuration file. Therefore we included the component CmpSettingsEmbedded instead of CmpSettings in our configuration of the compact runtime. The full component CmpSettings is reading and writing the configuration entries from and to a file on the filesystem. The embedded version of this component implements the exact same interface but uses the configuration settings which are set in sysdefines.h:

```
/**
 * Defines, used only in CmpSettingsEmbedded.
 * Defines the configuration settings, as well as free spaces,
 * to add new settings.
 * The format is:
 * "ComponentName", "KeyName", Value
 * If the "KeyName" is 0, this slot is reserved for new keys,
 * which might be added to this component.
 */
#define SETTG_ENTRIES_INT \
      {"CmpRouter", "NumRouters", 1}, \
      {"CmpBlkDrvCom", "Com.0.Port", 1}, \
      {"CmpBlkDrvCom", "Com.0.Baudrate", 115200}, \
      {"CmpMy", 0, 0}, \
      {0, 0, 0}
#define SETTG_ENTRIES_STRING \
      {"CmpRouter", "0.MainNet", "MyCom"}, \
      {"CmpBlkDrvCom", "Com.0.Name", "MyCom"}, \
      {"CmpMy", 0, 0}, \
      {0, 0, 0}
```

2.1.5.6 Gateway.cfg

This is a configuration file on your development host. It is used to configure the CODESYS Gateway. Specifically we want to add our serial port as a new communication interface to it.

1. First you should find out which serial port your target is connected to. If you have a target that's connected using a USB link, you may want to check the device manager of Windows to find out which com port is assigned to your device.

2. Got to your *CODESYS installation directory -> GatewayPLC* and open the file gateway.cfg.

3. Add the component "CmpBlkDrvCom" to the list of components:
   ```
   [ComponentManager]
   Component.1=CmpBlkDrvCom
   ```

4. Add a new Router instance:
   ```
   [CmpRouter]
   1.MainNet=MyCom
   ```

5. Configure the serial port:
   ```
   [CmpBlkDrvCom]
   Com.0.Port=19
   Com.0.Name=MyCom
   ```

```
Com.0.Baudrate=115200
Com.0.EnableAutoAddressing=1
```

Please restart your Gateway (using the systray icon 🔴).

To check if the communication works, you can use the tool "portmon" from sysinternals to monitor the traffic on your serial port. After you made the configuration settings above, you should see some packets sent by the Gateway if you try to scan the network with CODESYS.

2.1.6 Test the communication

After you made all the adaptations above, you should already see the device in the scan dialog:
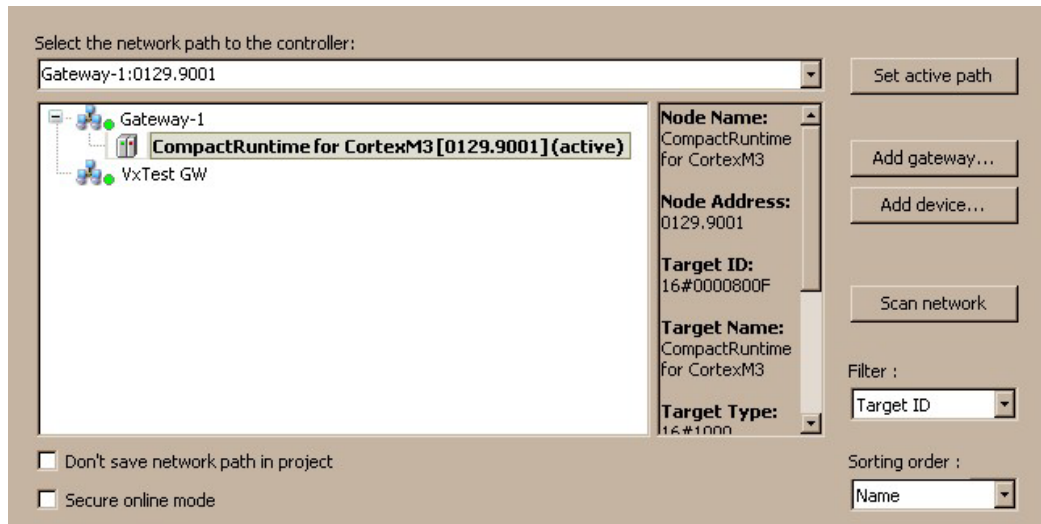
If not, check the section 2.3.2 for some tipps how to debug the problem.

2.1.7 Adapt all files for a first download

2.1.7.1 MyPlat.devdesc.xml

You should create a so called "device description" for your device. As a starting point, you should check out the files in *Templates -> Devices* of your CODESYS Control SDK. There you will find templates for various architectures and various configurations. Go to the folder of your target CPU and take one of the device descriptions with the prefix "Compact_" as a starting point.

The device description should already have the correct compiler settings. For the first tests, you should only change the device name:

- *DeviceDescription -> Device -> DeviceIdentification*

    o **Type:** The default is 4096 and this is suitable for every logical PLC.

    o **Id:** This is a combination of your VendorID and your TargetID. This should already match your settings.

    o **Version:** The version can be used by you to force the user to use a device description that matches your Firmware.

- *DeviceDescription -> Device -> DeviceInfo*

    o **Name:** The name of your PLC when you select it in CODESYS.

    o **Description:** Description which is displayed when you select your PLC.

    o **Vendor:** Your Vendor Name

    o **Icon:** Reference to an Icon that is displayed in the device tree (*.ico, 32x32).

    o **Image:** Image, which is displayed near the description, when you select your PLC in CODESYS (*.png, *.jpg, full-sized image, auto-scaled).

To select your PLC in CODESYS, you need to install it first. Go to *Tools -> Device Repository -> Install* and select your Device Description. After you pressed "OK", you should find your newly installed device with the name that you specified in the device description in the device list.

2.1.7.2 SysCpuHandlingMyPlat.c

This component contains some CPU-specific low level functionality. It usually consists of much assembler code, which is dependant on the CPU and the C-Compiler. Before you start implementing this functionality from scratch, you should contact your first level support at *3S – Smart Software Solutions* to get an example for your CPU. Even if it doesn't match the syntax of your Assembler it should be easier to port it than to start from scratch.

If your platform is too different and you need to start from scratch, you should start with the function *SysCpuCallIecFuncWithParams()* and implement it similar to this:

```
int CDECL SysCpuCallIecFuncWithParams(void* pfIECFunc, void* pParam,
int iSize)
{
      void (*pfFunc)();
      void *pStack;
      int iStackSize = (iStackSize + 3) & ~3;
      pfFunc = pfIECFunc;
      fun = (PFFUNCWITHPARAMS)pfIECFunc;

      __asm {
                  sub SP, iStackSize
                  mov pStack, SP;
      }
      memcpy(pStack, pParam, iSize);
      pfFunc();
      memcpy(pParam, pStack, iSize);
      __asm {
                  add SP, iStackSize
      }
      return ERR_OK;
}
```

Note, that this code only works, if your compiler doesn't place the local variables on the stack, but keeps them in callee save registers.

The function *SysCpuGetCallstackEntry()* and is used to unwind the callstack when debugging or in case of an exception. This code is pretty similar on most platforms, as most callstacks are just saved in form of linked lists on the stack. The function *SysCPUGetCallstackEntry2()* is just an extended version, which differenciates between a C and an IEC context. That's only important if your C-Compiler uses a different stack layout as the IEC Compiler of CODESYS. The unwinding code should look something like this:

```
typedef struct stack {
    struct stack *next;
    void *eip;
} stack_t;
RTS_RESULT CDECL SysCpuGetCallstackEntry(unsigned long *pulBP, void
**ppAddress)
{
    stack_t *tStackEntry = (stack_t*)pulBP;

    if (pulBP == NULL || *pulBP == 0 || ppAddress == NULL)

          if (ppAddress != NULL)
                *ppAddress = NULL;
          return ERR_PARAMETER;
    }
    if(    !CAL_SysMemIsValidPointer((void *)&tStackEntry,4, 0) ||
           !CAL_SysMemIsValidPointer((void *)tStackEntry->next,4, 0) ||
           !CAL_SysMemIsValidPointer((void *)tStackEntry->next->eip,4, 0))
    {
          *ppAddress = 0;
          return ERR_FAILED;
    }
```

```
    *ppAddress = tStackEntry->next->eip;
    *pulBP = tStackEntry->next->next;

    return ERR_OK;
}
```

There are two additional functions, which are used for atomic access of bits. These are called *SysCpuTestAndSet()* and *SysCpuTestAndReset()*. On nearly every CPU architecture, it should be possible to do an atomic read-modify-write operation or even an atomic bit-operation. If it is possible on your CPU, you should implement this API function accordingly. If it is not possible, or you just want to do this later, you can use the base implementation, which is just doing an interrupt lock around the read-modify-write operation:

```
RTS_RESULT CDECL SysCpuTestAndSet(unsigned long* pul, int iBit)
{
    return SysCpuTestAndSetBase(pul, iBit);
}

RTS_RESULT CDECL SysCpuTestAndReset(unsigned long* pul, int iBit)
{
    return SysCpuTestAndResetBase(pul, iBit);
}
```

All other functions are only for debugging. They are discussed in section 3.3.1.

2.1.7.3   SysExceptMyPlat.c

Beside the two install and exit functions, this API contains only one API function which you need to install, that is called *SysExceptMapException()*. This function simply maps the exception number, which is reported by your CPU to the Exception IDs which are known by CODESYS. For a list of the available exception IDs, have a look in *SysExceptItf.m4*. There is a list of defines, starting with RTSEXCEPT_*.

To handle an exception, you need to install an exception handler on your hardware. How this works depends pretty much on the hardware or on the fact if an OS is used or not. But in general, you should try to catch the exception as soon as possible.

After you caught the exception, you need to be able to get the cause of the exception. The cause of the exception will be later expressed by an exception code, which is a 32Bit number. Additionally you have to determine the code location where the exception occurred and the base-pointer to unwind the call-stack later.

For example – if the exception state was pushed on the stack, your code may look something like this:

```
void SysExceptHandler()
{
    RegContext reg;
    unsigned long *pStack;
    void *pSP;
    void *pBP;
    __asm {

        mov pStack, SP;
        mov pSP, SP;
        mov pBP, BP;
    }
    reg.IP = pStack[EXC_IP_IDX];
    reg.SP = pSP;
    reg.BP = pBP;
    CAL_SysExceptGenerateException(
        0, pStack[EXC_TYPE_IDX], reg);
}
```

2.1.7.4   SysMemMyPlat.c

SysMem provides an API for the runtime and the IEC Application to allocate or free two different kinds of data:

1. **Data:** This is used to manage general heap data of the runtime and the application.

2. **Areas:** The areas are used to store the code, data, retain data and I/Os of the IEC Application.

On most systems your C-Library will provide some malloc and free functions for you. If so, you should use them to implement *SysMemAllocData()*, *SysMemReallocData()* and *SysMemFreeData()*.

If you like, you can place the areas of your IEC Application at a specific address. For this, you need to tell your linker that this memory range is reserved:

- Keil uVision: *"Options for ..." -> "LXXX Misc" -> Reserve*

- TASKING.VX-Toolset: Add a new memory section to <projectname>.lsl and mark it as "Reserved".

- GNU Linker: Reserve Memory area in the linker script.

If you made sure that your C compiler doesn't use the memory area anymore, you can just return the fix address in SysMemAllocArea(). In SysMemFreeArea() you can either zero the whole area, or you can just do nothing.

Alternatively, you can just forward the call to SysMemAllocData() and SysMemFreeData():

```
void* CDECL SysMemAllocArea(char *pszComponentName, unsigned short
usType, unsigned long ulSize, RTS_RESULT *pResult)
{
    return SysMemAllocData(pszComponentName, ulSize, pResult);
}

RTS_RESULT CDECL SysMemFreeArea(char *pszComponentName, void* pCode)
{
    return SysMemFreeData(pszComponentName, pCode);
}
```

As long as the CODESYS Codegenerator supports code relocation for your target, you can leave *SysMemAllocCode()* and *SysMemFreeCode()* at the default implementation, which falls back to *SysMemAllocData()* and *SysMemFreeData()*. If it doesn't, you need to use a fix address in the Runtime as well as a fix start-address for the areas in your Device Description. So if your target doesn't support relocation, go to your device description and check that it has a statement like that:

```
<ts:setting name="start-address" type="integer" access="visible"
xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-0.1.xsd">
    <ts:value>0xE04000</ts:value>
</ts:setting>
```
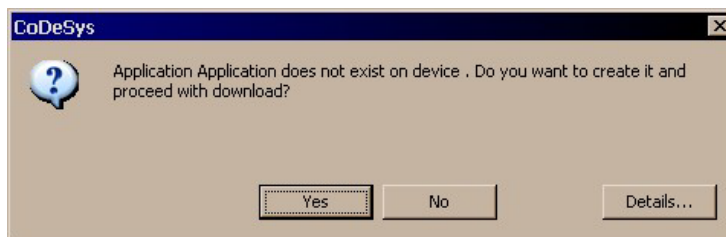
2.2 Testing the communication

- To test the communication basically, you should start with a very simple project with a cyclic counter:

CODESYSControlV3_Adaptation.doc
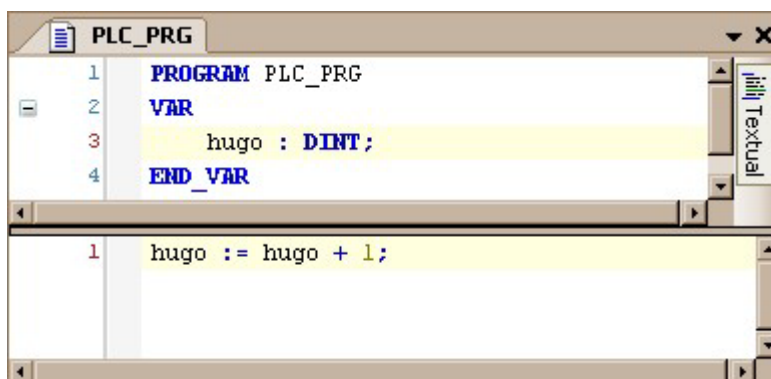
- Double click on the device in your device tree, and open the communication settings:
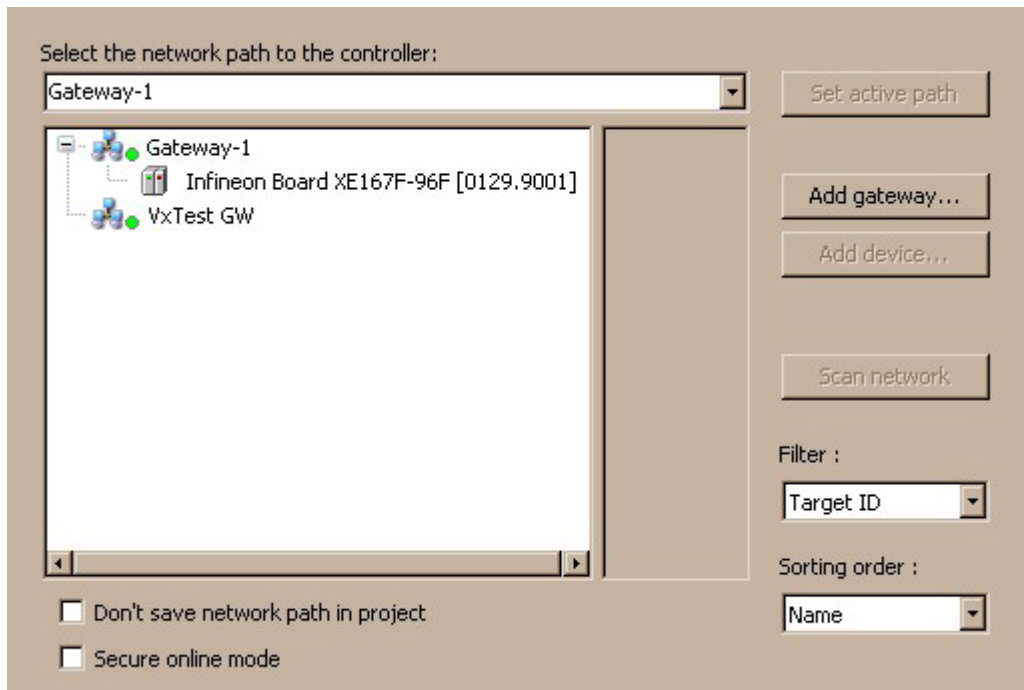

- Set the "active path" to your PLC and try to log in. You should see the following dialog box:


Note: If that dialog box doesn't appear, you will most likely have an issue in SysTimeMyPlat.c: SysTimeGetMs(). Because this function is used to detect communication timeouts and if the timebase is wrong, this leads to such communication errors.

- If you say "Yes" in the previous dialog, you should see a progress bar in the status bar:

Sending download info: Downloading ... (15 of 15 KByte (99%))
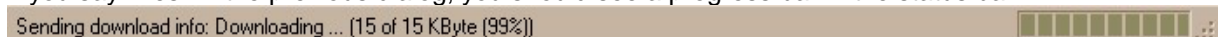
- If everything worked fine, your project should now be successfully downloaded and your program should be in stop:

| STOP | Program loaded | Program unchanged | Current user: (nobody) |
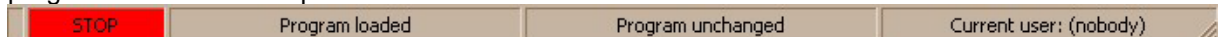
- If it didn' work:

  o You get an error message **"Create Bootproject failed"**:
    Try to logout and login again. If this works, you just have a problem with your flash driver. The PLC was unable to save the bootproject, but the application should run fine anyway.

  o You get an error message **"Communication Timeout"**:
    Most likely your PLC crashed during download. You should attach your debugger, try again and have a look into section 2.3.
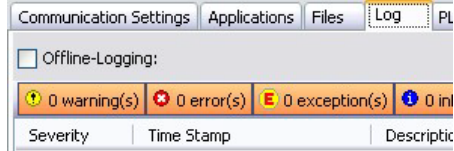
2.3 Debugging

2.3.1 Check the log messages

You should have the component CmpLogEmbedded in your runtime. This component is used by other components as well as your IEC application to collect all kinds of debug, info and error messages. There are three ways to see those logs:

- **Logger page:**
  Double click on your device, switch to the "Log"-tab and press the update button.


  Note, that this only works if your communication with the runtime works.

- **Serial Terminal:**
  If you have a spare serial port, you can add and implement the component "SysOut" to output the messages to your serial port.

- **Debugger:**
  If you don't have a spare serial port nor a working communication, you can use your debugger: Find out the address of the symbol "s_StdLogEntries" and add this address to the memory window of your debugger. If you adjust the number of displayed entries per line to LOG_MAX_INFO_LEN+20, you can see a readable log, like this:
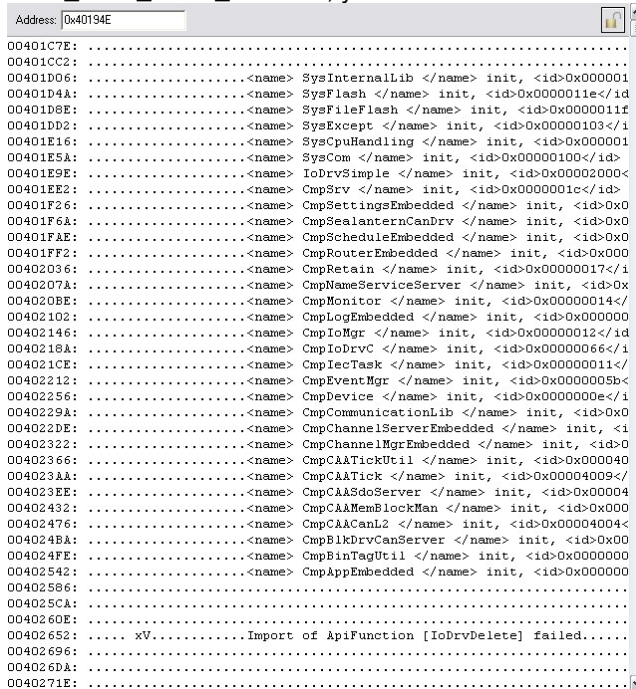
2.3.2 Device doesn't appear in the scan

- Before we start debugging the problem. We should check our configuration:

  o Gateway.cfg (check the configuration of your COM port, as described in 2.1.5.6.

  o Restart your Gateway (using the systray icon 🔴) and check if it sends s.th. on the serial port when you press the "Scan network" button. You can check the traffic with the tool "portmon" from sysinternals.

  o Check your configuration in "sysdefines.h". Check if you are using the correct device and the correct baudrate.

- o Check if your driver supports the configured baudrate and if you are using the correct serial port.

- Before we are debugging the driver on the Runtime side, we should start monitoring the packets, which are sent over the bus. Please start "portmon" from sysinternals, stop the CODESYS Gateway and connect portmon to your serial port. After a restart of the Gateway you should already see some packets on the bus. That means, that your host is configured correctly.

- If your serial driver is working (which we assume here), then it's only possible, that one of the components is not configured correctly and discards the packet. In the case of a device scan, we are talking directly to the component CmpDevice. So we should check with breakpoints in various positions if the packet came through:

  - o **SysComMyPlat.c: SysComOpen()**
    Step through this function and check if the correct port is opened and if you configured the hardware correctly.

  - o **CmpRouter.c: HandleLocally()**
    Set a breakpoint into the „default" branch and check in s_protocolHandler[] if the handler, which we try to call is registered there.
    If you see that the handler for the scan request is not registered, you should check if CmpDevice is configured into your runtime and if it is loaded correctly. If you can't find the reason in the configuration, you should try to debug through the init code of CmpDevice. Just set a breakpoint into CmpDevice.c: ComponentEntry() to see if the component is loaded and configured.

  - o **CmpDeviceSrv.c: DeviceServiceHandler()**
    If the scan request is already this far, everything looks fine for the request and you will need to debug the response path. There is no other way than stepping through the whole response path and have a look on all error conditions. You need to find out, why the request was accepted but the response was never sent.

2.3.3 Unable to download but no exception

If you are using the CmpAppEmbedded and you are not able to login, you most likely miss a setting in your device description to enable the compact download, which is used by CmpAppEmbedded >= V3.4.1.0. This is an optimization, which is done for small embedded devices and changes the download format.

```
<DeviceDescription>
    <Device>
        <ExtendedSettings>
            <ts:TargetSettings>
                <ts:section name="runtime_features">
<ts:setting name="compact_download" type="boolean" access="visible"
xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-
0.1.xsd"><ts:value>1</ts:value></ts:setting>
```

Note: This setting has no influence on the format of your bootproject.

2.3.4 Exception at download

Your device already appears in the scan and you are able to perform a login. For the first test, you should create an empty application and try to download this.

When you log in for the first time you should get a pop up dialog that informs you that no application is on the device and asks you if you want to perform a download. If this doesn't appear, your communication is not working, yet and you should have a look into section 2.3.2.

If you get an exception at download, this exception will most likely appear within our generated init code. But anyway this may have a few different causes:

- **Compiler Settings:**
  First of all, you should double check your compiler settings in your device descriptions. These

should exactly match your device. Take care about CPU, C-Compiler specific settings (only necessary on some architectures), FPU.

- **Memory Layout:**
  Check that the areas that are allocated in SysMemMyPlat.c are really free and correctly allocated. Check how they match to the areas of your device description.
  Note: If you make an "update device" deleted areas may still exist in your project. So please create a new project or add a new PLC to your project if you changed the number of areas in your device description!

- **SysCpuCallIECFuncWithParams**
  Check if your implementation of *SysCpuCallIECFuncWithParams()* works as expected.

To debug this problem, you should start in the function *AppRunAfterDownloadCode()* of CmpApp or CmpAppEmbedded. Set a breakpoint to this function and try to log in into your device. We assume that you hit this breakpoint before the actual exception.

Step through this function and check where it crashes. Most likely it will crash in one of the calls to SysCpuCallIECFuncWithParams. Depending on that we have the following possible error causes:

- pfReloc:

  - *SysCpuCallIECFuncWithParams()* crashes already when trying to call the function pointer:
    On a 16 Bit platform you should check the width of the function pointer. On all other systems you have most likely a wrong area offset specified, or your implementation of SysCpuCallIECFuncWithParams() is not working.

  - The call crashes somewhere in the IEC code:
    That means that the parameters which you passed to the function over the stack, where not correct, that you have the wrong compiler settings (FPU, Compiler Type, Architecture,...) or that the memory area that you returned in *SysMemAllocArea()* was wrong.

  - On architectures with segmented memory, you should check the settings "code-segment-size" and "data-segment-size" in the "memory-layout" of your device description.

- pfCodeInit

- pfGlobalInit

- pfDownloadPOU

  - Check if this function is called and check if the function pointer is valid.

  - This function is called over an indirection of the function call table of our IEC application. So maybe there was already a fault in pfGlobalInit where this table is setted up.

If you can't find the issue by your own, you should send the following information to your first level support contact at 3S – Smart Software Solutions:

- **Call Stack:**
  At least you need to find out the position, where the exception occurred. Also determine the kind of exception and maybe why it occurred (e.g. TLB load exception because of a NULL pointer access).

- **Device Description:**
  You can send either your device description XML file or create a project archive out of CODESYS.

- **Memory Layout:**
  Describe where your memory areas are allocated. It's important to know if you have fix addresses for your memory areas or if you allocate them dynamically.

- **Code Snippets:**
  It will be much of help if you add your implementation of SysCpuCallIECFuncWithParams() to your support request.

tech_doc_e.doc / V1.2

2.3.5 Communication Timeout after a few seconds/minutes

The communication timeout is calculated based on the time returned by SysTimeGetMs(). So when you get a timeout after some time, it is most likely because of a wrong calculation there. A very common error would be that you have too early wrap around in the timer value.

Abbroach to solve this:

- Temporarily set the implementation of SysTimeGetMs() to this:
  ```
  static unsigned long s_ulTimeMs;
  unsigned long CDECL SysTimeGetMs(void)
  {
          return s_ulTimeMs++;
  }
  ```

- Try to implement SysTimeGetUs() correctly.

- Create a small test project like this:
  ```
  PROGRAM PLC_PRG
  VAR
          us: SYSTIME;
          res: UDINT;
          last_us: ULINT;
          start_us: ULINT;
          error: BOOL;
  END_VAR

  res := SysTimeGetUs(pUsTime := start_us);
  res := SysTimeGetUs(pUsTime := us);

  WHILE (us – start_us) < 500 DO
          res := SysTimeGetUs(pUsTime := us);
          IF us < last_us THEN
                  error := TRUE;
          END_IF
          last_us := us;
  END_WHILE
  ```

- Verify your implementation of SysTimeGetUs() with the project above:
  - The time, returned by SysTimeGetUs() should be continuously growing until it wraps around at the boundary of the RTS_SYSTIME datatype.
  - Check with an external clock if the time base of SysTimeGetUs() is correct.

- Make SysTimeGetMs() behave similar to your SysTimeGetUs() implementation.

- Make sure that your intermediate values within your calculation don't exceed the boundary of the RTS_SYSTIME data type.

Another error source is the serial block driver. This might have problems with some combinations of buffer sizes. So check the following definitions in *sysdefines.h*:

- BLKDRVCOM_MAX_SER_READSIZE: Maximum number of bytes that are read with one call of SysComRead. Recommended values are 1..1024.

- NETSERVER_BUFFERSIZE: Keep this value above 1024

2.3.6 Other Errors

Here are a few things which you need to check if you couldn't find a hint for your problem in the sections above:

- Memory allocation at startup failed:
  Set a breakpoint in *SysMemAllocData()* and start the runtime system. You should not hit this breakpoint. If you hit it anyway, you should check why, and try to solve it.
  Known symptoms:
  - Application download just hangs in keep-alive ping pongs and doesn't respond anymore.

o The device doesn't appear in the scan anymore.

CODESYSControlV3_Adaptation.doc

3 Adding Features to the Runtime

One of the big advantages of the version 3.x of the CODESYS Control Runtime in comparison to version 2 is the flexibility. We just started with a Compact Runtime which provides only a very basic scheduler (CmpScheduleEmbedded + CmpAppEmbedded), no I/O Drivers (CmpIoMgrEmbedded), no Routing (CmpRouterEmbedded) and no Debugging Support (no CmpAppBP, CmpAppForce).

But depending on our requirements, we can extend our runtime system very easily by exchanging some components or adding others to our system.

We are using the RtsConfigurator again for this task, because it helps us resolving all necessary dependencies and sets the correct defines in compact.h and compact_NotImpl.h. With every Feature which is described in this chapter, we are starting again with a plain Compact Runtime.

3.1 Change the Scheduler

3.1.1 Embedded Scheduler

This schedule is already the default of the Compact Runtime, so no change is necessary. The scheduling is done without an external timer event. The CPU is just polling for communication events or for an IEC "task" which needs to be executed and executes it if the time has come.

This means:

- As long as an IEC task is running, the communication hangs completely.

- The IEC tasks cannot interrupt each other.

- If one communication cycle takes a bit longer it can produce a jitter for the start of the IEC task.

So in fact this scheduler can handle multiple, periodic IEC tasks but it is non deterministic.

3.1.2 Timer Scheduler

This scheduler fits well on small embedded systems, that provide more than one timer. So it is possible to setup multiple periodic IEC tasks on different timers. Therefore it is possible that a higher priority timer interrupts a lower priority one. Furthermore, the communication is running in a background loop and cannot harm the execution of the IEC tasks.

Additional Components:

- CmpScheduleTimer

Dependencies:

- SysTimer

This feature depends on the new System Interface "SysTimer", which needs to be implemented for your platform. So copy SysTimerOS.c from the SysTemplates to SysTimerMyPlat.c in your adaptation. In generell this interface consists of the following API:

- SysTimerOpen()
  Create a new timer handle, associated with a real system timer.

- SysTimerClose()
  Free timer handle.

- SysTimerGetContext()
  Get current context of one of the timers. This function is called externally, not from within the timer context. [optional]

- SysTimerFitTimer()
  Check if the passed timer handle matches the priority and interval which is also passed to this function. This is used to reuse existing timers for multiple tasks.

tech_doc_e.doc / V1.2

- SysTimerStart()
  Start a specific timer and enable it's interrupts.

- SysTimerStop()
  Stop a specific timer or at least disable the generation of the timer interrupts.

- SysTimerGetTimeStamp()
  You should return the current timer value, if you have a readable counter (incrementer or decrementer). [optional]

- SysTimerSetInterval()
  Change the interval of an existing timer. [optional]

- SysTimerGetInterval()

  Return the configured interval of an existing timer.

- SysTimerMaxTimer()
  Return number of available timers.

- SysTimerGetMinResolution()
  Return the Minimum Resolution of a timer.

- SysTimerRegisterBasePointer()
  This API is obsolete.

- SysTimerDebugLeave()
  This API is obsolete.

- SysTimerDebugEnter()
  This API is obsolete.

3.1.3 Multitasking Scheduler

This is a fully pre-emptive multitasking scheduler. It creates a real system task or thread for every IEC Task using the SysTask component. This also means that you will need some low-level task-/thread handling on your platform.

Additional Components:

- CmpSchedule

Dependencies:

- SysTask

- SysEvent

This feature depends on the two new system interfaces "SysTask" and "SysEvent". The SysTask component assumes that you have a realtime priority based scheduler. The Events are used to schedule the tasks. That means that every IEC task has it's own event on which it is waiting until the scheduler signals the event to wake the task up.

The SysTask interface consists of the following system dependent API:

- SysTaskCreate()
  Create a new task but keep it in suspended state.

- SysTaskExit()
  Terminate a foreign task (no suicide), but give it enough time to end it's cycle.

- SysTaskSetExit()
  Just set the flag TP.bExit in the task handle to true.

- SysTaskDestroy()
  Force the task to end immediately.

- SysTaskSuspend()
  Suspend a running task temporary.

- SysTaskResume()
  Resume a previously suspended task.

- SysTaskEnd()
  This function is for a suicide of the task itself.

- SysTaskWaitSleep()
  This function lets the specified task sleep. It is enough to execute this for the currently running task and to prohibit it for foreign ones (what is easier on most systems).

- SysTaskWaitInterval()
  Let the task sleep for one interval. This is only necessary if you want to bypass the event based scheduler. [optional]

- SysTaskGetCurrent()
  Return the handle for the currently running task.

- SysTaskGetCurrentOSHandle()
  Return the current OS handle. This is necessary for the exception handling.

- SysTaskSetPriority()
  Change the priority of an existing task. [optional]

- SysTaskSetProcessPriority()
  This function is only necessary for systems with interprocess communication. [optional]

- SysTaskGetPriority()
  Get the Tasks IEC priority. [optional]

- SysTaskGetOSPriority()
  Get the real priority of the task. [optional]

- SysTaskGetInfo()
  Return the Task_Info structure from a task handle (usually just a cast).

- SysTaskGetContext()
  Determine PC, SP and BP from the current (saved) task context.

- SysTaskCheckStack()
  Hook to check the task stack. [optional]

- SysTaskGetInterval()
  Get the task interval. [optional]

- SysTaskSetInterval()
  Change the task interval. [optional]

- SysTaskWaitSleepUs()
  Implement a precise sleep in microseconds. [optional]

The Events should have an event counter. That means that if an event is missed, the task will execute two times directly after another. Otherwise we might miss a cycle. You need to implement the following API:

- SysEventCreate()
  Create a new event or take one from a local event pool.

- SysEventDelete()
  Delete an existing event or give it back to a local event pool.

- SysEventSet()
  Signal an event (used by our scheduler, so maybe also from Interrupt context).

- SysEventWait()
  Wait on an event. This is obviously only called from the IEC tasks to wait for the next cycle.

3.2 Add Communication Channels

3.2.1 Serial

The serial block driver is the default one, which we support with our compact runtime. So you might already use this as a communication method.

3.2.2 UDP

Additional Components:

- CmpBlkDrvUdp

Dependencies:

- SysSocket

The SysSocket interface is based on the BSD socket interface and awaits a similar behaviour. The API is very big. That's why it isn't described here in detail. Instead here are the most essential functions, which are used from our Communication Interface (CmpBlkDrvUdp):

- SysSockSelect
- SysSockGetOption
- SysSockCreate
- SysSockClose
- SysSockSetOption
- SysSockBind
- SysSockGetHostName
- SysSockGetHostByName
- SysSockNtohl
- SysSockHtonl
- SysSockNtohs
- SysSockHtons
- SysSockSendTo
- SysSockRecvFrom
- SysSockIoctl
- SysSockInetNtoa
- SysSockInetAddr
- SysSockGetOSHandle

3.2.3 CAN

Additional Components:

- CmpBlkDrvCanClient
- CmpBlkDrvCanServer

Dependencies:

- CmpCAACanL2
- CmpCAAMemBlockMan
- CmpCAASdoClient

- CmpCAASdoServer
- CmpCAATick
- CmpCAATickUtil

Required:

- CAN-Mini-Driver

For a basic CAN-Mini-Driver, you just need to implement the following functionality:

- Initialization:
Call CL2_CmdRegister() with a list of all CAN Layer 2 Interface functions.

- CMD_Init():
Setup your chip in this function.

- CMD_Send():
Send Data to the bus. The control about the buffer is given back to CL2 with CL2_MsgSendAckn() within your send interrupt.

- CMD_Receive():
Read a message from the bus and write it into the passed message handle. The regular use of this function will be asynchronous, when you call it by your own within the **receive interrupt**:
```
CL2_MsgAlloc()
CMD_Receive()
CL2_MsgPutRQueue()
```

3.2.4 USB

Additional Components:

- CmpBlkDrvUsb

Required:

- USB-Mini-Driver

The USB-Mini-Driver needs to provide the following API:

- OpenDevice()
Search for a specific device, with VendorID and DeviceID and open it.

- Read()
Read in Bulk mode from an opened USB device.

- Write()
Write in Bulk mode to an opened USB device.

- CloseDevice()
Close the connection to an opened USB device.

3.2.5 SHM

Additional Components:

- CmpBlkDrvShm

Dependencies:

- SysEvent
- SysSemProcess
- SysShm

This block driver can be used, if you have two different processes running on the same processor, which should communicate with each other. Note, that you need some support for global, named Events, Semaphores and Shared Memory segments. Those mechanisms might be already provided from your operating system. If not, you have provide it by your own.

For example:

If an Event, called "myevent" is created two times from two different processes, the first one will create the event in the context of the operating system and the second one will just retrieve the already created event.

3.3 Add other Features

3.3.1 Debugging Support

Additional Components:

- CmpAppBP
- CmpAppForce

Note, that the component *CmpAppBP* relies on a correct breakpoint handling in the component SysCpuHandling. Maybe you didn't implement this in the first shot.

The component *CmpAppForce* adds support to force monitored variables to a specific value. It needs no special support from the system layer.

3.3.2 Full Trace Support

Additional Components:

- CmpTraceMgr

Dependencies:

- SysTimeRtc

Currently CODESYS supports two different kinds of traces. The default one creates a child application and reads the trace values with the usual monitoring service. For this trace, you need no special support in the runtime. But this one is also limited in it's functionality. For examples: You are not able to trace device parameters, you can't configure a trace from an IEC application and you can't store/restore it on the PLC.

The full trace is enabled with the following setting in the device description:
```
<ts:TargetSettings>
    <ts:section name="trace">
        <ts:setting name="tracemanager" type="boolean" access="visible">
            <ts:value>1</ts:value>
        </ts:setting>
```
Then, you will need the additional component *CmpTraceMgr*, which is dependent on the system component *SysTimeRtc*. At least, you need to implement the following function:

- SysTimeRtcGet()
  Return a localized UNIX timestamp.

Change History

Version	Description	Editor	Date
0.1	Issued based on version 3.4.1.0	IH	16.06.2010
0.2	Extended all sections and integrated comments from Max	IH	25.08.2010
1.0	Release after formal review and rework	MN	09.09.2010
1.1	Reference to new CODESYSControlV3_Manual corrected	AH	15.10.2010
1.2	Chap. 3.2.1 added (CDS-8637)	IH	21.10.2010
2.0	Release after formal review	MN	22.10.2010
2.1	CDS-29303; Spelling corrections	MN	17.09.2012
3.0	Release	MN	03.12.2012
3.1	Chap 2.3.5: Recommended buffer sizes corrected	MM	13.12.2012
4.0	Release after formal review	MaH	13.12.2012