

# **CODESYS Control V3 MicroRTS Programmer's Guide**

**Document Version 2.0**

**CONTENT**

<b>1</b>	<b>INTRODUCTION</b>	<b>5</b>
1.1	About This Manual	5
1.2	Related Documents	5
<b>2</b>	<b>REFERENCE IMPLEMENTATION QUICKSTART GUIDE</b>	<b>6</b>
2.1	What is the CODESYS V3 MicroRTS Reference Implementation?	6
2.2	Supported Features	6
2.3	Hardware and software requirements	6
2.4	Installation	7
2.4.1	Prerequisites	7
2.4.2	Installation Process	7
2.5	Evaluating Cortex-M3 MicroRTS	7
2.5.1	Overview	7
2.5.2	Building the Firmware	8
2.5.3	Connecting to the Target	8
2.5.4	Downloading the Firmware	9
2.5.5	Installing Device Description Files	10
2.5.6	Configuring the CODESYS V3 Gateway	10
2.5.7	Creating a Test Project in CODESYS IDE	10
2.5.7.1	Creating the simplest basic project	10
2.5.7.2	Accessing Hardware in CODESYS Application	13
2.5.8	LPC1768-Stick LED Indication	16
<b>3</b>	<b>CORTEX-M3 MICRORTS STARTER PACKAGE CONTENT</b>	<b>17</b>
3.1	Overview	17
3.2	Supported Targets	17
3.3	Toolchain	17
3.3.1	Build Tools for C Development	17
3.3.2	Drivers for Downloading and Debugging	18
3.4	Source Files	18
3.4.1	Overview	18
3.4.2	Platform Independent Source Files	18
3.4.3	Platform Specific Source Files	23
3.4.3.1	Overview	23
3.4.3.2	NXP-LPC1768 Folder Content	24
3.4.3.3	TI-LM3S9B96_uRTS Folder Content	26
3.5	Build Utilities	28
<b>4</b>	<b>ARCHITECTURE</b>	<b>29</b>

<b>4.1</b>	<b>Overview</b>	<b>29</b>
<b>4.2</b>	<b>MicroRTS Features</b>	<b>29</b>
<b>4.3</b>	<b>Differences Between the MicroRTS and Other Runtime System Profiles</b>	<b>30</b>
<b>4.4</b>	<b>Components Management</b>	<b>30</b>
4.4.1	Component Defined	30
4.4.2	Component Source Code Organization	31
4.4.2.1	Component Source Code	31
4.4.2.2	Root Module Requirements	33
4.4.2.3	Secondary Module Requirements	34
4.4.2.4	Subordinate Modules Requirements	34
4.4.3	Simplified Component Manager	35
4.4.3.1	Overview	35
4.4.3.2	Components List	36
4.4.3.3	Excluding Components Functionality	40
<b>4.5</b>	<b>Runtime Operation</b>	<b>41</b>
4.5.1	Startup Sequence	41
4.5.2	Operating Mode	42
<b>5</b>	<b>IMPLEMENTING THE MICRORTS</b>	<b>43</b>
<b>5.1</b>	<b>Overview</b>	<b>43</b>
<b>5.2</b>	<b>Creating Device Description File</b>	<b>43</b>
5.2.1	Overview	43
5.2.2	Specifying Device Identification and Device Information	44
5.2.3	Specifying Runtime Features	45
5.2.4	Configuring Codegenerator	46
5.2.5	Creating Memory Layout	47
5.2.5.1	Overview	47
5.2.5.2	Defining Code Area(s)	51
5.2.5.3	Defining Data Area(s)	52
5.2.6	Setting-up Tasks	53
<b>5.3</b>	<b>Organizing the MicroRTS Source Tree</b>	<b>54</b>
5.3.1	Source Tree Layout	54
5.3.2	Mandatory Configuration Macros for the MicroRTS Profile	56
<b>5.4</b>	<b>Defining Components</b>	<b>57</b>
5.4.1	Components Source Code Modifications	57
5.4.2	Adding Components to the MicroRTS Build Configuration	60
<b>5.5</b>	<b>Adapting Core System Components</b>	<b>60</b>
5.5.1	Overview	60
5.5.2	Specifying Target Identification	61
5.5.3	Implementing CPU-specific Functions	62
5.5.4	Implementing Memory Management	63

5.5.4.1	Overview	63
5.5.4.2	Defining the Runtime Stack	63
5.5.4.3	Defining the Runtime Free Storage (Heap)	63
5.5.4.4	Defining IEC Application Areas	65
5.5.4.5	Fixed-size Memory Blocks Allocation	67
5.5.5	Implementing System Ticks	68
5.5.6	Implementing Access to a Flash Memory	68
5.5.7	Implementing Exceptions Handling	70
<b>5.6</b>	<b>Adapting Communications</b>	<b>71</b>
<b>5.7</b>	<b>Configuring the Logger</b>	<b>71</b>
<b>5.8</b>	<b>Implementing a Debugging Console</b>	<b>71</b>

## 1 **Introduction**

### 1.1 **About This Manual**

The CODESYS V3 MicroRTS Programmer's Guide describes the basic architecture of a special profile of CODESYS V3 Runtime System (MicroRTS) intended for deploying on programmable logic controllers (PLC) and embedded devices based on CPUs and microcontrollers with a low memory budget. This document also contains brief guidelines to adapt MicroRTS to specific target devices.

### 1.2 **Related Documents**

It is recommended to refer to the following documents, which cover various aspects of CODESYS V3 OEM development and adaptation:

CODESYS Control V3 Manual – contains the detailed information about CODESYS Control V3 runtime system (RTS).

CODESYS Control V3 Migration and Adaptation – describes the basic approach to port CODESYS Control RTS to a bare hardware without an operating system, as well as to a platform with an unsupported operating system.

Tutorial: Creating own Runtime System Components and I/O Drivers – describes the process of creating custom runtime components and I/O drivers.

Generating libraries in CODESYS V3 – contains the detailed information on creating external and internal libraries.

## 2 Reference Implementation Quickstart Guide

### 2.1 What is the CODESYS V3 MicroRTS Reference Implementation?

The CODESYS V3 MicroRTS reference implementation, further referred to as the Cortex-M3 MicroRTS reference implementation or Cortex-M3 MicroRTS, is an adaptation of CODESYS V3 Runtime System for the NXP LPC1768 Cortex-M3 based microcontroller. Cortex-M3 MicroRTS supports all the features currently available in the MicroRTS profile of CODESYS V3 Runtime System.

The HITEX LPC1768-Stick evaluation board is used as a target device for demonstrating features implemented in Cortex-M3 MicroRTS.

The Cortex-M3 MicroRTS Starter Package is delivered with the source code and tools required for building, downloading and debugging the Cortex-M3 MicroRTS target binary and can be used by an OEM software developer as a learning tool for evaluating MicroRTS and/or as a starting point for porting MicroRTS to any other platform for which it is necessary to achieve the smallest possible flash memory and RAM footprint.

### 2.2 Supported Features

Cortex-M3 MicroRTS supports the following set of features:

- Downloading an IEC 61131-3 user application created with CODESYS V3 IDE to flash memory of the target device.
- In-flash execution of the user application.
- Monitoring of the user application variables.
- Writing and forcing values of the user application variables.
- Remote control of the application execution including Start/Stop and Single Cycle.
- Exceptions handling in an IEC code.
- Logging up to 5 log messages.

The current version of Cortex-M3 MicroRTS communicates with CODESYS V3 IDE via a virtual serial port that is automatically added to the Windows device tree when the LPC1768-Stick is connected to an USB port of the PC running CODESYS V3 IDE.

Cortex-M3 MicroRTS consumes 15.7 kB of RAM and 87.5 kB of flash memory.

The CODESYS application memory settings specified in the device description file are as follows:

- Code size (in flash memory): up to 64 kB (can be extended to 256 kB and more since the LPC1768 chip contains 512 kB of flash memory).
- Data size: up to 24 kB.

### 2.3 Hardware and software requirements

The PC should have the following characteristics:

- Running Windows XP/Vista/7 32/64-bit
- CODESYS V3.5 IDE
- 1 GB of free Hard Drive space
- Free USB Port

The HITEX LPC1768-Stick should be used as a target device.

## 2.4 Installation

### 2.4.1 Prerequisites

Before installing the Cortex-M3 MicroRTS Starter Package, please install the following software:

1. CODESYS IDE V3.5.1 of greater.
2. CodeSourcery Lite for Cortex-M3. This toolchain can be downloaded at: <https://sourcery.mentor.com/sgpp/lite/arm/portal/release1802> (IA32 Windows Installer). When the installation is finished, please create the TOOLCHAIN\_PATH environment variable that refers to the CodeSourcery toolchain installation path (by default c:\Program Files (x86)\CodeSourcery\Sourcery G++ Lite).
3. Before installation begins, please make sure that the HITEK LPC1768-Stick is **not connected** to an USB port of the PC.

### 2.4.2 Installation Process

1. Run *Setup\_CODESYSuRTSStarterPackageV3.5SP1.exe* (the name may vary depending on the starter package version). Click **Next** in the **Welcome** installation wizard window. The **License Agreement** wizard window will be displayed on the screen.
2. Read the terms of the license agreement and click **Yes**, if you accept them. The **Choose Destination Location** wizard window will be shown on the screen.
3. If you wish to change the default installation folder, click **Browse** in **Destination Folder**, select desired folder for installing the starter package and click **Next**.
4. In the **Select Program Folder** wizard window, enter the name of the program group which will be created during installation, then click **Next**. The **Setup Status** wizard window will be displayed on the screen indicating status of the installation process.
5. When the **FTDI Bus Driver Updater** dialog box appears on the screen, click **Next**, and then proceed confirming successful stages of FTDI drivers installation.
6. Click **Finish** in the **InstallShield Wizard Complete** window.

## 2.5 Evaluating Cortex-M3 MicroRTS

### 2.5.1 Overview

The following basic operations can be performed while evaluating Cortex-M3 MicroRTS:

1. Building the firmware.
2. Connecting to the target device and downloading the firmware.
3. Configuring the CODESYS V3 Gateway.
4. Installing the LPC1768-Stick device description file to the CODESYS device repository.
5. Creating a simple project in CODESYS.
6. Downloading the compiled CODESYS project to the target.
7. Monitoring, writing and forcing variables in the downloaded application.
8. Forcing exceptions to occur in the application.
9. Fetching the target device log entries to the CODESYS IDE device log page.

## 2.5.2 Building the Firmware

The Cortex-M3 MicroRTS Starter Package is supplied with the full sources and tools that are necessary to build the target firmware. The Cortex-M3 MicroRTS Starter Package installation program creates a set of shortcuts in the **CODESYS V3 uRTS StarterPackage-Tools-Cortex-M3-NXP-LPC1768** program group allowing to build, download and debug the firmware in the target device.

To build the firmware, just click the **Build Firmware** shortcut in the **CODESYS V3 uRTS StarterPackage-Tools-Cortex-M3-NXP-LPC1768** program group. The Windows console window will be displayed on the screen indicating the build process as shown in Figure 1.

If the build process succeeds, the message "Build OK" is displayed at the bottom line of the console window.

## 2.5.3 Connecting to the Target

The firmware can be downloaded to the LPC1768-Stick target device by use of the [Open On-Chip Debugger](#) (OpenOCD) and the [GNU Project Debugger](#) (GDB). In order to support this process, the [FTDI](#) device driver for supported Cortex-M3 chips is installed during the installation of the Cortex-M3 MicroRTS Starter Package. Before trying to download the firmware, the user has to make sure that the FTDI device drivers were successfully installed and the Virtual COM Port (VCP) device instance was successfully added to the Windows device tree.

1. Connect the LPC1768-Stick to a free USB port of the PC.
2. Select **Start – Run** and type

```
mmc devmgmt.msc
```

The **Device Manager** window will be displayed on the screen.

3. Open **Ports (COM & LPT)** list element in the device tree and make sure that the **LPC1768-Stick COM Port (COMxx)** device is available, where xx is a port number that should be used while configuring the CODESYS V3 Gateway.
4. Close the Device Manager window.

The target and the PC are now ready for downloading the firmware to the target.

```

C:\Windows\system32\cmd.exe
Embedded/CmpRouterEmbeddedAddrSrc.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
lib/CmpCommunicationLib.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
verEmbedded/CmpChannelServerEmbedded.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
bedded.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
pEmbeddedSrv.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
ndling.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
unCrc16.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
unCrc32.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
bedded/CmpChannelMgrEmbedded.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
d/CmpScheduleEmbedded.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
oMgrEmbedded.c
../../../../Components/CmpIoMgrEmbedded/CmpIoMgrEmbedded.c:30:10: note: #pragma message: RTS_
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
ver/CmpNameServiceServer.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
bedded.c
arm-none-eabi-gcc -mthumb -mcpu=cortex-m3 -O0 -ffunction-sections -fdata-sections -MD -c -I. -Isrc//
gEmbeddedSrv.c
cp "c:/program files (x86)/codesourcery/sourcery g++ lite/bin/./lib/gcc/arm-none-eabi/4.5.2/thumb2/
arm-none-eabi-ld -I codesys.ld --entry ResetISR -lgcc -Map=codesys.map --gc-sections -o gcc/CoDeSysS
gcc/CmpMonitor.o gcc/SysExcept.o gcc/SysExceptCortexM3.o gcc/SysInternalLibDefault.o gcc/SysCon.o g
mpChecksumCrc16.o gcc/CmpChecksumCrc32.o gcc/CmpChannelMgrEmbedded.o gcc/CmpScheduleEmbedded.o gcc/C
Build OK
Drücken Sie eine beliebige Taste . . .

```

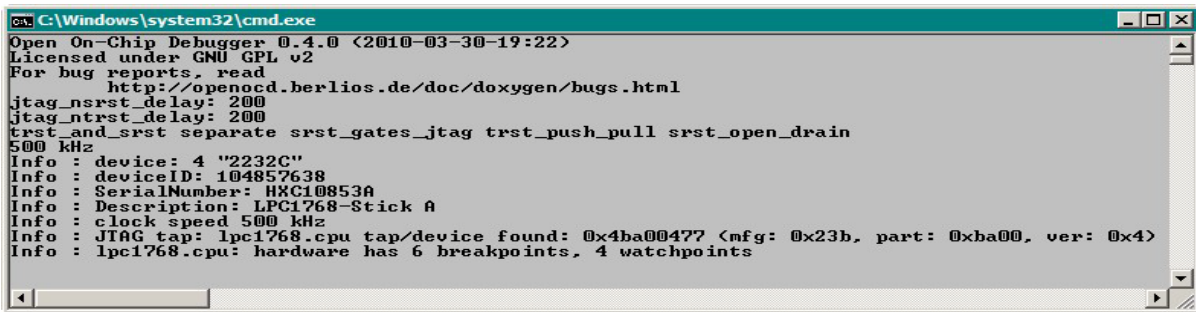
Figure 1. Console window during the firmware build process



## 2.5.4 Downloading the Firmware

If the LPC1768-Stick target device contains the Cortex-M3 MicroRTS firmware, when the target is connected to an USB port of the PC, the second LED of red color starts blinking with frequency of either 1 or 4 Hz. The sequence below only applies, if the target is empty – the first LED (closest to the target USB connector) of green color is lit constantly, and the rest two LEDs are off.

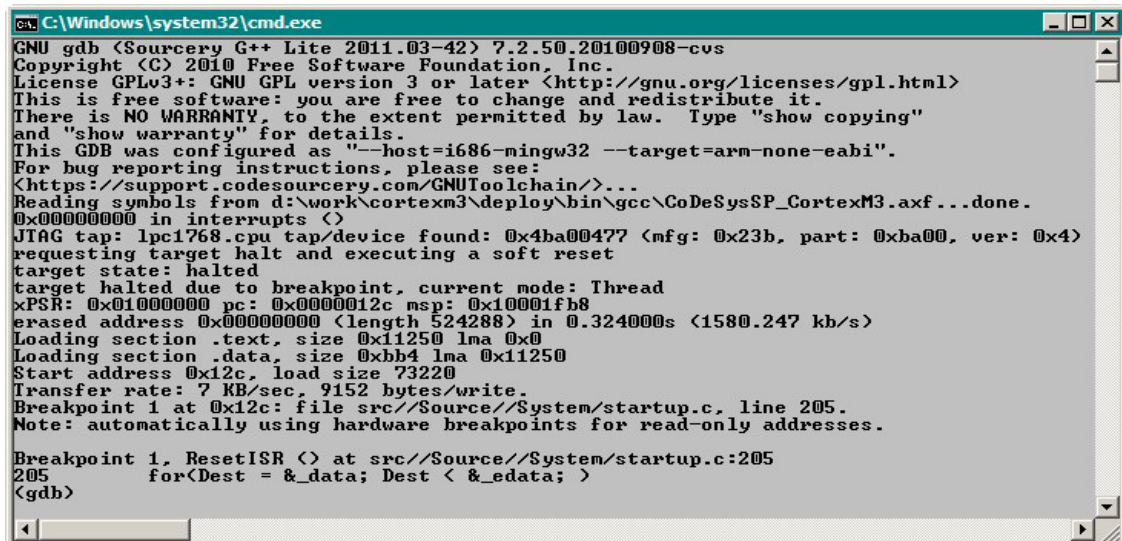
1. Click on **Start – All Programs** and select **CODESYS V3 uRTS StarterPackage–Tools–Cortex-M3–NXP-LPC1768 – Run OCD Server**. The console window shown in Figure 2 will be displayed on the screen.



```
C:\Windows\system32\cmd.exe
Open On-Chip Debugger 0.4.0 (2010-03-30-19:22)
Licensed under GNU GPL v2
For bug reports, read
  http://openocd.berlios.de/doc/doxygen/bugs.html
jtag_nsrst_delay: 200
jtag_trst_delay: 200
srst_and_srst separate srst_gates_jtag trst_push_pull srst_open_drain
500 kHz
Info : device: 4 "2232C"
Info : deviceID: 104857638
Info : SerialNumber: HXC10853A
Info : Description: LPC1768-Stick A
Info : clock speed 500 kHz
Info : JTAG tap: lpc1768.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
Info : lpc1768.cpu: hardware has 6 breakpoints, 4 watchpoints
```

Figure 2. OpenOCD console window

2. Click on **Start – All Programs** and select **CODESYS V3 uRTS StarterPackage–Tools–Cortex-M3–NXP-LPC1768 – Firmware Download**. The console window shown in Figure 3 will be displayed on the screen indicating the process of flashing the target.



```
C:\Windows\system32\cmd.exe
GNU gdb (Sourcery G++ Lite 2011.03-42) 7.2.50.20100908-cvs
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later (http://gnu.org/licenses/gpl.html)
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "--host=i686-mingw32 --target=arm-none-eabi".
For bug reporting instructions, please see:
<https://support.codesourcery.com/GNUToolchain/>...
Reading symbols from d:\work\cortexm3\deploy\bin\gcc\CoDeSysSP_CortexM3.axf...done.
0x00000000 in interrupts ()
JTAG tap: lpc1768.cpu tap/device found: 0x4ba00477 (mfg: 0x23b, part: 0xba00, ver: 0x4)
requesting target halt and executing a soft reset
target state: halted
target halted due to breakpoint, current mode: Thread
xPSR: 0x01000000 pc: 0x0000012c msp: 0x10001fb8
erased address 0x00000000 (length 524288) in 0.324000s (1580.247 kb/s)
Loading section .text, size 0x11250 lma 0x0
Loading section .data, size 0xbb4 lma 0x11250
Start address 0x12c, load size 73220
Transfer rate: 7 KB/sec, 9152 bytes/write.
Breakpoint 1 at 0x12c: file src//Source//System/startup.c, line 205.
Note: automatically using hardware breakpoints for read-only addresses.

Breakpoint 1, ResetISR () at src//Source//System/startup.c:205
205     for(Dest = &_data; Dest < &_edata; )
(gdb)
```

Figure 3. Flashing the target with gdb

3. Close the console window shown in Figure 3 and then repeat step 2 to perform flash programming again.
4. Close the console window and then close the OpenOCD console window.
5. Disconnect the target from an USB port of the PC.
6. Connect the target back to the same USB port of the PC. Make sure that the second LED of red color is blinking cyclically with frequency of 1 Hz. If it is not, repeat Steps 1–5 until the second LPC1768-Stick LED starts blinking.

**Note:** Typically, the sequence described in Steps 1–5 has to be repeated at least two times to re-flash the target because the LPC1768 chip has a bit different implementation of the chip reset function than expected by gdb.

At this point the target device is ready to communicate with CODESYS.

## 2.5.5 Installing Device Description Files

The Cortex-M3 MicroRTS Starter Package is supplied with two target description files for CODESYS V3:

*MicroRuntime\_LPC1768.devdesc.xml* – device description file containing the LPC1768 Cortex-M3 SoC device description, which is used for creating CODESYS V3 projects for the LPC1768-Stick target.

*MicroRuntime\_LM3S9B96.devdesc.xml* – device description file containing the TI-LM3S9B96 Cortex-M3 uRTS device description, which is used for creating CODESYS V3 projects for the Stellaris DK-LM3S9B96 Evaluation Board.

Both files are located in the <INSTALL\_FOLDER>\DeviceDescriptions folders, where <INSTALL\_FOLDER> is a root installation folder of the starter package.

These files are installed automatically while installing the Cortex-M3 MicroRTS Starter Package. To update or re-install the device description in the CODESYS device repository after changing them:

1. Start CODESYS V3 IDE.
2. Select **Tools – Install device** in the CODESYS V3 IDE main menu.
3. Navigate to the <INSTALL\_FOLDER>\DeviceDescriptions folder in the **Select Device Description** dialog box, then select either *MicroRuntime\_LPC1768.devdesc.xml* or *MicroRuntime\_LM3S9B96.devdesc.xml* and click **Open**. The device description will be installed to the CODESYS V3 IDE devices repository.

## 2.5.6 Configuring the CODESYS V3 Gateway

Typically, the Cortex-M3 MicroRTS Starter Package installation program configures the CODESYS V3 Gateway automatically. The steps listed below should be performed, if it is necessary to configure the gateway manually:

1. Run *Notepad* (or any other plain text editor) and open the *Gateway.cfg* configuration file located in *GatewayPLC* sub-folder of the CODESYS IDE installation folder (e.g. *C:\Program Files (x86)\3S CODESYS*).
2. Change/add the following entries in/to the [CmpRouter] section:  

```
0.NumSubNets=2  
0.SubNet.1.Interface=LPC1768VCP
```
3. Add the following entries to the [CmpBlkDrvCom] section:  

```
Com.0.Baudrate=115200  
Com.0.Port=<Virtual COM Port number, see section 2.5.3>  
Com.0.Name=LPC1768VCP
```
4. Save and close the *Gateway.cfg* configuration file.
5. Restart the CODESYS Gateway.

CODESYS V3 IDE is now ready to download user projects to the target.

## 2.5.7 Creating a Test Project in CODESYS IDE

### 2.5.7.1 Creating the simplest basic project

1. Start CODESYS V3 IDE clicking the **CODESYS V3.5** shortcut in the **3S CODESYS – CODESYS** program group and select **File – New Project** in the CODESYS main menu. The **New Project** dialog box will be displayed on the screen as shown in Figure 4.

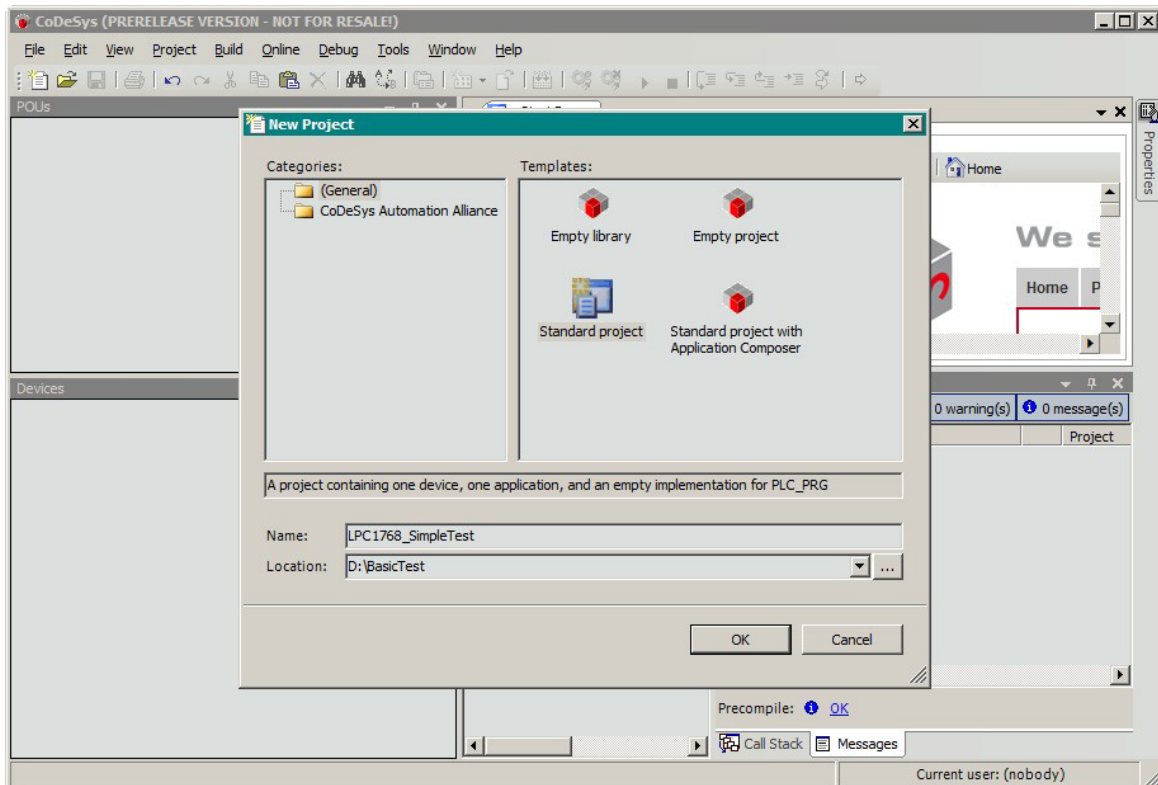


Figure 4. Creating a new project

2. Select *Standard project* in the **Templates** pane, choose the project location in the **Location** field and type in the name of the project to be created and click **OK** to close the dialog box. The **Standard Project** dialog box will be displayed on the screen as shown in Figure 5 prompting you to select an implementation language and a target for the project.
3. Select *LPC1768 Cortex-M3 SoC (3S-Smart Software Solutions GmbH)* as shown in Figure 5 and click **OK** to close the dialog box. The new project will be created and displayed in the CODESYS main window as shown in Figure 6.

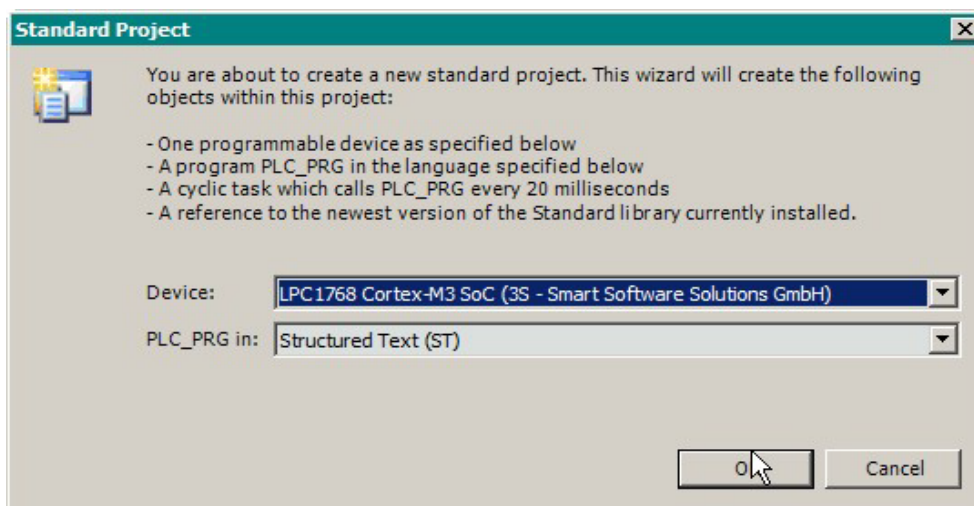


Figure 5. Selecting the target

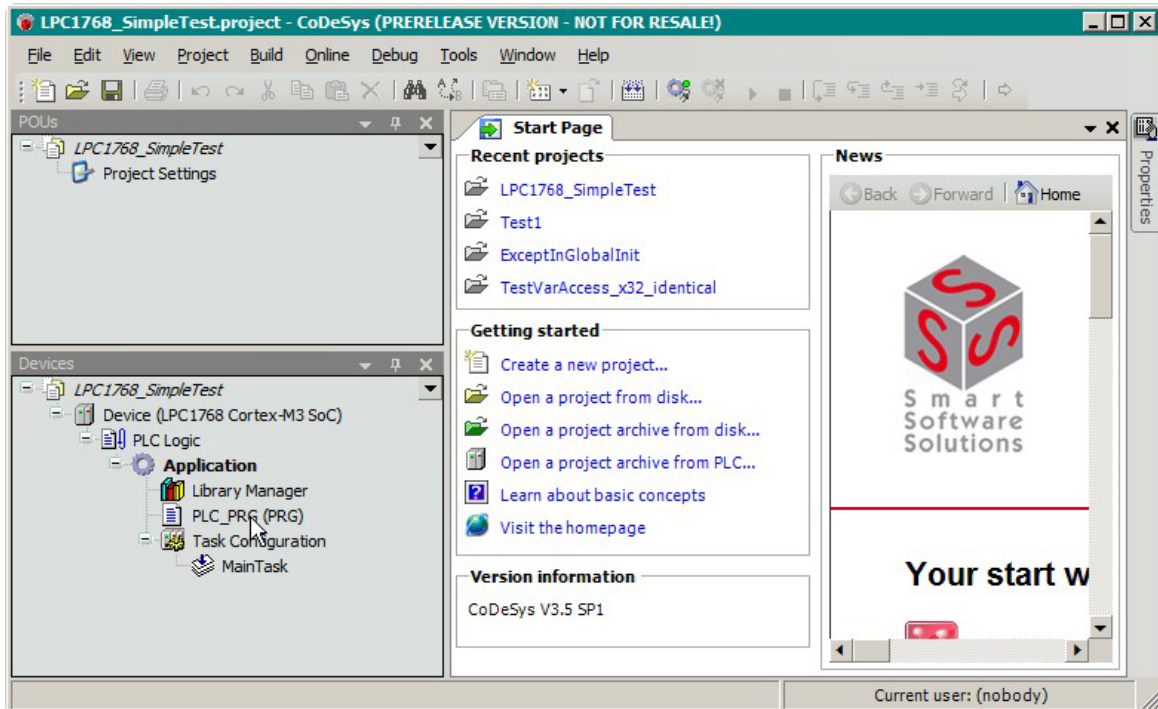


Figure 6. New project in the CODESYS V3 main window

4. Double-click the *PLC\_PRG (PRG)* item in the **Devices** tree and declare the *dwCounter* variable of type *DWORD* in the variables declaration pane of the Structured Text editor:

```
VAR
  dwCounter : DWORD := 0;
END_VAR
```

5. In the POU code pane, type in the expression that increments *dwCounter* at each cycle of the *PLC\_PRG* execution as shown in Figure 7.

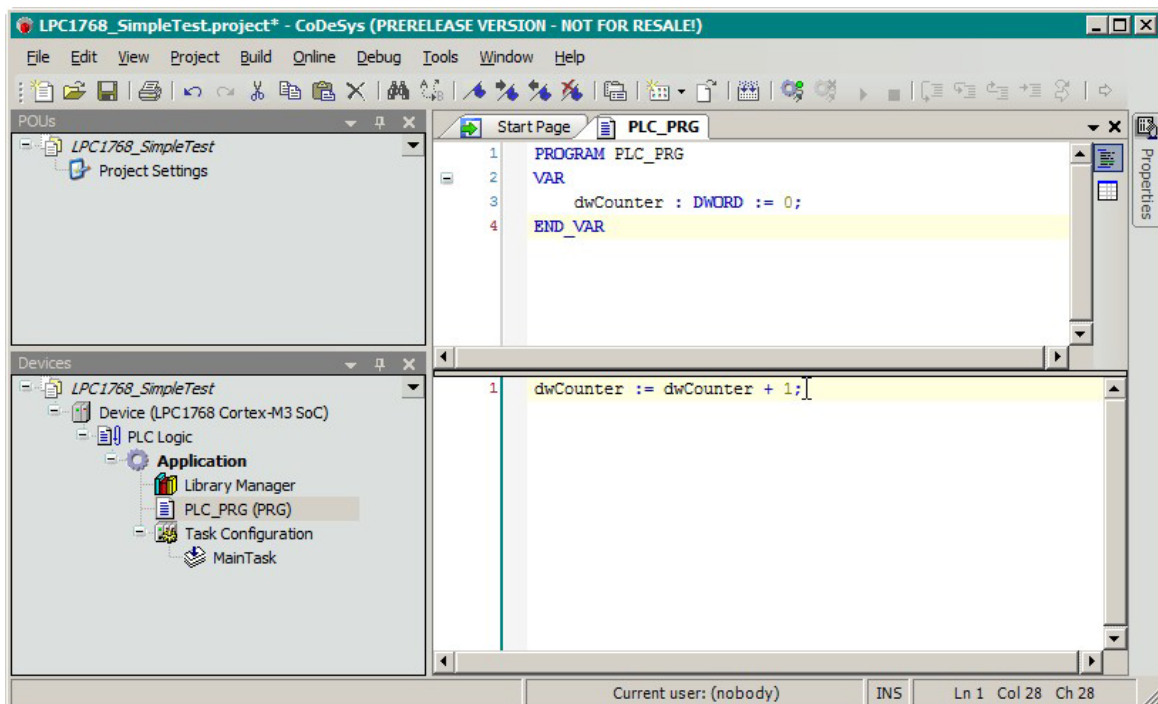


Figure 7. Simple test program

- Double-click on *Device (LPC1768 Cortex-M3 SoC)* in the **Devices** tree, select *Gateway-1* in the **Communication Settings** pane and click on **Scan network**. The *uRTS for LPC1768 Cortex-M3 SoC* node will be displayed in the network tree as shown in Figure 8.

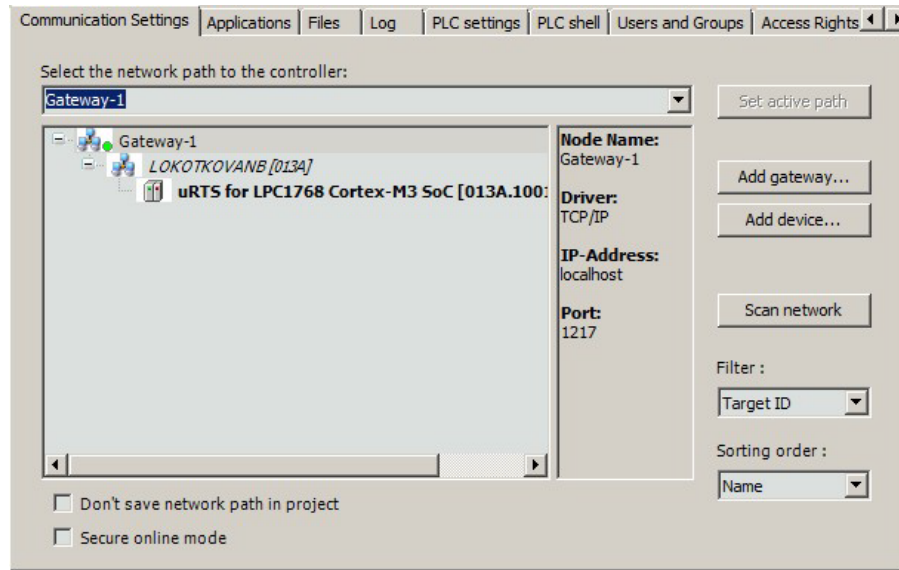


Figure 8. LPC1768-Stick target node in the network tree

- Double-click on *PLC\_PRG (PRG)* in the **Devices** tree, select **Online – Login** in the CODESYS main menu then click **Yes** in the **CODESYS** dialog box which appears on the screen. The project will be compiled and downloaded to the target. The second LED of red color will start blinking with frequency of 4 Hz indicating that the user application exists in the target.
- Select **Debug – Start** (or press F5) in the CODESYS main menu to run the project in the target. The CODESYS main window will look as shown in Figure 9 indicating the project execution and monitoring changes of the *dwCounter* variable.

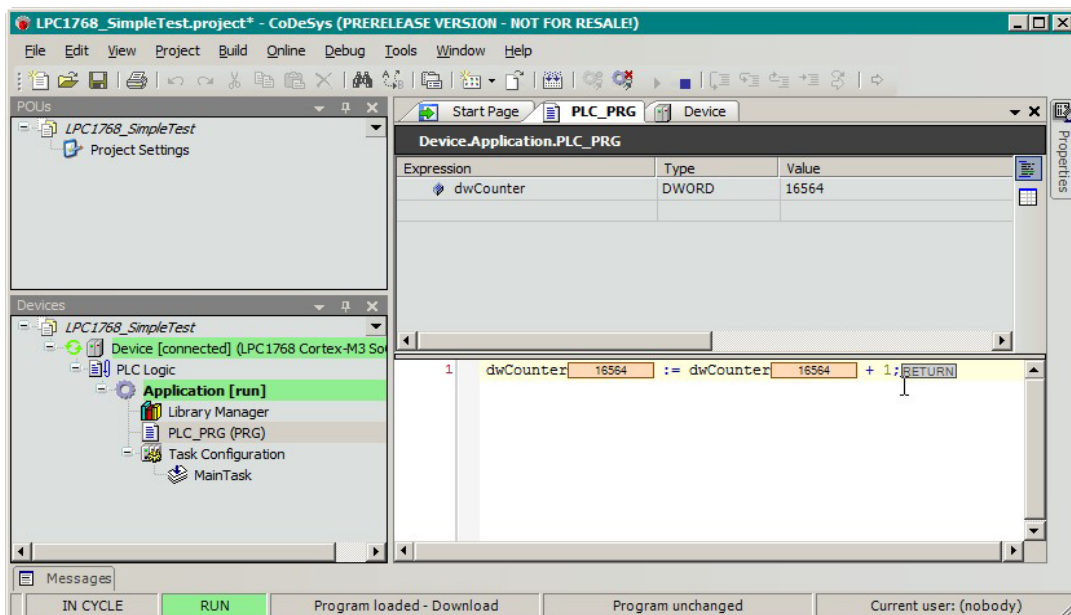


Figure 9. Monitoring the *dwCounter* variable and project execution

- Select **Online – Logout** to disconnect from the target.

### 2.5.7.2 Accessing Hardware in CODESYS Application

The LPC1768-Stick target has an additional LED connected to a GPIO line as it can be accessed directly in the user's application written in CODESYS. The instructions below explain how to control that LED:

1. Start CODESYS and open the project created in section 2.5.7.1.
2. Right-click on the *Application* node in the *Devices* tree and select **Add Object – Global Variable List** in the context menu. The **Add Global Variables List** dialog box will be displayed on the screen.
3. Enter the name *GPIOConsts* in the *Name* field for the global variables list to be created and then click **Open**. The new global variables list will be displayed in the CODESYS main window as shown in Figure 10.

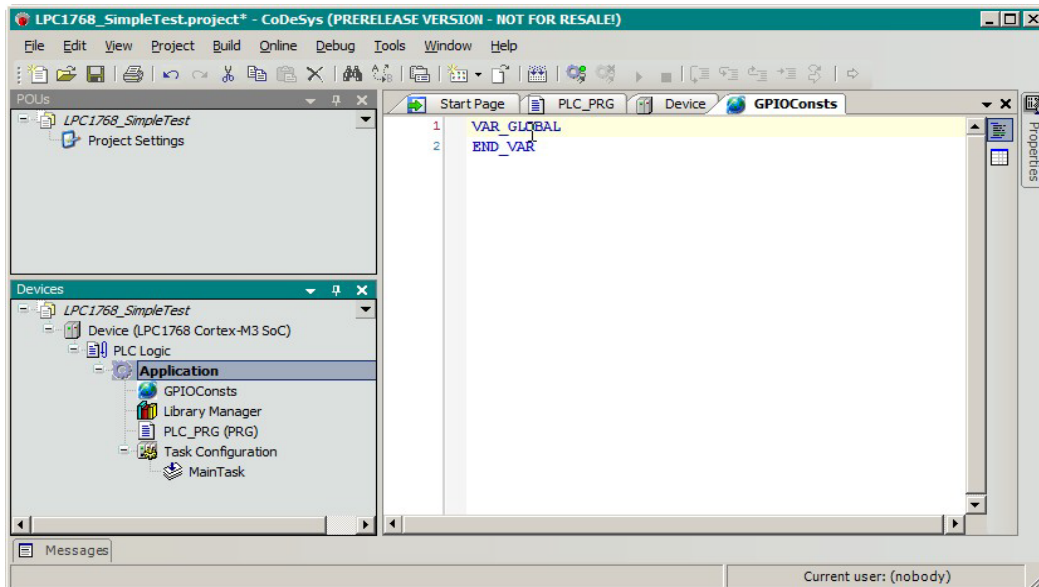


Figure 10. Monitoring variables in the demo project

4. Declare the following constants which will be used for accessing the GPIO line connected to the third LED:

```
VAR_GLOBAL
  GPIO_DIR_BASEADDR : __XWORD := 16#2009C000;
  GPIO_SET_BASEADDR : __XWORD := GPIO_DIR_BASEADDR + 16#18;
  GPIO_CLR_BASEADDR : __XWORD := GPIO_DIR_BASEADDR + 16#1C;
  GPIO_GET_BASEADDR : __XWORD := GPIO_DIR_BASEADDR + 16#14;
END_VAR
```

5. Right-click on the *Application* node in the **Devices** tree and select **Add Object – POU** in the context menu. The **Add POU** dialog box will be displayed on the screen.
6. Enter the name *GPIO\_Get* in the **Name** field, select **Function** in the **Type** group, specify *BOOL* as a return type for the function in the **Return type** field, and select *Structured Text (ST)* in the **Implementation Language** combo-box as shown in Figure 11 and then click **Open**.

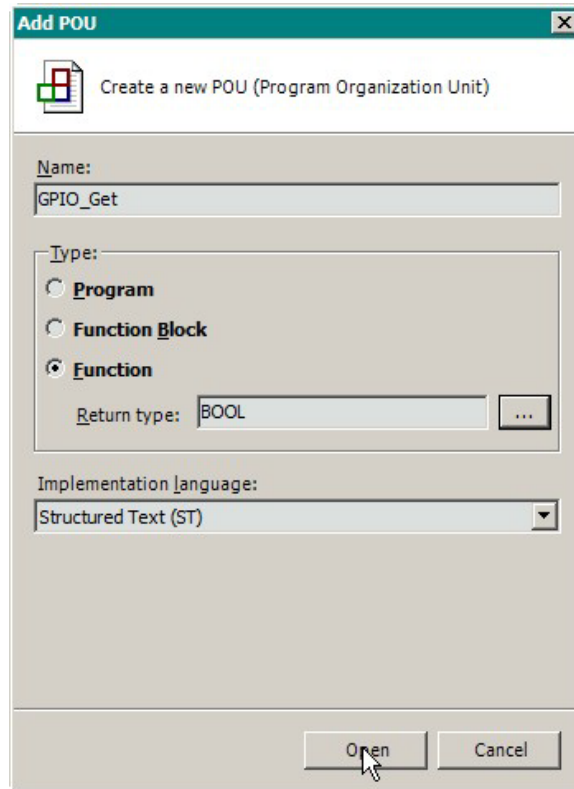


Figure 11. Creating the *GPIO\_Get* function

7. Create another function named *GPIO\_Set* with the same return type.
8. Double-click the *GPIO\_Get* element in the **Devices** tree to open the corresponding function in the Structured Text (ST) editor. In the declaration area, declare the following formal input parameters and local variable for the function:

```
VAR_INPUT
  Port : DWORD;
  Pin : DWORD;
END_VAR
```

```
VAR
  pAddr : POINTER TO DWORD;
END_VAR
```

9. In the ST editor implementation area, enter the following source code for the function:

```
pAddr := GPIO_GET_BASEADDR + Port * 8;
GPIO_Get := (SHR(pAddr^, Pin) AND 1) <> 0;
```

10. Double-click the *GPIO\_Set* element in the **Devices** tree to open the corresponding function in the ST editor. In the declaration area, declare the following formal input parameters and local variable for the function:

```
VAR_INPUT
  Port : DWORD;
  Pin : DWORD;
  State : BOOL;
END_VAR
```

```
VAR
  PortBase : __XWORD;
  pAddr : POINTER TO DWORD;
END_VAR
```

11. In the ST editor implementation area, enter the following source code for the function:

```
IF State THEN
  PortBase := GPIO_SET_BASEADDR;
```

```
ELSE
  PortBase := GPIO_CLR_BASEADDR;
END_IF

pAddr := PortBase + Port * 8;
pAddr^ := SHL(1, Pin);

GPIO_Set := State;
```

12. Double-click the *Library Manager* element in the **Devices** tree, click on **Add Library...**, and in the **Libraries** list of the **Add Library** dialog box displayed on the screen select the *Util* library, then click **OK**. The *Util* library will be added to the list of libraries available in the project.
13. Double-click the *PLC\_PRG* element in the **Devices** tree to open the corresponding IEC program in the ST editor. Add the following three variables in the declaration area:

```
// LED command
LedState : BOOL;

// half a period of LED blinking for LedBlinker POU instance
HalfPeriod : TIME := T#250MS;

// BLINK FB instance
LedBlinker : BLINK;
```

14. In the ST editor implementation area, enter the following source code:

```
dwCounter := dwCounter + 1;

LedBlinker( ENABLE := TRUE,
            TIMELOW := HalfPeriod,
            TIMEHIGH := HalfPeriod,
            OUT => LedState);

GPIO_Set(0, 25, LedState );
```

15. Rebuild, download and start the project. The third LED on the target will start blinking with frequency of 2 Hz.

## 2.5.8 LPC1768-Stick LED Indication

Three LEDs are used as follows:

- Green (1<sup>st</sup>, closest to the USB connector): lit continuously after connecting to an USB of the PC if the FTDI device drivers were successfully installed and corresponding devices were added to the Windows device tree on the PC.
- Red (2<sup>nd</sup>):
  - OFF – no firmware exists on the target;
  - blinks with frequency of 1 Hz – firmware exists on the target, no user application;
  - blinks with frequency of 4 Hz – firmware and user application exist on the target.
- Green (3<sup>rd</sup>) – can be controlled by the user application. See section 2.5.7.2 for details.



### 3 Cortex-M3 MicroRTS Starter Package Content

#### 3.1 Overview

The Cortex-M3 MicroRTS Starter Package is supplied with the following items included to the delivery set:

- HITEX LPC1768-Stick Hardware
- CD with the Cortex-M3 MicroRTS source code, auxiliary utilities and device description files for two supported target platforms.

This chapter describes the content of the installation program supplied on the Starter Package CD. All the files and folders are copied by the installation program to a destination folder that can be selected on a development PC when the installation starts. These files and folders are described below relatively to that destination folder.

The structure of the destination folder is as shown in the table below.

Folder/File	Description
CODESYSV3	Source code and auxiliary utilities
BuildUtils	M4 preprocessor and common M4 definition files for generating the RTS components interface and dependency header files.
Components	Platform-independent source files of the RTS.
Platforms	Platform-dependent source files for Cortex-M3
DeviceDescriptions	Device description files for the Cortex-M3 MicroRTS reference implementation.
Documentation	Documentation files mentioned in section 1.2.
Tools	Tools for building and debugging the RTS binaries.
FTDI	32- and 64-bit versions of <a href="#">FTDI</a> drivers.

#### 3.2 Supported Targets

The Cortex-M3 MicroRTS Starter Package is supplied with two target description files for CODESYS V3:

*MicroRuntime\_LPC1768.devdesc.xml* – device description file containing the LPC1768 Cortex-M3 SoC device description for creating CODESYS V3 projects for the LPC1768-Stick target.

*MicroRuntime\_LM3S9B96.devdesc.xml* – device description file containing the TI-LM3S9B96 Cortex-M3 uRTS device description for creating CODESYS V3 projects for the Stellaris DK-LM3S9B96 Evaluation Board.

Both files are located in the *DeviceDescriptions* folder and installed into the CODESYS V3 devices repository automatically while installing the Cortex-M3 MicroRTS Starter Package.

#### 3.3 Toolchain

##### 3.3.1 Build Tools for C Development

The [CodeSourcery Lite](#) toolchain is the set of C/C++ development tools needed for building binary executables for Cortex-M3-based CPUs and microcontrollers. The toolchain should be installed before installing the Cortex-M3 MicroRTS Starter Package. When the CodeSourcery installation is finished,

please create (or update) the `TOOLCHAIN_PATH` environment variable referring to the CodeSourcery installation folder (e.g. `c:\Program Files (x86)\CodeSourcery\Sourcery G++ Lite\`).

The toolchain GNU C compiler is invoked by the make utility supplied in the toolchain. The make utility itself is launched from the Windows command shell using *MakeAll.bat* files located in the target folders:

*CODESYSV3\Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\MakeAll.bat* – command file for building the runtime binary for the LPC1768-Stick target device. The corresponding make-file is located in the same folder.

*CODESYSV3\Platforms\Native\CortexM3\TI-LM3S9B96\_uRTS\Projects\CodeSourcery\MakeAll.bat* – command file for building the runtime binary for the Stellaris DK-LM3S9B96 Evaluation Board.

The additional tools and libraries which are not part of the CodeSourcery toolchain are located in the `CODESYSV3\Platforms\Native\CortexM3\Tools\` folder.

### 3.3.2 Drivers for Downloading and Debugging

The firmware can be downloaded to supported target devices and debugged using the [Open On-Chip Debugger](#) (OpenOCD) and [GDB](#). In order to support downloading and debugging the firmware, the [FTDI](#) device driver for supported Cortex-M3 chips is being installed during installation of the Cortex-M3 MicroRTS Starter Package. The driver installation binaries are located in the `Tools\FTDI` folder. If you wish to re-install it manually, please run `install.bat` in the `Tools\FTDI` folder.

## 3.4 Source Files

### 3.4.1 Overview

The Cortex-M3 MicroRTS Starter Package is delivered with the source code of the CODESYS V3 RTS components that can be included to the MicroRTS binary executable.

Each platform-independent component sub-folder (*Cmpxxxx*) contains a pair of files with the *Dep* name suffix (for example: *CmpAppEmbeddedDep.m4* and *CmpAppEmbeddedDep.h*). The `<ComponentName>Dep.m4` file of each `<ComponentName>` component contains the m4-preprocessor declarations used for generating the corresponding `<ComponentName>Dep.h` header file. This header file, called a dependency header, consists of the list of other components and their interface functions, which should be imported to the `<ComponentName>` component source files.

The *Components* folder contains pairs of files with the *Itf* name suffix (for example: *CmpAppItf.m4* and *CmpAppItf.h*). The `<ComponentName>Itf.m4` file of each `<ComponentName>` component contains the m4-preprocessor declarations used for generating the corresponding `<ComponentName>Itf.h` header file consisting of data types and functions declarations composing the `<ComponentName>` component interface.

### 3.4.2 Platform Independent Source Files

The platform-independent source files located in the `CODESYSV3\Components` folder are described in the table below.

Folder/File	Description
<code>_OptionalIncludes</code>	Common header files used across the RTS source code
<code>Profiles</code>	Header files specific for certain RTS profiles
<code>uRTS</code>	Header files specific for the MicroRTS profile
<code>CMMicro.h</code>	The root header file of the MicroRTS profile. This header file is automatically included to the components dependency header files.
<code>Aspects</code>	Aspect definitions and declarations specific for the MicroRTS profile.
<code>uCmpListAsp.h</code>	This header file is used in the MicroRTS Component Manager ( <code>CMMicro.c</code> ) for defining a list of RTS components and a table of function tables

Folder/File	Description
	<p>exported by the components (see uCmpListDecl.h). This file is included to CMMicro.c multiple times, each time with a different aspect enabled by the corresponding macro: INCLUDE_CMP_HOOKS_DECLARATIONS, INCLUDE_CMP_HOOKS_LIST, INCLUDE_CMP_EXTERNALS_LIST.</p>
uCmpListDecl.h	<p>This header file is used in the MicroRTS Component Manager (CMMicro.c) for declaring a list of RTS components and a table of function tables exported by the components.</p> <p>The components list header file contains the following definitions:</p> <pre>CDS3_DECLARE_COMPONENTS_LIST_BEGIN(uRtsCM3)   CDS3_ADD_COMPONENT_ENTRY(SysMem)   /* ... other system components */   CDS3_ADD_COMPONENT_ENTRY(SysInternalLib)   CDS3_ADD_COMPONENT_ENTRY(CmpHeapPool)   CDS3_ADD_COMPONENT_ENTRY(CmpMemPool)   /* ... other functional components */   CDS3_ADD_COMPONENT_ENTRY(CmpSrv) CDS3_DECLARE_COMPONENTS_LIST_END(uRtsCM3)</pre> <p>The name of the components list header file is supplied to the compiler used for building the MicroRTS binary executable with the RTS_CONFIG_FILE macro using the /D or -D option or in the sysdefines.h target configuration header file:</p> <pre>#define RTS_CONFIG_FILE rtsconfig/uRtsCortexM3.h</pre> <p>The aspects activation in uCmpListDecl.h is performed as follows:</p> <pre>/* The components list defined in RTS_CONFIG_FILE */ /* is processed here using different aspects.      */  /* Declare references to the components HookFunction */ /* routines and to the components export tables.     */ /* The INCLUDE_CMP_HOOKS_DECLARATIONS aspect        */ /* is active here (see uCmpListAsp.h).              */  #define INCLUDE_CMP_HOOKS_DECLARATIONS #include &lt;OptionalIncludes/Profiles/uRTS/Aspects/uCmpListAsp.h&gt; #include ANGLE_BRACKETS(RTS_CONFIG_FILE) #undef INCLUDE_CMP_HOOKS_DECLARATIONS  /* Define a table of components.                    */ /* The INCLUDE_CMP_HOOKS_LIST aspect is active here */ /* (see uCmpListAsp.h).                            */  #define INCLUDE_CMP_HOOKS_LIST #include &lt;OptionalIncludes/Profiles/uRTS/Aspects/uCmpListAsp.h&gt; #include ANGLE_BRACKETS(RTS_CONFIG_FILE) #undef INCLUDE_CMP_HOOKS_LIST  /* Define a table of components export tables.      */ /* The INCLUDE_CMP_EXTERNALS_LIST aspect is active here */ /* (see uCmpListAsp.h).                            */ #define INCLUDE_CMP_EXTERNALS_LIST #include &lt;OptionalIncludes/Profiles/uRTS/Aspects/uCmpListAsp.h&gt; #include ANGLE_BRACKETS(RTS_CONFIG_FILE) #undef INCLUDE_CMP_EXTERNALS_LIST</pre>
CmpAppEmbedded	<p>The IEC Application Manager (embedded version) component folder. The IEC Application Manager is responsible for controlling an IEC application downloaded to the target and for the application code and data handling. It also implements the application specific online services: downloading, debugging, start/stop, etc.</p>
CmpAppEmbedded.c	The IEC Application Manager root source file.
CmpAppEmbeddedSrv.c	The IEC Application Manager online services implementation file.
CmpAppEmbedded.h	The IEC Application Manager functions declarations shared between the

Folder/File	Description
	root source file and the online services implementation file.
CmpAppForce	The IEC Application Manager component folder that is responsible for forcing variables functionality.
CmpAppForce.c	The IEC Application Manager forcing interface implementation.
CmpBinTagUtil	The online services input/output streams reader and writer component folder.
CmpBinTagUtil.c	The root source file implementing the online services input/output streams reader and writer.
CmpBlkDrvCom	The serial port block driver folder.
CmpBlkDrvCom.c	The serial port block driver root source file.
CmpBlkDrvCom.h	The serial port block driver specific declarations.
CmpBlkDrvUdp	The UDP block driver folder.
CmpBlkDrvUdp.c	The UDP block driver root source file.
CmpBlkDrvUdp.h	The UDP block driver specific declarations.
CmpChannelMgrEmbedded	The Channels Manager (embedded version) folder. The Channels Manager is responsible for sending/receiving messages over selected channels between RTS and remote clients accessing the RTS online communications services.
CmpChannelMgrEmbedded.c	The Channels Manager root source file.
CmpChannelServerEmbedded	The Channels Server (embedded version) folder. The Channels Server is responsible for creating, maintaining and removing communication channels between RTS and remote clients which access the RTS online communication services.
CmpChannelServerEmbedded.c	The Channels Server root source file.
CmpChecksum	This folder contains implementation of CRC16 and CRC32 algorithms,
CmpChecksum.c	The root source file of the CRC implementation component.
CmpChecksumCrc16.c	The component's source file that implements a CRC16 algorithm.
CmpChecksumCrc32.c	The component's source file that implements a CRC32 algorithm.
CmpCommunicationLib	The communication library component folder containing implementation of the generic network address utility interface.
CmpCommunicationLib.c	The root source file of the communication library component.
CmpDevice	The device access component folder. This component is responsible for initial interactions between a remote client and RTS.
CmpDevice.c	The root source file of the device access component containing implementation of the device identification getter function.
CmpDeviceSrv.c	The device access online server source file.
CmpHeapPool	The buddy allocator implementation folder. This allocator is recommended for use in MicroRTS to replace malloc/free implemented in the standard C-library to reduce code size and to introduce additional flexibility.
CmpHeapPool.c	The buddy allocator component source root file.
CmplecTask	The IEC Tasks Management component folder. This component implements an IEC tasks management interface handling state of each IEC task and providing a factory for creating/deleting task instances.
CmplecTask.c	The root source file of the IEC Task Management component.
CmpIoDrvlec	The IEC I/O Drivers Management component folder. This component provides access to I/O drivers written in IEC (CODESYS) from the other RTS components written in C.
CmpIoDrvlec.c	The IEC I/O Drivers Management component root source file.
CmpIoMgrEmbedded	The I/O Manager component folder. The I/O Manager loads I/O drivers, controls the life-cycle of I/O device objects and provides read/write access

Folder/File	Description
	to I/O channels of device objects from an IEC application.
CmpIoMgrEmbedded.c	The I/O Manager root source file.
CmpLogEmbedded	The Logger component folder. The Logger is intended to store text messages issued by the RTS components for tracing the execution of RTS and an IEC application.
CmpLogEmbedded.c	The Logger component root source file.
CmpLogEmbeddedSrv.c	The Logger online service implementation file.
CmpMemPool	The Memory Pool Manager (Fixed-size Memory Blocks Manager) component folder. This component serves as a factory of allocators which provide memory blocks of fixed size out of static buffers or from the free storage (i.e. heap). Each allocator instance can also be used as a sequential container for objects occupying allocated memory blocks.
CmpMemPool.c	The Memory Pool Manager root source file containing two different implementations of the component interface.
FixedBlocksAllocator.c	A stand-alone implementation of the fixed-size memory blocks allocator. The FIXED_BLOCK_ALLOCATOR_SEPARATED macro should be specified in the build configuration to activate this functionality, if the RTS_MEMPOOL_VER2 is not specified. If the RTS_MEMPOOL_VER2 macro is specified in the build configuration, this module should be excluded from the build configuration.
FixedBlocksAllocator.h	The fixed-size memory blocks allocator stand-alone interface header file.
CmpMonitor	The Online Monitoring component folder. This component provides remote monitoring of variables in an IEC application.
CmpMonitor.c	The Online Monitoring component root source file.
CmpNameServiceServer	The Name and Address Resolution component folder. This component resolves the RTS network node name and address.
CmpNameServiceServer.c	The component root source file implementing the naming service interface.
CmpRouterEmbedded	The communication router component folder. The router component is in charge for transferring incoming packets to either its local network services server or to another router located on some other network node.
CmpRouterEmbedded.c	The component root source file.
CmpRouterEmbeddedAddrSrv.c	The router packets dispatcher implementation.
CmpRouterEmbeddedAddrSrv.h	The packets dispatcher interface header file used by the router.
CmpScheduleEmbedded	The IEC tasks embedded scheduler component folder.
CmpScheduleEmbedded.c	The IEC tasks scheduler root source file.
CmpSettingsEmbedded	The RTS settings reader/writer component folder. In a "file-less" RTS implementation, it only allows to fetch the values defined as static arrays of (key, value) entries in the RTS sysdefines.h header file.
CmpSettingsEmbedded.c	The component root source file.
CmpSettingsEmbedded.h	The header file shared between CmpSettingsEmbedded.c and CmpSettingsEmbeddedSrv.c containing the CmpSettingsEmbeddedSrv interface functions.
CmpSettingsEmbeddedSrv.c	The settings component online service implementation file.
CmpSrv	The CODESYS network protocol application layer server folder. This component is responsible for registration and invocation of application services available in RTS.
CmpSrv.c	The application layer implementation component root source file.
ComponentManager	The MicroRTS Component Manager and utilities folder. The Component Manager is responsible for the components life-cycle including components start-up, initialization and cyclical invocation of the components HookFunction. The Component Manager also resolves references to external library functions in an IEC code.

Folder/File	Description
CMMicro.c	The MicroRTS Component Manager source file.
CMUtils.c	The source file containing various string and number manipulations utility functions used across the other RTS components.
CMDep.h	The Component Manager dependency header file generated out of CMDep.m4.
CMDep.m4	The Component Manager dependency file containing the list of system components and their interface functions required for the Component Manager operation.
CMConvToStr.c	The source file implementing portable formatted string output routines. The PREFER_PORTABLE_SNPRINTF macro should be defined in the MicroRTS build configuration to use these routines instead of their equivalents implemented in the standard C library, which require more than 30 kB of code memory.
CMConvToStr.h	The portable formatted string output routines declarations.
SysCom	This folder contains a platform-independent root source file of the system component defining an interface to a serial port.
SysCom.c	The SysCom component root source file containing the SysCom external library functions (which are disabled in MicroRTS). In order to support the SysCom interface functions, the corresponding platform-dependent part should implement these functions in the SysCom secondary module (e.g. SysComCortexM3.c).
SysCpuHandling	This folder contains a platform-independent root source file of the system component defining an interface to CPU specific routines (SysCpuCalllecFuncWithParams, SysCpuGetCallstackEntry, SysCpuTestAndSetBit, SysCpuAtomicAdd, etc.).
SysCpuHandling.c	The SysCpuHandling component root source file containing the SysCpuHandling external library functions and platform-independent implementation of some CPU specific routines. To support the SysCpuHandling interface functions, the corresponding platform-dependent part should be implemented in the SysCpuHandling secondary module (e.g. SysCpuHandlingCortexM3.c).
SysExcept	The SysExcept component folder. This component is responsible for handling exceptions, which could occur during the execution of an IEC application. It is also contains functions for generating software exceptions.
SysExcept.c	The SysExcept component root source file. In order to support the SysExcept functionality, the corresponding platform-dependent part should implement platform-specific exceptions handling in the SysExcept secondary module (e.g. SysExceptCortexM3.c).
SysExcept.h	The SysExcept component internal header file used between the component root and secondary source files.
SysFileFlash	The SysFileFlash component folder. This component is responsible for mapping a set of flash memory regions to a list of file names to be able to emulate a very simple file system.
SysFileFlash.c	The SysFileFlash component root source file implementing all necessary functionality using the SysFlash component interface. The platform-specific configuration header file (sysdefines.h) should contain the FILE_MAP macro defining a table that maps a set of file names to a corresponding set of flash memory regions.
SysFlash	The SysFlash component folder. The SysFlash component provides a set of interface functions for accessing a flash memory device on a diskless target.
SysFlash.c	The SysFlash component root source file. To support the SysFlash interface functions, the corresponding platform-dependent part should implement them in the SysFlash secondary module (e.g. SysFlashCortexM3.c).
SysInt	The SysInt component folder. This component provides interface functions implementing a global lock/unlock mechanism and interrupts handling.

Folder/File	Description
SysInt.c	The SysInt component root source file. To support the SysInt interface functions, the corresponding platform-dependent part should implement them in the SysFlash secondary module (e.g. SysIntCortexM3.c).
SysInternalLib	The SysInternalLib component folder. This component provides a generic implementation of operations on some scalar data types, which are not fully supported by the CODESYS code generator used for producing an IEC application binary code.
SysInternalLibDefault.c	The SysInternalLib component root source file containing a generic implementation of FPU and long scalar types operations, which are exported and dynamically linked to an IEC application. This generic implementation can be overridden in the platform-specific secondary source file. All operations on 64-bit integer and real data types are excluded from the MicroRTS profile to reduce memory consumption.
SysMem	The SysMem component folder. The SysMem component provides a set of interface functions for allocating blocks of memory from special areas: free storage (heap), data area, code area and retain memory area.
SysMem.c	The SysMem component root source file. To support the SysMem interface functions, the corresponding platform-dependent part should implement some of them in the SysMem secondary module (e.g. SysMemCortexM3.c). It is recommended not to use malloc/free functions available in the standard C library. The CmpHeapPool component can be used for implementation of the SysMemAllocData/SysMemFreeData routines.
SysSocketEmbedded	The SysSocketEmbedded component folder. This component serves as a minimal implementation of the sockets networking interface.
SysSocketEmbedded.c	The SysSocketEmbedded component root source file containing a set of routines and data structures required to have a sockets-based networking interface relying on target specific functions for sending/receiving packets via a network adapter. These functions should be implemented in the SysSocketEmbedded secondary module (e.g. SysSocketEmbeddedCortexM3.c).
SysTarget	The SysTarget component folder. This component is responsible for the target device identification in CODESYS networks.
SysTarget.c	The SysTarget component root source file implementing most of the required functionality. The target specific header file (sysdefines.h) should contain a set of macros identifying the target: SYSTARGET_SIGNATUREID, SYSTARGET_DEVICE_TYPE, SYSTARGET_VENDOR_ID, SYSTARGET_DEVICE_ID, SYSTARGET_DEVICE_NAME, SYSTARGET_DEVICE_VERSION. Several functions exposed by the SysTarget interface should be implemented in the SysTarget secondary module (e.g. SysTargetCortexM3.c).
SysTime	The SysTime component folder. The SysTime component exposes functions to gain access to system millisecond-based (and, optionally, microseconds-based) ticks.
SysTime.c	The SysTime component root source file containing the CODESYS SysTime external library functions needed for the runtime operation. The system tick interface functions should be implemented in the platform-specific secondary module (e.g. SysTimeCortexM3.c).

### 3.4.3 Platform Specific Source Files

#### 3.4.3.1 Overview

The platform specific source and other auxiliary files, which implement interfaces exposed by system components (Sys) are located in the CODESYSV3\Platforms\Native\CortexM3 folder. This folder contains the following sub-folders:

1. *NXP-LPC1768* – MicroRTS system-dependent source and other files specific to the HITEX LPC1768-Stick target device.

2. *TI-LM3S9B96\_uRTS* – MicroRTS system-dependent source and other files specific to the Stellaris DK-LM3S9B96 evaluation board target device.
3. *Tools* – auxiliary tools and libraries needed for cross-development.

Each of the two platform-specific folders has the following common structure:

*DeviceDescription* – this folder contains a device description file for the target device implemented in the containing platform-specific folder.

*Projects* – this folder contains sub-folders for various toolchains used for building the RTS binary executable. Cortex-M3 MicroRTS currently supports only the CodeSourcery toolchain, so the only sub-folder of the *Project* folder is *CodeSourcery*.

*rtsconfig* – this folder contains two header files: the components list header file and the component exclusion list header file. The component list header file consists of the table of components, which are supposed to be available in the runtime system. The component exclusion list header file contains a set of <COMPONENT\_NAME>\_NOTIMPLEMENTED macros specifying components, which should be excluded from the runtime system.

*Sys* – this folder contains a set of platform-specific component secondary modules, i.e. C modules related to the system components in which some platform-specific functionality is implemented.

*sysdefines.h* – contains a set of configuration macros specific for the certain runtime system build configuration.

*syspecific.h* – contains a set of macros, which define the target memory map according to the target memory configuration specified in the linker configuration file (.ld). This header file also contains some additional platform-specific macros.

*targetdefines.h* – can contain an additional set of target specific macros.

The following sub-sections describe the contents of *Projects*, *rtsconfig* and *Sys* folders for each of the two target devices supported in the Cortex-M3 MicroRTS Starter Package

### 3.4.3.2 NXP-LPC1768 Folder Content

Folder/File	Description
Projects	Toolchain specific root folder
CodeSourcery	CodeSourcery toolchain specific folder
CODESYS.ld	The GNU linker configuration file containing descriptions of memory regions reserved for code and data areas in ROM and SRAM.
CODESYS.map	The map file produced by the GNU linker. This file is replaced with the new one every time the target binary is rebuilt.
debug.gdbinit	The GNU debugger (gdb) configuration file used for debugging the binary executable in the target device.
flash.gdbinit	The GNU debugger (gdb) configuration file used for re-flashing the binary executable in the target device without touching the flash memory region containing a CODESYS application.
flash_all.gdbinit	The GNU debugger (gdb) configuration file used for re-flashing the binary executable in the target device after erasing the entire flash memory.
GDBDebug.bat	The command file for initiating a gdb debugging session with the target. The OCDServer.bat should be successfully launched before starting a debugging session.
GDBFlash.bat	The command file for re-flashing the binary executable in the target device without touching the flash memory region containing a CODESYS application. The OCDServer.bat should be successfully launched before attempting to start this command file.
GDBFlash_all.bat	The command file for re-flashing the binary executable in the target device after erasing the entire flash memory. The OCDServer.bat should be successfully launched before attempting to start this command file.
imagestats.bat	The command file for creating stat.txt and report.log files. The .NET Framework 4.0 and the TOOLCHAIN_PATH environment variable referring



Folder/File	Description
	to the CodeSourcery toolchain installation folder should be available on a PC before launching this command file.
MakeAll.bat	The command file for rebuilding the binary executable. The TOOLCHAIN_PATH environment variable referring to the CodeSourcery toolchain installation folder should exist before launching this file.
Makefile	The make file used for building the RTS binary executable. If you wish to change the content of the runtime binary executable, please modify this file accordingly along with modifying the components list and components exclusion list header files.
OCDServer.bat	The command file for establishing an OCD connection with a target device connected to an USB port of a development PC. The OCD connection with a target is needed every time you need to re-flash the target device or to initiate a debugging session. Please launch this file prior to starting any of these command files: GDBDebug.bat, GDBFlash.bat, GDBFlash_all.bat.
Readme.txt	The MicroRTS Features description file.
report.log	The memory consumption report file generated by imagestats.bat. This file is created by processing the stat.txt detailed memory map and can be used for analyzing the ROM and SRAM utilization.
stat.txt	The detailed memory file produced from the binary executable with the GNU nm.exe utility.
src	LPC1768-Stick target device specific source files folder.
makedefs	Common definitions for the GNU make utility.
Source	LPC1768-Stick target device specific source files root folder.
Ethernet	Not used
IO	LPC17xx GPIO-subsystem related sources folder.
io_ports.c	Not used (specific for the LPC1768-Stick Demo application)
io_ports.h	Not used
lpc17xx_GPIO.c	LPC1768 GPIO driver
lpc17xx_GPIO.h	LPC1768 GPIO driver interface header
Stick	HITEX LPC1768-Stick configuration source files.
ConfigStick.c	LPC1768-Stick peripherals initialization and configuration routines.
ConfigStick.h	LPC1768-Stick peripherals initialization and configuration routines declarations
System	NXP LPC17xx and HITEX LPC1768-Stick system drivers folder.
core_cm3.c	Cortex-M3 core utilities implementation.
core_cm3.h	Cortex-M3 core utilities interface header.
cortexm3_fault.S	Cortex-M3 hard and soft fault handling entry points.
cortexm3_macro.S	Cortex-M3 core instructions implementations
defines.h	LPC1768-Stick demo software common type definitions.
interrupt.h	Interrupt handlers declarations.
LED_indi.c	LPC1768-Stick LED control utility.
LED_indi.h	LPC1768-Stick LED control interface header file.
LPC17xx.h	Cortex-M3 Core Peripheral Access Layer Header File
lpc17xx_clkpwr.c	LPC17xx clock and power control routines.
lpc17xx_clkpwr.h	LPC17xx clock and power control routines declarations.
lpc17xx_flsh.c	LPC17xx IAP (In-Application-Programming) routines for re-flashing.
lpc17xx_flsh.h	LPC17xx IAP routines declarations.
lpc17xx_nvic.c	LPC17xx NVIC access functions.

Folder/File	Description
lpc17xx_nvic.h	LPC17xx NVIC access functions declarations.
lpc17xx_pinsel.c	LPC17xx pins configuration functions.
lpc17xx_pinsel.h	LPC17xx pins configuration functions declarations.
startup.c	Interrupt Vector Table, stack definition, and the sysInit() routine called from main() and used for initializing the target device.
system_LPC17xx.c	LPC17xx peripheral access source file.
system_LPC17xx.h	Peripheral access declarations.
UART	LPC17xx UARTs access sources folder.
lpc17xx_libcfg_default.c	Not used
lpc17xx_libcfg_default.h	Not used
lpc17xx_uart.c	LPC17xx UARTs generic support library.
lpc17xx_uart.h	LPC17xx UARTs generic support library declarations.
UART.c	LPC1768-Stick UARTs support library.
UART.h	LPC1768-Stick UARTs support library declarations.
UIR	Not used
USB	Not used
rtsconfig	LPC1768-Stick target device build configuration folder
CortexM3_NotImpl.h	This header file contains a set of <COMPONENT_NAME>_NOTIMPLEMENTED macros, which specify components dependencies that have to be excluded from the RTS source files.
uRtsCortexM3.h	The components list header file containing a set of components to be included to the build configuration. This file is included to build using the following definition in the sysdefines.h header file: #ifndef RTS_CONFIG_FILE # define RTS_CONFIG_FILE rtsconfig/uRtsCortexM3.h #endif
Sys	Components platform-specific secondary modules folder.
CmpSettingsCortexM3.c	Platform specific implementation of target settings.
MainCortexM3.c	The MicroRTS binary executable entry point (main()) is located in this module.
SysComCortexM3.c	The SysCom component platform-dependent implementation.
SysCpuHandlingCortexM3.c	The SysCpuHandling component platform-dependent implementation.
SysExceptCortexM3.c	The SysExcept component platform-dependent implementation.
SysFlashCortexM3.c	The SysFlash and SysFileFlash components platform-dependent implementation.
SysIntCortexM3.c	The SysInt component platform-dependent implementation.
SysSocketEmbeddedCortexM3.c	Not used
SysTargetCortexM3.c	The SysTarget component platform-dependent implementation.
SysTimeCortexM3.c	The SysTime component platform-dependent implementation.

### 3.4.3.3 TI-LM3S9B96\_uRTS Folder Content

Folder/File	Description
Projects	Toolchain specific root folder
CodeSourcery	CodeSourcery toolchain specific folder
CODESYS.ld	The GNU linker configuration file containing descriptions of memory regions reserved for code and data areas in ROM and SRAM.
CODESYS.map	The map file produced by the GNU linker. This file is replaced with the new

Folder/File	Description
Projects	Toolchain specific root folder
	one every time the target binary is rebuilt.
debug.gdbinit	The GNU debugger (gdb) configuration file used for debugging the binary executable in the target device.
flash.gdbinit	The GNU debugger (gdb) configuration file used for re-flashing the binary executable in the target device with cleaning the flash memory region containing a CODESYS application.
GDBDebug.bat	The command file for initiating a gdb debugging session with the target. The OCDServer.bat should be successfully launched before starting a debugging session.
GDBFlash.bat	The command file for re-flashing the binary executable in the target device with cleaning the flash memory region containing a CODESYS application. The OCDServer.bat should be successfully launched before attempting to start this command file.
imagestats.bat	The command file for creating stat.txt and report.log files. The .NET Framework 4.0 and the TOOLCHAIN_PATH environment variable referring to the CodeSourcery toolchain installation folder should be available on a PC before launching this command file.
MakeAll.bat	The command file for rebuilding the binary executable. The TOOLCHAIN_PATH environment variable referring to the CodeSourcery toolchain installation folder should exist before launching this file.
Makefile	The make file used for building the RTS binary executable. If you wish to change the content of the runtime binary executable, please modify this file accordingly (along with modifying the components list and components exclusion list header files).
OCDServer.bat	The command file for establishing an OCD connection with a target device connected to an USB port of a development workstation. The OCD connection with a target is needed every time you need to re-flash the target device or to initiate a debugging session. Please launch this file prior to starting any of these command files: GDBDebug.bat, GDBFlash.bat, GDBFlash_all.bat.
Readme.txt	The MicroRTS Features description file.
report.log	The memory consumption report file generated by imagestats.bat. This file is created by processing the stat.txt detailed memory map and can be used for analyzing the SRAM and flash consumption.
stat.txt	The detailed memory file produced out of the binary executable with the GNU nm.exe utility.
src	Stellaris DK-LM3S9B96 target device specific source files folder.
cortexm3_fault.S	Cortex-M3 hard and soft fault handling entry points.
startup.c	Interrupt Vector Table, stack definition, and the sysInit() routine called from main() and used for initializing the target device.
rtsconfig	Stellaris DK-LM3S9B96 target device build configuration folder
CortexM3_NotImpl.h	This header file contains a set of <COMPONENT_NAME>_NOTIMPLEMENTED macros which specify components dependencies that have to be excluded from the RTS source files.
uRtsCortexM3.h	The components list header file containing a set of components to be included to the build configuration. This file is included to the build process using the following definition in the sysdefines.h header file: #ifndef RTS_CONFIG_FILE # define RTS_CONFIG_FILE rtsconfig/uRtsCortexM3.h #endif
Sys	Components platform-specific secondary modules folder.
CmpSettingsCortexM3.c	Platform specific implementation of target settings.
MainCortexM3.c	The MicroRTS binary executable entry point (main()) is located in this

Folder/File	Description
Projects	Toolchain specific root folder
	module.
SysComCortexM3.c	Not used.
SysCpuHandlingCortexM3.c	The SysCpuHandling component platform dependent implementation.
SysExceptCortexM3.c	The SysExcept component platform dependent implementation.
SysFlashCortexM3.c	The SysFlash and SysFileFlash components platform dependent implementation.
SysIntCortexM3.c	The SysInt component platform dependent implementation.
SysSocketEmbeddedCortexM3.c	The SysSocketEmbedded component platform dependent implementation.
SysTargetCortexM3.c	The SysTarget component platform dependent implementation.
SysTimeCortexM3.c	The SysTime component platform dependent implementation.

### 3.5 Build Utilities

The CODESYSV3\BuildUtils folder contains the m4 preprocessor, command line scripts to invoke it and common m4 definition files used for transforming the components interface and dependency m4-files to the corresponding interface and dependency header files. For information about the m4 mechanism, please refer to section 2.8 of the CODESYS Control V3 Manual.

## 4 Architecture

### 4.1 Overview

MicroRTS is a special profile of the CODESYS V3 runtime system (RTS) intended to be used on IEC 61131-3 programmable embedded devices and PLCs with limited sizes of permanent (ROM) and volatile (SRAM) memory.

MicroRTS is considered as a special profile because the MicroRTS binary executable is built from the same source tree that is used for building the full and compact RTS binaries.

This chapter describes the set of features supported by MicroRTS, differences between MicroRTS and other profiles of the CODESYS V3 RTS, and the MicroRTS architecture.

The MicroRTS architecture is derived from the common CODESYS V3 RTS architecture having in mind restrictions on memory consumption. For detailed information on the CODESYS V3 RTS architecture, please refer to section 2 of the CODESYS Control V3 Manual.

### 4.2 MicroRTS Features

MicroRTS is based on the Compact RTS profile and currently supports the following set of features:

- Downloading an IEC 61131-3 user application created with CODESYS V3 IDE to the flash memory of a target device.
- In-flash execution of the user application emulating an IEC multi-tasking in a single-tasking environment.
- Online Monitoring of variables.
- Changing values of variables by writing and forcing.
- Remote control of an IEC application execution including Start/Stop and Single Cycle.
- Exceptions handling in an IEC code.
- Support for logging messages generated by various RTS components.

Some features typically available in the full RTS are **not supported**:

- Online change of an IEC application.
- System event generation and handling.
- Asynchronous operations execution.
- Debugging an IEC application in CODESYS.
- User management.
- Tracing.
- PlcShell.
- File transfer.
- 64-bit data integer and real types (LWORD, LREAL).
- Most core external libraries are excluded from the runtime system build configuration.

At least two features listed above **are considered to be supported** in future versions of MicroRTS:

- Online change of an IEC application – if a target device has enough flash memory, and if re-flashing doesn't require target interrupts to be disabled for too long.
- Debugging with breakpoints – if a target CPU supports hardware breakpoints.

### 4.3 Differences Between the MicroRTS and Other Runtime System Profiles

The MicroRTS architecture has some differences:

1. The MicroRTS binary executable can only be linked statically.
2. The Component Manager is replaced for the light-weight equivalent providing only a minimal required set of functionality. This functionality includes controlling the components initialization and calling to the components HookFunction cyclically only if it is needed.
3. The component base interface is reduced as each component has only a single mandatory interface function called HookFunction. The ComponentEntry, CreateInstance, DeleteInstance, ExportFunctions, ImportFunctions and GetVersion are excluded from the components source code in the MicroRTS profile.
4. The Component Manager doesn't dynamically export any interface functions for the components composing the MicroRTS. The two Component Manager functions required to MicroRTS components (CMMicroGetAPI and CMMicroCallHook) are exported statically.
5. External library functions exported by MicroRTS components are linked with the Component Manager statically. The components implementing some external library functions contain tables of these functions generated from the components list and from the components interface definition files (*<ComponentName>Itf.m4*). These tables are imported to the Component Manager statically in compile time. When an IEC application is downloaded to a target device, the CMMicroGetAPI function is used to resolve references to these functions in the downloaded IEC code.

The next section describes the MicroRTS components layout in greater details.

## 4.4 Components Management

### 4.4.1 Component Defined

The CODESYS V3 runtime system is based on the component model in which each component is represented by an isolated set of source modules written in C. The component's source modules implement a piece of functionality and expose the component's interface containing functions which may be called by other components.

There are two types of components:

1. Functional components – platform-independent components implementing functionality specific for PLCs and/or control systems. The functional components contain a 'Cmp' prefix in their names.
2. System components – platform-dependent components, which implement interfaces allowing functional components to interact with environment of the runtime system (e.g. with an underlying operating system, specific runtime library, or hardware). The system components contain a 'Sys' prefix in their names.

The main difference between a functional and a system component is that a system component always has platform-dependent part of implementation.

There are also core and optional components. The former are the components without which it is impossible to get the runtime system running. The latter are the components which can be excluded from the runtime system build configuration.

A component's public interface is represented by a set of functions specific to the component's functionality and declared in the component's interface description file *<ComponentName>Itf.m4*. This file is transformed by the m4 preprocessor to a component's interface header file *<ComponentName>Itf.h*. If some component is supposed to be part of MicroRTS, it should also implement one mandatory private interface function called HookFunction which can only be accessed by the Component Manager for controlling the component life-cycle.

If some component needs to call interface functions of some other component that means the first component depends on the second one. The component's dependencies are expressed in the

component's dependency description file *<ComponentName>Dep.m4*. This file is transformed by the m4 preprocessor to the component's dependency header file *<ComponentName>Dep.h*.

For more information about the CODESYS V3 RTS component architecture please refer to sections 2.7 and 2.8 of the CODESYS Control V3 Manual.

## 4.4.2 Component Source Code Organization

### 4.4.2.1 Component Source Code

There are three types of source files mentioned in this document:

1. C-modules: source files with the .c extension containing definitions of variables and functions.
2. Header files: source files with the .h extension containing declarations of data types, functions and macro definitions.
3. Component description files: source files with the .m4 extension containing components interface descriptions and components dependencies declarations.

The components source code is generally split in two parts. The first one resides in the *Components* sub-folder (refer to section 3.4 for more information on folders organization) and contains platform-independent source files. Each component has two files: *<ComponentName>Itf.m4* and *<ComponentName>Itf.h* located in the *Components* folder and its own separate source folder matching to the component name that contains source files related to this component.

For example, the CmpAppEmbedded component (which is a core functional RTS component that handles an IEC application in the Compact and MicroRTS profiles) has the following sources structure:

Components	Platform-independent sources folder
CmpAppEmbedded	The IEC Application Manager (embedded version) component folder. The IEC Application Manager is responsible for controlling the IEC application downloaded to the target and application code and data handling. It also implements the application specific online services as downloading, debugging, start/stop, etc.
CmpAppEmbedded.c	The IEC Application Manager root source file.
CmpAppEmbeddedSrv.c	The IEC Application Manager online services implementation file.
CmpAppEmbedded.h	The IEC Application Manager functions declarations shared among the root source file and the online services implementation file.
CmpAppEmbeddedDep.h	The IEC Application Manager dependency header file automatically generated from the corresponding dependency description file.
CmpAppEmbeddedDep.m4	The IEC Application Manager dependency description file.
CmpAppItf.m4	The IEC Application Manager interface description file.
CmpAppItf.h	The IEC Application Manager interface header file automatically generated from the corresponding interface description file.

The second part of component's source code which depends on a specific platform is located in the corresponding sub-folder of the *Platform* folder. For example, the SysFlash component, which is a core system component of the Cortex-M3 MicroRTS, has the following sources structure:

Components	Platform-independent sources folder
SysFlash	The SysFlash component folder. The SysFlash component provides a set of interface functions for accessing a flash memory device on a diskless target.
SysFlash.c	The SysFlash component root source file. To support the SysFlash interface functions, the platform dependent part should implement them in the SysFlash secondary module (e.g. SysFlashCortexM3.c).
SysFlashDep.h	The SysFlash component dependency header file automatically generated from the corresponding dependency description file.
SysFlashDep.m4	The SysFlash component dependency description file.
SysFlashItf.m4	The SysFlash component interface description file.

SysFlashItf.h	The SysFlash component interface header file automatically generated from the corresponding interface description file.
Platforms	Platform-dependent sources folder
Native	Platform-dependent sources folder for targets without any OS
CortexM3	Common source folder for Cortex-M3-based CPUs
NXP-LPC1768	NXP LPC1768 microcontroller sources folder
Sys	Platform-dependent sources of system components
SysFlashCortexM3.c	The SysFlash component platform-dependent secondary source file.

The component's C-module that resides in the component's folder and contains the platform-independent part of implementation of the component's interface functions including the mandatory HookFunction and the component's export table is further called a component **root source file** or a component **root module**.

Each system component additionally has at least one platform-dependent C-module, which implements the component's interface in a platform-specific way. This C-module is further called a component **secondary module**. The system components root modules are typically associated with the secondary modules by the "implemented-in-terms-of" relation. For example, the SysFlash component root module SysFlash.c contains a call in its HookFunction to the corresponding function located in the secondary module:

```
/* The root header file. Should be included before any other header files */
#include "CmpStd.h"

/* The SysFlash component dependency header file */
#include "SysFlashDep.h"

/* This macro may only be used in the components root modules. */
/* It instantiates the component's interface object and the export table */
USE_STMT

/* the SysFlash component mandatory interface function */
static RTS_RESULT CDECL
HookFunction(RTS_UI32 ulHook, RTS_UINTPTR ulParam1, RTS_UINTPTR ulParam2)
{
    RTS_RESULT Result = ERR_OK;

    switch (ulHook)
    {
        /* ... other processing of system phases */
    }

    if(Result == ERR_OK)
    {
        /*
         * call to the component function located in the secondary module
         * which implements the component's functionality in a platform
         * specific way.
         */
        return SysFlashOSHHookFunction(ulHook, ulParam1, ulParam2);
    }

    return Result;
}
```

Another example demonstrating the "implemented-in-terms-of" relation is taken from the same SysFlash component root module (SysFlash.c):



```
/* This is a platform-independent part of the SysFlashErase interface function */
/* located in the SysFlashErase component root module SysFlash.c. */
RTS_RESULT CDECL SysFlashErase(FlashArea fa, RTS_SIZE ulSize, RTS_SIZE ulOffset)
{
    RTS_RESULT Result;
    RTS_SIZE ulToWork;
    RTS_SIZE ulWorked = 0;

    do
    {
        CMCallExtraCommCycleHook(CM_HOOK_TYPE_FLASH_ACCESS, 0);

        if (ulSize - ulWorked >= s_ulEraseBlockSize)
            ulToWork = s_ulEraseBlockSize;
        else
            ulToWork = ulSize - ulWorked;

        /* call to the component function that actually implements the flash erasing */
        /* functionality and is located in the component's secondary module */
        Result = SysFlashErase_(fa, ulToWork, ulOffset + ulWorked);

        ulWorked += ulToWork;
        if (Result != ERR_OK)
            break;

    } while (ulWorked < ulSize);

    return Result;
}
```

#### 4.4.2.2 Root Module Requirements

Each component of MicroRTS named *CmpName* shall have a root module. It is allowed to have one, and only one, root module.

The root module shall be placed to the component's platform-independent source folder (typically named *CmpName*).

The root module, as any other component's C-module, shall contain the following include directives at the beginning of it:

```
/* The root header file. Should be included before any other header files */
#include "CmpStd.h"

/* The CmpName component dependency header file */
#include "CmpNameDep.h"
```

The root module shall contain the `USE_STMT` macro placed at the beginning of it:

```
/* This macro may only be used in the components root modules. */
/* It instantiates the component's interface object and the export table */
USE_STMT
```

The root module shall contain the `HookFunction` interface function declared as follows:

```
/* the CmpName component mandatory interface function */
static RTS_RESULT CDECL
HookFunction(RTS_UI32 ulHook, RTS_UINTPTR ulParam1, RTS_UINTPTR ulParam2)
{
    RTS_RESULT result = ERR_OK;

    switch (ulHook)
    {
        /* processing the life-cycle phases of the runtime system */
    }

    return result;
}
```

The `HookFunction` function of a system component root module shall contain at least one call to the hook function *CmpNameOSHookFunction* defined in the secondary module of this component.

#### 4.4.2.3 Secondary Module Requirements

Each system component of MicroRTS named *CmpName* shall have a secondary module. It is a common practice to have only a single secondary module in the core runtime system components, but OEM-specific system components might have any number of secondary modules, if they are manually hooked to the corresponding root module with a chain of calls.

The secondary module shall be placed to the component's platform-dependent source folder.

The secondary module shall contain the following include directives at the beginning of it:

```
#include "CmpStd.h"

#ifdef PATHS_RELATIVE
#include "CmpName/CmpNameDep.h"
#else
#include "CmpNameDep .h"
#endif
```

If the secondary module is supposed to be used in the full and/or compact profile of the runtime system, the secondary module shall contain the `USEIMPORT_STMT` macro placed at the beginning of it:

```
/* This macro may only be used in the components secondary modules. */
USEIMPORT_STMT
```

The secondary module shall contain the *CmpName*OSHookFunction declared as follows:

```
/* the CmpName system component platform-specific implementation hook */
static RTS_RESULT CDECL
CmpNameOSHookFunction(RTS_UI32 u1Hook, RTS_UINTPTR u1Param1, RTS_UINTPTR u1Param2)
{
    RTS_RESULT result = ERR_OK;

    switch (u1Hook)
    {
        /* processing life-cycle phases of the runtime system */
    }

    return result;
}
```

If some component is supposed to be part of MicroRTS profile only, the `USEIMPORT_STMT` macro can be omitted since it is expanded to an empty line in the MicroRTS profile

#### 4.4.2.4 Subordinate Modules Requirements

The rest C-modules placed to the component's source folders, either platform-independent or platform-specific, are further called subordinate modules. These C-modules are typically added to the component's source folders to improve modularity.

Any system component of MicroRTS may have any number of subordinate modules.

If it is required for some component's subordinate module to access functions listed in the component's dependency description file, the following directives should be placed at the beginning of it:

```
#include "CmpStd.h"

#ifdef PATHS_RELATIVE
#include "CmpName/CmpNameDep.h"
#else
#include "CmpNameDep .h"
#endif
```

```
USEEXTERN_STMT
```

If some component is supposed to be part of MicroRTS profile only, the `USEEXTERN_STMT` macro can be omitted since it is expanded to an empty line in the MicroRTS profile.

## 4.4.3 Simplified Component Manager

### 4.4.3.1 Overview

The MicroRTS profile uses a special simplified version of the Component Manager that is implemented in the Components\ComponentManager\CMMicro.c module.

The MicroRTS Component Manager was designed to minimize the number of interactions between the Component Manager and the MicroRTS components and to reduce the Component Manager code size.

The MicroRTS Component Manager implements the following functionality:

1. Statically instantiates the table of components to be included to the MicroRTS binary executable. This table is represented by the implicitly defined `s_ComponentList` static array of type `CMP_DESCRIPTOR` declared in Components\\_OptionalIncludes\Profiles\urTS\CMMicro.h. The array is generated from the components list header file specified by the `RTS_CONFIG_FILE` macro in either the system build configuration header file `sysdefines.h` or directly by the `-D` or `/D` compile switch. The array generation is activated by successive application of the `INCLUDE_CMP_HOOK_DECLARATIONS` aspect macro and the `INCLUDE_CMP_HOOKS_LIST` aspect macro in Components\\_OptionalIncludes\Profiles\urTS\Aspects\uCmpListDecl.h:

```
/* Declare references to the components HookFunction routines and to the */
/* components export tables. The INCLUDE_CMP_HOOKS_DECLARATIONS aspect */
/* is active here (see uCmpListAsp.h). */
#define INCLUDE_CMP_HOOKS_DECLARATIONS
# include <OptionalIncludes/Profiles/urTS/Aspects/uCmpListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
#undef INCLUDE_CMP_HOOKS_DECLARATIONS

/* Define a table of components The INCLUDE_CMP_HOOKS_LIST aspect */
/* is active here (see uCmpListAsp.h). */
#define INCLUDE_CMP_HOOKS_LIST
# include <OptionalIncludes/Profiles/urTS/Aspects/uCmpListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
#undef INCLUDE_CMP_HOOKS_LIST
```

2. Statically instantiates the table of pointers to tables of function descriptors, each of which contains a set of objects referencing to the CODESYS external library functions implemented by the corresponding components. This table is represented by the `s_ComponentExternalsList` static array of type `CMP_EXT_FUNCTION_REF` declared in Components\CmpStd.h. The array is generated from the components list header file specified by the `RTS_CONFIG_FILE` macro in either the system build configuration header file `sysdefines.h` or directly by the `-D` or `/D` compile switch. The array generation is activated by application of the `INCLUDE_CMP_EXTERNALS_LIST` aspect macro in Components\\_OptionalIncludes\Profiles\urTS\Aspects\uCmpListDecl.h:

```
/* Define a table of components export tables. The INCLUDE_CMP_EXTERNALS_LIST */
/* aspect is active here (see uCmpListAsp.h). */
#define INCLUDE_CMP_EXTERNALS_LIST
# include <OptionalIncludes/Profiles/urTS/Aspects/uCmpListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
#undef INCLUDE_CMP_EXTERNALS_LIST
```

3. Defines the list of MicroRTS initialization phases which is used for components initialization. This list is represented by the `s_SystemPhases` static array of type `CMP_PHASE_DESCRIPTOR` declared locally in CMMicro.c:

```
/**
 * Hook types table (s_SystemPhases) containing
 * a list of hook types which are used during
 * the system initialization.
 */
CDS_DECLARE_HOOKS_LIST_BEGIN()
CDS_INSERT_HOOK(CH_INIT_SYSTEM)
CDS_INSERT_HOOK(CH_INIT_SYSTEM2)
CDS_INSERT_HOOK(CH_INIT)
CDS_INSERT_HOOK(CH_INIT2)
CDS_INSERT_HOOK(CH_INIT201)
CDS_INSERT_HOOK(CH_INIT3)
```

```

CDS_INSERT_HOOK(CH_INIT_TASKS)
CDS_INSERT_HOOK(CH_INIT_COMM)
CDS_DECLARE_HOOKS_LIST_END()

```

4. Initializes the runtime system. The CMMicroInit routine walks through the list of phases defined by the `s_SystemPhases` static array and call to the HookFunction routine of each component specified in the components table (`s_ComponentList`). The CMMicroInit routine, whose name is redefined to CMinInit in Components\\_OptionalIncludes\Profiles\urTS\CMMicro.h, is being called from the main() routine of the MicroRTS binary executable.
5. Provides access to the components HookFunction interface functions from the main() routine of the MicroRTS binary executable. For example, cyclical execution of some components HookFunction is implemented as follows:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Sys\MainCortexM3.c */
int main()
{
    RTS_RESULT Result;

    sysInit();

    /* CMinInit is actually a redefined name for CMMicroInit*/
    Result = CMinInit(NULL, s_ComponentList);

    while(!s_bExitLoop)
    {
        /* Cyclical execution of components HookFunction. */
        /* CMCallHook is a redefined name for CMMicroCallHook */

        CMCallHook( CH_COMM_CYCLE, 0, 0, FALSE) ;
    }

    return 0;
}

```

6. Provides access to the CODESYS external library functions implemented by the MicroRTS components. When an IEC application is downloaded to a device running MicroRTS, all references to external library functions in an IEC code are being dynamically resolved by function names and signature identifiers. This is done in the AppGetAPI routine of the IEC Application Manager as follows:

```

/* Components\CmpAppEmbedded\CmpAppEmbedded.c */
static RTS_RESULT CDECL AppGetAPI( char *pszAPIName,
                                   RTS_VOID_FCTPTR *ppfAPIFunction,
                                   RTS_UI32 *pulSignatureID,
                                   RTS_UI32 *pulVersion)
{
    RTS_RESULT Result;

    #ifndef RTS_COMPACT_MICRO
        /* Compact profile resolution... */
    #else
        Result = CMMicroGetAPI( pszAPIName,
                                ppfAPIFunction,
                                ((NULL != pulSignatureID) ? *pulSignatureID : 0));
    #endif

    switch (Result)
    {
        /* Result processing... */
    }

    return Result;
}

```

#### 4.4.3.2 Components List

The components list containing names of the components that need to be included to the MicroRTS binary executable is defined in a separate header file with the following layout:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\rtsconfig\urTsCortexM3.h */
/* WARNING! Please do not use a single include guard here! */
/* NXP LPC1768-Stick urTS Components List */

```

```
CDS3_DECLARE_COMPONENTS_LIST_BEGIN(uRtsCM3)
CDS3_ADD_COMPONENT_ENTRY(SysMem)
CDS3_ADD_COMPONENT_ENTRY(SysTime)
CDS3_ADD_COMPONENT_ENTRY(SysTarget)
CDS3_ADD_COMPONENT_ENTRY(SysCpuHandling)
CDS3_ADD_COMPONENT_ENTRY(SysInt)
CDS3_ADD_COMPONENT_ENTRY(SysCom)
CDS3_ADD_COMPONENT_ENTRY(SysFlash)
CDS3_ADD_COMPONENT_ENTRY(SysFileFlash)
CDS3_ADD_COMPONENT_ENTRY(SysExcept)
CDS3_ADD_COMPONENT_ENTRY(SysInternalLib)
CDS3_ADD_COMPONENT_ENTRY(CmpHeapPool)
CDS3_ADD_COMPONENT_ENTRY(CmpMemPool)
CDS3_ADD_COMPONENT_ENTRY(CmpLogEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpAppEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpAppForce)
CDS3_ADD_COMPONENT_ENTRY(CmpBinTagUtil)
CDS3_ADD_COMPONENT_ENTRY(CmpBlkDrvCom)
CDS3_ADD_COMPONENT_ENTRY(CmpChannelMgrEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpChannelServerEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpChecksum)
CDS3_ADD_COMPONENT_ENTRY(CmpCommunicationLib)
CDS3_ADD_COMPONENT_ENTRY(CmpDevice)
CDS3_ADD_COMPONENT_ENTRY(CmpIecTask)
CDS3_ADD_COMPONENT_ENTRY(CmpIoDrvIec)
CDS3_ADD_COMPONENT_ENTRY(CmpIoMgrEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpMonitor)
CDS3_ADD_COMPONENT_ENTRY(CmpNameServiceServer)
CDS3_ADD_COMPONENT_ENTRY(CmpRouterEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpScheduleEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpSettingsEmbedded)
CDS3_ADD_COMPONENT_ENTRY(CmpSrv)
CDS3_DECLARE_COMPONENTS_LIST_END(uRtsCM3)
```

There are two strict requirements for the components list header file:

1. The components list header file **should not** contain any single include guard directives preventing this header file from being included to some C module more than once. The components list header file is included to the CMMicro.c module multiple times intentionally.
2. System components should be placed at the top of the components list.

As stated previously, the components list header file should be specified as the `RTS_CONFIG_FILE` macro expansion value using one of the following ways:

1. In `sysdefines.h`

```
#ifndef RTS_CONFIG_FILE
# define RTS_CONFIG_FILE rtsconfig/uRtsCortexM3.h
#endif
```

2. As a compiler option:

```
CFLAGS+=-g -DRTS_CONFIG_FILE=rtsconfig/uRtsCortexM3.h -Dgcc ...
```

The component list header file is transformed in CMMicro.c to the set of components HookFunction functions forward declarations, to the table of component descriptors and to the table of pointers to tables of function descriptors, each of which contains a set of objects referencing to the CODESYS external library functions implemented by the corresponding components. This may seem to be confusing but uses a simple and powerful aspect-oriented approach:

1. The CMMicro.c module contains the following line:

```
/**
 * The following aspects are generated with this header file:
 * 1) Table of components (with components hook functions): s_ComponentList
 * 2) Table of components export tables: s_ComponentExternalsList
 */
#include <_OptionalIncludes/Profiles/uRTS/Aspects/uCmpListDecl.h>
```

2. The `uCmpListDecl.h` header file includes the components list header file three time in a row with different aspect macros activated before each subsequent inclusion:

```
/* This macro allows to use the path specified with the par */
```

```

/* parameter as part of #include preprocessor directive. */
#ifndef ANGLE_BRACKETS
# define __ANGLE_BRACKETS__(par) <par>
# define ANGLE_BRACKETS(par) __ANGLE_BRACKETS__(par)
#endif

#ifndef RTS_CONFIG_FILE
# error RTS_CONFIG_FILE should be specified containing the RTS components list file!
#endif

/* The components list defined in RTS_CONFIG_FILE is processed here using */
/* different aspects. */

/* Declare references to the components HookFunction routines and to the */
/* components export tables. The INCLUDE_CMP_HOOKS_DECLARATIONS aspect */
/* is active here (see uCmplListAsp.h). */
# define INCLUDE_CMP_HOOKS_DECLARATIONS
# include <OptionalIncludes/Profiles/uRTS/Aspects/uCmplListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
# undef INCLUDE_CMP_HOOKS_DECLARATIONS

/* Define a table of components The INCLUDE_CMP_HOOKS_LIST aspect */
/* is active here (see uCmplListAsp.h). */
# define INCLUDE_CMP_HOOKS_LIST
# include <OptionalIncludes/Profiles/uRTS/Aspects/uCmplListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
# undef INCLUDE_CMP_HOOKS_LIST

/* Define a table of components export tables. The INCLUDE_CMP_EXTERNALS_LIST */
/* aspect is active here (see uCmplListAsp.h). */
# define INCLUDE_CMP_EXTERNALS_LIST
# include <OptionalIncludes/Profiles/uRTS/Aspects/uCmplListAsp.h>
# include ANGLE_BRACKETS(RTS_CONFIG_FILE)
# undef INCLUDE_CMP_EXTERNALS_LIST

```

- The uCmplListAsp.h aspects definition header file expands the CDS3\_DECLARE\_COMPONENTS\_LIST\_BEGIN, CDS3\_DECLARE\_COMPONENTS\_LIST\_END and CDS3\_ADD\_COMPONENT\_ENTRY macros differently according to the aspect macro currently activated.
- If the INCLUDE\_CMP\_HOOKS\_DECLARATIONS macro is activated, the CDS3\_DECLARE\_COMPONENTS\_LIST\_BEGIN and CDS3\_DECLARE\_COMPONENTS\_LIST\_END macros are expanded to empty lines, and each CDS3\_ADD\_COMPONENT\_ENTRY macro is expanded to a pair of forward declarations:

```

extern RTS_RESULT CDECL ComponentName_HookFunction(RTS_UI32 ulHook,
                                                    RTS_UINTPTR ulParam1,
                                                    RTS_UINTPTR ulParam2);

extern const CMP_EXT_FUNCTION_REF ComponentName_ExternalsTable[];

```

The first forward declaration imports the *ComponentName* component HookFunction and the second forward declaration – the *ComponentName* component table of external library function descriptors.

- If the INCLUDE\_CMP\_HOOKS\_LIST macro is activated, the CDS3\_DECLARE\_COMPONENTS\_LIST\_BEGIN macro is expanded to the following definition:

```

static CMP_DESCRIPTOR s_ComponentList[] =
{

```

Each CDS3\_ADD\_COMPONENT\_ENTRY is expanded to the corresponding *s\_ComponentList* array element:

```

{ ComponentName_HookFunction, 0, ERR_OK },

```

The CDS3\_DECLARE\_COMPONENTS\_LIST\_END macro is expanded to the last element of the *sComponentsList* array followed by the array end token:

```

{ NULL, 0, ERR_OK }
};

```

As a result, the components list header file listed above is transformed to the following array:

```
static CMP_DESCRIPTOR s_ComponentList[] =
{
  { SysMem_HookFunction, 0, ERR_OK },
  { SysTime_HookFunction, 0, ERR_OK },
  { SysTarget_HookFunction, 0, ERR_OK },
  { SysCpuHandling_HookFunction, 0, ERR_OK },
  { SysInt_HookFunction, 0, ERR_OK },
  { SysCom_HookFunction, 0, ERR_OK },
  { SysFlash_HookFunction, 0, ERR_OK },
  { SysFileFlash_HookFunction, 0, ERR_OK },
  { SysExcept_HookFunction, 0, ERR_OK },
  { SysInternalLib_HookFunction, 0, ERR_OK },
  { CmpHeapPool_HookFunction, 0, ERR_OK },
  { CmpMemPool_HookFunction, 0, ERR_OK },
  { CmpLogEmbedded_HookFunction, 0, ERR_OK },
  { CmpAppEmbedded_HookFunction, 0, ERR_OK },
  { CmpAppForce_HookFunction, 0, ERR_OK },
  { CmpBinTagUtil_HookFunction, 0, ERR_OK },
  { CmpBlkDrvCom_HookFunction, 0, ERR_OK },
  { CmpChannelMgrEmbedded_HookFunction, 0, ERR_OK },
  { CmpChannelServerEmbedded_HookFunction, 0, ERR_OK },
  { CmpAppEmbedded_HookFunction, 0, ERR_OK },
  { CmpChecksum_HookFunction, 0, ERR_OK },
  { CmpCommunicationLib_HookFunction, 0, ERR_OK },
  { CmpDevice_HookFunction, 0, ERR_OK },
  { CmpIecTask_HookFunction, 0, ERR_OK },
  { CmpIoDrvIec_HookFunction, 0, ERR_OK },
  { CmpIoMgrEmbedded_HookFunction, 0, ERR_OK },
  { CmpMonitor_HookFunction, 0, ERR_OK },
  { CmpNameServiceServer_HookFunction, 0, ERR_OK },
  { CmpRouterEmbedded_HookFunction, 0, ERR_OK },
  { CmpScheduleEmbedded_HookFunction, 0, ERR_OK },
  { CmpSettingsEmbedded_HookFunction, 0, ERR_OK },
  { CmpSrv_HookFunction, 0, ERR_OK },
  { NULL, 0, ERR_OK }
};
```

6. If the `INCLUDE_CMP_EXTERNALS_LIST` macro is activated, the `CDS3_DECLARE_COMPONENTS_LIST_BEGIN` macro is expanded to the following definition:

```
static const CMP_EXT_FUNCTION_REF* s_ComponentExternalsList[] =
{
```

Each `CDS3_ADD_COMPONENT_ENTRY` is expanded to the corresponding `s_ComponentExternalsList` array element:

```
    ComponentName_ExternalsTable,
```

The `CDS3_DECLARE_COMPONENTS_LIST_END` macro is expanded to the last element of the `s_ComponentExternalsList` array followed by the array end token:

```
    NULL
};
```

As a result, the components list header file listed above is transformed to the following array:

```
static const CMP_EXT_FUNCTION_REF* s_ComponentExternalsList[] =
{
  SysMem_ExternalsTable,
  SysTime_ExternalsTable,
  SysTarget_ExternalsTable,
  SysCpuHandling_ExternalsTable,
  SysInt_ExternalsTable,
  SysCom_ExternalsTable,
  SysFlash_ExternalsTable,
  SysFileFlash_ExternalsTable,
  SysExcept_ExternalsTable,
  SysInternalLib_ExternalsTable,
  CmpHeapPool_ExternalsTable,
  CmpMemPool_ExternalsTable,
  CmpLogEmbedded_ExternalsTable,
```

```
CmpAppEmbedded_ExternalsTable,  
CmpAppForce_ExternalsTable,  
CmpBinTagUtil_ExternalsTable,  
CmpBlkDrvCom_ExternalsTable,  
CmpChannelMgrEmbedded_ExternalsTable,  
CmpChannelServerEmbedded_ExternalsTable,  
CmpAppEmbedded_ExternalsTable,  
CmpChecksum_ExternalsTable,  
CmpCommunicationLib_ExternalsTable,  
CmpDevice_ExternalsTable,  
CmpIecTask_ExternalsTable,  
CmpIoDrvIec_ExternalsTable,  
CmpIoMgrEmbedded_ExternalsTable,  
CmpMonitor_ExternalsTable,  
CmpNameServiceServer_ExternalsTable,  
CmpRouterEmbedded_ExternalsTable,  
CmpScheduleEmbedded_ExternalsTable,  
CmpSettingsEmbedded_ExternalsTable,  
CmpSrv_ExternalsTable,  
NULL  
};
```

The *ComponentName\_HookFunction* routines and *ComponentName\_ExternalsTable* objects are defined in the components root modules implicitly by the following lines:

```
/* The root header file. Should be included before any other header files */  
#include "CmpStd.h"  
  
/* The ComponentName component dependency header file */  
#include "ComponentNameDep.h"  
  
/* This macro may only be used in the components root modules. */  
/* It instantiates the component's interface object and the export table */  
USE_STMT
```

The *ComponentName\_HookFunction* routine is a wrapper generated by m4 preprocessor in the *ComponentName* component's root module. This wrapper just calls the private *HookFunction* routine located in that root module.

#### 4.4.3.3 Excluding Components Functionality

Some components that are not included to the components list can still remain to be referenced from the components composing the runtime system. This may produce some dead-code in the MicroRTS binary image, and this code sometimes consumes a decent amount of ROM space.

To explicitly exclude the code that refers to interface functions of some unwanted components, an additional header file should be created and included to *sysdefines.h* of specific build configuration. This header file should contain a set of *COMPONENTNAME\_NOTIMPLEMENTED* macros. The example below was taken from the Cortex-M3 MicroRTS reference implementation for NXP LPC1768:

```
#ifndef _CORTEXM3_NOTIMPL_H_  
#define _CORTEXM3_NOTIMPL_H_  
  
#define CMPMONITOR2_NOTIMPLEMENTED  
#define CMPAPPBP_NOTIMPLEMENTED  
#define CMPEVENTMGR_NOTIMPLEMENTED  
#define CMPASYNCMGR_NOTIMPLEMENTED  
#define CMPIODRVPV1C1MASTER_NOTIMPLEMENTED  
#define CMPIODRVPV1C2MASTER_NOTIMPLEMENTED  
#define CMPIODRVPARAMETER2_NOTIMPLEMENTED  
#define CMPIODRVPARAMETER_NOTIMPLEMENTED  
#define CMPIODRVPROFINET_NOTIMPLEMENTED  
#define CMPLOGBACKEND_NOTIMPLEMENTED  
#define CMPSIL2_NOTIMPLEMENTED  
#define CMPTRACEMGR_NOTIMPLEMENTED  
#define CMPMEMGC_NOTIMPLEMENTED  
#define CMPUSERMGR_NOTIMPLEMENTED  
#define SYSOUT_NOTIMPLEMENTED  
#define SYSSEM_NOTIMPLEMENTED  
#define SYSSHM_NOTIMPLEMENTED  
#define SYSTASK_NOTIMPLEMENTED  
#define SYSTIMER_NOTIMPLEMENTED  
#define SYSTIMERTC_NOTIMPLEMENTED
```



```
#endif /* _CORTEXM3_NOTIMPL_H_*/
```

This header file should be included to the sysdefines.h header file for specific build configuration.

## 4.5 Runtime Operation

### 4.5.1 Startup Sequence

This section describes the Cortex-M3 MicroRTS startup sequence that is performed on the HITEX LPC1768-Stick target device.

1. After power-up, the reset interrupt service routine ResetISR located in Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\src\Source\System\startup.c is called. This routine initializes data segments, and control is passed to the binary executable entry point main() that resides in Platforms\Native\CortexM3\NXP-LPC1768\Sys\MainCortexM3.c. Subsequent operation of the runtime system is performed on the main() routine execution context.
2. The main() routine shown below calls to the sysInit routine (located in startup.c) to initialize integrated peripherals of the target device.

```
/* Platforms\Native\CortexM3\NXP-LPC1768\Sys\MainCortexM3.c */
int main()
{
    RTS_RESULT Result;

    /* Peripherals initialization */
    sysInit();

    /* CMinit is actually a redefined name for CMMicroInit */
    /* The second parameter is not used here. */
    Result = CMinit(NULL, s_ComponentList);

    while(!s_bExitLoop)
    {
        /* Cyclical execution of components HookFunction. */
        /* CMCallHook is a redefined name for CMMicroCallHook */

        CMCallHook( CH_COMM_CYCLE, 0, 0, FALSE) ;
    }

    return 0;
}
```

3. The CMMicroInit routine (this name is redefined to CMinit) is being called from the main routine. At this point the MicroRTS Component Manager performs initialization of CODESYS runtime system – the CMMicroInit routine walks through the list of MicroRTS initialization phases and calls to the CMMicroCallHook routine passing the phase identifier as the first parameter as follows:

```
static RTS_UI32 s_ulLastCommCycleHookCall = 0;

/**
 * Hook types table (s_SystemPhases) containing
 * a list of hook types which are used during
 * the system initialization.
 */
CDS_DECLARE_HOOKS_LIST_BEGIN()
CDS_INSERT_HOOK(CH_INIT_SYSTEM)
CDS_INSERT_HOOK(CH_INIT_SYSTEM2)
CDS_INSERT_HOOK(CH_INIT)
CDS_INSERT_HOOK(CH_INIT2)
CDS_INSERT_HOOK(CH_INIT201)
CDS_INSERT_HOOK(CH_INIT3)
CDS_INSERT_HOOK(CH_INIT_TASKS)
CDS_INSERT_HOOK(CH_INIT_COMM)
CDS_DECLARE_HOOKS_LIST_END()
```

```
RTS_RESULT CDECL CMMicroInit(char *pszSettingsFile)
{
    int iPhase;
    RTS_RESULT Result = ERR_OK;

    /* Walk over the list of phases and call to the HookFunction routine */
    /* in each component during the system initialization */

    for (iPhase = 0; iPhase < CDS_HOOKS_LIST_SIZE(); ++iPhase)
    {
        Result = CMMicroCallHook(CDS_HOOK_LIST_ENTRY_ID(iPhase), 0, 0);

        if (CH_INIT_SYSTEM2 == CDS_HOOK_LIST_ENTRY_ID(iPhase))
        {
            /* after CH_INIT_SYSTEM2 */
            s_ulLastCommCycleHookCall = CAL_SysTimeGetMs();
        }
    }
    return Result;
}
```

4. The CMMicroCallHook routine walks through the list of components, checks if the previous call to the currently selected component's HookFunction succeeded, and if it did – calls to this HookFunction. The call result is stored in the component descriptor of each component.
5. When the CMMicroInit routine finishes, the main() routine starts calling the CMMicroCallHook routine cyclically passing the CH\_COMM\_CYCLE phase identifier as a first parameter – MicroRTS is now in the operating mode.

#### 4.5.2 Operating Mode

In the operating mode, the main routine calls the CMMicroCallHook cyclically with the CH\_COMM\_CYCLE phase identifier passed as a first parameter. The CMMicroCallHook walks through the list of component descriptors and checks whether the previous call to CMMicroCallHook succeeded, i.e. the ERR\_OK error code was returned. If the previous call failed or if the component requested the Component Manager to stop calling its HookFunction, the CMMicroCallHook skips this component.

If some components don't require any activity to be organized by cyclic invocation of their HookFunction, they can return the ERR\_NO\_COMM\_CYCLE error code, which is stored in the corresponding component descriptors, and the CMMicroCallHook will no longer call their HookFunction:

```
/* Components\System\System.c */
static RTS_RESULT CDECL HookFunction(RTS_UI32 ulHook, RTS_UINTPTR ulParam1, RTS_UINTPTR ulParam2)
{
    RTS_RESULT Res = ERR_OK;

    switch (ulHook)
    {
        /* ... */

        /* this error code prevents HookFunction from being called cyclically */
        case CH_COMM_CYCLE:
            return ERR_NO_COMM_CYCLE;

        default:
            break;
    }

    if(Res == ERR_OK)
        return SysMemOSHookFunction(ulHook, ulParam1, ulParam2);

    return Res;
}
```

## 5 Implementing the MicroRTS

### 5.1 Overview

This chapter describes the CODESYS V3 runtime system adaptation process specific for the MicroRTS profile. For more information about the CODESYS V3 adaptation, please refer to the CODESYS Control V3 Manual and the CODESYS Control V3 Migration and Adaptation.

The adaptation process consists of the following steps:

1. Creating a device description file – the device description file contains the device and vendor identification information, specific runtime features and CODESYS V3 compiler options, memory layout of the target device, the list of libraries associated with the device and other options.
2. Organizing the source code – the MicroRTS source tree has to be structured as it would be easy to build, download and debug the binary executable. The other things to have in mind while organizing the source tree are to provide the possibility to update sources to the latest version released by 3S and to make room for creating other adaptations in future.
3. Defining the components list – this step is needed to specify the list of components which are planned to be part of MicroRTS on a specific target.
4. Adapting specific system components – typically, the platform-independent components can be left unchanged during adaptation – the only thing that has to be done is to configure some of them by defining a set of configuration macros in the sysdefines.h header file.

Some of the system components should be modified to implement the corresponding interactions between MicroRTS and its environment on a target device: SysTarget, SysCpuHandling, SysTime, SysFlash, SysMem, SysExcept and, depending on the type of block driver chosen for communication between CODESYS IDE and a target device, SysCom or SysSocketEmbedded.

5. Building and debugging the MicroRTS – this step depends on the toolchain chosen for building the MicroRTS binary executable.

### 5.2 Creating Device Description File

#### 5.2.1 Overview

The device description file contains the device and vendor identification information, specific runtime features and CODESYS V3 compiler options, memory layout of a target device, the list of libraries associated with a device and other options used by CODESYS V3 IDE to create, compile and download user applications written in IEC 61131-3 programming languages to a target device.

The device description file is an XML file with the .devdesc.xml extension. For more information on the device description file content and format, please refer to section 6.4 of the CODESYS Control V3 Manual.

This sub-section contains only brief guidelines on creating the device description file based on templates supplied with the Cortex-M3 MicroRTS Starter Package.

There are at least two device description files in the *DeviceDescriptions* folder that resides in the starter package installation folder: *MicroRuntime\_LPC1768.devdesc.xml* and *MicroRuntime\_LM3S9B96.devdesc.xml*. The following explanation is based on *MicroRuntime\_LPC1768.devdesc.xml* content.

There are several configuration nodes and sections denoted by the corresponding xml tags, which should be modified while working on the new device description file that is based on *MicroRuntime\_LPC1768.devdesc.xml*.

Node/Section	Description
DeviceIdentification	Device type, identifier and version
DeviceInfo	Device readable name and description, vendor name.

Node/Section	Description
ExtendedSettings	Device extended settings
TargetSettings	
runtime_features	Runtime features supported by a target device
codegenerator	Instruction set selection and compiler options
memory-layout	Target device memory layout
taskconfiguration	IEC application tasks options

For more information about I/O configuration, see chapter 6 of the CODESYS V3 Control Manual.

## 5.2.2 Specifying Device Identification and Device Information

The following two sections of a device description file should contain device identification parameters and human-readable device and vendor names:

```
<DeviceDescription>
  <Device>
    <DeviceIdentification>
      <Type>4096</Type>
      <Id>0000 8013</Id>
      <Version>3.5.0.0</Version>
    </DeviceIdentification>
    <DeviceInfo>
      <Name name="local:typename">LPC1768 Cortex-M3 SoC</Name>
      <Description name="local:typedescription">A 3S Target for NXP LPC1768</Description>
      <Vendor name="local:3S">3S - Smart Software Solutions GmbH</Vendor>
      <OrderNumber>??</OrderNumber>
    </DeviceInfo>
    ...
  </Device>
</DeviceDescription>
```

The *DeviceIdentification – Type* node value specifies a device type. The value 4096 denotes the CODESYS-programmable device type, i.e. a device that can be used as a target for downloading and executing applications created in CODESYS V3. The value that is set for this node should also be defined in the sysdefines.h header file (or as a compiler option -D or /D) in the runtime system build configuration:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */

#ifndef SYSTARGET_DEVICE_TYPE
# define SYSTARGET_DEVICE_TYPE SYSTARGET_TYPE_PROGRAMMABLE
#endif
```

The *DeviceIdentification – Id* node contains a unique device type identifier provided by 3S. The first word of the *Id* node value (0000 in the example shown above) is a manufacturer (or vendor) identifier assigned by 3S to each OEM-customer. The second word (8013 in the example) is a specific target device identifier. An OEM-customer is typically provided with both parts of *Id* within shipment of the starter or runtime development package. The value that is set for this node should also be defined in the sysdefines.h header file (or as a compiler option -D or /D) in the runtime system build configuration:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */

#ifndef SYSTARGET_VENDOR_ID
# define SYSTARGET_VENDOR_ID RTS_VENDORID_3S
#endif

#ifndef SYSTARGET_DEVICE_ID
# define SYSTARGET_DEVICE_ID 0x8013
#endif
```

The *DeviceIdentification – Version* node contains the version number of the device description file for a specific device. If the *DeviceIdentification – Type* value is greater than or equal to 4096, the version

number should be specified exactly as it is shown above: n.n.n.n. This node value should also be specified in the sysdefines.h header file:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */
#ifndef SYSTARGET_DEVICE_VERSION
#define SYSTARGET_DEVICE_VERSION 0x03050000
#endif
```

The *DeviceInfo – Name* node contains a string value that will appear in the **Device** combo-box of the CODESYS **Standard Project** dialog box, which is displayed on the screen when the new CODESYS project is being created. This string value is also displayed in the CODESYS project device tree.

The *DeviceInfo – Description* node contains a description of the target device.

The *DeviceInfo – Vendor* name value contains a manufacturer name that is displayed along with the device name in the **Device** combo-box of the CODESYS **Standard Project** dialog box.

The device name and the vendor name have to be specified in the sysdefines.h header file:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */
#ifndef SYSTARGET_DEVICE_NAME
#define SYSTARGET_DEVICE_NAME "LPC1768 Cortex-M3 SoC"
#endif

#ifndef SYSTARGET_VENDOR_NAME
#define SYSTARGET_VENDOR_NAME "3S-Smart Software Solutions GmbH"
#endif
```

The *DeviceInfo – OrderNumber* node may contain a manufacturer specific order number for the device described with this device configuration file.

### 5.2.3 Specifying Runtime Features

The set of features supported by CODESYS V3 IDE while communicating with the runtime system on a target device is specified in the *Device – ExtendedSettings – TargetSettings – runtime\_features* section.

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

There are several options in this section that have to be carefully set for devices which are planned to be used with the MicroRTS profile:

Runtime Feature	Value	Comment
only_explicit_features_supported	1	This option limits the CODESYS functionality as the only set of features (menu commands) explicitly specified below are enabled.
compact_download	1	Enables the compact binary format generated by CODESYS for downloading to the target.
max_number_of_apps	1	Limits the number of applications to 1
fixed_app_name	Application	Specifies the application name.
breakpoints_supported	0	Disables remote debugging.

Runtime Feature	Value	Comment
cycle_control_in_iec	1	The task function block is always called, regardless of the application status.
cycle_control_version_2	1	The call to <code>__sys_rts_cycle_2()</code> is generated in the application code instead of <code>__sys_rts_cycle()</code> .
boot_application_supported	1	The bootable application is always created.
write_variables_supported	1	Enables support for writing variables when CODESYS IDE is connected to the target device.
connect_device_supported	1	Enables connection between CODESYS IDE and the target device.
core_application_handling_supported	1	Login, Logout, Start, Stop, Single Cycle and Reset command are enabled in CODESYS IDE during remote connection with the target device.
force_variables_supported	1	Enables support for forcing values for variables when CODESYS IDE is connected to the target device.

For more information about the runtime features configuration, please refer to section 6.4.5.1.1 of the CODESYS Control V3 Manual.

The `max_number_of_apps` runtime option should also be specified in the `sysdefines.h` header file of the MicroRTS build configuration:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */
#define APPL_NUM_OF_STATIC_APPLS 1
```

The fixed name of a single IEC application should be set as a runtime parameter of the IEC Application Manager component and the SysFileFlash component:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */
#define SETTG_ENTRIES_STRING \
{"CmpRouter", "0.MainNet", "MyCom"}, \
{"CmpBlkDrvCom", "Com.0.Name", "MyCom"}, \
{"CmpAppEmbedded", "Application.1", "application"}, \
{0, 0, 0}

#define FILE1_SIZE 0x10000

#define FILE_MAP FILE_DESC m_FileSystem[] = \
{
/* Name      Offset  MaxSize  read index write index */ \
{"Application.app", 0x0, FILE1_SIZE, 0xFFFFFFFF, 0xFFFFFFFF}, \
};
```

## 5.2.4 Configuring Codegenerator

The *Device – ExtendedSettings – TargetSettings – codegenerator* section is used for selecting and configuring the CODESYS V3 IDE IEC 61131-3 compiler backend.

```
<DeviceDescription>
  <Device>
    <ExtendedSettings>
      <ts:TargetSettings xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-0.1.xsd">
        <ts:section name="codegenerator">
          <ts:setting name="CPU" type="codegenerators" access="edit">
            <ts:value>ARM Cortex-M3</ts:value>
          </ts:setting>
          other code generator options...
        </ts:section>
        other sections...
      </ts:TargetSettings>
    </ExtendedSettings>
  </Device>
</DeviceDescription>
```

Some of the options in this section have to be carefully set for devices which are planned to be running MicroRTS:

Compiler Option	Value	Comment
Floating Point Unit	0	Tells the code generator that the target CPU doesn't have a hardware unit implementing floating point operations.
lreal-data-type	0	Disables support for the LREAL data type. It is important to note that the code size will be increased dramatically, if this option is set to 1. In this case, the <code>SYSINTERNAL_DISABLE_64BIT</code> macro has to be removed from the <code>sysdefines.h</code> header file of the MicroRTS build configuration leading to increase of the runtime system code size by almost 30%.
lreal-as-real	1	All occurrences of type LREAL in IEC applications will be implicitly converted to a REAL.

There are two code generator-related macros that should be defined in the `sysdefines.h` header file of the MicroRTS build configuration:

```
#define SYSINTERNAL_DISABLE_64BIT
#define SYSINTERNAL_DISABLE_MATH
```

For more information about the rest code generator options, please refer to section 6.4.5.1.6 of the CODESYS Control V3 Manual.

## 5.2.5 Creating Memory Layout

### 5.2.5.1 Overview

The *Device – ExtendedSettings – TargetSettings – memory-layout* and *areas* sections are used for defining and configuring memory areas that will be used by the CODESYS V3 IDE compiler for placing an IEC application code and for locating global, input, output, retain and other kinds of variables in memory of a target device.

```
<DeviceDescription>
  <Device>
    <ExtendedSettings>
      <ts:TargetSettings xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-0.1.xsd">
        <ts:section name="memory-layout">
          <ts:setting name="code-segment-header-size" type="integer" access="visible">
            <ts:value>104</ts:value>
          </ts:setting>
          other memory layout options...
        </ts:section>
        <ts:section name="areas">
          <ts:setting name="number" type="integer" access="visible">
            <ts:value>4</ts:value>
          </ts:setting>
        </ts:section>
        other sections...
      </ts:TargetSettings>
    </ExtendedSettings>
  </Device>
</DeviceDescription>
```

The detailed description of options available in this section can be found in section 6.4.5.1.2 of the CODESYS Control V3 Manual.

This section contains the information essential for the MicroRTS profile.

Please note that some of the options specified in the *memory-layout* section should be defined accordingly in the `sysdefines.h` header file of the MicroRTS build configuration.

There is a set of common parameters that should be specified in the memory-layout section:

Memory Layout Option	Value	Comment
code-segment-header-size	104	Code segment header size in the compact download format.
memory-size	Target specific	Memory segment size for placing %M-referenced variables of an IEC application.
input-size	Target specific	Input segment size for placing %I-referenced variables.
output-size	Target specific	Output segment size for placing %Q-referenced variables.
retain-size	Target specific	Data segment size for placing variables declared as RETAIN, i.e. the variables whose values should be kept unchanged in case of power down. Please note that this option value should be at least 24 bytes as less as the actual size of non-volatile memory planned to be used for storing RETAIN variables.
retain-in-own-segment	1	A separate memory segment should be used for placing RETAIN variables.
constants-in-own-segment	1	Constants of user-defined data types, strings and arrays will be placed to a separate segment.
pack-mode	CPU specific	Structure members alignment (in bytes). The value 8 is used for ARM-based targets.
stack-alignment	CPU specific	Stack-alignment (in bytes). The value 8 is used for ARM-based targets.
allocation-plus-in-percent	0	Disables any re-allocations of segments memory in the target.
code-segment-prolog-size	CPU-specific	Number of bytes at the beginning of each code area, which will be used by a compiler for its internal purposes. The value 12 is used for ARM-based targets.

There is also a special parameter for specifying the total number of different memory areas available on the target:

```
<DeviceDescription>
  <Device>
    <ExtendedSettings>
      <ts:TargetSettings xmlns:ts="http://www.3s-software.com/schemas/TargetSettings-0.1.xsd">
        <ts:section name="memory-layout">
          memory layout options...
        </ts:section>
        <ts:section name="areas">
          <ts:setting name="number" type="integer" access="visible">
            <ts:value>4</ts:value>
          </ts:setting>
          ... other memory areas options
        </ts:section>
        other sections...
      </ts:TargetSettings>
    </ExtendedSettings>
  </Device>
</DeviceDescription>
```

The same memory area can be used for various purposes, e.g. for storing an IEC application code and constants or for placing input, output and internal variables. So, there are two options that have to be specified while describing each memory area: *flags* and *area-flags*:

Depending on the budget of available flash memory and SRAM, multiple memory areas can be specified.

For example, the NXP LPC1768-Stick target contains 4 memory areas:

1. Fixed-size area of size 0x10000 (64 kB) with the absolute start address 0x30000 for storing an IEC application code and constants.



2. Fixed-size data area of size 0x4000 (16 Kb) with the absolute start address 0x2007C000 for placing input, output, memory and internal variables.
3. Fixed-size data area of size 0x2000 (8 Kb) with the absolute start address 0x20080000 for placing input, output, memory and internal variables.
4. Fixed-size data area of size 0x1000 (4 Kb) with the absolute start address 0x20082000 for placing retain and persistent variables.

The content of memory areas description used for the NXP LPC1768-Stick target is as follows:

```
<ts:section name="areas">

  Total number of memory areas
  <ts:setting name="number" type="integer" access="visible">
    <ts:value>4</ts:value>
  </ts:setting>

  Code area description
  <ts:section name="area_0">

    (0x40 | 0x2 ) - can contain code and constants
    <ts:setting name="flags" type="integer" access="visible">
      <ts:value>0x42</ts:value>
    </ts:setting>

    0x10 - fixed size area
    <ts:setting name="area_flags" type="integer" access="visible">
      <ts:value>0x10</ts:value>
    </ts:setting>

    Absolute start address of the area
    <ts:setting name="start-address" type="integer" access="visible">
      <ts:value>0x30000</ts:value>
    </ts:setting>

    Area size mininum
    <ts:setting name="minimal-area-size" type="integer" access="visible">
      <ts:value>0x10000</ts:value>
    </ts:setting>

    Area size maximum
    <ts:setting name="maximal-area-size" type="integer" access="visible">
      <ts:value>0x10000</ts:value>
    </ts:setting>

    Area can not be reallocated to grow in size
    <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
      <ts:value>0</ts:value>
    </ts:setting>

  </ts:section>

  1st data area description
  <ts:section name="area_1">

    Input, output, memory and internal variables
    <ts:setting name="flags" type="integer" access="visible">
      <ts:value>0xFE9D</ts:value>
    </ts:setting>

    0x10 - fixed size area
    <ts:setting name="area_flags" type="integer" access="visible">
      <ts:value>0x10</ts:value>
    </ts:setting>

    Area absolute start address is 0x2007C000 (LPC1768 1st SRAM bank)
    <ts:setting name="start-address" type="integer" access="visible">
      <ts:value>0x2007C000</ts:value>
    </ts:setting>

    Area size mininum
    <ts:setting name="minimal-area-size" type="integer" access="visible">
      <ts:value>0x4000</ts:value>
    </ts:setting>

```

```
</ts:setting>

Area size maximum
<ts:setting name="maximal-area-size" type="integer" access="visible">
  <ts:value>0x4000</ts:value>
</ts:setting>

Area cannot be reallocated to grow in size
<ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
  <ts:value>0</ts:value>
</ts:setting>
</ts:section>

2nd data area, start address 0x20080000 (LPC1768 2nd SRAM bank), size 0x2000
<ts:section name="area_2">
  <ts:setting name="flags" type="integer" access="visible">
    <ts:value>0xFE9D</ts:value>
  </ts:setting>
  <ts:setting name="area_flags" type="integer" access="visible">
    <ts:value>0x10</ts:value>
  </ts:setting>
  <ts:setting name="start-address" type="integer" access="visible">
    <ts:value>0x20080000</ts:value>
  </ts:setting>
  <ts:setting name="minimal-area-size" type="integer" access="visible">
    <ts:value>0x2000</ts:value>
  </ts:setting>
  <ts:setting name="maximal-area-size" type="integer" access="visible">
    <ts:value>0x2000</ts:value>
  </ts:setting>
  <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
    <ts:value>0</ts:value>
  </ts:setting>
</ts:section>

Retain data area description
<ts:section name="area_3">

  (0x100 | 0x20) - persistent and retain data
  <ts:setting name="flags" type="integer" access="visible">
    <ts:value>0x120</ts:value>
  </ts:setting>

  Fixed-size area
  <ts:setting name="area_flags" type="integer" access="visible">
    <ts:value>0x10</ts:value>
  </ts:setting>

  Start address 0x20082000 (LPC1768 2nd SRAM bank)
  <ts:setting name="start-address" type="integer" access="visible">
    <ts:value>0x20082000</ts:value>
  </ts:setting>

  Area size minimum
  <ts:setting name="minimal-area-size" type="integer" access="visible">
    <ts:value>0x1000</ts:value>
  </ts:setting>

  Area size maximum
  <ts:setting name="maximal-area-size" type="integer" access="visible">
    <ts:value>0x1000</ts:value>
  </ts:setting>

  Area cannot be reallocated to grow in size
  <ts:setting name="allocation-plus-in-percent" type="integer" access="visible">
    <ts:value>0</ts:value>
  </ts:setting>
</ts:section>
</ts:section>
```

### 5.2.5.2 Defining Code Area(s)

In the MicroRTS profile, the code area is typically an area in the non-volatile (flash) memory where an IEC application code is placed and executed.

The MicroRTS profile uses a so-called compact download format of an IEC application binary (see section 2.3.1.1 of the CODESYS Control V3 Manual). Because of this and in order to properly handle the code area, the memory-layout section should contain the *code-segment-header-size* option with the value set to 104 as shown above in the snippet of the device description file content.

The *code-segment-prolog-size* option should also be specified in some cases letting the CODESYS compiler to use a sub-region in the code area for internal purposes. For example, in the Cortex-M3 MicroRTS reference implementation, this option is set to 12.

While defining a code memory area for a flash-based target device, use the following options in the corresponding area description:

Memory Area Option	Value	Comment
flags	0x	Area can contain code and constants.
area-flags	0x10	Area of fixed size should be located at some dedicated absolute start address.
start-address	Target specific	Specify the start address of a memory region reserved on a target device for an IEC application code.
minimal-area-size	Target specific	Specify the minimal area size. For flash media, set this value equal to maximal-area-size letting CODESYS to generate the code segment of constant size regardless of the actual size of an IEC application code.
maximal-area-size	Target specific	Specify the size of a memory region reserved on a target device for an IEC application code.
allocation-plus-in-percent	0	Segment re-allocation is not allowed.

The MicroRTS binary executable build configuration should be supplied with information for reserving the code area on the target device. For embedded devices similar to LPC1768 and GNU C-based toolchains, there are two places for doing that:

1. Linker configuration file: in the Cortex-M3 MicroRTS reference implementation this file (CODESYS.ld) contains the following options which define the code area:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\CODESYS.ld */
MEMORY
{
    FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x0080000
    ... other memory options
}

SECTIONS
{
    /* fixup for the code area start address */
    PROVIDE(__FLASH_START = 0x30000);
    /* fixup for the code area size */
    PROVIDE(__FLASH_SIZE = 0x10000);

    ... other memory section description information
}
```

2. sysspecific.h: this header file contains redefinitions for accessing corresponding fixup entries specified in the linker configuration file:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysspecific.h */
#ifdef __GNUC__ /* GNU Toolchain from e.g. CodeSourcery */
/* Values, defined in the linker script */
extern int __FLASH_START;
extern int __FLASH_SIZE;
#define FLASH_START (&__FLASH_START)
#define FLASH_SIZE (&__FLASH_SIZE)
#endif
```

### 5.2.5.3 Defining Data Area(s)

The MicroRTS profile can contain any number of data areas. The most important thing to have in mind is that these areas should be placed to some reserved SRAM regions.

While defining data memory areas for a MicroRTS target device, use the following options in the corresponding area description:

Memory Area Option	Value	Comment
flags	0xFE9D	Area can contain input, output, memory and internal variables.
area-flags	0x10	Area of fixed size should be located at some dedicated absolute start address.
start-address	Target specific	Specify the start address of a memory region reserved on a target device for an IEC application data.
minimal-area-size	Target specific	Specify the minimal area size. For MicroRTS, set this value equal to maximal-area-size letting CODESYS to generate the data area of constant size regardless of the actual size used by an IEC application.
maximal-area-size	Target specific	Specify the size of a memory region reserved on a target device for an IEC application data placed to this data area.
allocation-plus-in-percent	0	Segment re-allocation is not allowed.

The MicroRTS binary executable build configuration should be supplied with information for reserving data areas on the target device. For embedded devices similar to LPC1768 and GNU C-based toolchains, there are two places for doing that:

1. Linker configuration file: in Cortex-M3 MicroRTS reference implementation this file (CODESYS.Id) contains the following options which define the code area:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\CODESYS.Id */
MEMORY
{
  E_SRAM1 (rwx) : ORIGIN = 0x2007C000, LENGTH = 0x7000
  ... other memory options
}
SECTIONS
{
  /* fixup for the 1st data area start address */
  PROVIDE(__DATA_AREA1_START = 0x2007C000);
  /* fixup for the 1st data area size */
  PROVIDE(__DATA_AREA1_SIZE = 0x4000);

  /* fixup for the 2nd data area start address */
  PROVIDE(__DATA_AREA2_START = 0x20080000);
  /* fixup for the 2nd data area size */
  PROVIDE(__DATA_AREA2_SIZE = 0x2000);

  /* fixup for the retain data area start address */
  PROVIDE(__RETAIN_START = 0x20082000);
  /* fixup for the retain data area size */
  PROVIDE(__RETAIN_SIZE = 0x1000);

  ... other memory section description information
}

```

2. syspecific.h: this header file contains redefinitions for accessing the corresponding fixup entries specified in the linker configuration file:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\syspecific.h */
#ifdef __GNUC__ /* GNU Toolchain from e.g. CodeSourcery */
  /* Values, defined in the linker script */
  extern int __DATA_AREA1_START;
  extern int __DATA_AREA1_SIZE;
  #define DATA_AREA1_START (&__DATA_AREA1_START)
  #define DATA_AREA1_SIZE (&__DATA_AREA1_SIZE)

```

```

extern int __DATA_AREA2_START;
extern int __DATA_AREA2_SIZE;
#define DATA_AREA2_START (&__DATA_AREA2_START)
#define DATA_AREA2_SIZE (&__DATA_AREA2_SIZE)

extern int __RETAIN_START;
extern int __RETAIN_SIZE;
#define RETAIN_START (&__RETAIN_START)
#define RETAIN_SIZE (&__RETAIN_SIZE)
#endif

```

These macros defined for data areas in the `syspecific.h` header file are used by the SysMem component `SysMemAllocArea` routine. For more information, please refer to section 5.5.4.4.

## 5.2.6 Setting-up Tasks

The MicroRTS profile executes an IEC application code in a single tasking environment. The `CmpScheduleEmbedded` component is used to emulate a co-operative multitasking in IEC applications.

The *Device – ExtendedSettings – TargetSettings – taskconfiguration* section is used for defining and configuring IEC tasks that will be used by the CODESYS V3 IDE.

While defining IEC tasks configuration for a MicroRTS target device, use the following options:

Tasks Option	Recommended Value	Comment
supportmicroseconds	0	IEC task cycle interval cannot be set with microseconds granularity. Leave this option unchanged, if the <code>SysTimeGetUs</code> function of the <code>SysTime</code> component is not implemented.
supportfreewheeling	0	IEC free-running (free-wheeling) tasks are not supported in a single tasking environment. It is recommended to support only interval and event tasks in the MicroRTS profile.
supportinterval	1	IEC interval tasks are supported. The interval tasks are scheduled to run cyclically with specified cycle periods.
supportevent	0	IEC application event driven tasks are not supported. An IEC application event is an event of changing the value of some variable of type <code>BOOL</code> from <code>FALSE</code> to <code>TRUE</code> . This option can be enabled.
supportextendedwatchdog	1	Watchdog monitoring of IEC tasks is turned on to prevent them from stalling.
supportexternal	0	External event driven tasks are not supported.
defaulttaskpriority	1	Priority value assigned by CODESYS to a newly created task.
mintaskpriority	0	Minimal priority value.
maxtaskpriority	1	Maximal priority value.
maxnumoftasks	2	Available number of IEC tasks (sum of <code>maxeventtasks</code> , <code>maxintervaltasks</code> and <code>maxfreetasks</code> ).
maxeventtasks	0	Available number of event-driven IEC tasks.
maxintervaltasks	2	Available number of interval tasks.
maxexternalevents	0	External event driven tasks are not supported.
maxfreetasks	0	Available number of free-wheeling tasks.
cycletimedefault	t#20ms	Interval value assigned by CODESYS to a newly created task.

The MicroRTS binary executable build configuration should be supplied with tasks-related information according to the options specified in the `taskconfiguration` section:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */
#define NUM_OF_STATIC_IEC_TASKS 2
#define MAX_IEC_TASKS NUM_OF_STATIC_IEC_TASKS

```

Please note that the NUM\_OF\_STATIC\_IEC\_TASKS macro value should be set to the total number of IEC tasks of different types which is specified by the *maxnumoftasks* option in the *Device – ExtendedSettings – TargetSettings – taskconfiguration* section. If this section allows to add more IEC tasks to the CODESYS project than specified by NUM\_OF\_STATIC\_IEC\_TASKS, MicroRTS will stop executing an IEC application immediately with the corresponding exception.

## 5.3 Organizing the MicroRTS Source Tree

### 5.3.1 Source Tree Layout

The layout of the MicroRTS reference implementation source tree is described in section 3.4. The source tree includes the Components folder containing platform-independent source code of functional and system components, and two target specific folders: Platforms\Native\CortexM3\NXP-LPC1768 and Platforms\Native\CortexM3\TI-LM3S9B96\_uRTS, which contain the platform-dependent implementation of system components for two chips based on the ARM Cortex-M3 architecture.

This folders structure can be used as it is while working on a specific adaptation of MicroRTS, but if it is necessary to develop several types of PLCs based on the same chip, the platform-dependent folders structure could be different as shown below. It is recommended to keep the *Components* folder structure unchanged, although there is a possibility to convert it to a flat files list.

Folder/File	Description
Platforms	Root folder for platform-dependent sources
Native	Folder for targets without OS
CortexM3	Folder for ARM Cortex-M3-based targets
MainCortexM3.c	C module containing the main() entry point shared among all targets
syscortexm3cfg.h	Set of configuration macros shared among all target devices. This header file should be included to all PLC-specific sysdefines.h files listed below.
Sys	Folder for the system components implementation shared among all target devices
SysMemCortexM3.c	Shared platform-dependent implementation of SysMem
SysCpuHandlingCortexM3.c	Shared platform-dependent implementation of SysCpuHandling
SysTimeCortexM3.c	Shared platform-dependent implementation of SysTime
SysExceptCortexM3.c	Shared platform-dependent implementation of SysExcept
SysTargetCortexM3.c	Shared platform-dependent implementation of SysTarget
SysIntCortexM3.c	Shared platform-dependent implementation of SysInt
SysComCortexM3.c	Shared platform-dependent implementation of SysCom
SysSocketEmbeddedCortexM3.c	Shared platform-dependent implementation of SysSocketEmbedded
<ChipNameFolder>	Folder for the <i>ChipName</i> chip that is based on Cortex-M3
Sys	Folder for the system components implementation shared among all the <i>ChipName</i> -based target devices
SysFlash<ChipName>.c	SysFlash implementation for the <i>ChipName</i> chip
SysCom<ChipName>.c	SysCom implementation for the <i>ChipName</i> chip
SysSocket<ChipName>.c	SysSocket implementation for the <i>ChipName</i> chip
CmpSettings<ChipName>.c	CmpSettings implementation for the <i>ChipName</i> chip
<PLCType1>	Folder for the <i>PLCType1</i> target based on the <i>ChipName</i> chip.
sysdefines.h	Set of configuration macros specific for the <i>PLCType1</i> target build

Folder/File	Description
	configuration. Should include syscortexm3cfg.h.
syspecific.h	Set of memory-layout macros specific for the <i>PLCType1</i> target build configuration
targetdefines.h	Set of additional configuration macros specific for the <i>PLCType1</i> target build configuration
<ComponentList>.h	Components list header file for the <i>PLCType1</i> target
<NotIncluded>.h	Components exclusion list header file for the <i>PLCType1</i> target
Projects	MicroRTS build projects for the <i>PLCType1</i> target for different toolchains
<Toolchain1Folder>	<i>Toolchain1</i> project folder
<Toolchain2Folder>	<i>Toolchain2</i> project folder.
<PLCType2>	Folder for the <i>PLCType2</i> target based on the <i>ChipName</i> chip.
sysdefines.h	Set of configuration macros specific for the <i>PLCType2</i> target build configuration. Should include syscortexm3cfg.h.
syspecific.h	Set of memory-layout macros specific for the <i>PLCType2</i> target build configuration
targetdefines.h	Set of additional configuration macros specific for the <i>PLCType2</i> target build configuration
<ComponentList>.h	Components list header file for the <i>PLCType2</i> target
<NotIncluded>.h	Components exclusion list header file for the <i>PLCType2</i> target
Projects	MicroRTS build projects for the <i>PLCType2</i> target for different toolchains
<Toolchain1Folder>	<i>Toolchain1</i> project folder
<Toolchain2Folder>	<i>Toolchain2</i> project folder.

The following considerations are kept in mind while making such a structure of platform-dependent sources:

1. There can be several PLC types (*PLCType1*, *PLCType2*, ...) in development that are based on the same *ChipName* chip which, in turn, is based on the ARM Cortex-M3 core. Thus, there can be a shared folder (Platform\Native\CortexM3) containing the Cortex-M3 specific implementation of some core system components in the Sys sub-folder: SysMem, SysCpuHandling, SysTime, SysExcept, SysTarget, SysInt, SysCom (for the 16550-compatible serial port implementation) and SysSocketsEmbedded (for the same implementation of sockets). This folder could also contain the module containing the main() entry point and syscortexm3cfg.h containing a set of shared macros specific for the MicroRTS profile of the runtime system.
2. The Platform\Native\CortexM3 folder can contain the <ChipNameFolder> sub-folder for the same *ChipName* chip used for developing *PLCType1*, *PLCType2*, etc. The <ChipNameFolder> sub-folder can contain the Sys sub-folder containing the *ChipName* specific implementation of system components for chip specific peripherals including the flash memory unit, chip-specific serial ports and chip-specific Ethernet adapters. There can also be a shared implementation of the CmpSettings component here.
3. The Platform\Native\CortexM3\<ChipNameFolder> folder can contain sub-folders for each PLC type: *PLCType1*, *PLCType2*, etc. containing a set of PLC type specific build configuration header files and component list header files.
4. Each Platform\Native\CortexM3\<ChipNameFolder>\<PLCTypeN> folder can also contain the *Projects* sub-folder. This sub-folder can be used for creating separate sub-folders for the MicroRTS build and make projects for different toolchains or IDEs.

This structure allows to reuse the source code developed for one PLC project among several PLC projects.

### 5.3.2 Mandatory Configuration Macros for the MicroRTS Profile

The following set of macros should be defined in either sysdefines.h or syscortexm3cfg.h for the MicroRTS profile of the runtime system:

The components list header file definition macro:

```
#define RTS_CONFIG_FILE <specify relative path to your components list header file>
```

The static linkage model activation macro:

```
#define STATIC_LINK
```

The compact runtime system profile activation macro:

```
#define RTS_COMPACT
```

The MicroRTS profile activation macro:

```
#define RTS_COMPACT_MICRO
```

The 64-bit data types deactivation macro:

```
#define SYSINTERNAL_DISABLE_64BIT
```

The FPU complex operation external library deactivation macro:

```
#define SYSINTERNAL_DISABLE_MATH
```

“Division by zero” exceptions handling macros:

```
#define SYSINTERNALLIB_DISABLE_INT32_DIVBYZERO_CHECK  
#define SYSINTERNALLIB_DISABLE_REAL64_DIVBYZERO_CHECK
```

The snprintf, sprint re-definition macro to replace the standard C-library implementations:

```
#define PREFER_PORTABLE_SNPRINTF
```

The set of macros to disable most of the core components export tables for external libraries:

```
#define CM_DISABLE_EXTREF  
#define CMPAPPEMBEDDED_DISABLE_EXTREF  
#define CMPHEAPPOOL_DISABLE_EXTREF  
#define CMPMEMPOOL_DISABLE_EXTREF  
#define CMPBINTAGUTIL_DISABLE_EXTREF  
#define CMPBLKDRVCOM_DISABLE_EXTREF  
#define CMPCHANNELMGREMBEDDED_DISABLE_EXTREF  
#define CMPCHANNELSERVEREMBEDDED_DISABLE_EXTREF  
#define CMPCHECKSUM_DISABLE_EXTREF  
#define CMPCOMMUNICATIONLIB_DISABLE_EXTREF  
#define CMPDEVICE_DISABLE_EXTREF  
#define CMPMONITOR_DISABLE_EXTREF  
#define CMPNAMESERVICESERVER_DISABLE_EXTREF  
#define CMPSRV_DISABLE_EXTREF  
#define CMPCHECKSUM_DISABLE_EXTREF  
#define CMPEVENTMGR_DISABLE_EXTREF  
#define CMPLOGEMBEDDED_DISABLE_EXTREF  
#define CMPROUTEREMBEDDED_DISABLE_EXTREF  
#define CMPCHEDULEEMBEDDED_DISABLE_EXTREF  
#define CMPSETTINGSEMBEDDED_DISABLE_EXTREF  
#define SYSCOM_DISABLE_EXTREF  
#define SYSINT_DISABLE_EXTREF  
#define SYSEXCEPT_DISABLE_EXTREF  
#define SYSFILEFLASH_DISABLE_EXTREF  
#define SYSSOCKETEMBEDDED_DISABLE_EXTREF  
#define SYSFLASH_DISABLE_EXTREF  
#define SYSTARGET_DISABLE_EXTREF
```

If some of these macros are not defined, memory consumption could significantly increase.



## 5.4 Defining Components

### 5.4.1 Components Source Code Modifications

The MicroRTS profile can only contain components which are prepared to be part of this profile. As stated previously in section 4.3, the full-fledged Component Manager used in the full and compact profiles of the runtime system was replaced for the light-weight equivalent providing only a minimal required set of functionality. Therefore, in order to use the same source tree for all profiles, the following changes have to be made in each component, if it has not been already done:

1. The ComponentEntry, ExportFunctions, ImportFunctions, CmpGetVersion component interface functions should be excluded from the component's root module using the RTS\_COMPACT\_MICRO marco:

```
/* Components\CmpAppEmbedded\CmpAppEmbedded.c fragment */
#ifndef RTS_COMPACT_MICRO
DLL_DECL int CDECL ComponentEntry(INIT_STRUCT *pInitStruct)
{
    /* ... */
    return ERR_OK;
}
static int CDECL ExportFunctions(void)
{
#ifndef CMPAPPEMBEDDED_DISABLE_EXTREF
    EXPORT_STMT;
#endif
    return ERR_OK;
}
static int CDECL ImportFunctions(void)
{
    /* Macro to import functions */
    IMPORT_STMT;
    return ERR_OK;
}
static RTS_UI32 CDECL CmpGetVersion(void)
{
    return CMP_VERSION;
}
#endif
```

2. If some part of component initialization is performed inside the ComponentEntry function, the ComponentEntry function should be kept available in that component root module and conditionally called from the component's HookFunction on some early phase of initialization:

```
/* Components\CmpIecTask\CmpIecTask.c fragment */
static RTS_RESULT CDECL HookFunction(RTS_UI32 ulHook,
                                     RTS_UINTPTR ulParam1,
                                     RTS_UINTPTR ulParam2)
{
    switch (ulHook)
    {
        case CH_INIT:
        {
            #ifdef RTS_COMPACT_MICRO
                ComponentEntry(NULL);
            #endif
            InitTasks();
            break;
        }
        /* ... other phases handling */
        default:
            break;
    }

    return ERR_OK;
}
```

In this case, the ComponentEntry function should also be modified to exclude usage of its single parameter:

```

/* Components\CmpIecTask\CmpIecTask.c fragment */
DLL_DECL int CDECL ComponentEntry(INIT_STRUCT *pInitStruct)
{
    /* If RTS_COMPACT_MICRO is defined, the pInitStruct parameter is always NULL */
    #ifndef RTS_COMPACT_MICRO
        pInitStruct->CmpId = COMPONENT_ID;
        pInitStruct->pfExportFunctions = ExportFunctions;
        pInitStruct->pfImportFunctions = ImportFunctions;
        pInitStruct->pfGetVersion = CmpGetVersion;
        pInitStruct->pfHookFunction = HookFunction;
        pInitStruct->pfCreateInstance = CreateInstance;
        pInitStruct->pfDeleteInstance = DeleteInstance;

        s_pFCMRegisterAPI = pInitStruct->pfFCMRegisterAPI;
        s_pFCMRegisterAPI2 = pInitStruct->pfFCMRegisterAPI2;
        s_pFCMGetAPI = pInitStruct->pfFCMGetAPI;
        s_pFCMGetAPI2 = pInitStruct->pfFCMGetAPI2;
        s_pFCMCallHook = pInitStruct->pfFCMCallHook;
        s_pFCMRegisterClass = pInitStruct->pfFCMRegisterClass;
        s_pFCMCreateInstance = pInitStruct->pfFCMCreateInstance;
    #endif
        s_iTasks = 0;
        s_hIecTaskPool = RTS_INVALID_HANDLE;
        s_hIecApplicationPool = RTS_INVALID_HANDLE;
        s_semExclusiveSection = RTS_INVALID_HANDLE;
    #ifndef RTS_SIL2
        s_bSynchronize = 0;
        s_bSynchronizeApplication = 0;
    #endif
        s_nWaitForStopTimeoutMs = 0;
        memset(s_szVisuTask, 0, sizeof(s_szVisuTask));

    /* ... */

    return ERR_OK;
}

```

The same change has to be made in the secondary modules of system components:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Sys\SysMemCortexM3.c fragment */
RTS_RESULT CDECL SysMemOSInit(INIT_STRUCT *pInit)
{
    #ifndef RTS_COMPACT_MICRO
        s_pFCMRegisterAPI = pInit->pfFCMRegisterAPI;
        s_pFCMRegisterAPI2 = pInit->pfFCMRegisterAPI2;
        s_pFCMGetAPI = pInit->pfFCMGetAPI;
        s_pFCMGetAPI2 = pInit->pfFCMGetAPI2;
        s_pFCMCallHook = pInit->pfFCMCallHook;
        s_pFCMRegisterClass = pInit->pfFCMRegisterClass;
        s_pFCMCreateInstance = pInit->pfFCMCreateInstance;
    #endif

    return ERR_OK;
}

```

3. The Events Manager and the User Manager components are not supported in the MicroRTS profile, so it is strongly recommended to exclude all the code referring to or related with these components:

```

/* Components\CmpLogEmbedded\CmpLogEmbedded.c fragment */
#ifndef CMPEVENTMGR_NOTIMPLEMENTED
static RTS_HANDLE s_hEventLogAdd = RTS_INVALID_HANDLE;
#endif
static RTS_RESULT CDECL HookFunction( RTS_UI32 ulHook,
                                     RTS_UINTPTR ulParam1,
                                     RTS_UINTPTR ulParam2)
{
    switch (ulHook)
    {
        case CH_INIT_SYSTEM:
        {
            #ifdef RTS_COMPACT_MICRO
                ComponentEntry(NULL);
            #endif
        }
    }
}

```

```

    break;
}
case CH_INIT:
{
    break;
}
case CH_INIT2:
{
#ifndef CMPEVENTMGR_NOTIMPLEMENTED
    if (CHK_EventCreate)
        s_hEventLogAdd = CAL_EventCreate(EVT_LogAdd, COMPONENT_ID, NULL);
#endif
    CAL_LogInitServer();
    break;
}
case CH_INIT3:
{
#ifndef CMPUSERMGR_NOTIMPLEMENTED
    if (CHK_UserMgrObjectAdd)
    {
        RTS_RESULT Result = ERR_OK;
        RTS_HANDLE hObject;
        hObject = CAL_UserMgrObjectAdd(USERDB_OBJECT_LOGGER, &Result);
        if (Result != ERR_OK && hObject == RTS_INVALID_HANDLE)
            return ERR_FAILED;
    }
#endif
    break;
}

/* ... */

default:
    break;
}
return 0;
}

```

4. The light-weight Component Manager used in the MicroRTS profile doesn't perform any calls to the components HookFunction routines for shutdown phases, so it is recommended to exclude the corresponding code from the components source code:

```

/* Components\CmpLogEmbedded\CmpLogEmbedded.c fragment */
static RTS_RESULT CDECL HookFunction( RTS_UI32 ulHook,
                                     RTS_UINTPTR ulParam1,
                                     RTS_UINTPTR ulParam2)
{
    switch (ulHook)
    {
        /* other life-time phases handling ..*/
#ifndef RTS_COMPACT_MICRO
        case CH_EXIT_COMM:
            break;
        case CH_EXIT_TASKS:
            break;
        case CH_EXIT3:
        {
            if (CHK_UserMgrObjectOpen && CHK_UserMgrObjectRemove)
            {
                /* ... */
            }
            break;
        }
        case CH_EXIT2:
        {
            CAL_LogExitServer();
            /* ... */
            break;
        }
        case CH_EXIT:
            break;
        case CH_EXIT_SYSTEM:
        {
            /* ... */

```

```
    break;
  }
#endif
  default:
    break;
}
return 0;
}
```

## 5.4.2 Adding Components to the MicroRTS Build Configuration

The following steps are required to add some component to the MicroRTS build configuration:

1. Create the components list header file for the target build. The components list header file should contain the following entries:

```
/* Components list header file */
CDS3_DECLARE_COMPONENTS_LIST_BEGIN(ListName)
  CDS3_ADD_COMPONENT_ENTRY(ComponentName1)
  CDS3_ADD_COMPONENT_ENTRY(ComponentName2)
  ...
  CDS3_ADD_COMPONENT_ENTRY(ComponentNameN)
CDS3_DECLARE_COMPONENTS_LIST_END(ListName)
```

2. The detailed information about the components list header file is listed in section 4.4.3.2. Please make sure that this header file **doesn't contain** a single include guard directive looking like this:

```
/* Don't do it in your components list header file! */
#ifndef __MY_HEADER1_H__
#define __MY_HEADER1_H__
/* MyHeader1.h content */
/* ... */
#endif
```

3. Define the `RTS_CONFIG_FILE` macro with the value containing a path to the components list header file either in the `sysdefines.h` header file of specific build configuration or as a `-D` compiler option:

```
/* sysdefines.h header file */
#ifndef RTS_CONFIG_FILE
# define RTS_CONFIG_FILE rtsconfig/uRtsCortexM3.h
#endif
```

4. Create the components exclusion list header file containing `_NOTIMPLEMENTED` macros for the components which were not included to the components list header file. Include this file to the `sysdefines.h` header file:

```
/* sysdefines.h header file */
#include "rtsconfig/CortexM3_NotImpl.h"
```

The information about the components exclusion list header file is listed in section 4.4.3.3.

5. Make your `sysdefines.h` header file accessible for the compiler used for building the MicroRTS binary executable. This can typically be done either by using the corresponding `-I` or `/I` compiler option or using the `IPATH` variable in the build configuration make-file.
6. Include the components source modules to the toolchain make-file or IDE project.

## 5.5 Adapting Core System Components

### 5.5.1 Overview

In most cases, the platform-independent functional components of the CODESYS V3 runtime system are ready to use without any changes in OEM-adaptations of MicroRTS. But the platform-dependent system components have to be adapted to a specific target CPU and/or hardware configuration of the target device.

For the MicroRTS profile, the following system components should be considered as first candidates for platform-specific adaptation:

1. **SysTarget**: should implement functions that provide a node name and a serial number of specific target device for the communication and device identification components.
2. **SysCpuHandling**: should contain a CPU-specific implementation of an IEC code entry point (**SysCpuCallIecFuncWithParams**), atomic operations on integer operands and stack unwinding operations.
3. **SysMem**: should contain a target specific implementation of memory management routines.
4. **SysTime**: should implement, at least, the **SysTimeGetMs** routine that returns a milliseconds counter.
5. **SysFlash**: should contain a set of functions for reading and writing the flash ROM unit that is going to be used for storing and executing an IEC application code.
6. **SysInt**: should implement, at least, a pair of functions implementing global lock/unlock operations (**SysIntEnableAll/SysIntDisableAll**).
7. There should also be a target-dependent function which is called at the very beginning of execution of MicroRTS to perform initial CPU and peripherals configuration. In the Cortex-M3 MicroRTS reference implementation, this function, named **sysInit**, is called on entering the MicroRTS entry point **main()**.
8. If there is a need to configure some components via the **CmpSettingsEmbedded** component interface, the separate **CmpSettingsEmbedded** root module can be created to organize and handle the parameters storage.
9. The **SysExcept** component could optionally be adapted, if it is required to handle exceptions in an IEC code.
10. Finally, if it is required to have support for communication between CODESYS V3 IDE and a target, via serial line or Ethernet connection, the corresponding component (**SysCom** or **SysSocketEmbedded**) should be implemented.

The process of adapting core system components, in general, may consist of the following steps:

1. Create a secondary C module named *SysComponentNameTargetName.c* for each component to be adapted in the target specific **Sys** folder. For example, for the **SysMem** component and the Cortex-M3 target CPU it would be **SysMemCortexM3.c**.
2. Add all the interface functions, which should be implemented by each component in a platform specific way, to the corresponding secondary modules. Make adaptation for the mandatory functions, the optional functions may return **ERR\_NOTIMPLEMENTED** error code.
3. Add a pair of functions (*SysComponentNameOSInit* and *SysComponentNameOSHookFunction*) referenced in the corresponding component root modules to the secondary modules of each component. For example, in the **SysMem** component these functions have names **SysMemOSInit** and **SysMemOSHookFunction**.

### 5.5.2 Specifying Target Identification

The **SysTarget** component provides the target identification information needed for the target to be part of CODESYS V3 communication infrastructure.

There are two functions in this component: **SysTargetGetNodeName** and **SysTargetGetSerialNumber**.

The first function returns the target node name that is displayed in the CODESYS V3 **Communications Settings** tab when the corresponding network node is found during network rescanning process. It is enough to implement this function as follows:

```
/* sysdefines.h header file */
RTS_RESULT CDECL SysTargetGetNodeName(RTS_WCHAR * pwszName, unsigned int *pnMaxLength)
{
    unsigned int uiLen;

    if (pnMaxLength == NULL)
        return ERR_PARAMETER;
}
```

```
uiLen = *pnMaxLength;
if (CAL_SysTargetGetConfiguredNodeName(pwszName, &uiLen) != ERR_OK)
{
    if (pwszName != NULL)
        CAL_CMUtlStrToW(SYSTARGET_NODE_NAME, pwszName, *pnMaxLength);
    *pnMaxLength = strlen(SYSTARGET_NODE_NAME) + 1;
}
else
    *pnMaxLength = uiLen;
return ERR_OK;
}
```

The SYSTARGET\_NODE\_NAME macro should be defined in the sysdefines.h header file as listed in section 5.2.2.

The SysTargetGetSerialNumber can optionally be implemented to return a target device serial number.

### 5.5.3 Implementing CPU-specific Functions

There are three mandatory functions in the SysCpuHandling component interface, which should be implemented in the secondary SysCpuHandling component module:

**SysCpuCallIecFuncWithParams:** an IEC code entry point. This routine can be implemented in a separate assembly module, or as a C routine (if inline assembly is supported by the toolchain). It should prepare a stack space with the size enough for passing parameters supplied by the pParam parameter.

The pParam content size is supplied by the iSize parameter, so the memory size, that has to be allocated on stack, should be equal to iSize plus a stack alignment delta, if iSize is not aligned by the natural stack alignment constant specific for a CPU. The natural stack alignment constant is specified by the *Device – ExtendedSettings – TargetSettings – memory-layout : stack-alignment* option in the device description file.

When the stack memory is allocated, the pParam content is copied to this space, and then the pIecFunc routine is called.

When the pIecFunc function returns, the stack memory content reserved prior to calling to pIecFunc should be copied back to pParam.

In most cases, the SysCpuCallIecFuncWithParams routine is readily available in the starter package delivered by 3S.

**SysCpuTestAndSetBit:** a primitive used to atomically set or reset a specified bit in the value referenced by the first parameter.

**SysCpuAtomicAdd:** a primitive used to atomically add or subtract the value supplied with the second parameter to/from the value referenced by the first parameter.

There is a generic implementation for these two functions provided by 3S. The only thing that has to be done to use this generic implementation is to define the following macros in the sysspecific.h header file:

```
/* sysspecific.h header file */
#define SYS_INT_PARAM          RTS_UI32
#define SYS_INT_LOCK(param)    SysIntDisableAll(&param)
#define SYS_INT_UNLOCK(param) SysIntEnableAll(&param)
```

Other functions exposed by the SysCpuHandling component interface can return ERR\_NOTIMPLEMENTED, especially if it is not required to support exceptions handling.

Please refer to the SysCpuHandlingCortexM3.c source code supplied with the Cortex-M3 MicroRTS Starter Package for more information about these functions.

The SysInt component contains two mandatory functions mentioned above: SysIntDisableAll and SysIntEnableAll. They are used to disable and enable all the interrupts that can be masked on the target CPU. These functions can be implemented in the SysInt component's secondary module or in a separate assembly file.

## 5.5.4 Implementing Memory Management

### 5.5.4.1 Overview

There are several topics that need to be considered while implementing a memory management infrastructure for the MicroRTS profile:

1. Definition of the runtime stack.
2. Free storage (a.k.a heap) implementation.
3. IEC code and data areas organization.

The last thing that is worth mentioning is an allocation of fixed-size memory blocks. This functionality may be required while developing device drivers.

### 5.5.4.2 Defining the Runtime Stack

The runtime stack for an embedded device is typically specified in the integrated development environment (IDE) used for the target firmware development, or directly in the source code, sometimes in conjunction with the linker configuration file.

In the Cortex-M3 MicroRTS reference implementation, the stack size is set to 1280 32-bit words (5120 bytes) by declaring a special static variable *stack\_loc*, which is then explicitly placed as the first element to the interrupt vector table:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\src\Source\System\startup.c */
static unsigned long stack_loc[0x500];
/*****
 * Interrupt Vector Table
 *****/
Interrupt interrupts[] __attribute__((section(".isr_vector"))) =
{
    // The initial stack pointer
    (Interrupt)((unsigned long)stack_loc + sizeof(stack_loc)),
    ResetISR, // 1 reset handler
    ... other vectors
};
```

The stack pointer value placed to the interrupt vector table contains an ending address of *stack\_loc* since the stack pointer “grows” towards lower addresses.

Please note, that there is one essential option specified in the *sysdefines.h* header file, which affects the stack usage significantly:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\sysdefines.h */
#define BTAG_MAX_NESTED_TAGS 8
```

This macro is used to limit the depth of recursive calls in the online services while deserializing incoming requests from CODESYS. For the normal operation of communication services, it is typically enough to define this macro as 4.

### 5.5.4.3 Defining the Runtime Free Storage (Heap)

The MicroRTS profile itself doesn't require any free storage (i.e. a heap) to be implemented, since there are no calls to *malloc/calloc/free* in the MicroRTS core components. But sometimes it is necessary to have a free storage for implementing some dynamic buffers allocation/deallocation while developing device drivers.

The low-end embedded CPU and microcontrollers are normally equipped with an integrated RAM unit for accessing one or several banks of static RAM. Each bank has a limited size (16K, 32K or more). In many cases it is required to organize a free storage in several banks, and even in separate regions of available memory with non-adjacent base addresses.

The *CmpHeapPool* component can be used for the free storage implementation that meets the requirements listed above. This component implements a flavor of the buddy algorithm that splits available memory regions to blocks with sizes of power of 2 and then attempts to satisfy incoming requests for memory blocks in an optimal way. There is one parameter defined by the *HMEM\_MIN\_BLOCK\_SIZE\_POWER* macro that can optionally be specified for this component in the *sysdefines.h* header file. The default value for this macro is set to 6, so as the minimal size of blocks managed by the component is 64 bytes.

The example of usage can be obtained in the SysMem component implementation:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Sys\SysMemCortexM3.c */
#include <CmpHeapPoolItf.h>

/* WARNING! It is not thread-safe! */
static RTS_HANDLE hHeapPool = RTS_INVALID_HANDLE;

RTS_RESULT CDECL SysMem0SHookFunction(RTS_UI32 ulHook, RTS_UINTPTR u1Param1, RTS_UINTPTR u1Param2)
{
    switch (ulHook)
    {
        case CH_INIT_SYSTEM:
        {
            {
                /* two regions reserved in CODESYS.ld are used for the heap */
                RTS_RESULT res;
                RTS_MEM_REGION heap_regions[] =
                {
                    { (RTS_UINTPTR)HEAP_START, (RTS_SIZE)(HEAP_END - HEAP_START) },
                    { (RTS_UINTPTR)HEAP_REGION2, (RTS_SIZE)HEAP_REGION2_SIZE },
                    { (RTS_UINTPTR)0, 0 }
                };
                hHeapPool = CAL_HeapPoolInit(heap_regions, &res);
            }
            break;
        }
        default:
            break;
    }
    return ERR_OK;
}

void* CDECL SysMemAllocData(char *pszComponentName, RTS_SIZE ulSize, RTS_RESULT *pResult)
{
    return CAL_HeapPoolAlloc(hHeapPool, ulSize, pResult);
}

RTS_RESULT CDECL SysMemFreeData(char *pszComponentName, void* pData)
{
    return CAL_HeapPoolFree(hHeapPool, pData);
}

```

The regions used in the implementation above are reserved in the linker configuration file:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\CODESYS.ld */
MEMORY
{
    /* ... */
    /* Additional heap memory (3072 bytes) */
    E_SRAM2 (rwx) : ORIGIN = 0x20083400, LENGTH = 0xC00
}
/* ... */
SECTIONS
{
    /* ... other regions ... */
    PROVIDE(__HEAP_REGION2 = 0x20083400);
    PROVIDE(__HEAP_REGION2_SIZE = 0xC00);
    .text :
    {
        _text = .;
        KEEP(*(.isr_vector))
        *(.text*)
        *(.rodata*)
        _etext = .;
    } > FLASH
    .data : AT(ADDR(.text) + SIZEOF(.text))
    {
        _data = .;
        *(vtable)
        *(.data*)
        _edata = .;
    } > SRAM
    __exidx_start = .;
    .ARM.exidx :

```



```

{
*(.ARM.exidx* .gnu.linkonce.armexidx.*)
} > SRAM
__exidx_end = .;
.bss :
{
    _bss = .;
    *(.bss*)
    *(COMMON)
    _ebss = .;
} > SRAM
_end = .;
PROVIDE(__HEAP_START = _end);
/* 32 upper bytes area is used by IAP! */
PROVIDE(__HEAP_END = 0x10007FC0);
}

```

The `__HEAP_START` and `__HEAP_END` fixups are defined in the first SRAM bank of the LPC1768 right after the bss and data segments. The `__HEAP_REGION2` and `__HEAP_REGION2_SIZE` fixups refer to the region in the second SRAM bank right after the IAP (In Application Programming) buffer required for the SysFlash component to write to the flash memory.

The `syspecific.h` header file contains redefinitions for these fixups, as they can easily be used as normal variables in a C code:

```

/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\syspecific.h */
extern int __HEAP_START;
extern int __HEAP_END;
#define HEAP_START (&__HEAP_START)
#define HEAP_END   (&__HEAP_END)

extern int __HEAP_REGION2;
extern int __HEAP_REGION2_SIZE;
#define HEAP_REGION2 (&__HEAP_REGION2)
#define HEAP_REGION2_SIZE (&__HEAP_REGION2_SIZE)

```

#### 5.5.4.4 Defining IEC Application Areas

Sections and describe how to specify the IEC code and data memory areas in the device description file. This information is needed for the CODESYS IEC compiler to place executable code and data of the user application.

This sub-section contains the information on how to manage the IEC memory areas in the runtime system.

The SysMem component interface exposes two functions that need to be implemented in the component's secondary module in a target specific way. The compact and MicroRTS profiles of the CODESYS runtime system typically use absolute addresses for the IEC memory areas. The IEC memory areas specified in the device description file for the target should be explicitly reserved in the linker description file (or in the locator description file) and the corresponding fixups may be redefined in the `syspecific.h` header file as described in sections and . Then the two IEC area handling functions have to be implemented in the SysMem component's secondary module: `SysMemAllocArea` and `SysMemFreeArea`.

In the Cortex-M3 MicroRTS reference implementation these functions look as follows:

```

/* IEC Area Descriptor */
typedef struct tagDataAreaSpec
{
    /* area start address */
    void* pAreaAddress;
    /* area size */
    RTS_UI32 size;
    /* 0: volatile data; or DA_RETAIN */
    RTS_UI16 type;
    /* 1: allocated; 0: not allocated */
    RTS_UI16 inUse;
} DataAreaSpec;

```

```
static DataAreaSpec s_DataAreasList[] =
{
  { (void*)DATA_AREA1_START, (RTS_UI32)DATA_AREA1_SIZE, 0, 0 },
  { (void*)DATA_AREA2_START, (RTS_UI32)DATA_AREA2_SIZE, 0, 0 }
};

static DataAreaSpec s_RetainAreasList[] =
{
  { (void*)RETAIN_START, (RTS_UI32)RETAIN_SIZE, (RTS_UI16)DA_RETAIN, 0 }
};

static DataAreaSpec* DataAreaSpec_get(DataAreaSpec* pAreaList,
                                      RTS_SIZE list_size,
                                      void* pAreaAddress,
                                      RTS_SIZE requested_size)
{
  /* if pAreaAddress != NULL, then the allocated area with this address is requested */
  /* if pAreaAddress == NULL, then a free area is requested */
  RTS_UI16 in_use_condition = (RTS_UI16) (NULL != pAreaAddress);
  int idx;

  for (idx = 0; idx < list_size; ++idx)
  {
    if (pAreaList[idx].size >= requested_size && in_use_condition == pAreaList[idx].inUse)
    {
      if (NULL != pAreaAddress && pAreaList[idx].pAreaAddress != pAreaAddress)
        continue;
      return &pAreaList[idx];
    }
  }

  return NULL;
}

void* CDECL SysMemAllocArea( char *pszComponentName,
                            unsigned short usType,
                            RTS_SIZE ulSize,
                            RTS_RESULT *pResult)
{
  void* pAreaAddr = NULL;
  DataAreaSpec* pArea = NULL;
  DataAreaSpec* pAreaList = NULL;
  RTS_SIZE list_size = 0;
  RTS_RESULT Result = ERR_FAILED;

  if (IsArea(usType, DA_RETAIN))
  {
    pAreaList = s_RetainAreasList;
    list_size = sizeof(s_RetainAreasList) / sizeof(DataAreaSpec);
  }
  else if (!IsArea(usType, DA_CODE))
  {
    pAreaList = s_DataAreasList;
    list_size = sizeof(s_DataAreasList) / sizeof(DataAreaSpec);
  }

  if (0 != list_size)
  {
    pArea = DataAreaSpec_get( pAreaList, list_size, NULL, ulSize );

    if (NULL != pArea)
    {
      pAreaAddr = pArea->pAreaAddress;
      pArea->inUse = (RTS_UI16) 1;
      Result = ERR_OK;
    }
  }

  RTS_SETRESULT(pResult, Result);

  return pAreaAddr;
}
```

```
RTS_RESULT CDECL SysMemFreeArea(char *pszComponentName, void* pCode)
{
    RTS_RESULT Result = ERR_PARAMETER;
    if (NULL != pCode)
    {
        DataAreaSpec* pArea = DataAreaSpec_get( s_RetainAreasList,
                                                sizeof(s_RetainAreasList) / sizeof(DataAreaSpec),
                                                pCode,
                                                0);

        if (NULL == pArea)
        {
            pArea = DataAreaSpec_get( s_DataAreasList,
                                      sizeof(s_DataAreasList) / sizeof(DataAreaSpec),
                                      pCode,
                                      0);
        }

        if (NULL != pArea)
        {
            Result = ERR_OK;
            pArea->inUse = (RTS_UI16) 0;
        }
    }

    return Result;
}
```

The DataAreaSpec structure is used to declare static arrays of data area descriptors for input, output, global and retain variables. When some area is requested during an IEC application startup, the SysMemAllocArea routine returns an area address obtained from the corresponding static array of descriptors, and marks this area descriptor as it is now in use.

When some area is deallocated (if an IEC application is being deleted) by the call to SysMemFreeArea, the corresponding area descriptor is marked as it is no longer in use.

**Note:** If you wish to implement retain variables using an integrated on-chip SRAM, the microcontroller chip power supply should be equipped with a battery and special circuitry to switch over the CPU power when the main power is interrupted.

#### 5.5.4.5 Fixed-size Memory Blocks Allocation

While developing device drivers or other system software, it is sometimes necessary to allocate and deallocate memory blocks of the same size which are then used as objects of some user defined data type. The use of general purpose memory allocation functions (malloc/free) is typically not the best approach to achieve this goal, since it could lead to memory fragmentation and even exhaustion.

There are two mechanisms available in the CODESYS V3 runtime system: a generic memory pool interface and a separate FixedBlocksPool interface exposed by the CmpMemPool component.

The CmpMemPool generic interface is used for allocation/deallocation memory blocks of fixed size from static memory buffers and/or from the free storage. Each CmpMemPool pool object can also be used as a sequential container for objects occupying memory blocks allocated from this or some other memory pool.

The FixedBlocksPool is a pure memory allocator that can be used for allocation/deallocation memory blocks of fixed size from static memory buffers and/or from the free storage, and also for reclaiming all memory blocks provided by the allocator using a single function call. Reclaiming means that all the blocks managed by some pool are now free but not returned to the free storage.

There are two differences between the pool object provided by a generic CmpMemPool interface and the one provided by FixedBlocksPool:

1. FixedBlocksPool doesn't have any bookkeeping overhead in each memory block.
2. FixedBlocksPool doesn't have an internal linked list for containing objects occupying memory blocks allocated by this or another pool.

There are two macros related to the CmpMemPool component:

**RTS\_MEMPOOL\_VER2:** if this macro is defined in the `sysdefines.h` header file, the `FixedBlocksPool` becomes available for usage and the generic `CmpMemPool` interface becomes to be implemented in terms of `FixedBlocksPool` using the same allocation/deallocation algorithm.

**FIXED\_BLOCK\_ALLOCATOR\_SEPARATED:** if the `RTS_MEMPOOL_VER2` is **not** defined, and the `FIXED_BLOCK_ALLOCATOR_SEPARATED` macro is defined in the `sysdefines.h` header file, the `FixedBlocksPool` implementation resides in the `FixedBlocksAllocator.c` module and is excluded from the `CmpMemPool.c` module. In this case the `FixedBlocksAllocator.c` module should be included into the `makefile` or IDE project used for building the MicroRTS binary executable.

### 5.5.5 Implementing System Ticks

The `SysTime` component interface exposes three functions used by core components of RTS for obtaining the information about system time: `SysTimeGetMs`, `SysTimeGetUs`, `SysTimeGetNs`.

The `SysTimeGetMs` function is mandatory. It returns the monotonic rising milliseconds counter. This function should always be implemented in the `SysTime` component secondary module because it is used by communication components dealing with timeouts and, in some cases, by the IEC tasks scheduler. In Cortex-M3 MicroRTS this function is implemented in the simplest way returning the value of the counting variable that is incremented by the Cortex-M3 `TIMER0` interrupt service routine (for details please refer to the `isr_TIMER0` routine implementation in `Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\src\Source\System\startup.c`).

While implementing `SysTimeGetMs` in a similar way, please make sure that the priority of the interrupt used for incrementing the milliseconds counting variable is higher than priorities of other interrupts, especially of those in which servicing requires other lower priority interrupts to be disabled.

The rest two functions (`SysTimeGetUs`, `SysTimeGetNs`) are not implemented in MicroRTS.

### 5.5.6 Implementing Access to a Flash Memory

The MicroRTS profile implies that an IEC application code is placed and executed directly in an integrated or external flash memory. The `SysFlash` and `SysFileFlash` components are in charge to provide support for this functionality. The flash memory is logically split in two areas: the first area (`FA_CODE`) is used for storing an IEC application binary code and constant data, and the second one (`FA_FILE`) for emulating a simplified file system that is required for proper handling of IEC applications.

The `SysFlash` component consists of, at least, two modules: `Components\SysFlash\SysFlash.c` and the platform-dependent secondary module implementing the real read/write/erase access operations for a specific flash memory device. The secondary module should implement the following routines:

1. `SysFlashInit`: initializes the platform-dependent flash abstraction layer implementation.
2. `SysFlashErase_`: erases a specific region of the `FA_CODE` or `FA_FILE` flash memory area.
3. `SysFlashRead_`: reads a specific region of the `FA_CODE` or `FA_FILE` flash memory area.
4. `SysFlashWrite_`: writes to a specific region of the `FA_CODE` or `FA_FILE` flash memory area.
5. `SysFlashFlush_`: writes a cache buffer to a specific region of the `FA_CODE` or `FA_FILE` flash memory area.
6. `SysFlashGetPhysicalAddress`: returns the beginning address of the `FA_CODE` or `FA_FILE` memory area.
7. `SysFlashGetSize`: returns the size of the `FA_CODE` or `FA_FILE` memory area.

The integrated flash memory of modern microcontrollers is accessed either by reading and writing special registers in the CPU address space or by utilizing a special flash access interface implemented in the microcontroller's system firmware. In either case, the flash media can typically be written or erased on a per-block basis, which means that only one block of fixed size can be written or erased at a time. Therefore, the exact implementation of `SysFlashWrite_` and `SysFlashErase_` functions can be quite complicated, since it might be necessary to have an additional translation layer converting write and erase requests given for arbitrary flash regions to write and erase operations that can be performed for physical blocks of flash media.

The Cortex-M3 MicroRTS reference implementation uses IAP (In-Application Programming) interface available in the LPC1768 chip to access integrated flash memory (see Platforms\Native\CortexM3\NXP-LPC1768\Sys\SysFlashCortexM3.c and Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\src\Source\System\lpc17xx\_flash.c for details). All the write requests are stored in a special IAP cash buffer reserved in the linker configuration file:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\CODESYS.ld fragment */
MEMORY
{
  FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 0x0080000

  /* 32 upper bytes area is used by IAP, 64 bytes reserved */
  SRAM (rwx) : ORIGIN = 0x10000000, LENGTH = 0x7FC0

  ... other entries

  /* IAP buffer area */
  IAP_BUF (rwx) : ORIGIN = 0x20083000, LENGTH = 0x400

  ... other entries
}
```

The write requests are cached while the size of cached data is less than the cash size (1024 bytes) or until the SysFlashFlush\_ routine is called. In latter case, the whole cash content is being written to the corresponding sector of flash media. Such a non-trivial handling is forced by the requirement that only 256/512/1024/4096-aligned memory SRAM addresses are allowed to be used for copying SRAM content to flash. The read requests are implemented by a simple memory copy operation.

SysFlashGetPhysicalAddress and SysFlashGetSize routines return the beginning address and size of flash memory used for storing code, constant data and file system emulation. If the flash area type passed to these routines is FA\_CODE, they should return the values specified in the device description file for code area (see section 5.2.5.2).

The CmpSettingsEmbedded component may also contain the following parameters for the SysFlash and SysFileFlash component:

```
/* Platforms\Native\CortexM3\NXP-LPC1768\sysdefines.h */

#define FILE1_SIZE 0x10000
#define FLASH_ERASE_BLOCK_SIZE 0x10000
#define FLASH_WRITE_BLOCK_SIZE 1024

/* File system emulation: */
/* Application.app can be accessed via SysFile interface. */
#define FILE_MAP FILE_DESC m_FileSystem[] = \
{ \
  /* Name      Offset  MaxSize  read index write index */ \
  {"Application.app", 0x0, FILE1_SIZE, 0xFFFFFFFF, 0xFFFFFFFF}, \
};

#define SETTG_ENTRIES_INT \
{"SysFlash", "WriteBlockSize", FLASH_WRITE_BLOCK_SIZE }, \
{"SysFlash", "EraseBlockSize", FLASH_ERASE_BLOCK_SIZE}, \
{0, 0, 0}
```

The *m\_FileSystem* array contains a flat list of file names mapped to the corresponding flash memory regions to provide a file system emulation required by the CmpAppEmbedded component. In the example above, the Application.app file size is the same as the code area size, which means that the code area will be erased, if the SysMemFreeArea function is called.

The two parameters *WriteBlockSize* and *EraseBlockSize* can be used to specify sizes of blocks that can be written and erased in a single low-level operation. These parameters are used in the SysFlash component root module.

## 5.5.7 Implementing Exceptions Handling

In order to have a possibility to support exceptions handling mechanism implemented in MicroRTS, the C runtime library of the toolchain should support the setjmp/longjmp functionality to provide “non-local jumps”.

The following steps should be made to support proper handling for exceptions that might occur in an IEC code:

1. Add the SysExcept component root module to the build configuration and specify the corresponding component name in the components list header file.
2. Create a secondary platform-dependent C module for the SysExcept component and add it to the build configuration.
3. Define the following macros in the sysdefines.h header file:

```
/* sysdefines.h */
#define RTS_STRUCTURED_EXCEPTION_HANDLING
#define RTS_IECTASK_STRUCTURED_EXCEPTION_HANDLING
#define EXCPT_MAX_NUM_OF_SEH_HANDLER 1
#define EXCPT_NUM_OF_STATIC_CONTEXT 2
#define EXCPT_DEFAULT_NUM_OF_INTERFACES 1
```

3. Create an exception handling routine in the SysExcept component secondary module and make a call to this routine from the primary exception/fault handler typically written in assembly language to pick up the correct exception context.

This routine should check whether or not the exception occurred in an IEC code area. If it was the case, a variable of type RegContext has to be filled out with the values of exception context registers including the frame pointer (BP), the instruction pointer (IP) and the stack pointer. Please make sure that the RegContext variable contains the context information referring to the exact point where exception occurred. (Note: In the CortexM3 MicroRTS reference implementation, it's appeared to be necessary to disable the Write Buffer for CPU-to-Memory transfers, otherwise most types of exceptions fired by the Cortex-M3 core were imprecise. See the NVIC\_SCBDelInit routine in Platforms\Native\CortexM3\NXP-LPC1768\Projects\CodeSourcery\src\Source\System\lpc17xx\_nvic.c for details.)

Then determine the reason for the exception and convert it to a platform independent value defined in the SysExcept component interface (see RTSEXCEPT\_ macros in SysExceptIrf.m4). Put the reason value to a local variable of type ExceptionCode (to the ulCode field). Set to 0 the bOSException field in this variable.

Reset a hardware bit (if it is required for the CPU in use) indicating the reason for an exception to continue execution.

Make a call to the SysExceptGenerateException routine with the first parameter set to RTS\_INVALID\_HANDLE, the second – to the value of local variable of type ExceptionCode filled in as explained above, and the third – to the value of local variable of type RegContext containing the correct exception context.

4. In the platform initialization sysInit routine, enable all the exceptions, which need to be handled.
5. Implement the following functions in the SysCpuHandling component secondary module:

**SysCpuGetCallstackEntry:** this routine should obtain a frame pointer and a return address of the previous caller out of the current frame pointer.

**SysCpuGetCallstackEntry2:** this routine should make a call to the SysCpuGetCallstackEntry, if an exception occurred in an IEC code area. Otherwise, it should utilize another approach to correctly unwind the call stack in the MicroRTS code segment.

**SysCpuGetInstancePointer:** this routine should retrieve a pointer to an instance of Functional Block. The location depends on the CODESYS code generator for a specific CPU.

**SysCpuGetMonitoringBase:** this routine is used to obtain a frame pointer of an IEC function for

monitoring local variables.

## 5.6 Adapting Communications

The communication infrastructure of the CODESYS runtime system can be adapted for the MicroRTS profile in a similar way as it is done for the compact profile. The following considerations are worth mentioning:

1. Only one block driver can be used in the MicroRTS and compact profiles. If two block drivers are added accidentally, the first of them listed in the components list header file will be active.
2. Care should be taken when reserving communication buffers. Reserve only a minimal necessary amount of memory and double-check buffers sizes after building the MicroRTS binary executable.
3. Some third-party libraries that can be used for implementing the SysSocketEmbedded component interface contain code leading to an unaligned access exception on ARM-based CPUs. The Cortex-M3 MicroRTS implementation for the TI-LM3S9B96 is shipped with such a library, that's why the UNALIGN\_TRP fault is not enabled in this implementation.

## 5.7 Configuring the Logger

To support the CmpLogEmbedded component in the MicroRTS:

1. Add the CmpLogEmbedded component to the components list header file.
2. Exclude CMPLOG\_NOTIMPLEMENTED and CMPLOGSRV\_NOTIMPLEMENTED macros from the exclusion header file.
3. Include CmpLogEmbedded.c and CmpLogEmbeddedSrv.c modules to the build configuration.
4. Define the following macros in the sysdefines.h header file:

```
#define LOG_STD_MAX_NUM_OF_FILES 0
#define LOG_STD_MAX_FILE_SIZE 0
#define LOG_STD_MAX_NUM_OF_ENTRIES 5
#define LOG_MAX_INFO_LEN 96
```

**Note:** Please keep the value of LOG\_STD\_MAX\_NUM\_OF\_ENTRIES as small as possible.

## 5.8 Implementing a Debugging Console

The source code delivered with the Cortex-M3 MicroRTS Starter Package contains a light-weight infrastructure for implementing a debugging console over serial line (or some other similar interface).

If the target device has a serial port not used for the runtime communication purposes, the following steps could be made to enable a debugging console:

1. Implement the following routine in the platform-dependent source code:

```
#ifdef SERIAL_CON_ENABLED

#if !defined(SER_CON_INT_BUFFER_SIZE) || (SER_CON_INT_BUFFER_SIZE < 256)
#define SER_CON_INT_BUFFER_SIZE 256
#endif

#ifndef CON_PORT
#define CON_PORT UART2_BASE
#endif

#ifndef RTS_SIZE_DEFINED
typedef size_t RTS_SIZE;
#endif

extern int CMUtlPortableVsnprintf(char* str, RTS_SIZE buf_len, const char* fmt, va_list ap);

void con_out(const char* format, ...)
```

```
{
    va_list args;
    RTS_SIZE len = 0;
    char buffer[SER_CON_INT_BUFFER_SIZE];

    va_start(args, format);
    CMUtlPortableVsnprintf( buffer, SER_CON_INT_BUFFER_SIZE, format, args );
    va_end(args);

    for (len = 0; len < (SER_CON_INT_BUFFER_SIZE - 2) && buffer[len] != '\0'; ++len);

    /* len = strlen(buffer); */
    buffer[len++] = '\0';

    UARTCharPutLine( CON_PORT, (uint8_t*)buffer, len);
}

#else
void con_out(const char* format, ...) { }
#endif
```

The CON\_PORT macro specifies the serial port base address to be used for a console port.

2. Implement the UARTCharPutLine routine or use an existing implementation from a third-party library. This routine should write a sequence of bytes to a serial port synchronously.
3. Define the SERIAL\_CON\_ENABLED macro in the sysdefines.h header file (or in the make-file).

The following code snippet illustrates the usage serial console in the MicroRTS source code:

```
#include "CmpStd.h"

#ifdef PATHS_RELATIVE
    #include "SysExcept/SysExceptDep.h"
#else
    #include "SysExceptDep.h"
#endif

USEIMPORT_STMT

/* If this header file is included, all occurrences of M_DBG */
/* are replaced for empty strings. */
/* #include <OptionalIncludes/Utils/dbg/udbg.h> */

/* If this header file is included, all occurrences of M_DBG */
/* are replaced for calls to printf. */
#include <OptionalIncludes/Utils/dbg/idbg.h>

void cm3_hard_fault_trap (RTS_UI32* excpt_context, RTS_UI32 iecBP)
{
    unsigned int stacked_r0;
    unsigned int stacked_r1;
    unsigned int stacked_r2;
    unsigned int stacked_r3;
    unsigned int stacked_r12;
    unsigned int stacked_lr;
    unsigned int stacked_pc;
    unsigned int stacked_psr;

    stacked_r0 = ((unsigned long) excpt_context[0]);
    stacked_r1 = ((unsigned long) excpt_context[1]);
    stacked_r2 = ((unsigned long) excpt_context[2]);
    stacked_r3 = ((unsigned long) excpt_context[3]);

    stacked_r12 = ((unsigned long) excpt_context[4]);
    stacked_lr = ((unsigned long) excpt_context[5]);
    stacked_pc = ((unsigned long) excpt_context[6]);
    stacked_psr = ((unsigned long) excpt_context[7]);

    M_DBG ("\n\nHard Fault occurred!\r\n");
    M_DBG1 ("R0 = %X\r\n", stacked_r0);
    M_DBG1 ("R1 = %X\r\n", stacked_r1);
    M_DBG1 ("R2 = %X\r\n", stacked_r2);
    M_DBG1 ("R3 = %X\r\n", stacked_r3);
    M_DBG1 ("R12 = %X\r\n", stacked_r12);
}
```



```
M_DBG1 ("LR [R14] = %X subroutine call return address\r\n", stacked_lr);
M_DBG1 ("PC [R15] = %X program counter\r\n", stacked_pc);
M_DBG1 ("PSR = %X\r\n", stacked_psr);
M_DBG1 ("BFAR = %X\r\n", (*((volatile unsigned long *) (0xE000ED38))));
M_DBG1 ("CFSR = %X\r\n", (*((volatile unsigned long *) (0xE000ED28))));
M_DBG1 ("HFSR = %X\r\n", (*((volatile unsigned long *) (0xE000ED2C))));
M_DBG1 ("DFSR = %X\r\n", (*((volatile unsigned long *) (0xE000ED30))));
M_DBG1 ("AFSR = %X\r\n", (*((volatile unsigned long *) (0xE000ED3C))));
M_DBG1 ("SCB_SHCSR = %X\r\n", SCB->SHCSR);

/* NOTE: CPU reset might be issued here */
lpc17xx_wdt_reset();

while (1);
}
```

If the *\_OptionalIncludes/Utils/dbg/idbg.h* header is included to a C module, all occurrences of the *M\_DBG* macros that follow after this include directive will be replaced for calls to `printf` (which, in turn, will be replaced for `con_out`, if the `SERIAL_CON_ENABLED` macro is defined).

If the *\_OptionalIncludes/Utils/dbg/udbg.h* header is included to a C module instead of *idbg.h*, all occurrences of the *M\_DBG* macros that follow after this include directive will be replaced for empty lines.

## Change History

<b>Version</b>	<b>Description</b>	<b>Editor</b>	<b>Date</b>
0.1	Issued	AL	25.05.2012
0.2	Review	AH	14.06.2012
1.0	Release	AH	10.07.2012
1.1	CDS-29303	MN	17.09.2012
2.0	Release	MN	03.12.2012