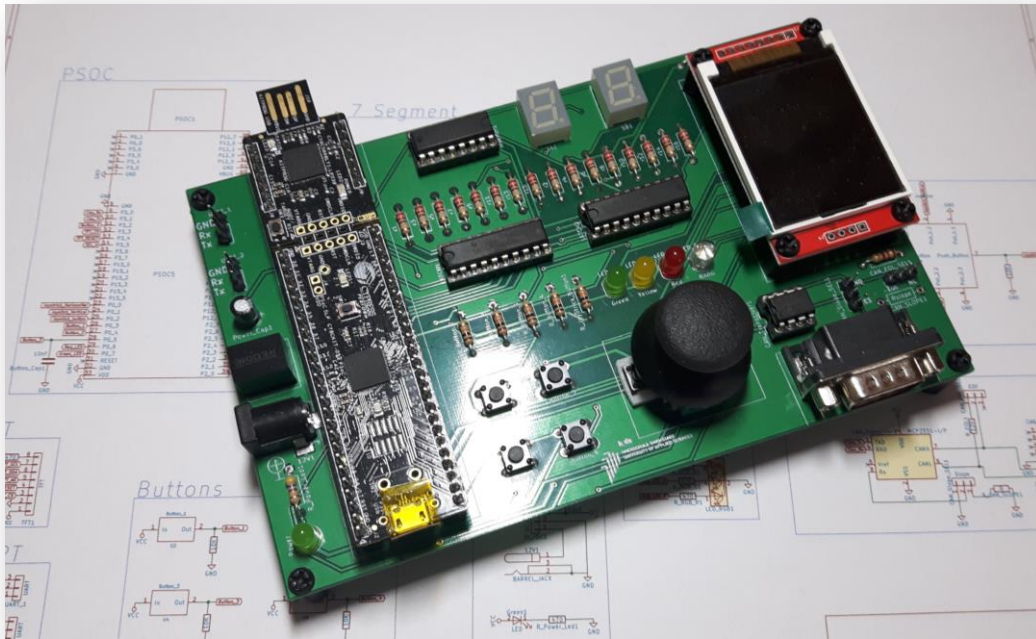


Embedded Architectures and Operating Systems

Workbook (v2018c)

Prof. Dr.-Ing. Peter Fromm



© University of Applied Sciences Darmstadt

Copyright

Copyright © 2018 by Prof. Peter Fromm, University of Applied Sciences Darmstadt

All rights reserved. No part of this publication may be reproduced, distributed, or transmitted in any form or by any means, including photocopying, recording, or other electronic or mechanical methods, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical reviews and certain other noncommercial uses permitted by copyright law. For permission requests, write to the publisher at the address below.

Prof. Dr.-Ing. Peter Fromm
Microcontroller and Information Technology
Hochschule Darmstadt - University of Applied Sciences
Birkenweg 8
64295 Darmstadt

peter.fromm@h-da.de

Content

Introduction	1
Some Background	4
1 Creating a first embedded “Hello World” project	7
1.1 Let’s light the LED	8
1.2 Let’s press a button.....	14
1.2.1 Pure hardware solution	14
1.2.2 A first software control loop	14
1.2.3 Real logic - the toggling LEDs	16
1.3 Introducing interrupts	18
1.3.1 A simple PIN interrupt.....	18
1.3.2 Interrupt names, priorities and service routines	22
1.3.3 Many different places to set interrupts.....	23
1.4 Light Effects	26
1.4.1 Traffic Light	26
1.4.2 Fader.....	27
1.5 Some simple console output	29
1.5.1 Iteration 1 - USART peripheral.....	29
1.5.2 Iteration 2 - Terminal Program	31
1.5.3 Iteration 3 - Buffers	32
1.6 A first timer	33
1.6.1 Iteration 1 - Setting up the peripherals	33
1.6.2 Iteration 2 - Hardware Debugging	35
1.6.3 Iteration 3 - Lessons learned.....	37
1.6.4 Iteration 4 - Clean up	38
2 Complex Device Drivers	39
2.1 Joystick and RGB LED.....	40
2.1.1 Setting up the project structure	40
2.1.2 Skinny sheep and interface design	43

2.1.3	Implementation of the Skinny Sheep (LED driver)	45
2.1.4	Lessons learned.....	48
2.1.5	Adding the fur - the joystick driver	49
2.2	A Seven Segment and Button Driver.....	51
2.2.1	Seven Segment Driver	52
2.2.2	Button Driver.....	54
3	A first application	57
3.1	Reaction Game.....	58
3.1.1	Iteration 1	60
3.1.2	Iteration 2	60
3.2	A simple encrypted protocol.....	62
3.2.1	Iteration 1 - Elementary data transmission	63
3.2.2	Iteration 2 - Ringbuffer class	64
3.2.3	Iteration 3 - Combining the ringbuffer and the driver.....	65
3.2.4	Iteration 4 - Protocol development	66
3.2.5	Iteration 5 - Encryption Algorithm	69
3.2.6	Iteration 6 - Getting it all together	69
4	Erika OS	71
4.1	Setting up the OS	72
4.1.1	Iteration 1 - A first task.....	72
4.1.2	Iteration 2 - A first blinking LED.....	74
4.1.3	Iteration 3 - A simple watch	79
4.1.4	Iteration 4 - Adding a reset button.....	80
4.2	Technical background - Interrupts.....	82
4.2.1	Configuring interrupts using ISR components.....	82
4.2.2	Special component configuration	84
4.3	Inter-Task Communication, Messaging and Critical Sections	86
4.3.1	Iteration 1 - Task creation and calling order.....	86
4.3.2	Iteration 2 - Changing the timing / a first critical section.....	88
4.3.3	Iteration 3 - Messaging	91
4.3.4	Iteration 4 - Using events to improve messaging	98

4.4	OSEK Error Handling and Hook Functions	100
4.4.1	Iteration 1 - Pre and Post Task Hook	100
4.4.2	Iteration 2 - Adding timestamps for traceability	101
4.4.3	Iteration 3 - Central Error Handler	102
4.4.4	Iteration 4 - OSShutdown	102
5	Reactive Systems / State Machines	103
5.1	Reaction Game	104
5.1.1	Requirements	104
5.1.2	Analysis	105
5.1.3	Erika elements	106
5.1.4	State Maschine	108
5.1.5	Arcadian Style	109
5.2	MP3 - Player	112
5.2.1	Requirements	112
5.2.2	Analysis and Design	113
5.2.3	Iteration 1 - Initial implementation	114
5.2.4	Iteration 2 - Lookup Table	114
5.2.5	Iteration 3- Adding real song data (optional)	114
5.3	Smart Volume Control	115
5.4	Electronic lock	116
6	Embedded Architectures	117
6.1	A light version of the Autosar RTE - Electronic Gaspedal	118
6.1.1	Iteration 1 - Configuration of the RTE	119
6.1.2	Iteration 2 - Add the RTE to your project	120
6.1.3	Iteration 3 - Getting it compilable	120
6.1.4	Iteration 4 - Getting it running	121
6.1.5	Iteration 5 - Extensions (optional)	121
6.2	Timing Supervision	122
6.2.1	Hardware Watchdog Driver	122
6.2.2	Alive Watchdog	123
6.2.3	Deadline Monitoring (optional)	123

6.3	Communication Stack.....	125
7	Real Fun	126
7.1	Morse coder/decoder	127
7.2	The game of PONG	128
7.3	Distributed version SOCCER	129
8	Annex - PSOC and LabBoard Pinning	130
9	Annex - Firmware Update	132
10	Annex - Using Doxygen	133

Introduction

Motivation and Goal

Nobody learns playing the violin by listening to Mozart. This is especially true for developing high quality embedded systems. Although many students already have worked with microcontrollers before, the experience shows that developing and validating a complex embedded architecture remains a challenging task for many reasons:

- Modern microcontrollers have very complex peripherals. Developing and understanding basic software is far from trivial.
- Previous experience in development is often restricted to limited implementation task, technical analysis and taking sensible design decisions are huge challenges
- Design patterns for embedded software development are not known.
- Modern architectural standards like Autosar remain vague and magic.
- Safety and security requirements become more and more relevant for embedded devices.

The goal of this workbook is to enable students to build up an embedded system from scratch, to analyze complex problems, to plan the development steps including analysis, design, coding and test phases, to understand and use embedded design patterns and to be able to use industrial solutions like Autosar OS and Autosar RTE.

Topics like safety and security for embedded devices will be touched but not focused. For getting a deeper understanding of these highly interesting and relevant topics, additional courses / projects will be offered.

Structure of the document

The document consists of a series of exercises, which will be implemented on the faculty's PSOC-LabBoard. Every chapter will form an independent exercise, which shall be implemented by the student as lab at home.

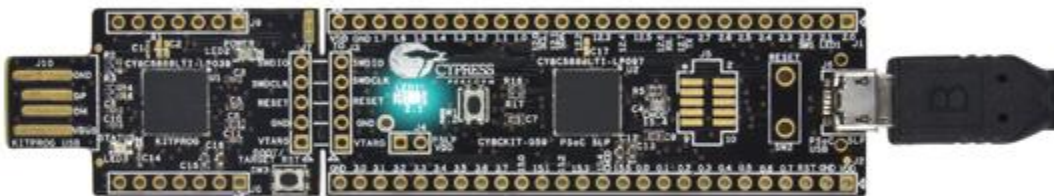


Figure 1 - PSOC5LP

The first chapters will address basic challenges focusing on getting to know the board and the development environment. You will create first simple digital and software applications and learn to read the board schematics.

In the middle section, the focus will be put on introducing the Erika OS, a free version of Autosar OS, which has been migrated to the PSOC5 board as well as some elementary design patterns. Developing own tasks, inter-task communication as well as different design patterns for state machine design and development of complex device drivers will be a focus here.

In the last chapter, the gained knowledge will be used to implement some more complex but fun applications.

At the beginning of every exercise, a brief overview of the expected effort, difficulty level and learning focus will be given. This will help you to evaluate your own performance. As a rough guidance:

- Just being able to solve category A exercises - this will very likely not be sufficient
- Being able to solve category B exercises but require significantly more time - you have reached the level of just passing the subject
- Being able to solve category B in time - this starts to look solid
- Being able to solve category C but require more time - definitely moving in the right direction
- Being able to solve category C in time - great job
- Being able to solve category C in time and extend them with own ideas - please contact me for a job offer ;-)

Some remarks on formatting

Especially in the first exercises, you will find a step by step recipe for the programs to get used to the hardware and development environment.

Blocks formatted in Bold and in a box are very important and need to be fully understood.

Side information which provides background information or invites you into some private investigation stories is formatted like this.

Question:

In some of the exercises, you will have to answer some questions and find some information in order to complete the exercise. These blocks are formatted italic.

Credits

The Labboard has been designed and produced together with a group of master students in 2017/2018 who own credits for the work.

- Aaron Correy created the first idea for the board and searched for the components.
- Hesham Mohamed created the schematics and layout.
- Carlos Martinez ported the Erika OS to the board and created the PSOC creator component for it.
- Thomas Barth reviewed the board, added some important improvements and provided the display driver.
- And last but not least Juan Marcano, who spent days and weeks in the lab soldering and testing all the boards.

The idea of the board is to provide an easy to use platform for the development of own projects but at the same time being powerful enough to run industrial solutions. The installed hardware components use typical peripherals found in most embedded applications, ranging from simple GPIO's to ADC, PWM, SPI, CAN, UART and more. The flexibility of the PSOC allows a simple extension also on hardware side. The exercises in this workbook therefore only serve as a starting point, be creative, try implementing own ideas and get it running!

I wish you lots of fun exploring the fascinated world of embedded systems!

Prof. Peter Fromm

Some Background

The PSOC-LabBoard contains a set of hardware components for own experiments:

- 3 traffic light LEDs
- 1 RGB LED
- 4 buttons
- 1 joystick + button
- 2 UART channels
- 1 CAN channel
- 1 TFT touch display with integrated SD card reader
- Optional external power supply (for most experiments, the power supply through the USB connector will do)

The following 2 pictures illustrate the layout of the board. For more detailed information, please check the schematics which are provided as own document.

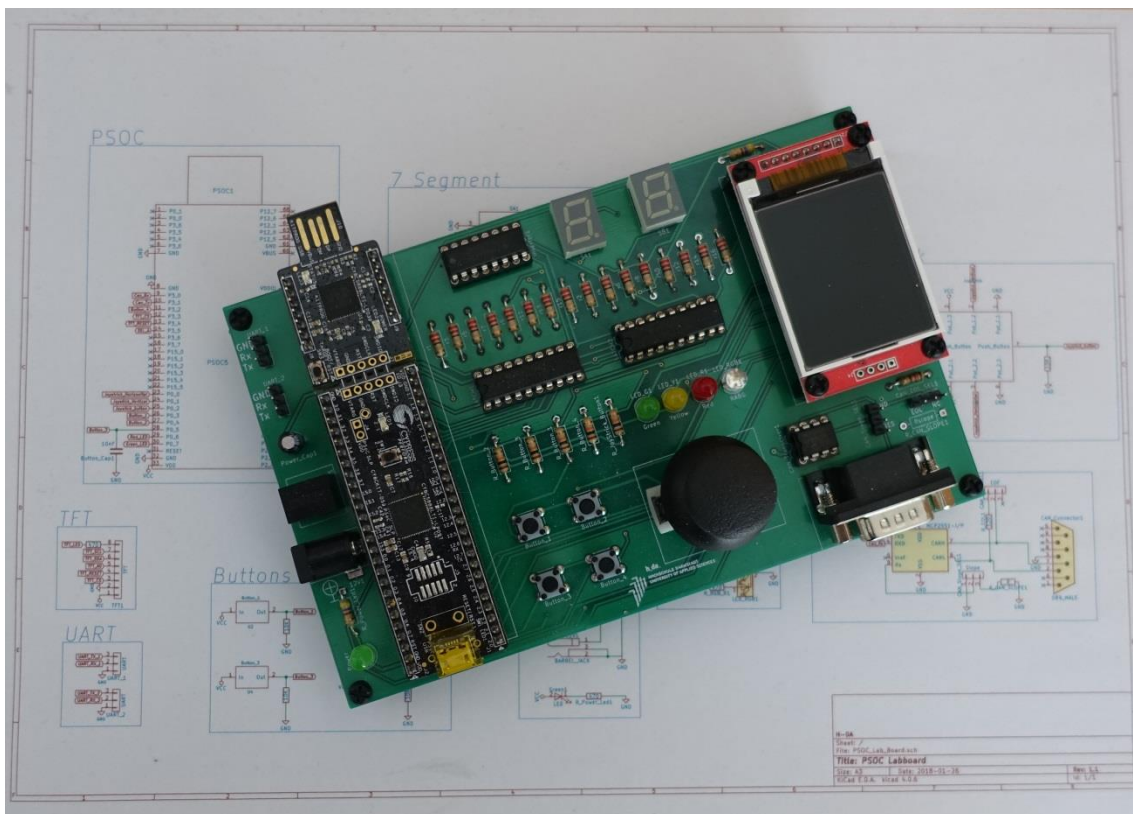


Figure 2 - The PSOC LabBoard

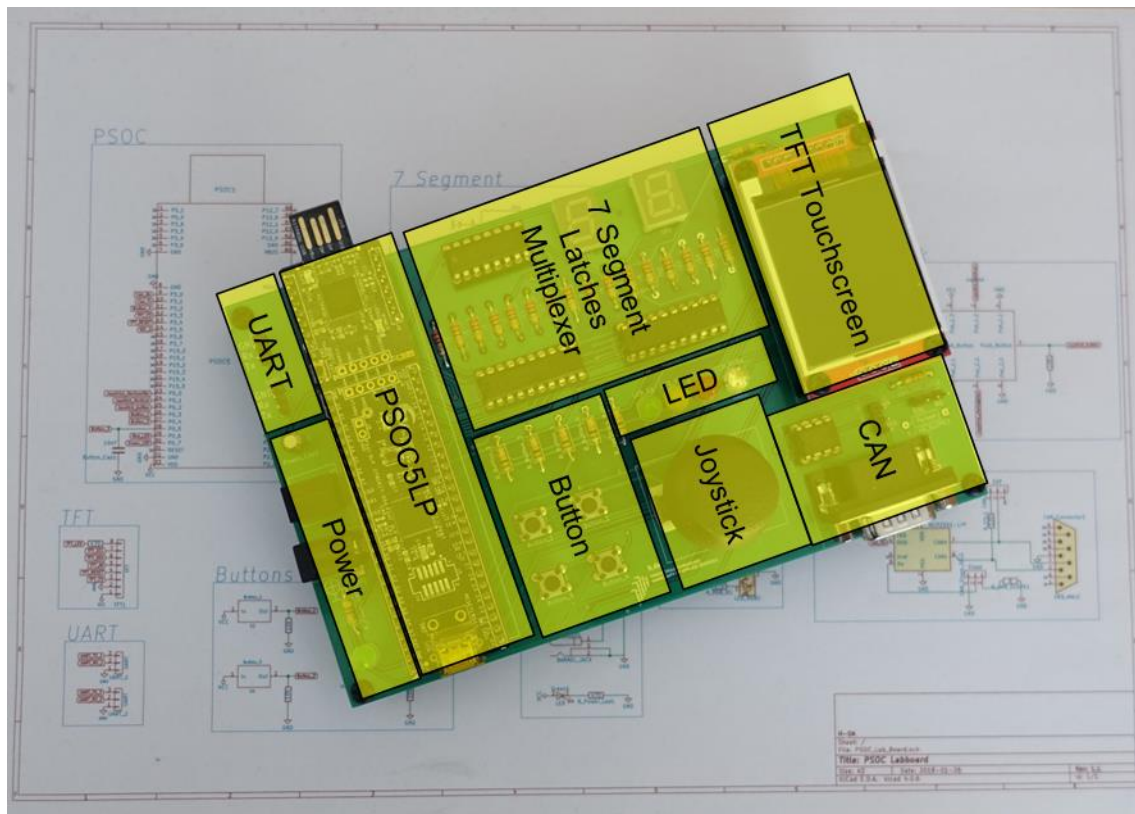


Figure 3- Location of the different functional groups

About PSOC 5

PSOC stands for “programmable system on chip”. Other than normal microcontrollers, which allow you to use pre-defined peripherals only, PSOC provides so-called Universal Building Blocks (UDB) which allows you to add and configure many peripheral ports or specific hardware functions you might need for the system.

Some PSoC® 5LP Highlights

- 32-bit Arm Cortex-M3 CPU, 32 interrupt inputs
- 24-channel direct memory access (DMA) controller with data transfer between both peripherals and memory
- 24-bit fixed-point digital filter processor (DFB)
- 20+ Universal Building Blocks and Precise Analog Peripherals
- Up to 62 CapSense® sensors with SmartSense™ Auto-tuning
- Multiplexed AFE with programmable Opamps, 12-bit SAR ADC and 8-bit DAC
- 736 segments LCD drive for custom displays
- Packages: 68-pin QFN, 99-pin WLCSP, 100-pin TQFP

Both the hardware design as well as the software development is done in the PSOC creator tool, which can be installed freely on your PC. The version used at the university can be downloaded from the following website: <https://web.eit.h-da.de/wiki/index.php/PSoC>

1 Creating a first embedded “Hello World” project

By implementing the following exercises, you will learn how to create hardware building blocks, how to connect the building blocks to physical pins and how to write a first application using the PSOC creator software.

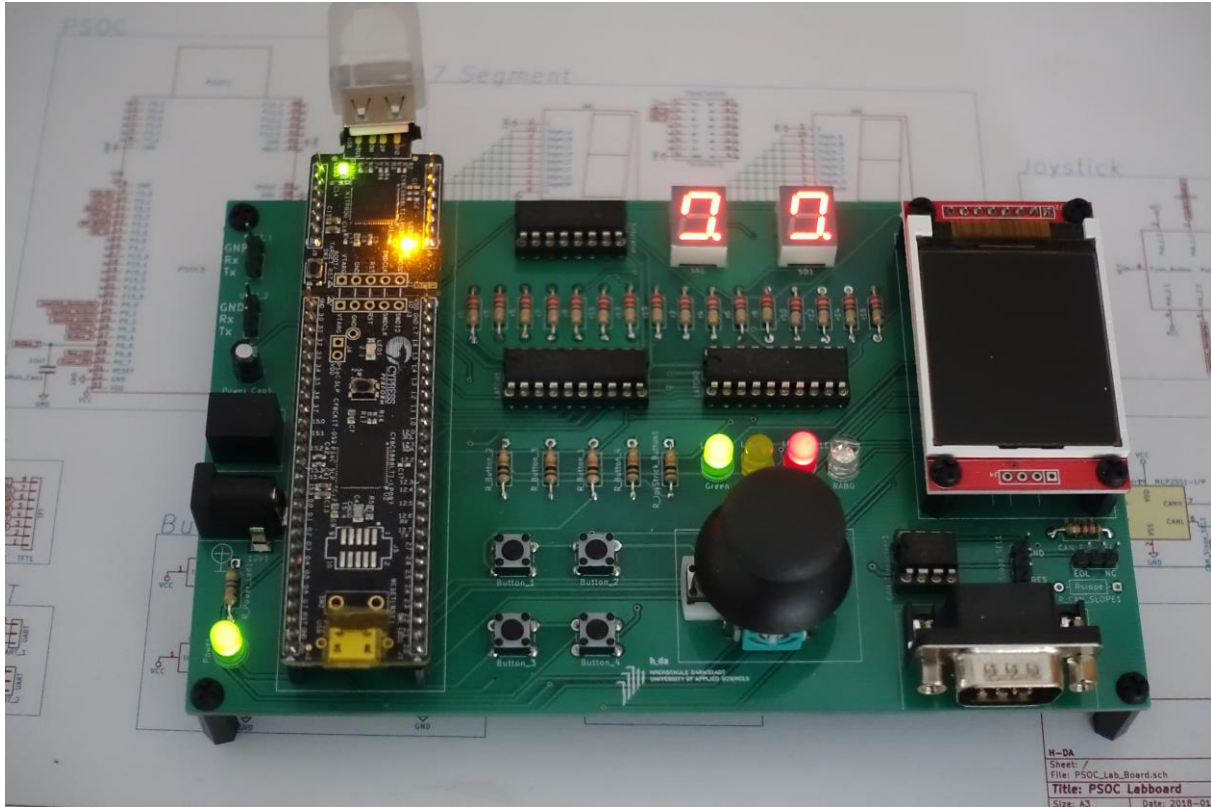


Figure 4 - The LEDs become alive

1.1 Let’s light the LED

Effort: 1h	Category - A
Exploring the PSOC5 Board and PSOC Creator IDE, placement and configuration of digital components	

Similar to Eclipse and other integrated development environments, we will use workspaces and projects to organize our code. In this context, a workspace is a folder which contains several projects. A project contains several source files which will be compiled and can be executed on the target as an elf-File. Unless we create a library project, but let’s start with the simple things first.

The first thing we have to create is a workspace. For this, we select *File | New | Project* from the main menu in the PSOC Creator IDE and tick *Workspace*. We provide a sensible name for the workspace and select the location, where the workspace will be created.

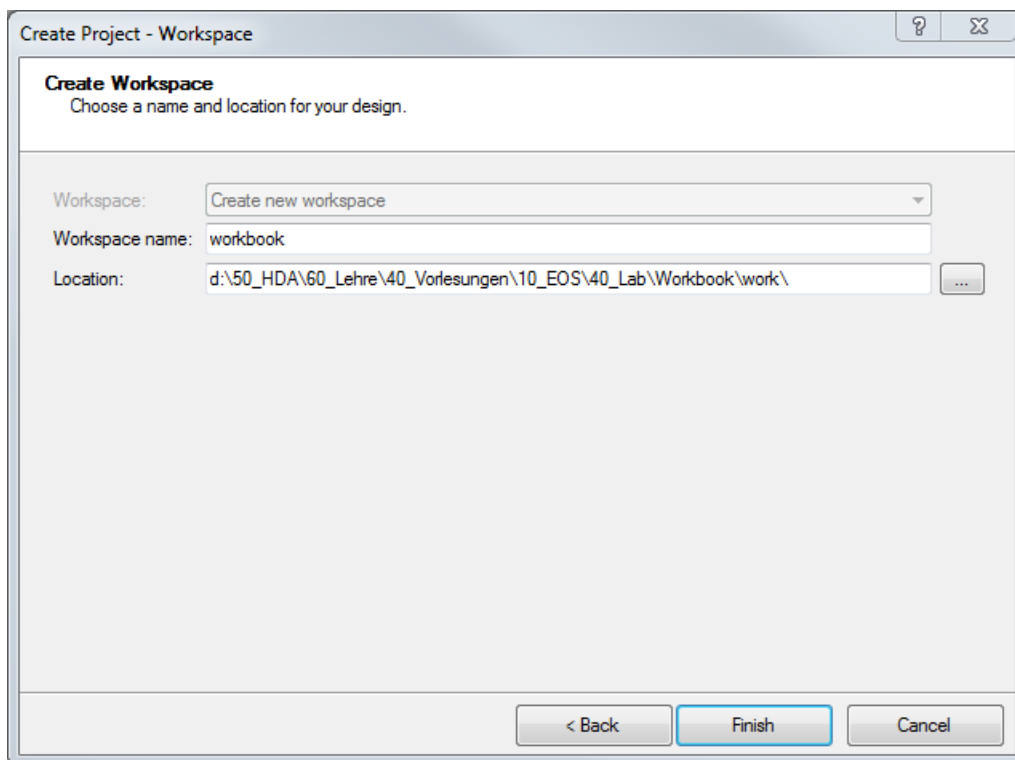


Figure 5 - Creating a Workspace

As a next step, a project will be created. Using *File | New | Project* from the main menu a project for the target device PSoC 5LP will be selected.

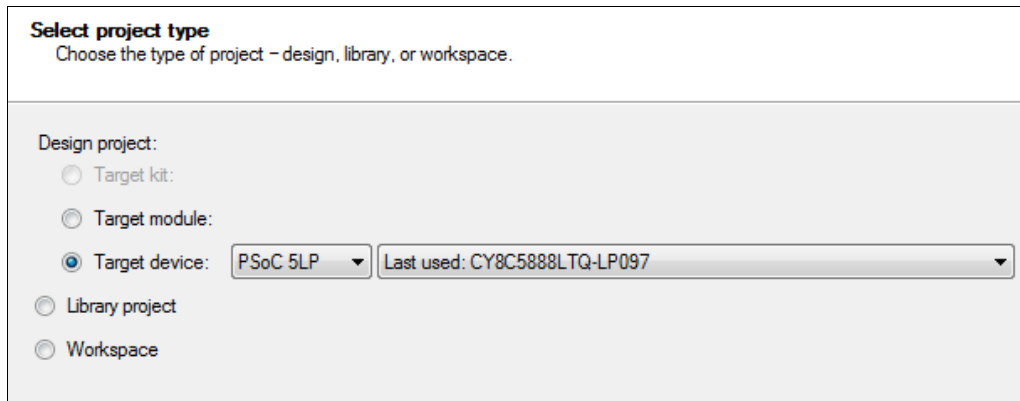


Figure 6 - Creating a project

In the next tab, select *Empty Schematic* and chose a self-explaining name. Your workspace should now look as follows:

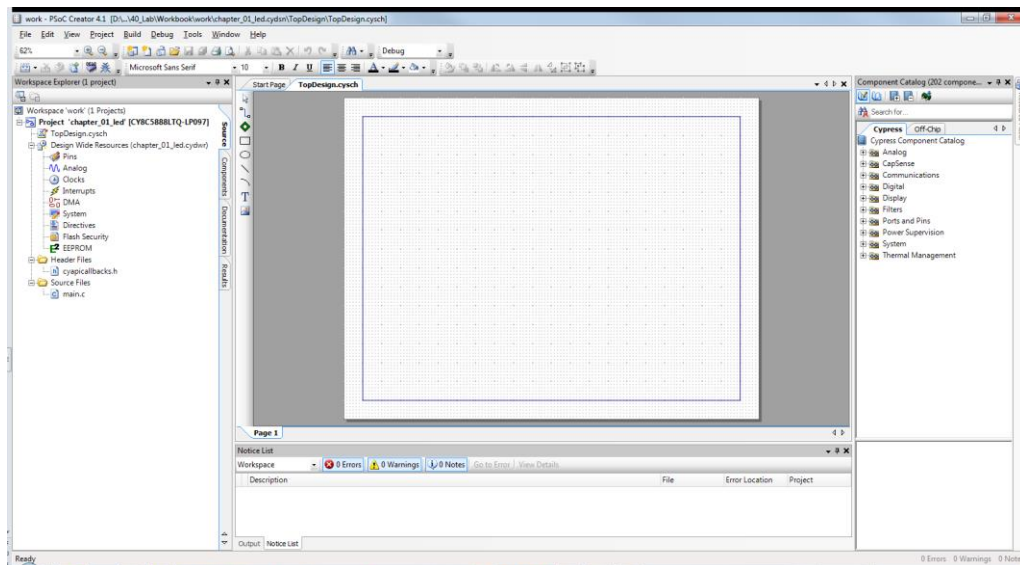


Figure 7 - Empty project

For our first project, we simply want to turn on the red LED. For this, we drag a Digital Output Pin from the component catalog on the right side to the top level design schematics.

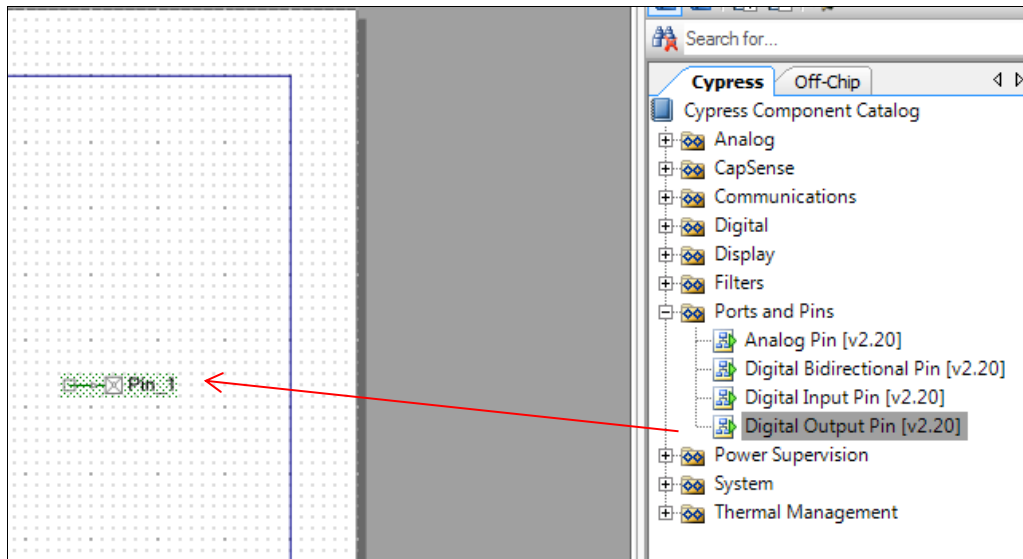


Figure 8 - Adding a first component

We add a Logical High ‘1’ signal following the same recipe. Try the search for field in the upper section of the Component Catalog. Using a connector, we connect both elements.

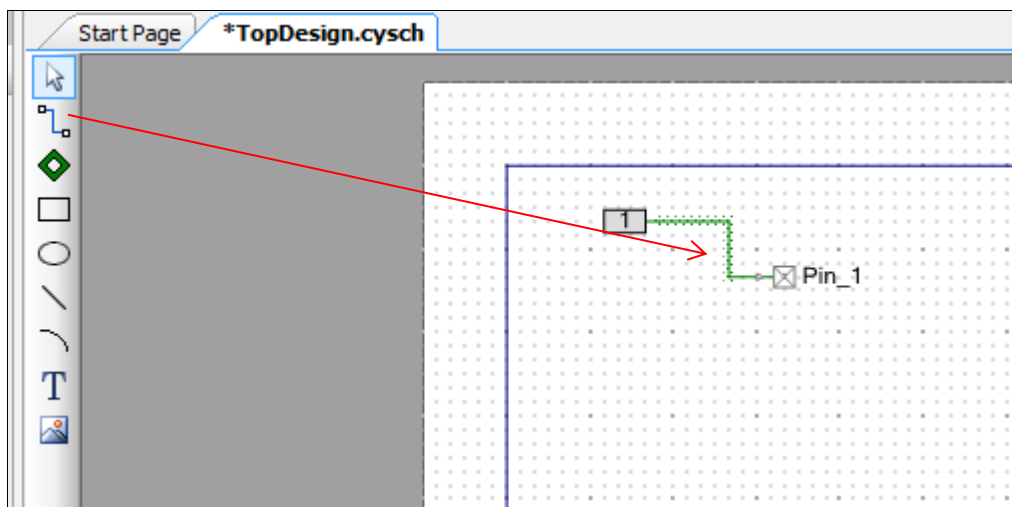


Figure 9 - Connecting two components

The symbol Pin_1 is a logical pin, which now needs to be connected to a physical pin of the PSOC. This is done by selecting the Pins Editor in the Design Wide Resources folder of the project space. BY checking the schematics, we can see that the red LED is connected to Port 0 Pin 6.

Please note that the pin numbers from the PSOC chip, the pin numbers of the breakout board, our schematic pins and the port id's have different values. We refer to the port id in this document.

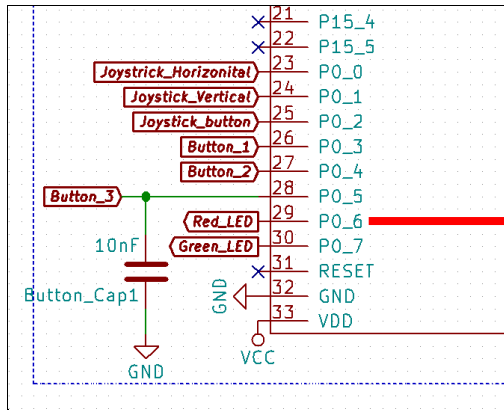


Figure 10 - Checking the schematics for the red LED

We select the correct physical pin in the drop down list.

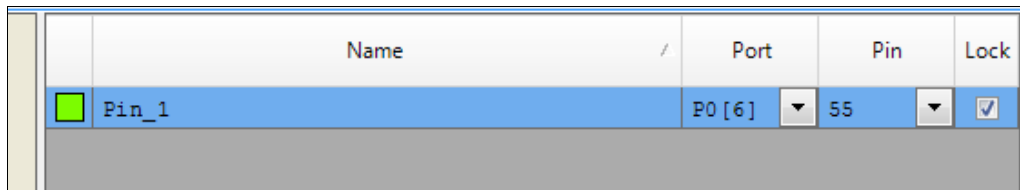


Figure 11 - Selecting the physical pin

Question:

Write down the pin numbers for, as we will need them a couple of times.

Led red _____

Led green _____

Led yellow _____

Button 3 _____

Furthermore we can verify that the LED is connected using a pull-down resistor, i.e. setting the pin to logical '1' should turn the LED on.

Before assigning a peripheral function to a pin, you must verify by checking the schematics if this is a good idea. An output functionality connected to a button might e.g. create a short circuit burning the PSOC. Capacitors might have a negative impact on the edges of digital signals etc.

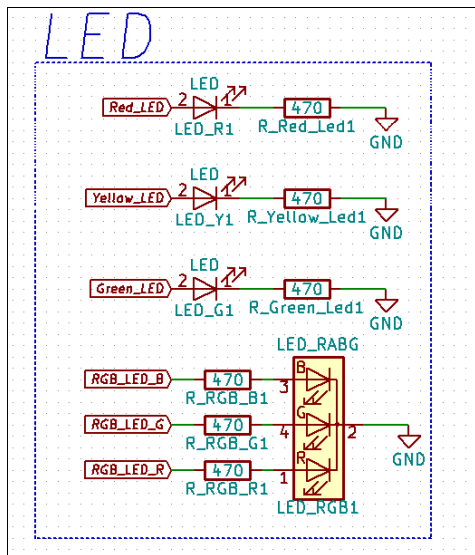


Figure 12 - Pull Down resistor of the LED

Question:

How do you calculate the value of the pull-down resistor used for the LED.

Compile and flash the program. The red LED now should be turned on. All other peripherals may have a random value.

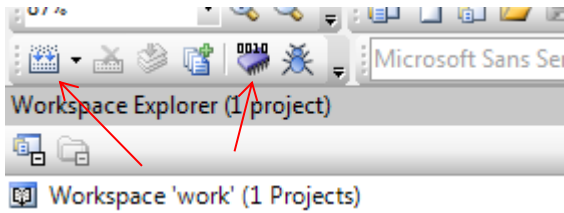


Figure 13 - Compile and Flash

To clean the project a bit up, we should change the name pin_1 to a more self-explaining name, e.g. led_red. For this, double click on the button in the schematic window. A configuration window will pop up, which allows you to configure the peripheral as needed.

Please also check the button “Datasheet”, which contains a manual for the selected peripheral including code snippets.

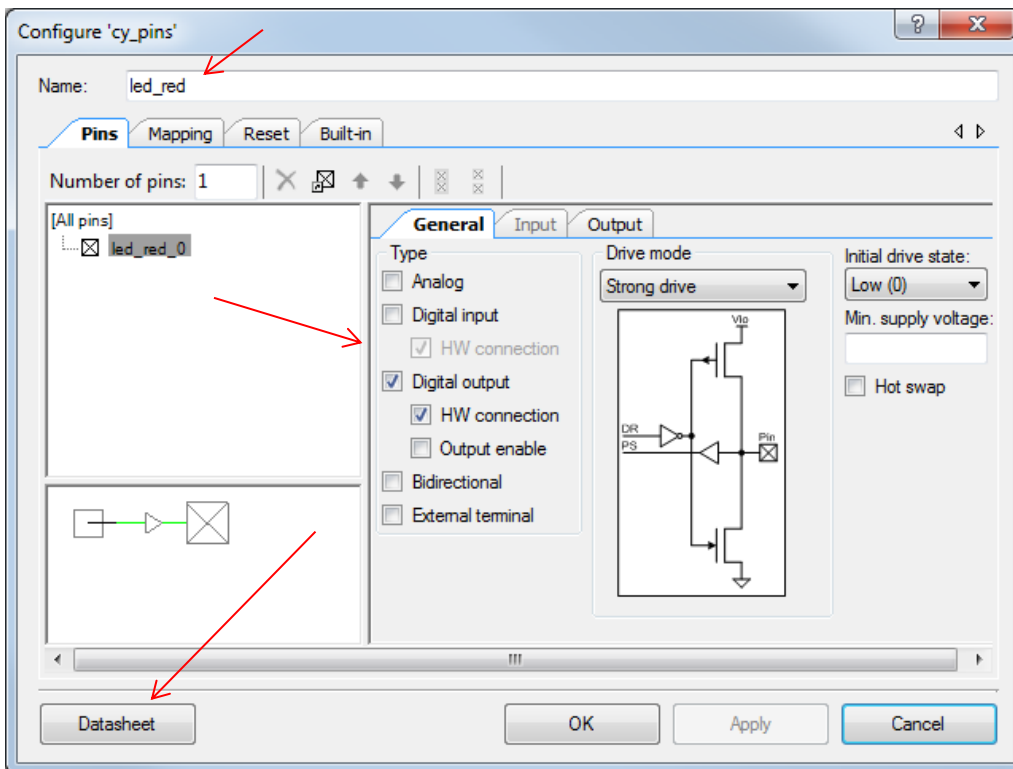


Figure 14 - Configuring a peripheral

1.2 Let's press a button

Effort: 2h	Category - A
Exploring the generated low level drivers, writing a simple C-function	

1.2.1 Pure hardware solution

In our second project, we want the LEDs to be a little bit more interactive. Your job is to implement a project which is fulfilling the following requirements

- If the button 3 (lower/left) is pressed, the green and red LED shall be on, the yellow LED shall be off
- If the button is not pressed, the LED status is inverted, i.e. yellow is on and the others are off

A very simple implementation can be done in hardware using a digital input as well as three output and a logical not element.

The schematics for this simple setup look as follows:

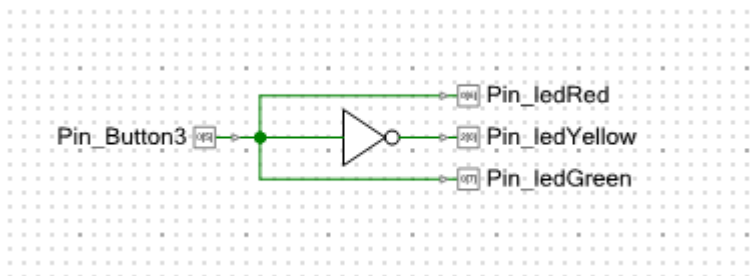


Figure 15 - Button / LED combination

The structure of this schematic can be divided into 3 blocks, which are forming a typical design pattern for embedded systems:

- Input (the Pin_Button3)
- Logic (the connections incl. the inverter)
- Output (the LED pins)

1.2.2 A first software control loop

To introduce a bit more flexibility, let's replace the logic block by a piece of software. To avoid conflicts, we first delete the connections and the Not-gate from the design. You might have noticed that while compiling the project, additional files have been generated in the folder Generated_Source/PSoC5. These files are elementary drivers for the hardware ports we have added to the design. Let's have a first look at them.

Every component has a couple of files, which are created. The file <module_name>.h is the main API file, e.g. Pin_Button3.h.

Please note that the files are created using the names you have provided for the components. Therefore it is recommended to choose sensible names right away from the start. The default names are no sensible names.

Let's have a look at the API of Pin_Button3.h

```
void    Pin_Button3_Write(uint8 value);
void    Pin_Button3_SetDriveMode(uint8 mode);
uint8   Pin_Button3_ReadDataReg(void);
uint8   Pin_Button3_Read(void);
void    Pin_Button3_SetInterruptMode(uint16 position, uint16 mode);
uint8   Pin_Button3_ClearInterrupt(void);
```

These functions can be grouped into 3 blocks, which you will find for most drivers:

- Initialization functions (e.g. SetDriveMode)
- Access functions (Read, Write)
- And Interrupt functions

As we have a very simple pin, there is no need for a specific initialization, the default settings are fine. More interesting is the Read function, which obviously returns the “press-status” of the button.

Looking at the LED drivers, we will find a similar structure.

In the folder Source Files, we find the file main.c where we can add our own code. The structure of main is a super-loop, which should be rather self-explaining by looking at the generated comments. The job of our first program is to

- Read the press-status from the button
- And to set the LEDs accordingly

The resulting code should look like this:

```
//This will include all drivers automatically
#include "project.h"

int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    for(;;)
    {
        // Let's read the button
        uint8 buttonStatus = Pin_Button3_Read();

        //And set the LEDs accordingly
        if (buttonStatus != 0)
        {
            Pin_ledGreen_Write(1);
            Pin_ledYellow_Write(0);
            Pin_ledRed_Write(1);
        }
        else
        {
            //missing code
        }
    } //end for
}
```

When compiling the project, you might get some compiler and/or fitter errors. I.e. the fitter might complain, that the terminal of the button and LED pins are not connected. As we only want to access the pin by software, we should remove this by unticking the field HW connection in the configuration of the pins.

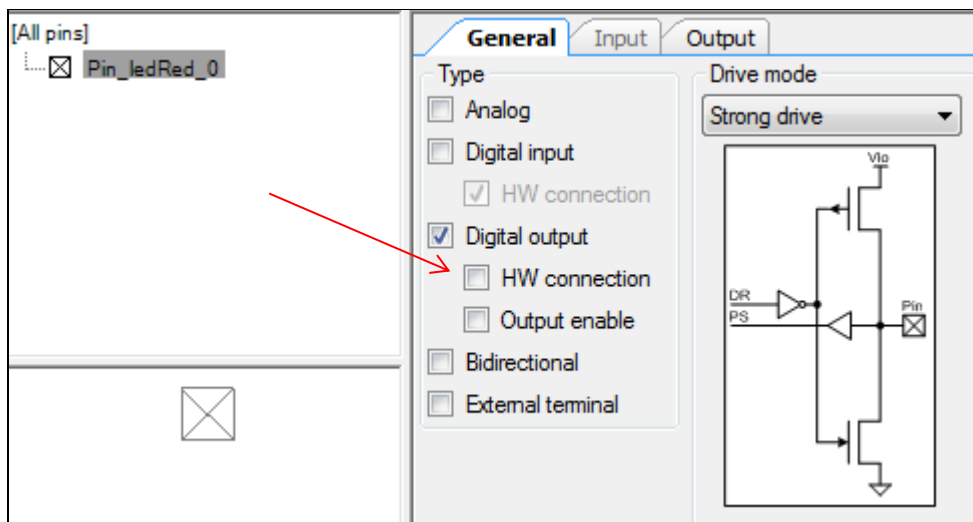


Figure 16 - Removing the hardware connection

1.2.3 Real logic - the toggling LEDs

To add a little bit challenge to the game, we want to toggle the LEDs whenever button 3 is pressed. Add the required functionality to the code.

Some cheat hints:

- You have to detect the point of time where the port value changes, i.e. you need to store the previous port state and compare it with the current one.
- You also need to store the LED state to toggle it
- How about creating an own function, using some static variables, which returns true in case the port toggle has been detected.
- When testing the program, you might notice that the LEDs will not always toggle as expected. Explain why.

1.3 Introducing interrupts

Effort: 2h	Category - A
Interrupt configuration, ISR programming, debugger	

1.3.1 A simple PIN interrupt

The disadvantage of the previous toggling solution is rather obvious. We are spending most of the valuable CPU time to check if the button has been pressed. A more elegant implementation is using an interrupt. An interrupt is a hardware signal, which is being processed by an interrupt controller. In case this signal occurs (e.g. a rising edge of an input line) the interrupt controller stops the normal program flow and calls a specific function, the interrupt service routine (also abbreviated as ISR or in some other documents called handler). Once this function has finished, the normal flow of operations will continue.

In this project (extension of the previous project), we will use an external interrupt (i.e. button 3) to toggle the LEDs. For this we have to perform the following steps (which are kind of a basic pattern to implement interrupts)

1. We must tell the pin of button 3 to create an interrupt once it is pressed
2. We then have to inform the interrupt controller what needs to be done (i.e. set the interrupt priority and address of the ISR)
3. Then we have to write the code for the ISR
4. And create a communication between the ISR and normal program execution

For configuring the pin, we open the pin configuration and select the tab “Input”. In this tab, we set the Interrupt to rising edge.

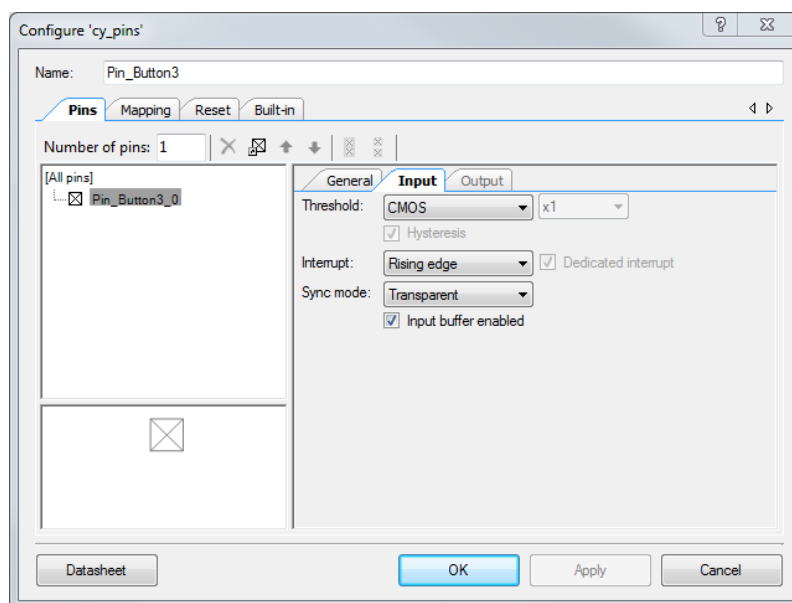
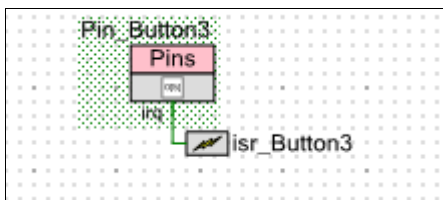


Figure 17 - Adding an interrupt functionality to a port

Question:

Why did we use a rising edge? What would be the effect of choosing falling edge?

As a next step, we have to connect an interrupt component to the interrupt port of the pin and give it a self-explaining name, i.e. `isr_Button3`.

**Figure 18 - Adding an ISR component**

After pressing compile, we will find a newly generated folder `isr_Button3`, which contains a corresponding header and source file.

Let's have a closer look at the generated start function:

```
void isr_Button3_Start(void)
{
    /* For all we know the interrupt is active. */
    isr_Button3_Disable();

    /* Set the ISR to point to the isr_Button3 Interrupt. */
    isr_Button3_SetVector(&isr_Button3_Interrupt);

    /* Set the priority. */
    isr_Button3_SetPriority((uint8)isr_Button3_INTC_PRIOR_NUMBER);

    /* Enable it. */
    isr_Button3_Enable();
}
```

To understand a little bit the background, we will investigate the line

```
isr_Button3_SetVector(&isr_Button3_Interrupt);
```

What is happening here? We write the address of the function which shall be called if an interrupt occurs (`isr_Button3_Interrupt`) into the interrupt vector table.

```
void isr_Button3_SetVector(cyisraddress address)
{
    cyisraddress * ramVectorTable;

    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    ramVectorTable[CYINT_IRQ_BASE +
(uint32)isr_Button3__INTC_NUMBER] = address;
}
```

To avoid confusion, we will introduce the following naming conventions

`isr_Button3` is called the interrupt object. It describes the entry in the vector table, i.e. the interrupt number (`isr_Button3__INTC_NUMBER`) and interrupt priority (`isr_Button3__INTC_PRIOR_NUMBER`). The function which is called by the interrupt is called interrupt service routine or interrupt handler. In this case this would be `isr_Button3_Interrupt`.

In other words, we have already finished step 2 of the previous Todo list.

Step 1 unfortunately is a little bit hidden in the generated code, but basically the selection of the interrupt source is configuring a comparator logic in one of the generated `cy...` files. Use Beyond Compare or any other compare tool to find the differences after changing the value.

The prototype for the ISR is generated in the file `isr_Button3.c`. However, as we want to separate own code and generated code, we are going to create an own ISR function and use the API function

```
void isr_Button3_StartEx(cyisraddress address);
```

to store the address of our own function.

By checking the generated code for the default ISR, you might notice that you can alternatively define a handler, which is set in the central file `cyapicallback.h`. Usually you have to provide two macros for a callback, one to enable the handler and the other one to call the intended handler.

In the file `main`, we now have to create a prototype for the ISR, an ISR implementation and some code (in this case a global variable) to transfer information from the ISR to the main program.

The toggling is still implemented in a busy waiting fashion. We could of course also add the LED setting code into the ISR, but this would make the ISR comparable big and also spoil the Input / Logic / Output pattern. Later on - when using an operating system, we will find more elegant patterns (e.g. events) to realize an efficient communication between ISRs and tasks.

By pressing the bug icon, you will start the debugger. Set a breakpoint in the ISR and see when it is called. Changed the pin interrupt logic to falling edge and compare the behavior.

Questions

Why do we have to declare toggle as volatile?

What happens if we forget to clear the interrupt source in the ISR?

```
volatile uint8 toggle = 0;

/**
 * Prototype of your own ISR
 **/
CY_ISR_PROTO(isr_Button3_RisingEdge_Interrupt);

int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    /* Place your initialization/startup code here (e.g.
    MyInst_Start()) */
    isr_Button3_StartEx(isr_Button3_RisingEdge_Interrupt);

    for(;;)
    {
        if (toggle == 0)
        {
            Pin_ledGreen_Write(1);
            Pin_ledYellow_Write(0);
            Pin_ledRed_Write(1);
        }
        else
        {
            Pin_ledGreen_Write(0);
            Pin_ledYellow_Write(1);
            Pin_ledRed_Write(0);
        }
    }
}

/**
 * This interrupt service routine will be called whenever a Button 3
 * rising edge interrupt occurs
 **/
CY_ISR(isr_Button3_RisingEdge_Interrupt)
{
    //Required to clear the interrupt source
    Pin_Button3_ClearInterrupt();

    if (toggle == 0)
    {
        toggle = 1;
    }
    else
    {
        toggle = 0;
    }
}
```

1.3.2 Interrupt names, priorities and service routines

When opening the interrupt window you can set the priority of the interrupt. The Interrupt Number, which defines the vector address in the vector table is created by the system.

The lower the number, the higher the priority.

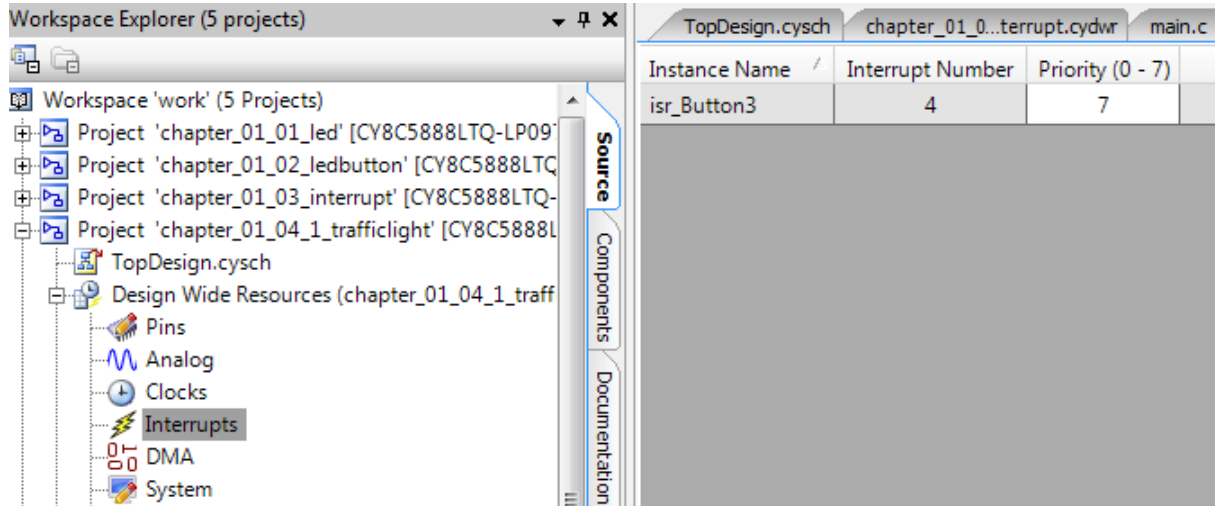


Figure 19 - Configuring the interrupt priority

1.3.3 Many different places to set interrupts

The creation of interrupts in PSoC creator can be a confusing task, as there are different places where this can be done. To illustrate this a bit more in detail, we will check the UART component. A UART component can create an RX interrupt (whenever a byte has been received) and a TX interrupt (whenever the next byte can be sent).

We will start by simply adding a UART component to our design and have a look at the parameters in the advanced tab.

You will notice, that the RX and TX interrupts are both disabled, although the RX interrupt “On Byte Received” (upper left area) has been activated.

Now change the buffer size to 5. Both interrupts are now enabled. The reason for this is the hardware buffer size of 4 bytes of this selected UART (different UARTs). As long as the required buffer from the user is 4 bytes or lower, only the hardware buffer will be used. In this case, no CPU interrupt is required. If a larger buffer is requested by the user, a software ringbuffer will be implemented by the generator.

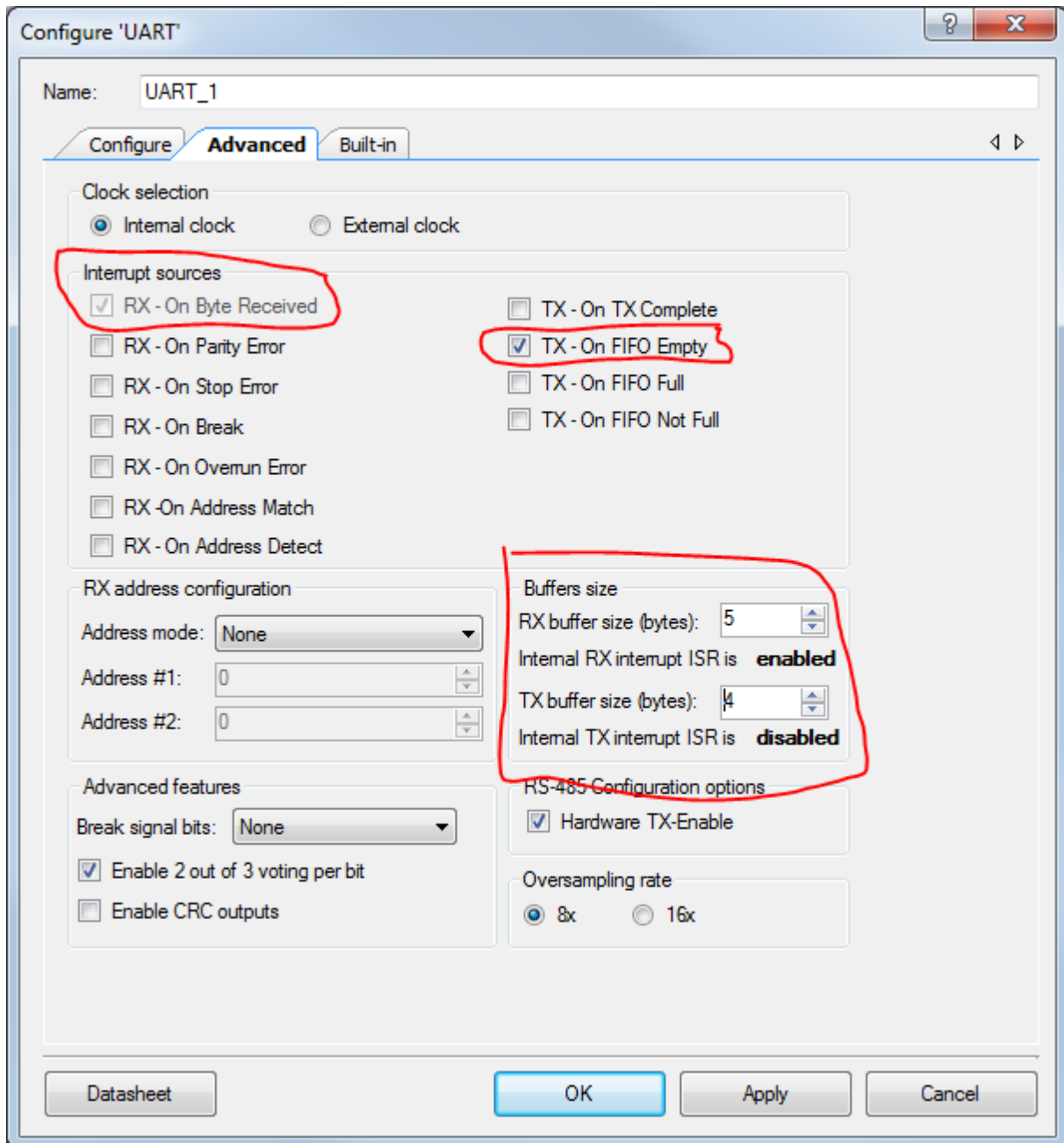


Figure 20 - Interrupt configuration of an UART component

After setting both buffers to a size larger than 4, we can see the interrupt entries in the interrupt table.

Instance Name	Interrupt Number	Priority (0 - 7)
isr_Button3	4	7
UART_1_RXInternalInterrupt	0	7
UART_1_TXInternalInterrupt	1	7

Figure 21 - Interrupt table containing internal interrupt

Other than with the pin interrupt we have created before, we do not need to create an interrupt service routine manually, as this already has been done by the generator. This generated ISR usually is implemented as a buffer. Check the file `UART_1_INT.c` for the implementation.

Please note, that the name of the generated interrupt service routine is `UART_1_RXISR` and differs from the Instance Name in the table. The instance name reflect the interrupt object, i.e. the vector number and priority, whereas the ISR name reflects the name of the software function which shall be called if this interrupt is fired.

The auto generated interrupt service routines are very nice and often contain more or less the functionality we require, however sometimes we want to implement our own logic instead.

There are 3 different options for this:

Option 1 - use callbacks

Callbacks can be used if you want to add some functionality to the existing code. The limitation however is, that you cannot modify the generated code as it is.

Option 2 - turn of the generator

You can copy the generated code into your source tree and then turn off the API generation. This gives you full control of the code, but you have to maintain the complete file manually from now on.

Option 3 - add own ISR

You may add an interrupt component to the RX and TX interrupt pins as shown below. Of course you should turn off the internally generated interrupts for this, i.e. set the buffer size to a value of 4 or smaller. If you do not do this, 2 interrupts will be created upon receiving and another 2 upon sending - probably not what you need.

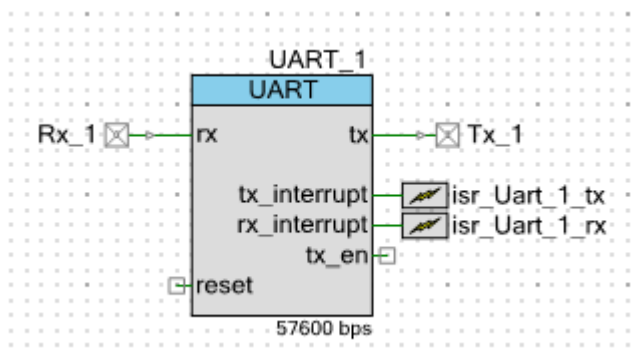


Figure 22 - Own ISR's for transmitting and receiving data

1.4 Light Effects

Effort: 2h	Category - A
Delay jobs, PWM signals, ISR's and superloops, function design	

Until now, the LED was simply turned on and off using a GPIO output. By introducing delays and extend the output to PWM signals, we can create some more advanced light effects.

1.4.1 Traffic Light

In our first example, we want to create a traffic light again using the three LED's.. You can copy the hardware schematics from the previous project to your workspace as a starting point.

The requirements are as follows

- By default, the color of the traffic light is red
- After pressing the button 3 once, the following sequence will be shown
 - For 2s: Red off, Yellow on, Green off
 - For 5s: Red off, Yellow off, Green on
 - For 2s: Red off, Yellow on, Green off
- Afterwards red on, the other off until the button is pressed again.

The code in the main superloop should look as follows.

```
for(;;)
{
    //Set initial value
    setTrafficLight(0, RED);

    //Wait for button pressed
    //Add your code for detecting a button press here,
    //using the interrupt concept from the previous exercise

    setTrafficLight(2, YELLOW);
    setTrafficLight(5, GREEN);
    setTrafficLight(2, YELLOW);
}
```

Implement the function for setting the traffic light according the following API specification.


```
/**
 * Set the corresponding LED for the selected duration
 * \param uint16 delay - delay time in seconds
 * \param ledColor_t color - the LED which shall be turned on, all
 * others off
 */
void setTrafficLight(uint16 delay, ledColor_t color)
```

For implementing the delay job, you may use the command `CyDelay`.

Which parameters are required?

Type: _____

Description: _____

Max delay time possible: _____

The command `CyDelay` is a blocking delay command, i.e. the controller will be in a busy waiting state while it is executed. Such commands are not really recommended for professional programs, as concurrent threads might be blocked in an unacceptable manner.

1.4.2 Fader

In this exercise, we want to create a fading traveling light. To control the brightness of an LED, we vary the power it consumes by connecting the output pin to a PWM signal, instead of a simple GPIO functionality. By varying the period value, we can change the brightness of the LED.

- Add 3 Output Pins to your schematic. If you copy them from a previous example, do not forget to activate the hardware connection again.
- Then, add 3 PWM blocks and set the number of PWM signals to 1
- Add a clock to the system and connect the clock signal with the PWM inputs

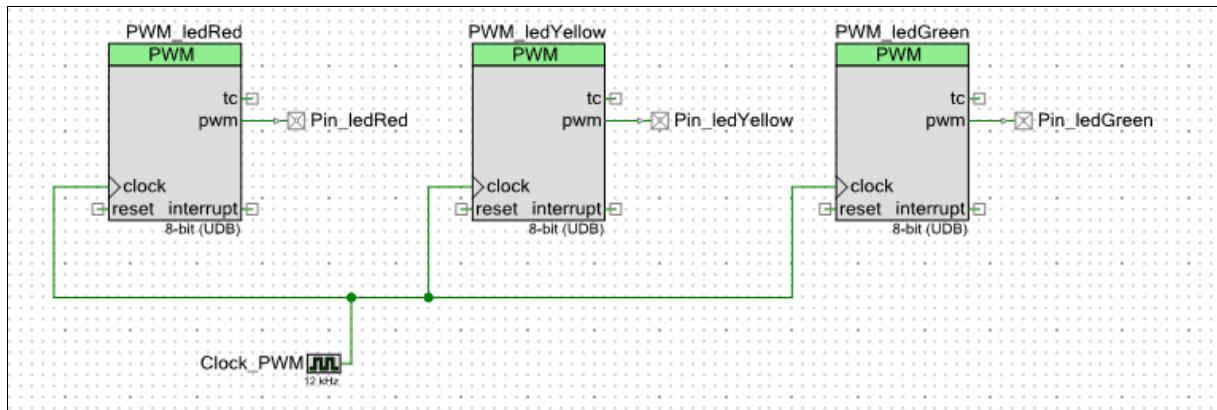
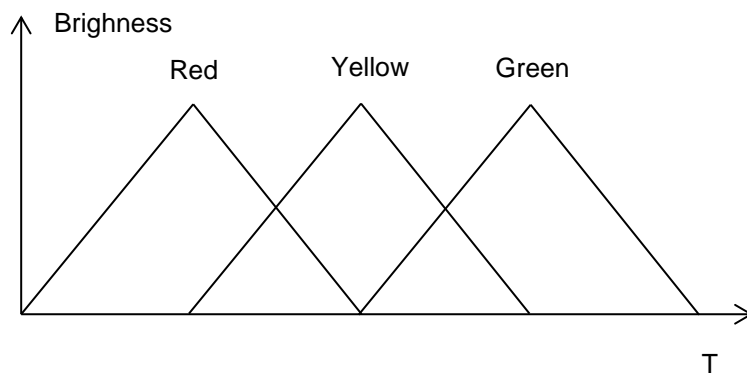


Figure 23 - Using PWM to control the LED brightness

Analyze the API of the generated PWM modules. Which functions do you need for this example?

Your job is to implement a fading function, which will fade the three LEDs using the following function:



1.5 Some simple console output

Effort: 1h	Category - A
UART, FTDI Bridge, Terminal Program	

For many applications it would be helpful to have some simple printf / scanf like console input and output available. Although the board has a display which could be used for this purpose, there is a much easier alternative.

When checking the pin function of the PSOC's header pins, (Annex 1 of this document) you might notice that some pins are connected to specific PSOC hardware. P2.1 is connected to the board LED, P2.2 is connected to the board button and P12.7 and P12.6 are connected to the UART_TX and UART_RX of the programmer's FTDI chip.

This allows us to use the debugger USB connection of the board for console input and output.

1.5.1 Iteration 1 - USART peripheral

The first thing we need to do is to create a USART peripheral and to connect this with the 2 pins.

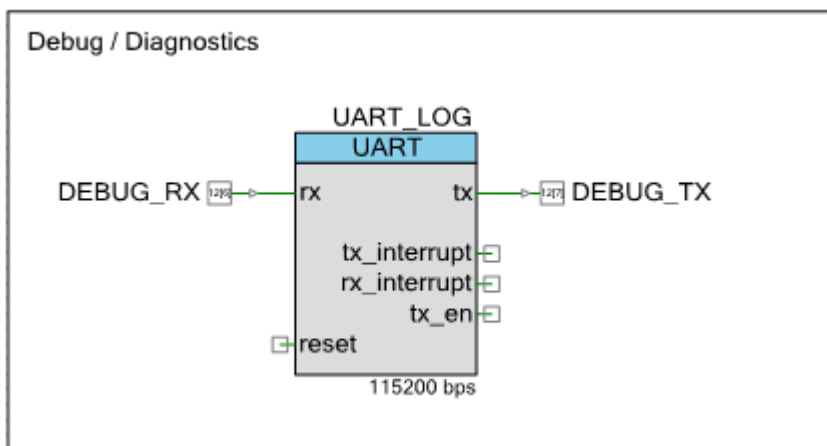


Figure 24 - USART Peripheral

The peripheral will be called UART_LOG and the 2 pins will be renamed to DEBUG_RX and DEBUG_TX. Furthermore we will set the baudrate to 115200bps. For the other settings, we will use the default parameters 8N1.

How many ASCII character can be transmitted via a USART having a transmission rate of 115200bps? Explain your answer!

Explain the impact of the parameters Data Bits, Parity Type, Stop Bits and Flow Control.

The next step is to connect the USART to the identified pins, which will transfer the signals to the FTDI bridge of the programmer.

	Name	Port	Pin	Lock
<input type="checkbox"/>	DEBUG_RX	P12 [6]	20	<input checked="" type="checkbox"/>
<input type="checkbox"/>	DEBUG_TX	P12 [7]	21	<input checked="" type="checkbox"/>

Figure 25 - Pins used for the FTDI bridge

That's more or less it. In the main program, we now can add some lines of code to print some console output.

```
UART_LOG_Start();
UART_LOG_PutString("Hello world\n");
```

1.5.2 Iteration 2 - Terminal Program

The data will now be send through the USB connection to the PC, but we still need an application to display it. There are a variety of console applications available, my preferred one is HTERM, which can be downloaded from

<http://www.der-hammer.info/terminal/>

To find out which USART port is valid, open the device manager. Please note, that the port will be different for every PC / board combination.

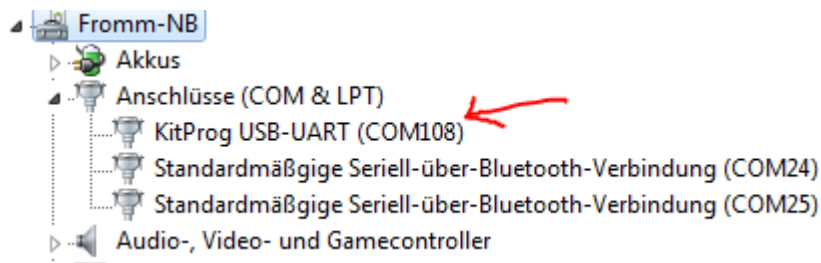


Figure 26 - Locating the correct UART port

Open the terminal program, select the correct COM port and you should be able to see your output (after resetting the microcontroller).

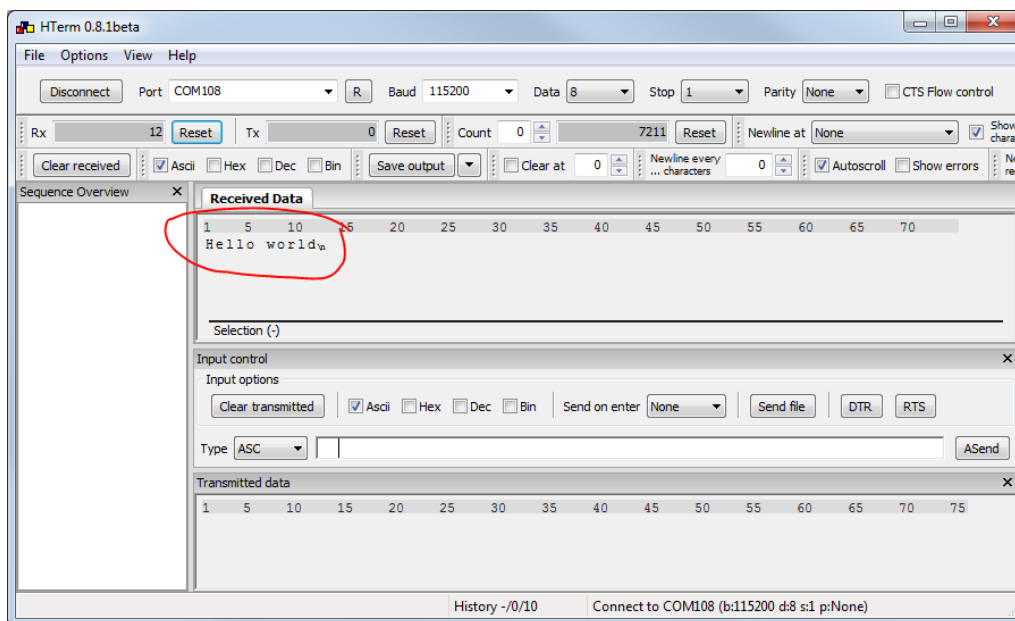


Figure 27 - Console Output on HTERM

1.5.3 Iteration 3 - Buffers

You might noticed that the RX and TX buffer by default are set to 4 bytes, which is the size of the hardware Fifo buffer of the USART. By increasing the number, an additional software Fifo (ringbuffer) will be added to the driver. Try it out!

1.6 A first timer

Effort: 2h	Category - B
Timer, Hardware Debugging	

For many application, you need time information, e.g. to measure the duration of an event or to wait for a certain period. In this example, we will use a clock, a counter and a global variable to implement a simple software counter.

1.6.1 Iteration 1 - Setting up the peripherals

We want to increment a global variable every 1ms. For this, we create the following setup

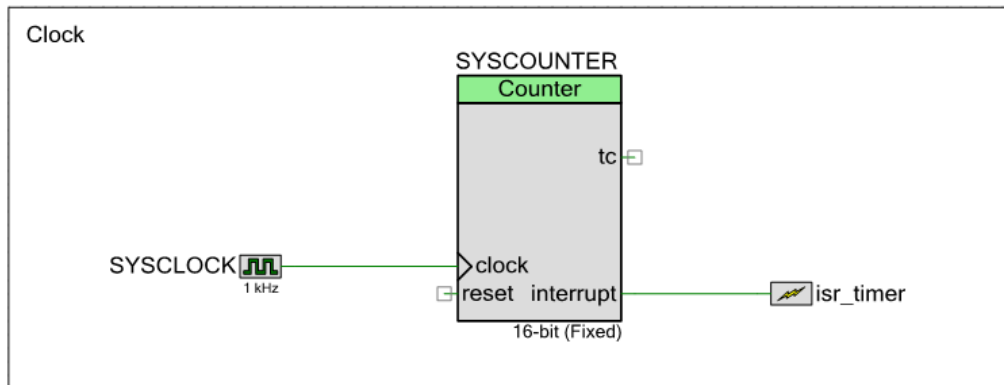


Figure 28 - Clock and Counter

The frequency of the clock will be set to 12MHz (half of the busclock, so only a very simple divider is required). Furthermore we will disable the “Start on Reset” of the clock in the clocks menu to have a better control of its functionality..

Type /	Name	Domain	Desired Frequency	Nominal Frequency	Accuracy (%)	Tolerance (%)	Divider	Start on Reset	
System	XTAL	DIGITAL	24 MHz	? MHz	±0	-	0	<input type="checkbox"/>	
System	XTAL 32kHz	DIGITAL	32.768 kHz	? MHz	±0	-	0	<input type="checkbox"/>	
System	Digital Signal	DIGITAL	? MHz	? MHz	±0	-	0	<input type="checkbox"/>	
System	USB_CLK	DIGITAL	48 MHz	? MHz	±0	-	1	<input type="checkbox"/>	IMOx2
System	ILO	DIGITAL	? MHz	1 kHz	-50, +100	-	0	<input checked="" type="checkbox"/>	
System	IMO	DIGITAL	3 MHz	3 MHz	±1	-	0	<input checked="" type="checkbox"/>	
System	PLL_OUT	DIGITAL	24 MHz	24 MHz	±1	-	0	<input checked="" type="checkbox"/>	IMO
System	MASTER_CLK	DIGITAL	? MHz	24 MHz	±1	-	1	<input checked="" type="checkbox"/>	PLL_OUT
System	BUS_CLK (CPU)	DIGITAL	? MHz	24 MHz	±1	-	1	<input checked="" type="checkbox"/>	MASTER_CLK
Local	UART_LOG_IntClock	DIGITAL	921.6 kHz	923.077 kHz	±1	±3.937	26	<input checked="" type="checkbox"/>	Auto: MASTER_CLK
Local	SYSCLOCK	DIGITAL	12 MHz	12 MHz	±1	±5	2	<input type="checkbox"/>	Auto: MASTER_CLK

Figure 29 - Clock configuration menu

For the counter, we select a period of 12.000. This should create the 1ms tick ($12\text{MHz} / 12000 = 1\text{kHz}$), activate the On TC interrupt in the advanced tab and. make sure that the mode is set to continuous.

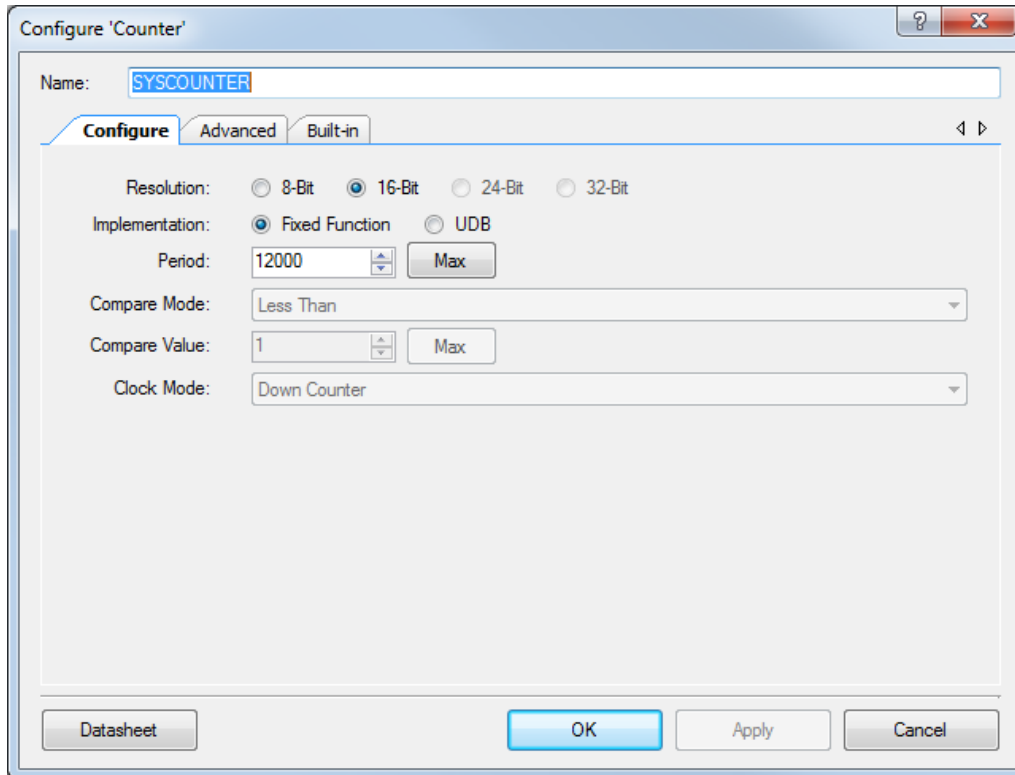


Figure 30 - Counter Configuration

We create a first test program, which will simply count up a global variable every 1ms


```

volatile unsigned int time = 0;

CY_ISR_PROTO(isr_timerTick);

int main(void)
{
    CyGlobalIntEnable; /* Enable global interrupts. */

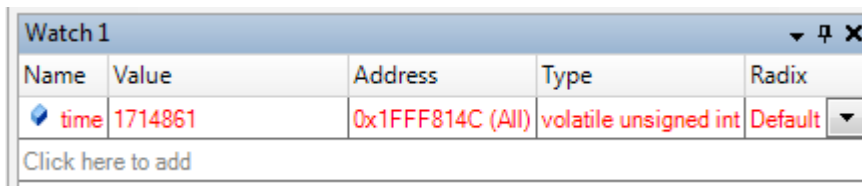
    /* Place your initialization/startup code here */
    isr_timer_StartEx(isr_timerTick);
    SYSCLOCK_Start();
    SYSCOUNTER_Start();

    for(;;)
    {
        /* Place your application code here. */
    }
}

CY_ISR(isr_timerTick)
{
    time++;
}

```

Start the program and after 1s pause it. By hovering over the variable `time` or by adding it to a watch list you can check the value - which should be somewhere in the range of 1000. Actually, the value is much higher.




Name	Value	Address	Type	Radix
 time	1714861	0x1FFF814C (All)	volatile unsigned int	Default
Click here to add				

Figure 31 - Wrong timer value

1.6.2 Iteration 2 - Hardware Debugging

Obviously we have to debug this problem. This however is not that simple.

- As we do not have a logical error, stepping through the code will not help
- As the issue is related to timing, any halt of the program will significantly modify the behavior - we are debugging a different system

Sounds like we have to develop a real debugging strategy.

The first step would be to slow down the time behavior so that we are able to see something. By changing the clock rate to 12kHz, the interrupt should come every second, instead of every millisecond.

The next simple thing we can do is to set a breakpoint in the ISR to get a feeling how fast this one is being called.

It seems that the first interrupt comes after 1s, but the next ones seem to come faster. We can doublecheck this by slowing the clock even further down or by disabling the

breakpoint and to have the program run for another 1s. Again you will notice that the value of `time` has increased a lot.

Conclusion so far: It seems that the first interrupt comes as expected, but the subsequent ones come too fast. To verify this conclusion, we try to make the behavior even more visible.

We could of course try to add a `UART_Log` command to the ISR, but this will typically not work, because the ISR will be fired much faster than the UART can transmit the data. We need a less intrinsic approach. For timing issues, a simple toggle GPIO port usually is a great help, because setting this port only requires very less time and we can use a LED or scope to visualize the port toggling.

By extending the ISR as follows

```
CY_ISR(isr_timerTick)
{
    time++;
    Pin_ledGreen_Write(time % 2);
}
```

we will see that the green LED is turned on after approx.. 1 s (as expected) but then stays on. If we connect an oscilloscope to the port pin of the LED, we see the following image:

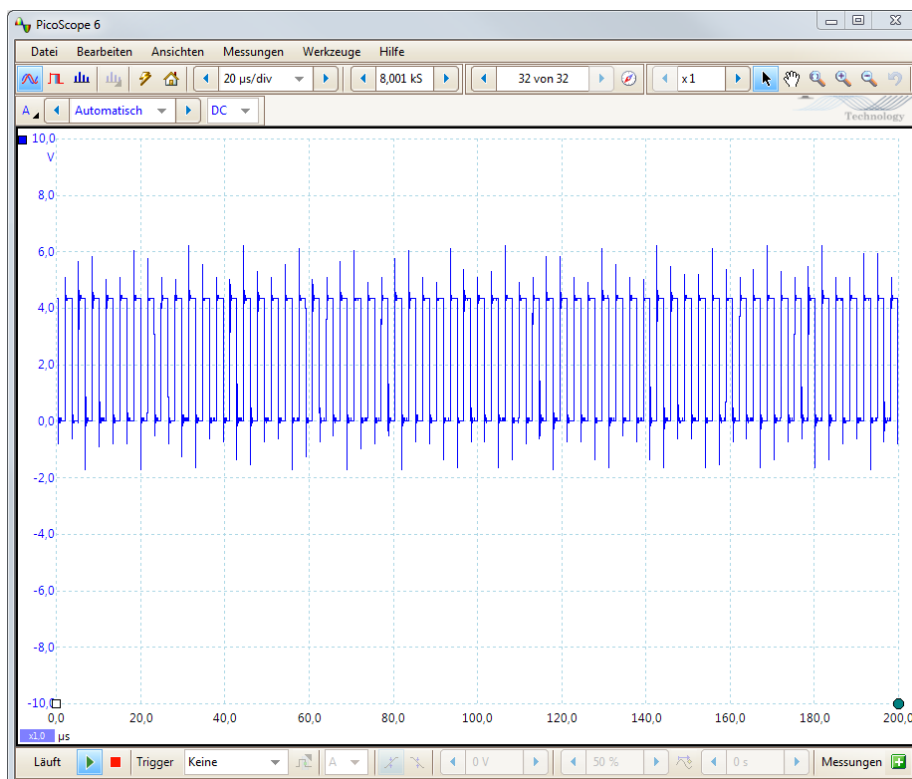


Figure 32 - Portpin LED in ISR

The LED is toggling with a frequency of around 0.3MHz, i.e. the ISR is called permanently. Let's explore this behavior a little bit deeper by connecting the other 2 LEDs to the interrupt and TC pin of the counter block.

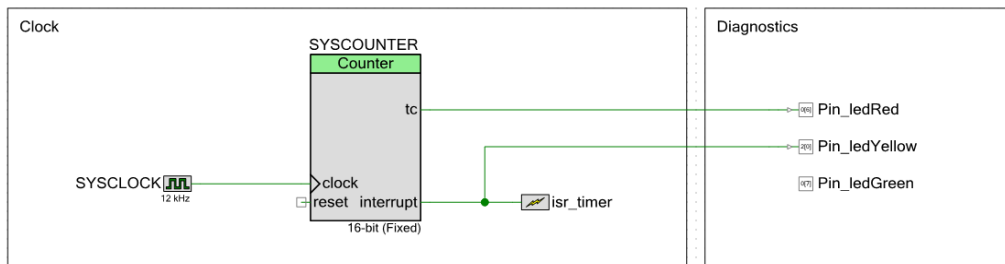


Figure 33 - More LED diagnostics

The yellow LED will remain turned on, but the red one is off. Time to read the datasheet. The datasheet says:

Period reload	Y (always reload on reset or TC)
---------------	----------------------------------

So actually we would expect the period to be reloaded automatically whenever the TC fires. But it seems that this function is not working as expected - a bug in the documentation. So, let's try to reset the counter explicitly once the terminal count (TC) is reached:

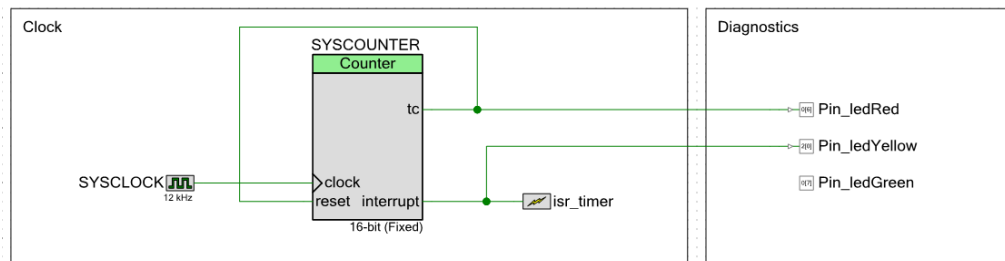


Figure 34 - Corrected counter functionality

Wow - the green LED blinks with 1Hz and the other 2 LED's are off - except some very short spikes, which are not visible to the human eye. ¹

1.6.3 Iteration 3 - Lessons learned

- SHOULD work does not mean DOES work.
- Especially low level hardware configuration and time related bugs cannot be found by “classic” debug strategies like using breakpoints and stepping through the code.

¹ This “buggy” behavior of the counter peripheral has been explored with v4.1 of the PSOC creator. It might be different with other tool version ;-(

- You have to add diagnostic code (and hardware) to understand what is happening.
- Documentation is helpful, but cannot always be relied on.
- Toggling ports, LED's and oscilloscopes are a great help to find transient bugs.
- Finding the root cause of a complex bug might be time consuming² but is a MUST to build a stable system.
- Good programmers look LEFT AND RIGHT when crossing a one way street.

1.6.4 Iteration 4 - Clean up

Now we have to restore the counter to the previous frequency and add some logic (e.g. UART output and update flags) to check the final functionality.

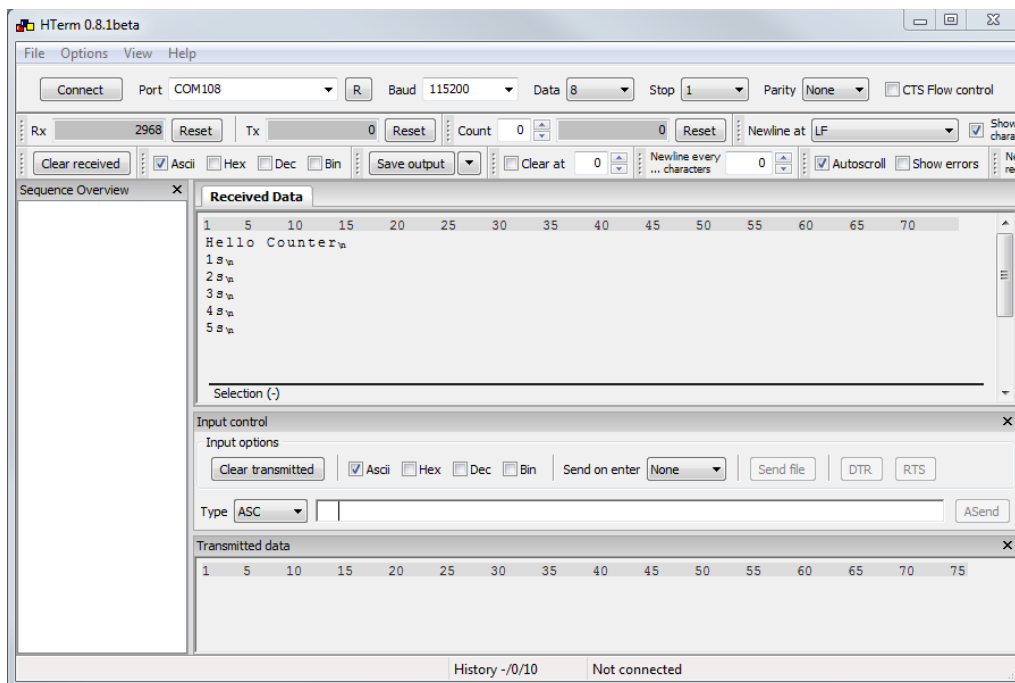


Figure 35 - The expected output

² When creating the exercise, it took me roughly 1 day to find the problem - most of the time being spent away from the computer thinking WHAT could be the issue and how can I check this.

2 Complex Device Drivers

In order to build more challenging application, we need to develop some drivers for the devices we can find on the board. For this, although we are programming in C and not C++, we will use an object oriented approach. I.e. every device we can identify will get a complex driver for intuitive use. These drivers will be reused in all the projects to come.

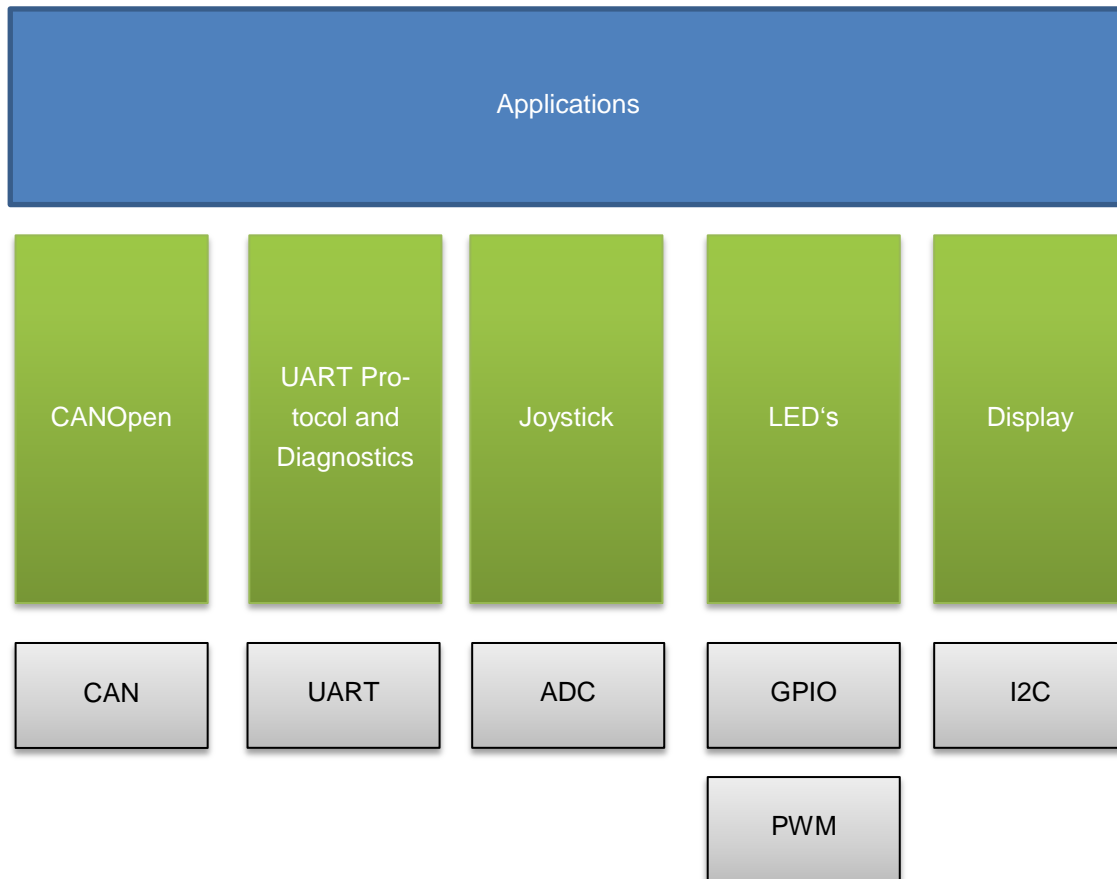


Figure 36 - Architecture

2.1 Joystick and RGB LED

Effort: 3h	Category - B
Project structure, ADC driver, complex driver, development process	

The projects we have developed so far mainly consisted out of generated drivers and a main file. This and also the next projects to come will be a bit more complex, we will have several files, which have to work together and which probably will be reused in other projects. Therefore the project setup as well as the design of the modules becomes more important.

2.1.1 Setting up the project structure

In this project, we want to use the joystick to change the color of the RGB-Led. For this, we will use the well known PWM hardware module as well as an ADC peripheral. Actually, we will require 2 ADC channels for the joystick (X and Y) and 3 PWM channels for the RGB LED, one per color.

The general structure of the design is shown in the picture below. We can clearly identify the layered design of the system. On the top level, the main file will contain the functional logic. Instead of accessing the generated drivers directly, an additional device or complex driver layer is introduced, which will implement the logic for the devices joystick and RGB LED. The green classes³ will be written by us, whereas the gray low level driver code will be generated by the system.

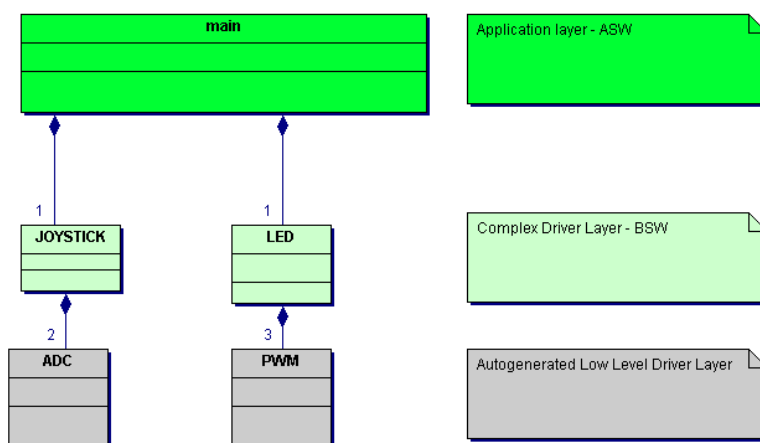


Figure 37 - Layered architecture

³ The term class is used to describe a functional unit, which can either be implemented as C++ class or C-file.

We can easily imagine, that the number of source files will increase with the complexity of the systems we are going to develop.

Other than Eclipse, the PSOC Creator IDE follows a logical structure concept, which means, that the structure visible in the project space can be different from the physical structure on the hard disc. This is shown in the picture below. Although the file `cyapi_callbacks.h` resides in a logical folder Header Files, this folder does not exist in our file system. Instead, the file is simply stored on the top level of the project directory. The same is true for the file `main.c` and any other file we are adding to the system through the IDE.

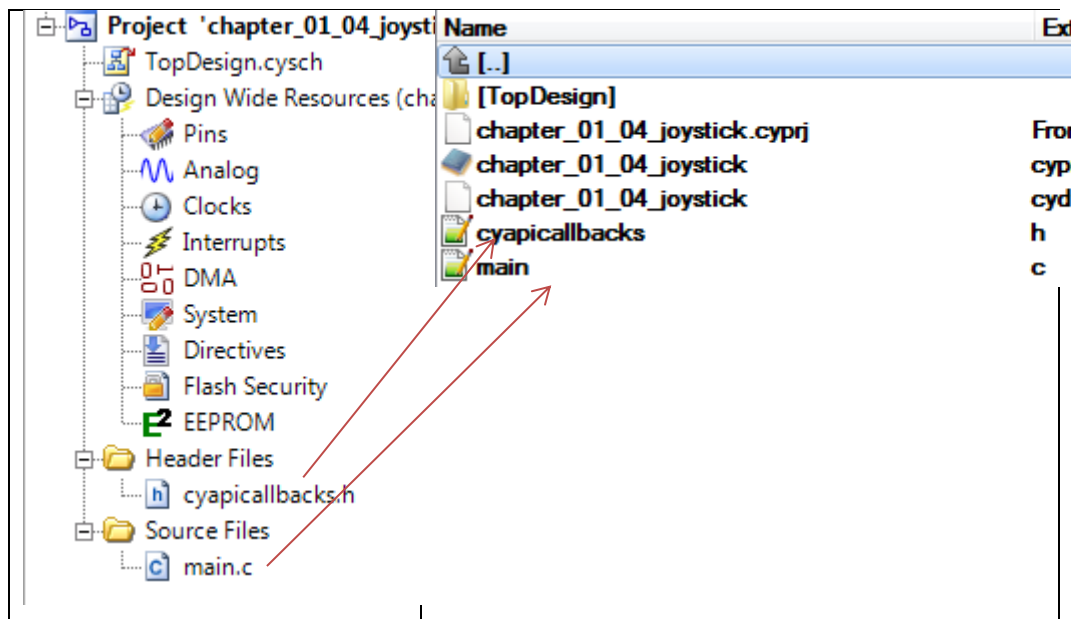


Figure 38 - Hacky default physical and logical structure of PSOC Creator

To avoid this “hacky” structure, we follow the procedure below for every new project.

1. We create a new project in our workspace
2. On file system level, we add a folder source to the project and in this folder source, we add 2 new folders asw (for application software) and bsw (for basic software, complex drivers)
3. Then we move the existing files `main.c` and `cyapi_callbacks.h` into the folder asw
4. We copy the template `codefile.h|c` from Moodle
5. And create a renamed copy wherever we need a file. In our example, we will create a `joystick.h|c` and `LED.h|c` in the folder bsw.
6. We switch to the PSOC creator GUI and remove the folders Header Files and Source Files and create 2 new folders called asw and bsw
7. In these folders, we add the existing items from the physical folders, resulting in the following cleaned up structures

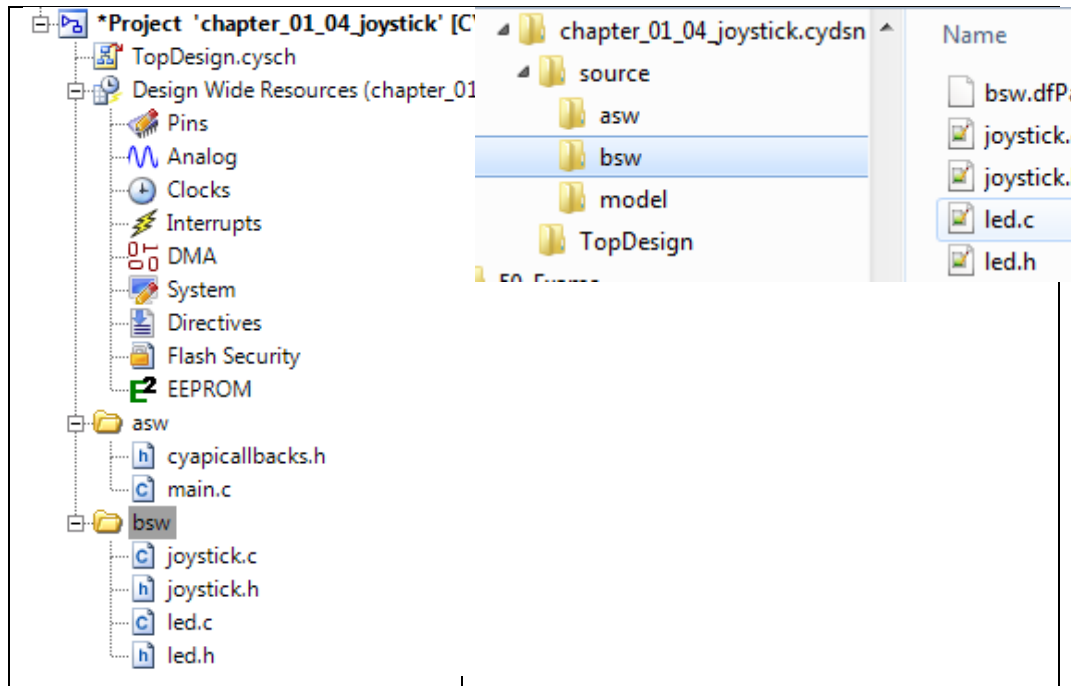


Figure 39 - Improved logical and physical structure

Before we can compile this empty project, we have to do 2 additional steps

- The code in the templates need to be cleaned up, i.e. the include guards and include directives must be corrected (plus of course function names etc.)
- We have to add the new directories to the include path of the IDE by opening the build settings (right click on the project)

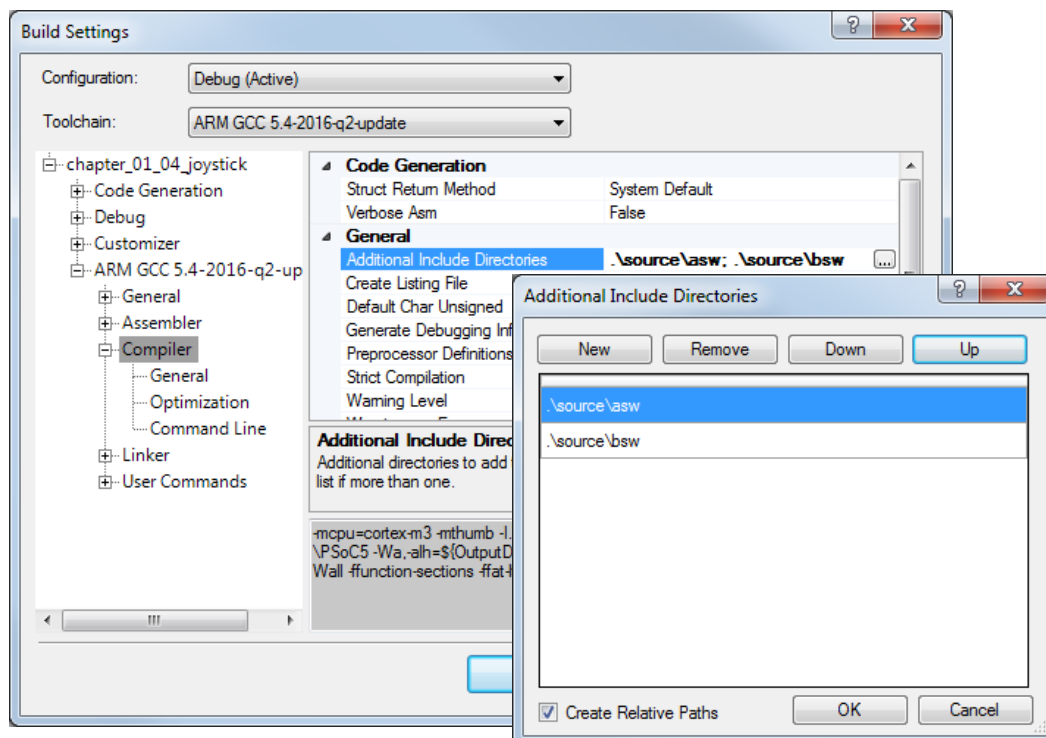


Figure 40 - Include path configuration

This structuring process at the beginning requires some work, but without a proper structure the development of more complex projects will end in a huge mess. Therefore you should get used to this step, no matter in which environment you are working! The process explained in this exercise step by step should also be followed in the exercises to come, even if it is not mentioned there explicitly!

You might have noticed that the template files simulate a C++ class when being opened with Together. This allows you to create reverse engineered class diagrams for design documents. Furthermore, the templates contain Javadoc / Doxygen⁴ style comments for autogenerated documentation.

2.1.2 Skinny sheep and interface design

Once we have set up the structure, we can start with the coding work. As the complexity grows, we must however develop a strategy before hacking. A recommended strategy is the so-called skinny sheep. In this approach, you try to implement a first very simple system, validate its functionality and then add more and more functions. The second principle we are following is to design good interfaces for our components.

Concerning the skinny sheep, we can either start with the implementation of the RGB LED or with the implementation of the joystick. As testing the joystick requires the RGB LED (unless we want to check the values in the debugger only), it makes sense to focus on the RGB LED in a first step. Instead of using the joystick signal to change the color, we can easily use hardcoded calls to test its functionality.

For designing the interfaces, we start by summarizing the requirements or use cases. What do we want to do with the module?

Functional requirements

Req-Id	Description
FR1	We want to turn the three standalone LED's on and off
FR2	We want to toggle the three standalone LED's
FR3	We want to set the color of the RGB LED
FR4	And we want to be able to initialize the component (default requirement for any driver)

⁴ Doxygen can be downloaded as freeware: <http://www.doxygen.nl/>

In addition, we have the following Non-functional requirements, which will impose constraints on our design.

Req-Id	Description
NFR1	To improve usability, we furthermore want to define some self- explaining enums to access the different LED's.
NFR2	Furthermore we want to have a uniform return type for any driver call, which will be represented by a common enum. For this, copy the files <code>derivate.h</code> and <code>global.h</code> from Moodle and add them to the bsw folder.

This leads to the following interface design. Very often, the functional requirements can be mapped to API functions almost one by one.

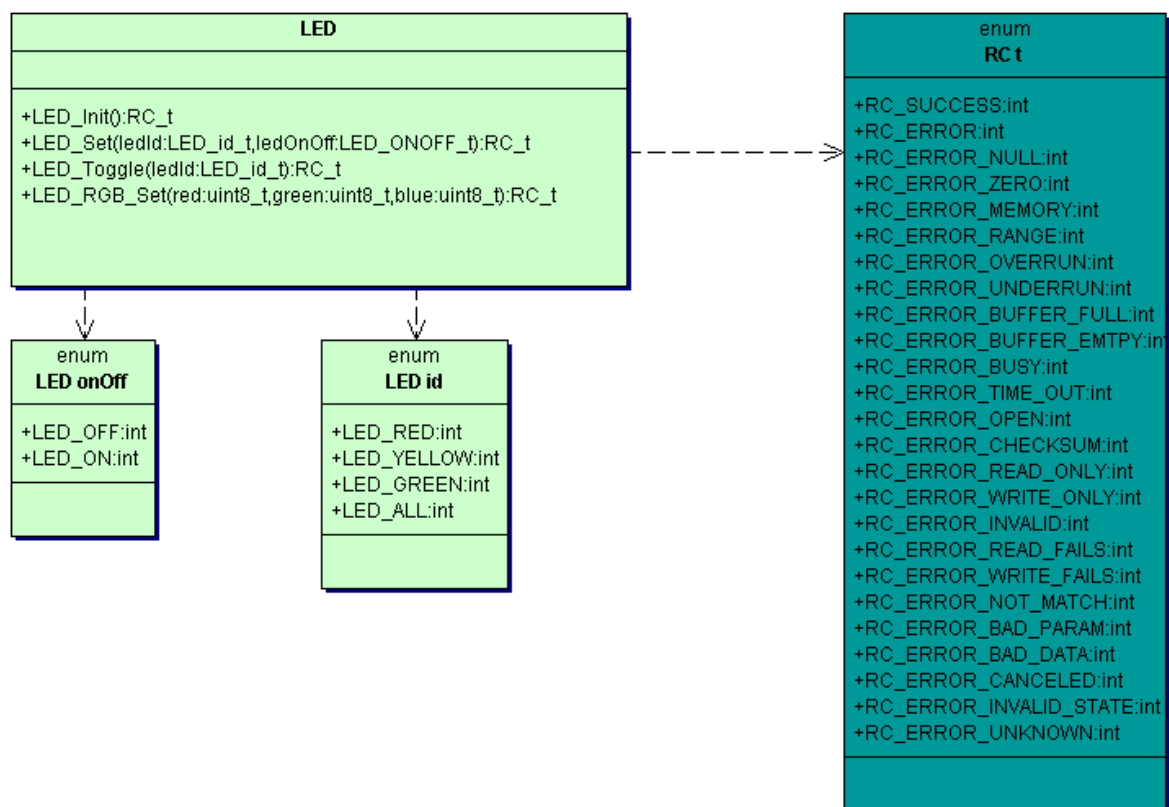


Figure 41 - Interface design for the LED class

Please note some important design patterns

- Instead of using the build in types like char, short and int, which may have different sizes on different platforms, we use the types `uint8_t`, `sint8_t`, `uint16_t` etc. instead, which are defined in the file `global.h`. We can be sure that a

uint8_t will always be mapped to an 8 bit unsigned datatype, which may be different per platform. I.e. we only have to switch to a new `global.h` when re-using the code on a different platform, but we do not have to change the application code.

- Every globally visible identifier starts with the prefix `LED_` to build a C++ like namespace for the module.

2.1.3 Implementation of the Skinny Sheep (LED driver)

In the next step, implement the required peripherals.

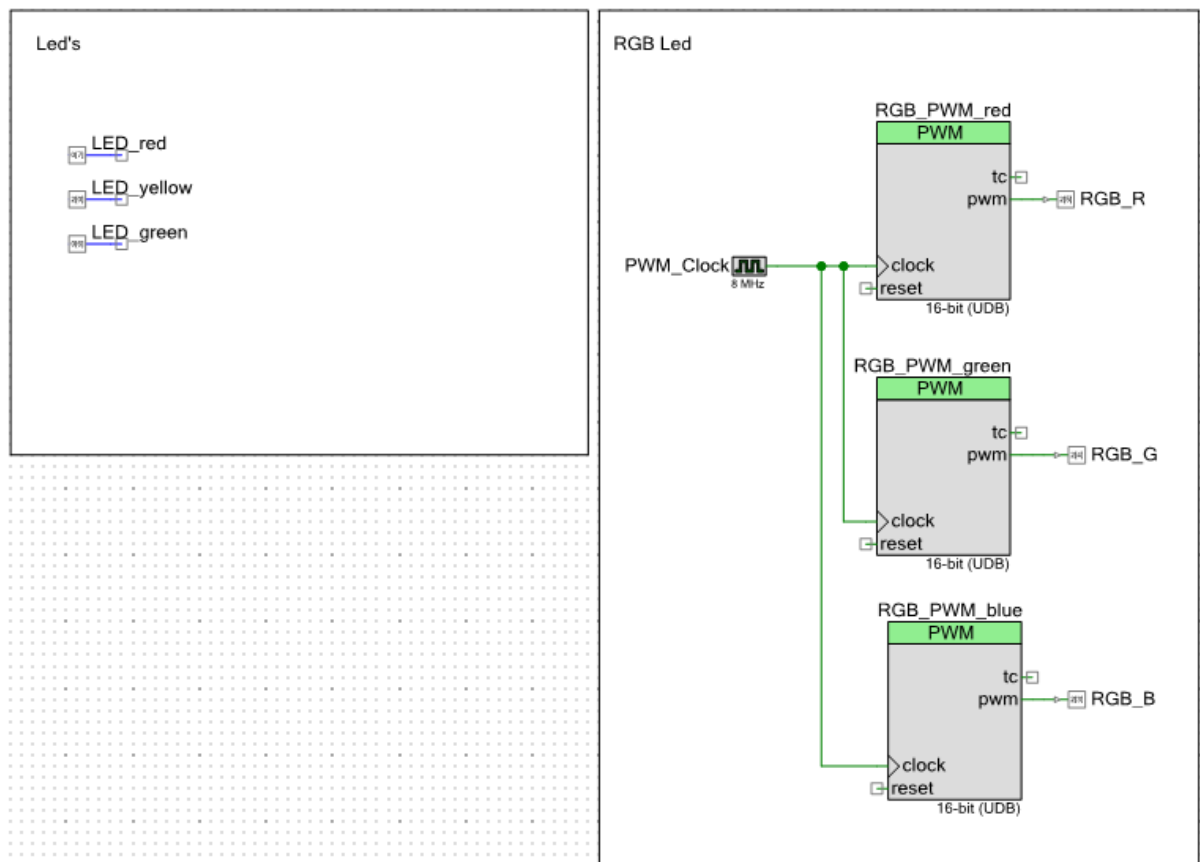


Figure 42 - Peripherals for the LED's

And connect them to the ports as given in the schematics. Now we can start to implement the designed functions.

Iteration 1

`LED_init()`

- Call the initialization functions of the PWM Signals
- Turn off all LED's (consider using the function `LED_set` for this)

`LED_set()`

- Set the LED as defined by the enum to on or off
- If LED_ALL is selected, all LED's will be turned on or off

Write a small testcase to test the implemented very skinny sheep.

Iteration 2

In the next iteration, we will add the toggling logic. Basically, a toggling LED can be described as a very simple state machine:

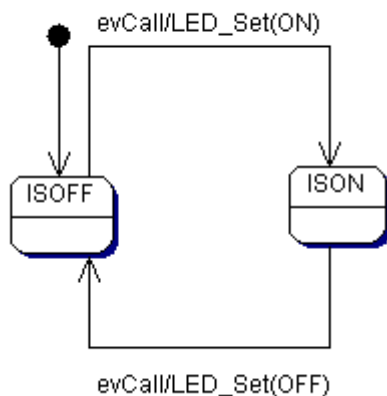


Figure 43 - Toggle logic described as state machine

This implies, that we need to provide an additional state variable per LED. As this state variable will only be modified by the LED functions, we should declare it as a file static variable, e.g. as an array. The size of the array depends on the number of LED's we have. In our example we have 3 LED's. But maybe we want to use the driver in a future system, which might have a different number of LED. Therefore we should try to find a design pattern which allows us to maintain the number of LED's at a single spot in code. As we already have the enum to provide the id for the LED's we can use the following design trick:

```

typedef enum {
    LED_RED,           /**< Selection of red LED */
    LED_YELLOW,        /**< Selection of yellow LED */
    LED_GREEN,         /**< Selection of green LED */
    LED_ALL            /**< Selection of all LED's */
} LED_id_t;
  
```

In this example, we have added an additional enum value LED_ALL at the end of the list. LED_RED will have the value 0, LED_YELLOW 1 and finally LED_ALL will be equal to 3. The size of the array we need! As long as we add additional LED_XYZ before LED_ALL, LED_ALL will contain the number of LED's we have in the system.

As a consequence, we can define the state variables for the LED's as follows:

```

static LED_ONOFF_t LED__state[LED_ALL] = {LED_OFF, LED_OFF,
LED_OFF};
  
```

Using this file static variable, we can implement the function `LED_Toggle`. Don't forget to initialize this variable in the `init`-function and to update the value in the `LED_set()` function.

Write another testcase to test the new function.

The usage of such a state variable is also called shadow register, as we are storing the port state in an additional variable. Instead of providing this extra file static variable, we could also read the GPIO port of the LED and invert it. However, not all ports do allow such a read operation. Furthermore, this structure allows us a pretty generic implementation which will work for different hardware solutions.

Iteration 3

Now it's time to have a closer look at the RGB LED. The resulting color is set by setting different intensities, i.e. PWM ratios, for the individual (R)ed, (G)reen and (B)lue LED.

You might have noticed that the PWM registers have been configured as 16bit values, whereas the function `LED_RGB_set()` takes three 8 bit parameters? Bad design? The resolution of the PWM peripheral can also be set to 8 Bit, however the larger value range of the output allows us to create a calibration curve, which improves the ambience of the glow function.

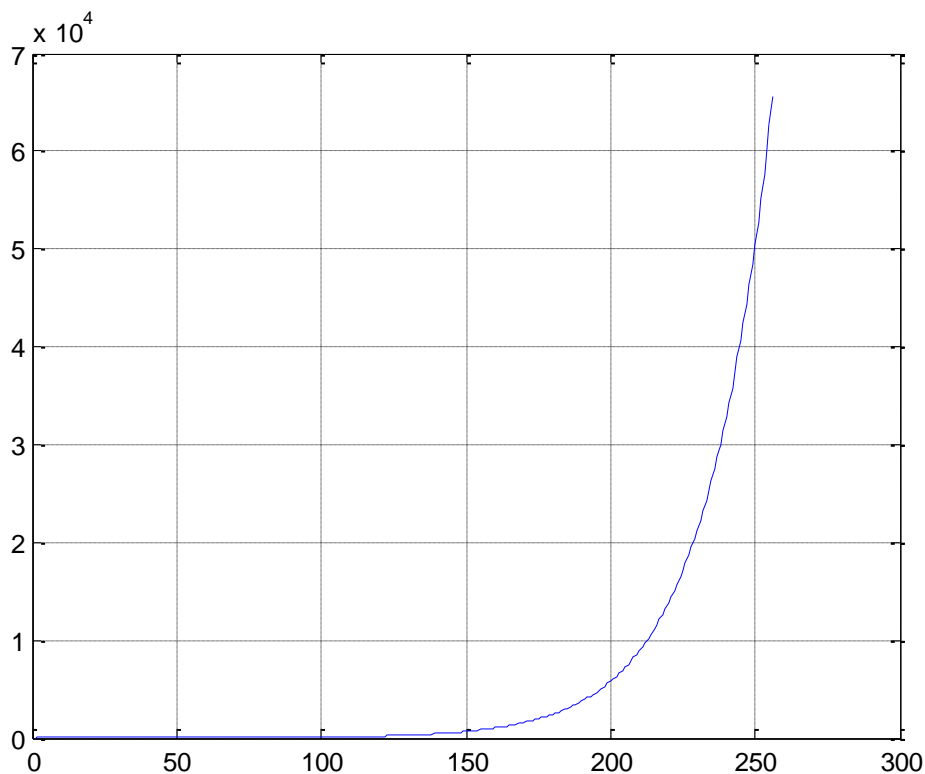


Figure 44 - Calibration curve for LED brightness

The easiest way to implement such a curve is a const lookup table.

```
const static uint16 LED_Pulse_Width[256] = {
    0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3,
    3, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5, 6, 6, 6, 6, 6, 7,
    7, 7, 8, 8, 8, 8, 9, 9, 10, 10, 10, 11, 11, 12, 12, 13, 13, 14, 15,
    15, 16, 17, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
    31, 32, 33, 35, 36, 38, 40, 41, 43, 45, 47, 49, 52, 54, 56, 59,
    61, 64, 67, 70, 73, 76, 79, 83, 87, 91, 95, 99, 103, 108, 112,
    117, 123, 128, 134, 140, 146, 152, 159, 166, 173, 181, 189, 197,
    206, 215, 225, 235, 245, 256, 267, 279, 292, 304, 318, 332, 347,
    362, 378, 395, 412, 431, 450, 470, 490, 512, 535, 558, 583, 609,
    636, 664, 693, 724, 756, 790, 825, 861, 899, 939, 981, 1024, 1069,
    1117, 1166, 1218, 1272, 1328, 1387, 1448, 1512, 1579, 1649, 1722,
    1798, 1878, 1961, 2048, 2139, 2233, 2332, 2435, 2543, 2656, 2773,
    2896, 3025, 3158, 3298, 3444, 3597, 3756, 3922, 4096, 4277, 4467,
    4664, 4871, 5087, 5312, 5547, 5793, 6049, 6317, 6596, 6889, 7194,
    7512, 7845, 8192, 8555, 8933, 9329, 9742, 10173, 10624, 11094,
    11585, 12098, 12634, 13193, 13777, 14387, 15024, 15689, 16384,
    17109, 17867, 18658, 19484, 20346, 21247, 22188, 23170, 24196,
    25267, 26386, 27554, 28774, 30048, 31378, 32768, 34218, 35733,
    37315, 38967, 40693, 42494, 44376, 46340, 48392, 50534, 52772,
    55108, 57548, 60096, 62757, 65535
};
```

Use the table above to implement the RGB function. Provide a testcase to check if the glow ambience is good or if a recalibration is required.

2.1.4 Lessons learned

Congratulations, you have just finished the implementation of your first complex driver. Let's summarize the process once more

- The first step was to create an overview and understanding of the complete system. We have used an UML class diagram for this, but a block diagram would also do the trick. The focus at this stage was the structure - we did not care about any details.
- Out of the complete system, we identified the LED driver as the first part of the skinny sheep to be implemented.
- We have started by writing down the functional and non-functional requirements for this driver. This answered the question: WHAT do we want to do with the driver.
- Then we have created a design using the UML class diagram to describe the interfaces. As we are still working with relatively small system, we had to create only one class.
- Then we checked which low level peripherals are required and added them to the top level hardware design.
- We then implemented the first simple functions and tested the system.
- This was repeated several times, adding and testing more and more functionality.

This process should be followed for all future implementations.

2.1.5 Adding the fur - the joystick driver

In the next step, we want to implement the joystick driver, fulfilling the following requirements

Req-Id	Description
FR1	We want to treat the x and y position as <code>sint8_t</code> value, having the following conventions <ul style="list-style-type: none">• Joystick left: <code>x = -128</code>• Joystick right: <code>x = 127</code>• Joystick down: <code>y = -128</code>• Joystick up: <code>y = 127</code>
FR2	And we want to be able to initialize the component (default requirement for any driver)
NFR1	Furthermore we want to have a uniform return type for any driver call, which will be represented by a common enum.
NFR2	Follow all design and naming conventions presented so far.
NFR3	Avoid implicit casts.
NFR4	Avoid global variables.

Create a class diagram describing the API of the joystick driver

For reading in the analogue value of the joystick position, we are using an Analogue Digital Converter (ADC) peripheral. Cypress provides a couple of different converters.

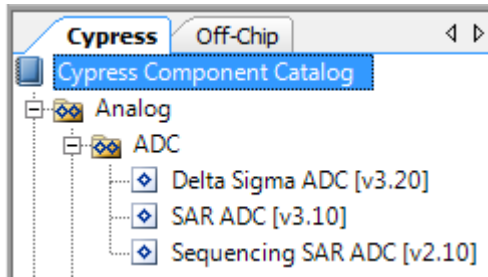


Figure 45 - Available ADC converters

Research and describe the functionality of the different converters. What are the advantages/disadvantages of the different technologies?

Implement the joystick driver by following the process used for the LED driver.

Test the functionality by implementing a testfunction which changes the color and brightness of the RGB LED based on the position of the joystick.

2.2 A Seven Segment and Button Driver

Effort: 3h	Category - B
Hardware analysis, driver design	

In this chapter, you will specify, design, implement and test another complex driver for the seven segment display. As you will be working with a couple of new hardware devices, the first step is to familiarize yourself with the following components. Find the technical specification in the internet and read it.

- Latch 74573
- Inverter CD74HC4049E

Then check the schematics and answer the following questions:

Describe the functionality of a latch.

What is the inverter being used for?

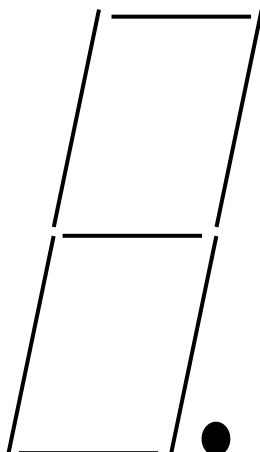
Why are we using the inverter/latch circuitry instead of connecting both 7 segment displays directly to the PSOC?

2.2.1 Seven Segment Driver

For the 7 segment driver, the following requirements are defined:

Req-Id	Description	Iteration
FR1	We want to write the values 0..9, A..F to every display individually	
FR2	We want to clear the display	
FR3	We want to read the value currently being displayed.	
FR4	We want to write a value 0..255 as hex value to both displays	
FR5	We want to set and clear the decimal point	
FR5	We want to initialize the display and set it to a cleared state	
NFR1	We will follow the project structure described in the previous exercise	
NFR2	We will use proper naming conventions and comments as introduced before	
NFR3	We will use only types defined in global.h	
NFR4	We will reduce the scope of variables as much as possible	
NFR5	Whatever can be const will be const	

Analyze the requirements and add the iteration for implementation next to it BEFORE starting with the implementation.



Check the schematics and add the port pin / bit required to set a specific LED of the seven segment display to the drawing

Create a class diagram describing the API of the seven segment driver

Implement and test the driver!

Hints:

- Use const lookup tables for defining the bit patterns for the displays
- Use a control register to have a simplified access to the displays
- In order to connect all pins, you have to go to the system tab and set Debug Select to GPIO

2.2.2 Button Driver

In the next step, we want to implement the button driver. The idea is to develop a small application which allows us to count the two seven segments up and down using the four buttons. Sounds like a trivial exercise - but unfortunately it contains some hidden complexity.

Let's have a look at the drivers we have implemented so far:

- LED's
- Seven Segment
- Joystick

All these drivers have been implemented as synchronous drivers, i.e. whenever they were called, the function immediately set or returned a value from the peripheral. Concerning the LED's and Seven Segment driver this absolutely makes sense. We write a value to the driver and the OUTPUT peripheral reacts immediately.

The same is somehow true for the joystick. Whenever we are calling the driver, we want to have the X and Y position exactly at that moment. Let's have a look at the code and the hardware configuration of the ADC.

The ADC by default is configured to run in a free-running, permanent mode. However, while the hardware is sampling, reading out the register would produce a wrong value. Therefore we have to wait until the current sampling is finished before we can read the data.

```
RC_t JOYSTICK_ReadPosition(sint8_t* x, sint8_t* y)
{
    //Wait for end of conversion

    JOYSTICK_ADC_XY_IsEndConversion(JOYSTICK_ADC_XY_WAIT_FOR_RESULT
);

    //Read channels
    uint16_t posX = JOYSTICK_ADC_XY_GetResult16(0);
    uint16_t posY = JOYSTICK_ADC_XY_GetResult16(1);

    //Convert to scaled 8bit signal
    *x = (sint8_t)((2048 - posX) / 16);
    *y = (sint8_t)((2048 - posY) / 16);

    return RC_SUCCESS;
}
```

Notice the function JOYSTICK_ADC_XY_IsEndConversion? This function stays in a while one loop until the sampling has finished.

This kind of synchronization is called busy waiting and should be avoided as much as possible, because it prevents the CPU from doing something sensible. In the worst case the CPU remains in such a loop forever.

In the case of the ADC this busy waiting period is limited to a few μs . In our applications we will not notice it. A better approach, which of course also provides quite some overhead would be the following:

1. Trigger the ADC to start sampling and to fire an interrupt when the sampling is finished.
2. Continue with some other code.
3. Once the sampling has finished, the interrupt is fired and the data can be fetched.

This pattern represents an asynchronous call, as we are firing a command and then wait for some unknown time until the answer is coming.

In this example, the context switch caused by the interrupt probably consumes more CPU time than the sampling and the code will be way more complex, therefore in this case, the busy waiting might be an acceptable choice.

But let's come back to the button driver. It seems that we have only two requirements:

Req-Id	Description
FR1	We want to read the button state of every button (pressed, not pressed)
FR2	And we want to be able to initialize the component (default requirement for any driver)

Design Approach 1

Let's simply implement a driver which returns the button press status in an enum `PRESSED / NOTPRESSED`.

Pro: Very simple function

Con: The user has to enter into a busy waiting loop if he wants to detect the exact moment when the button is being pressed

Design Approach 2

Hmm, sounds like we should use an interrupt for every button.

Pro: Very good detection of the moment, when a button is being pressed

Con: We have to use 4 external interrupt functions for this. Very often, microcontrollers do not provide that many of these functions.

Design approach 3

Let's combine both solutions. We can use a logical or gate to route all 4 pins onto 1 interrupt. When the interrupt is fired, we do not know which pin caused this. But inside the interrupt service routine we can use the simple function of design approach 1 to check for the responsible pin. As the interrupt service routine will be reached within a few μs , the pin very likely still will be pressed.

Buttons

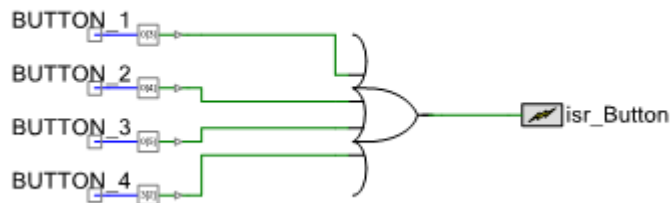


Figure 46 - 4 Pins 1 Interrupt

Pro: Very responsive and only 1 interrupt used

Con: Small risk that the pin is not pressed when being probed in the ISR, e.g. because a higher priority interrupt caused a high latency.

Implement and test the driver using approach 3. Your application should be able to count both seven segment displays up and down independently using the four buttons.

Actually we have a fifth button in our system - the joystick can be pressed as well. Shall we add this button to the button driver or to the joystick driver? Explain!

3 A first application

Using the complex drivers we have developed so far, we will implement some simple bare metal applications. The focus will be the design of slightly more complex systems, continuing using the principles we have learned in the previous chapter - NO HACKS!



3.1 Reaction Game

Effort: 4h	Category - B
Simple application, driver integration	

In this first application we are going to develop a simple reaction game.

Req-Id	Description
FR1	After pressing a start button, the system will wait for a random time.
FR2	Then, a random 7 segment will show a random value 1..4.
FR3	The user must press the corresponding button as fast as possible.
FR4	If the correct button is pressed, the time is measured and a point is given and a success message will be shown.
FR5	If the wrong button or more than 1 button are pressed no point will be given and an error message will be shown.
FR5	This will be repeated 10 times.
FR6	At the end of the game, the points, the individual time and the total time will be printed on the screen via the USART port.
NFR1	We will follow the project structure described in the previous exercise and create a new game.h c file, which will contain a run method which is called in the endless loop of main.
NFR2	We will use proper naming conventions and comments as introduced before
NFR3	We will use only types defined in global.h
NFR4	We will reduce the scope of variables as much as possible
NFR5	Whatever can be const will be const
NFR6	Reuse the complex drivers from the previous exercises

Draw an activity diagram to illustrate the behavior of the game.

First of all, plan sensible iterations. Plan testcases at the end of every iteration to verify the implemented behaviour

3.1.1 Iteration 1

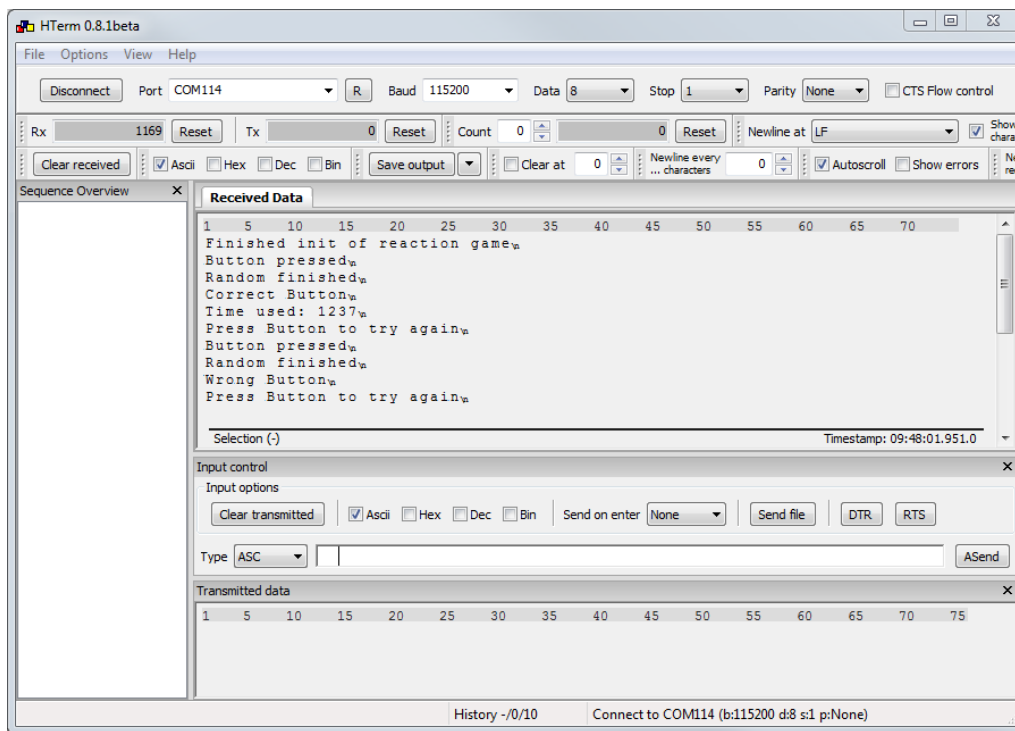
- Copy the required drivers from the previous project
- Set up the project folder structure (physical and logical)
- Copy the driver schematics
- Connect the pins as required
- Testcase 1: The project is compilable

3.1.2 Iteration 2

- Create the game.hlc file from the template and design the API
- Implement the init function for the game
- Testcase 1: The 7 segment display is cleared
- Testcase 2: A welcome message is shown on the UART
- ...

Add additional iterations as required.

Some sample output:



3.2 A simple encrypted protocol

Effort: 4h	Category - B
Communication protocol, ringbuffer, USART, encryption, own driver	

We want to implement a simple encryption protocol to secure the transmission of a UART protocol between 2 boards. For this job, we will develop an own buffered communication driver, define our own protocol and select a (one or more) feasible encryption algorithm. The requirements are given in the form of user stories:

The user wants

- to enter a key through the PC keyboard
- to enter some data which will be encrypted and transmitted through the USART
- to receive the data via the USART and see the decrypted content on the PC screen

Obviously building the application in one big-bang is way too complex. Instead we will use several iterations to solve aspects of the problem, before we build the complete system. The challenge is to find the good iterations. A good recipe is to identify the challenges related to the task and to find approaches to deal with them:

1. The communication will require 2 boards but we only have one
2. How can we identify the start and end of a transmission and how do we store the transmitted data
3. What kind of encryption do we need and how can we implement it

There might be more challenges, but let's try to find some answers for those first

1. Luckily our board has two UARTS, so we might simulate the communication on one board as a starting point. The same board will send and receive the data. Hint: Use a loop-back connection for one of the UART's
2. For this we have to define some kind of protocol for the data. As a storage medium, we need a ringbuffer - the simple most pattern for a FIFO buffer
3. We can start with a very simply encryption like XORing every byte. The challenge will be to define a generic interface which allows us to add more complex encryptions like AES or similar later on.

The key idea is to implement simple subsets of the program and to get them working, before you start with the complete application.

3.2.1 Iteration 1 - Elementary data transmission

In the upper left corner of the board you can identify two 3-pin headers which are labelled UART_1 and UART_2. Using some jumper wires, we will later-on connect two boards.

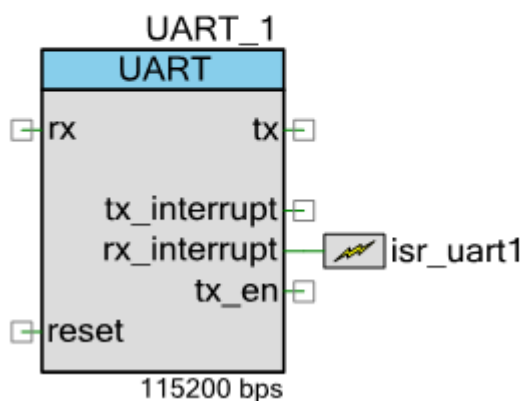
As we have only one board available, we will use one of the two UARTS on our board to build a first prototype. We can use the schematic editor to connect the pins for a loop back connection.

How do you connect the pins of the two boards?

Board 1	o	Board 2	o
Tx		Tx	
Rx		Rx	
GND		GND	

How do you connect the pins for a loopback connection?

Testsetup for USART communication



- Add the UART_LOG from the previous exercises and 1 additional UART to the system

- Connect the pins of the USARTs as required (don't forget to deactivate the HW connection of the default pin for this).
- Add an ISR to the RX interrupt
- Configure all USARTS to 115200, 8N1
- Configure the pins
- Add the default files global.h and derivate.h to your systems bsw folder

Before continuing, we will test this setup by implementing the following signal flow

- Any data, which is received by UART_LOG (by sending out data via HTerm) will be forwarded (by software) to UART_1
- Through the loopback, this data is received by UART_1 and will be forwarded (by software) to UART_LOG to be shown on HTERM

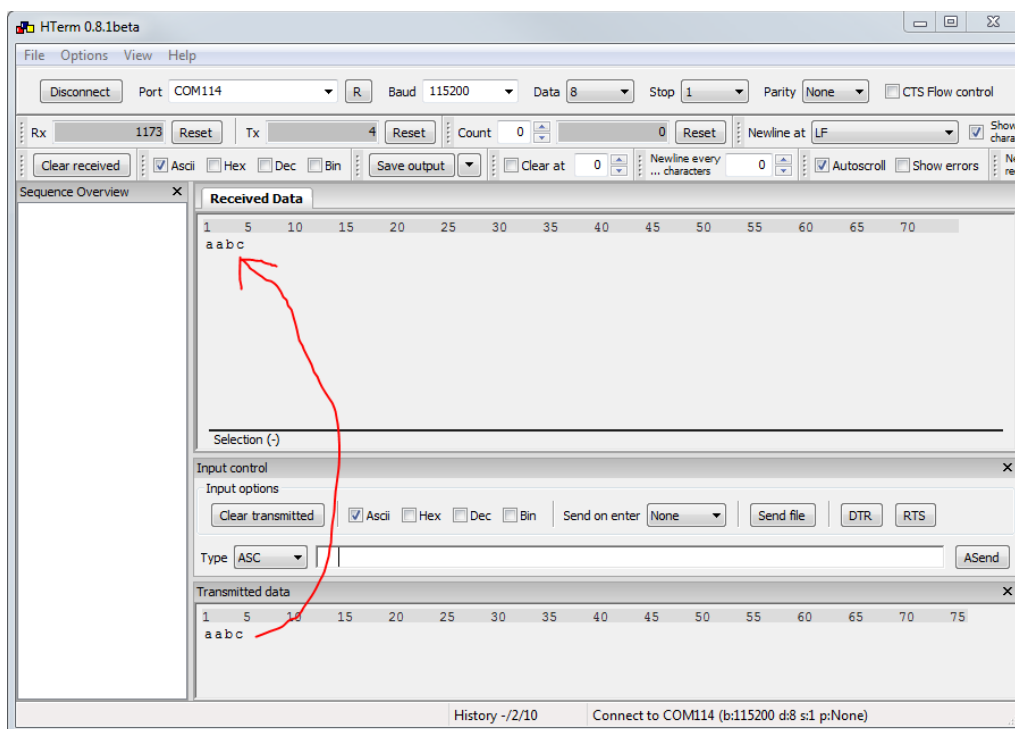


Figure 47 - Signalflow between all USARTs is working

3.2.2 Iteration 2 - Ringbuffer class

In the next step, we want to activate the interrupts for receiving the data. The received data shall be stored in a FIFO ringbuffer, which will be implemented as an own service class. This ringbuffer functionality will extend / replace the built in driver.

The functionality of a ringbuffer is described in the picture below.

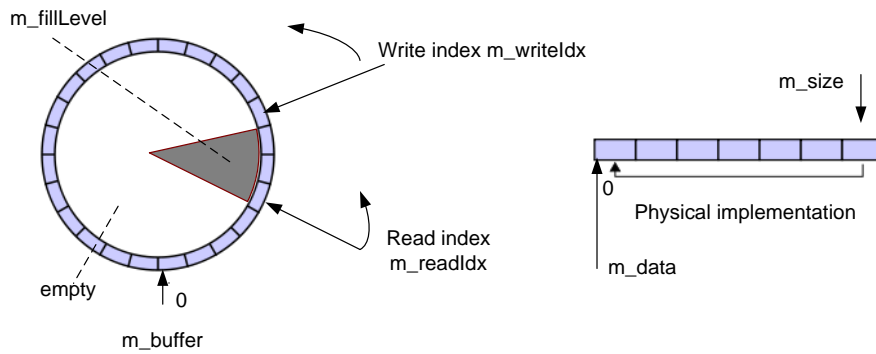


Figure 48 - Ringbuffer

Initially, both the read index and write index point to the position 0. Once a data element is written, the write index is incremented so that the next data element can be written to the next position. When reading a data element, the first element written will be read, then the second and so on. If the buffer is full, no data can be written. If the buffer is empty, no data can be read.

Implement the buffer in an object oriented way, by creating a struct containing the internal data of the buffer. A pointer to this struct will be passed as first parameter to all functions, i.e. this parameter translates to the “this” pointer in C++.

Write a testcase to create a buffer and read and write some data. Make sure to test the special behavior when the buffer is full or empty.

3.2.3 Iteration 3 - Combining the ringbuffer and the driver

Now we can write the code for the two ISR's and store the received data in the corresponding ringbuffer object. For this, create two global ringbuffer objects. In the ISR, make sure that you read all elements which might have been received in the meantime. Rough code structure:

```
CY_ISR(isr_uart1_rx)
{
    uint8_t data;

    while(UART_1_GetRxBufferSize() != 0)
        //Todo: Add some counter to terminate after 4 bytes max
        {
            data = (uint8_t)UART_1_GetByte();
            //Todo: Store the data in your ringbuffer object
        }
}
```

Extend the testcase of iteration 1 to test the signal flow.

3.2.4 Iteration 4 - Protocol development

A typical serial protocol consists of the following elements

Prefix	Length	Payload 1	...	Payload n	CRC	Postfix
--------	--------	--------------	-----	--------------	-----	---------

The postfix and prefix are unique patterns which allow a parser to identify the start and end of the protocol. The length information allows the receiver to allocate sufficient storage to store the complete payload. The CRC checks for possible transmission errors.

We will however start with a much simpler protocol having the structure

Length	Payload 1	...	Payload n	Postfix = 0 ⁵
--------	--------------	-----	--------------	-----------------------------

For the implementation, we will add a protocol class. This class will be responsible to encrypt / decrypt a C-string and to transfer it to and from the UART. Let's start by designing the API and to describe the functionality a bit more in detail.

Sending

The send function is rather straight forward. We simply pass a string which will be processed immediately in a synchronous way.

- A C-string will be given to the send function
- The function will encrypt the string (will be added in the next iteration)
- And create a protocol string using the defined format above
- This protocol then will be send out via the USART

Receiving

The receive function is more complex. On the one hand side we do not know, when the data will come and we do not know the size of the protocol, which makes it hard to provide sufficient memory to store it. Of course we could define something like a max size, but this is rather inflexible and hacky.

Let's start by analyzing the first problem. In the previous iteration we have implemented an own ringbuffer which is used to store the data. This happens in the ISR. I.e. we might want to add a check to the ISR if a complete protocol is available. If this is the case, we "inform" the main program that the processing of the protocol can start.

⁵ We agree that 0 will never be part of the payload and also the size will never be 0. Furthermore 0 also acts as termination for a C-string, which allows us to use some functions of the string library if needed. Check the definition of a C-string in the Internet if you are not sure how the terminating 0 is used.

The alternative approach would be to check in the superloop, if a protocol is available and then become active.

The first approach sounds more elegant, but as we do not have a solution to “wake up” the main program from an ISR - this is something where events from the RTOS⁶ become very handy - we will go for the second solution in this exercise and implement the other pattern once we have introduced the Erika RTOS.

In addition to checking if a full protocol is there or not, we can also use the same function to query for the size of the protocol, which allows us to provide a large enough string to store the data.

This results in the following three functions which need to be implemented for the protocol handling in an own protocol file:

```
/**
 * Check for a full protocol in the ringbuffer
 * @param RB_ringbuffer_t const * const buffer
 *      - IN: pointer to the ringbuffer to be checked
 * @param uint8_t * length
 *      - OUT: length of the protocol found
 * @return TRUE if protocol is available, FALSE otherwise
 */
boolean_t PROT_avail(RB_ringbuffer_t const * const buffer,
                    uint16_t * length);

/**
 * Write function for the ringbuffer
 * @param char const * const data - IN: string to be sent
 * @return RC_SUCCESS if all ok, RC_ERROR if error occurred
 */
RC_t PROT_write(char const * const data);

/**
 * Read function for the ringbuffer
 * @param RB_ringbuffer_t *const me
 *      - IN/OUT: pointer to the ringbuffer object
 * @param RB_data_t *const data
 *      - OUT: data element read (string). Size of the allocated must
 *      be large enough to hold the data.
 * @return RC_SUCCESS if all ok,
 *      RC_ERROR_BUFFER_EMPTY if buffer was empty,
 *      RC_ERROR_BAD_DATA if the buffer does not contain a full protocol
 */
RC_t PROT_read(RB_ringbuffer_t *const me, char *const data);
```

⁶ Real Time Operating System

Draw an activity diagram for the read function

Implement the code for the functions above and add a testcase to the program to verify their functionality.

3.2.5 Iteration 5 - Encryption Algorithm

In the almost final iteration we will now add an encryption algorithm. For the sake of simplicity, the key will be a single byte, which will be used to XOR every byte of the payload (except the EOF termination character 0).

These functions will only be used locally inside the already existing receive and send functions. Therefore we will declare those as a set of file static functions in the `protocol.c` file. Of course, we have to add an additional parameter `key` to the API of the existing functions.

Design, implement and test the new functionality. In the testcase, print the original and encrypted message.

3.2.6 Iteration 6 - Getting it all together

At this point we have solved all the individual challenges. If the design is good, it should be pretty straight forward to extend the structure to work with two boards, fulfilling the following requirements:

- Either side can enter a key through HTERM
- Either side can enter a text string (termination e.g. by CR) through HTERM
- Upon reception of a full protocol, the received data as well as the encrypted data will be shown

Hint: You need a state machine for the menu operations and independent functions for sending and receiving.

Sample output:

```
Testprogram Encryption
=====

Menu
1 - Enter Key
2 - Send message

Menucommand: 1
  Please enter a key (0...255). Terminate with '.'<\n>
  Key set to: 12<\n>

Menu
1 - Enter Key
2 - Send message

Menucommand: 2
  Please enter a message. Terminate with '.'
  Message transmitted: Hello

Menu
1 - Enter Key
2 - Send message

Received: Hello

Received: Idmmn - Same output with wrong key
```

4 Erika OS

Although we could already implement some interesting applications using a bare metal structure, we will now see that the introduction of an embedded operating system will allow us to implement better readable and maintainable code, using tasks, events, alarms and other mechanisms.

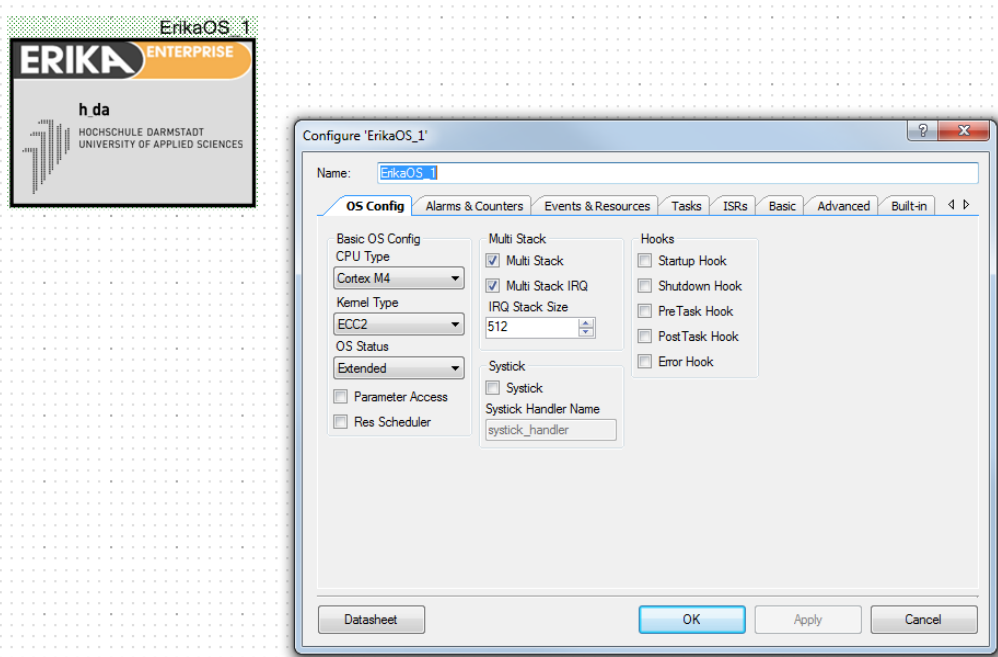


Figure 49 - Erika OS Component

4.1 Setting up the OS

Effort: 4h	Category - A
Erika OS, Adding components, tasks, alarms, events	

In this sample project we will setup Erika OS, the free version of the AUTOSAR operating system.

4.1.1 Iteration 1 - A first task

- In the first step you have to download the latest Erika component release from Moodle and store it locally on your computer.
- Then, change the Workspace Explorer View to Components (by selecting the vertical tab) and with a right click select "Import Component".
- Select Import from Archive and enter the location of the previously downloaded component

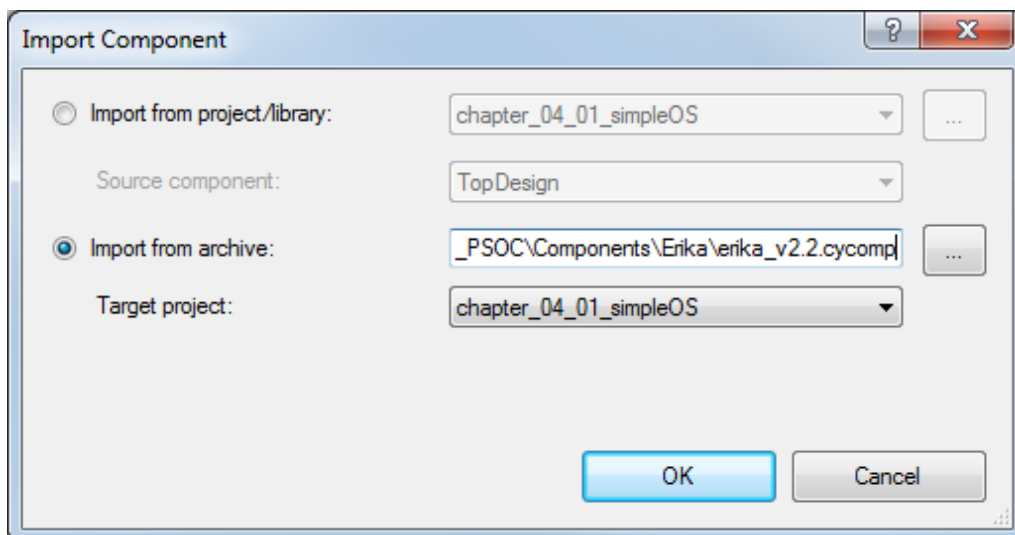


Figure 50 - Import Erika OS component

- The new component now should be visible in the project and can be added to the TopDesign

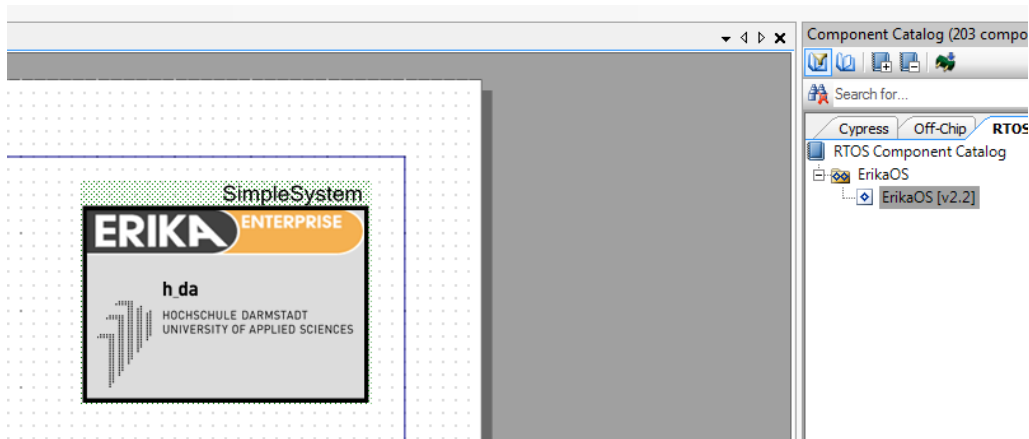


Figure 51 - Erika OS added to the TopDesign

- Doubleclick on the component, navigate to the tab tasks, and enter a first task "tsk_init".
 - Activate AutoStart and
 - set Task Schedule to Non

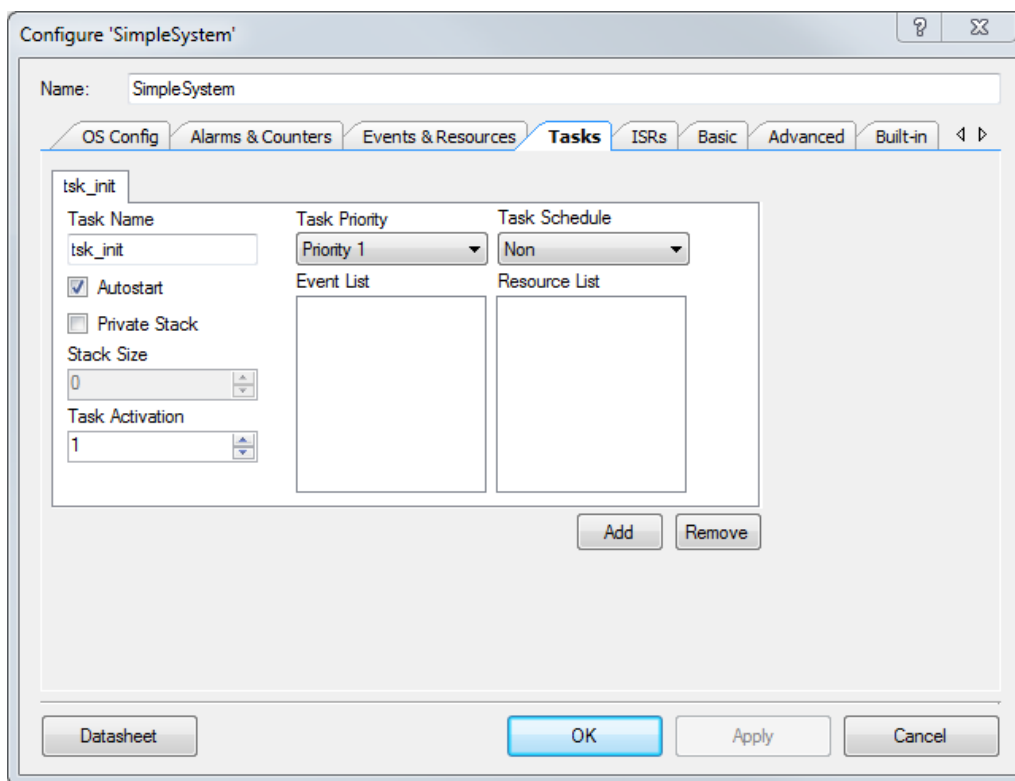


Figure 52 - A first task

In the main file, add the following code

```
#include "project.h"
#include "global.h"

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    // Start Operating System
    for(;;)
        StartOS(OSDEFAULTTAPPMODE);
}

/*****
*****
* Task Definitions
*****
*****/

TASK(tsk_init)
{
    TerminateTask();
}
```

Set a break point on the line `TerminateTask` and start the program in the debugger. Congratulations, you just started your first task.

4.1.2 Iteration 2 - A first blinking LED

In order to get a bit more action let's add the button and led modules from the previous project.

- Copy the complex driver files to your project
- Add the required peripherals and configure the pins.

As we will need the HMI peripherals in several projects, it is recommended to place them on an own sheet. Using windows copy and paste you can easily copy them from one project to another. Unfortunately I have not found a way to copy the port configuration yet - let me know if you find a solution.

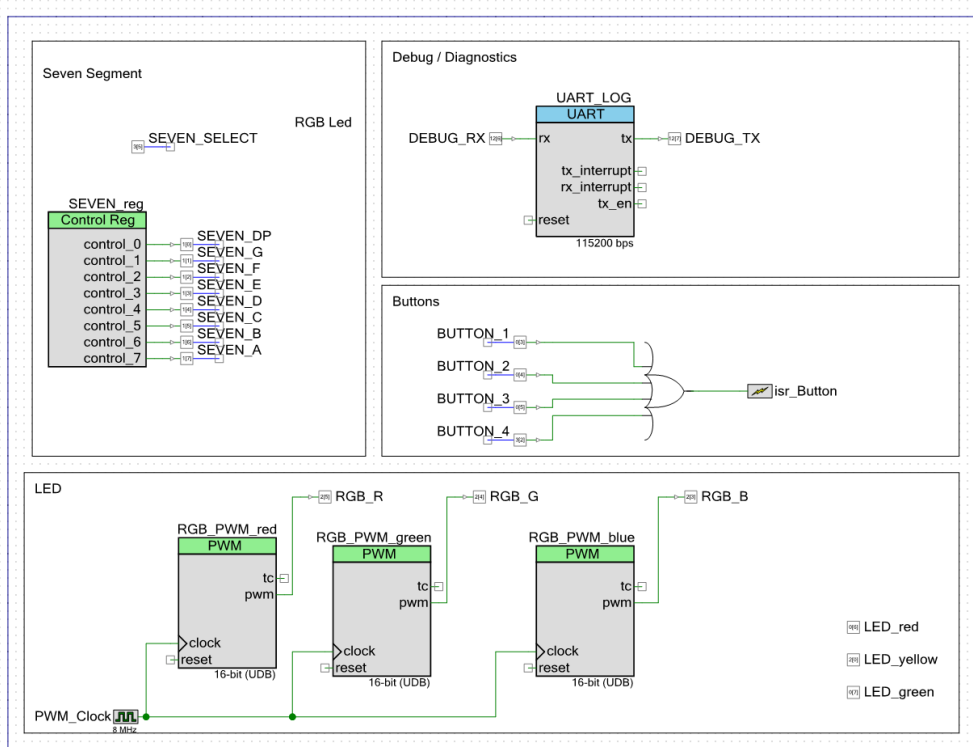


Figure 53 - Typical HMI peripheral configuration

#	Name	Port	Pin	Lock
<input checked="" type="checkbox"/>	BUTTON_1	P0 [3]	51	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	BUTTON_2	P0 [4]	53	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	BUTTON_3	P0 [5]	54	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	BUTTON_4	P3 [2]	31	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	DEBUG_RX	P12 [6]	20	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	DEBUG_TX	P12 [7]	21	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	LED_green	P0 [6]	55	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	LED_red	P0 [7]	56	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	LED_yellow	P2 [0]	62	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	RGB_B	P2 [3]	65	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	RGB_G	P2 [4]	66	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	RGB_R	P2 [5]	68	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_A	P1 [7]	19	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_B	P1 [6]	18	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_C	P1 [5]	16	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_D	P1 [4]	15	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_DP	P1 [0]	11	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_E	P1 [3]	14	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_F	P1 [2]	13	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_G	P1 [1]	12	<input checked="" type="checkbox"/>
<input checked="" type="checkbox"/>	SEVEN_SELECT	P3 [5]	34	<input checked="" type="checkbox"/>

Figure 54 - HMI Port Configuration

For a blinking LED we will need a systick handler and a counter object, which will be used to reschedule the tasks based on time events.

- In the OS Config tab, activate the SysTick feature.

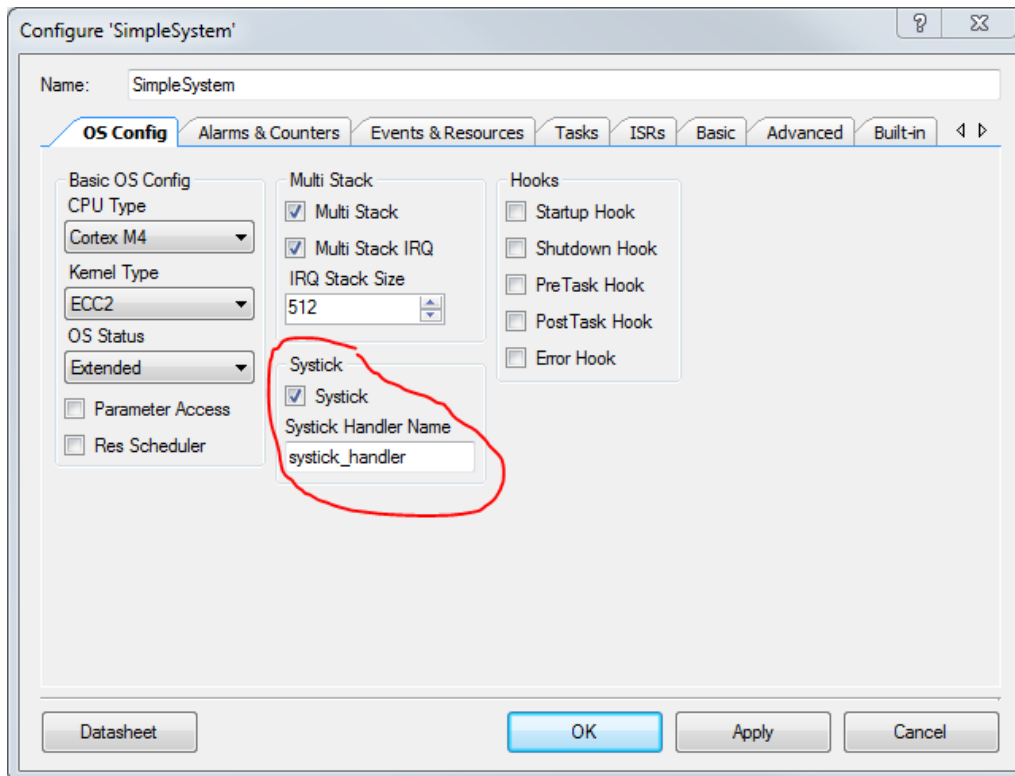


Figure 55 - SysTick configuration

- and add a first counter object
- Furthermore, we can add a first alarm `alarm_ledBlink`, which will activate the task `task_ledBlink`

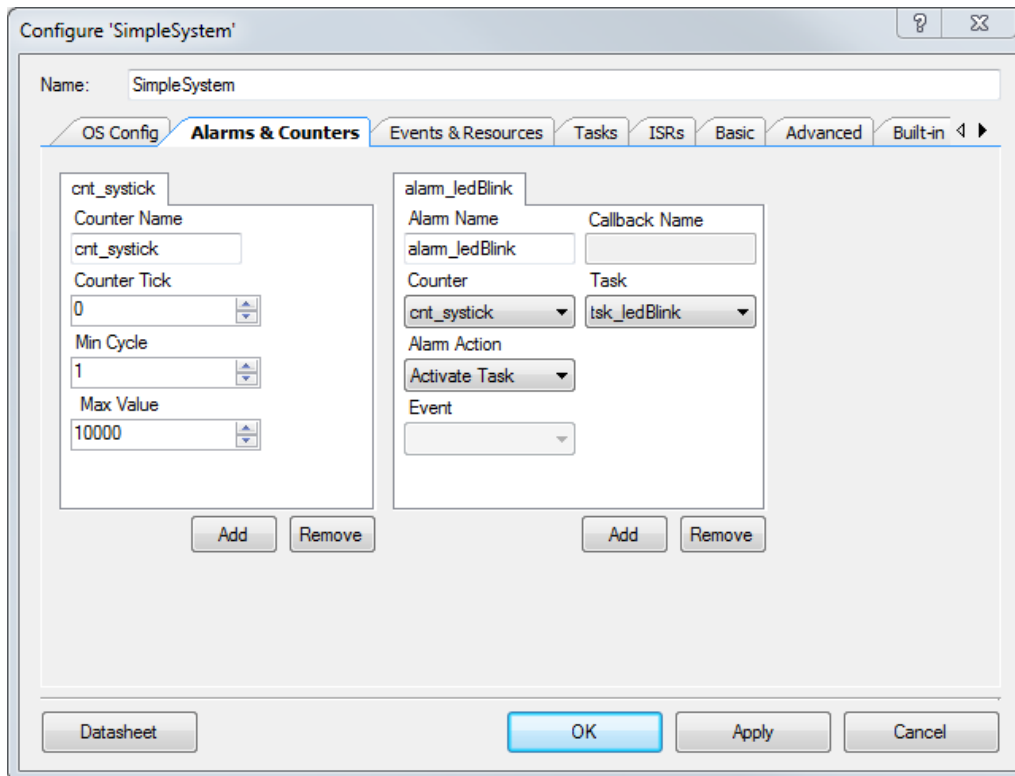


Figure 56 - Counter Object and Alarm

In the main.c file, we now have to add the code for the systick handler and we have to configure the systick timer to the required time base, in our example 1ms. Furthermore we will add a `unhandledException` handler which will terminate the program in the debugger just in case.

```
//ISR which will increment the systick counter every ms
ISR(systick_handler)
{
    CounterTick(cnt_systick);
}

int main()
{
    CyGlobalIntEnable; /* Enable global interrupts. */

    //Set systick period to 1 ms. Enable the INT and start it.
    EE_systick_set_period(MILLISECONDS_TO_TICKS(1,
                                                BCLK__BUS_CLK__HZ));

    EE_systick_enable_int();
    EE_systick_start();

    // Start Operating System
    for(;;)
        StartOS(OSDEFAULTAPPMODE);
}

void unhandledException()
{
    //Ooops, something terrible happened...
    //check the call stack to see how we got here...
    __asm("bkpt");
}
```

The `tsk_init` is extended in order to initialize the peripherals and to setup the task activations. Note, that you first have to initialize the drivers and afterwards call the `EE_system_init()` function again to override the interrupt configuration with the data configured in the OS:

```
TASK(tsk_init)
{
    //Init MCAL Drivers
    LED_Init();
    SEVEN_Init();

    //Reconfigure ISRs with OS parameters.
    //This line MUST be called after the hardware driver initialisa-
    tion!
    EE_system_init();

    //Start the alarm with 100ms cycle time
    SetRelAlarm(alarm_ledBlink, 100, 100);

    TerminateTask();
}
```

The blink task then comes as

```
TASK(tsk_ledBlink)
{
    LED_Toggle(LED_ALL);
    TerminateTask();
}
```

The blink task is activated by the alarm every 100ms. But what happens the rest of the time? Instead allowing the OS to “busy wait” until the next rescheduling takes place, we could add another task which will be used as a background task.

Now let's play around with the task priorities. Describe the system behavior with the following configurations:

tsk_ledBlink: Prio 2

tsk_background: Prio 8

tsk_ledBlink: Prio 8

tsk_background: Prio 2

How can you explain the different behavior?

4.1.3 Iteration 3 - A simple watch

We now want to use an event, which will increment the seven segment display every 1s. The event will be fired from the tsk_ledBlink() to a new task tsk_sevenSet()

Add the event and new task to the configuration. Please note, that you have to add the event to the task event mask to be received. A task receiving events is an extended task and therefore should have a private stack.

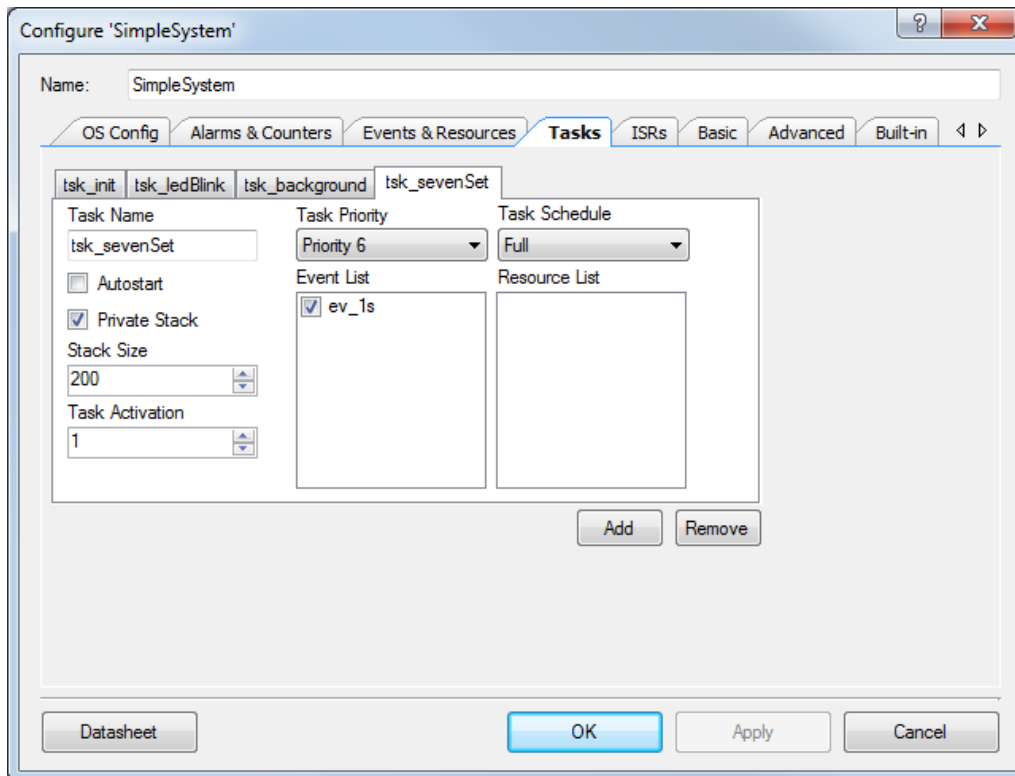


Figure 57 - A first extended task

4.1.4 Iteration 4 - Adding a reset button

In a last step, we want to introduce an button ISR, which will fire another event `ev_reset` to the `tsk_sevenSet` which will reset the time count to 0 in case any of the buttons is pressed.

Please read the next chapter before starting with this exercise!

First, we have to check for the name of the ISR object, which in our example is `isr_Button`.

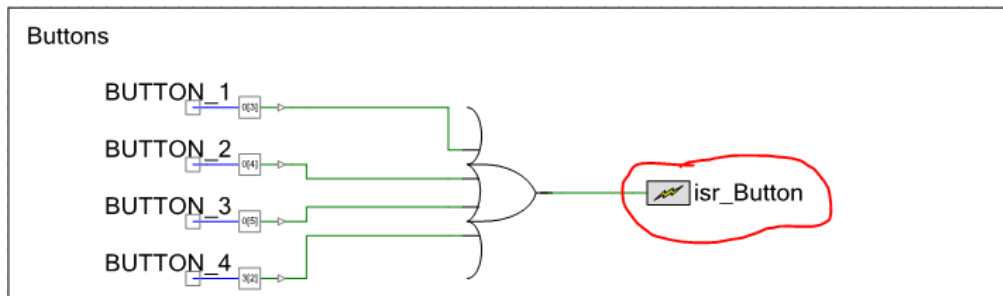


Figure 58 - ISR object name

This name is then added to the OS configuration. As we want to fire an event from the ISR, it must be a category 2 interrupt service function.

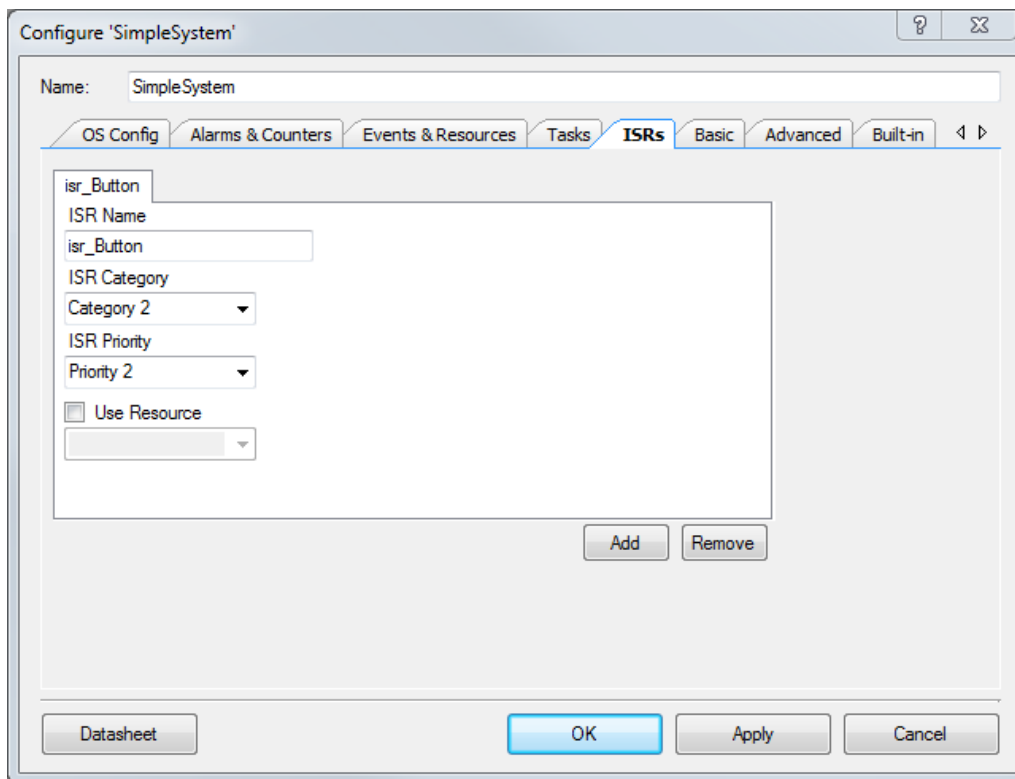


Figure 59 - Configuring the ISR in the OS

And as a final step we have to add the code of the ISR to our project and modify the tasks as needed.

```
ISR2(isr_Button)
{
    SetEvent(tsk_sevenSet, ev_reset);
}
```

4.2 Technical background - Interrupts

Before working on this chapter, make sure, that you have understood the basic principles of interrupts described in the chapter 1.3 Introducing interrupts before.

In Erika OS, we will use by design an interrupt handler, which has the same name as the interrupt object. The interrupt entries which have been set by the corresponding start and init functions of the various components, will be overwritten with this new handler in the OS function `EE_system_init()`, which therefore must be called after the component init functions.

4.2.1 Configuring interrupts using ISR components

Step 1

Add an ISR component to your design as shown in the previous chapter. In our example, we will add a external interrupt to handle RX interrupts from a UART called lidar (as it is receiving messages from a lidar sensor).

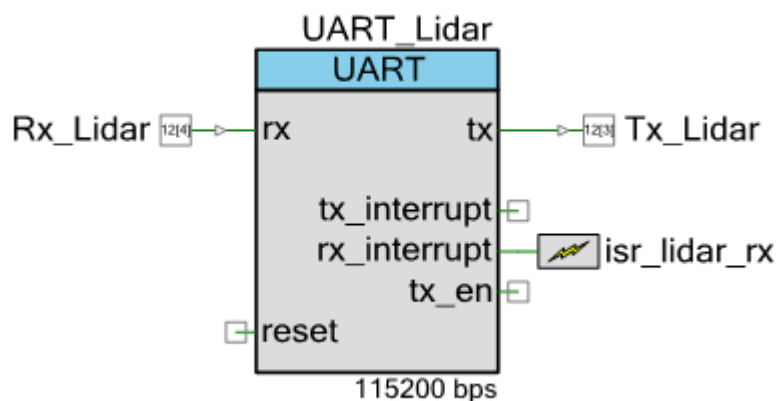


Figure 60 - Adding an RX interrupt to a UART port

Step 2

The name of the interrupt object (`isr_lidar_rx`) now has to be added to the Erika OS configuration. Make sure that you use exactly the same spelling as in the isr component.

If you want to use OS-calls in the interrupt handler, e.g. firing events, you have to make it a category 2 interrupt.

Choose the required priority. Lower numbers have a higher priority.

If needed, add resources to protect critical regions.

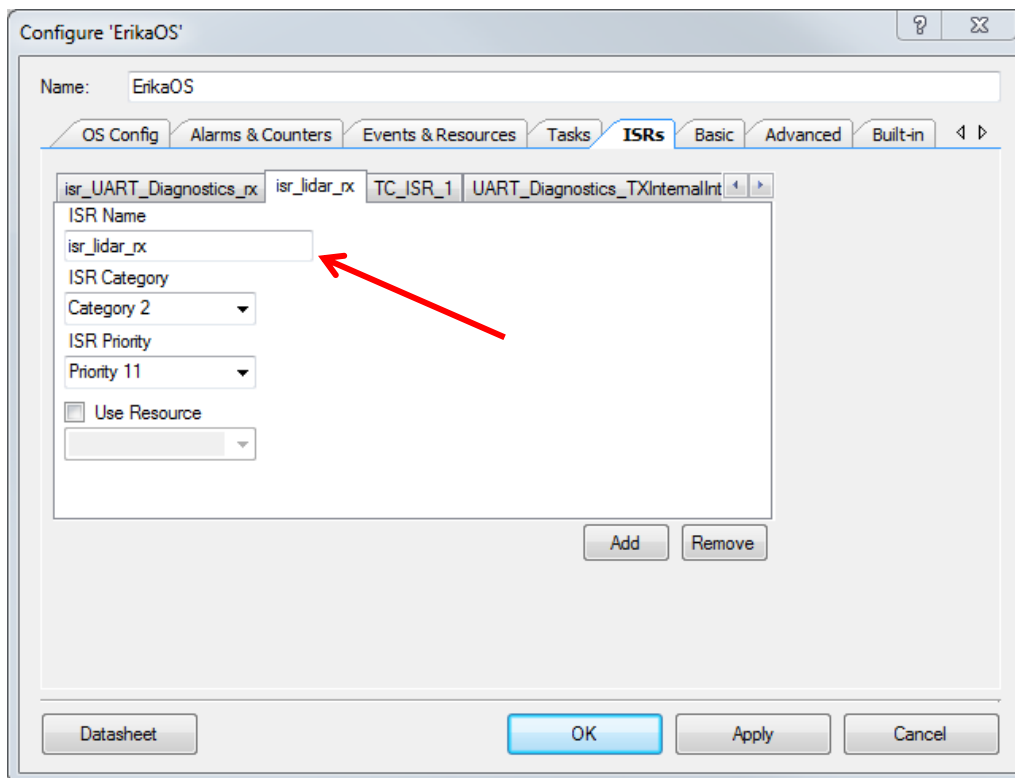


Figure 61 - Adding an interrupt to Erika OS configuration

Step 3

Now verify, that the interrupt object is visible in the interrupt tab. The priority can be ignored, as it will be overwritten with the value provided in the Erika OS configuration.

Start Page	*TopDesign.cysch	Lidar.cydwr	main.c	CAN_INT.c	CAN.c
Instance Name	Interrupt Number		Priority (0 - 7)		
CAN_isr	16		1		
I2C_OLED_I2C_IRQ	15		7		
isr_lidar_rx	4		7		
isr_UART_Diagnostics_rx	3		7		
TC_ISR_1	0		7		
UART_Diagnostics_TXInternalInterrupt	1		7		
UART_Lidar_TXInternalInterrupt	2		7		

Figure 62 - Verifying the interrupts in the system table

Step 4

Add an ISR handler to your system. The name of the handler is identical to the name provided in the isr component and Erika configuration. Make sure to use the correct ISR category macro.

Typically, this ISR will only serve as a wrapper calling your own handler implementation to improve reusability.

```
ISR2(isr_lidar_rx)
{
    //Call your own handler
    LIDAR_sensor_rx_isr();
}
```

Step 5

Update the function `hardware_init` or `init` task. Please note, that the OS configuration must be called after all other init functions as it overrides the entries in the interrupt vector table.

```
void hardwareInit()
{
    // All hardware needs to be initialized after the OS to
    // guarantee that no ISRs use event functions before the OS
    //has started.
    // As the hardware start function reconfigure the interrupt
    // it will overwrite the OS configs.
    // To avoid this we call EE_system_init again
    // Interrupts are disabled while hardware is configured
    // to avoid calling wrong ISRs

    /* ALL HARDWARE INITIALIZATION MUST BE DONE HERE */
    CyGlobalIntDisable;
    //UART Inits
    UART_Lidar_Start();

    //Reconfigure ISRs with OS parameters.
    EE_system_init();

    CyGlobalIntEnable;
    return;
}
```

4.2.2 Special component configuration

CAN

CAN provides a built-in interrupt vector and interrupt handler. The interrupt vector is called `CAN_isr` and the handler is called `CAN_ISR`. In order to use it, you have to create an own handler

UART internal handler

When working with protocols, the structure of an ISR typically is as follows

- Read the byte from the UART
- Check for transmission errors
- If everything is ok, store the byte in a FIFO (ringbuffer), otherwise do some error handling
- Check if the protocol is complete (e.g. by checking for end of protocol bytes or the length)
- Inform a task by firing an event that a full protocol is available and can be processed

This behavior cannot be implemented by using the built in buffer. Therefore it is recommended to set the size of the RX (and TX) buffer to max 4 (hardware buffer size on this MCU) and to connect an external ISR. This will allow you to implement your own ISR instead of using the built-in variant. Check the chapter on the Encryption Protocol for details.

Other than in this exercise, which permanently polled the ringbuffer status in the super loop we now might fire an event to a reception task to decode a protocol and to print it on the screen.

Try to port the code of the encryption protocol to an Erika architecture! Most of the code probably can be reused, if you designed your system properly - check also the exercise for the communication stack for some suggestions.

As setting up a project requires quite some efforts, we will save this project as a template project. For this, right click on the project folder and select “Copy to My Templates”.

4.3 Inter-Task Communication, Messaging and Critical Sections

Effort: 4h	Category - B
Erika OS, resources, data exchange between tasks, critical sections	

In this exercise, we will investigate different design patterns for data exchange between tasks and have a look at critical sections.

To reduce efforts when setting up the project we will use the template project from the previous exercise. For this, simply select this project as template when creating the new project - assuming that you have created the template before.

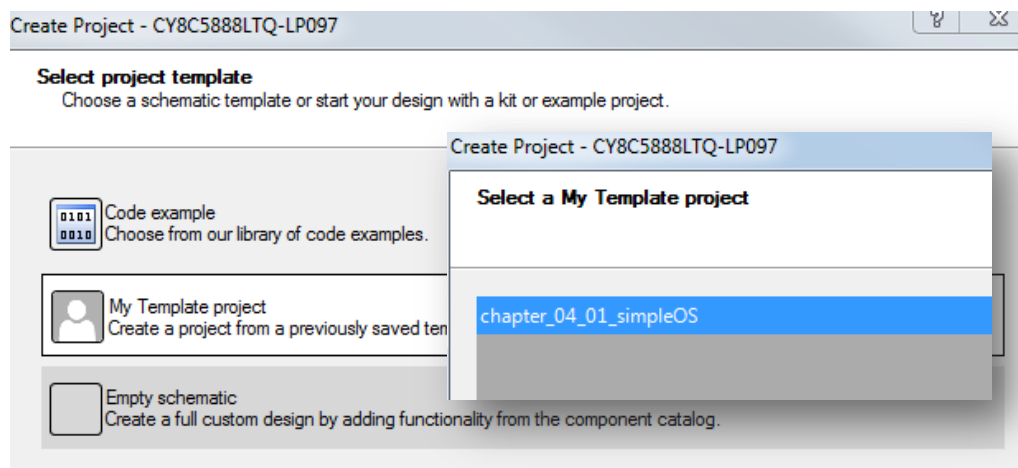


Figure 63 - Creating a new project using a template

The storyline for this exercise is as follows:

- we have 2 sensors, which are providing data for a control task
- the control task reads out the data and depending on the value performs an action
- the two sensors will be called by 2 cyclic basic tasks
- the control loop will be called by a control task

4.3.1 Iteration 1 - Task creation and calling order

In the first iteration, we are creating the required tasks, alarms and counters.

Note: The inittask and background task are required for every system and therefore are not explicitly mentioned.

Task Name	Basic / Extended	Stack Size	Priority
tsk_sensor1	Basic	Default	5
tsk_sensor2	Basic	Default	6
tsk_control	Basic	Default	7

tsk_control has the highest priority, tsk_sensor1 has the lowest priority. In other words, the control task may preempt the two sensor tasks and tsk_sensor2 may preempt tsk_sensor1.

In addition, create a system counter and three alarms for activating the tasks.

In the main file, add the code for the init task and configure the three alarms to be cyclically fired in 100ms intervals.

```

TASK(tsk_init)
{
    //Init MCAL Drivers
    UART_LOG_Start();

    //Reconfigure ISRs with OS parameters.
    //This line MUST be called after the hardware driver
    //initialisation!
    EE_system_init();

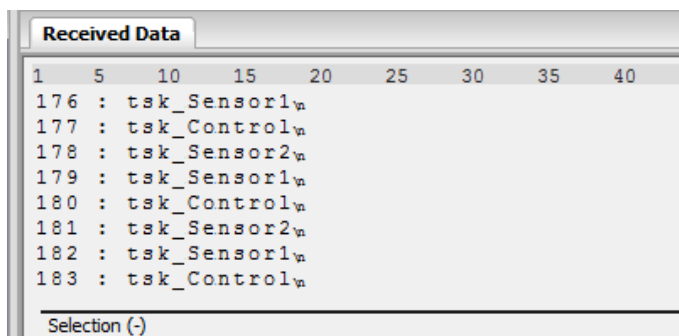
    //Start the alarm with 100ms cycle time
    SetRelAlarm(alarm_actSensor1,100,100);
    SetRelAlarm(alarm_actSensor2,100,100);
    SetRelAlarm(alarm_actControl,100,100);

    ActivateTask(tsk_background);

    TerminateTask();
}

```

In order to investigate the calling order a bit deeper, we will add the following or a similar print routine, which is called in the individual tasks and produces the following output in the console



.Figure 64 - Output of the three cyclic tasks

```
volatile uint32_t counter = 0;

void printCall(char const * const name)
{
    char t[10] = {0};
    itoa(counter++, t, 10);

    UART_LOG_PutString(t);
    UART_LOG_PutStringConst(" : ");
    UART_LOG_PutStringConst(name);
    UART_LOG_PutStringConst("\n");
}

TASK(tsk_sensor1)
{
    printCall("tsk_Sensor1");

    TerminateTask();
}
```

Compare the activation order in the init task with the calling order. In which order are the tasks activated by the scheduler? How can this be explained?

4.3.2 Iteration 2 - Changing the timing / a first critical section

We now want the tasks to be executed in the order

- tsk_sensor1
- tsk_sensor2
- tsk_control

Investigate the meaning of the 2 time parameters in the SetRelAlarm command. What do they mean / influence?

Change the parameters to have the three tasks executed in the given order and a delay of 1ms, i.e

- tsk_sensor1: 100ms, 200ms, 300ms, ...
- tsk_sensor2: 101ms, 201ms, 301ms, ...
- tsk_control: 102ms, 202ms, 302ms, ...

Which parameters do you have to set to achieve the mentioned behavior?

```
SetRelAlarm(alarm_actSensor1, _____, _____);
```

```
SetRelAlarm(alarm_actSensor2, _____, _____);
```

```
SetRelAlarm(alarm_actControl, _____, _____);
```

When checking the output in the console, you will see the following slightly corrupt data:

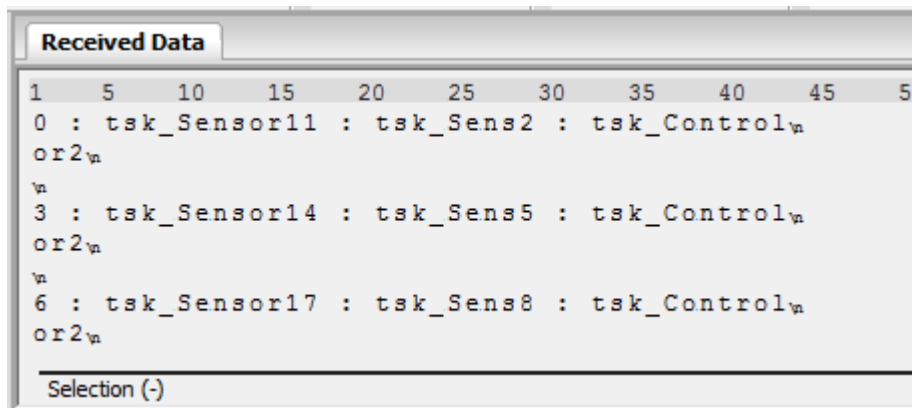


Figure 65 - Corrupted data

How can you explain the corrupted data

- 1) Draw a task / state diagram illustrating what is happening after 100, 101, 102, ...ms
- 2) Calculate the number of chars which can be transmitted during 1ms
- 3) Based on the previous 2 answers, explain the behavior

Obviously the function call `printCall(<task name>)` in every task is a critical section. If the task is interrupted while processing this call, the output gets corrupted. Probably not that critical in case of a `printf` statement, but let's assume the data controls an engine. A plane might crash due to such a bug.

A possible solution would be to increase the delay between the task calls in such a way, that all data can be transmitted over the serial port. In this example - at least as long as we are not placing longer strings into the output, 10ms should be plenty.

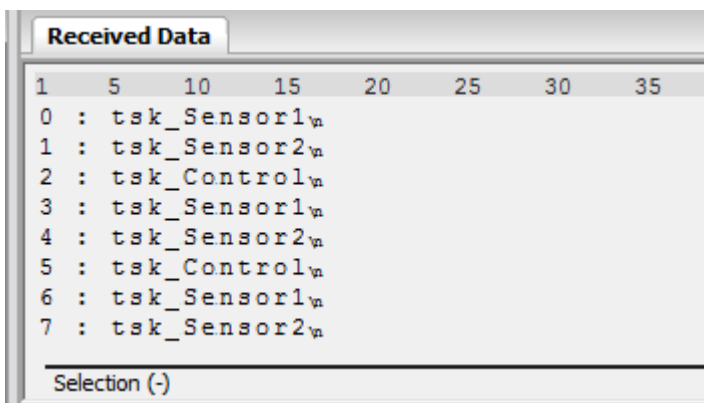


Figure 66 - Corrected output

However, this solution must be considered to be a bad hack, as any modification of the string length in the printf statements again will lead to a wrong behavior.

The proper solution is the introduction of a semaphore or mutex, which is called resource in OSEK. For this, add a resource `res_printf` to the OSEK configuration, activate it for the three application tasks and place it around the critical region

```
GetResource(res_printf);
//Start of critical section

UART_LOG_PutString(t);
UART_LOG_PutStringConst(" : ");
UART_LOG_PutStringConst(name);
UART_LOG_PutStringConst("\n");

//end of critical section
ReleaseResource(res_printf);
```

The output on the console now should be correct, as the resource will ensure, that also a higher priority task will not enter this region, if a lower priority task has entered (and not left) it before.

Note: If the output still is corrupted, you probably forgot to activate the resource for all the tasks.

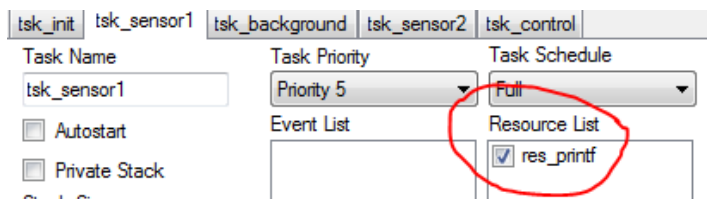


Figure 67 - Ressource activation

4.3.3 Iteration 3 - Messaging

The OSEK kernel itself does not provide any message concept. Instead, intertask communication is part of higher middleware layer like OSEK COM or AUTOSAR RTE.

The typical implementation is to use shared memory (i.e. global data) for this. We will create two global sensor objects, having the following structure, which will be set by the two sensor tasks and read out by the control task.

```
typedef struct{
    uint32_t timestamp;
    uint32_t value;
    uint32_t reliability;
} sensor_t;

volatile sensor_t sensor1 = {0,0,0};
volatile sensor_t sensor2 = {0,0,0};
```

To simulate the sensor behavior, we will use a function which will return a dummy sensor object based on the counter value (which is incremented in every print call).

```
sensor_t getSensor()
{
    sensor_t s = {0,0,0};

    s.value = counter;
    s.timestamp = counter;
    s.reliability = counter;

    return s;
}
```

The expected value in the global sensor objects are incrementing but identical numbers. This can be checked by the following function, which is called in the control task.

```
void checkSensor(char const * const name, volatile sensor_t const *
const s)
{
    if (s->reliability == s->timestamp && s->reliability == s->value)
    {
        //all ok
    }
    else
    {
        GetResource(res_printf);
        //Start of critical section
        UART_LOG_PutStringConst(name);
        UART_LOG_PutStringConst(" - wrong data in the sensor\n");
        //end of critical section
        ReleaseResource(res_printf);
    }
}
```

Add the calls to the tasks:

```
TASK(tsk_sensor1)
{
    printCall("tsk_Sensor1");

    //Simualted sensor call
    sensor1 = getSensor();

    TerminateTask();
}

TASK(tsk_sensor2)
{
    printCall("tsk_Sensor2");

    //Simualted sensor call
    sensor2 = getSensor();

    TerminateTask();
}

TASK(tsk_control)
{
    printCall("tsk_Control");
    checkSensor("S1", &sensor1);
    checkSensor("S2", &sensor2);

    TerminateTask();
}
```

In the first testrun, everything looks ok. No error message is displayed on the console.

Now let's add some more verbosity output by adding an additional printCall in the tasks, showing the call of the setSensor and checkSensor function.

```
TASK(tsk_sensor2)
{
    printCall("tsk_Sensor2");

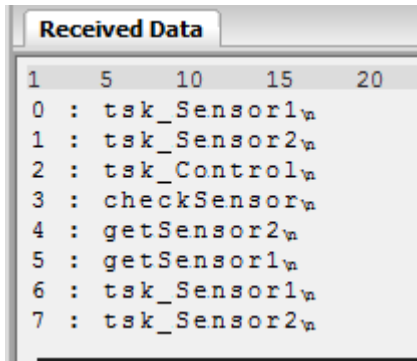
    //Simualted sensor call
    printCall("getSensor2");
    sensor2 = getSensor();

    TerminateTask();
}
```

The resulting output is strange at a first glance. We would expect something like

- tsk_sensor1
- getSensor1
- tsk_sensor2
- getSensor2
- tsk_control
- checkSensor

But we get:



1	5	10	15	20
0	:	tsk_Sensor1		
1	:	tsk_Sensor2		
2	:	tsk_Control		
3	:	checkSensor		
4	:	getSensor2		
5	:	getSensor1		
6	:	tsk_Sensor1		
7	:	tsk_Sensor2		

Figure 68 - Output getting/checking the sensor value

How can this be explained? Hint: Getting and releasing a resource are scheduling points for the OS.

As we have seen before, the priority of the tasks as well as the use of resources influence the scheduling order. Whenever the tasks are rescheduled, the task with the highest priority will be activated first. This reduces rescheduling points and the OS system load. However, the resulting calling order might not be as expected. This is especially critical if the tasks represent an Input-Logic-Output cycle, which expects a certain order.

In this example, the control task having the highest priority is called first, which means that at least for the first activation, no valid sensor data will be there. As the order of task activation is controlled by the OS, we have to check the validity of any data before we use it!

Even if the order is strange, no data corruption takes place, because due to the inverted order after the `printf` the task with the highest priority, the control task is activated first. Let's stress the system a bit more by placing a `CyDelay(1)` command in the `getSensor` function:

```
sensor_t getSensor()
{
    sensor_t s = {0,0,0};

    s.value = counter;
    CyDelay(1);
    s.timestamp = counter;
    s.reliability = counter;

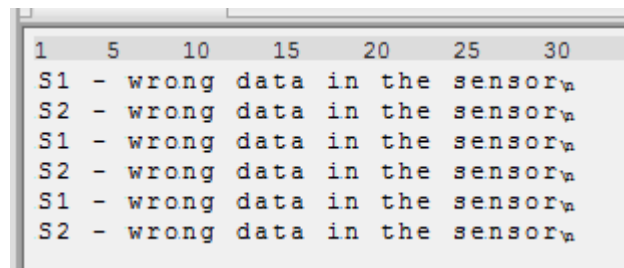
    return s;
}
```

Furthermore, we have to terminate the `printCall` function before the critical section to avoid side effects due to the rescheduling.

```
void printCall(char const * const name)
{
    char t[10] = {0};
    itoa(counter++, t, 10);

    return;
...
}
```

Now, we are getting a lot of wrong sensor data error messages



```
1   5   10   15   20   25   30
S1 - wrong data in the sensor
S2 - wrong data in the sensor
S1 - wrong data in the sensor
S2 - wrong data in the sensor
S1 - wrong data in the sensor
S2 - wrong data in the sensor
```

Figure 69 - Error message wrong sensor data

Let's use the debugger to find out why this has happened:

Watch 1			
Name	Value	Address	Type
sensor1	{ }	0x1FFF8288 (All)	struct { uint32_t timestamp; uint32_t value; uint32_t reliability; }
timestamp	248	0x1FFF8288 (All)	unsigned long
value	245	0x1FFF828C (All)	unsigned long
reliability	248	0x1FFF8290 (All)	unsigned long
Click here to add			

Figure 70 - Wrong sensor value in the debugger

Explain the different values of the sensor structure. Which critical section is causing this problem?

In addition to the obvious critical section which has been identified by you, there is however another critical section, which is a bit more hidden.

How can you fix this? The obvious but unfortunately not sufficient solution would be to place a resource around the structure assignments in the function.

However, the codeline

```
sensor1 = getSensor();
```

looks like a single instruction, but when you check the generated assembly, you might notice the load multiple and store multiple commands, which are not atomic at all.

```

143:
144:    //Simualted sensor call
145:    sensor1 = getSensor();
0x00000236 mov    r0, sp
0x00000238 bl     1a8 <getSensor>
0x0000023C ldr    r3, [pc, #14]    ; (254 <Function_sensor1+0x28>)
0x0000023E ldmia.w sp, {r0, r1, r2}
0x00000242 stmia.w r3, {r0, r1, r2}
146:

```

Figure 71 - Non atomic processing of the return value

I.e. you have to place the resource around every call of the getSensor function.

```

GetResource(res_printf);
//Start of critical section

sensor2 = getSensor();

//end of critical section
ReleaseResource(res_printf);

```

Finding critical sections is far from trivial. Whenever non-atomic global or static data is accessed, either directly or through a pointer, we might have a critical section, which needs to be protected.

Another pattern for critical sections are driver calls which should not be interrupted, like the blocking UART call in the example.

If you activate a transmission buffer for the UART component, you might notice that the console output looks fine also without using resources. The reason for this behavior is the following:

Sending out more than 10 bytes over the serial port will require more than 1ms. But sending them into a buffer will be way faster. Is the system threadsafe? Not really, as soon as within the period required for writing to the buffer a rescheduling appears, the problem pops up again.

4.3.4 Iteration 4 - Using events to improve messaging

One pitfall of the previous solution is the rather un-deterministic order in which the different tasks are called. In our example, the `checkSensor` should be called once the corresponding sensor has been updated. The simplest option would be to call the functions in a classic sequential way.

```
TASK(tsk_control)
{
    sensor1 = getSensor();
    checkSensor("S1", &sensor1);

    sensor2 = getSensor();
    checkSensor("S2", &sensor2);

    TerminateTask();
}
```

Let's assume that the update time of the sensor is asynchronous, i.e. we do not know when the data will come. This can be simulated by pressing a button. If button one is pressed, `sensor1` will be updated, if button 2 is pressed, `sensor2` will be updated.

To achieve a more sequential behavior, we will process the (fast) reading of the sensor in the ISR and use an event to notify the task that the sensor data has been updated.

Please note, that the three OS functions `WaitEvent()`, `GetEvent()`, `ClearEvent()` are typically called one after the other. Furthermore, the event driven task should have a sufficiently high priority to be executed directly upon event occurrence.


```
TASK(tsk_control)
{
    EventMaskType ev;

    while (1)
    {
        WaitEvent(ev_Sensor1Updated | ev_Sensor2Updated);
        GetEvent(tsk_control, &ev);
        ClearEvent(ev);

        if (ev & ev_Sensor1Updated)
        {
            checkSensor("S1", &sensor1);
        }

        if (ev & ev_Sensor2Updated)
        {
            checkSensor("S2", &sensor2);
        }
    }

    TerminateTask();
}
/*****
 * ISR Definitions
 *****/

ISR2(isr_Button)
{
    if (BUTTON_IsPressed(BUTTON_1))
    {
        sensor1 = getSensor();
        SetEvent(tsk_control, ev_Sensor1Updated);
    }

    if (BUTTON_IsPressed(BUTTON_2))
    {
        sensor2 = getSensor();
        SetEvent(tsk_control, ev_Sensor2Updated);
    }
}
```

4.4 OSEK Error Handling and Hook Functions

Effort: 1h	Category - 1
Erika OS, hooks, error handling, error escalation	

In this sample project we will investigate the built in debug and trace features of the OS as well as error handling and escalation options.

The starting point for this project is the simple OS sample we have developed in exercise “Setting up the OS” before. I.e. you can continue working in this project or create a new one based on old one, using the “copy to my template” functionality of the PSOC Creator.

4.4.1 Iteration 1 - Pre and Post Task Hook

On the tap OS Config you can activate the OS hooks you intend to use. An OS hook is an user defined function which can be called upon certain OS events, e.g. upon a task switch or similar.

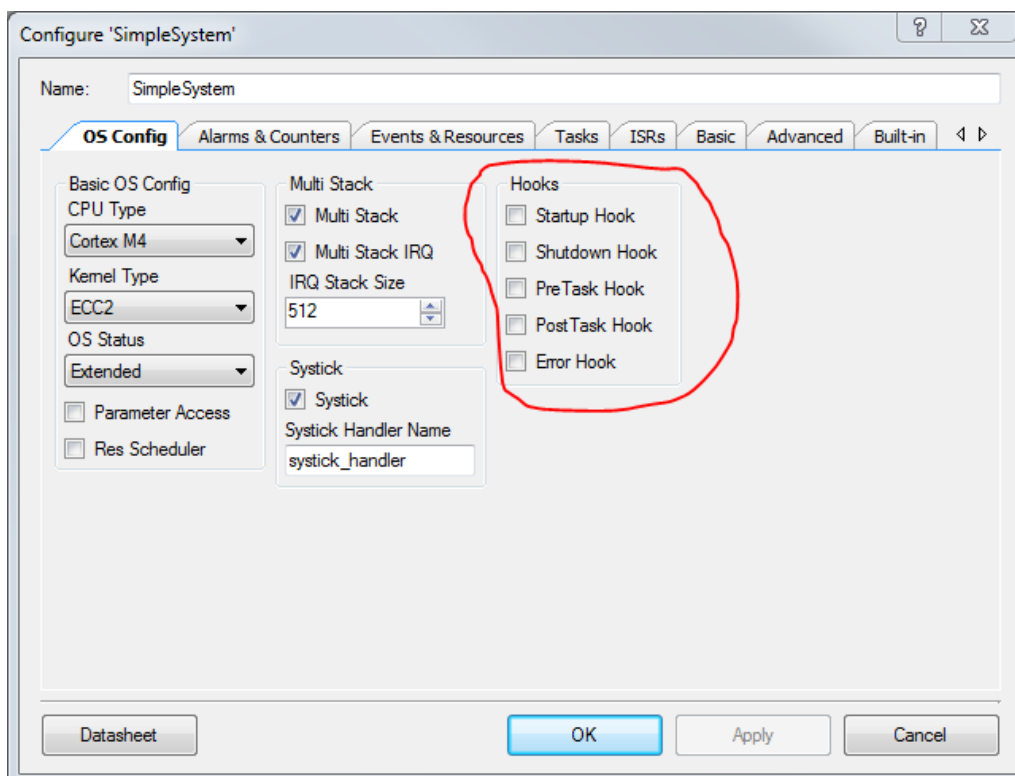


Figure 72 - Hook function activation in the OS Config

Let's start by activating the Post Task Hook.

In the code, disable all calls to the seven segment display, as this will be used to show the last task being called:

```
/* *****  
*****  
* Hook Definitions  
*****  
***** */  
  
void PostTaskHook()  
{  
    TaskType lasttask;  
    GetTaskID(&lasttask);  
  
    SEVEN_SetHex((uint8_t)lasttask);  
}
```

Similar hooks can be defined e.g. for the ShutDownHook and ErrorHook. Check the OSEK specification to find out when these hooks will be called.

4.4.2 Iteration 2 - Adding timestamps for traceability

Another typical use case is to store timestamps together with the task id in a trace buffer (e.g. ringbuffer). Implement such a buffer storing the last 100 task activations, overriding the older entries. Using the debugger, analyse the trace and check if it contains the expected entries. Alternatively use the background task to send the data to the UART, but make sure, that the amount of data can be transmitted physically.

4.4.3 Iteration 3 - Central Error Handler

Instead of leaving the check of OS call returnvalues to the user, it is a good design practice to implement a central error handler using the OS error hook function. This hook function will be called whenever an OS call returns an error. The error code will be passed as a parameter to this function.

Please note that the LOG_PRINT only acts as pseudo code, as a writing operation to the UART using v_args typically consumes too much memory and time.

```
void ErrorHandler(StatusType err)
{
    switch (err)
    {
        case E_OS_ILLEGAL_ADDRESS :
            LOG_PRINT("Illegal Address\n");
            break;
        //...
        default:
            LOG_PRINT("Errorhook: %d\n",err);
            break;
    }
}
```

4.4.4 Iteration 4 - OSShutdown

In case of serious OS errors, the operating system will shut itself down. This shutdown can also be initiated by the application by calling the function ShutdownOS(E_OK). Typically, the parameter E_OK is passed to show that the system was shutdown by the user.

In the shutdown hook, a while(1) loop typically is used to keep the system in a defined state, which may be broken by the watchdog timer.

```
void ShutdownHook(StatusType err)
{
    TaskType lasttask;
    GetTaskID(&lasttask);

    LOG_PRINT("Shut down from task: %d with error %d\n",lasttask,
err);

    while(1)
    {
        asm("nop"); //Wait for the watchdog counter to
                    //reset the system
    }
}
```

5 Reactive Systems / State Machines

Many embedded applications are implemented as state machines. In such architectures, the system will react in a different way depending on the state it is in. Very often, reactive system are implemented without a proper design of the state logic. This might end in complex and therefore unmaintainable if-then-else structure. In this chapter, we will design and implement state machines using the switch case and lookup-table pattern.

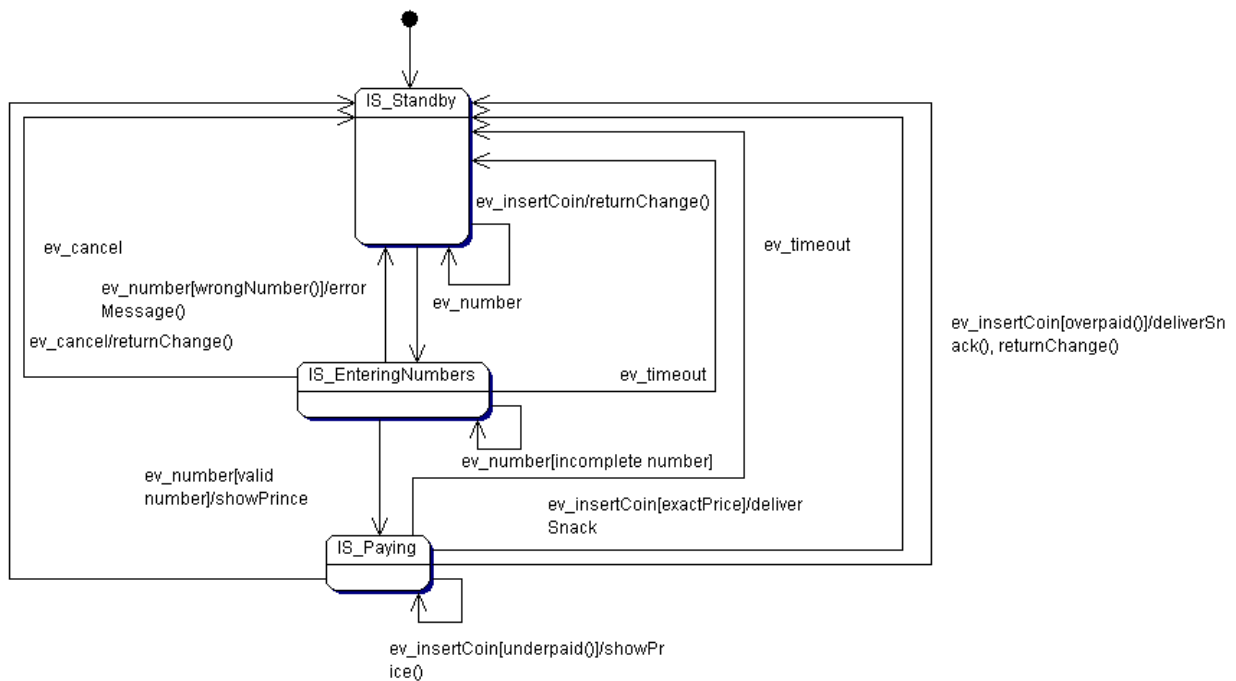


Figure 73 - State Diagram for Snackmachine

5.1 Reaction Game

Effort: 4h	Category - B
Erika OS, events, statemachine design, task design	

Your task is to develop a simple reaction game similar to the one developed in chapter 3 using Erika OS and to extend this functionality with some Arcadian light effects.

5.1.1 Requirements

Your task is to develop an embedded application with the following requirements

Req-Id	Description
NFR1	The application runs as an Erika-OS Application
FR2	Upon startup, the program will show a welcome message via the serial port.
FR3	After pressing one of the two white buttons, the program will wait for a random time. After waiting for 1s to 3s a random value (1 or 2) will be displayed on both 7segment displays
FR4	The user has to press the right button in case a '1' is displayed and the left button in case a '2' is displayed
FR5	In case the correct button is being pressed, the measured reaction time in [ms] will be shown and the game can be started again by pressing one of the two buttons.
FR6	In case a wrong button is pressed, an error message will be displayed and the game can be started again by pressing one of the two buttons.
FR7	In case the user does not press a button within 1 s, the message "Too slow" will appear and the game can be started again by pressing one of the two buttons.
FR8	One game consists out of 10 rounds
FR9	At the end of a game, print the score (i.e. correct number of button pressed), the total time and the average time
NFR10	Use the switch-case pattern to implement the state machine. Use private functions for the actions and guards.
NFR11	Write the code of the reaction game logic in an own file
NFR12	Provide good comments and self-explaining variable und function names.
NFR13	Follow the coding rules mentioned in the provided code file template.

Example output via the serial port:

```
Reaction test program round 1
press one of the two buttons to start...
```

```
Great - correct button pressed
```

```
Reaction time in ms: 249
```

```
=====
```

5.1.2 Analysis

Events are input signals which can be fired by the user or by the system to initiate a state change.

Example:

- After pressing button1 or button2, the program will wait for a random time.

Which events are mentioned in this requirement?

Event 1: _____

Event 2: _____

Which state transition is mentioned in this requirement?

In addition to the two user input events, two additional system events can be identified from the requirements.

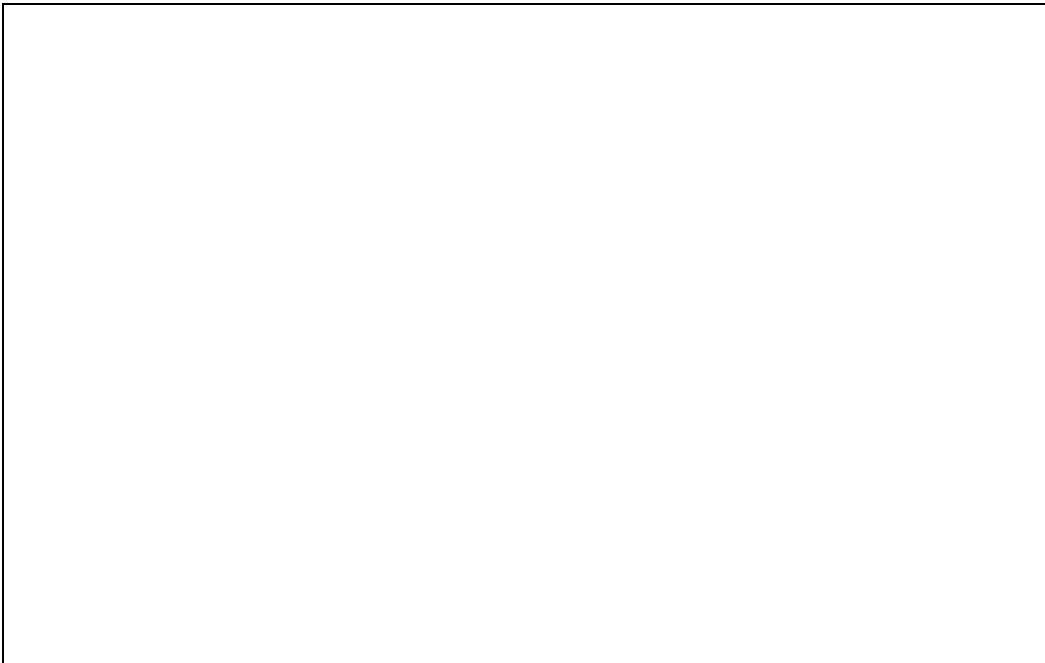
Event 3: _____

Event 4: _____

5.1.3 Erika elements

Alarms can be used to implement time signals. They can be compared with timer interrupts but provide more flexibility. Furthermore, several alarms can share one physical timer resource.

Provide the code snippets which will create the 1ms tick for the base counter.

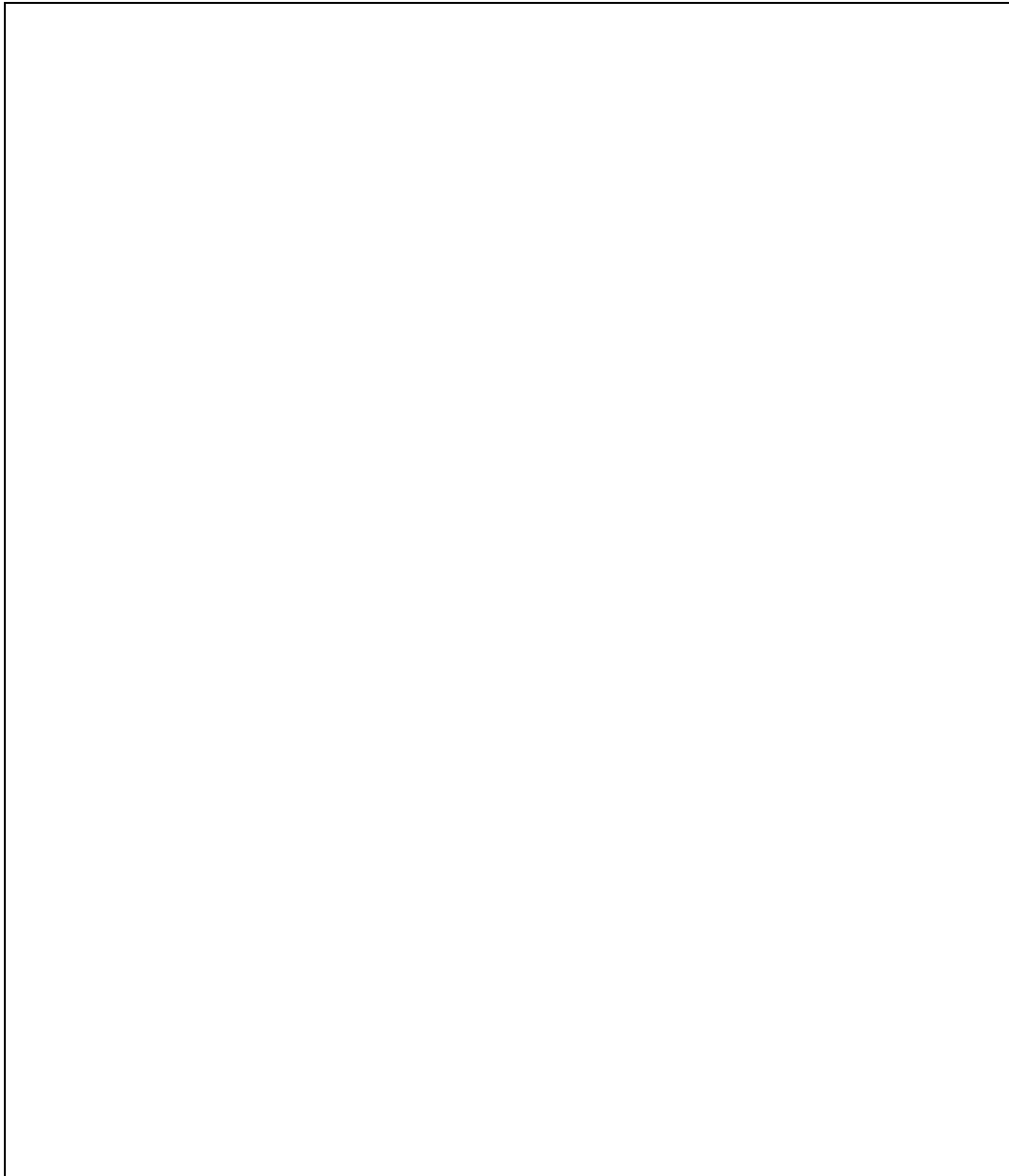


In our application, we will use the alarm `alarm_Tick1ms` which will be fired every 1ms. Every time the alarm is fired, the task `tsk_Timer` will be called. Alternatively, you may use a callback handler for this.


Check the API call `SetRelAlarm` and provide the code to configure the alarm to be fired every 1ms.

`SetRelAlarm(_____, _____, _____)`

Alternatively, an alarm can also be configured to be fired only once. Where might we need this feature and how do you configure this?



Add ONE ISR function for the button pressed. Will this function be an ISR1 or ISR2 category ISR? Explain.



5.1.4 State Maschine

Draw a state machine of the system.

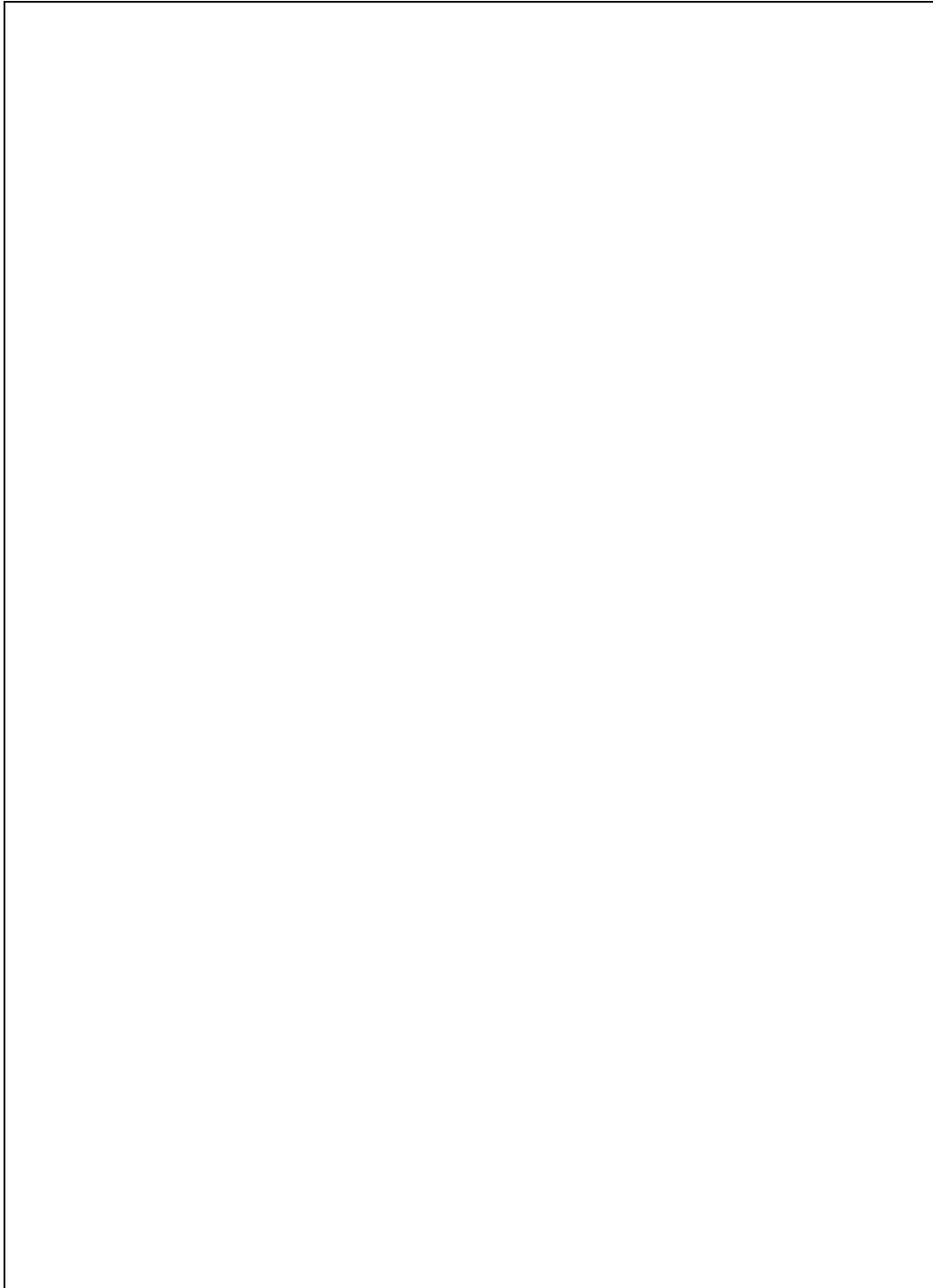
Describe the states using the **Is<WaitingforSomething>** mini sentence convention.

Use the following convention to describe a transition between two states:

Event [optional guard] / Action

A guard is an additional condition which is checked when an event occurs. In our game, this could be for example a check, if the correct button has been pressed. Guards are very often functions returning a Boolean value.

State Machine of the reaction game:



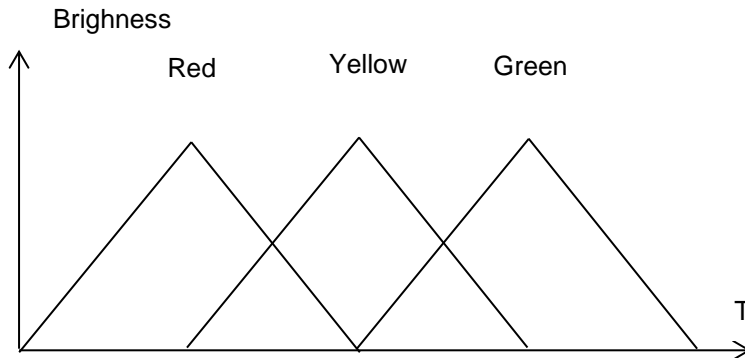
Implement the code for the program based on the given requirements and design.

5.1.5 Arcadian Style

In order to make the game a bit fancier, we want to add some Arcadian style light effect. The light effects may not interfere with rest of the functionality.

Fader:

In a first step, we want to create a fading traveling light. The three LED's are glowing using the following pattern:



You might want to check some 80ies Knightrider movies to get into the groove, e.g.

<https://www.youtube.com/watch?v=Mo8Qls0HnWo>

Glower:

Using the RGB LED, we want to implement an easily configurable glowing function. Using a const - table like the following (pseudo code)

```
const RG_Glow_t RG_glowtable_1[] = {
//Red    Green  Blue   TimeInMS
{255,    0,     0,     500},
{0,255,   0,     500},
{0,0,     255,   500},
{0,0,     0,     100},
{255,    255,   255,   100},
{0,0,     0,     100},
{255,    255,   255,   100},
{0,0,     0,     100},
{255,    255,   255,   100}
};
```

will create the sequence:

- 500ms red
- 500ms green
- 500ms blue

- 100ms off
- 100ms white
- 100ms off
- 100ms white
- 100ms off
- 100ms white

This sequence will be repeated permanently.

Provide a good declaration for `RG__Glow_t`



How many tasks do you need for the new Arcadian functions? Explain.



5.2 MP3 - Player

Effort: 8h	Category - B
Erika OS, events, statemachine design, task design, lookup table pattern	

Your task is to implement as simple MP3 player statemachine.

5.2.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
FR2	Button 1..4 will be used to control the player.
FR3	In case of the standby mode, the buttons will have the following behavior: <ul style="list-style-type: none"> • Button 1 - select next song • Button 2 - select previous song • Button 3 - start to play the selected song
FR4	In case of the playing mode, the buttons will have the following behavior: <ul style="list-style-type: none"> • Button 1 - increase volume • Button 2 - decrease volume • Button 3 - pause the song (press again to resume) • Button 4 - stop the song and go back to standby mode
FR5	If one song has ended, the next song will start playing automatically.
FR6	The remaining playing time of a song will be shown on the right seven segment display
FR7	The volume will be shown on the left seven segment display
FR8	The playlist as well as the currently selected song will be shown via the UART or on the onboard display (optional)
NFR9	Use the switch-case pattern to implement the state machine. Use private functions for the actions and guards.
NFR10	Write the code of the MP3 player logic (and additional classes) in own files
NFR11	Provide good comments and self-explaining variable und function names.
NFR12	Follow the coding rules mentioned in the provided code file template.
NFR13	Use structures to encapsulate the data of a file ((OO design), e.g. the class MP3 player structure will contain a state variable, a volume, a selected song....

5.2.2 Analysis and Design

Draw the state machine based on the given requirements. Clearly identify guards and actions.

A large, empty rectangular box with a thin black border, intended for drawing a state machine diagram. The box is currently blank.

5.2.3 Iteration 1 - Initial implementation

Based on the pattern presented in the lecture, implement the state machine logic using the switch case pattern. In this iteration, we will simulate the playing of the song by simply showing the song title via the UART.

Sub-Iteration A

- Configure the OS
- Create Tasks, Activation Code
- Create empty `statemachine.run(event)` method / function
- Add a `log(event, state)` function to the empty state machine, which will print the event and state via the USART
- Test – Check if all events are fired correctly

Sub-Iteration B

- Add empty action and guard function as file static (private) functions
- Implement the state machine logic
- Test – Check if all transitions are correctly implemented

Sub-Iteration C...n

- Add the code to the actions and guards required for a single state and/or transition
- Test the behavior

5.2.4 Iteration 2 - Lookup Table

Implement the logic of the state machine using the lookup table pattern. You may add the lookup-table run function and transition table to the already existing MP3 files.

5.2.5 Iteration 3- Adding real song data (optional)

Download the provided song data (or create own data) and implement a sound functionality. The provided sound data stores the amplitude of the data in an 8-bit format, having a sample rate of 22kHz. As the frequency is pretty high, you might want to use a timer interrupt for creating the sound signal.

5.3 Smart Volume Control

Effort: 3h	Category - B
Erika OS, events, statemachine design, dynamic time based events	

Your task is to implement a smart volume control.

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
FR2	If you press the button 1, the volume will be increased by 1
FR3	If you press the button 2, the volume will be decreased by 1
FR4	The volume value is limited to the range 0...99
FR5	If you keep the button 1 pressed the following will happen: <ul style="list-style-type: none"> • Every second, the volume will be increased by 1 • After having pressed 3 seconds, the volume will be increased by 1 every 0.5 seconds • After having pressed for another 3 seconds, the volume will be increased by 1 every 0.1 seconds
FR6	The same functionality is implemented for button 2 for decreasing the volume
FR7	If you release the button at any time, the sequence will start with the slow speed again
FR8	If no button is pressed, the volume value will not change.
FR9	If the maximum value is reached, no further increase is possible.
FR10	If the minimum value is reached, no further decrease is possible.
NFR11	The exceptional behavior of both buttons being pressed at the same time may be ignored.
FR12	The volume will be shown on the seven segment display

5.4 Electronic lock

Effort: 3h	Category - B
Erika OS, events, state machine design, dynamic time based events	

Your task is to implement an electronic lock. The user has to press the buttons 1..4 in a correct sequence in order to open the lock. In case of an error, the lock will block.

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
FR2	The lock is coded with a variable sequence of the values 1..4, e.g. in case the user selected a length of 4, possible sequences include 1234 or 1221 ...
FR3	Upon power on, the user is asked to enter the sequence (yellow led is on).
FR4	If the correct sequence has been given the lock will open (green led is one).
FR5	In case of a wrong button, the red led will be turned on and the system will block for <ul style="list-style-type: none">• 5s after the first wrong input• 10s after the second wrong input• permanently after the third wrong input

6 Embedded Architectures

More and more embedded architectures try to separate application software from basic software. This facilitates reuse and allows an easier use of code generation tools like Matlab Simulink. A prominent example of this separation concept is the Autosar RTE.

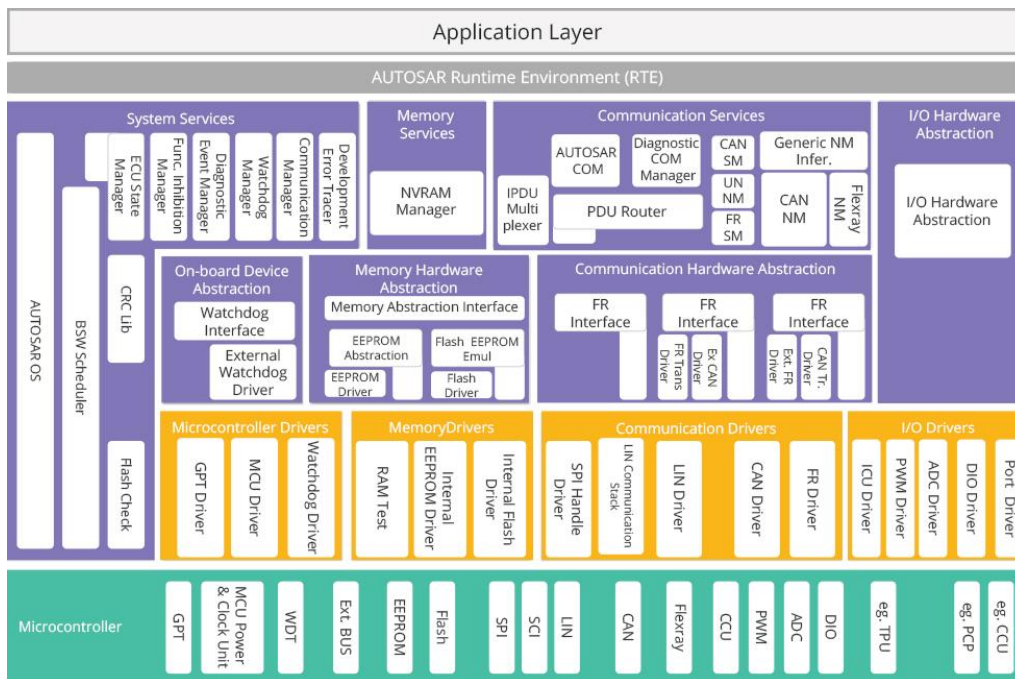


Figure 74 - Autosar RTE⁷

⁷ <https://www.mobiliya.com/industries/automotive/autosar>

6.1 A light version of the Autosar RTE - Electronic Gaspedal

Effort: 8h	Category - B
RTE, Autosar, Software Components, Runnables, Activation Concepts	

In this exercise you will develop an ECU following the Autosar RTE concept. In the concept, the applications software and basic software (drivers and OS) are separated by an intermediate layer called RTE, which provides data containers for a message based communication between runnables and manages events (cyclic and data), which will activate the various runnables. **The runnables represent the user code. They only communicate via RTE signal objects, hardware driver and OS calls are not allowed.** This separation facilitates re-use over projects and controller boundaries.

The picture below illustrates the basic functionality of the system. Please note, that the focus of this exercise is the RTE, therefore the runnables are very simple.

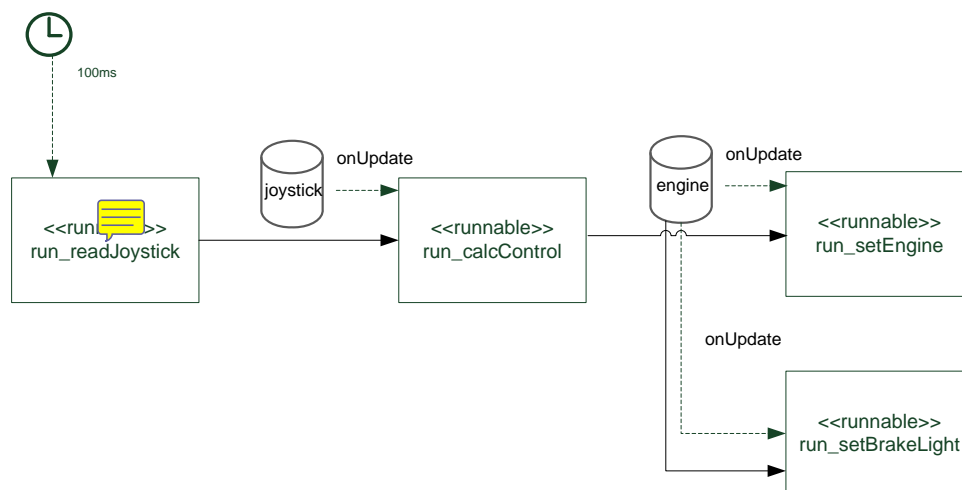


Figure 75 - Electronic Gaspedal ECU

the runnable `run_readJoystick` will be called every 100ms. This runnable will update the `joystick` signal, using the API `pullPort()`. As this signal is realized as asynchronous signal (details see below), an event will be fired upon update.

This event will trigger the runnable `run_calcControl`. This runnable will check the value of the `joystick` signal (datatype `sint8_t`). If the value is bigger than 0, the `engine` signal (`uint8_t`) will be set to 2 x the `joystick` value. If it is below or equal to 0, the `engine` value will be 0.

Once this value is updated, the two output runnables will be triggered.

`run_setEngine` will send the engine signal to the driver (`pushPort`), whereas `run_setBrakeLight` will check the engine value. In case the engine value is 0, the

signal `brakeLight` will be turned on, in case of a value bigger than 0 it will be turned off. The `brakeLight` signal then is also send to the hardware.

6.1.1 Iteration 1 - Configuration of the RTE

Download the RTE Generator from the Moodle page. You need Python to run it - <https://www.python.org/downloads/>

In the folder `\in` you will find a file `ecu.xml`. This files serves as input for the generator. You can use Eclipse or Notepad++ to edit this file. The folder `\out` will contain the generated files.

Signal classes describe the datatype or interface of the different signal objects. As our signals are closely related to peripherals, we will use the peripheral names as identifiers.

The signal objects themselves describe the abstract communication interfaces of the runnables.

- The config section of the file does not need to be changed
- In the section `signalclasses`, a first class `adc` already has been added. Add 2 more classes `pwm` and `gpio` which will be used for the `engine` and `brakeLight` signal objects. The `pwm` class will trigger the output runnable and therefore should be asynchronous. The `gpio` class will not trigger any other runnable and therefore can be synchronous.
- In the section `signalobjects` the individual objects (global data) will be defined. Again, use the `joystick` signal as a starting point for the 2 other signal objects `engine` and `brakeLight`. For the drivers, you may set the value to default, which will create a default API in the `type.h|.c` files. For the engine, don't forget to define a `onUpdate` Event.
- In the section `runnables`, add a runnable `run_readJoystick` to the cyclic block and the other three runnables into the event block

Run the generator using the provided batch script. The output of the script will be written into the file `log.txt`. You should see something similar to the output below. In case you see error messages, you very likely made some mistakes when editing the `xml` file.

```
RTE Generator Erika 1
Processing: in/ecu.xml based on version Erika 1
Checking the configuration...
Creating the signalclasses...
  class: .\out\adc_signal.h
  class: .\out\adc_type.h
  class: .\out\adc_type.c
  class: .\out\pwm_signal.h
  class: .\out\pwm_type.h
  class: .\out\pwm_type.c
  class: .\out\gpio_signal.h
  class: .\out\gpio_type.h
  class: .\out\gpio_type.c
Creating the signal object pool...
  Writing Header File...
    Object: joystick
    Object: engine
    Object: brakeLight
  Writing Source File...
    Object joystick
    Object engine
    Object brakeLight
Creating the signal activation engine...
  Writing Source File...
```

Go to the `\out` folder and analyze the generated files! Check the provided comments to understand the task of every file.

6.1.2 Iteration 2 - Add the RTE to your project

Create a new Erika project and add a new folder `\rte` to your `\source` folder. Copy the generated files to this folder and also add the file `rte.h` from Moodle to this folder.

Create a Erika project having the following resources:

- `tsk_Init()`
- `tsk_Background()`
- `tsk_EventDispatcher()` - this task will receive all RTE events and call the corresponding runnables.
- `tsk_CyclicDispatcher()` - this task will be triggered every 10ms and call the cyclic runnables.
- Add a 10ms alarm for the cyclicDispatcher task
- Add events for all onUpdate events you have added to your project.

Implement the code for the functions of `rte.h`. The functions should call all runnables listed in the generated tables in the file `rte_activation.c`.

6.1.3 Iteration 3 - Getting it compilable

Complete the declaration in the `*_type.h` files and add some skinny sheep code to the corresponding source files.

Add a file `swc_application.h|.c` to your `\asw` folder containing some empty implementations for the runnables.

6.1.4 Iteration 4 - Getting it running

In the code for the joystick driver, read the joystick ADC value using the driver implemented in exercise 2.1. I.e. the middle position represents the value 0, the right and upper point the value 127 and the lower and left point the value -128.

The pwm driver should write the engine value to the RGB LED, using the green channel.

The gpio driver should write a ON signal to the red LED in case the brakeLight is on (TRUE), or an OFF signal in case the signal is off (FALSE).

Now let's add the code to the runnables as described in the initial chapter of this exercise.

It is recommended to start with the joystick runnable and to use e.g. the `UART_LOG` port to create some verbosity.

6.1.5 Iteration 5 - Extensions (optional)

The RTE has some limitations, which may be addressed if you want to dive a bit deeper into this topic.

More signals

Instead of only using the joystick to differentiate between driving and braking, you might want to add a brakePedal signal object, which is set by an ISR in case a button is pressed.

Critical Sections

At the moment, the data of the signal objects is not protected against concurrent access. One option would be to add OS-Resources to the signal, which will protect the access. For this, the XML format and Python generator needs to be extended.

6.2 Timing Supervision

Effort: 4h	Category - B
Watchdog, hardware manuals, startup, alive monitoring, deadline monitoring	

Your task is to implement a timeout supervision concept for the PSOC. In case the system comes to a halt, e.g. caused by an endless loop or shutdown of the OS, a reboot should be initiated. This concept is called alive monitoring. As this is a pretty brute force error handling, an additional deadline monitoring shall be implemented, which supervises individual runnables.

6.2.1 Hardware Watchdog Driver

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
NFR2	Use existing functions of the system as far as sensible and possible.
NFR3	Info: The watchdog is an architectural (fixed) functionality of the Cortex. I.e. there is no component available, instead you should check the architectural manual and create startup code.
FR4	Create a new software driver called watchdog which provides the functions described below.
FR5	<pre>/** * Activate the Watchdog Trigger * \param WDT_TimeOut_t timeout - [IN] Timeout Period * @return RC_SUCCESS */ RC_t WD_Start(WDT_TimeOut_t timeout);</pre>
FR6	Define a sensible enum value for the timing period, based on the provided hardware functionality.
FR7	<pre>/** * Service the Watchdog Trigger * @return RC_SUCCESS */ RC_t WD_Trigger();</pre>
FR8	<pre>/** * Checks the watchdog bit * @return TRUE if watchdog reset bit was set */ boolean_t WD_CheckResetBit();</pre>
FR9	Develop a first test framework, fulfilling the following requirements:
FR10	Initialise the watchdog trigger with the longest period.

FR11	Trigger the watchdog in the background task
FR12	By receiving an event (button or uart), call the OS shutdown function
FR13	Upon startup, show (via the UART_LOG), if the system was rebooted after a power on reset (POR) or after a watchdog reset.

6.2.2 Alive Watchdog

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
NFR2	Use the driver implemented in the first exercise and integrate it into the code of the Electronic Gaspedal Exercise.
FR3	Add a global bitfield to the driver which is initiated with the value {0}
FR4	Add a function <code>WD_Alive(uint8_t myBitPosition)</code> to the driver, which sets the bit at the corresponding position.
FR5	This function shall be called by every runnable using a unique position, i.e. <code>Runnable_0</code> sets bit at position 0, <code>Runnable_1</code> sets bit at position 1 and so on.
FR6	In the background task, the <code>WD_Trigger()</code> function will be called if all bits are set, i.e. all runnables reported their alive status. Furthermore, all bits are cleared.
FR4	Add this concept to your system and implement testcases to verify the functionality.

6.2.3 Deadline Monitoring (optional)

Alive monitoring typically serves as a last line of defense, as the system can only react with a rather harsh reaction like reset.

In order to control the timing a bit less harsh, deadline monitoring can be applied, by checking the runtime of a single runnable or group of runnables.

The concept would be as follows:

- Before calling the runnables in a task, an alarm will be started.
- During normal operation, all runnables will be finished before the alarm elapses and the alarm can be cancelled.

- If one or more runnables take too long, the alarm will be fired and an error handling task will be activated, which e.g. can disable certain less critical runnables.

Try to implement this concept and supervise the runtime of the cyclic and event triggered runnables of the exercise Electronic Gaspedal.

6.3 Communication Stack

Uart send als action, UART receive als Event für Protokoll Zustandsautomaten

7 Real Fun

7.1 Morse coder/decoder

7.2 The game of PONG

7.3 Distributed version SOCCER

8 Annex - PSOC and LabBoard Pinning

Table 4-1. J1 Header Pin Details

PSoC 5LP Prototyping Kit GPIO Header (J1)		
Pin	Signal	Description
J1_01	P2.0	GPIO
J1_02	P2.1	GPIO/LED
J1_03	P2.2	GPIO/SW
J1_04	P2.3	GPIO
J1_05	P2.4	GPIO
J1_06	P2.5	GPIO
J1_07	P2.6	GPIO
J1_08	P2.7	GPIO
J1_09	P12.7	GPIO/UART_TX
J1_10	P12.6	GPIO/UART_RX
J1_11	P12.5	GPIO
J1_12	P12.4	GPIO
J1_13	P12.3	GPIO
J1_14	P12.2	GPIO
J1_15	P12.1	GPIO/I2C_SDA
J1_16	P12.0	GPIO/I2C_SCL
J1_17	P1.0	GPIO
J1_18	P1.1	GPIO
J1_19	P1.2	GPIO
J1_20	P1.3	GPIO
J1_21	P1.4	GPIO
J1_22	P1.5	GPIO
J1_23	P1.6	GPIO
J1_24	P1.7	GPIO
J1_25	GND	Ground
J1_26	VDDIO	Power

Table 4-2. J2 Header Pin Details

PSoC 5LP Prototyping Kit GPIO Header (J2)		
Pin	Signal	Description
J2_01	VDD	Power
J2_02	GND	Ground
J2_03	RESET	Reset
J2_04	P0.7	GPIO
J2_05	P0.6	GPIO
J2_06	P0.5	GPIO
J2_07	P0.4	GPIO/BYPASS CAP
J2_08	P0.3	GPIO/BYPASS CAP
J2_09	P0.2	GPIO/BYPASS CAP
J2_10	P0.1	GPIO
J2_11	P0.0	GPIO
J2_12	P15.5	GPIO
J2_13	P15.4	GPIO/CMOD
J2_14	P15.3	GPIO/XTAL_IN
J2_15	P15.2	GPIO/XTAL_OUT
J2_16	P15.1	GPIO
J2_17	P15.0	GPIO
J2_18	P3.7	GPIO
J2_19	P3.6	GPIO
J2_20	P3.5	GPIO
J2_21	P3.4	GPIO
J2_22	P3.3	GPIO
J2_23	P3.2	GPIO/BYPASS CAP
J2_24	P3.1	GPIO
J2_25	P3.0	GPIO
J2_26	GND	Ground

Table 4-3. Pin Details of J7 Header

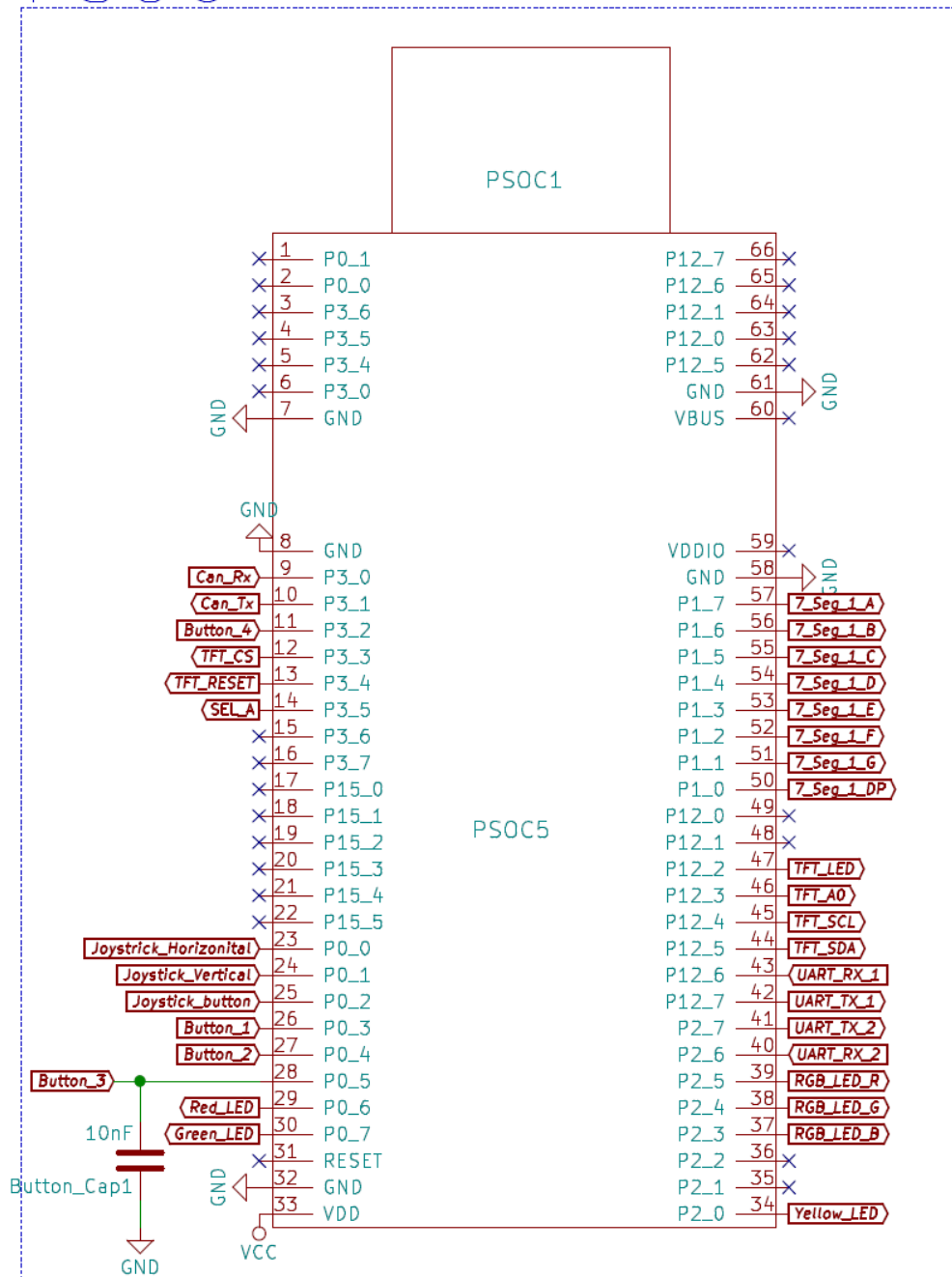
PSoC 5LP to KitProg Header (J7)		
Pin	Signal	Description
J7_01	VDD	Power
J7_02	GND	Ground
J7_03	P12.4	PROG_XRES
J7_04	P12.3	PROG_SWCLK
J7_05	P12.2	PROG_SWDIO

Table 4-4. Pin Details of J3 Header

PSoC 5LP Prototyping Kit GPIO Header (J3)		
Pin	Signal	Description
J3_01	VTARG	Power
J3_02	GND	Ground
J3_03	XRES	XRES
J3_04	P1.1	SWCLK
J3_05	P1.0	SWDIO

Figure 76 - PSOC Pin Header

PSOC



9 Annex - Firmware Update

Depending on the PSOC creator version you are using, an update of the programmer firmware is required. For this, please start the PSOC Programmer (available in your Windows / Cypress menu)

- Click on the board you want to update (usually only one is available, unless you have connected more boards)
- Switch to the Utilities tab and press the button “Upgrade firmware”

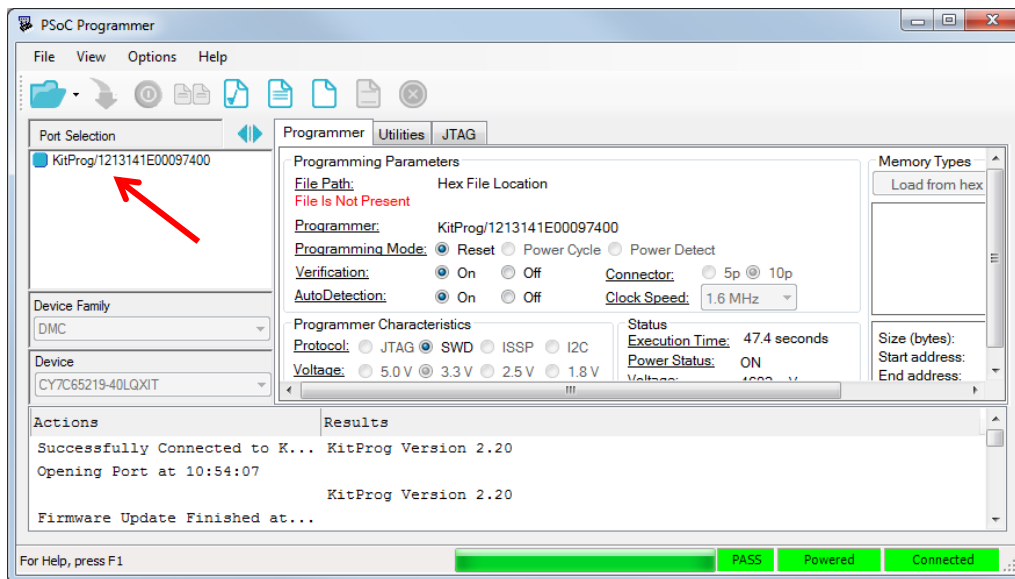


Figure 78 - Firmware Upgrade

10 Annex - Using Doxygen

Doxygen can be downloaded from the website <http://www.doxygen.nl/>

The easiest way to use it is through the Wizard which is coming with the package. Alternatively you may use it as a command line tool and e.g. add it to your build process.

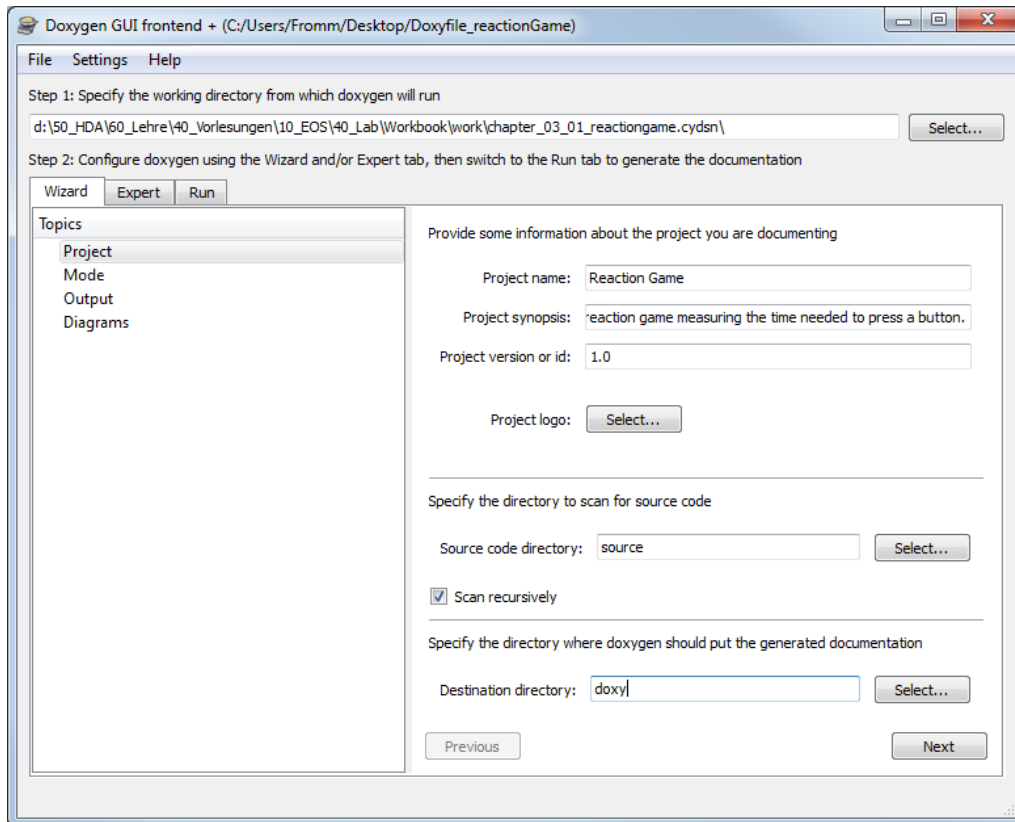


Figure 79 - Doxywizard Start Panel

In the field working directory (Step 1) enter the location of your project.

The other fields are rather self-explaining:

- Select the top level source directory of your project as source folder and check “Scan recursively” to make sure that all files are documented
- As output directory, it is recommended to select a folder on the same level or higher as your source folder, to avoid, that the generated html documentation is documented again.

Once you are done with setting the other options, press the run button in the run-tab and click on Show HTML Output once the execution has finished.

Use the file menu to save and load your configuration.

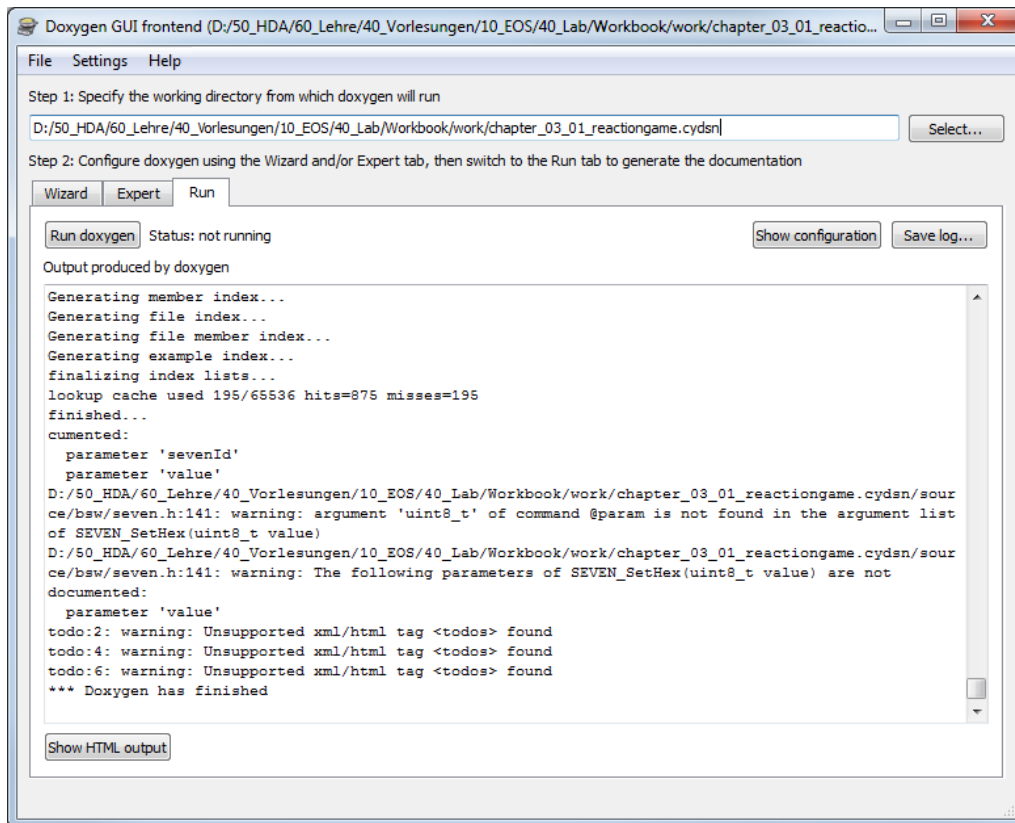


Figure 80 - Run Doxygen

Once you have saved your doxyfile, you can create the following batchfile (called postbuild.bat) and place it in your postbuild command to automatically generate the documentation and open it in Firefox

```

REM call Doxygen with default configuration file
doxygen

REM open firefox with the generated documentation
start firefox.exe file:///~dp0/doxy/html/index.html

```

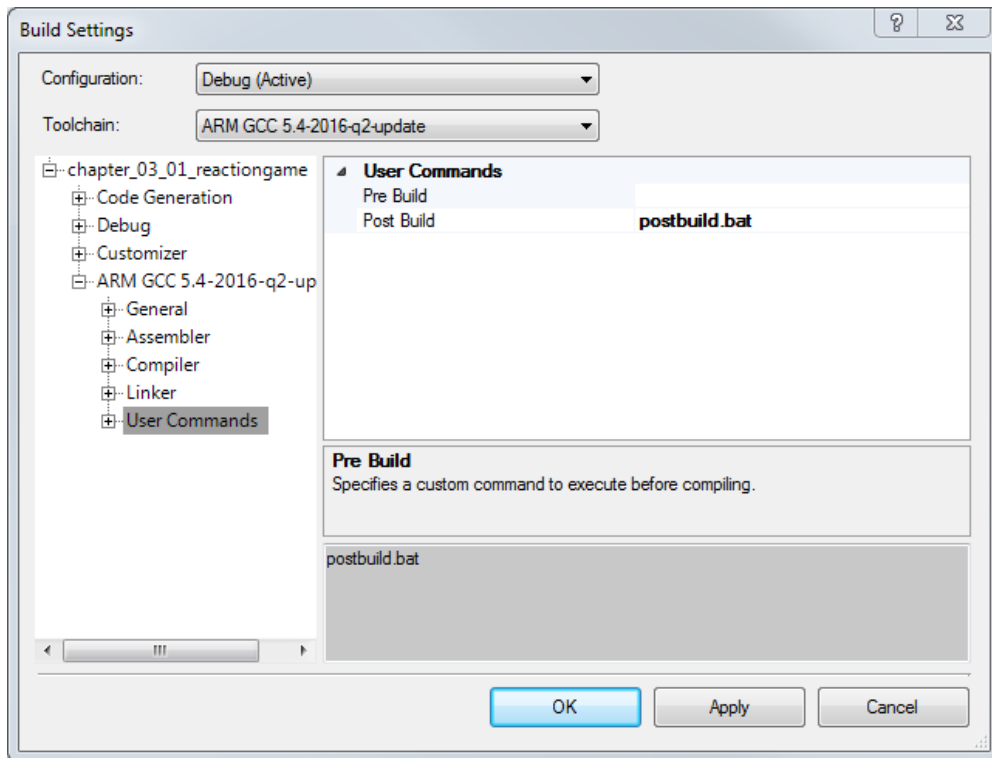


Figure 81 - Autogenerated documentation

