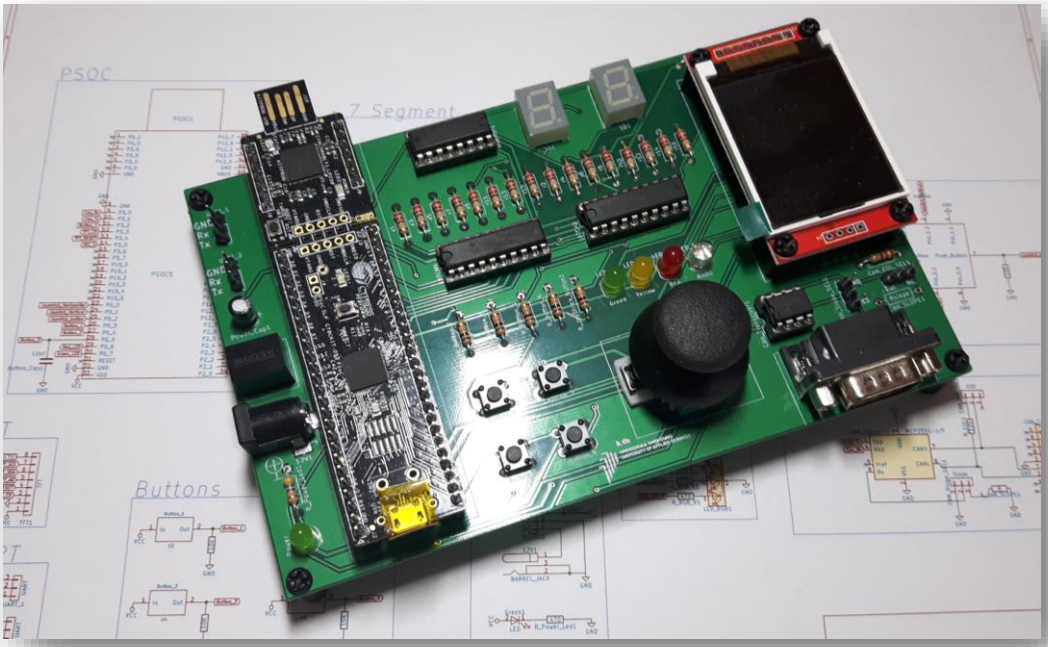
Embedded Architectures and Operating Systems

Workbook (v2025a)

Prof. Dr.-Ing. Peter Fromm



Copyright

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

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.

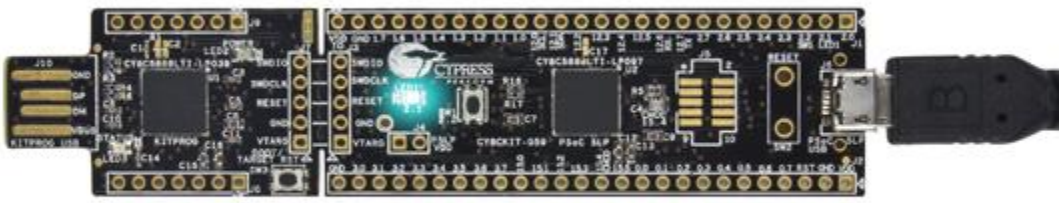


Figure 1 - PSOC5LP

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 since 2017/2018 who own credits for the work.

- Aaron Correya developed 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.
- Sandeep Raju and Sameer Chilmattur ported the Tracealayser to Erika
- Marco Seiller created the RTELight Editor and Generator
- 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.

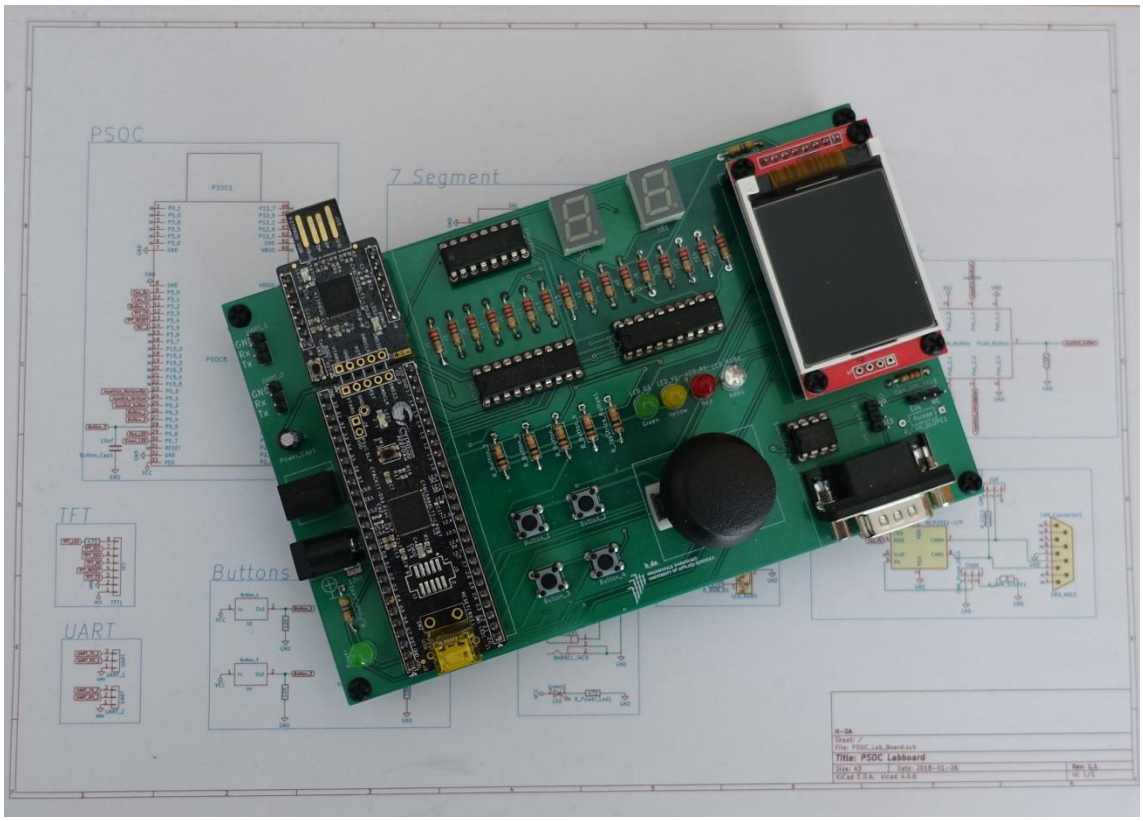


Figure 2 - The PSOC LabBoard

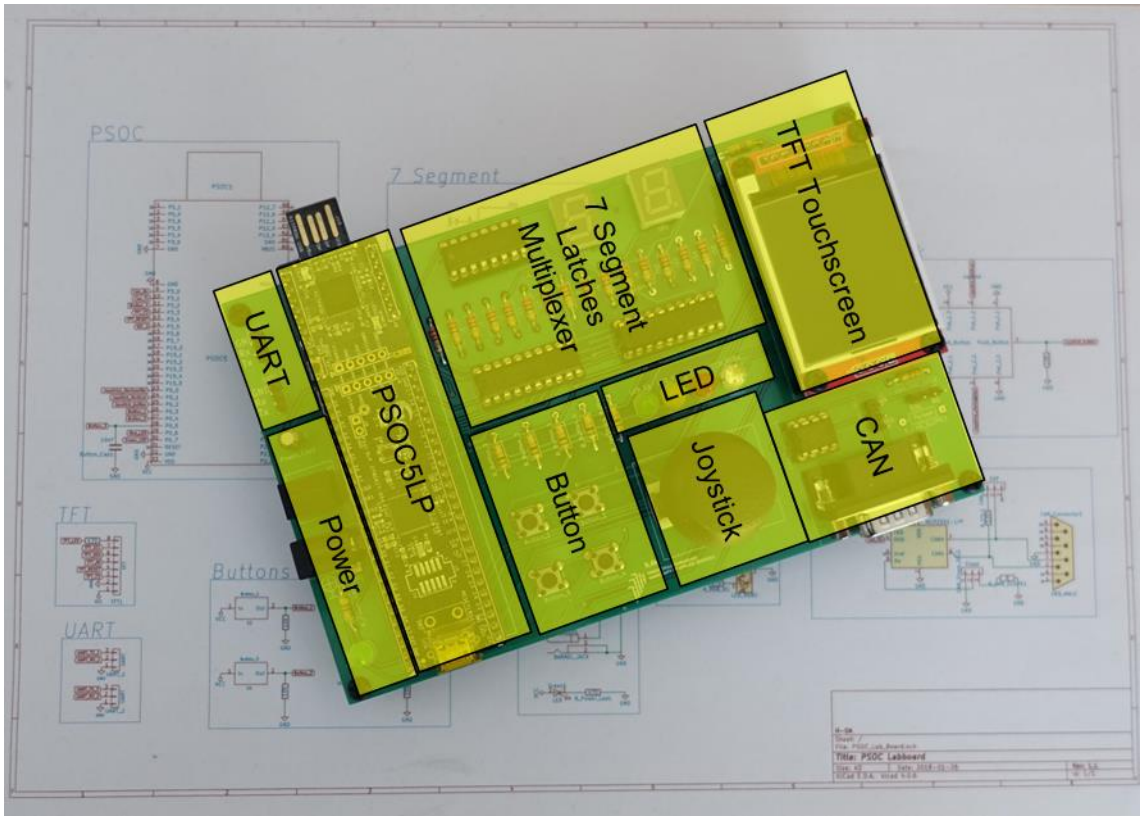


Figure 3- Location of the different functional groups

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.

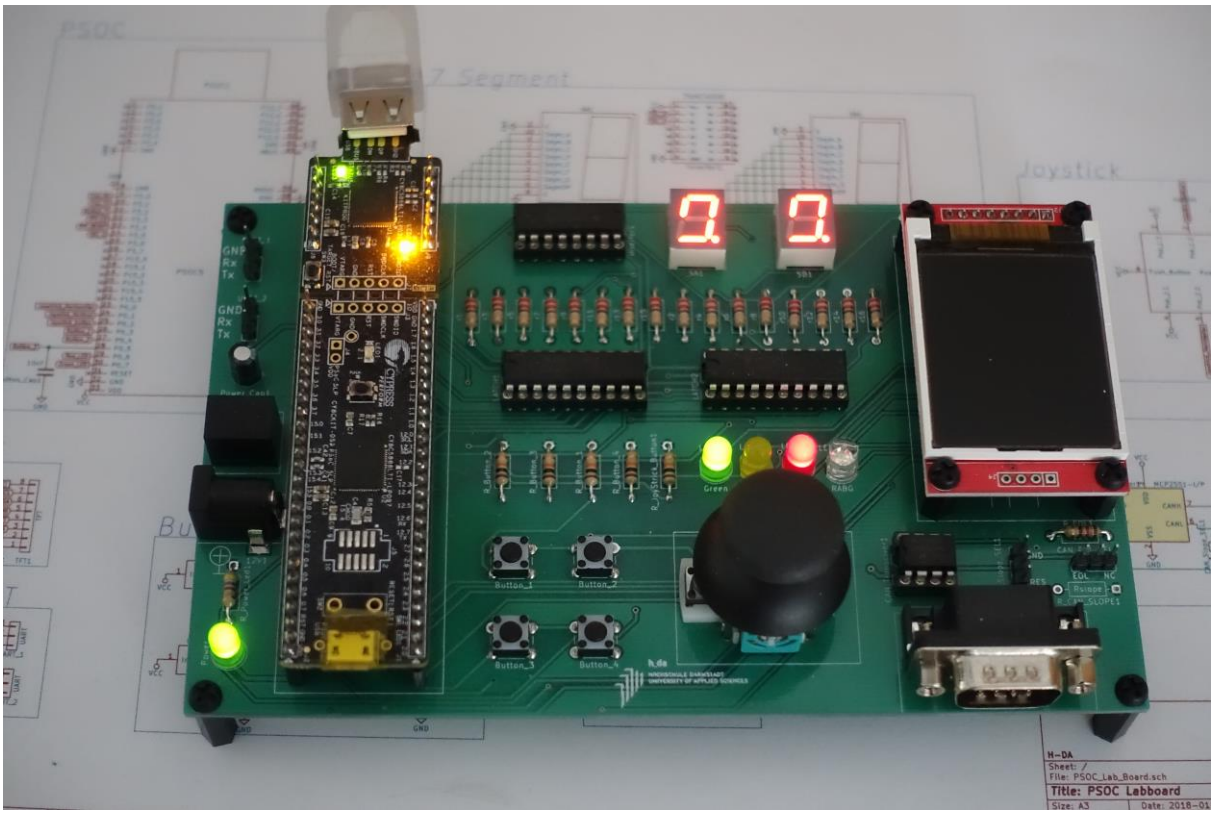


Figure 4 - The LEDs become alive

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.

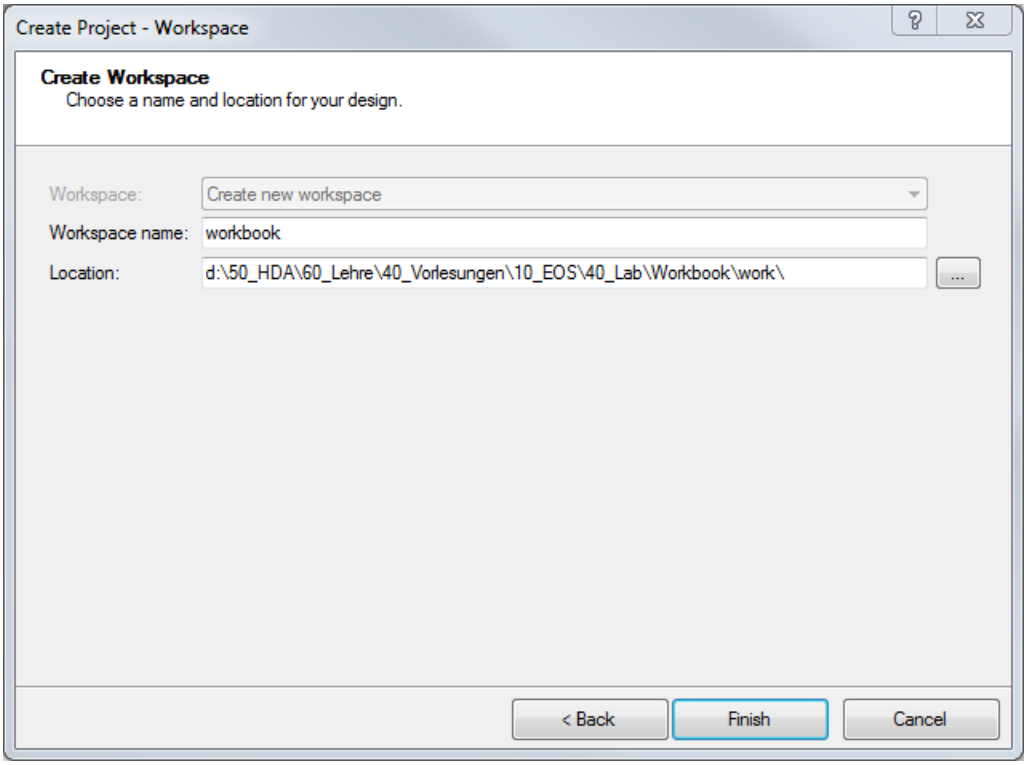


Figure 5 - Creating a Workspace

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.

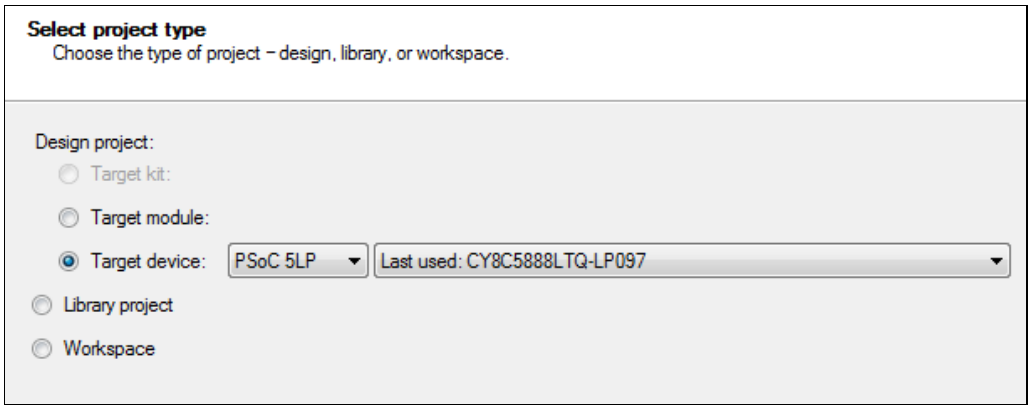


Figure 6 - Creating a project

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

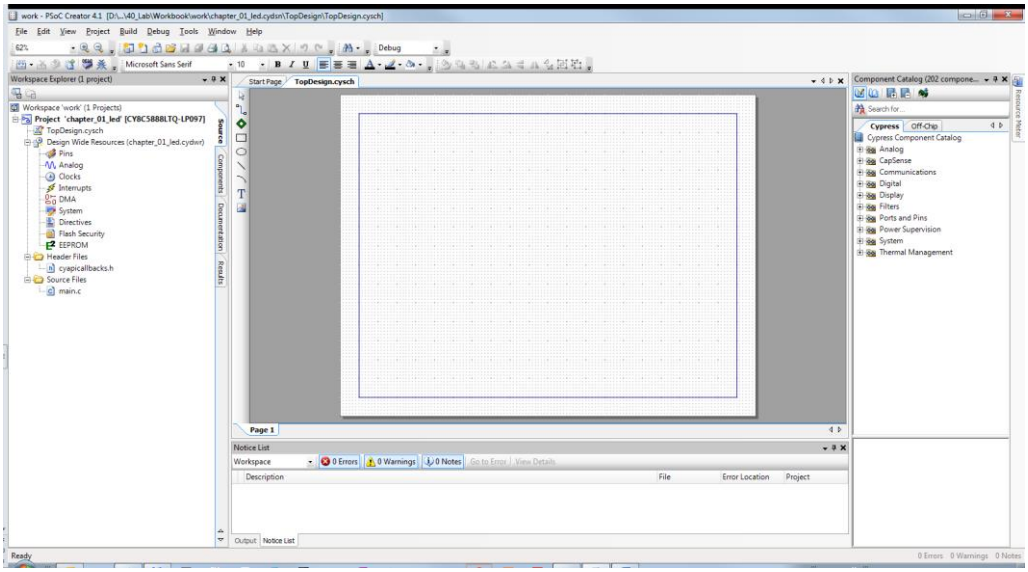


Figure 7 - Empty project

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.

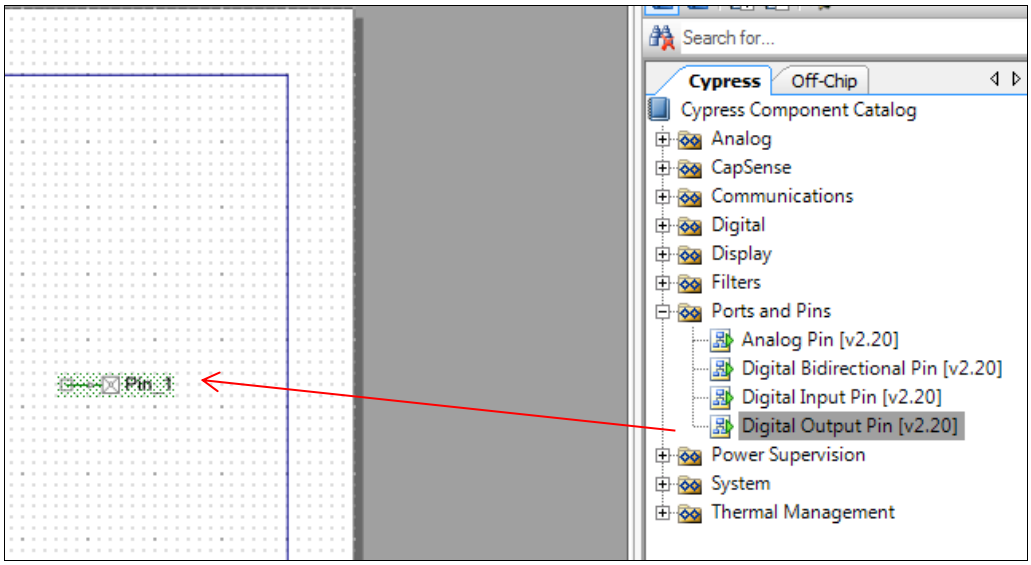


Figure 8 - Adding a first component

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

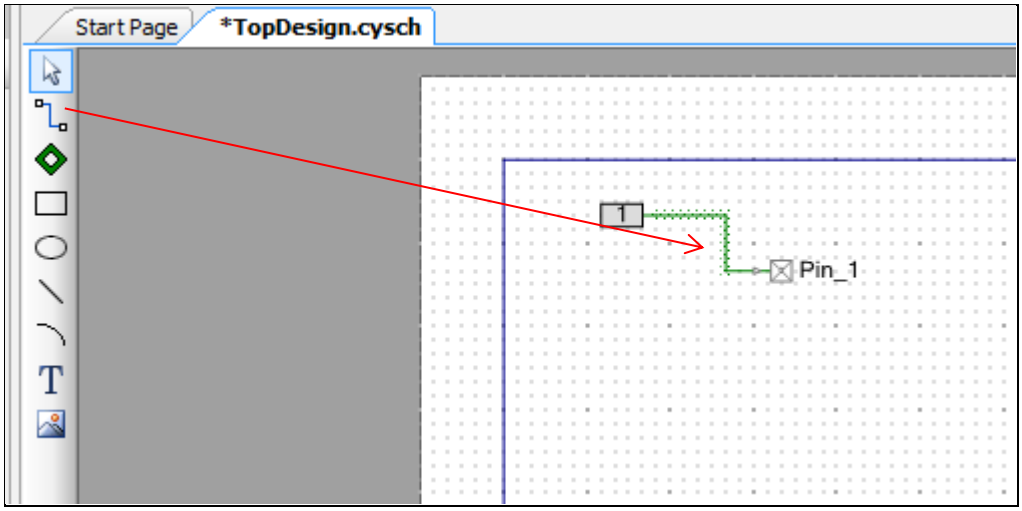


Figure 9 - Connecting two components

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.

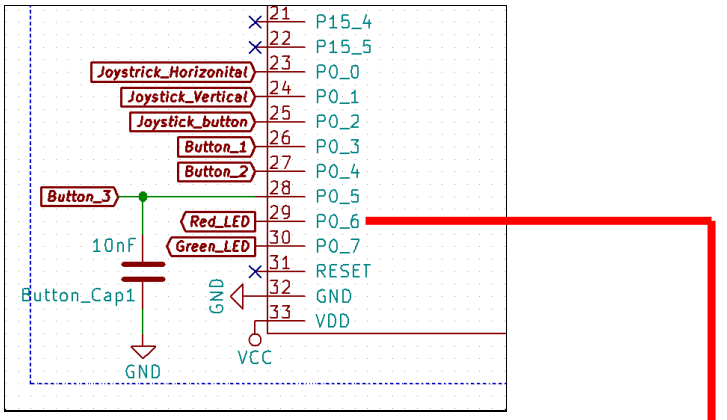


Figure 10 - Checking the schematics for the red LED

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

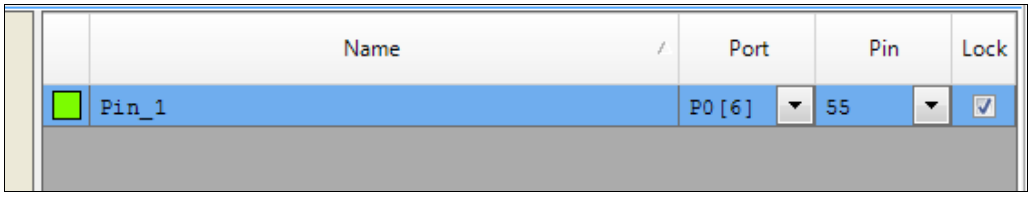


Figure 11 - Selecting the physical pin

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.

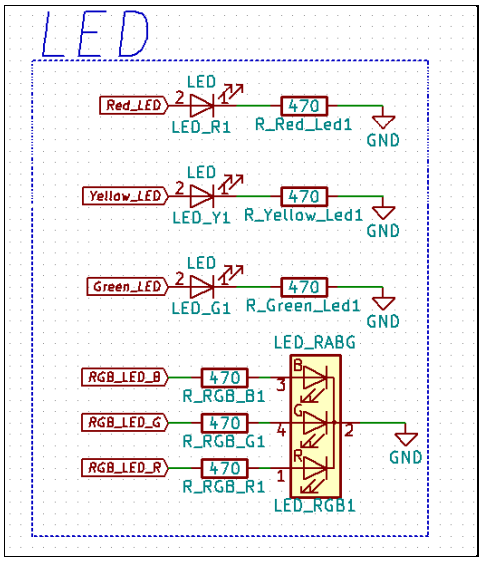


Figure 12 - Pull Down resistor of the LED

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.

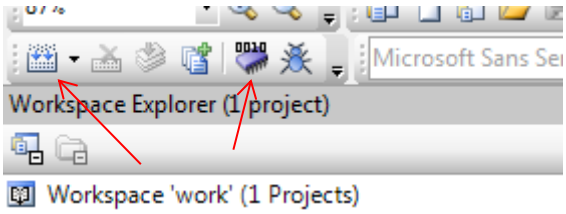


Figure 13 - Compile and Flash

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.

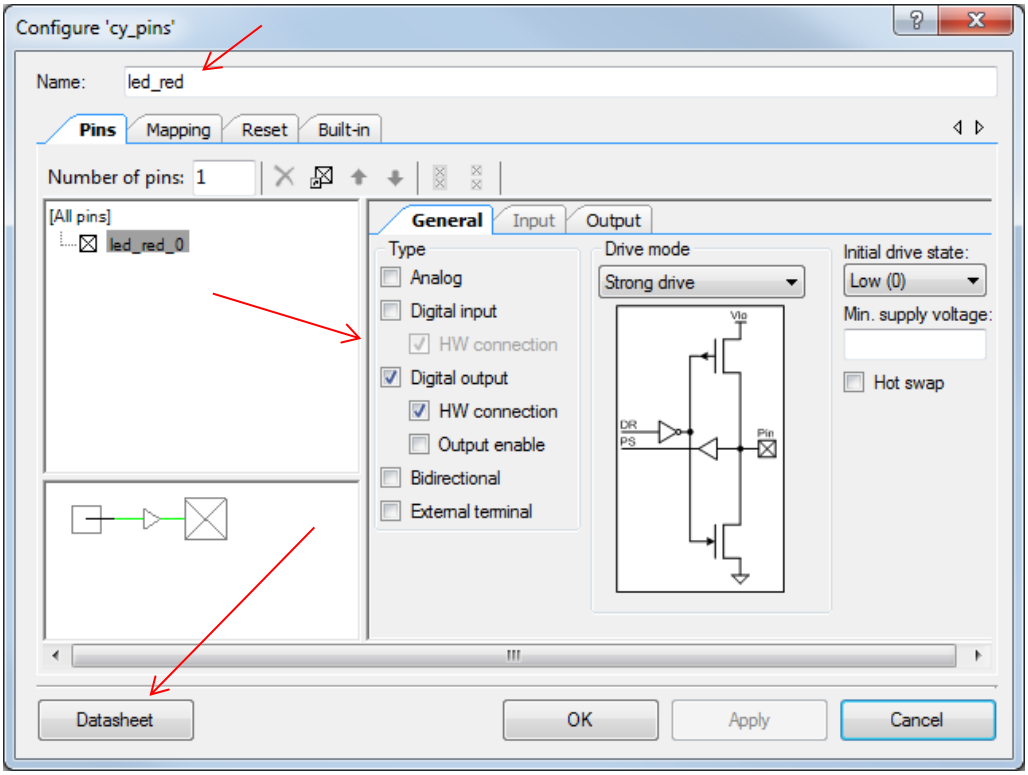


Figure 14 - Configuring a peripheral

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:

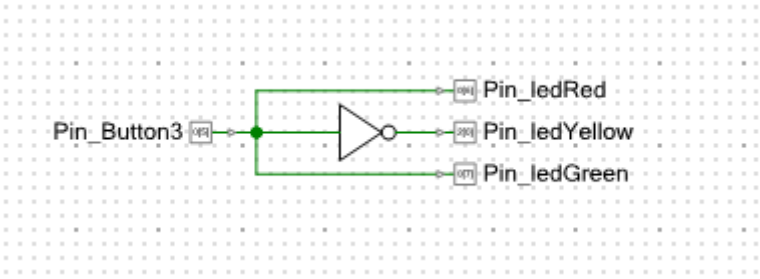


Figure 15 - Button / LED combination

The structure of this schematic can be divided into 3 blocks, which a 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.

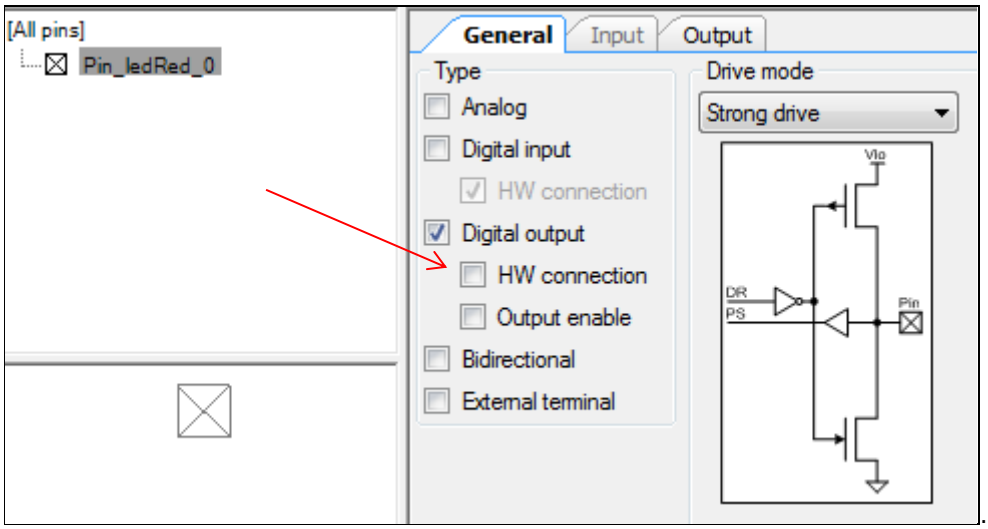


Figure 16 - Removing the hardware connection

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.

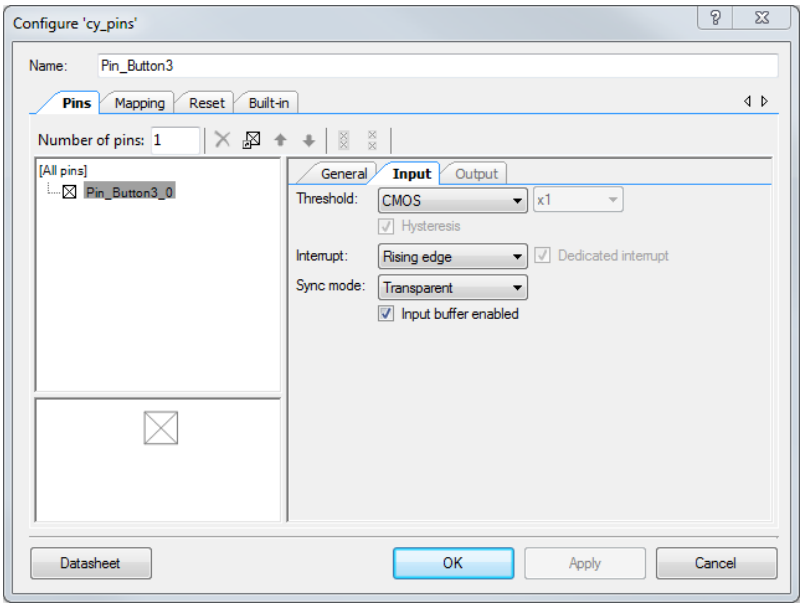


Figure 17 - Adding an interrupt functionality to a port

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.

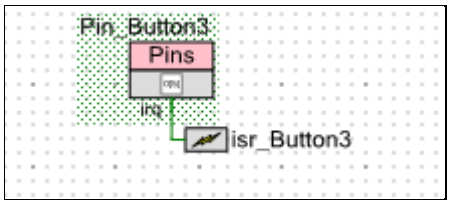


Figure 18 - Adding an ISR component[1]

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;
```

[1] We can also connect an interrupt component to a pin directly. This allows to add additional logic elements like a not or an or gate. The generated UDB blocks slightly differ and it depends on the application use case, which approach is more suitable.

```
    ramVectorTable = (cyisraddress *) *CYINT_VECT_TABLE;

    ramVectorTable[CYINT_IRQ_BASE + (uint32)isr_Button3__INTC_NUMBER] = ad-
dress;
}
```

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.

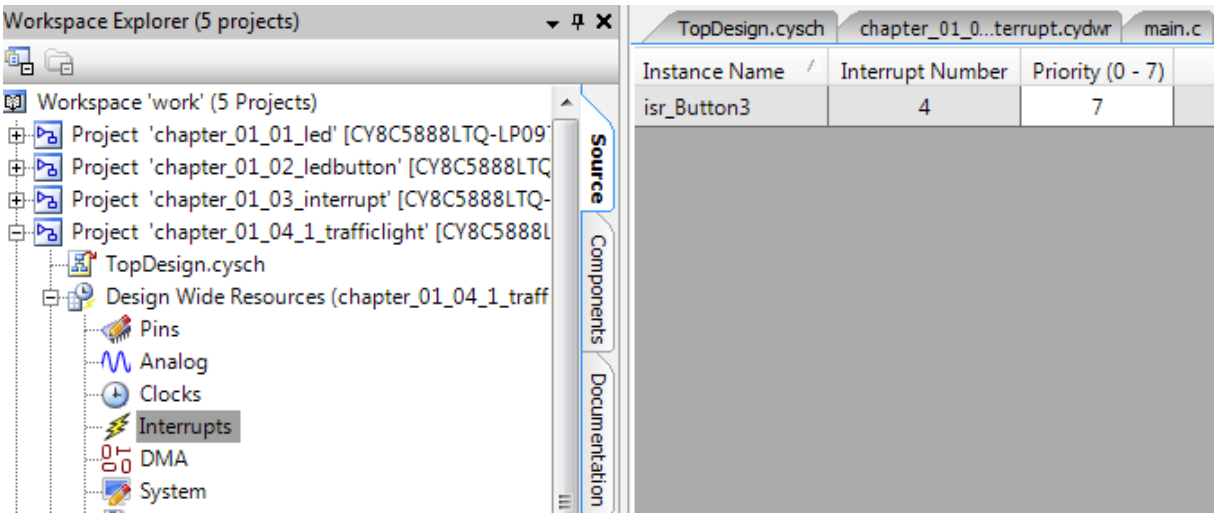


Figure 19 - Configuring the interrupt priority

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.

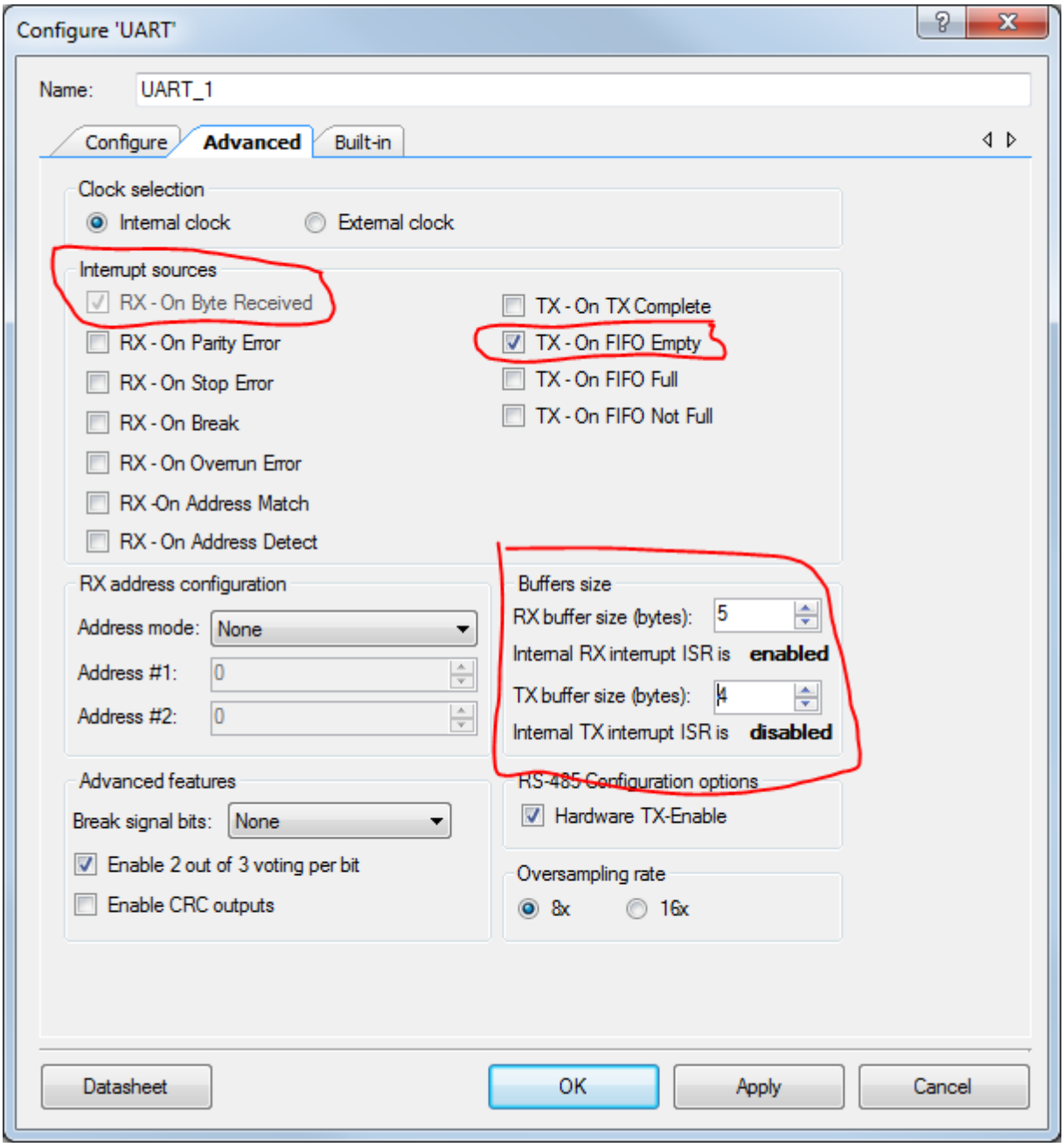


Figure 20 - Interrupt configuration of an UART component

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.

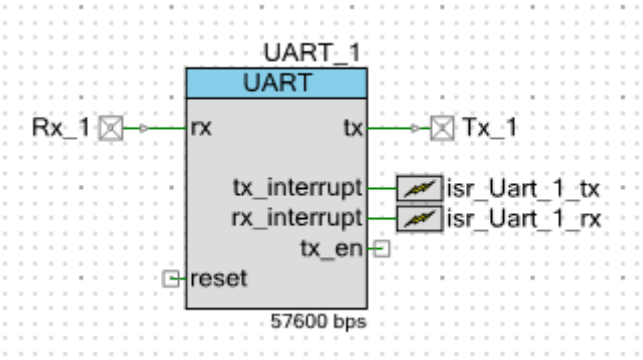


Figure 22 - Own ISR’s for transmitting and receiving data

1.4 Light Effects

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

Until know, 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 inacceptable 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

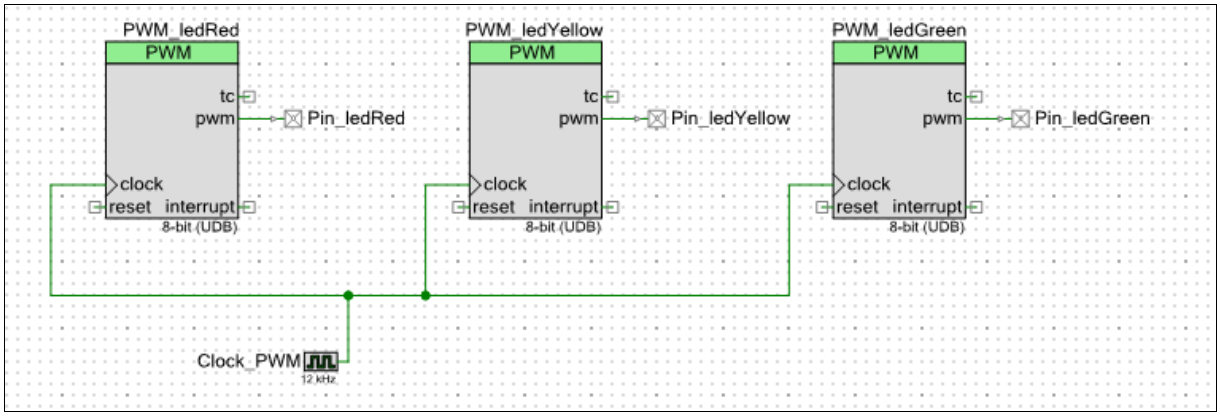
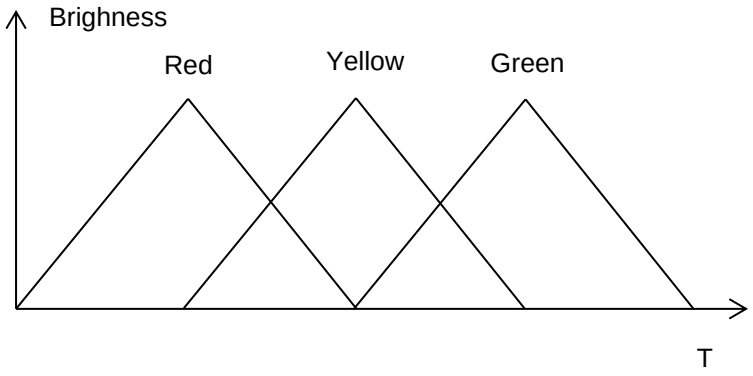


Figure 23 - Using PWM to control the LED brightness

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.

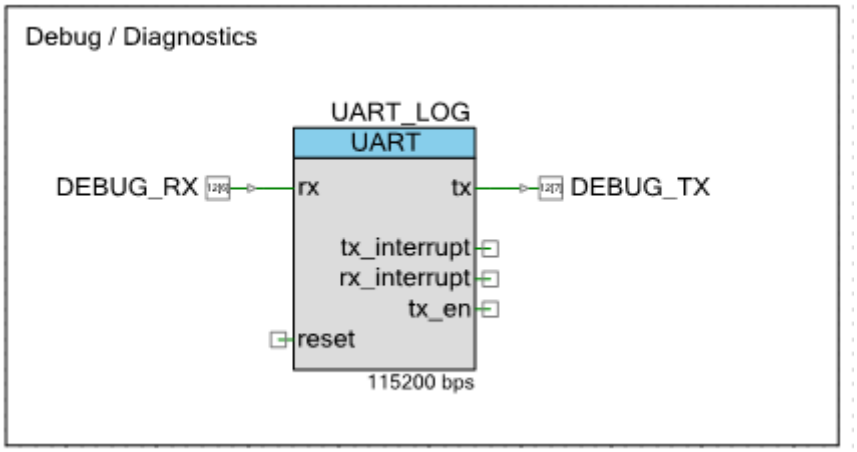


Figure 24 - USART Peripheral

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.

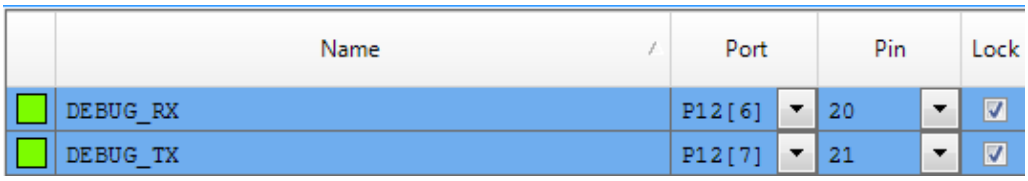


Figure 25 - Pins used for the FTDI bridge

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.

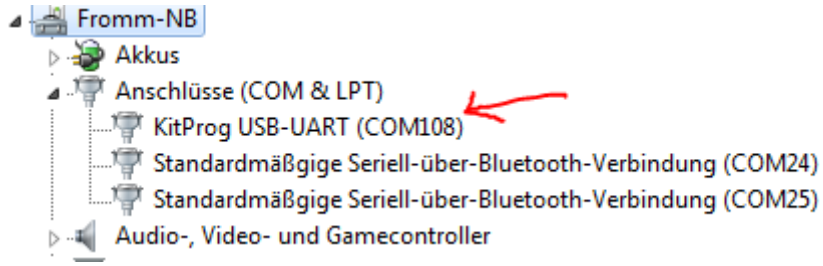


Figure 26 - Locating the correct UART port

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

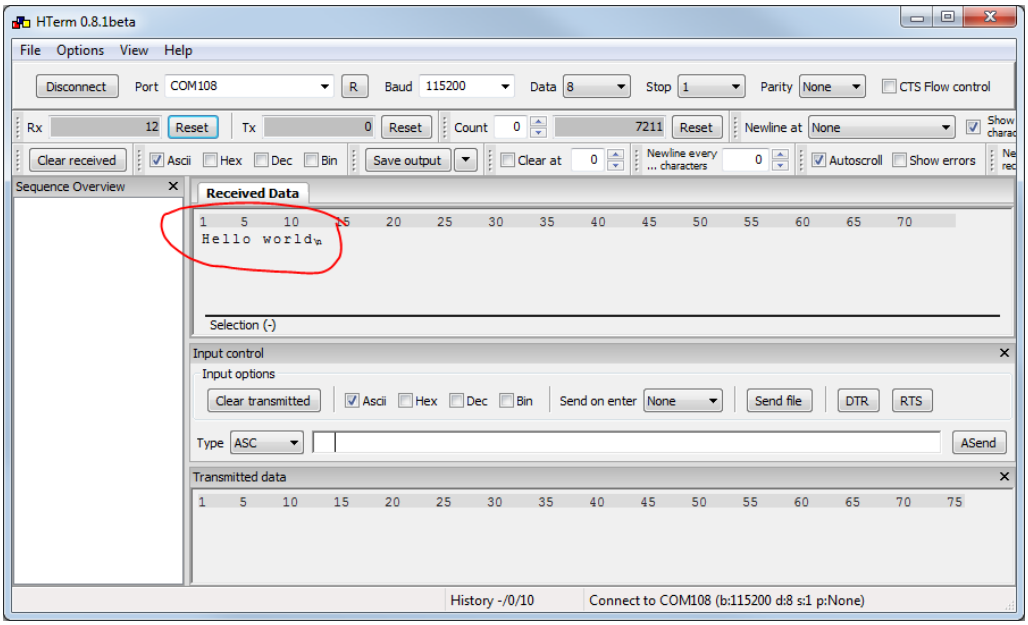


Figure 27 - Console Output on HTERM

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

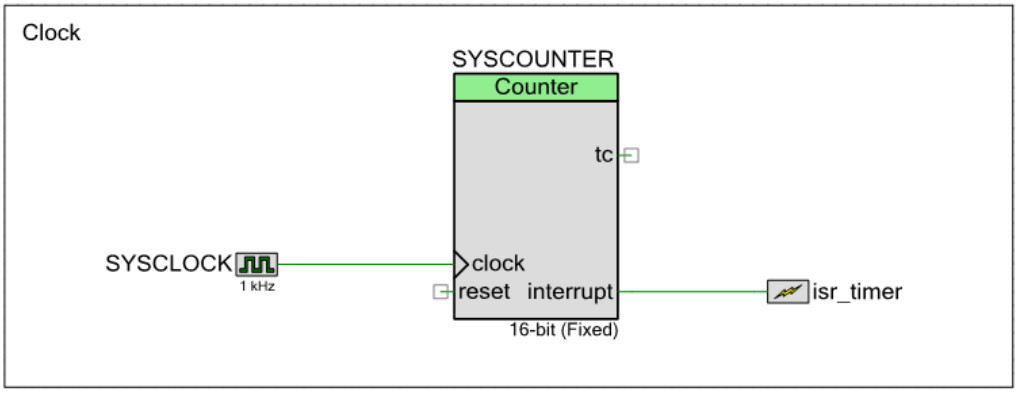
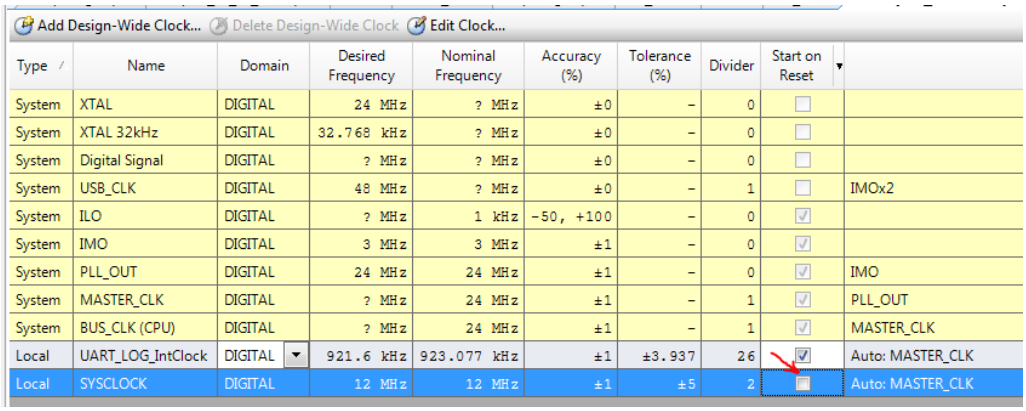


Figure 28 - Clock and Counter

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	☐	
System	XTAL 32kHz	DIGITAL	32.768 kHz	? MHz	±0	-	0	☐	
System	Digital Signal	DIGITAL	? MHz	? MHz	±0	-	0	☐	
System	USB_CLK	DIGITAL	48 MHz	? MHz	±0	-	1	☐	IMOx2
System	ILO	DIGITAL	? MHz	1 kHz	-50, +100	-	0	☑	
System	IMO	DIGITAL	3 MHz	3 MHz	±1	-	0	☑	
System	PLL_OUT	DIGITAL	24 MHz	24 MHz	±1	-	0	☑	IMO
System	MASTER_CLK	DIGITAL	? MHz	24 MHz	±1	-	1	☑	PLL_OUT
System	BUS_CLK (CPU)	DIGITAL	? MHz	24 MHz	±1	-	1	☑	MASTER_CLK
Local	UART_LOG_IntClock	DIGITAL	921.6 kHz	923.077 kHz	±1	±3.937	26	☑	Auto: MASTER_CLK
Local	SYSCLOCK	DIGITAL	12 MHz	12 MHz	±1	±5	2	☐	Auto: MASTER_CLK

Figure 29 - Clock configuration menu

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

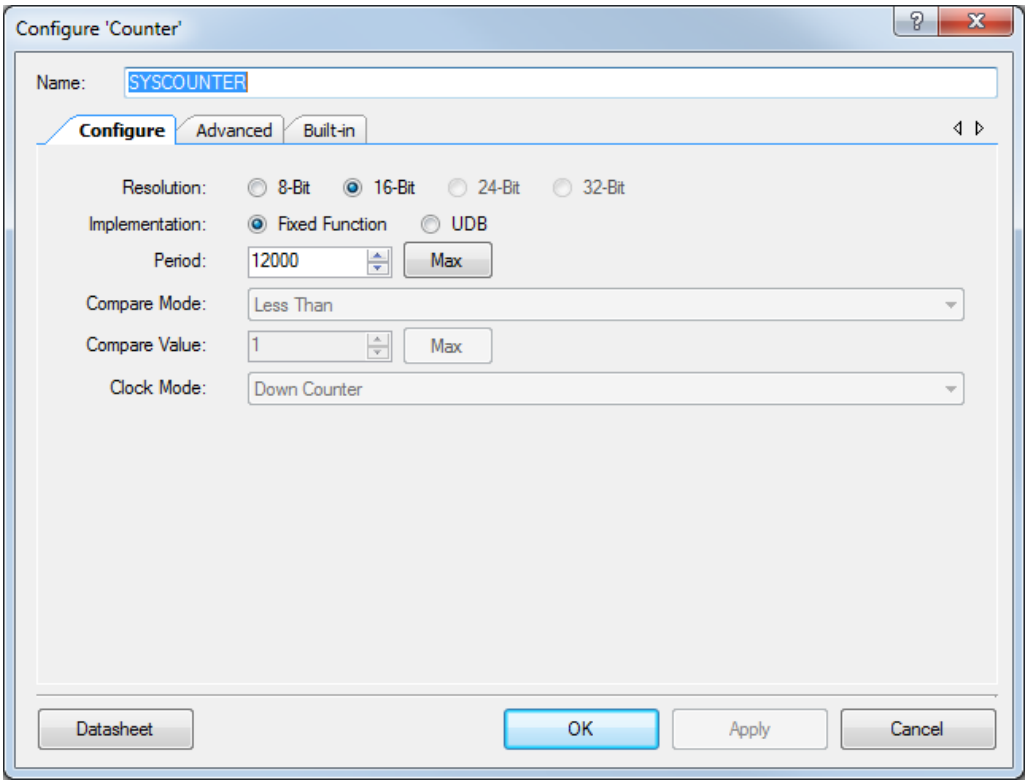


Figure 30 - Counter Configuration

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.

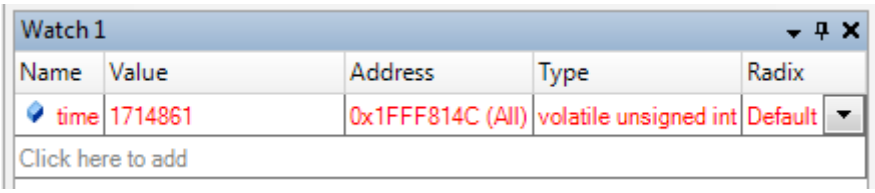


Figure 31 - Wrong timer value

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 byslowing the clock even further down or by disabling the breakpoint and to have the program run for another 1s. Again you will notive 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:

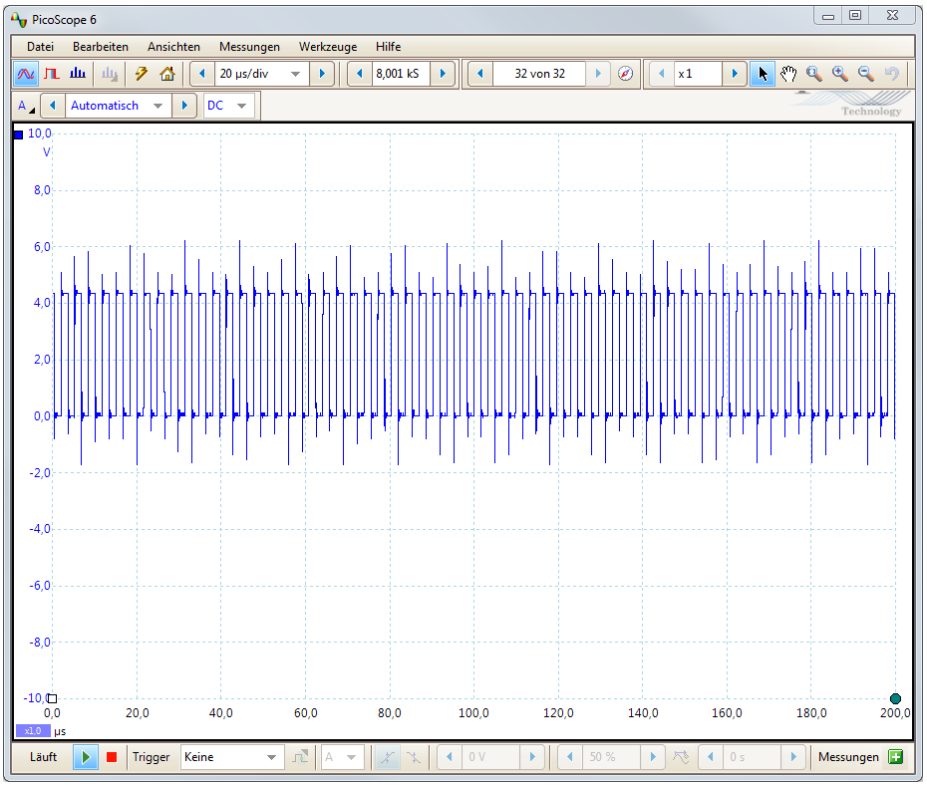


Figure 32 - Portpin LED in ISR

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.

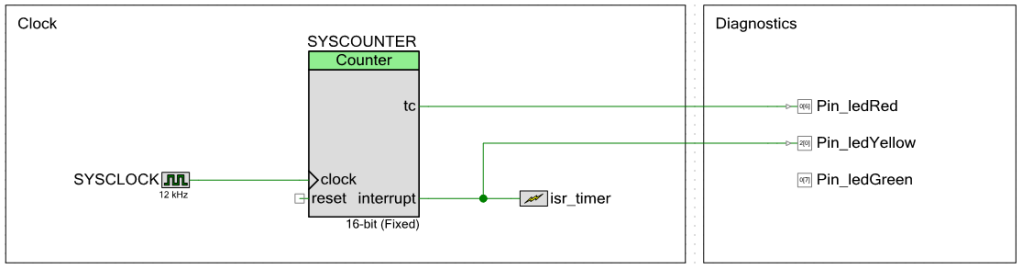


Figure 33 - More LED diagnostics

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 or a misunderstanding of the documentation. So, let’s try to reset the counter explicitly once the terminal count (TC) is reached:

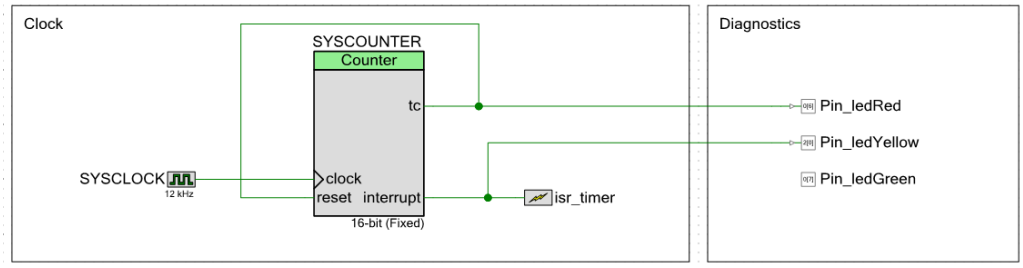


Figure 34 - Corrected counter functionality

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. [2]

Is this a real proper solution or is it more of a hack? Well, this is not a black/white question. For many years, I considered this to be a proper solution, until a collegue found out that this fix does not work when selecting a UDB block. After hours of searching the root cause, the collegue found the following description, hidden on page 2345 of the hardware manual:

Counter Interrupt

An interrupt output is available to communicate event occurrences to the CPU or to other components. The interrupt can be set to be active on a combination of one or more events. The interrupt handler should be designed with careful consideration for determining the source of the interrupt and whether it is edge- or level-sensitive, and clearing the source of the interrupt. The interrupt can be cleared by reading the status register (that is, by calling the Counter_ReadStatusRegister() API.)

Figure 35 -The magic command to reset counter interrupts

I.e. not the hardware peripheral was the root cause of the problem, but the interrupt. The connection between TC and reset is not required and should be removed! Instead, the Counter_ReadStatusRegister-API is placed as first command in the ISR.

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.
- 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 consuming[3] but is a MUST to build a stable system.
- Good programmers look LEFGT AND RIGHT when crossing a one way street.
- API’s even when designed by experts are not always that intuitive. Would you expect that reading a status register clears the interrupt?

[2] 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 ;-(

[3] 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. Then it took some more years and a collegue to find the real root cause.

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.

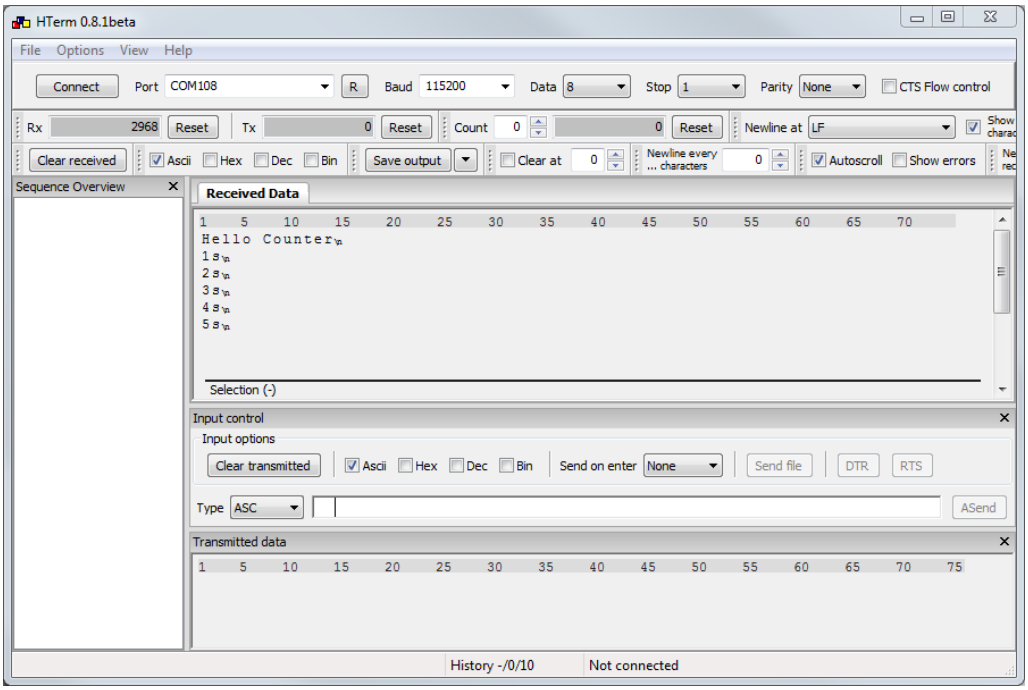


Figure 36 - The expected output

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.

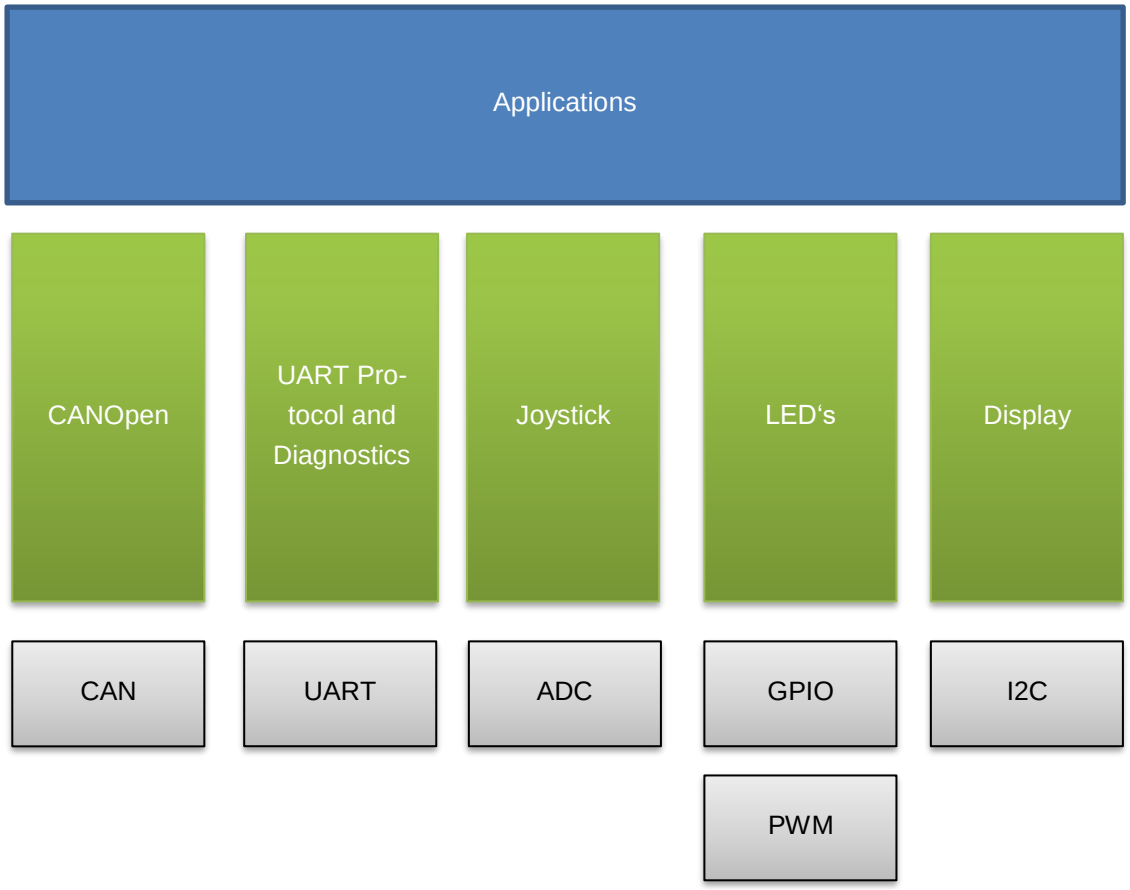


Figure 37 - Architecture

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 classes[4] will be written by us, whereas the gray low level driver code will be generated by the system.

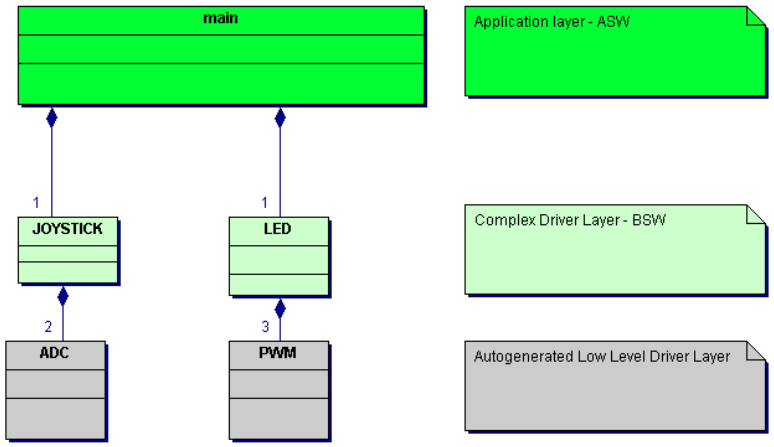


Figure 38 - Layered architecture

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 `cyapicallbacks.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

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

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.

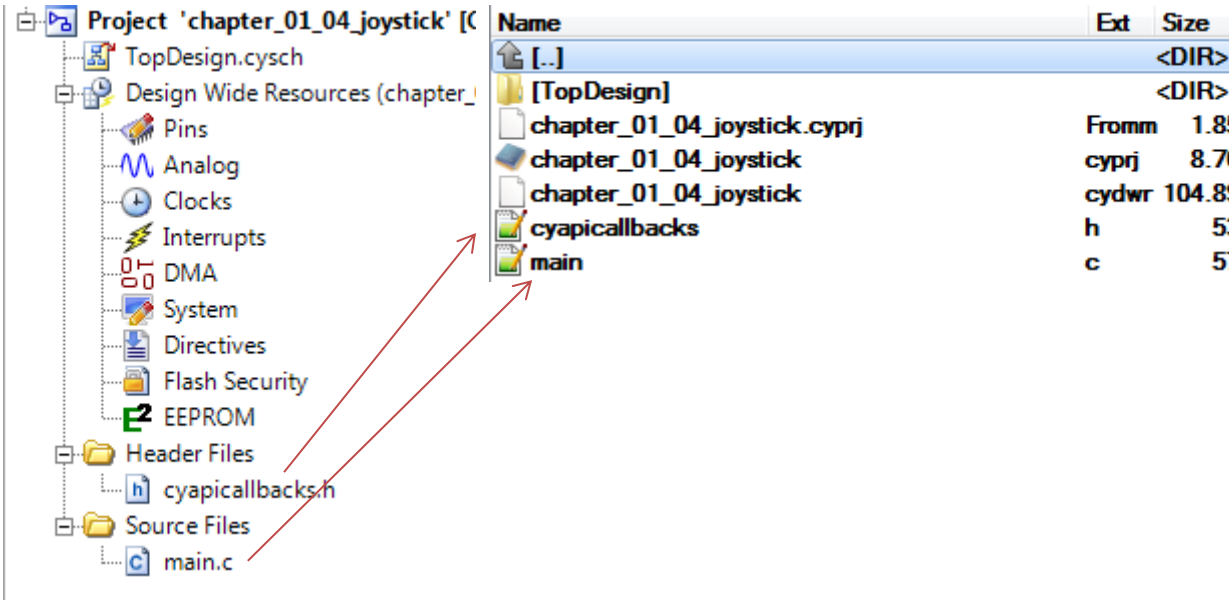


Figure 39 - Hacky default physical and logical structure of PSOC Creator

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 cyapicallbacks.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

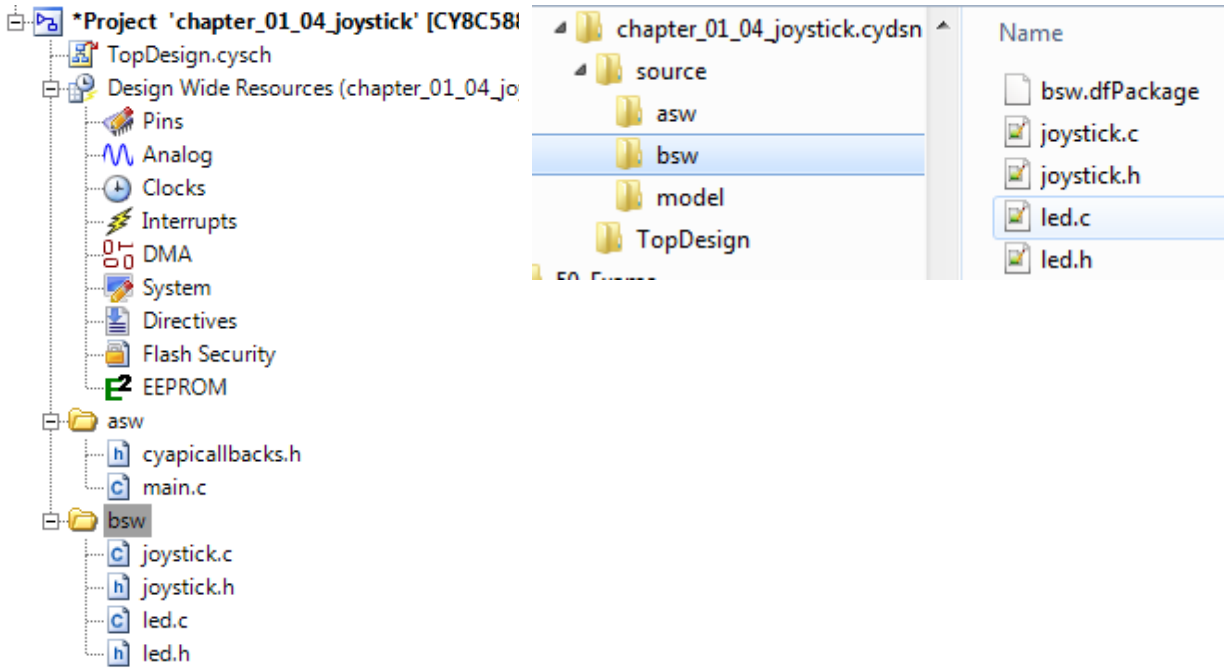


Figure 40 - Improved logical and physical structure

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)

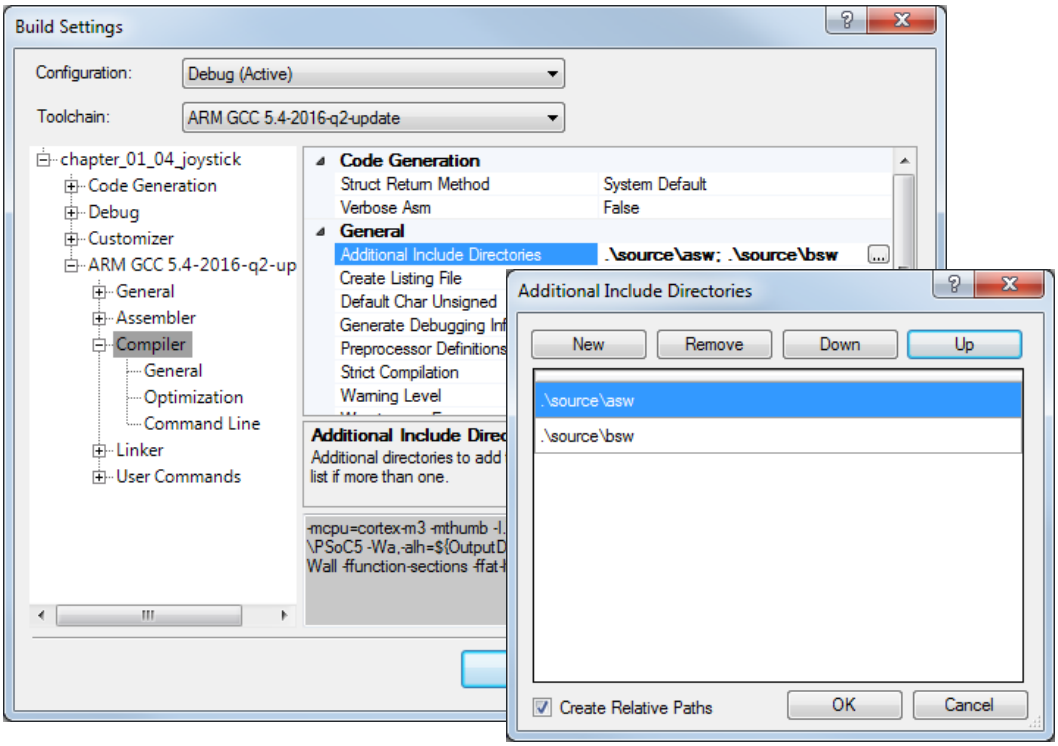


Figure 41 - Include path configuration

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 / Doxygen[5] 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)

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 repre-

[5] Doxygen can be downloaded as freeware: http://www.doxygen.nl/

	sented by a common enum. For this, copy the files `derivate.h` and `global.h` 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.

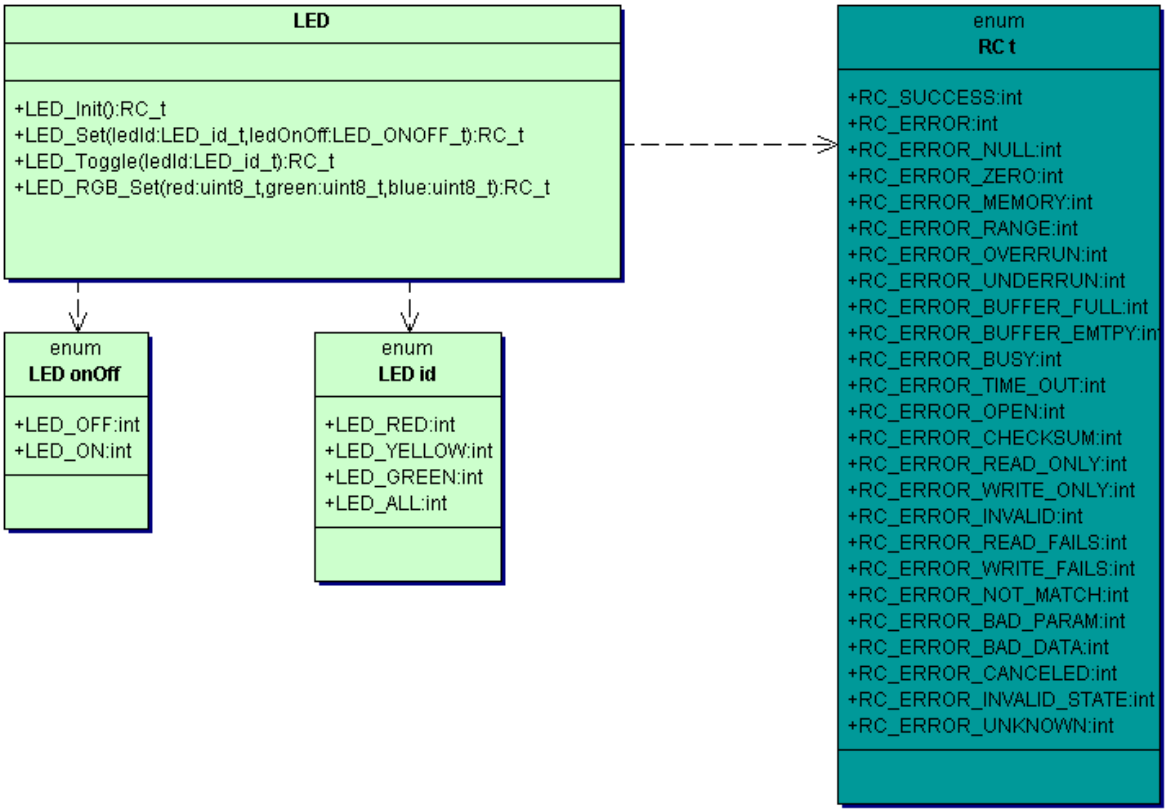


Figure 42 - Interface design for the LED class

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 reusing 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.

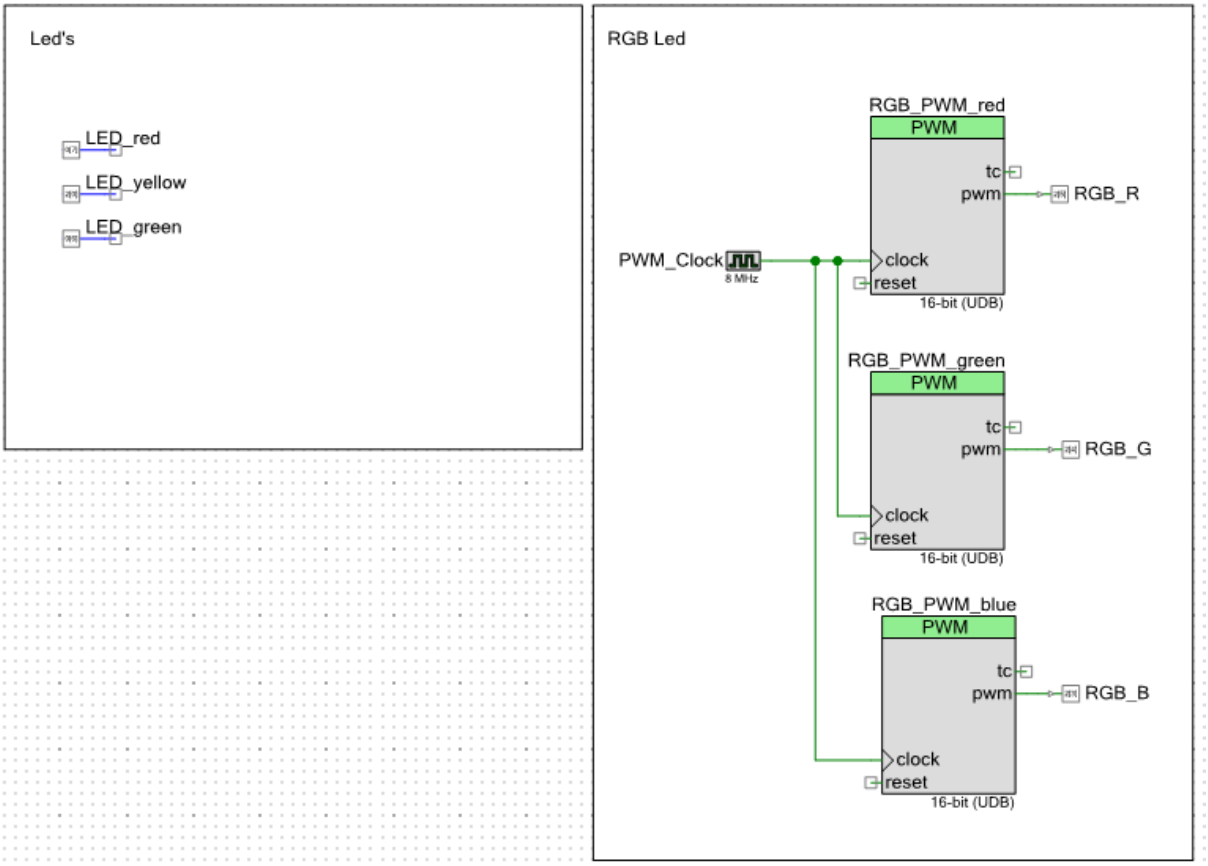


Figure 43 - Peripherals for the LED's

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:

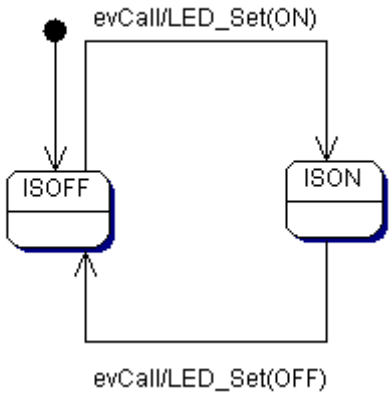


Figure 44 - Toggle logic described as state machine

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.

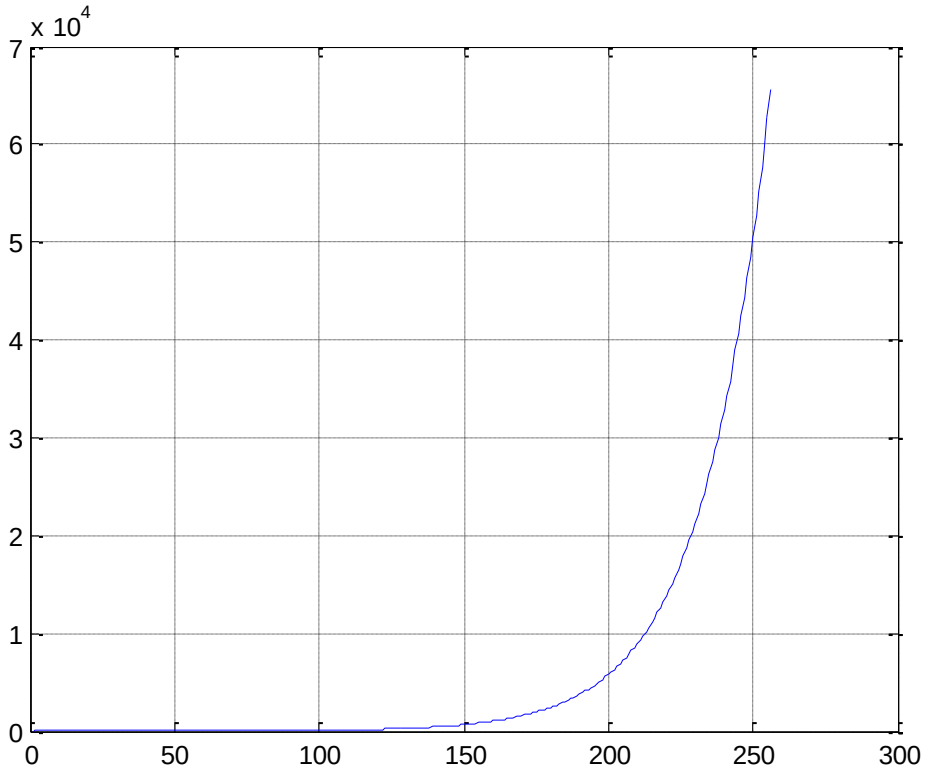


Figure 45 - Calibration curve for LED brightness

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, 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, 6, 6, 6, 6, 7,
    7, 7, 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 tread the x and y position as sint8_t value, having the following conventions • Joystick left: x = -128 • Joystick right: x = 127 • Joystick down: y = -128 • Joystick up: y = 127
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.

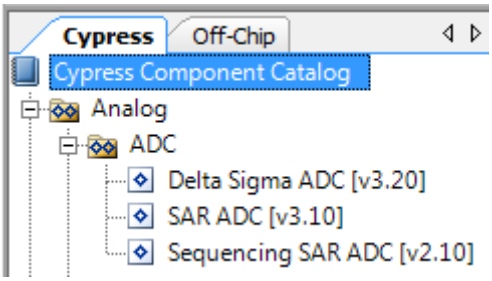


Figure 46 - Available ADC converters

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?

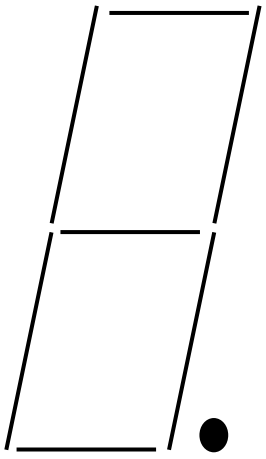
Why are we using the inverter/latch circuitry instead of connecting both 7 segment displays directly to the PSOC?

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.

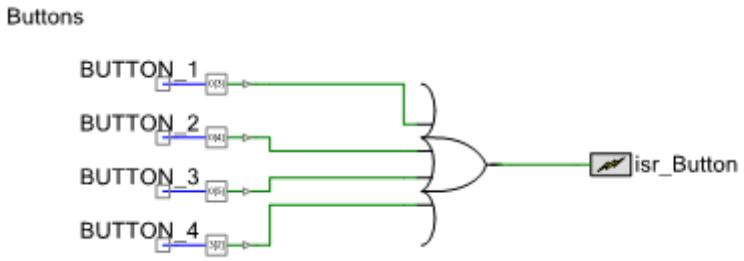


Figure 47 - 4 Pins 1 Interrupt

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!

2.3 Getting the TFT display up and running

Effort: 3h	Category - B
Integrating existing drivers, TFT display	

In this chapter we will get the TFT display up and running by integrating an existing driver (the efforts for writing an own driver would be rather high).

2.3.1 Integrating the driver

In a first step we will download the driver and integrate it into our project. Theoretically, we could simply copy all required files into the bsw folder, but as the number of files is rather large, we will add another layer of folders:

Folder	Content
\bsw	Will contain general files like global.h and derivate.h
\bsw\labboard	Will contain the complex drivers (button, led, joystick) we have created so far. This folder will be used for most projects.
\bsw\tft \bsw\tft\fonts	This package will contain the tft driver package including some fonts.
\bsw\services	Here, we will place general purpose helper routines, like ringbuffers, logging modules and similar. The tft driver packages requires e.g. the logging component and math_helper utilities from this folder.

Download the services.zip and tft.zip packages from Moodle and copy them into the bsw folder of your project. Add logical folders in the PSOC creator and don't forget to make the new folders known to the compiler.

The project should compile and link without errors.

Hint: the picture below only shows some of the labboard drivers. Depending on your application, you might also need the button and seven segment drivers.

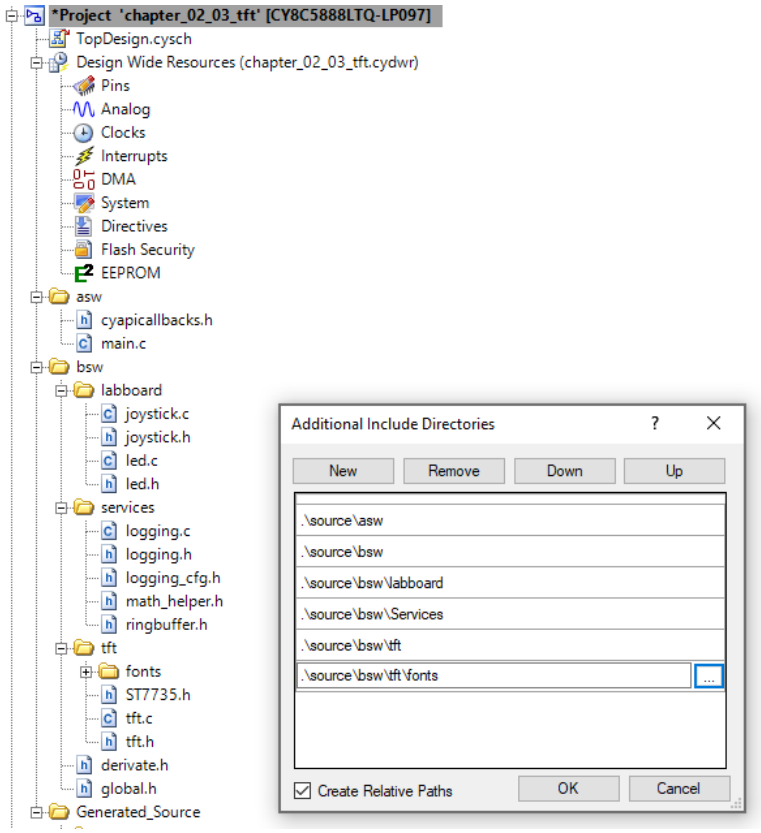


Figure 48 – Project structure with TFT component

2.3.2 Configuring the hardware interface

The TFT display requires the following interfaces to make it work

- A SPI interface for the data transfer
- A GPIO pin to differentiate between data and command words (DC)
- A GPIO Pin to reset the display (if needed)
- A PWM interface for the backlight control
- As we have only one SPI device on the bus, chip select can be hard-wired.

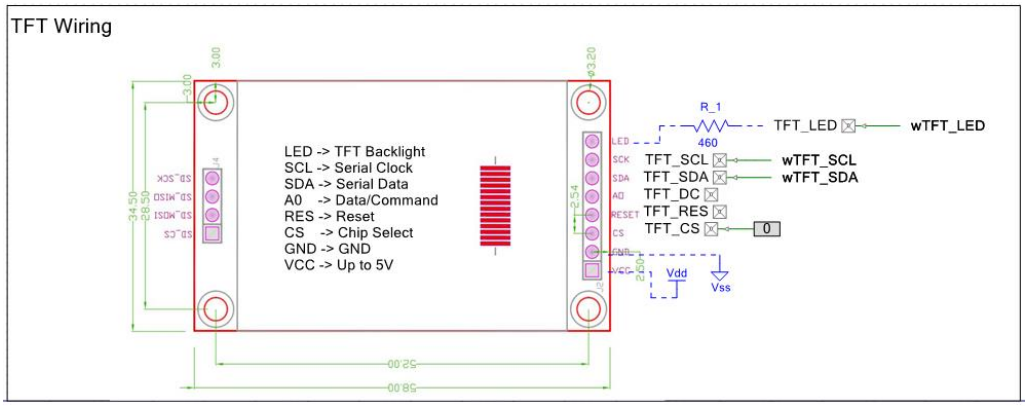


Figure 49 – TFT wiring

The communication between the PSoC and the TFT is established by using a SPI communication interface. As we do not read data from the TFT, we only use the MOSI (MasterOutSlaveIn) connection, which is connectedto SDA (Serial Data) of the TFT. SCL (Serial Clock) synchronizes the data

transfer. SS (Slave/Chip Select) usually is used to address a slave on the bus. As we only have one slave connected, we do not need it. Note that SDA and SCL are high frequency lines.

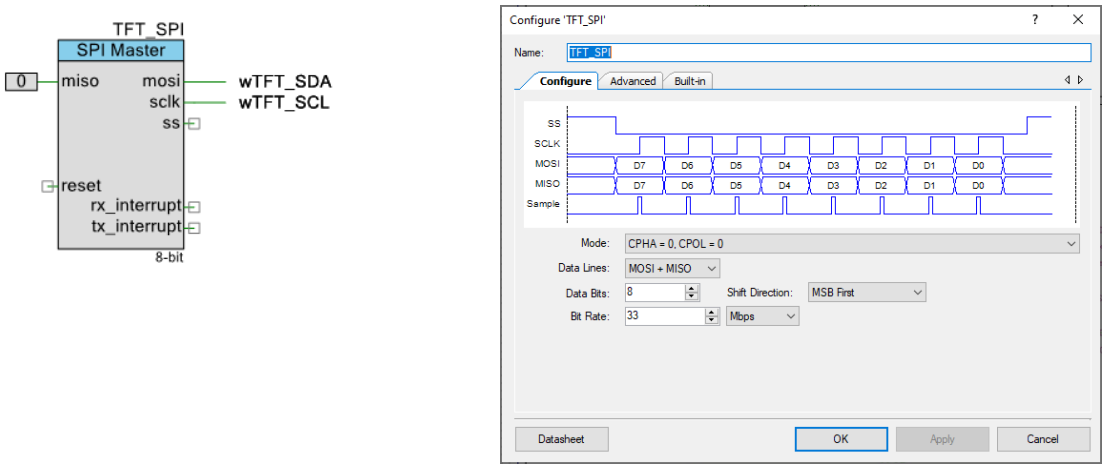


Figure 50 – TFT Hardware modules

We use a PWM to make the backlight brightness adjustable. The PWM has a period of 100 (Percentage value). If we want to have a frequency of 50Hz at the LED, we need a clock of 5kHz.

Based on the schematics, we will need the following pins:

TFT_CS	P3[3]
TFT_DC	P12[3]
TFT_LED	P12[2]
TFT_RES	P3[4]
TFT_SCL	P12[4]
TFT_SDA	P12[5]

Figure 51 – TFT Pinning

2.3.3 Skinny Sheep

Now let's start to explore the TFT-driver API. For this, open the file tft.h and check the content.

Adding the following 3 lines to your project (of course before the for(;;) loop!)

```
TFT_init();
TFT_setBacklight(100);
TFT_print("Hello world\n");
```

Should print "Hello world" on the TFT display.

2.3.4 A first game

Now let's add the following game logic:

- Depending on the joystick position, a cross is moving over the screen.
- If a button is being pressed, a red filled circle will be drawn at the current position.

- If another button is pressed, the screen is cleared

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

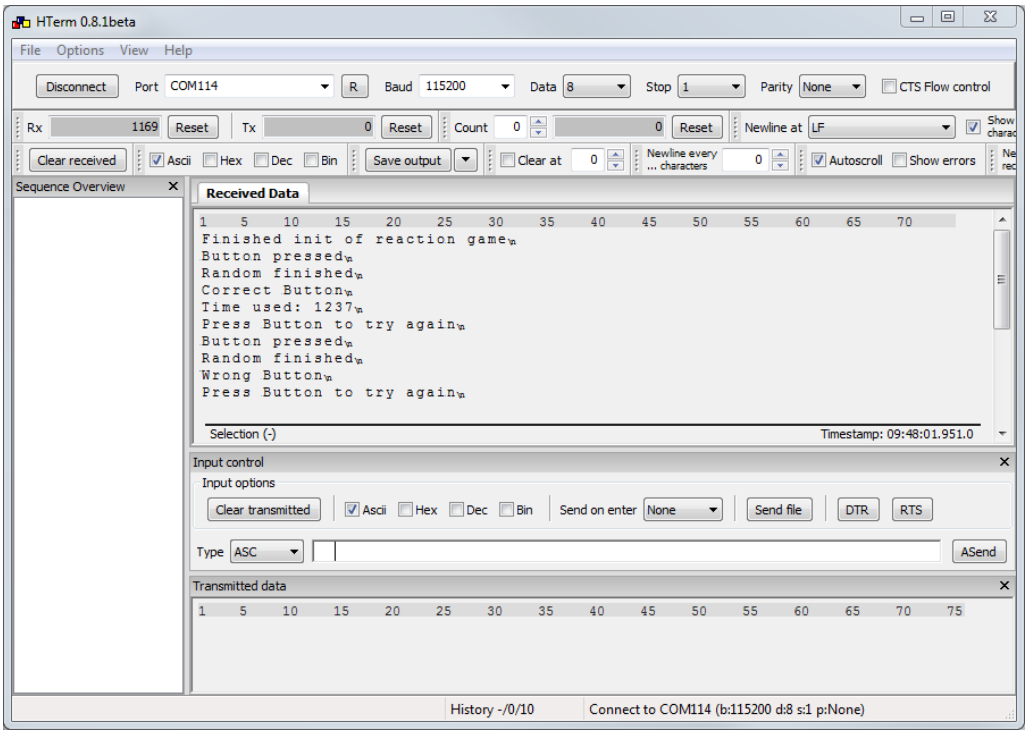
- Copy the required drivers from the previous project
- Set up the project folder structure (physical and logical)
- Copy the driver schematics
- Connect the pins as required
- Testcase 1: The project is compilable

3.1.2 Iteration 2

- Create the game.h|c 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:


HTerm 0.8.1beta
File Options View Help
Disconnect Port COM114 R Baud 115200 Data 8 Stop 1 Parity None CTS Flow control
Rx 1169 Reset Tx 0 Reset Count 0 0 Reset Newline at LF
Clear received Ascii Hex Dec Bin Save output Clear at 0 Newline every ... characters 0 Autoscroll Show errors
Sequence Overview
Received Data
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70
Finished init of reaction game\n
Button pressed\n
Random finished\n
Correct Button\n
Time used: 1237\n
Press Button to try again\n
Button pressed\n
Random finished\n
Wrong Button\n
Press Button to try again\n
Selection (-)
Timestamp: 09:48:01.951.0
Input control
Input options
Clear transmitted Ascii Hex Dec Bin Send on enter None Send file DTR RTS
Type ASC ASend
Transmitted data
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75
History -/0/10
Connect to COM114 (b:115200 d:8 s:1 p:None)

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.

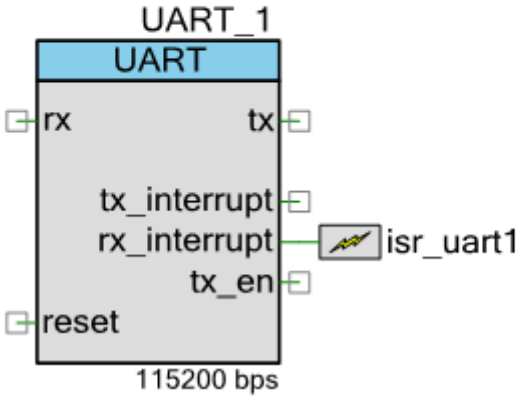
As we have only one board available, we will use one of the two UARTS on our board to build a first prototype. We can use the schematic editor to connect the pins for a loop back connection.

How do you connect the pins of the two boards?

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

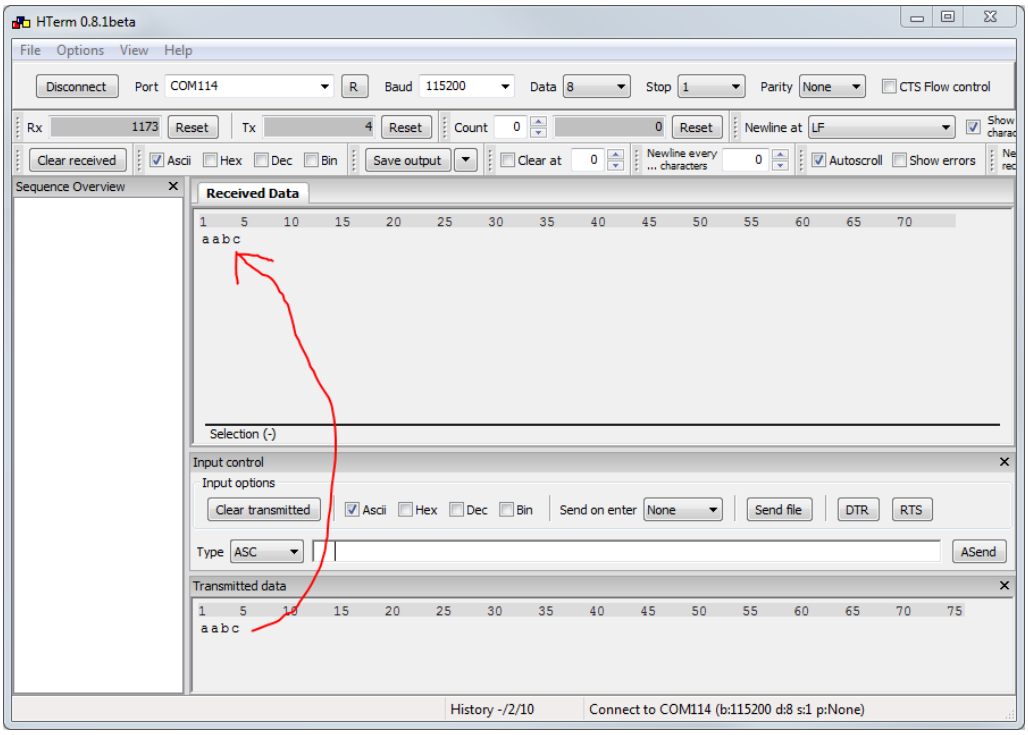


Figure 52 - Signalflow between all USARTs is working

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.

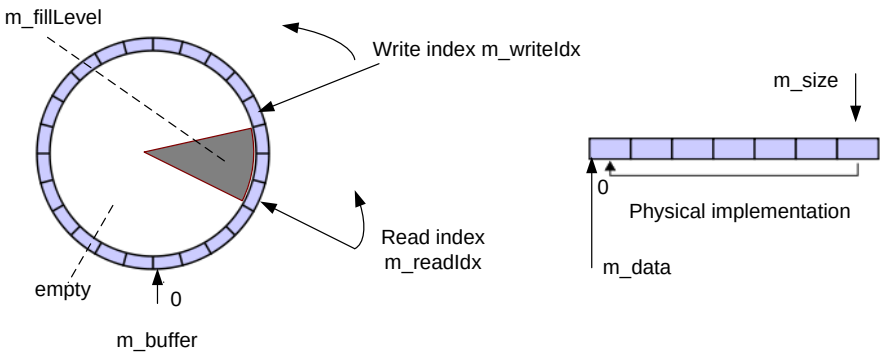


Figure 53 - Ringbuffer

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[6]

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.

[6] 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.

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.

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 RTOS[7] become very handy - we will go for the second solution in this exercise and implement the other pattern once we have introduces 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:

[7] Real Time Operating System

```
/**
 * 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 occured
 */
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);
```

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, fullfilling 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.

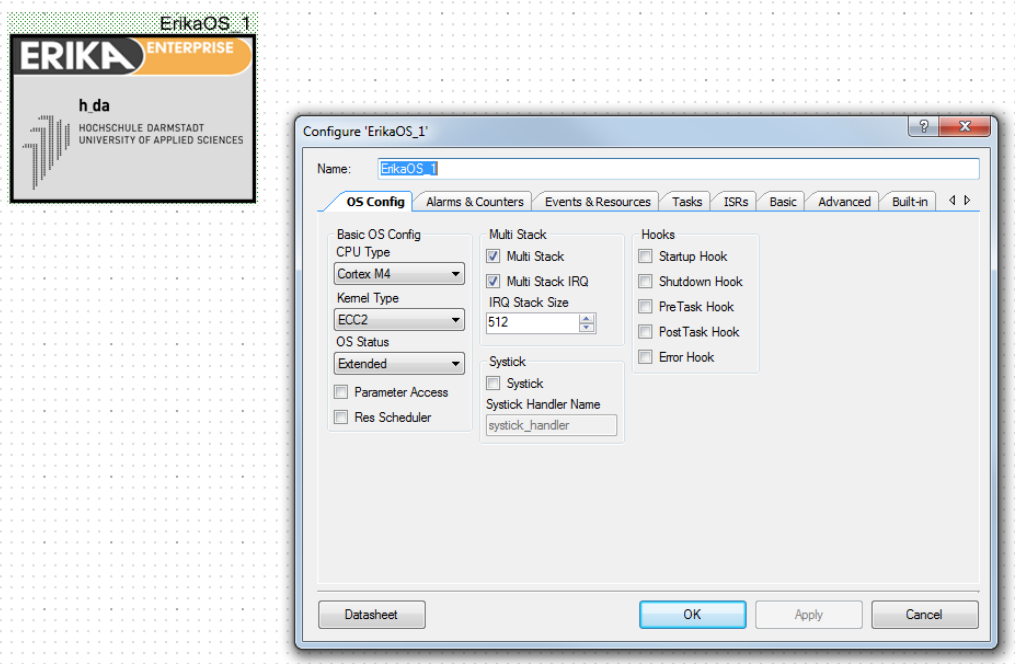


Figure 54 - Erika OS Component

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

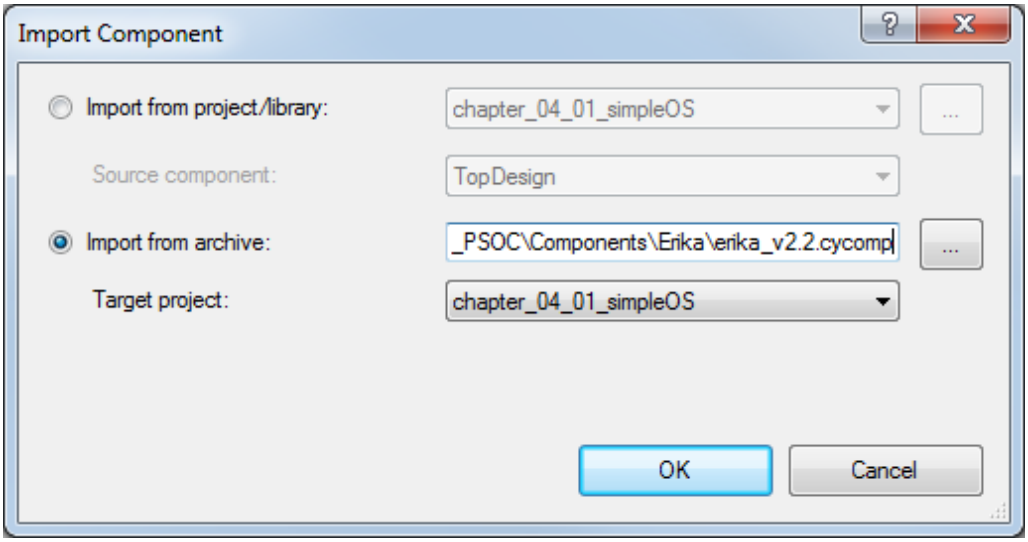


Figure 55 - Import Erika OS component

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

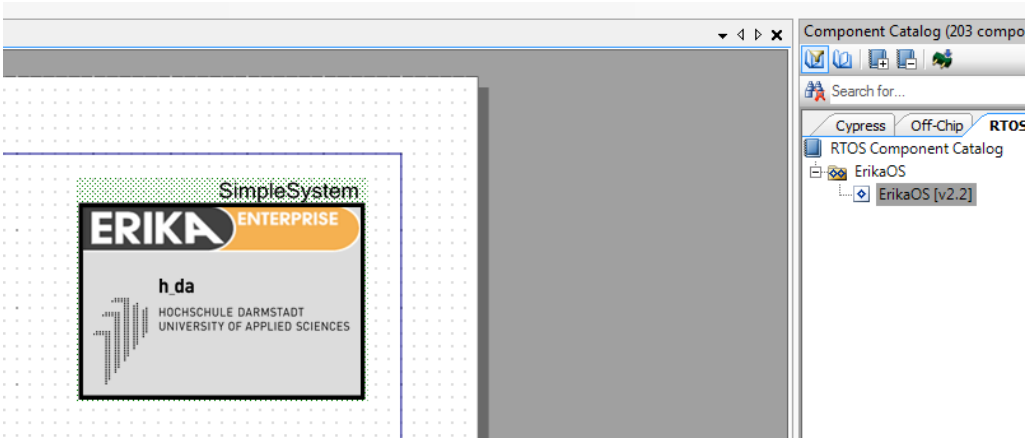


Figure 56 - Erika OS added to the TopDesign

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

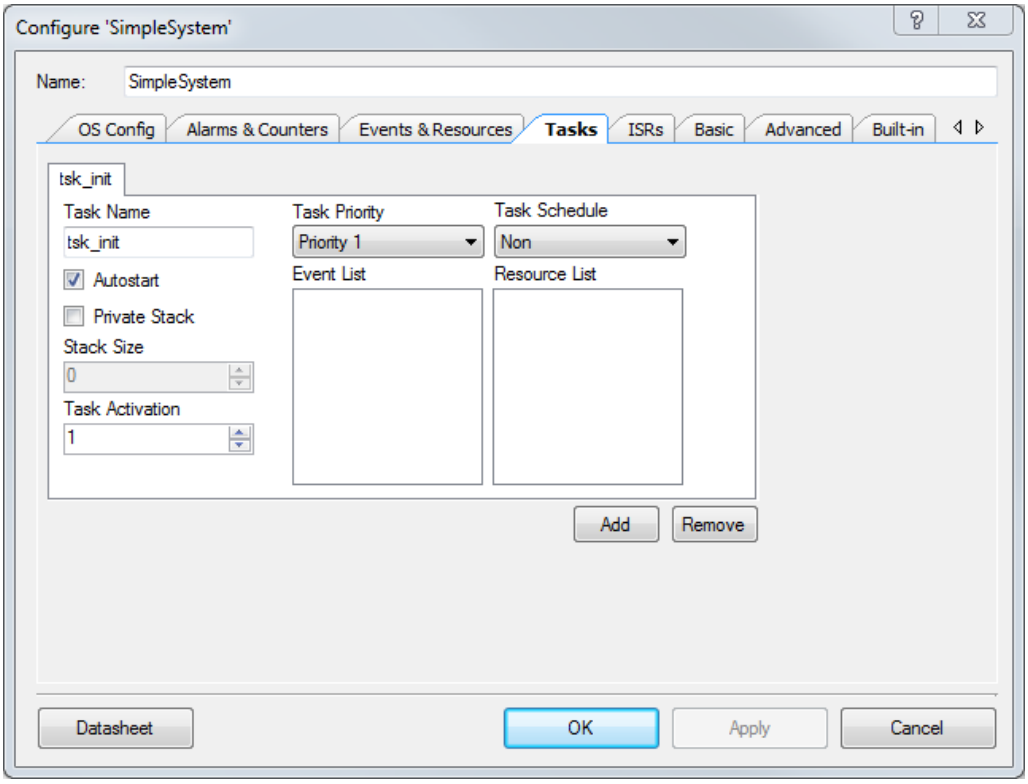


Figure 57 - A first task

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(OSDEFAULTAPPMODE);
}


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

TASK(tsk_init)
{

   TerminateTask();

}
```

Set a break point on the line TerminateTask and start the program in the debugger. Congratulation, 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 confugre 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 easly 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.

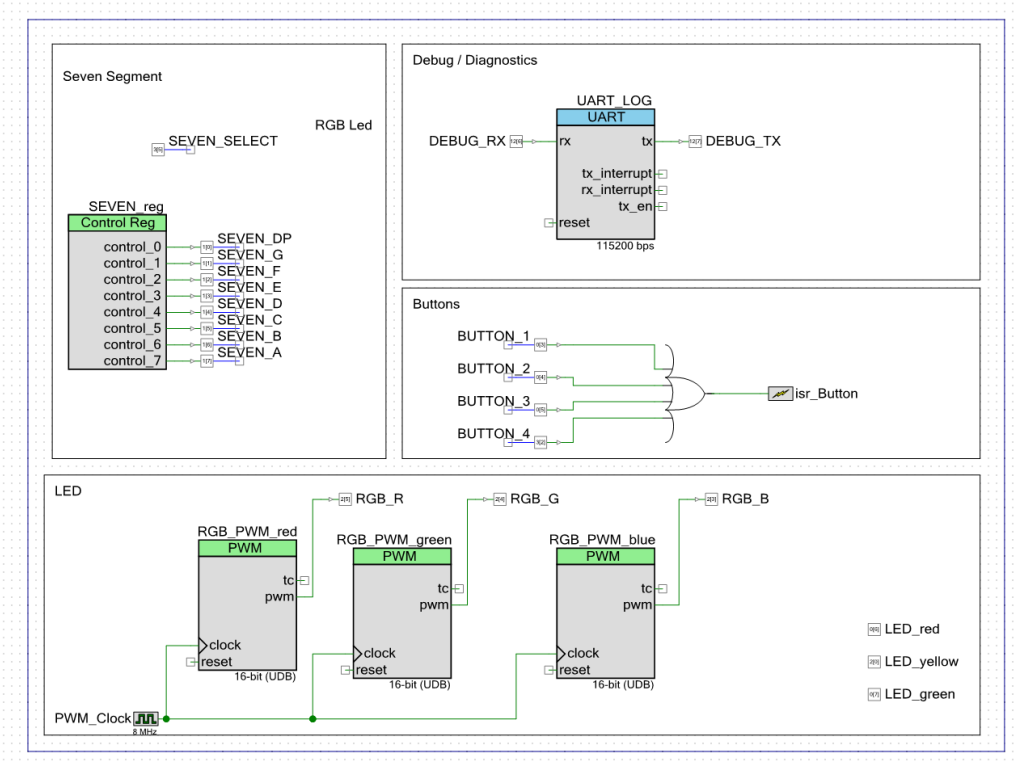


Figure 58 - Typical HMI peripheral configuration

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

Figure 59 - 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.

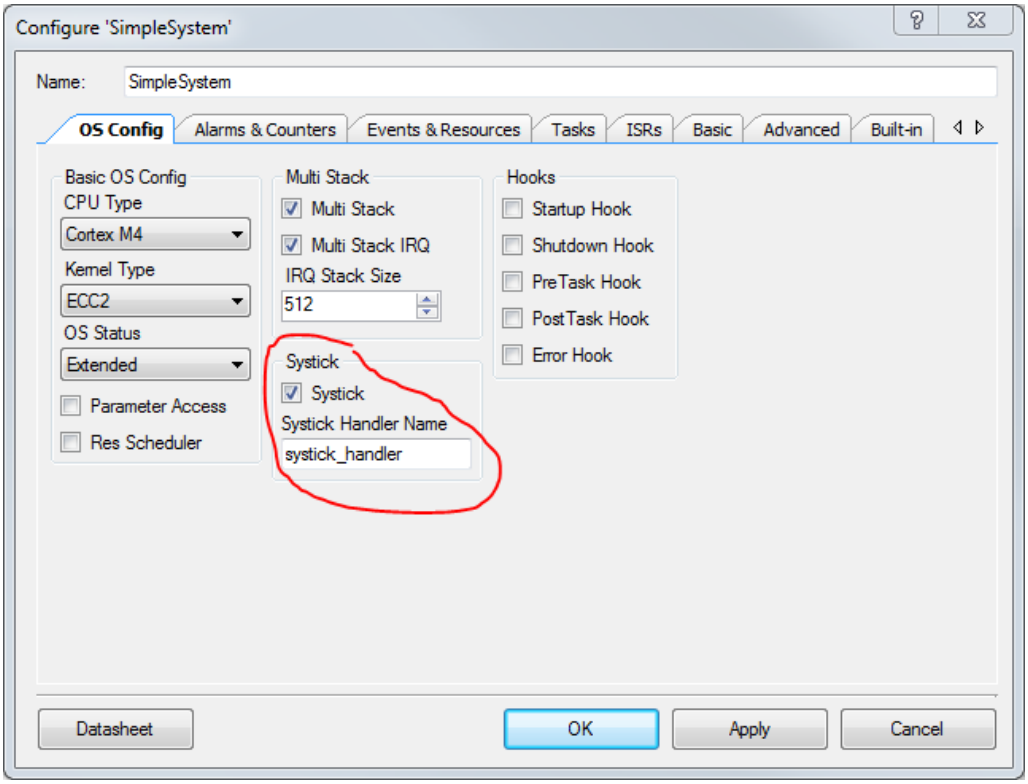


Figure 60 - Systick configuration

- and add a first counter object

- Furthermore, we can add a first alarm alrm_ledBlink, which will activate the task tsk_ledBlink

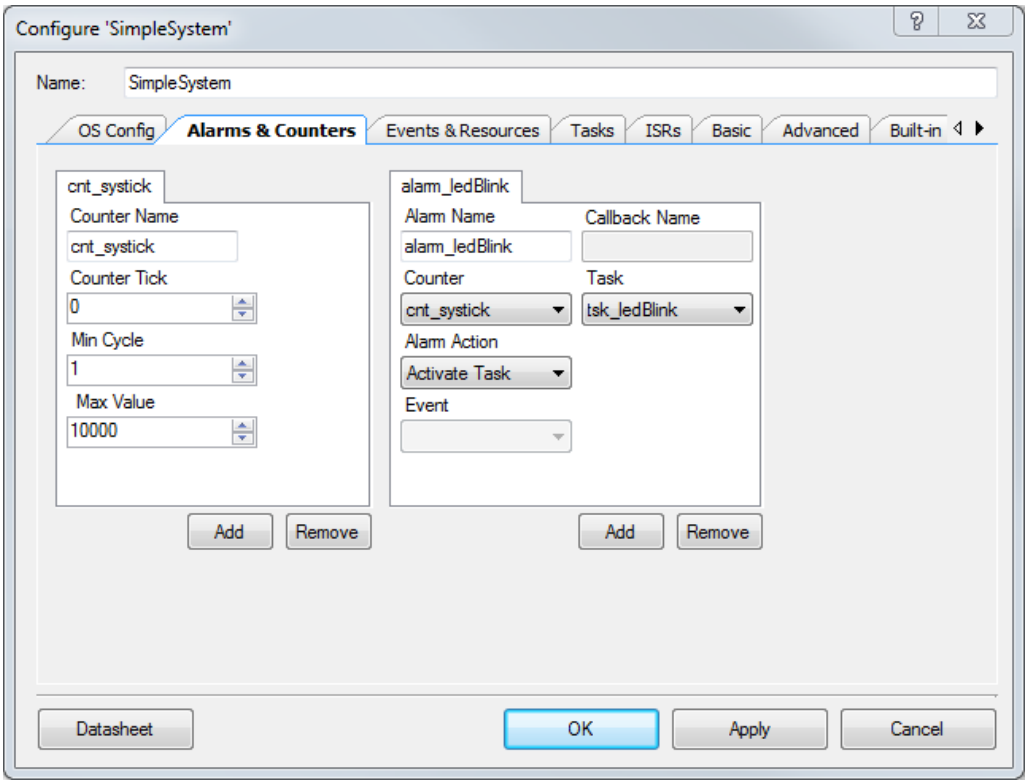


Figure 61 - Counter Object and Alarm

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();


    // 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
    //initialization!
    EE_system_init();

    // Must be started after interrupt reconfiguration
    EE_systick_start();

    //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.

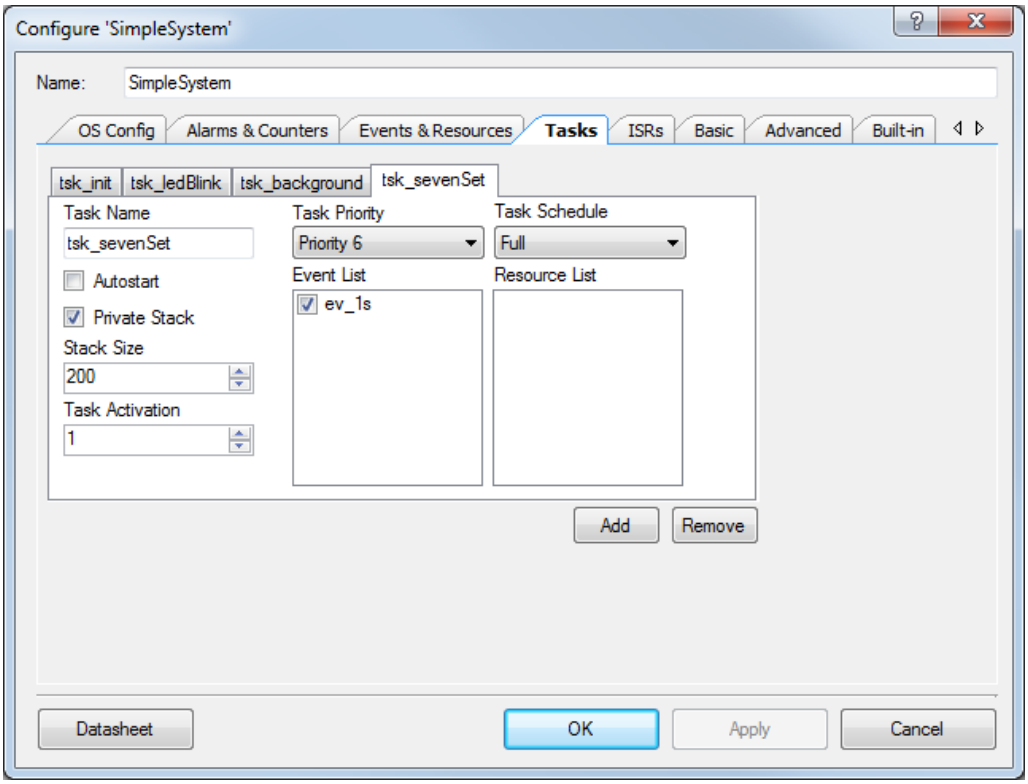


Figure 62 - A first extended task

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.

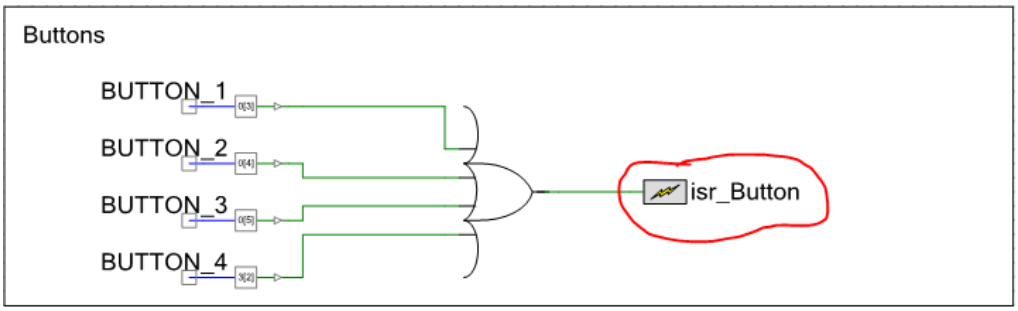


Figure 63 - ISR object name

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

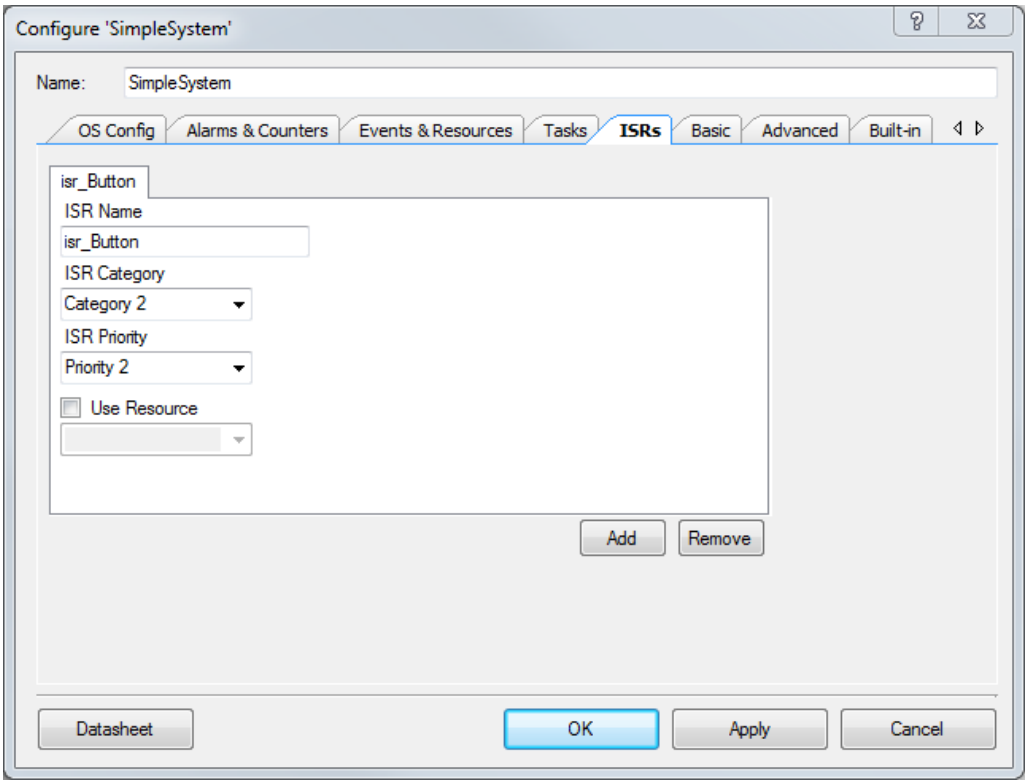


Figure 64 - Configuring the ISR in the OS

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).

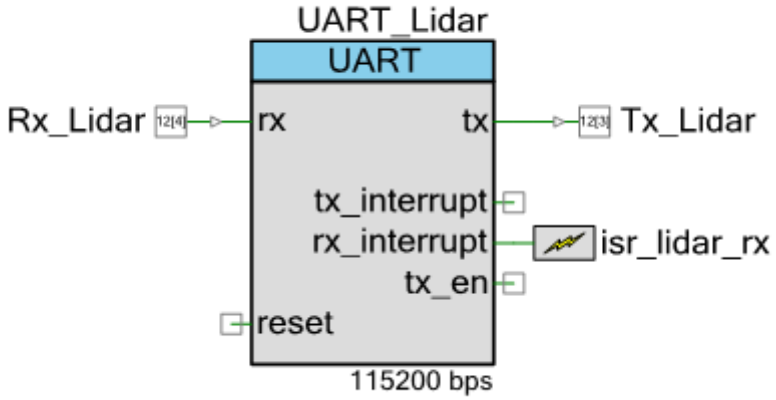


Figure 65 - Adding an RX interrupt to a UART port

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.

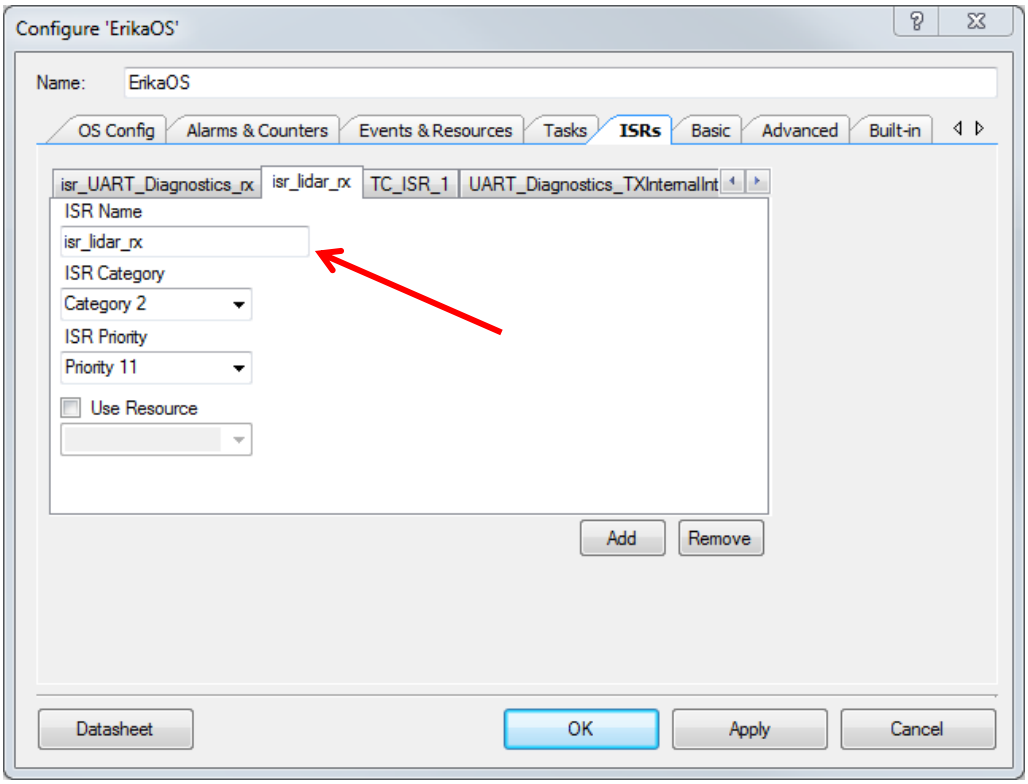


Figure 66 - Adding an interrupt to Erika OS configuration

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 67 - 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, story 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.

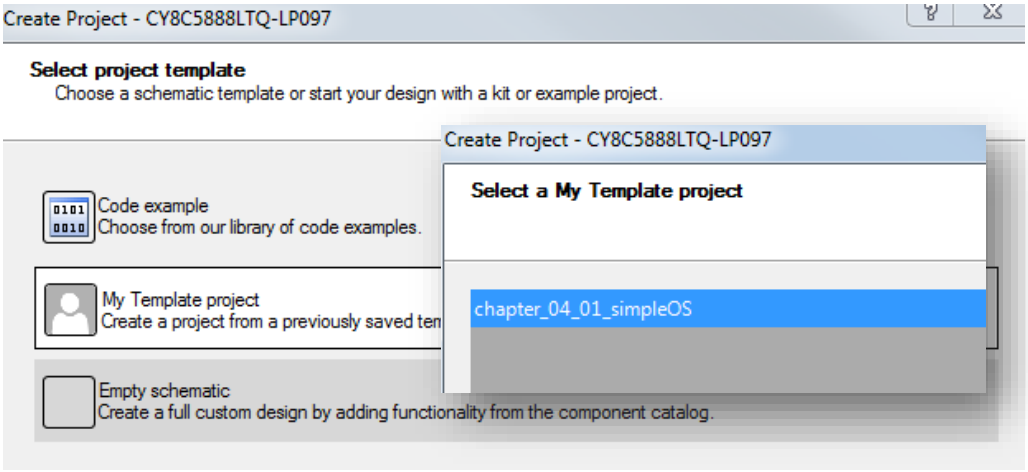


Figure 68 - Creating a new project using a template

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

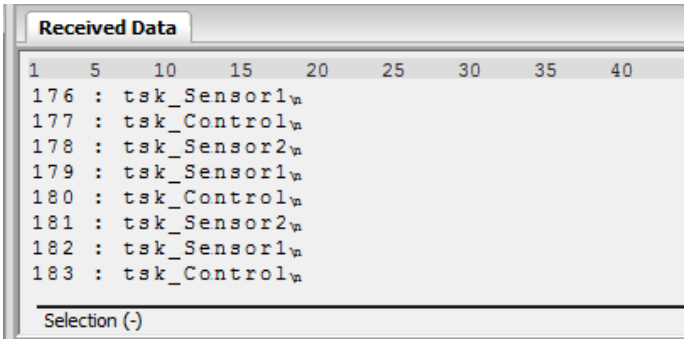
tsk_control has the highest priority, tsk_sensor1 has the lowest priority. In other words, the control task may preempt the two sensor tasks and tsk_sensor2 may preempt tsk_sensor1.

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();
    EE_systick_start();

    //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 69 - 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 achive 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:

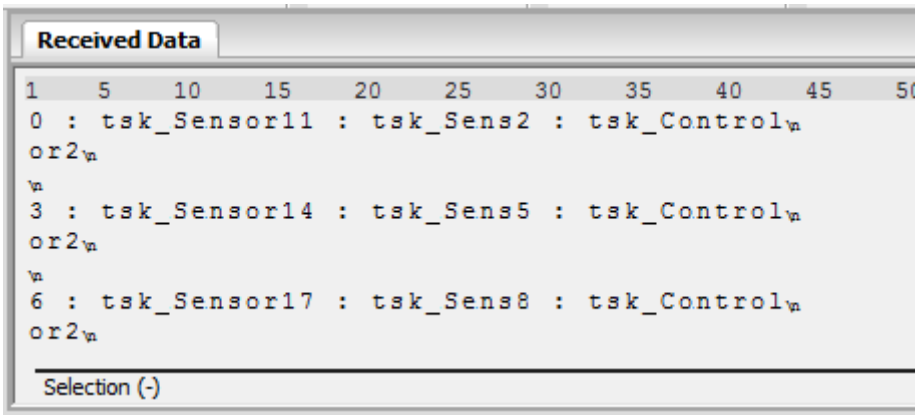


Figure 70 - Corrupted data

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.

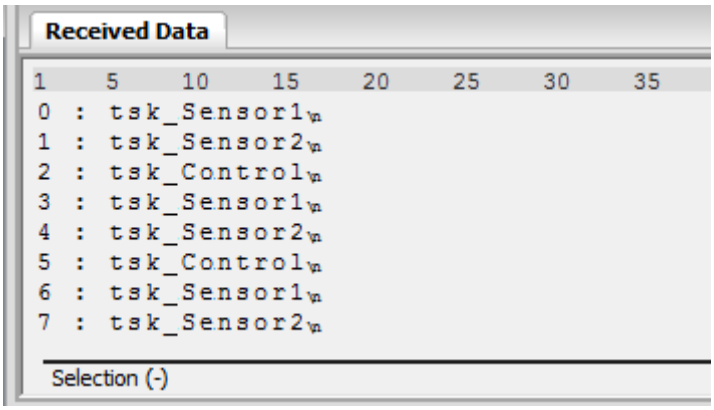


Figure 71 - Corrected output

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.

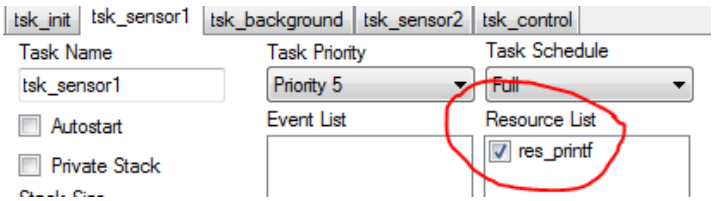


Figure 72 - Ressource activation

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:

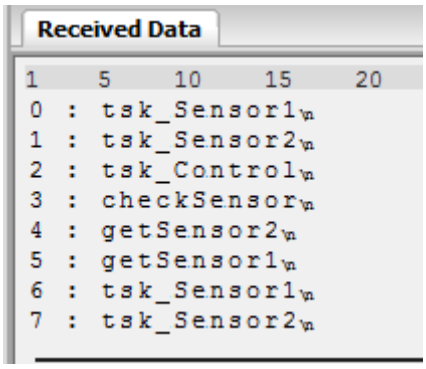


Figure 73 - Output getting/checking the sensor value

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

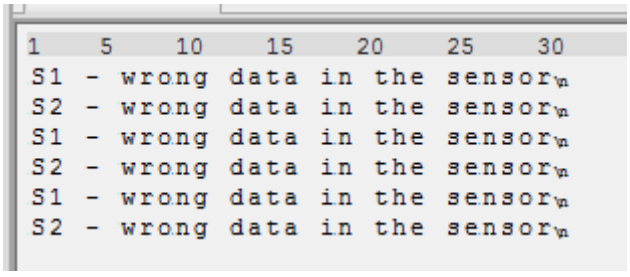


Figure 74 - Error message wrong sensor data

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

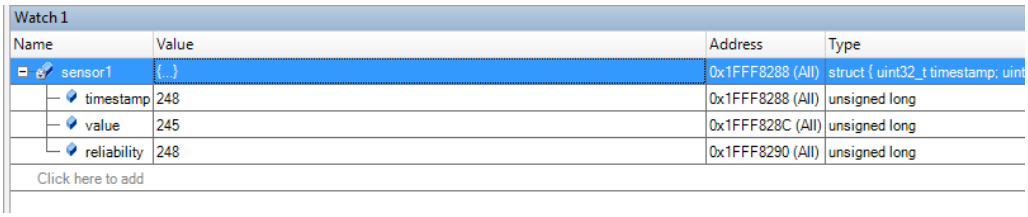


Figure 75 - Wrong sensor value in the debugger

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 <Functsk_sensor1+0x28>)
0x0000023E ldmia.w  sp, {r0, r1, r2}
0x00000242 stmia.w  r3, {r0, r1, r2}
 146:
```

Figure 76 - 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 simplemost 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 occurance.

```
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.

Note: It seems that the integration of the Tracealyser API interferes with the Pre and Post-Task Hooks. I.e. for this exercise, please use an Erika Release prior to 2.5.1.

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.

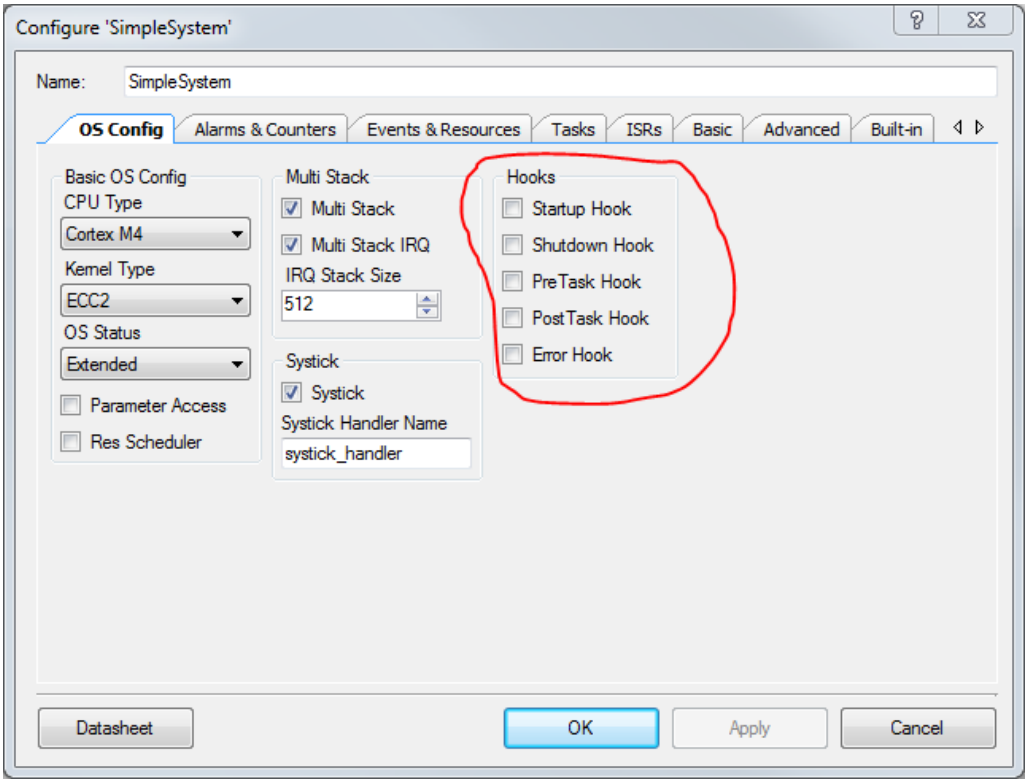


Figure 77 - Hook function activation in the OS Config

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 ErrorHook(StatusType err)
{
    switch (err)
    {
        case E_OS_ILLEGAL_ADDRESS :
            LOG_PRINT("Illegal Adress\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.

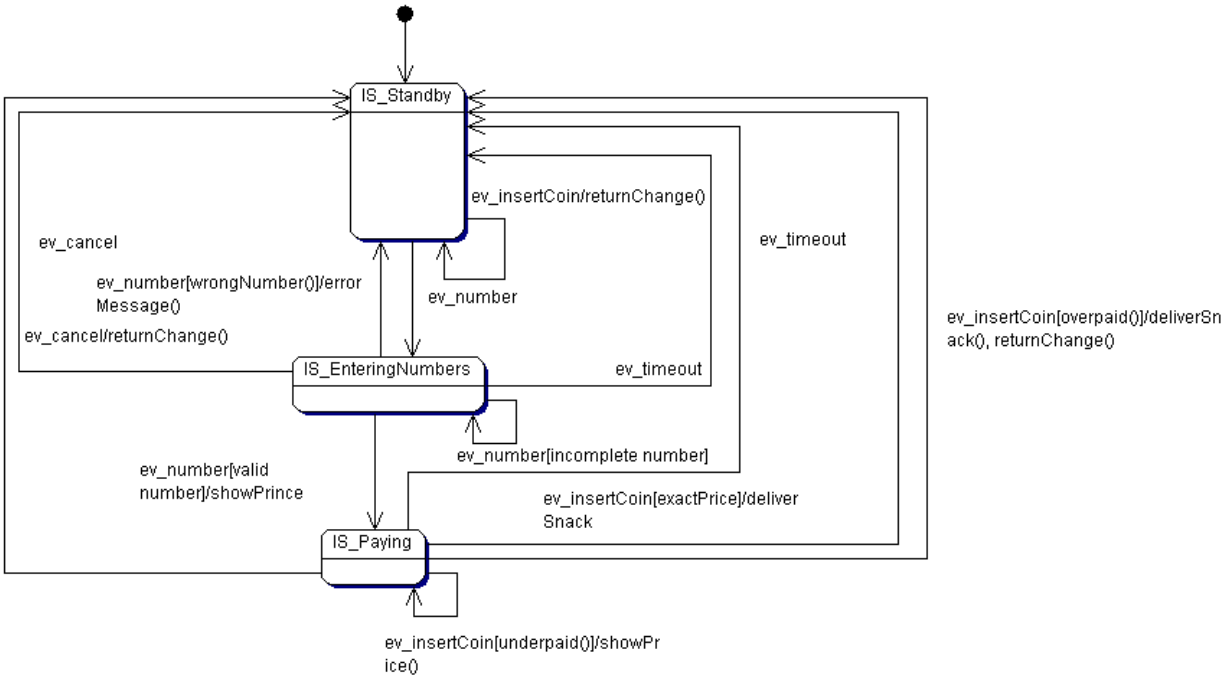


Figure 78 - State Diagram for Snackmachine

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.

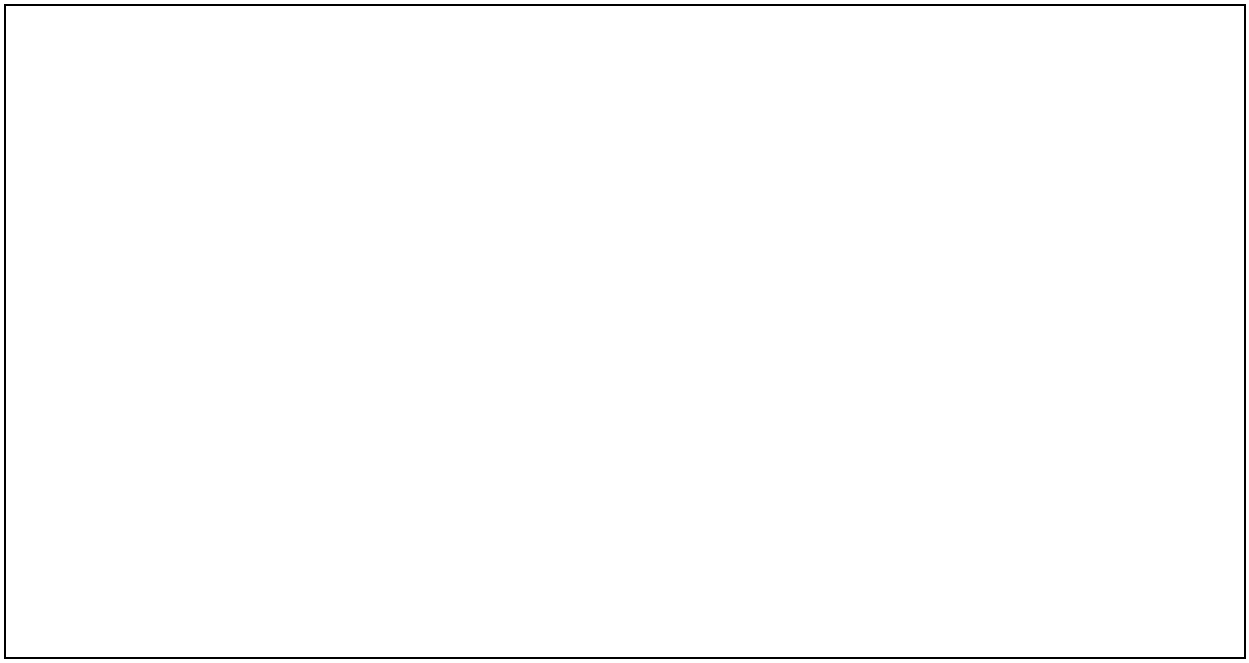
Event 3: _____

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.

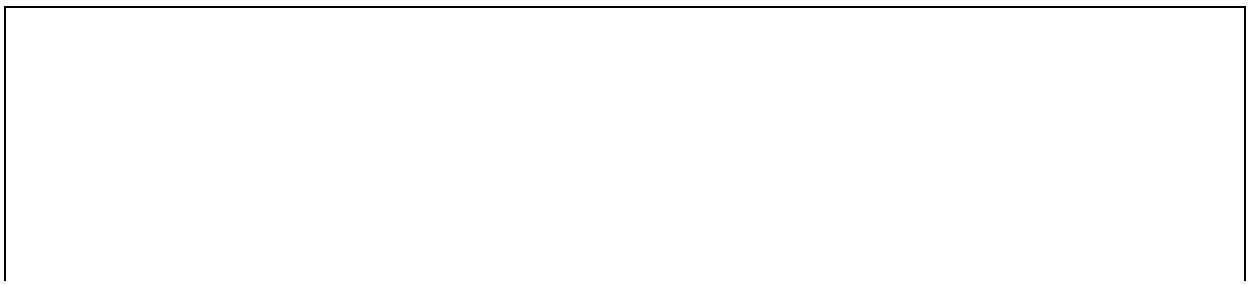


In our application, we will use the alarm `alrm_Tick1ms` which will be fired every 1ms. Every time the alarm is fired, the task `tsk_Timer` will be called. Alternatively, you may use a callback handler for this.

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:

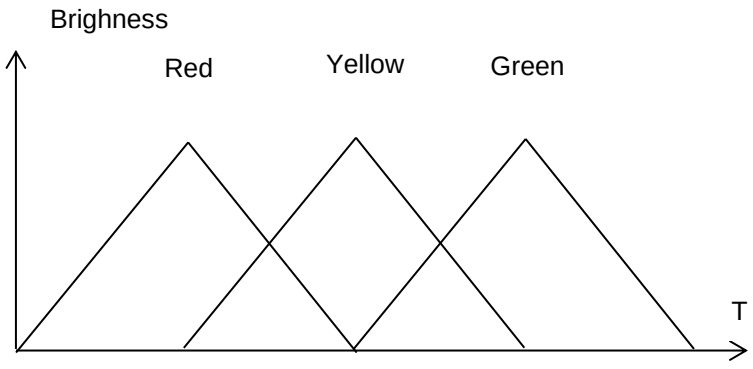
Implement the code for the program based on the given requirements and design.

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 grove, 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: • 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: • 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.

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 machien 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 immediately increased by 1
FR3	If you press the button 2, the volume will be immediately 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: • 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 • 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.

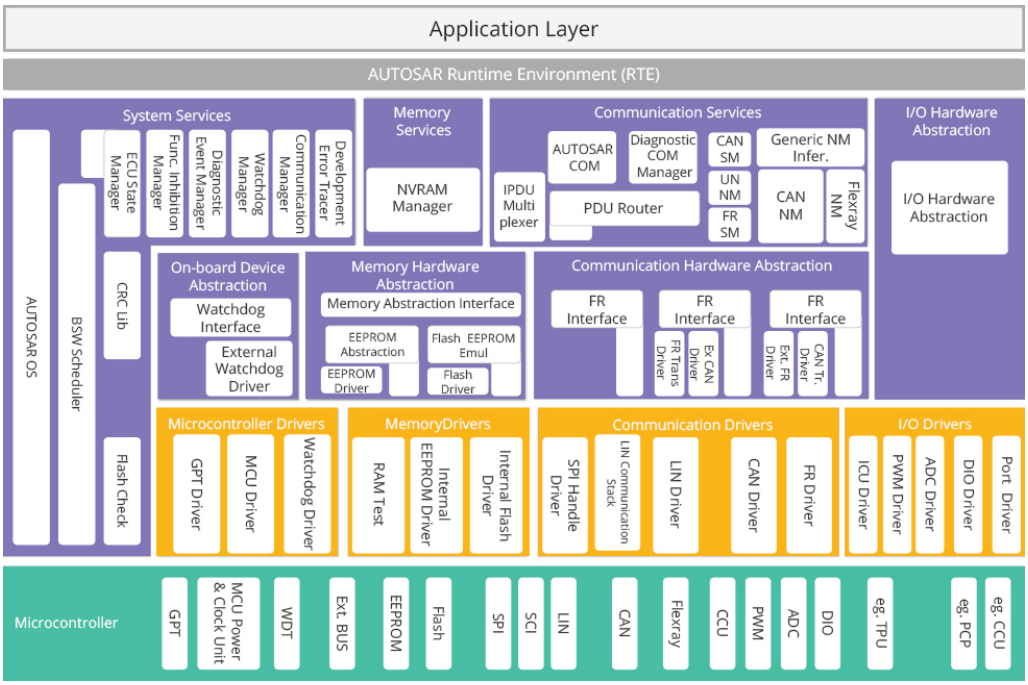


Figure 79 - Autosar RTE[8]

[8] 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 this architecture, 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. These only communicate via RTE signal objects, direct 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 kept very simple.

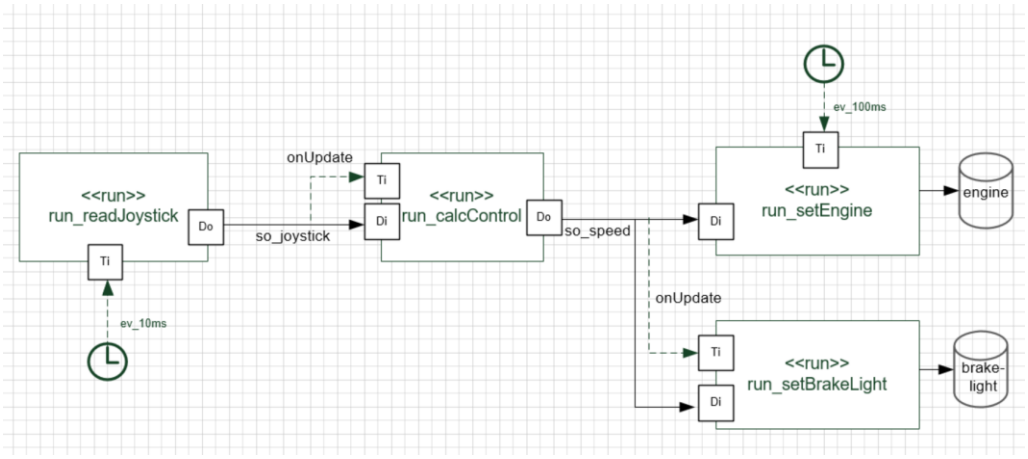


Figure 80 - Electronic Gaspedal ECU

The runnable `run_readJoystick` will be called every 10ms. This runnable will update the `joystick` signal, using the API `pullPort()`. As this signal is realized as asynchronous signal, 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 `speed` 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, `run_setBrakeLight` will be triggered.

`run_setEngine` will be called every 100ms and copy the `speed` signal to the `engine` signal (if it is not too old) and call the driver (`pushPort`). In case of a too old speed signal, the value 0 shall be written to the engine.

`run_setBrakeLight` will also check the `speed` value when it is updated. In case the `speed` 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/ . Please also check the help index.html and readme file provided in the package.

Use the Python Editor to create the required artifacts. Please leave the naming conventions as default. Select the output folders as needed. It is recommended to store a design folder next to the asw, bsw and rte folders (as shown in the lecture)

Based on the model description above, enter the required data in the tool. Once you are done, run the code generator. You will have to localize the generator python script the first time you run it. Please check the help file for this.

Go to the defined `\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.

Create an Erika project having the following resources:

- tsk_Init()
- tsk_Background()
- Application tasks as required
    - you can either create a single task for all runnables. In this case you omit the risk for races, but long running / low priority runnables cannot be interrupted.
    - or you create a separate cyclic and event driven task (simple, but not really good from the functional structure)
    - or you create an io and a control task (better functional structure)
    - note that you will need an additional system task for the age incrementation
- Add Events as defined in the model
- Add Alarms as defined in the model
    - In case you have more than one cyclic trigger, you can also create a single alarm which is firing a callback. Inside the callback you manually activate the tasks / fire the events as needed.
    - The will allow you to use less (expensive) OS resources.

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.

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. Alternatively, you can also use the ADC driver created by PSOC creator.

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 `so_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 – Error Handling

You might have noticed that the RTE offers a variety of error signaling options which can and should be used to detect different error conditions.

- The driver commands `pushPort` and `pullPort` return a possible error code of the driver.

In addition to this driver error code, the signal itself contain explicit and implicit error codes

- The signal status identifies any error condition identified in the RTE, e.g. an error code returned from a driver – this allows the application to verify that valid data is being used.
- The signal age, which measures the time since the last valid update. It is up to the application to decide which age still is acceptable. For a short period, it might e.g. ok to use old data, if you can expect, that the next update will work again.

Besides the data, you should also supervise the activation of runnables – please check the next exercise for this.

Describe a concept for handling the different error types:

Error type	Possible cause	How would you handle this?
Driver		
Signal Status		
Signal Age		

6.1.6 Iteration 6 - 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

Identify critical section and update the RTE configuration as needed. Update the `rte_types.h` file as required (check the comments).

Display task

You might want to use the TFT display to show some data of the process. As this is a rather long running process, the display runnable should be executed in an own, low priority context.

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 a 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 created startup code.
FR4	Create a new software driver called watchdog which provides the functions described below.

FR5	`/**`
	` * Activate the Watchdog Trigger`
	` * \param WDT_TimeOut_t timeout  - [IN] Timeout Period`
	` * @return RC_SUCCESS`
	` */`
	`RC_t WD_Start(WDT_TimeOut_t timeout);`

FR6	Define a sensible enum value for the timing period, based on the provided hardware functionality.

FR7	`/**`
	` * Service the Watchdog Trigger`
	` * @return RC_SUCCESS`
	` */`
	`RC_t WD_Trigger();`

FR8	`/**`
	` * Checks the watchdog bit`
	` * @return TRUE if watchdog reset bit was set`
	` */`
	`boolean_t WD_CheckResetBit();`

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 user input (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 WD_Alive(uint8_t myBitPosition) to the driver, which sets the bit at the corresponding position.
FR5	This function shall be called by every runnable using a uniqe position, i.e. Runnable_0 sets bit at position 0, Runnable_1 sets bit at position 1 and so on.
FR6	In the background task, the WD_Trigger() 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 Electronic Clock

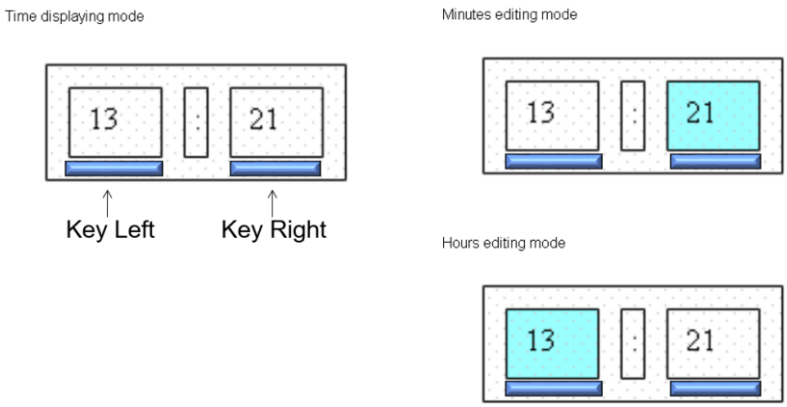
Effort: 8h	Category - B
RTE, Autosar, Statemachine Decomposition, Complex Device Driver	

In this exercise, you will implement the clock example briefly discussed in the lecture. Please use the lecture samples as a reference and modify them as needed. A learning focus is the implementation of a more complex statemachine, i.e. the initial design needs to be refactored using the “Active-Object Pattern” presented in the lecture. Furthermore, we will use the TFT display and need to develop a complex device driver for showing the time. The exercise contains several challenges:

- Refactoring the state machine and creating active objects
- Using the lookup table pattern for the state machine implementation
- Design the signal flow and activation logic
- And last but not least to get the TFT up and running

It is up to you to develop a development strategy. However, it is strongly advised to tackle the challenges one after the other. A possible strategy could be

- Create Flat State Machines using the Active Object Pattern
- Analyze the events and check where they will be fired (by an Alarm, another task, an ISR, the RTE…)
- Work on the input data and develop and algorithm to translate physical button levels into events.
- Design the signal flow considering the identified event sources (please check the annex on the diagram conventions)
- Implement the state machines and active objects (use UART_LOG commands to create some verbosity)
- Add the TFT display driver.

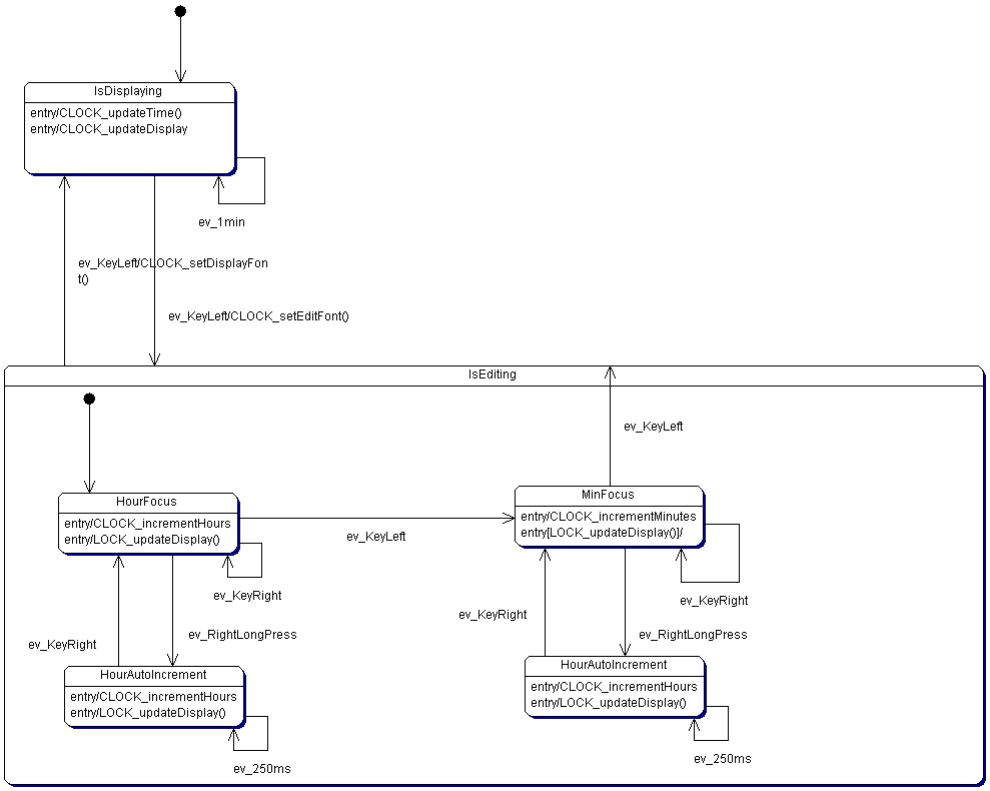


Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS, using the RTE architecture and the Python Code Generator.
FR1	The behavior of the clock is shown in the State Diagram below
FR2	Button 1..2 will be used to control the clock. • Button 1 will create the event ev_KeyLeft • Button 2 will fire the event ev_KeyRight ev_KeyLeft will iterate through the various display and edit modes. ev_KeyRight will be used to increment the minute or hour widget.
FR3	If the right button is pressed continuously for at least 1s, the event ev_RightLongPress will be fired.
FR4	In display mode, the clock numbers shall be shown as white letters on a black background. In Editing Mode, the letters (per widget, i.e. hours OR minutes) shall be shown as black letters on white background.
NFR2	As the display driver blocks the system for a rather long period, the update of the TFT diplay will be done in a low priority task, which is cyclically called.
NFR3	To avoid race conditions, use a resource while copying the clock data to a local display object in the display task.
NFR4	The state machine shall be implemented using a lookup table pattern.

6.3.1 State Diagram

Hierarchical State Diagram



Refactor the Hierarchical State diagram into a Container State Machine and a Widget State Machine using the pattern described in the lecture.

Add the Container State Diagram here!

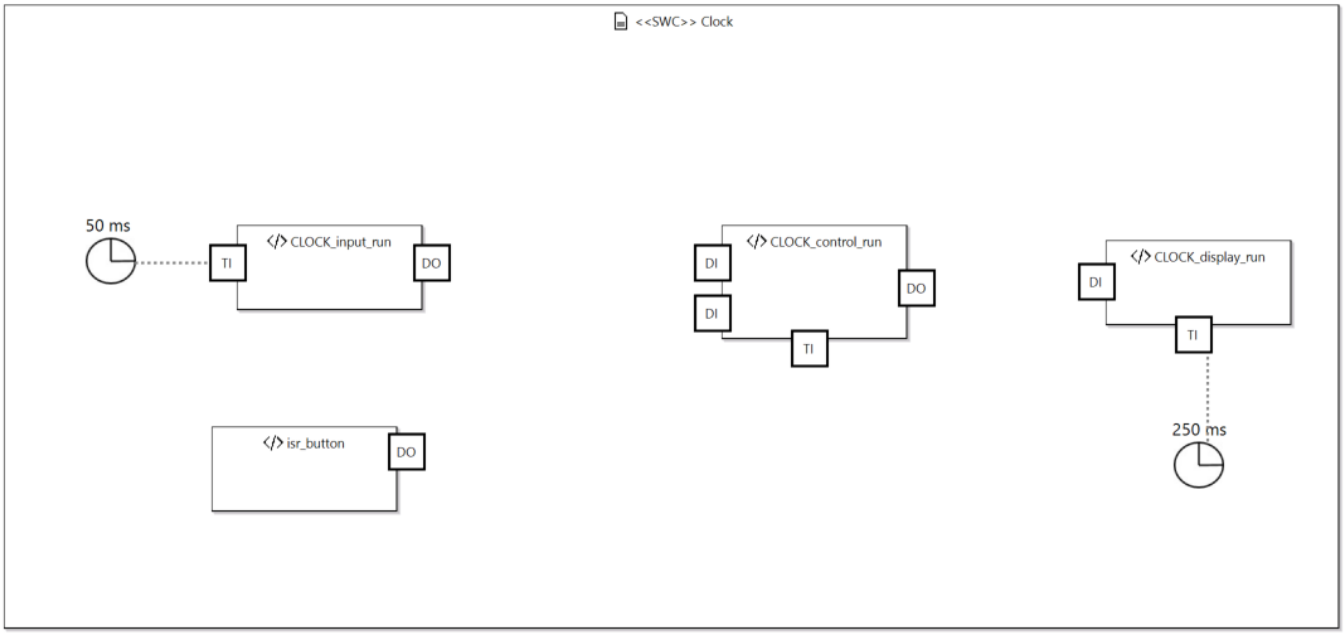
Add your Widget State Diagram here

6.3.2 Signal Flow

We will use the well known RTE pattern (and the corresponding tool) to implement the clock. The clock state machine runnable(s) will be executed in a event task, the display runnable as well as the input runnable, which will create e.g. the longPress event and other time based signals will be executed in a cyclic task.

Analyse the concept for critical sections and add resources where needed.

Complete the Signal Flow Diagram below (note: for representing the various events, use a signal `so_event`, which will have an `onUpdate` Trigger for the state machine).



Writing to the same global signal `so_event` from 2 different contexts may result in a loss of data / race – even when you have added resources around the setter / getter functions. Explain why!

How could you improve the signal flow above to avoid this possible race? Note: You may modify / delete / add the runnables if needed.

6.3.3 Signals versus global variables

In the lecture we have presented a concept to implement the complete clock including the state machine in a signal object. This approach has some advantages and disadvantages:

Advantage:

- Any global data is a signal, which is located in a well defined memory region (signal pool) supporting simple MPU configuration (if the controller would have a MPU)
- Every signal object has a well defined API incl. protection of critical sections
- The signal metadata allows error detection (signal status, signal age)
- The consumers requiring the data can be automatically notified (`onData` and `onError` events)

Disadvantage:

- Updating the signal becomes a bit messy, as the API only supports complete read/write operations, but no access to “payload-methods” like
  `processEvent(ev)`

Another example, where this limitation becomes obvious would be e.g. a ringbuffer signal. We want to use the `read` and `write` method to read the data, but not a complete assignment operation implemented by the `set` and `get` operation.

Depending on the requirements, we could either provide a pointer or reference access to the payload, which would allow us to use specific data read/write methods on the payload directly (introducing some new challenges) or we could use global variables (which of course misses the advantages of signals).

Considering the above, which data items would you implement as global object and which objects would you implement as signals in this exercise?

Object	Global / Signal	Justification

6.3.4 Implementation

For the implementation of the state machine, use one of the Lookup table patterns presented in the lecture.

- Use the RTE Generator to create the code framework.
- It is recommended to start with the implementation of the input runnable. Make sure, that you get the different events based on the pressing duration of a button.
- In a next step, it is recommended to implement the container state machine and to add actions for the dispatcher functions, printing the dispatched events.
- Then continue with the widget state machines.
- And add a display signal.

Some additional hints:

- Check the statemachine code and create a statemachine class / structure, containing pointers to the state machine configuration as well as a state variable.
- In a first step, you may implement the code for three different state machine, i.e. container, hour widget and minute widget.
- The logic for the hour and minute widget is almost identical, the difference is only the ticktime and range – how could you add that to the structure. One idea: instead of passing the state machine object to the processEvent function, you could pass the active object. As C does not provide polymorphism, this might be a bit tricky…
- You can place the implementation for the state machine in the `clock_type.h|.c` file. You may use the code provided as sample code from the lecture as starting point, but you will have to adopt it to your needs.
- Alternatively, you can place it in the control runnable. This is probably the preferred way especially if you use globals to represent the clock statemachines.

- If you want to use LOG_I or other functions using sprint(), you have to allocate AT LEAST 800 byte memory for extended task stack sizes. Add 400more bytes for every additional function level calling LOG.
- For the TFT driver, download the package from Moodle. Check exercise 2.3 “Getting the TFT display up and running” for additional information on the package.

6.3.5 Test

Create systematic testcases using the state machine test strategy as presented in the lecture.

6.3.6 Simplifications

As this exercise is pretty complex, you may consider the following simplifications for a first implementation

- Implement a single flat statemachine instead the active objects
- Simplify the state machine logic by only considering the event EV_KEYRIGHT for setting the clock, i.e. the auto-increment state is ommitted

6.4 Matlab Embedded Coder

7 Distributed Systems

For more complex applications, networking and service distribution adds another level of complexity. What are the requirements for an efficient protocol? How are communication stacks integrated into an embedded architecture? These and more questions will be addressed in this chapter.

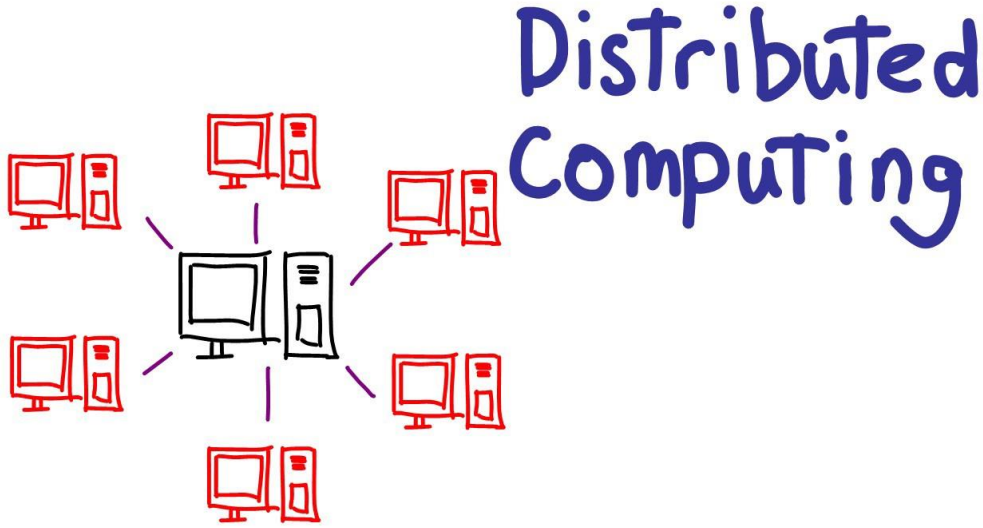


Figure 81 - Distributed Systems and Networking

7.1 UART JSON Parser

Effort: 8h	Category – B, C
ISR/Task Handshake, Dynamic Messaging, JSON, External Library, TFT display	

Note: This exercise is a bit more complex and will require some design decisions from your side. It is recommended to develop the iterations in steps. The first iterations are still pretty straight forward, the later iterations will require more experience.

In this exercise, you will implement a JSON Parser, which will read in a JSON string over the UART, parse it and display the transmitted drawing commands on the TFT display of the labboard. This will involve three different contexts:

- The data will be received in a UART isr and stored in a ringbuffer
- Once a complete JSON string is received, this string will be parsed in the high priority task tsk_json
- If a drawing command was found, this command will be transmitted via a messaging service to low priority tft drawing task tsk_hmi.

7.1.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS.
NFR2	Use existing functions of the system as far a sensible and possible.
NFR3	We will use the library https://github.com/zserge/jsmn for JSON parsing
NFR4	Put the developed code in sensible files and functions. Avoid global data unless absolutely required! A good choice would e.g. to create a parser.c|.h and drawer.c|.h
NFR5	You will find a couple of header files with the complete data types and API definitions in Moodle. Use these files without modification and add implementation files as needed.
FR1	The UART ISR will receive the incoming data and after checking the correct reception will store the byte in a global dynamic message buffer
FR2	In case of a reception error, a sensible error handling needs to be implemented, e.g. discard all bytes of the current JSON string and resynchronize with the next transfer. Of course, an error message shall be provided to the user.
FR3	The receiving task will parse the JSON string and send the content to a low priority output task, using a messaging system
FR4	Parsing errors must be detected and handled
FR5	The output task will use the TFT display to draw the content of the JSON

	protocol.

7.1.2 Background

“JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language Standard ECMA-262 3rd Edition - December 1999. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures.

In JSON, they take on these forms:

An object is an unordered set of name/value pairs. An object begins with {left brace and ends with }right brace. Each name is followed by :colon and the name/value pairs are separated by ,comma.”[9]

Some examples:

- { "name" : "Jack", "age" : 27 }
- { “array” : [1, 2, 3], “coordinate” : { “x” : 3, “y” : 4} }

7.1.3 Iteration 1 – Simple ISR / Task Handshake

We will use an RX ISR to process the incoming data over the UART. Please check chapter ”1.3 Introducing interrupts” for background information on how to set up interrupts in ERIKA.

The job of the ISR is as follows:

- Receive a byte of data and store it in a global ringbuffer object (to be implemented by yourself)
- Whenever an end-of-string character is received (e.g. 0), an event is fired to a task tsk_json
- The task waits for the event. Then it reads out the data of the ringbuffer and displays it via the UART_LOG (KitProg USB)

Some notes on the ringbuffer implementation
A ringbuffer is a FIFO buffer. It typically stores the current read and write index as well as the number of stored bytes and the size as parameters. An Init function sets up the buffer (allocates the memory

[9] https://www.json.org/json-en.html

and inits the attributes), a put and a get routine will store / read one element of data (if possible) and a clear function resets the buffer.

As you may have several buffer objects in a project, it is recommended to implement the buffer in an object oriented fashion.

7.1.4 Iteration 2 – Dealing with races

The implementation of the first iteration however has one disadvantage – it is probably not threadsafe. As the data read and write operations are happening asynchronously (in many cases), we must check this a little bit more in detail.

Potential Problem 1 – critical regions / data corruption

As the ringbuffer is a global object, its data may be corrupted in case of conflicting access operations. You might remember from the lecture, that atomic data is not affected by this, so a (wrong) conclusion might be – as long as the payload is atomic, the access is not critical. Why is this assumption wrong? The access to the data is controlled by the ringbuffer attributes (also called metadata), i.e. the filllevel, read and write index and size.

- A write operation will only happen, if the ringbuffer is not full (aka fillevel < size)
- A read operation will only happen, if the ringbuffer contains data (aka filllevel > 0)

Let's have a look at a typical read/write operation.

```
RC_t RB_put(RB_ringbuffer_t* const me, rb_data_t data)
{
    if (me->fillLevel < me->size)
    {
        me->buffer[me->writeIndex++] = data;
        me->writeIndex %= me->size;
        me->fillLevel++;

        return RC_SUCCESS;
    }
    else
    {
        return RC_ERROR_BUFFER_EMTPY;
    }
}
```

As the metadata in the implementation above is updated after the access to the buffer, we can argue as follows:

1. If a write operation is interrupted by a read operation (bad style anyway) after the data has been added and before the metadata is updated (and the buffer is empty[10], the read operation will return an empty buffer error. No real problem, let's try again later.
2. If a read operation is interrupted is interrupted by a write operation at the same position and the buffer is full[11], the buffer will return a full buffer error and the data cannot be written. Error handling needs to be done by the application.

[10] How about a non empty buffer? In this case it will simply read an older value. fillevel will be decremented by read and incremented by write after the interruption, the indexes are updated independently.

3. How about a write operation, which is interrupted by another write operation? In this case the payload data may be overwritten (if the write index is not incremented and we might see an invalid overrun of the buffer, as the fillevel will be incremented twice. I.e. a buffer having only one free position will be corrupted.
4. How about two read operations? Here, a similar problem may appear, the read index is incremented by 2, i.e. one data element may be read twice, while another is not read at all. Or we read a previously old element, if the buffer only contains 1 more element.

What does this imply?

As long as we have exactly one read and one write process, the access to the ringbuffer will be threadsafe, especially if the payload allows atomic access. In case of a non-atomic payload, the data which will be used for the write operation may be corrupted by the interrupt (e.g. if it is global data) or not (if it is local data and no stack corruption or other nasty problems are happening.

As soon as we have several read/write operations (not a typical use case), data corruptions may happen, especially related to the metadata.

How can we solve this issue? The easiest way is to put a semaphore around the critical region. Let's do this!

Potential Problem 2 – synchronization issues

Let's assume the following sequence

- Data "Hello world\0" is written to the UART
- Event is fired
- Some delay happens (e.g. higher priority task or another interrupt) before the receiver task reads out the data
- Data "!!!\0" is written to UART
- Delay has ended
- As a consequence, the receiver task now reads out the complete but wrong data "Hello world\0!!!\0" in one cycle, although this should have happened in two activations.

Obviously, the problem in this case is the very simple implementation of the event masks. If two events are sent without receiving and clearing the event mask after the first event, one event will get lost.

Events are not buffered in Erika.

How can this problem be fixed?

Option 1 – Receiver checks for multiple data packages

The receiver parses the data and terminates the processing of the data when the EOP character (\0) is received. This is a very simple and efficient way of solving the problem, but requires a clear and unambiguous EOP character (e.g. by introducing escaped characters[12] to avoid conflicts).

[11] In case if a non full buffer, the data simply will be written. Same story for the metadata as in the previous case.

[12] https://en.wikipedia.org/wiki/Escape_character

However, what happens, if a second protocol is in the buffer? In this case the protocol will not be read out, as one event was lost. Only if the next protocol n has been received, the protocol n-1 will be processed, not what we really want to have.

Instead of only reading out one protocol after the event, we need to check if another complete protocol is in the buffer and read this out as well. And of course a third and forth protocol might be there as well…

How can you find out if another complete protocol is in the buffer?

Option 2 – Sender informs receive about protocol size

We could send the receiver task the number of bytes of the current protocol, e.g. in a global variable. However, would only be a hacky implementation of the ideas of option 1 and this would not be thread-safe at all, as the global variable might get overwritten by the second transmission. So, let's forget this solution right away.

Option 3 – Extending the ringbuffer to a flexible message buffer

We might send a complete protocol in a ringbuffer cell, e.g. by storing strings or dynamic arrays. This sounds good, but it has some challenges:

- How can we add a received byte to a string stored in a ringbuffer. Possible, but very complex in terms of memory management and probable expensive in terms of CPU time.
- Furthermore, we do not know the size of the string until we have received the EOP character. I.e. we need to provide a sufficiently large intermediate buffer.
- For the real transmission, we have to create a ringbuffer element large enough for storing the payload. The memory will be allocated upon writing the data, and freed upon reading the data.

This solution is very elegant, but pretty complex and it requires some complex memory management. Especially if several buffers are in use, we might phase memory holes after some time.

Tasks:

- Identify all critical regions of the ringbuffer and protect them using Erika Resources. Add comments to the code explaining why you think that this region is critical.
- Implement Option 1 to solve the synchronization issue.

- How can you test your implementations? Provide testcases to verify the correct implementation of the resources and synchronization mechanism.

Note: If we have a scenario, in which we do not know the size of the data until we have received it, a normal ringbuffer typically is the best choice for storing it. However, we need to take caution when clocking out the data on the receiver side to avoid races.

7.1.5 Iteration 3 - Adding a JSON Parser

We will use the JSON parser “Jasmine” provided under https://github.com/zserge/jsmn

The parser consists only of one file, which is provided in Moodle. Add the file to your project and study the API. Before you start coding, create a design by answering the following questions.

- How do you pass the data from the ringbuffer to the JSON parser?
- What kind of error handling will be required, on which levels?
- As the program will become a bit more complex, adding UART traceability messages are highly recommended.

Test the parser by sending some JSON Test protocols, e.g.

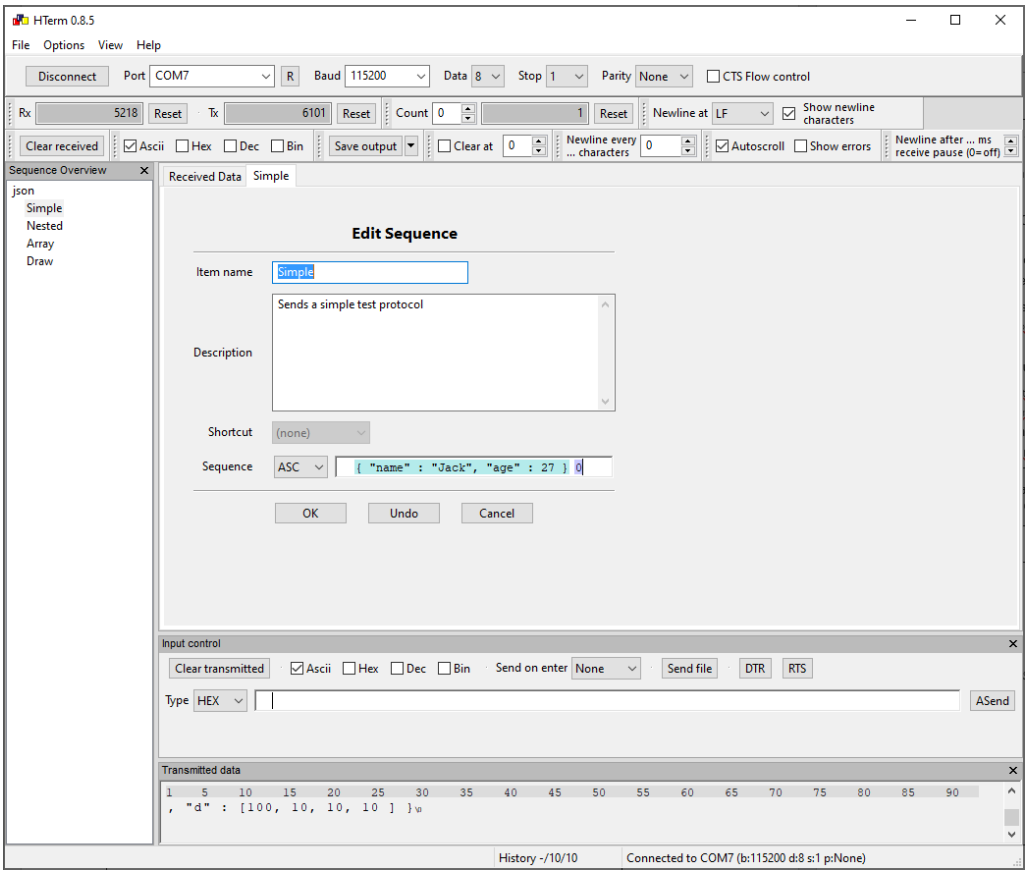
```
//*********************************************
// { "name" : "Jack", "age" : 27 }
// { "person" : { "name" : "Jack", "age" : 27 } }
// { "array" : [1, 2, 3], "coordinate" : [1, 2] }
// { "coordinate" : { "x" : 3, "y" : 4} }
```

and by printing the received string and the parsed tokens.

Sample output for the first test protocol:

```
tsk_json : Processing data { "name" : "Jack", "age" : 27 } length 32
tsk_json : Parsed token 0 - JSMN_OBJECT: { "name" : "Jack", "age" : 27 }
tsk_json : Parsed token 1 - JSMN_STRING: name
tsk_json : Parsed token 2 - JSMN_STRING: Jack
tsk_json : Parsed token 3 - JSMN_STRING: age
tsk_json : Parsed token 4 - JSMN_PRIMITIVE: 27
```

Hint: When using HTerm, you can store test protocols in a file for simplified access. You can mix ASCII and HEX output to add the terminating 0 character to your string. An HTERM file containing some sample protocols is provided in Moodle.


HTerm 0.8.5
File Options View Help
Disconnect
Port COM7
Baud 115200
Data 8
Stop 1
Parity None
CTS Flow control
Rx 5218 Reset
Tx 6101 Reset
Count 0
Reset
Newline at LF
Show newline characters
Clear received
Ascii Hex Dec Bin
Save output
Clear at 0
Newline every ... characters
Autoscroll
Show errors
Newline after ... ms receive pause (0=off)
Sequence Overview
json
Simple
Nested
Array
Draw
Received Data
Simple
Edit Sequence
Item name
Simple
Description
Sends a simple test protocol
Shortcut
(none)
Sequence
ASC
{ "name" : "Jack", "age" : 27 } 0
OK
Undo
Cancel
Input control
Clear transmitted
Ascii Hex Dec Bin
Send on enter None
Send file
DTR
RTS
Type HEX
ASend
Transmitted data
1 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90
, "d" : [100, 10, 10, 10] }
History -/10/10
Connected to COM7 (b:115200 d:8 s:1 p:None)

Check the following test case

- Transmit a short JSON string (OK)
- Then transmit a long JSON string (OK)
- Then transmit a short JSON string (probably not NOK)

What bug did you note?

Analyse the code of the library – how can you fix this?

Hint: In a first step, you may add the code directly into the parsing task. As this task will become pretty large, we will clean it up once it works and place the code into a parser file, which will abstract the JSMN parser in an object oriented way. For this, please carefully read the documentation and check the code in order to void out how to avoid multiple definitions, which probably will show up as you include the jsmn.h file more than once.

Add the file parser.h to your project and implement the functions as specified in the header file.

7.1.6 Iteration 4 – Adding Semantics to the parser

The provided JSON library is pretty helpful, but of course it does not know anything about the content of the JSON string. For drawing objects on the screen, we will work with the following JSON protocol. It will draw lines (and more) on the display using the following conventions (extend them as needed, wanted):

- command “c” for color setting, followed by a color string
- command “d” for drawing, followed by 4 values for the origin and target coordinates x1, y1, x2, y2

Example protocol for a red square:

```
{ "c" : "red", "d" : [10, 10, 10, 100], "d" : [10, 100, 100, 100], "d" : [100, 100, 100, 10], "d" : [100, 10, 10, 10 ] }
```

Add the file drawer.h to your system, which will be based on the parser. Other than the generic parser, the drawer knows and understands the semantics of the above described protocol and translates the string into a drawer object (given in the header file).

Implement the function to translate the found JSON tokens into a drawer object. and test the correction functionality of the functions.

7.1.7 Iteration 5 – Adding a messaging mechanism

In the final iteration, the parsed drawer objects will be sent as messages to the HMI task. The HMI task will draw on the TFT, a rather time consuming story. Therefore the priority of the HMI task will be lower as compared to the communication tasks.

Other than FreeRTOS or PXROS, OSEK does not have a flexible (in terms of support of any message size) threadsafe messaging mechanism. However, such a mechanism can be implemented by combing a ringbuffer storing objects with dynamically allocated memory for the payload with events.

In a first step, please complete the given data structure below. The idea is to have a ringbuffer `messagebox_t`, which is storing `message_t` objects. Every `message_t` object consists of a `uint16 m_size` attribute and a `void * m_pBuffer`, pointing to a buffer which is allocated during the transmission of a message and deleted afterwards.

The `messsagebox_t` ringbuffer in addition to the typical attributes of a ringbuffer also stores an `EventMaskType m_ev`, containing the event which will be fired in case of a message to be sent and a `TaskType m_task`, containing the receiving task id. The size of the ringbuffer is given by the constant `POSTBOX_SIZE`.

Hint: copy the already implemented ringbuffer and rename it to message. Then, extend and modify it as needed

```
typedef struct
{




} message_t;

#define POSTBOX_SIZE    10
typedef struct
{









} messagebox_t;
```

Below, please find a detailed description of the sending routine. The reception routine is implemented in a similar way.

- If the ringbuffer has a free slot, a message object will be created and send out. In case of a full ringbuffer, the error code `RC_ERROR_BUFFER_FULL` will be returned
- The memory for the message will be allocated using the `malloc` operation. In case no memory can be allocated (malloc returned 0), the error `RC_ERROR_MEMORY` will be returned.
- The data to be transmitted is copied into the allocated memory. Please note the `void*` data type. For copying the data, you will have to cast the pointer to a suitable pointer type, considering, that any size of date (and therefore any alignment) needs to be considered. `Memcopy` or any other library function therefore should not be used for copying the data.
- The size will be stored and the other attributes will be updated.
- The event will be fired to the receiver task.
- When everything worked as expected `RC_SUCCESS` will be returned.
- Please note for this simplified implementation, that you do not have to consider reentrancy and/or threadsafety.

Tasks:

- Complete the message mechanism using the API below.
- Implement a JSON interpreter, which will translate the given JSON strings and primitives into drawing transmission packages. Define appropriate structures for this.
- Then, use the messaging concept to transmit the drawing object to the HMI task.
- Inside the HMI task, use the drawing object to draw the content

7.1.8 Iteration 6 – Drawing on the display

Implement the receiver task tsk_hmi receiving the messages of the tsk_json.

Add the TFT library to your project and add an SPI component (hint: check the corresponding chapter of this workbook) .

Then read the content out of the transmitted message and draw it on the screen.

Messaging API

```
/**
 * Initialises the ringbuffer and sets up the memory for the metadata
 * \param MSG_messagebox_t *const me    : [IN/OUT] Message Box Object
 * \param uint16_t const size           : [IN] Size of the ringbuffer
 * \param EventMaskType const ev        : [IN] Event which will be fired in case of
a new message
 * \param TaskType const task           : [IN] Receiver task of the event
 * \return RC_t: RC_SUCCESS in case of no error else error code
 */
RC_t MSG_init(MSG_messagebox_t *const me, uint16_t const size, EventMaskType const
ev, TaskType const task);


/**
 * Send a message by creating a temporary transfer buffer,
 * copying the data into this buffer and firing an event
 * to the receiver task
 * \param MSG_messagebox_t *const me    : [IN/OUT] Message Box Object
 * \param void const* const pData       : [IN] Data to be transmitted
 * \param uint16_t const size           : [IN] Size of the transmitted data
 * \return RC_t: RC_SUCCESS in case of no error else error code
 */
RC_t MSG_sendMessage(MSG_messagebox_t *const me, void const* const pData, uint16_t
const size);


/**
 * Returns the size of the next message in the buffer.
 * Required for providing storage on the receiver side
 * \param MSG_messagebox_t *const me    : [IN/OUT] Message Box Object
 * \param uint16_t *const size          : [OUT] Size of the next message, 0 in case
of no message
 * \return RC_t: RC_SUCCESS in case of no error else error code
 */
RC_t MSG_getSizeOfNextMessage(MSG_messagebox_t *const me, uint16_t *const size);


/**
 * Wait for the next message (using OS services)
 * \param MSG_messagebox_t *const me    : [IN/OUT] Message Box Object
 * \return RC_t: RC_SUCCESS in case of no error else error code
 */
RC_t MSG_waitNextMessage(MSG_messagebox_t *const me);


/**
 * Reads the message from the transfer buffer and
 * releases the allocated memory
 * \param MSG_messagebox_t *const me    : [IN/OUT] Message Box Object
 * \param void *const pData             : [OUT] Pointer to the memory for storing
the received message
* \return RC_t: RC_SUCCESS in case of no error else error code
 */
RC_t MSG_receiveMessage(MSG_messagebox_t *const me, void* pData, uint16_t const
size);
```

7.2 CAN Communication

Effort: 4h	Category – B
CAN Communication, Basic CAN, Full CAN, Handlers	

Note: This exercise is a bit more complex in terms of technology and debugging. Furthermore you need 2 boards and a CAN connection cable for the exercise.

In this exercise, you will implement a simple CAN communication between 2 labboards. Both boards will read in the joystick as well as the buttons, transfer this data into a CAN protocol and send the data to the other board.

The receiver board will display the data on the TFT.

This exercise serves as a starting point for the CanOpen exercises as well as the distributed game examples like soccer.

CAN networks are designed for short messages with data length not more than 8 bytes, and a bit rate up to 1 Mbps.

The CAN protocol offers several advantages over other serial communication protocols:

- CAN is a message-based protocol; CAN network nodes are not assigned specific addresses. This provides flexibility to add or remove a node from the network without affecting the rest of the network. In addition, if one of the nodes fails, the others continue to work and communicate properly.
- CAN messages can be prioritized.
- CAN has five levels of error checking, to ensure reliable traffic and data integrity.
- The CAN network has system-wide data consistency, that is, if a message is corrupt at a receiving node, the message is not accepted by any of the other receiving nodes.
- Corrupted messages are automatically retransmitted as soon as the bus is idle again.

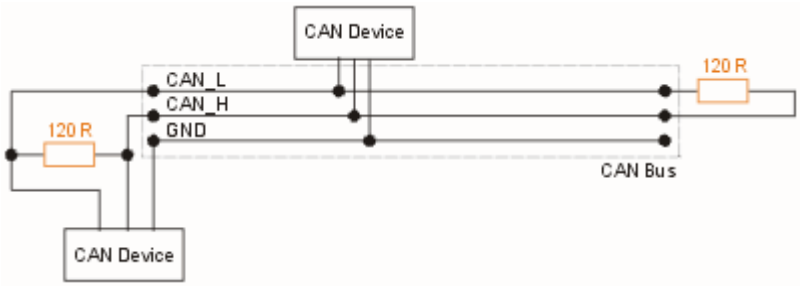
7.2.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS and the RTElight.
NFR2	Use existing functions of the system as far a sensible and possible.
NFR3	Put the developed code in sensible files and functions. Avoid global data unless absolutely required – e.g. for interfacing the handlers!
FR1	The 4 buttons as well as the joystick position will be read every 50ms.
FR2	The data will be copied into a CAN signal for transmission.
FR3	On receiver side, the CAN signal will be copied into a display signal and the data will be shown on the display.
FR4	Optional extension: You may connect more than 2 boards to the network

	and develop a network management concept to send the data to a specific receiver only.

7.2.2 Wiring

The diagram below describes the basic wiring of a CAN network. Please note the EOL resistor, which must be set at the outer nodes. The labboard has a jumper to set the EOL.



7.2.3 Iteration 1- Sending protocols using Basic CAN

The CAN bus provides a variety of features, many of them are vendor specific. In order to get started with a very simple first use case, we will configure our CAN node in BASIC mode.

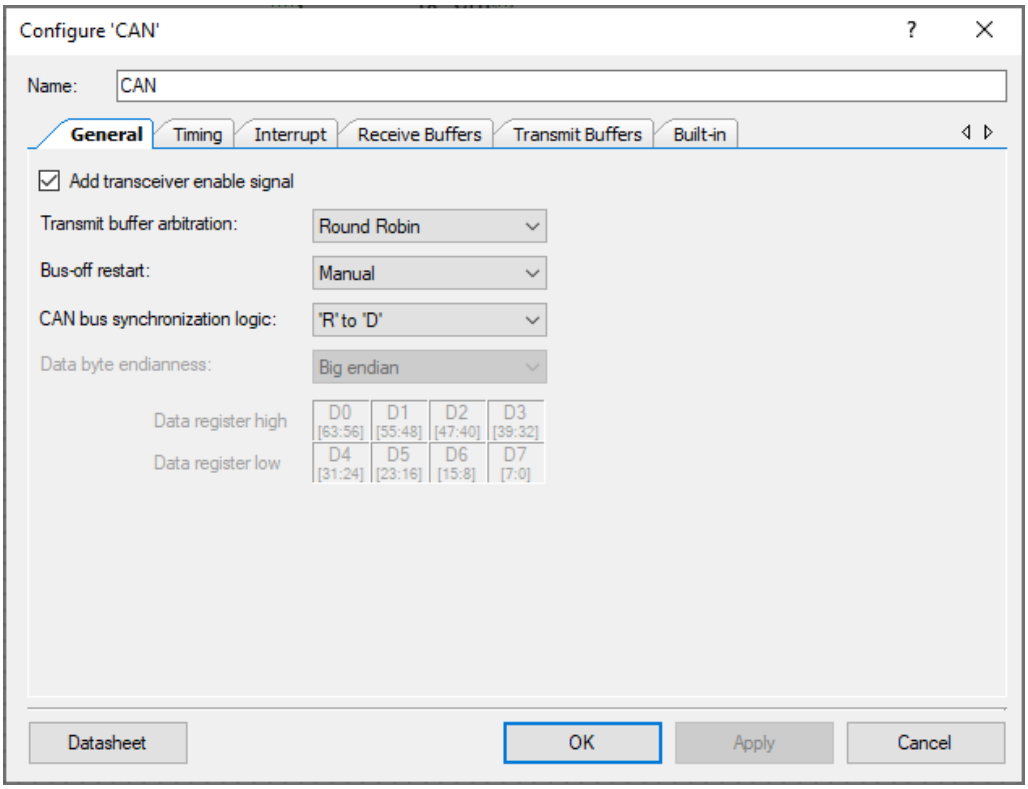
PSoC offers two different modes for CAN communication: Full CAN and Basic CAN. Full CAN take strong advantage of the hardware and provides a more simple to use SW interface, but is slightly limited in transmission features. The following are the important differences between a full CAN message and a Basic CAN message:

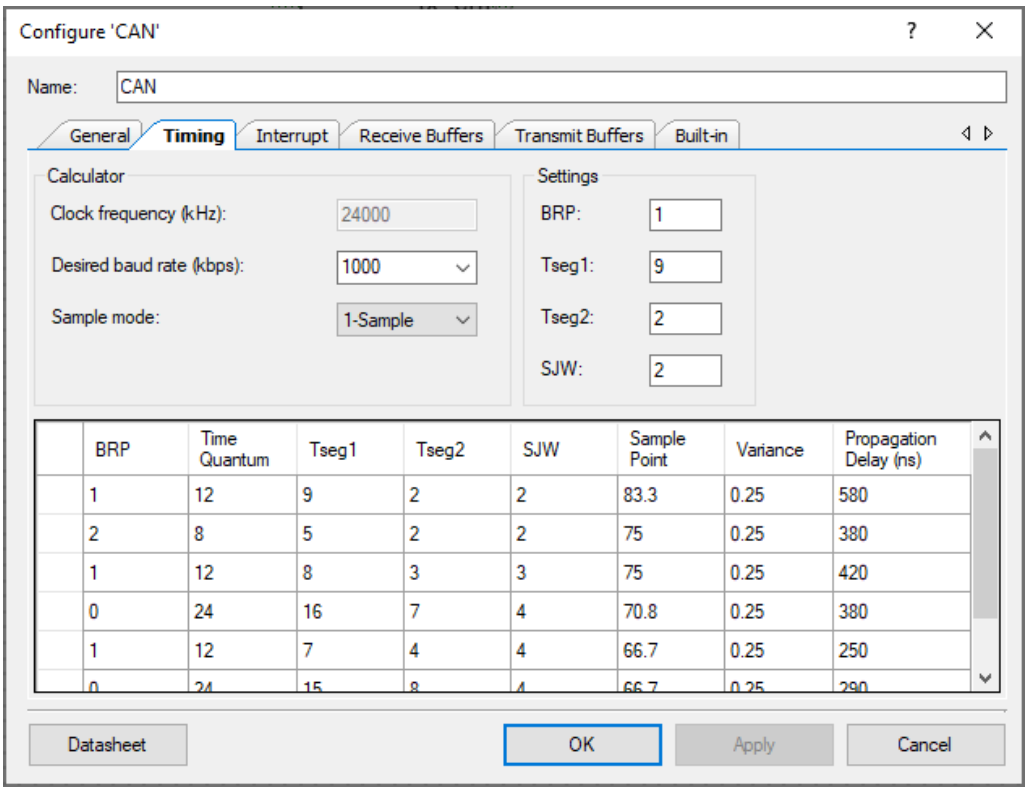
- A Full CAN communication can be easily set up with the help of a GUI, with a very limited amount of programming involved. Basic CAN requires all of the parameters to be set in firmware.
- Full CAN uses hardware for message filtering. Basic CAN requires the CPU to be interrupted each time a message is received, to determine whether it is accepted or not.
- Full CAN can only receive a single type of message for each mailbox whereas Basic CAN can accept messages with a range of identifiers for each mailbox.

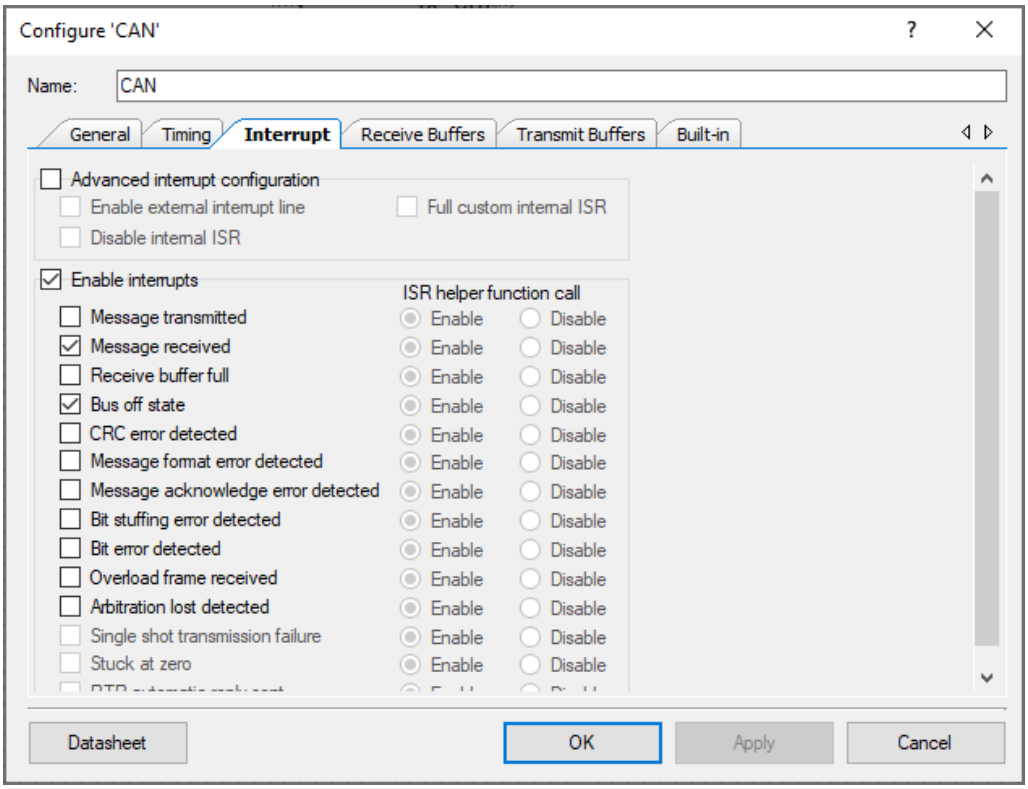
For details, pleae check the document “PSoC 3 and PSoC 5LP – Getting Started with Controller Area Network (CAN)”.

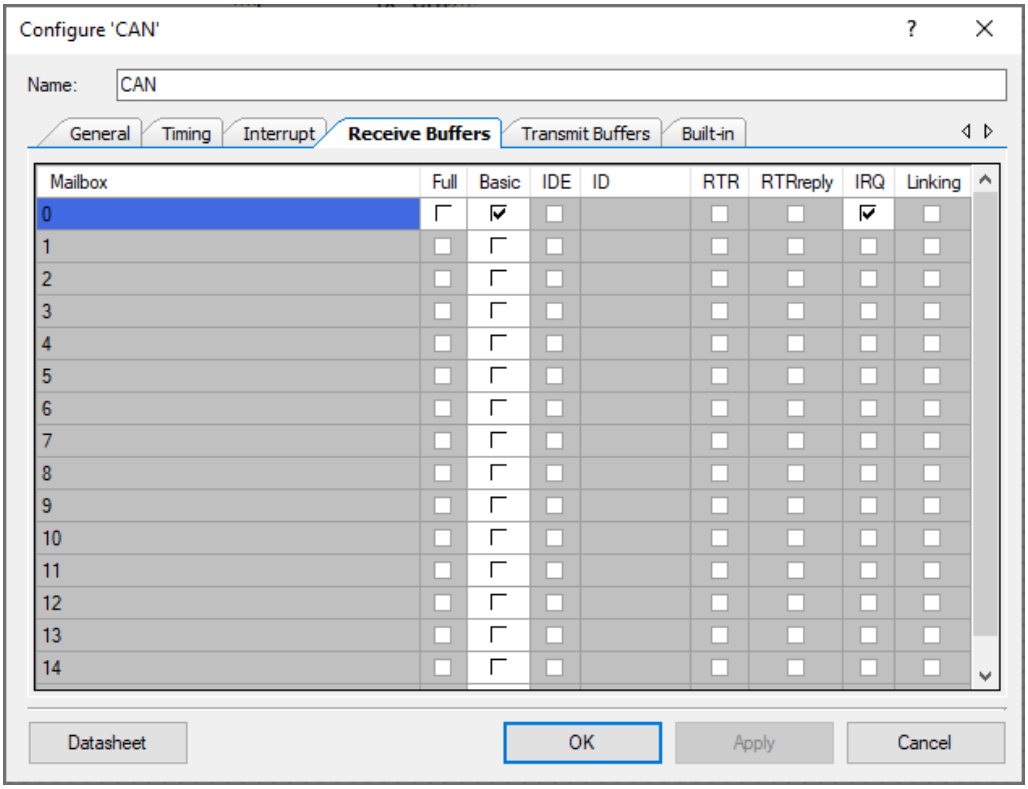
As we want to understand how the interface works, we will start with Basic CAN.

Add a CAN component to your project and set the parameters as follows. This will set up a basic CAN node running at 1Mbit.


Configure 'CAN'
Name:
CAN
General
Timing
Interrupt
Receive Buffers
Transmit Buffers
Built-in
Add transceiver enable signal
Transmit buffer arbitration:
Round Robin
Bus-off restart:
Manual
CAN bus synchronization logic:
'R' to 'D'
Data byte endianness:
Big endian
Data register high
D0 [63:56]
D1 [55:48]
D2 [47:40]
D3 [39:32]
Data register low
D4 [31:24]
D5 [23:16]
D6 [15:8]
D7 [7:0]
Datasheet
OK
Apply
Cancel


Configure 'CAN'
Name:
CAN
General
Timing
Interrupt
Receive Buffers
Transmit Buffers
Built-in
Calculator
Clock frequency (kHz):
24000
Desired baud rate (kbps):
1000
Sample mode:
1-Sample
Settings
BRP:
1
Tseg1:
9
Tseg2:
2
SJW:
2
BRP | Time Quantum | Tseg1 | Tseg2 | SJW | Sample Point | Variance | Propagation Delay (ns)
1 | 12 | 9 | 2 | 2 | 83.3 | 0.25 | 580
2 | 8 | 5 | 2 | 2 | 75 | 0.25 | 380
1 | 12 | 8 | 3 | 3 | 75 | 0.25 | 420
0 | 24 | 16 | 7 | 4 | 70.8 | 0.25 | 380
1 | 12 | 7 | 4 | 4 | 66.7 | 0.25 | 250
Datasheet
OK
Apply
Cancel


Configure 'CAN'
Name:
CAN
General
Timing
Interrupt
Receive Buffers
Transmit Buffers
Built-in
Advanced interrupt configuration
Enable external interrupt line
Full custom internal ISR
Disable internal ISR
Enable interrupts
ISR helper function call
Message transmitted
Message received
Receive buffer full
Bus off state
CRC error detected
Message format error detected
Message acknowledge error detected
Bit stuffing error detected
Bit error detected
Overload frame received
Arbitration lost detected
Single shot transmission failure
Stuck at zero
Enable
Disable
Datasheet
OK
Apply
Cancel


Configure 'CAN'
Name:
CAN
General
Timing
Interrupt
Receive Buffers
Transmit Buffers
Built-in
Mailbox
Full
Basic
IDE
ID
RTR
RTRreply
IRQ
Linking
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Datasheet
OK
Apply
Cancel

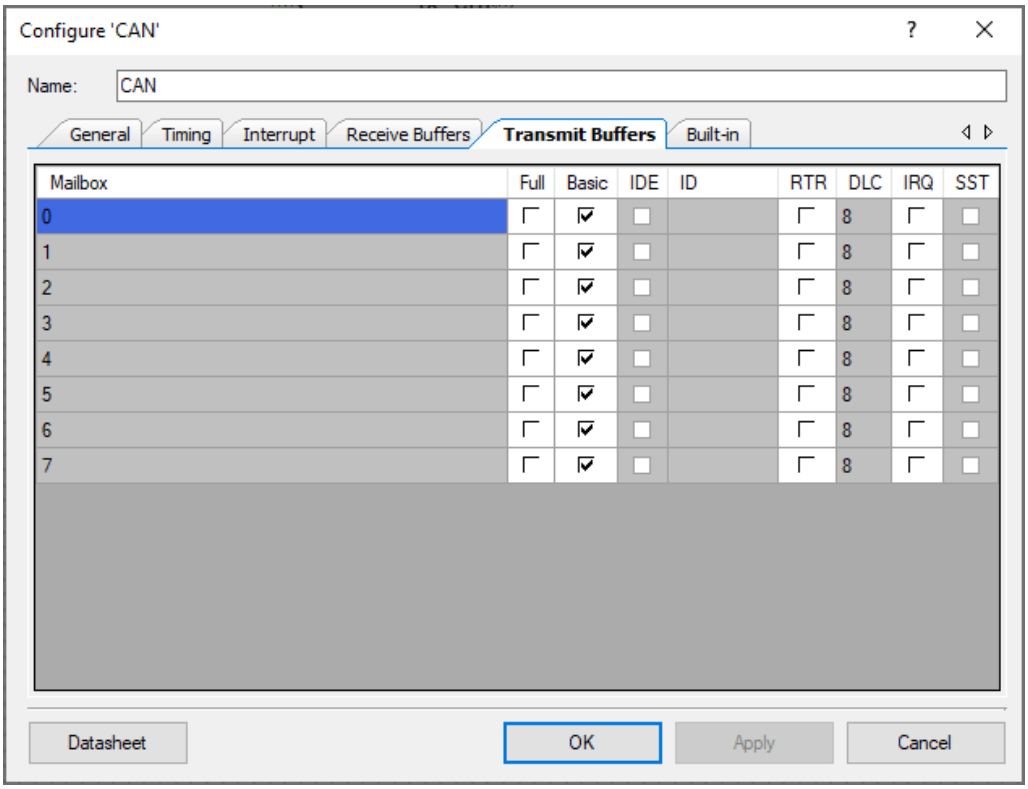


Figure 82 – Sample Configuration for 1Mbit Basic CAN

Now, let's create a basic cyclic task and let's transmit some data over the CAN.

For this, we have to analyze 2 important data structures:

CAN_DATA_BYTES_MSG	This is a simple uint8_t array containing 8 elements, which are representing the payload of a CAN message.	
CAN_TX_MSG	This is a CAN object describing a complete CAN frame:	
	id	The communication Id, identifying the message. The COB Id also defines the priority and can be used for acceptance filtering on the receiver side.
	ide	0 sets the identifier length to 11 bit (default), 1 allows for 29bit addressing.
	rtr	0 identifies a normal message, 1 creates a request frame (without payload)
	dlc	The number of bytes to be transmitted (1..8)
	msg	Pointer to the payload array

The following code will send a CAN message of the size 1 to the bus:

```
TASK(tsk_cyclic)
{
    static uint8_t count = 0;
    LED_Toggle(LED_RED);

    count++;

    //Setup payload
    CAN_DATA_BYTES_MSG payload = {0,0,0,0,0,0,0,0};
    payload.byte[0] = count;

    //Set up message
    CAN_TX_MSG msg;
    msg.id = 0x100;     //CobId
    msg.ide = 0;        //11bit identifier (else 29)
    msg.rtr = 0;        //Normal message
    msg.dlc = 1;        //Length
    msg.msg = &payload; //Data to be transmitted

    CAN_SendMsg(&msg);

    TerminateTask();
}
```

7.2.4 Iteration 2- Receiving protocols using Basic CAN

The reception of communication protocols as well as most other more complex protocols like SPI or I2C are using the following pattern:

- A generated ISR handler will check for incoming data (or error, transmissions etc., depending on what has been configured
- In case of available data, a mailbox specific handler is called which contains a user code block to add the wanted behavior, very often by copying the incoming data into an application buffer and signaling the reception to the application.

The interrupt function can be found in CAN_INT.c. The ISR acts as a dispatcher, calling the required handler based on the mailbox configuration.

```
404     void CAN_MsgRXIsr(void)
405     {
406         uint8 mailboxNumber;
407         uint16 shift = 0x01u;
408 
409         /* Clear Receive Message flag */
410         CAN_INT_SR_REG.byte[1u] = CAN_RX_MESSAGE_MASK;
411 
412         /* `#START MESSAGE_RECEIVE_ISR` */
413 
414         /* `#END` */
415 
416         #ifdef CAN_MSG_RX_ISR_CALLBACK
417             CAN_MsgRXIsr_Callback();
418         #endif /* CAN_MSG_RX_ISR_CALLBACK */
419 
420         for (mailboxNumber = 0u; mailboxNumber < CAN_NUMBER_OF_RX_MAILBOXES; mailboxNumber++)
421         {
422             if ((CY_GET_REG16((reg16 *) &CAN_BUF_SR_REG.byte[0u]) & shift) != 0u)
423             {
424                 if ((CAN_RX[mailboxNumber].rxcmd.byte[0u] & CAN_RX_INT_ENABLE_MASK) != 0u)
425                 {
426                     if ((CAN_RX_MAILBOX_TYPE & shift) != 0u)
427                     {
428                         /* RX Full mailboxes handler */
429                         switch(mailboxNumber)
430                         {
431                             case 0u : CAN_ReceiveMsg0();
432                             break;
433                             default:
434                             break;
435                         }
436                     }
437                     else
438                     {
439                         /* RX Basic mailbox handler */
440                         CAN_ReceiveMsg(mailboxNumber);
441                     }
442                 }
443             }
444             shift <<= 1u;
445         }
446     }
447 #endif /* CAN_RX_MESSAGE */
448 
```

Figure 83 – CAN ISR and dispatcher

The called handlers can be found in the file CAN_TX_RX_func.c. In a first test, send a CAN protocol to the bus (e.g. by connecting a second PSOC) and check if the handler is called by setting a breakpoint in the function.

```
603 /*******************************************************************************
604 * FUNCTION NAME:   CAN_ReceiveMsg
605 ********************************************************************************
606 *
607 * Summary:
608 *  This function is the entry point to Receive Message Interrupt for Basic
609 *  mailboxes. Clears the Receive particular Message interrupt flag. Generated
610 *  only if one of the Receive mailboxes is designed as Basic.
611 *
612 * Parameters:
613 *  rxMailbox: The mailbox number that trig Receive Message Interrupt.
614 *
615 * Return:
616 *  None.
617 *
618 * Reentrant:
619 *  Depends on the Customer code.
620 *
621 *******************************************************************************/
622 void CAN_ReceiveMsg(uint8 rxMailbox)
623 {
624     #if (CY_PSOC3 || CY_PSOC5)
625         if ((CAN_RX[rxMailbox].rxcmd.byte[0u] & CAN_RX_ACK_MSG) != 0u)
626     #else  /* CY_PSOC4 */
627         if ((CAN_RX_CMD_REG(rxMailbox) & CAN_RX_ACK_MSG) != 0u)
628     #endif /* CY_PSOC3 || CY_PSOC5 */
629         {
630             /* `#START MESSAGE_BASIC_RECEIVED` */
631
632             /* `#END` */
633
634             #ifdef CAN_RECEIVE_MSG_CALLBACK
635                 CAN_ReceiveMsg_Callback();
636             #endif /* CAN_RECEIVE_MSG_CALLBACK */
637
638             #if (CY_PSOC3 || CY_PSOC5)
639                 CAN_RX[rxMailbox].rxcmd.byte[0u] |= CAN_RX_ACK_MSG;
640             #else  /* CY_PSOC4 */
641                 CAN_RX_CMD_REG(rxMailbox) |= CAN_RX_ACK_MSG;
642             #endif /* CY_PSOC3 || CY_PSOC5 */
643         }
644 }
```

Figure 84 – CAN Transmission Handler

If the interrupt is not hit, check the configuration of the CAN. The first mailbox must have the IRQ flag enabled!

In the next step, we implement a callback function which is used in the handler. This callback function will get the rxmailbox identifier as parameter. For this, add the following lines in cyapicallbacks.h

```
#define CAN_RECEIVE_MSG_CALLBACK
#define CAN_ReceiveMsg_Callback() can_rx_basic_cb(rxMailbox)
```

And add a function declaration into CAN_TX_RX_func.c and don't forget to include cyapicallbacks.h

```
/* `#START TX_RX_FUNCTION` */

#include "cyapicallbacks.h"

extern void can_rx_basic_cb(uint8_t mailbox);
/* `#END` */
```

In order to access the data, the PSOC firmware provides a couple of macros[13]

[13] The naming depends on the name of the hardware peripheral. Furthermore, the parameter names have been modified to improve readability. For the project specific names, please check can.h

Macro	Function
CAN_GET_RX_ID(mailbox)	Get the CobId of the message
CAN_GET_RX_IDE(mailbox)	Size of the identifier
CAN_GET_DLC(mailbox)	Length of the message
CAN_RX_DATA_BYTE(mailbox, i)	General API to access a byte of the RX buffer
CAN_RX_DATA_BYTE1(mailbox) … CAN_RX_DATA_BYTE8(mailbox)	Wrapper for CAN_RX_DATA_BYTE(mailbox, i)

Using these macros, we can implement a first BASIC CAN reception function in main.c, sending the content of the received data to the UART.

```
void can_rx_basic_cb(uint8_t mailbox)
{
    CAN_DATA_BYTES_MSG payload = {0,0,0,0,0,0,0,0};

    //Get the metadata of the message
    uint16_t cobid = CAN_GET_RX_ID(mailbox);
    uint8_t len = CAN_GET_DLC(mailbox);

    //Read in the payload
    for (uint8_t i = 0; i < len; i++)
    {
        payload.byte[i] = CAN_RX_DATA_BYTE(mailbox, i);
    }

    LOG_I("CAN RX","%x [%d] %x %x %x %x %x %x %x %x", cobid, len,
payload.byte[0], payload.byte[1], payload.byte[2], payload.byte[3], pay-
load.byte[4], payload.byte[5], payload.byte[6], payload.byte[7]);

}
```

7.2.5 Iteration 3 – Full CAN

The transmission and reception of data using FULL CAN mailboxes requires both for the transmission and the reception own handlers. The transmission handler is also required in case no TX interrupt is set.

The recipe is similar to BASIC CAN, we have to provide two handler functions and activate them in cyapicallbacks.h

```
    //RX FULL CAN Mailbox 0
    #define CAN_RECEIVE_MSG_0_CALLBACK
    #define CAN_ReceiveMsg_0_Callback() can_rx_full_0_cb()

    //TX FULL CAN Mailbox 0
    #define CAN_SEND_MSG_0_CALLBACK
    #define CAN_SendMsg_0_Callback() can_tx_full_0_cb()
```

The reception handler will be called by the ISR. The transmission handler will be called by the function `uint8 CAN_SendMsg0(void)`.

Internally, we can use the macros we have already encountered for the BASIC CAN, but as the configuration including CobId is done in the GUI, the API is a bit simpler.

On the transmission side, we configure a message having a CobId 0x001 and a length of 2 bytes. Furthermore we will add a reception mailbox for CobId 0x001 [14]

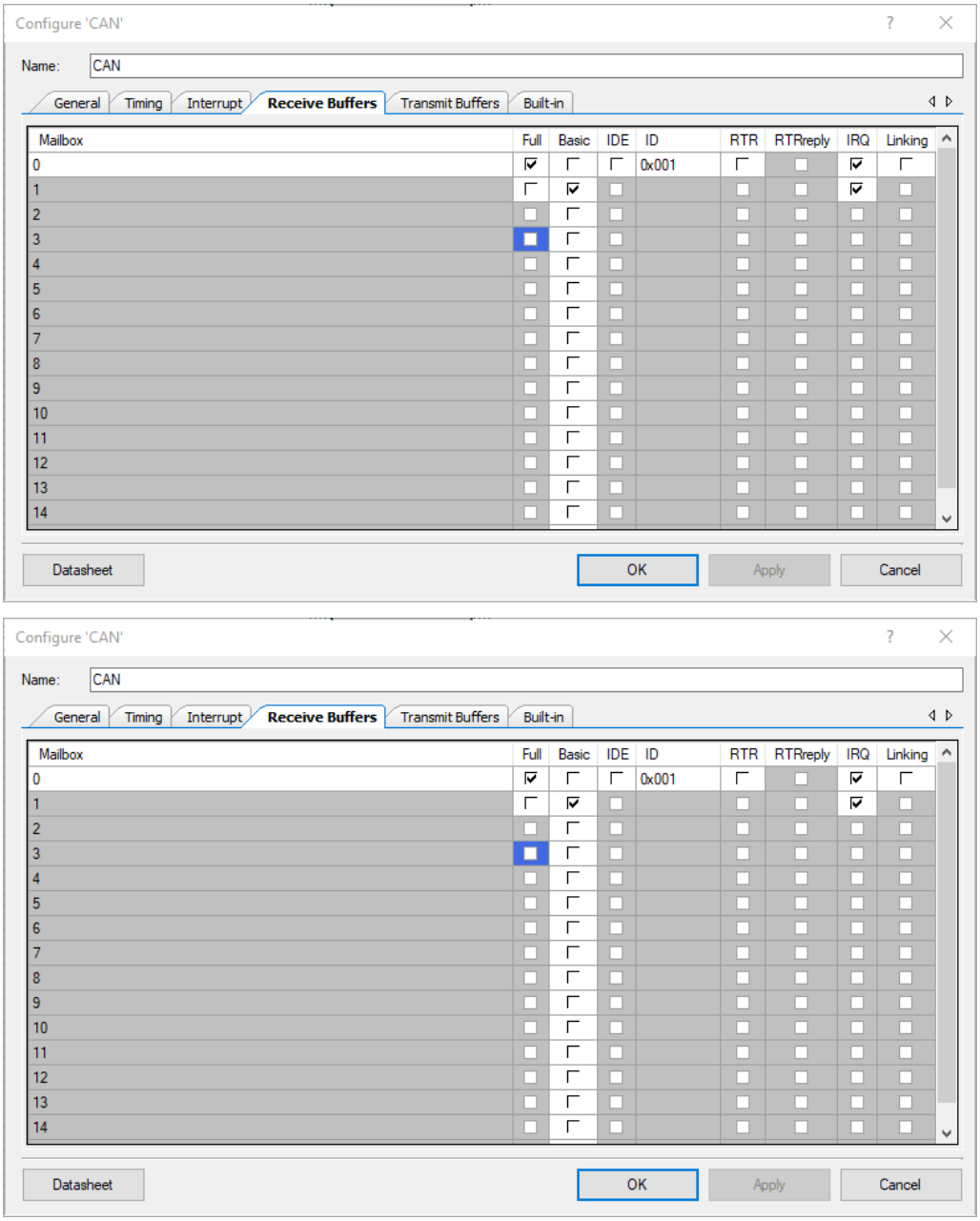


Figure 85 – Full CAN configuration

The code for the handlers looks as follows:

[14] Note, that FULL CAN mailboxes need to be configured before BASIC CAN mailboxes, at least on RX side to make sure, that specific messages are handled by the FULL CAN configuration.

```
/**
 * The FULL CAN Handler on mailbox 0
 * Handles messages with CobId 0x01
 */
extern void can_rx_full_0_cb()
{
    //We have a fixed mailbox number and handler in this case
    const uint8_t mailbox = 0;

    CAN_DATA_BYTES_MSG payload = {0,0,0,0,0,0,0,0};

    //Get the metadata of the message
    uint16_t cobid = CAN_GET_RX_ID(mailbox);
    uint8_t len = CAN_GET_DLC(mailbox);

    //Read in the payload
    for (uint8_t i = 0; i < len; i++)
    {
        payload.byte[i] = CAN_RX_DATA_BYTE(mailbox, i);
    }

    LOG_I("CAN RX FULL","%x [%d] %x %x %x %x %x %x %x %x", cobid, len, pay-
load.byte[0], payload.byte[1], payload.byte[2], payload.byte[3], payload.byte[4],
payload.byte[5], payload.byte[6], payload.byte[7]);

}


/**
 * The TX FULL CAN Handler on mailbox 0
 */
extern void can_tx_full_0_cb()
{
    static uint16_t count = 0;
    count++;
    CAN_TX_DATA_BYTE1(0) = count / 0x100;
    CAN_TX_DATA_BYTE2(0) = count % 0x100;
}
```

For sending the data cyclically, we only have to call the mailbox 0 TX function:

```
    // Full CAN
    CAN_SendMsg0();
```

7.2.6 Iteration 4 – Let's get the application up and running

Please note, that the code inside the handlers only serves as an example. In order to synchronise handlers with your application, global variables, RTE signals or OS messages are required.

For the transmission of the joystick data, we use the following conventions:

- Button 1 – Bit 0 of Byte 1
- Button 2 – Bit 1 of Byte 1
- Button 3 – Bit 2 of Byte 1
- Button 4 – Bit 3 of Byte 1
- Joystick x – Byte 2
- Joystick y – Byte 3

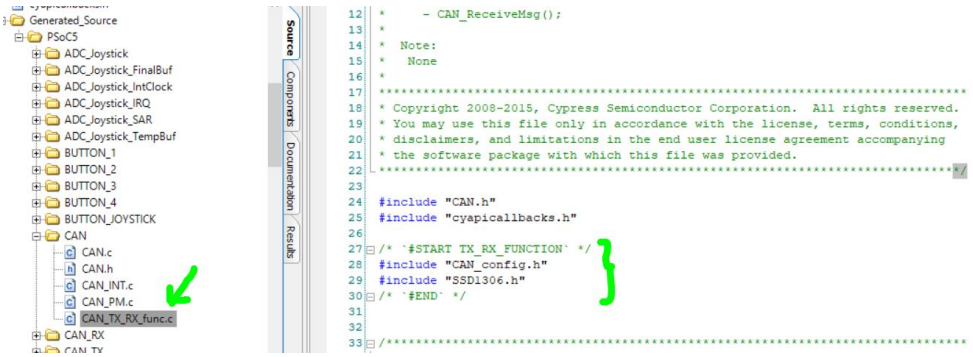
The CobId will be 0x100;

For the display of the data, you may design a simple UI on the TFT, e.g. moving a cross deepening on the joystick position and e.g. changing the colour depending on the button state.

7.3 CanOpen Communication Stack

7.3.1 Getting the CAN component up and running

7.3.2 CanOpen



8 Real Fun

Combining all the different patterns we have seen so far, we now will start with the implementation of some real fun applications.

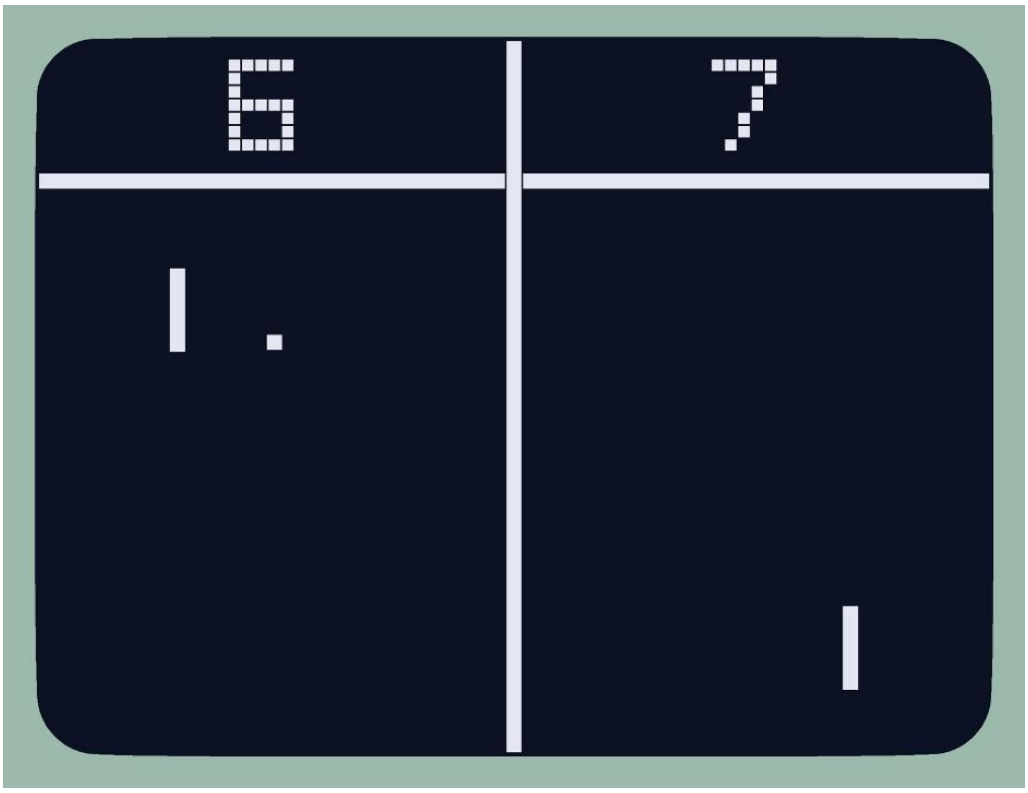


Figure 86 – Game of Pong / Soccer

8.1 Morse coder/decoder

Effort: 8h	Category – B
Signal processing, RTE Architecture, Algorithms	

You work on the next generation communication system for the German Navy. The communication system of the Navy is based on the well-known Morse Code, as shown in the picture below.

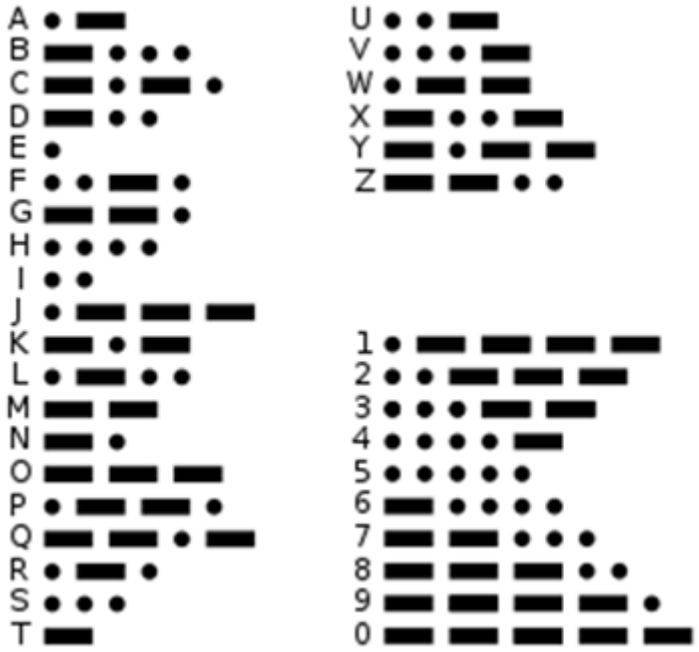


Figure 87 – Morse Code Alphabet

Your job is to implement a software, which allows to write and read Morse code on a PSOC microcontroller by reading from / writing to a GPIO pin.

Example: The famous sequence “SOS / ••• --- •••” would look like this:

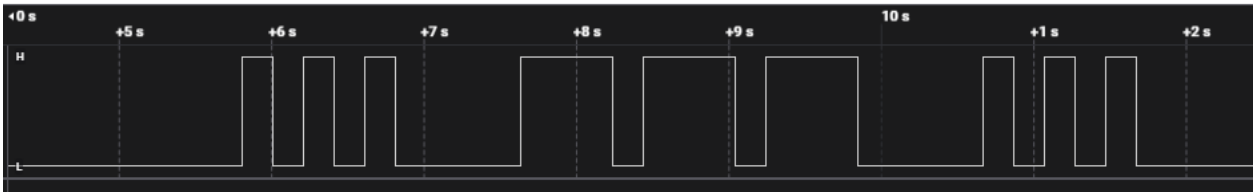


Figure 88 – Morse Code Sequence SOS

8.1.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS and the RTElight.

NFR2	Use existing functions of the system as far a sensible and possible.
NFR3	Put the developed code in sensible files and functions. Avoid global data unless absolutely required – e.g. for interfacing the handlers!
FR1	The user can provide data via the UART port, which then is translated into a Morse sequence
FR2	Alternatively, the Morse sequence can be created manually by pressing one of the buttons
FR3	The Morse code will be fed to the green LED for visualization and also to the UART2 Tx pin at the side of the PSOC
FR4	Every letter / number is represented by a sequence of short dots and long dashes • The length of a dot is one unit, the length of a dash is three units • The space between dots and dashes of the same letter is one unit • The space between letters of a word is 3 units • The space between words is seven units • Every new transmission of a sequence will start with a space (letter or word, depending on the first token) • The length of a unit is configurable (#define), we will start with 100ms
FR5	The Morse code will be read in via the Rx pin of UART 2 (e.g. when using a second board for sending). Furthermore, a loopback will be provided for the transmitted signal for testing purposes.
FR6	The red LED will show the incoming signal.

8.1.2 Hardware Architecture

In a first step, we want to design the hardware architecture. Of course, you can and probably should start with a skinny sheep, which only requires 2 GPIO pins, one for input and one for output, let's have a look at the full blown solution we would like to have at the end.

Some hints:

- Use Control and Status Registers to create the software interface
- Use OR gates to create alternative inputs
- Use #defines to abstract the hardware interface (supports the skinny sheep)
- Use an AND gate in the loopback line to disable it if needed.
- The Morse input pin should be configured as pulldown resistive to reduce disturbancies at the OR gate.
- Add diagnostic pins as required.

First version with direct pins:

```
#define MORSE_WRITE_PIN(a) LED_green_Write(a)
#define MORSE_READ_PIN() Pin_Morse_Read()
```

Later version using status and control register:

```
#define MORSE_WRITE_PIN(a) Morse_Tx_Reg_Write(a)
#define MORSE_READ_PIN() Morse_Rx_Reg_Read()
```

8.1.3 Software Architecture and Signal Flow

The application will look as follows:

- The user provides data via a serial port to the PSOC
- Whenever a complete string (identified by a <CR> character) is received, the signal `so_string_tx_signal` will be passed from the `isr_uartrx` to the runnable `MORSE_tx_run`.
- The `MORSE_tx` _run will add the data to the transmission ringbuffer
- `MORSE_tx_run` will (also) be called cyclically (10ms). Upon a cyclic event, it will be checked if the ringbuffer contains some data and the transmission state machine (see below) will be invoked.
- Another cyclic runnable `MORSE_rx_run` will be called every 10ms and checks the reception pin of our application. After the reception of a word, the morse code will be send as a string of dots, dashes and spaces (T for token, W for word spaces) to the `MORSE_decode_run` using the signal `so_morse_rx_signal` (Example string for the SOS sequence: "…T---T…").
- Upon reception of the string, the runnable will decode the word and send the ASCII string (`so_string_rx_signal`) `to HMI_display_run`, which will show the content on the TFT display.
- The functions for the TFT will be executed in an own low priority task.

Provide a signal flow diagram, showing the isr's, runnables, data signals and trigger events stated above.

Use the RTE Light to create the code framework for the RTE.

8.1.4 State Machine Sending Morse Code

In the next step, we will focus on the sending logic.

First, create the implementation for the string signal. The string signal contains a string of a fixed max size (e.g. 100 chards) and a length parameter.

The ISR receives the bytes from the UART and uses the set method of the string signal to transfer the data to the runnable `MORSE_tx_run`.

Hints:

- You may use a static local string object as intermediate storage of the data.
- Providing methods for the string datatype to clear the string and to add a single char to the string will make life easier 😉. Use the user code sections in the signal file for this.
- The logic of the runnable will differ depending in the event which fired the runnable:
    - On_data: add the string to the transmission ringbuffer

- Cyclic: execute the transmission state machine

A lookup table can be used to translate an ASCII token into the Morse code dot/dash sequence.

```
const static MORSE_token_t MORSE__table[] =
{
    {'a', ".-"},
    {'b', "-..."},
    {'c', "-.-."},
    {'d', "-.."},
…
    {'4', "....-"},
    {'5', "....."},
    {'6', "-...."},
    {'7', "--..."},
    {'8', "---.."},
    {'9', "----."},
    {'0', "-----"}
};
const static uint16_t MORSE__table_SIZE =
sizeof(MORSE__table)/sizeof(MORSE_token_t);
```

The data structure of a single entry of the table is as follows:

```
struct sMorseToken {
  char token;                   /**< ASCII Value */
  const char *const sequence;   /**< Morse Sequence, consisting of . and - */
};
typedef struct sMorseToken MORSE_token_t;
```

We are using a state machine which will be called cyclically to implement the logic. Concerning the state machine implementation, please consider the following hints:

- The state machine will be called cyclically, i.e. the (virtual) event for all transitions will be the 10ms tick event. As this event is not fired explicitly, you can either add it in a smart way to the system or you implement the transitions only by using guards.
- Many transitions will be time triggered. Provide a timer function containing an action to set the timer and a guard function checking if the timeout of the timer has been reached.
- You will need around 3 states and 9 transitions.
- Provide own static functions for all guards and actions.
- Create a MORSE object storing the required static data of the Morse coder (like state, ring-buffers,…)

Draw the state diagram.

Implement and test the sending logic.

8.1.5 Reception algorithm

We will now move to the reception of the Morse code; the focus will be the runnable `MORSE_rx_run`. The job of this runnable is to cyclically poll an input pin and to translate the physical signal into a string sequence containing dots, dashes and spaces (T for token spaces, W for word spaces, E for end

space). Example string for the SOS sequence including token spaces: "…T---T…E". You receive the following additional requirements:

- One time unit (i.e. a dot) will be 100ms, a dash and a token space will have the duration of 300ms and a space between 2 words will be 700ms.
- We will allow a tolerance of +/- 20%, i.e. a permanent high level between 80ms and 120ms will be considered as a dot, a permanent low level between 240ms and 360ms will be a token space, etc.
- The runnable will be called cyclically every 10ms
- The received dots / dashes and spaces will be stored in the corresponding string object and send to the decoder runnable if no more data is received for 1000ms (indicated by letter E).

The code shall not be placed in the runnable directly, but instead a static function shall be created. Provide an activity diagram to describe the algorithm of this function.

Test the algorithm by printing the decoded dots and dashes on the UART.

Note: The algorithm with the provided tolerances will work fine for the Morse sequences generated by the PSOC. How about human sequences? What needs to be done to decode human sequences in a reliable way?

8.1.6 Decoding Algorithm

The last piece of work is the development of the decoding algorithm. For this, we can use the lookup table again.

- Inside the decode runnable, cut the string with the Morse sequence into substrings, using the space identifiers as deliminators
- Then write a function which takes a substring and returns the corresponding ASCII value from the lookup table

Develop the decoder function. Add verbosity as needed. Once we have the ASCII string, add it to the last signal object and send it to the TFT for display.

8.1.7 Test

Connect the output pit to the input pin (either by wire or by adding a connection in the top design).

Happy morsing 😉

8.2 The game of PONG

Effort: 8h	Category – B
Signal flow, RTE Architecture, Algorithms, OO design, Active Objects	

We want to develop of couple of 80ies Arcadian games, which can be played on the labboard. The first one will be the game of pong. In this arcadian, we have to keep a ball in the game by moving a bat. The ball is reflected by the walls. To make things harder in later rounds, the bat will become smaller and the ball will move faster.

Another important aspect will be: After implementing the other games, we want to be able to select the game in the way of an arcadian hall. I.e. we have a set of games and control unit input (buttons and joystick) will either be used to select or to play a game. Although this will not be required in the first game, we should prepare the data structures and interfaces of the game to allow a simple extendibility.

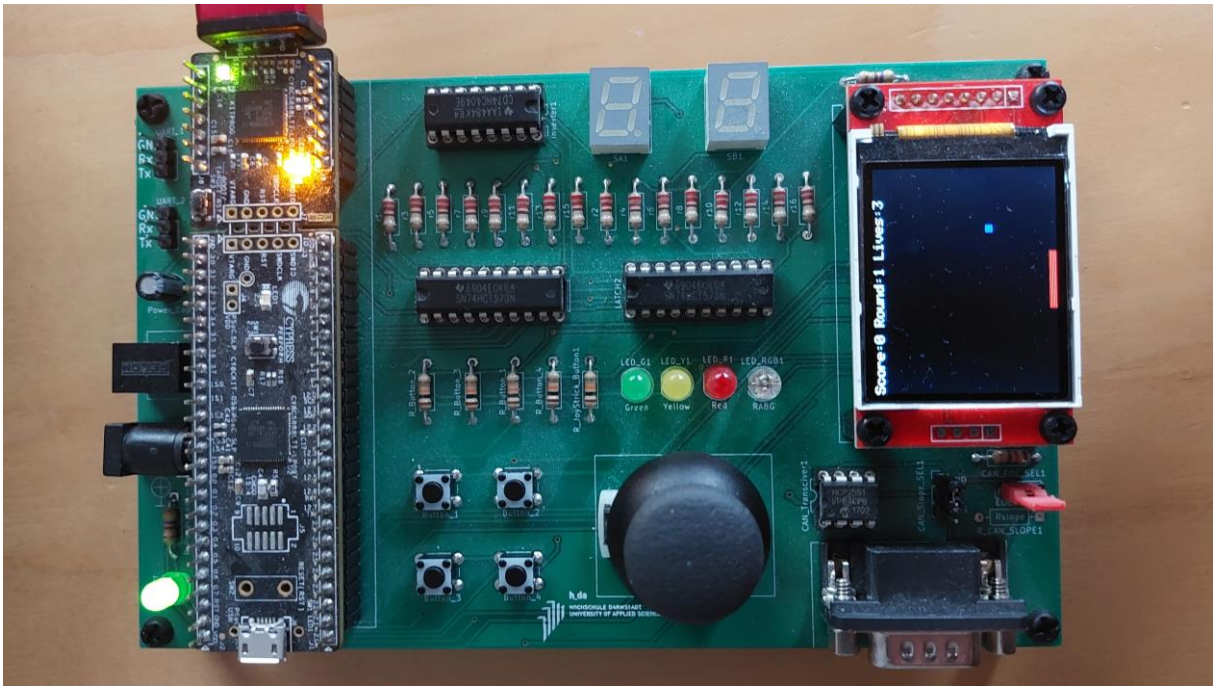


Figure 89 – Game of Pong

8.2.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS and the RTElight.
NFR2	Use existing functions of the system as far a sensible and possible. For the game objects, it is recommended to use the OO pattern for C.
NFR3	Put the developed code in sensible files and functions. Avoid global data unless absolutely required!
FR1	The user can play the game of Pong.

FR2	Upon start, a welcome screen will show a welcome message.
FR3	After pressing button 3, the game will start. Upon start, the player has 3 lives.
FR4	The game consists of up to 255 rounds. Every round ends after 10 successful bounds. For every new round, the size of the bat will decrease and the speed of the ball will increase. Furthermore, more random effects may be added to the reflection of the ball and the movement of the bat, to make the game harder. A life will be added after every successful round. The maximum number of lives is 5.
FR5	In case the player misses the ball, a life will be lost. In case the player has no more lives left, the game is over.
FR6	For every successful bounce by the bat, a point is added to the score.
FR7	The round, number of life's and score is shown on the top of the game
NFR8	The game will be played with a 90° rotated screen, i.e. the left (or right – whatever you prefer) side of the board will show down to have more movement for the bat.
FR9	The ball will reflect from the bat and the walls. The reflection angle at the bat will also be affected from the area where it is reflected. A reflection at the right side will create a stronger speed coordinate to the right as compared to a reflection in the middle. Same applies for left.
FR10	In order to support the idea of active objects, the button press events will have the following effects on the game • Button 1 – pause and resume • Button 2 – terminate • Button 3 – double the size of the bat for the current round (this may be used 10 times only) • Button 4 – slow down the ball speed for 10s (or until the next bounce at the bat)
NFR10	The ball and all other game objects shall move in a fluent way without flickering
NFR11	As we will integrate several games later, we will put all pong files into a subfolder pong in the asw source area.

8.2.2 RTE Design

In a first step, create a signal flow diagram. Consider, that the input commands in later editions of the arcadian game may also come from different sources, e.g. instead of getting signals only from the button or joystick on your board, we may also have the situation that two boards are interconnected, allowing two players to compete. E.g. the CAN or UART interface may also serve as input sources, which should be easily added later. Therefore, the game logic and input should be separated.

Another use case which must be kept in mind: we will select and play different games later, i.e. you should think about common game interfaces and decide, if a game is a runnable or a (alternative) function being called inside a runnable and how the different signals can be routed to different games. A possible design pattern is to consider a single game as an active object.

Another aspect we should consider when designing the game, is the rather long runtime of the drawing routine. It could make sense to set the objects in a fast control task and to redraw them in a low priority HMI task. This is a design requirement, we cannot decide at this point of time. We will start with a single task for calculating the new position and redrawing the objects. In case we face real-time issues, we will add another low priority HMI task, which will be updated with a slower rate.

Developing such a system is pretty complex. It is recommended to work in iterations and to verify the extendibility of your design after every step. In case you feel that the design becomes hacky, you should go for a refactoring loop.

8.2.3 PONG Game Design

Once we have decided on the overall structure of the game, create a class diagram describing the PONG game structure. Use the model view control pattern as a basis for the design.

8.2.4 Moving a graphical object on the TFT screen

Moving a graphical object on the screen does not sound that complex. Let's simply clear the screen and redraw all objects with the new position in a cyclic way. Although this approach will work, the view experience will be pretty bad – the objects will flicker like hell and the update is rather low. The reason is the large amount of data which must be punped over the SPI for a complete redraw of the screen.

A better approach will be to only clear the object which is moving. But also this approach might result in a flickery experience, depending on the object we are moving.

The optimal solution will be to calculate which pixels of an object must be redrawn and to clear/set only the changed pixels. Develop an algorithm for this movement concept considering the bat in a first step. Which parameters will influence the changed pixels – how can you calculate which pixels need to be changed.

8.2.5 Moving the bat

We will start with moving the bat. The logic is as follows

- Depending of the position of the joystick, the speed of the bat will be determined
- The bat will move according to the speed value, i.e. a good idea would be to determine the speed as pixels per round. It is highly recommended to add such design decisions as comments to the corresponding data types.
- When redrawing the bat, we need to clear the area of the tail and fill the area at the front of the movement.
- Furthermore, we need to check and synchronise the orientation of the screen and joystick coordinate system.

Implement the bat and test the correct functionality

8.2.6 Moving the ball

Moving the ball follows the same pattern like the bat. The only difference is, that we have a movement in x- and y-direction and that we need to implement the reflection at the wall and the bat. At least, if the bat is at the right position.

The easiest way is to describe the movement of the ball with a speed vector, i.e. a speed in x direction and a speed in y direction. When reflecting the ball at the screen boundaries and bat, we just need to flip the sign of the corresponding part of the speed vector.

Another aspect we must consider, is the dependency of the ball movement from the screen, e.g. screen dimensions, bat height, area for the score etc. It might be a good idea to a screen_cfg.h file containing all these global configurations.

And last but not least you must check for successful and missed bounces and return this information to the game for further actions related to the score, round and number of lives.

8.2.7 Adding the score, round and lives

Now we just need to add a headline to the screen, showing the score, round and number of lives. As a first step, simply restart the game in case all lives have been lost.

8.2.8 Active Objects

The following code shows a typical implementation of the game so far. The runnable GAME_play_run receives the signals of the joystick and the buttons and passes them to the game function PONG_play. Later on, we will have several games at this position, which can be played alternatively.

```
void GAME_play_run(RTE_event ev){

       /* USER CODE START GAME_play_run */

    //Get button and joystick state

    SC_BUTTON_data_t btn = RTE_SC_BUTTON_get(&SO_BUTTON_signal);
    SC_JOYSTICK_data_t joy = RTE_SC_JOYSTICK_get(&SO_JOYSTICK_signal);

    PONG_play(&btn, &joy);

    /* USER CODE END GAME_play_run */
}
```

The PONG_play function may look as follows. Error handling statements have been deleted for the sake of readability. Furthermore, the button events are not processed yet.

Please note the switch case statement, which is showing a first code rot pattern. Why? Actually, the reaction on the bat bounce (hit or miss) as well as the button events are part of a state machine, which is not explicitly designed and coded yet. Instead, the logic is hidden in a series of switch-case and if statements.

```
/**
 * Main game routing for pong, must be called in a cyclic way
 * \param SC_BUTTON_data_t const& btn   : [IN] The four button states
 * \param SC_JOYSTICK_data_t const& joy : [IN] The joystick state
 */
RC_t PONG_play(SC_BUTTON_data_t const *const btn, SC_JOYSTICK_data_t const *const
joy)
{

    RC_t res = RC_SUCCESS;


    //Check for joystick position and update bat
    //As we rotate the board by 90°, we use the y position of the joystick

    //Provides the ADC value in the range -128 .. 127
    sint8_t speed = joy->m_pos.m_y;

    //Scale to sensible value, max approx 10
    speed = (speed * 10) / 128;

    //Call the move routine for the bat
    res = BAT_move(&PONG.m_bat, speed, PONG.m_screen.m_width);

    //Call the move routine for the ball
    res = BALL_move(&PONG.m_ball, &PONG.m_bat, PONG.m_screen.m_width,
PONG.m_screen.m_height);

    BALL_batBounce_t ballState = BALL_getBounceResult(&PONG.m_ball);
    switch (ballState)
    {
        case BALL_BATBOUNCE:
            PONG_bounce();
            SCREEN_updateScore(&PONG.m_screen,PONG.m_lives, PONG.m_score,
PONG.m_round);
            break;
        case BALL_BATMISS:
            PONG_missesBounce();
            SCREEN_updateScore(&PONG.m_screen,PONG.m_lives, PONG.m_score,
PONG.m_round);
            break;
        case BALL_NORMAL: /* Do Nothing */ break;
        default:
            break;
    }

    //Call the screen redraw method
    res = SCREEN_redraw(&PONG.m_screen);
    if (RC_SUCCESS != res)
    {
        LOG_E("PONG", "Error SCREEN_redraw %d", res);
    }

    return RC_SUCCESS;
}
```

```
/**
 * Action when ball bounces
 */
static RC_t PONG_bounce()
{
    PONG.m_score++;

    if (PONG.m_score % PONG_BOUNCE_PER_ROUND == 0)
    {
        if (PONG.m_lives < PONG_MAXLIVES)
        {
            PONG.m_lives++;
        }
        PONG.m_round++;

        PONG.m_ball.m_maxspeed++;

        if (PONG.m_bat.m_size > PONG_BAT_MINSIZE+1)
        {
            PONG.m_bat.m_size -=2;
        }

        PONG.m_ball.m_pos.m_x = PONG.m_screen.m_width/2;
        PONG.m_ball.m_pos.m_y = PONG.m_screen.m_height/2;

        PONG.m_ball.m_speed.m_x = PONG.m_ball.m_maxspeed;
        PONG.m_ball.m_speed.m_y = -PONG.m_ball.m_maxspeed;

    }

    return RC_SUCCESS;
}

/**
 * Action when ball misses a bounce
 */
static RC_t PONG_missesBounce()
{
    if (PONG.m_lives > 0)
    {
        PONG.m_lives--;

        PONG.m_ball.m_pos.m_x = PONG.m_screen.m_width/2;
        PONG.m_ball.m_pos.m_y = PONG.m_screen.m_height/2;

        PONG.m_ball.m_speed.m_x = PONG.m_ball.m_maxspeed;
        PONG.m_ball.m_speed.m_y = -PONG.m_ball.m_maxspeed;
    }
    else
    {
        //Game over, let's simply restart at this point of time
        PONG_init();
    }

    return RC_SUCCESS;
}
```

In order to improve the design, the first step is to explicitly draw the statemachine of the PONG game. Start by analyzing, which events we have in this application.

Check the existing code blocks of your code. If you have created a good design, it will be pretty straight forward to create an active object for the pong game. Finish the implementation and test the game.

8.3 Snake

Effort: 12h	Category – B
Signal flow, RTE Architecture, Algorithms, OO design, Active Objects, Hierarchical State Machines, Architecture and Code Refactoring	

The second game to be implemented will be Snake. In this game, you have to maneuver a snake over the game area. The game will last, until the snake hits itself. To make it a bit more fun, the snake will grow and move faster during the game.

Although the game and the algorithm are different, from an architectural perspective, there are quite some similarities with the game of Pong which can be reused.

8.3.1 Requirements

Req-Id	Description
NFR1	The system will be developed based on Erika OS and the RTElight.
NFR2	Use existing functions of the system as far a sensible and possible. For the game objects, it is recommended to use the OO pattern for C.
NFR3	Put the developed code in sensible files and functions. Avoid global data unless absolutely required!
FR1	The user can play the game of Snake.
FR2	Upon start, a welcome screen will show a welcome message.
FR3	After pressing button 3, the game will start by showing a welcome screen. Another press of button 3 will start the game. Upon start, the player has 3 lives.
FR4	The game consists of up to 255 rounds. Every round ends after 500 successful movements of the snake. For every new round, the size (and optionally speed) of the snake will increase. A life will be added after every successful round. The maximum number of lives is 5.
FR5	In case the snake head hits the tail or the wall, a life will be lost. In case the player has no more lives left, the game is over.
FR6	For every successful change of direction of the snake, a point is added to the score.
FR7	The round, number of life's and score is shown on the top of the game
NFR8	The game will be played with a 90° rotated screen, i.e. the left (or right – whatever you prefer) side of the board will show down to have more movement for the bat.
FR9	The snake will be controlled by the joystick. For every change of direction, the joystick has to start in the initial position. A movement to the right will

	initiate a 90° clockwise rotation of the snake direction. A movement to the left in the counterclockwise direction. Up and down movements will have no effect.
FR10	In order to support the idea of active objects, the button press events will have the following effects on the game • Button 1 – pause and resume • Button 2 – terminate • Button 3 – decrease the size of the snake for the current round (this may be used 10 times only) • Button 4 – slow down the snake speed for 10s
NFR10	The ball and all other game objects shall move in a fluent way without flickering
NFR11	As we will integrate several games later, we will put all snake files into a subfolder snake in the asw source area.
FR12	We will add the game to the already existing PONG application. An additional menu will allow us to select a game.
NFR13	The menu shall be implemented in such a way, that additional games can be added easily.
NFR14	Furthermore, we will use the container/widget pattern for state machines to combine the state machine of the game selection with the state machines of the individual games
FR15	For selecting the games, the buttons press events will have the following impact • Button 1 – select previous game • Button 2 – select next game • Button 3 – start game

8.3.2 State Diagrams

Draw the state diagram of the menu (container) and snake game (widget).

State Diagram menu

State Diagram snake

8.3.3 Development Strategy

We obviously have 2 main requirements:

- Implementing the new game Snake
- Implementing the menu

As the menu as well as the additional game probably will have an impact on the architecture, you should decide which part you want to implement first. A justification could be to check, which part will have a stronger impact on the architecture.

Which part, snake algo or menu are you going to implement first? What impacts on the architecture do you expect?

8.3.4 Refactoring

Before we start implementing the code for Snake based on the existing implementation of Pong, some ideas for refactoring of the game structure we might want to consider while implementing the full application

- Do the signals provide the required data for snake as well (they should). Maybe you need to add some attributes e.g. to the joystick signal in order to detect a movement in addition to the position. Try to keep such changes local and avoid side effects to the old implementation.
- In Pong, you probably designed the screen as a part of the Pong application. Does this still make sense?
- Do we need a common screen class for both games or a screen per game? Or does it make more sense, to have the redraw routines implemented as methods for the bat, ball, snake classes? Every option has its pro's and cons.
- Some states of the pong game might move to the menu state machine, as they are common for both games (e.g. the pause state)
- The original event structure of the buttons might need to be changed / converted into a state-machine compatible format.

8.3.5 Implementation of the menu and overall game logic

Implement the menu and the game as a container state machine using a lookup table implementation pattern.

Note: We have a hidden event, which comes as tick event every time we call the game runnable. This needs to be fed into the state machine as well.

The logic of the menu is as follows:

- Upon startup, the menu will be shown
- Using Button 1 and 2 the user can scroll through the entries
- Pressing button 3 will activate the currently selected game and show the welcome screen of the game
- Pressing button 3 again will start the game
- During the operation of the game, button 3 and 4 will initiate game specific actions (check the game requirements for details)
- Pressing button 1 will pause and resume the game
- Pressing button 2 will terminate the game in any state and go to the game selection state (showing the menu)

8.3.6 Implementation of Snake

Implement the snake movement. Start by designing an appropriate data structure storing the "pixels" of the snake. Note, that the size of the snake grows after every round. I.e. either create a sufficient large array or free/allocate the required memory after every round.

Then implement the movement of the snake, which works as follows:

- The snake can move in the following directions: 0°, 90°, 180° or 270°

- Every time, a joystick left or right movement is detected (movement means – the joystick moves from the center position to a right or left position)

Idea: Use a ringbuffer to store the snake positions. In order to have a fluent movement, only the last element of the snake needs to be cleared and the first needs to be drawn.

In case the snake head hits a wall or the snake itself, a life is lost. When the snake has lost all lives, the game is over.

8.4 Distributed version of PONG: SOCCER

Effort: 12h	Category – C
CAN or UART, Protocol Design, State Machine Synchronization, DMA	

The idea of this project would be to combine 2 games of pong into a new game soccer for 2 players. For this, 2 boards need to be interfaced by using UART or CAN. Instead of having the ball bounce from the upper frame of the display like in Pong, the ball will traverse to the other board and be shot back by the other player. In case player B misses the ball, player A will increase the score.

The sides of the screen will reflect the ball like in Pong. Every player will control his bat by using his joystick.

The key idea is shown in the picture below. The Board and Screens of the player are partly shown “inverted” to explain the idea of the game.

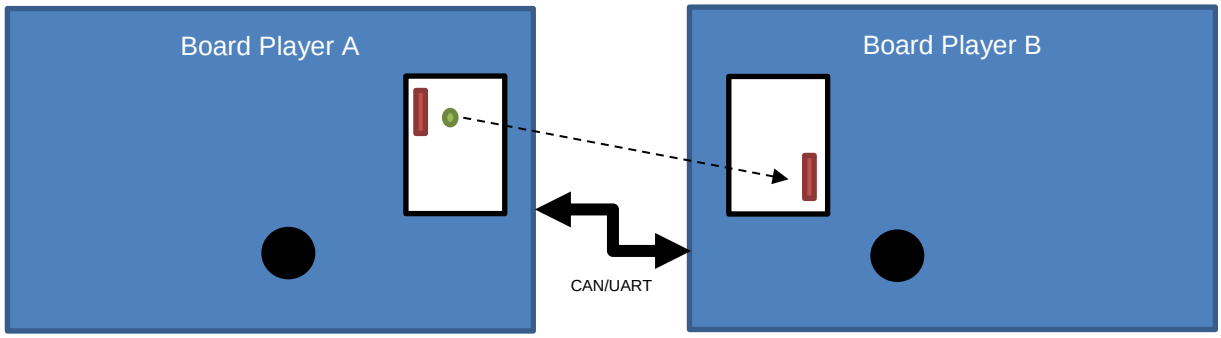


Figure 90 – Soccer

8.4.1 Implementation

Having the game of Pong as a starting point, we need to

- Change the logic of reflection (easy)
- Add a communication protocol
- Extend the state machine of the game

Extending the state machine means, that we have two states in the playing mode. ISPLAYING_BALLVISIBLE and ISPLAYING_BALLOTHERBOARD. If one board is in the mode ISPLAYING_BALLVISIBLE, the other board must be in the state ISPLAYING_BALLOTHERBOARD.

This is more challenging that it may seem from a first glance

- We want to have the same code base on both boards
- We do not know, which boards

This means, we need to synchronise both boards at the beginning. They have to agree which board is starting. One option could be, to have the board starting where the user presses the button 3 to start

the game first. This boards send the other board a protocol to claim the start. Again, this brings a couple of challenges:

- The protocol transmission is not atomic, i.e. both could send the claim at the same time
- One player might want to start the game, but the other player has not pressed button 3 to start.

The second issue can be resolved rather easy, as we can translate the claim protocol into a start event.

The first problem can be ignored for a first implementation, by assuming, that the players will agree who is pressing the button first.

Concerning the communication during the game, we have 2 options

- Either both boards are permanently broadcasting their state
- Or the board have the ball only translates the state when the ball leaves the board

The first option allows more error checking and synchronisation, the second one probably is easier to be implemented.

8.4.2 Hints

Use another RTE signal for the communication between the boards. This automatically introduces a certain flexibility when we want to change the communication interface later-on.

If you plan to use the UART, it is recommended to define a static protocol with a fixed number of bytes. This allows you to use DMA for translating the protocol into the signal.

- The DMA fires an interrupt once a full protocol is received by counting the characters
- The interrupt calls the scaler of the signal, reading out the DMA buffer and creating a signal
- An OnData event of the signal will trigger the application

Alternatively, you can use the traditional concept of an EOP character and ISR's.

When using the CAN, try to limit yourself to 8 bytes payload (should be no problem). I.e. avoid the necessity of introducing a transport layer.

9 Debugging Techniques

Debugging embedded systems may be challenging, as we need to get information from the embedded controller to our host system, typically a PC. This should happen in a non-intrinsic way, i.e. the time behavior should be affected as less as possible – a non-trivial requirement especially for multithreaded applications. Furthermore, we might need to get diagnostic information after the development phase, without a debugger being connected. As a consequence, we have to develop own debugging aids for a variety of different problems.

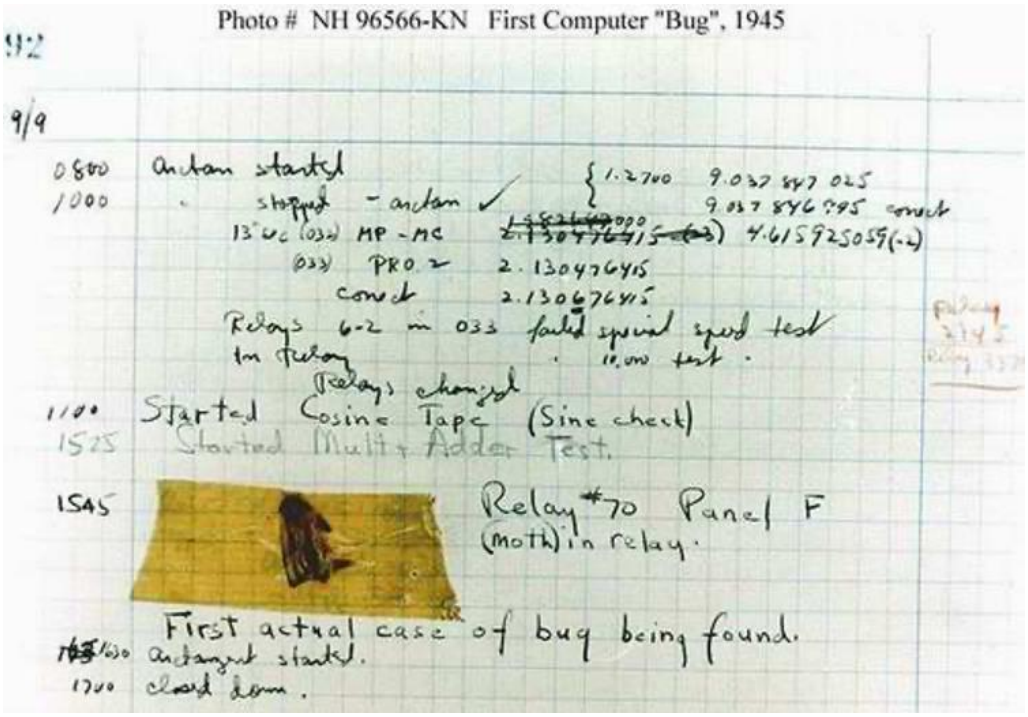


Figure 91 – The first bug

9.1 Using the PSOC Debugger

Effort: 4h	Category - A
Debugger basics, using external pins for debugging	

How can you find a bug? The typical answer might be – use the debugger. The answer is not correctly wrong, but unfortunately the debugger is not a magic tool to find wrong code, it only provides some (limited) features to analyse the system. In other words, the debugger can be considered as a system analyser. It mainly supports the developer by providing features such as

- Stopping upon breakpoints
- Reading (and writing) to variables

More advanced systems allow the developer to look at the execution history by collecting and evaluating trace data.

9.1.1 Freeze Mode Debugging

Freeze mode debugging describes debugging techniques, for which the target is stopped. Most embedded targets support this technique by using breakpoints. During a breakpoint, the CPU is halted and the cost system can read/write data via a debug port, nowadays often a JTag.

Breakpoints

Let's assume we have a small application, which should draw a square on the TFT upon button press, which is not working as it should.

A breakpoint is set as shown in the window below to the entry point of the function drawing the TFT.

The primary use case is to check if this line of code is executed, i.e. if the input signals (in the example a button press) triggered a sequence, which finally calls this line. If yes, the problem might be related to the TFT function. If no, the problem is hidden in the input sequence.

In the example below, upon button press an interrupt will be fired, which calls an ISR sending an event to the task tsk_event.

If the breakpoint is hit after pressing the corresponding button, we may assume, that the ISR/event mechanisms used so far are working fine and we continue debugging the TFT function. If the breakpoint is not hit, we need to find out why. Before stepping through the code, it is time to use the most powerful tool when debugging – our brain. What could be possible reasons for the code not being executed? In this example, we might come up with the following list, which then is examined by setting breakpoint at appropriate other places.

- Is the ISR called?
- Is the event fired?
- Does the task receive the event?
- Is the logic upon receiving the event correct?

Let's assume, the task does not receive the event. What could be the cause for this?

- The task maybe is not activated?

- The event mask does not include the event?
- An ISR or a higher priority task prevents the scheduler to execute the task

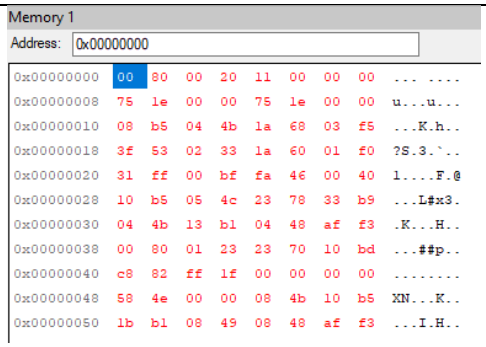
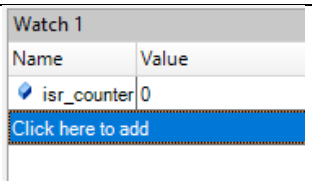
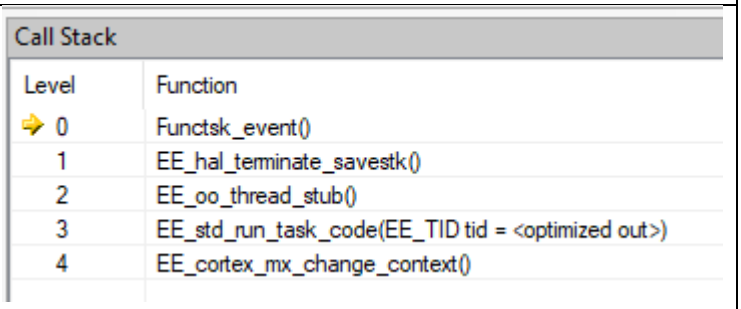
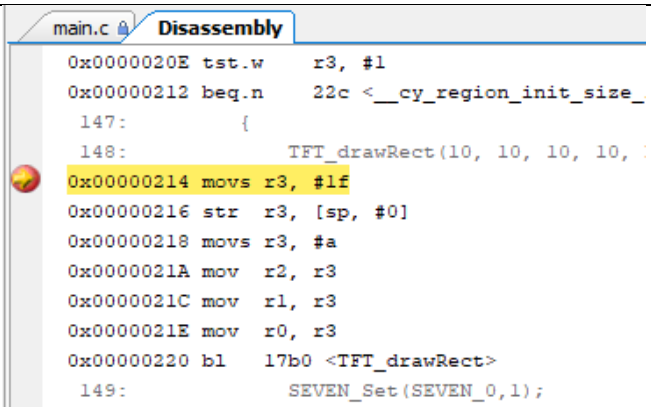
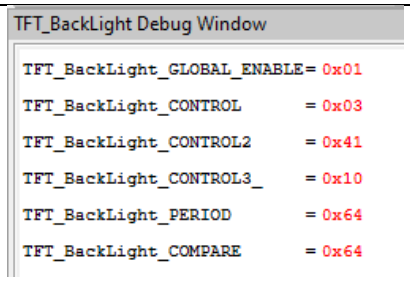
Again, breakpoints may be used to answer these questions. Gradually you will learn more and more abput the true system behavior

```
138
139
140     while(1)
141     {
142         WaitEvent (ev_Button_1 | ev_Button_2 | ev_Button_3 | ev_Button_4);
143         GetEvent(tsk_event, &ev);
144         ClearEvent(ev);
145
146         if (ev & ev_Button_1)
147         {
148             TFT_drawRect(10, 10, 10, 10, RED);
149             SEVEN_Set(SEVEN_0,1);
150         }
151
152         if (ev & ev_Button_2)
153         {
154             SEVEN_Set(SEVEN_0,2);
155         }
156
157         if (ev & ev_Button_3)
158         {
159             SEVEN_Set(SEVEN_0,3);
160         }
161
162         if (ev & ev_Button_4)
163         {
164             SEVEN_Set(SEVEN_0,4);
165         }
166
167     }
168
```

Figure 92 – Setting a breakpoint

Once a breakpoint is hit, various additional windows can be used to analyse the system state.

Locals	The locals window shows the value of local variables, if they are visible to the debugger (restrictions may exist e.g. for locals stored in registers only.	Locals Name | Value | Address | Type | Radix ev | 1 | 0x1FFFA02C (All) | unsigned int | Default
Registers	The registers window can be used to explore the values of the CPU registers.	Registers r0 = 0x00000000 r1 = 0x00000004 r2 = 0x00000001 r3 = 0x00000001 r4 = 0x00000000 r7 = 0x00000000 r8 = 0x00000000 r9 = 0x00000000 r10 = 0x00000000 r11 = 0x00000000 lr = 0x0000020D pc = 0x00000214 xpsr= 0x01000000 msp = 0x1FFFA020 psp = 0x00000000

Memory	The memory window can be explored to investigate the content of the memory	
Watch	Global variables can be analysed in the watch window	
Call Stack	The call stack provides the list of functions which have been executed as a sequence for reaching the breakpoint. You can click on the functions calls to trace back the execution sequence.	
Disassembly	Shows the assembly code and true assembly location of the breakpoint	
Component	Shows the current state of peripheral registers	

Watchpoints

Watchpoints are similar to breakpoints, but the work on (global) data instead on code locations. I.e. you can break the code if a specific variable is read or written to. Please not, the code stops in the area of the read/write operation, but not necessarily exactly on the line.

Breakpoint Limitations

Unfortunately, breakpoints come with a couple of limitations.

Breakpoints are intrinsic	This is probably the biggest problem when debugging realtime applications, as hitting a breakpoint will seriously affect the time behaviour and as a result, e.g. sensor values are getting lost, communication is not working etc, resulting in an invalid state once the breakpoint has been hit. As a consequence, resuming the system once a breakpoint has been hit will not work	Use run mode techniques instead (see below)
Available number of breakpoints is limited	Most controllers only provide a limited number of breakpoints	We could assembly instructions setting a breakpoint, e.g. asm("bkpt"); This could be useful in a central error handler or other locations which indicate an error condition. When a debugger is not connected, the debug instruction will behave like a nop.
Breakpoint location may be fuzzy and even optimized out	In the debugger, we set a breakpoint on a C-code line. In reality however, the breakpoint is set on an address containing an assembly instruction. Especially when using a high optimization level, the assembly instruction may be a bit away from the C-code line or even optimized out.	Use an empty instruction to get an address for an explicit breakpoint using the asm("nop"); instruction.
Breakpoints and interrupts	Depending on the system, breakpoints may also stop the interrupt controller or not. This may (and typically) will result in interrupts being lost when a breakpoint is hit	
Breakpoints and peripherals	Especially on the PSOC, breakpoints are rather slow. As a result, peripherals will continue to run while the breakpoint is being processed. As a consequence, registers of peripherals will not contain the value which was valid when the breakpoint was set, but typically a much younger value	Use run mode techniques instead (see below)

Stepping

Stepping through the code is also based on breakpoints. A breakpoint is set (automatically) on the next line and the CPU is started. I.e. all limitations discussed so far are also valid when stepping through the code.

9.1.2 Run Mode Debugging

As opposite to freeze mode debugging, which stops the CPU during the debug operation, run mode debugging allows reading and writing data “on the fly”. Technically, different solutions exist on the market. The goal with all solutions is to affect the system behaviour as less as possible. I.e. the debugging should be as much non-intrinsic as possible. Strictly spoken, a truly non-intrinsic solution does not exist on the market, as every operation will affect the time behaviour of the target to a certain extend. But if this effect is short enough, we may assume the system to remain in a defined state during the debugging operation. If this is true, depends strongly on the debug scenario and the realtime constrains present.

- Often, the system is stopped via the JTag for a very short interval to perform the required read/write operation
- An alternative (more expensive) solution is to use a trace solution, which allows the developer to read data and to check the control flow in an almost non-intrinsic way (as sometimes wait-states have to be injected)
- And last but not least software monitors may be added to the code, which transmit the required data via port interfaces

As the PSOC does not provide built-in run-mode debugging, own solutions must be implemented. Some ideas are presented below. Feel free to extend them as needed…

UART Log and other ports

The easiest interface probably still is the UART port, as you can easily connect this to a PC USB port using a FTDI adapter or by using the logic analyser.

Of course, you can simply use the generated API of the UART component, but this has some drawbacks

- When porting the code to a different project, you have to keep the name of the UART component
- The API only accepts strings

To overcome these limitations, let’s develop a logging component, which fulfils the following requirements

Req-Id	Description
FR1	The component provides a printf like interface to print flexible logging messages.
FR2	As a first parameter, a identifier string will be printed (e.g. identifying the function from where it is called)
FR3	The next parameter is a string containing % specifiers like print f, followed by a variable list of variables to be printed
FR4	The component shall be configurable to work with any UART component, independent of the name (hint: use the concat preprocessor operation)
FR5	The component shall have different verbosity levels VERBOSE – every message is printed

	INFO – only INFO and higher levels are printed WARNING – only WARNING and higher levels are printed ERROR – only ERROR and higher levels are printed DEBUG – only DEBUG and higher levels are printed LOG_OFF – Logging is turned off
FR6	Use the API for printing logs of different levels LOG_V, _I, _W, _E, _D
NFR1	Provide a configuration file to configure the verbosity level and UART component

In order to verify the time impact of the component, you can perform the following tests

- Use a UART component without software buffer, i.e. we only use the 4 bytes hardware buffer to send out a message. The expected approximate delay (using a 115000, 8N1 UART configuration) is (NumberOfBytes – 4) / 10 = blockingTimeInMs
- Now let's add a sufficiently large UART buffer and check the resulting runtime of the function. The blocking time should be significantly lower, as the blocking time now is dependant on the copy operation to the buffer.

Hint: Use the timing analyser of the next exercise to check the timing.

Time with 4byte HW buffer

Time with 400byte SW buffer

Alternative Interfaces

An alternative interface to provide debugging information is the DAC. By providing different voltage levels which are representing different states, we can use a Logic Analyser or Scope to read out the data.

A possible use case could be to hunt a leakage bug where we expect a problem with the dynamic release of resources, i.e. not every malloc is followed by a free in a messaging scenario. We can use a global variable which is incremented upon every malloc and decremented by every free operation.

This variable can be written to a DAC (e.g. using a DMA). When the value drops below a minimum value, we have an indicator that free is called to often. When the value grows permanently, we have an indicator that we have mallocs without free.

Another use case would be to visualize the state of a state machine over time.

As opposite to a simple GPIO port, a DAC can provide more states for evaluation. Probably not the resolution of the signal, but depending on the quality/resolution of the DAC and measurement equipment, 10 and more states can be coded. By setting the DAC depending on the state of the state machine, we can follow the states over time using a logic analyser. If we use another DAC or set of GPIO pins to represent events, the complete behaviour of the state machine can be visualised and traced.

RAM based Freeze Frames / Persistent Storage

Especially the GPIO and DAC port techniques come with the limitation, that the logged data is overwritten after a certain time. Using the UART, the data typically persists for a longer period, as it is stored in the terminal application and in most cases can be stored in a textfile.

An alternative – moving towards a diagnostic system - is to create a (smaller) buffer on the target system. This allows a very fast storage of process data, which can be evaluated after a testrun. A simple option e.g. is to store the data on the target and after the testrun use a UART interface or similar to transfer it to a host system. This is particularly helpful, when no physical data line can be used during the testrun.

An extension to this is to use persistent memory like Flash or EEPROM to store the diagnostic data. This process is typically more complex and more intrinsic, but has the advantage that the data persists after a reset or power-on cycle.

9.2 Using the Logic Analyser

Effort: 2h	Category - A
Logic Analyser, Debugging, Pin Behaviour	

As the programs have been very simple up to now, no real debugging has been required. Of course the PSOC also has a debugger, but using a simple freeze mode debugger (allowing to set breakpoints and to step through the code) on an embedded system often is not enough and has some important drawbacks:

- When halting the system on a breakpoint, the time behaviour will be corrupted, i.e. the system will often not work after stopping, e.g. because a communication port is not serviced in time
- Stopping the PSOC CPU unfortunately is not in sync with stopping the peripherals, i.e. we will often see, that the peripherals continue working for a certain time, although the CPU already has stopped, causing strange effects

More expensive debugger, e.g. from Lauterbach or Green Hills provide way more advanced features but the key problem persists. Stopping the core corrupts the behaviour of the system. What we need is a non-intrinsic way of debugging, i.e. we want to be able to get information without changing the (real) time behaviour of the system. For example, by setting a pin to high or low level. Strictly spoken this also affects the behaviour of the system, but the interval is so short, that in most application it can be neglected.

Another typical debugging use case is to check the correct behaviour of a port pin, e.g. a UART communication pin. When connecting this pin to a UART receiver and we get the correct output, everything is fine. If not – and most of the time this will be the case – we do not know if the sender or receiver is working as expected and we might want to look at the physical signal.

A helpful tool for this is an oscilloscope or a logic analyser. Both work in a similar way. The signal voltage is measure and the voltage value is shown as a plot.

Property	Oscilloscope	Logic Analyser
Signal Type	Digital / Analog	Mostly digital only
Sample Rate	Up to GB/s	Up to MB/s
Sample Buffer	Short	Medium

The AZ Delivery Logic Analyser is a low cost 24MHz 8 Channel Analyser, which allows to debug signal having a frequency below approx. 1MHz. As most of the application will work with a 1ms cycle time, this will be good for most use case. But of course it will not work to debug a 30MHz SPI bus.

9.2.1 Setting up the device

Please connect the flat ribbon cable to the pins of the logic analyser. Thou outermost cable (typically red) should be connected to channel 1, the next cable to channel 2 and so on. The black cable should be connected to one of the 2 ground pins.

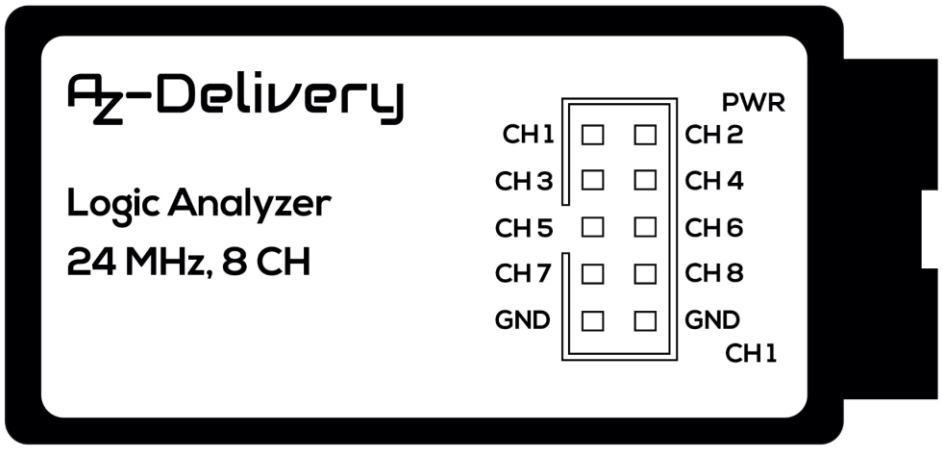


Figure 93 – Pin Connection AZ-Delivery Logic Analyser

This logic analyser is compatible to the professional variant of the company Saleae, therefore we can install the Logic tool from that company.

https://logic2api.saleae.com/download?os=windows&arch=x64

Alternatively, the Sigrok API can be used with a variety of tools. Check the eBook at

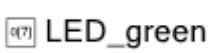
https://cdn.shopify.com/s/files/1/1509/1638/files/AZ172_C2-7_EN_B01MUFRHQ2_26c04f7a-b9ce-4dcd-aff7-9287750a4366.pdf?v=1721031008

Note: Use google, in case the links have been changed.

9.2.2 A first application

For demonstrating the use of the logic analyser, we will build a very simple toggling LED application.

In the hardware schematic, add a pin to drive the green LED.



Then, add the following code to your project, which will make the LED toggle every 100ms.

```
int main()
{
    uint8_t ledState = 0;

    for(;;)
    {
        ledState = (ledState + 1) % 2;
        LED_green_Write(ledState);
        CyDelay(100);
    }
}
```

Connect the Logic Analyser GND wire to a GND pin of the PSIC and channel 1 to Pin 0[7]. Start the logic analyser software, you should see the following output:

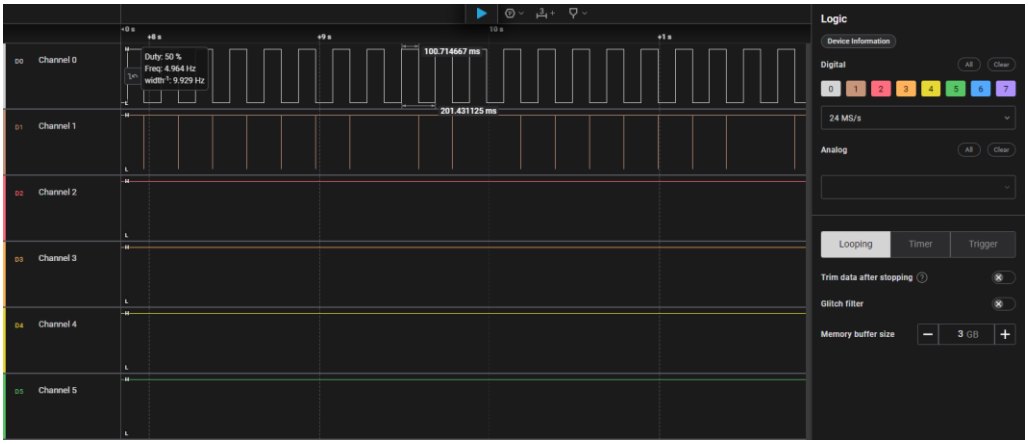


Figure 94 – a toggling LED on the logic analyser

As expected, channel 0[15] shows the toggling of the LED.

You might notice, that the time is not 100% precise. Explain why!

Furthermore, we see some strange behavior on channel 1. Explain.

Now, comment out the CyDelay Line and run the program.

You should see the following output:

[15] Note, that the default naming of the channels in the Logic 2 software differs from the pin naming in the hardware. You can change the names in the Logic software to describe the functionality of the pin, in our case e.g. LED green

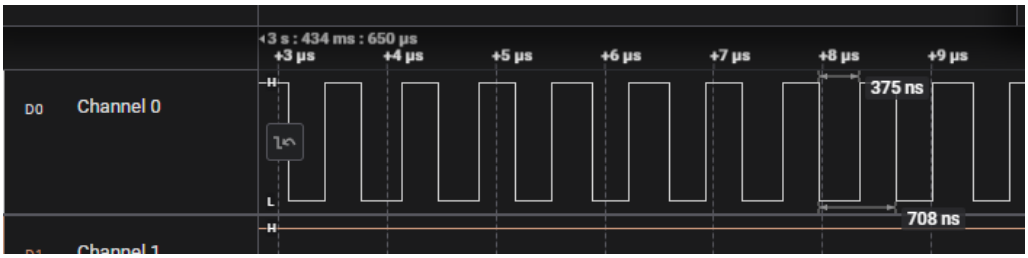


Figure 95 – Measuring code execution time

Obviously, it can be read as the execution time of the code in the loop takes 375ns.

How can you check, if this is a plausible value? Checking the clock tree, we will find out that in this project the CPU runs with 72MHz[16].

This translates to 375 10^{-9} x 72 10^{6} = 27 CPU cycles, which might roughly be the number of assembly instructions in our code. This can be verified by either creating a LIST file or by checking the disassembly window in the debugger.

```
0x00000084 <main>:
  33: #include "project.h"
  34: #include "global.h"
  35: int main()
  36: {
0x00000084 push {r4, lr}
  38:     uint8_t ledState = 0;
0x00000086 movs r4, #0
  39:
  40:     for(;;)
  41:     {
  42:         ledState = (ledState + 1) % 2;
0x00000088 adds r0, r4, #1
0x0000008A ldr  r3, [pc, #18]   ; (a4 <CYDEV_CACHE_SIZE+0x8>)
0x0000008C ands r0, r3
0x0000008E bpl.n    98 <main+0x14>
0x00000090 subs r0, #1
0x00000092 orn  r0, r0, #1
0x00000096 adds r0, #1
0x00000098 uxtb r4, r0
  43:
  44:         LED_green_Write(ledState);
0x0000009A mov  r0, r4
0x0000009C bl   46c <LED_green_Write>
0x000000A0 b.n  88 <main+0x4>
0x000000A2 nop
0x000000A4 .word    0x80000001
```

Figure 96 – Disassembly of the toggling LED

Hint: From now on, use the logic analyser as an additional debug aid for your further exercises!

9.2.3 Analysing more complex signals

Up to now, we have used the logic analyser to analyse rather slow digital signals. The logic analyser however can also be used to analyse e.g. communication signals, like SPI, UART, I2C and similar.

[16] Default is 24MHz.

The picture below shows the sniffing of a UART port on channel 0. The signal on channel 2 is a buggy signal caused by crosstalk.

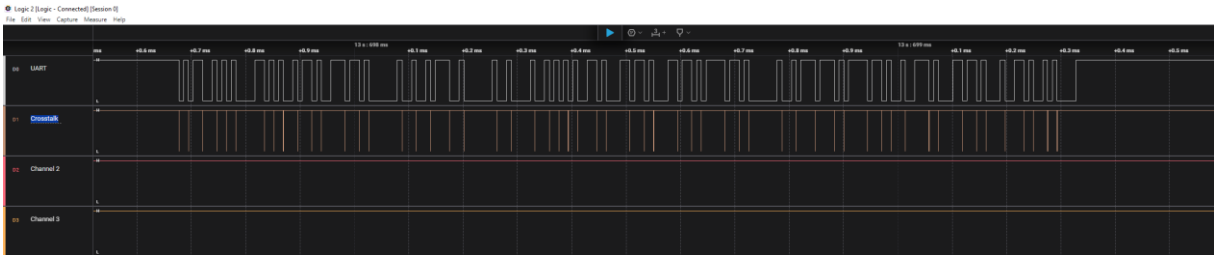


Figure 97 – UART Logging with the logic analyser

The advantage of using a logic analyser over programs like HTerm is that you can also see at what time a protocol has been transmitted. This might be relevant, when you expect a certain sequence of question/answer protocols exchanged between different controllers.

In order to see the content of the transmission, you can add an analyser to a channel. For this, select the Analyzer tab and select and configure the correct analyser (Note – you can download additional analysers from the Internet if needed).

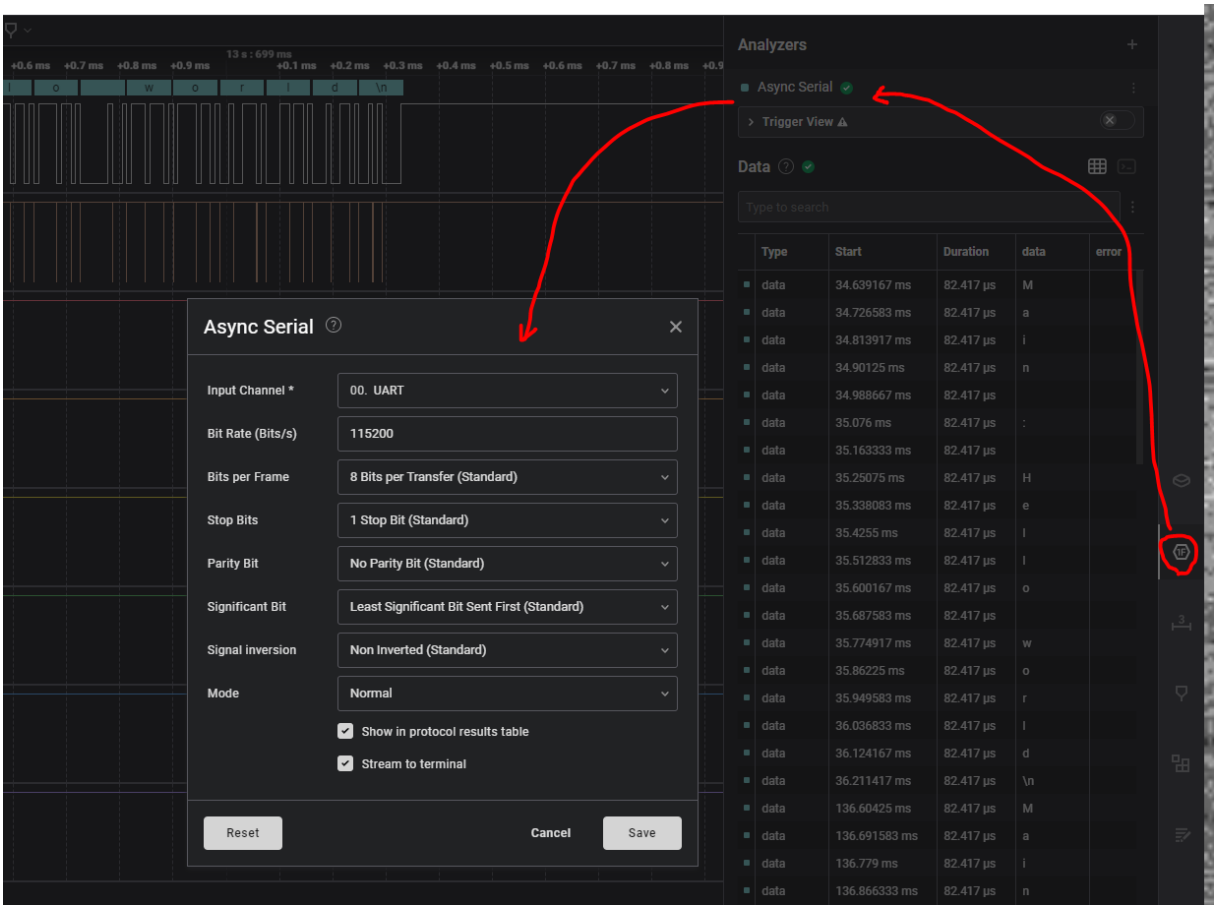


Figure 98 – Channel analyser configuration

The content of the protocol will be shown on top of the data stream as well as in the data table. You can switch between a console view (similar to HTerm) and a data view including information like timestamps and transmission errors.

Another very helpful feature is the trigger view, which allows you to create simple triggers and show a window of signals based on this trigger. In the following example, the system will show the Green LED (Channel 2) being set just before the protocol being sent out.

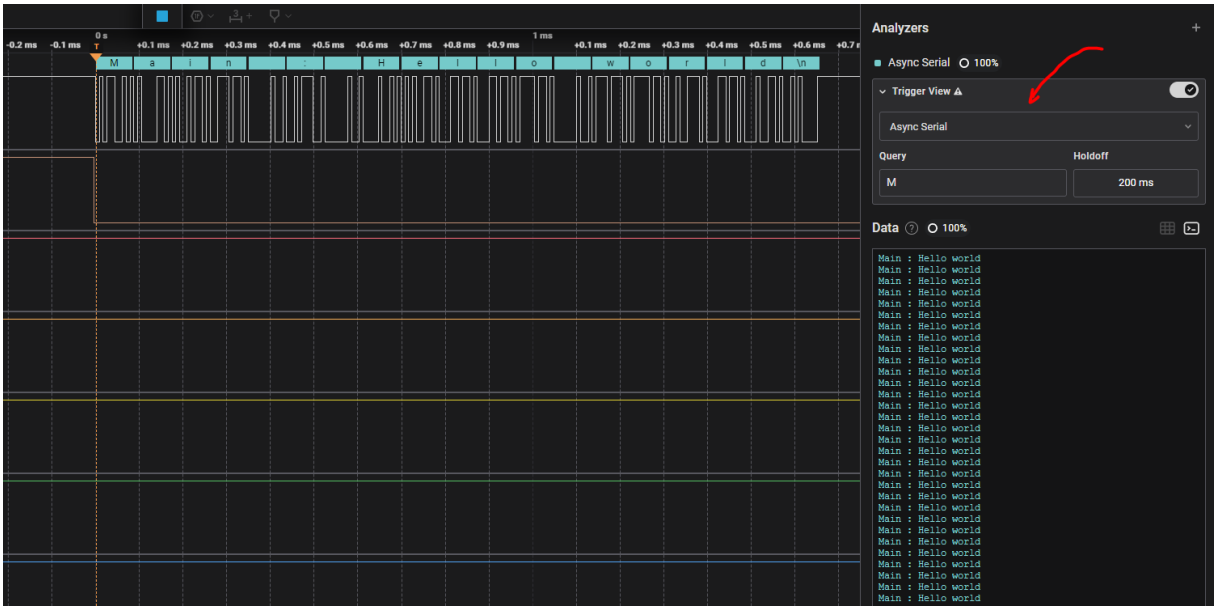


Figure 99 – Trigger using an analyser

Code creating the sequence above (Note the slightly earlier edge of the GPIO pin as compared to the UART signal):

```
    for(;;)
    {
        ledState = (ledState + 1) % 2;

        LED_green_Write(ledState);
        LOG_D("Main","Hello world");

        CyDelay(100);
    }
```

9.3 Timing Analyzer

Effort: 5h	Category - B
Logic Analyser, Stae Machines, Interrupts, Timing	

In complex applications, different tasks are processed with different priorities and within certain timing constraints. It is important, that each interrupt is executed with correct timing, and that we also check the execution time, in order to keep an eye on the CPU load.

For simple applications the debugger works well – but as you already have learned is, there is no perfect synchronization of the CPU and the hardware when a breakpoint is reached. That means, that e.g. a hardware counter might go on counting a little bit while the CPU is already stopped. That means, that debugging in the presence of interrupts does not make sense.

But, how can we check, which interrupt preempts another interrupt and how long? Therefore, we mustn't stop the system. We're simply measuring the timing behaviour. We have two options for that:

- Measure internally, by counting CPU cycles or using a timer.
- Measure externally, by attaching a logic analyzer.

Therefore, your task is to develop an easy to use API, which we can use for time measurement of any code regions in your code. Multiple analysers must be working in parallel. This API will be used in the next lab experiments as well.

9.3.1 Requirements

We want to be able to configure the analyzer based on the intended use. Three techniques should be selectable:

- SysTick timer at 1ms for long-term measurements
- CPU cycle counter from the DWT trace unit for precision measurements
- Hardware pins for external measurement with a logic analyzer (which can be used in addition to the time measurements mentioned above)

Req-Id	Description
NFR1	The system will be developed as "bare metal," without an operating system.
NFR2	All status information and IDs of the analyzers will be stored in self-explanatory structs and described by self-explanatory enums.
NFR3	Use the concept of object oriented C as presented in the presemester class.
NFR4	For configuring access to the output pins, use function pointers to the respective API call.
FR5	We want to **initalize** the API. This includes the start of the necessary peripherals.
FR6	We want to **create (or configure)** an analyzer with one of the following modes: • DWT Cycle Counter • DWT Cycle Counter + Output Pin • SYSTICK

	• SYSTICK + Output Pin • Output Pin only Moreover we want to give the instance of the analyzer a string name, which will be printed out in the print function. Hint: Use the red, green and yellow LEDs as output pins. Recommended API call: `TimingAnalyzer_create(me, mode, pin, name);`
FR7	We want to **print the status** of a single timer: • For SysTick timers, the elapsed time will be given in milliseconds. • For the DWT counter, the elapsed time will be given in milliseconds with six decimal places, and the elapsed CPU cycles will also be displayed. The string to be printed must be fully assembled before being sent to the UART (i.e., UART_LOG_PutString() should only be called once per print). Hint: For concatenation of strings, keep in mind, that the return value of `sprintf()` is the number of bytes written by `sprintf()`.
FR8	We want to **print all** existing analyzers.
FR9	We want to **start** an analyzer. This is only possible when it is configured and not currently running. If the analyzer is paused, it should resume.
FR10	We want to **stop** an analyzer and calculate the elapsed time. This is only possible when the analyzer is running or paused. The elapsed time will be stored in the analyzer's struct.
FR11	We want to **pause** an analyzer. The elapsed time since the last start will be added to the analyzer's total duration. This is only possible when the analyzer is running.
FR12	We want to **resume** an analyzer. This is only possible when the analyzer is paused.
FR13	We want to use the **Systick Interrupt** with an interval of 1ms.

Some more non-functional requirements

Req-Id	Description
NFR14	Fetching the cycle counter or SysTick counter must be the first action within the start, stop, pause, or resume functions for precise measurement.

9.3.2 Planning

As the complexity of this exercise may introduce a certain challenge, we will try to cut it into smaller pieces and to follow the concept of a skinny sheep, i.e. we are going to implement a simple version with limited capabilities first and then add feature by feature.

A pretty simple version of the timing analyzer comprises of the following features

- We have one measurement principle (e.g. the cycle counter)
- We only have start and stop features
- Implemented in a functional way

The next version then will add

- Additional measurement concepts
- Introduce the object oriented design
- And extend the functionality pause and resume

Very important: Every increment should follow the iterations

- Figure out, how a specific technology (e.g. counter) works
- Refine the given requirements and design the interface / algorithm
- Implement the code
- Test the code

9.3.3 Analysis

Before we start with the implementation, let's try to develop a first idea by checking the following technical concepts.

The time measurement using the cycle counter highly depends on the CPU frequency. The current CPU frequency can be accessed via the define `BCLK__BUS_CLK__HZ`. Check this value and compare it to your clock settings. Which CPU frequency do you use in your project?

The time measurement using the cycle counter highly depends on the CPU frequency. The current CPU frequency can be accessed via the define `BCLK__BUS_CLK__HZ`. Check this value and compare it to your clock settings. Which CPU frequency do you use in your project?

Which will be the longest intervals that can be measured with the SysTick and with the DWT Cycle Counter without generating an overflow?

How do we calculate the time in ns from the CPU cycle counter without loss of precision and without overflows?

Perform an object oriented analysis for the timing analyzer objects. Which elements should be included in the data structure for a timing analyzer?

Which enums do we need to describe the current config and status of the timer?

Which API calls do we need?

Which peripherals need to be started / initialized in the start function?

9.3.4 Implementation

In order to implement the required functionality, you need to understand the DWT Cycle Counter and the Systick Counter. For this, you need to find the relevant information. The board consists of different components, provided by different companies. The following picture shows the structure.

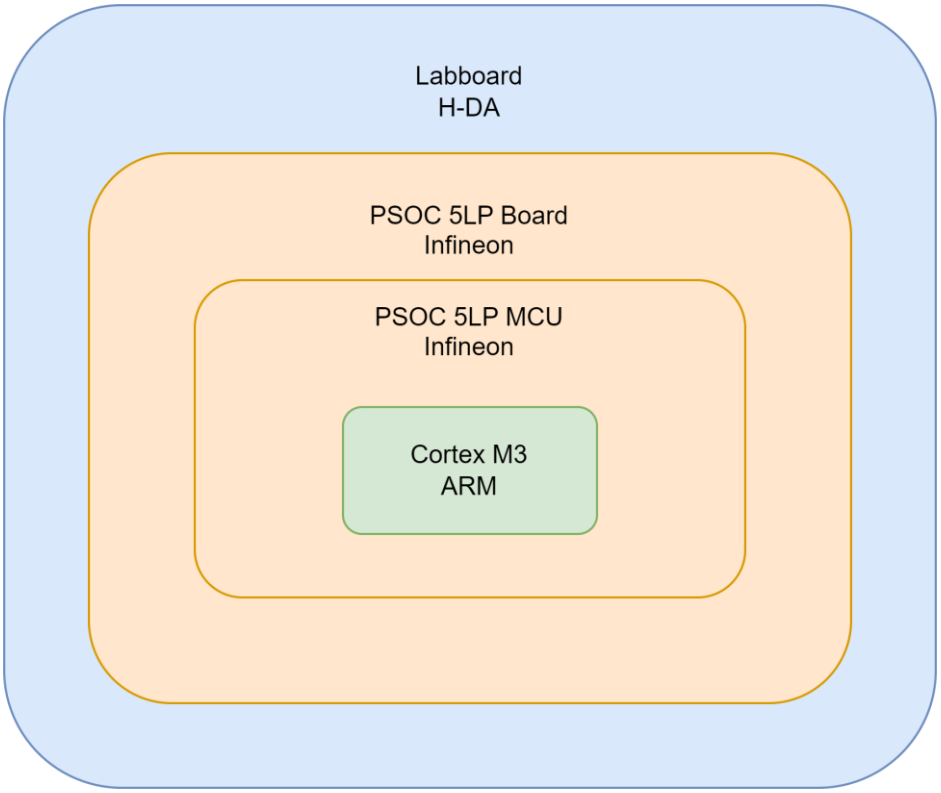


Figure 100 – Labboard Vendor Structure

Both the DWT Counter as well as the Systick Counter are properties of the ARM core itself, i.e. they are provided by the company ARM. A good starting point to find documentation about the ARM is: https://developer.arm.com/

The DWT counter has been introduced in the lecture. Check the code and make sure that you understand the registers being used.

For the Coredebug register as well as the cycle counter check the ARM Core manual on

https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/Debug-register-support-in-the-SCS/Debug-Exception-and-Monitor-Control-Register--DEMCR?lang=en

https://developer.arm.com/documentation/ddi0403/d/Debug-Architecture/ARMv7-M-Debug/The-Data-Watchpoint-and-Trace-unit/CYCCNT-cycle-counter-and-related-timers?lang=en

Also, the Systick counter is a component of the Cortex core, i.e. there is no hardware peripheral component for this.

You might want to check the following page for reference information

https://developer.arm.com/documentation/dui0552/a/cortex-m3-peripherals/system-timer--systick

Unfortunately the provided documentation does not provide information on the specific implementation on the PSOC. E.g. aspects such as which API do I need to use remain open.

As we have already figured out, that both components belong to the core, we need to check the file `core_cm3.h`, which contains all core specific definitions.

For the DWT, the following structure is interesting: `DWT_Type`

For the Systick, accordingly: `SysTick_Type`

In addition, please check the file `cylib.c`, which contains relevant API functions. The Infineon community is also a good place to find additional samples and answers.

https://community.infineon.com

Implement a skinny sheep to get both counters running and then design a proper API.

A note on critical regions. Critical regions or resources are shared typically stateful resources, which may be read by several consumers, but only written once or at a specific place. In the implementation, the DWT counter as well as the SysTick counter are such resources. If one timer object changes the state (e.g. by resetting the counter to 0), this will affect all other timing analysis objects – probably not what we want to have. I.e. only read-operations are allowed for the DWT counter and SystTick Timer after the peripherals have started.

9.3.5 Test Basic Functionality

Now we need to write code snippets to test the functionality of the API.

Write code in the main.c with CyDelay() to prove that:

- start / stop and print works properly
- pause / resume works as expected
- multiple timers can be used in parallel without interfering

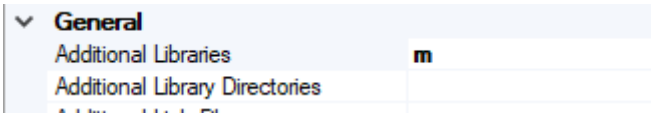
Paste the terminal output from your basic tests:

9.3.6 Test Mathematical Functions

Measure the durations of the following calculations:

- 1000 integer additions
- 1000 integer multiplications /divisons
- 1000 float additions
- 1000 float multiplications /divisons
- 1000 Square root calculations / sin() / sinf()

Note: You need to add the math library m to the linker.



Test	Time (ms)	CPU cycles	Time (Logic Analyzer)	Comment
integer additions				
integer multiplications				
integer divisons				
float additions				
float multiplications				
float divisons				
sqrt()				
sin()				
sinf()				

9.3.7 Interrupts

Set up the following test scenario:

Implement two timer interrupts, both with same priority (7). The following actions within the Interrupt Service Routines (ISR) shall be performed:

- `ISR_1ms` (running @ 1 ms)
  - Start a time measurement (test the different timing sources)
  - clear the pending bits
  - Stop
  - return
- `ISR_2seconds` (running @ 2 seconds)
  - Start a time measurement (test the different timing sources)
  - Do some calculations or implement a delay for approx. 1 second
  - Stop or pause the time measurement
  - Print the elapsed time
  - return

Write a test table for the following scenarios (see next page)

1. Measure the duration of the code in both ISRs
2. Change the priority of `ISR_1ms` to prio4 (higher priority than `ISR_2seconds`)
3. Change the cycle time of Timer1 from 1ms to 0.1ms
4. Change the priority of ISR2 to prio0

Explain, where the differences in the time measurement come from when stepping though the above test cases.

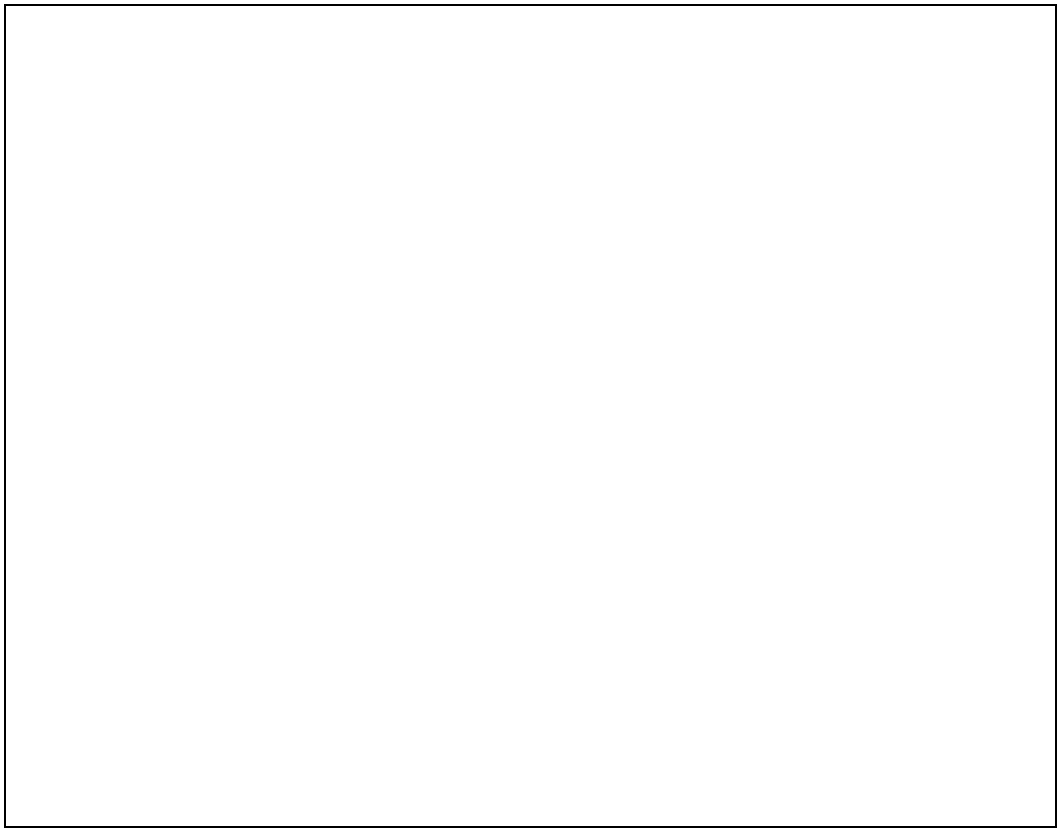
Moreover, compare the time measurement from the logic analyzer to the internal measurement methods and describe the source for possible differences.

Test Nr.	Duration (SysTick)		Duration (cycle counter)		Duration (Logic Analyzer)		Observation	Justification
	ISR_1ms	ISR_2sec	ISR_1ms	ISR_2sec	ISR_1ms	ISR_2sec		
1								
2								

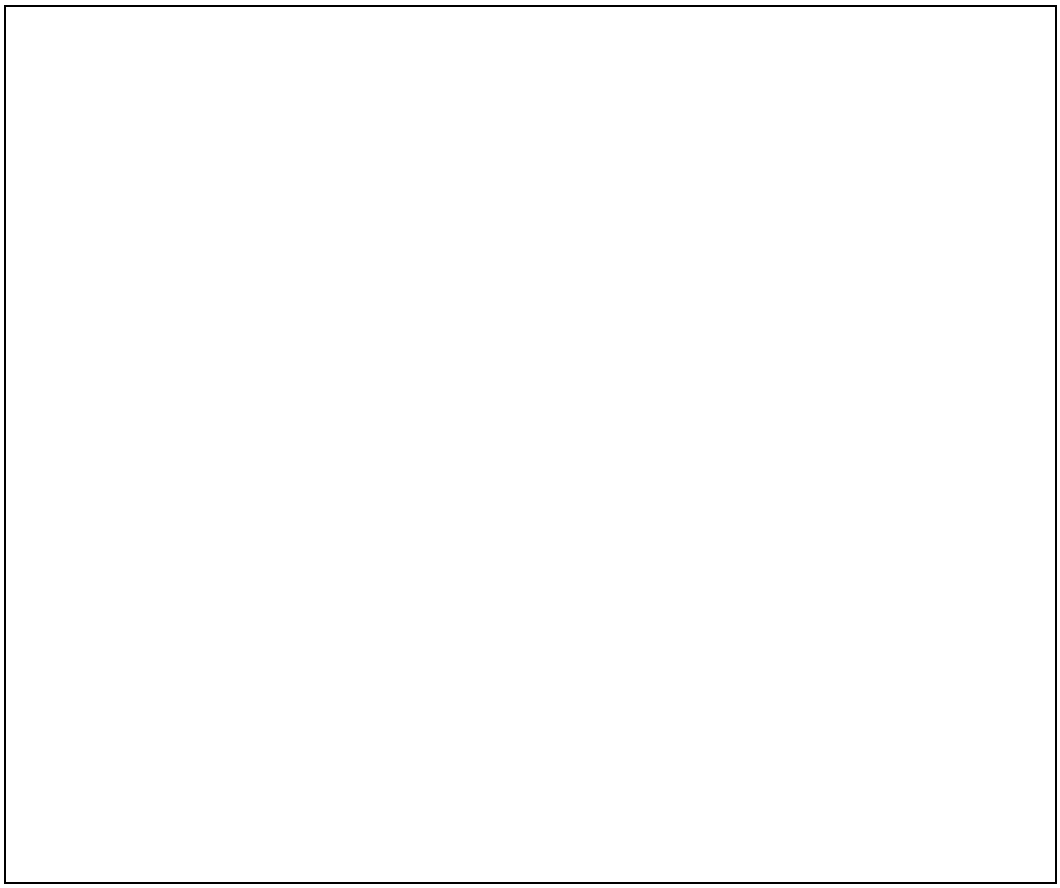
3								
4								

9.3.8 Extra Task (optional)

As we know, the debugger significantly changes timing behavior. However, consider the internal and external measurement methods mentioned above: does the timing behavior truly remain unaffected? Briefly discuss which situations might cause issues and how we can minimize their impact on the timing behavior of our tasks:



Check the disassembly and find out, how many cycles approximately get lost when calling start / stop / pause / resume to that point, where the counters are read.



9.4 Stack Measurement

Effort: 5h	Category - C
Stack measurement, linker control file, RTOS task stack, ISR stack, pointers	

Bugs caused by too low stack configurations are hard to detect and to debug. In case the reserved stack area is overwritten, any data which is stored next to this area will be corrupted.

- These bugs belong to the group of stochastic bugs. They only happen from time to time, as the required stack size depends on the dynamic program control flow (i.e. which functions are called
- The bug is not visible when it happens (i.e. during the overwriting of data) but will only become visible when we read the corrupted data later on. And also, in this case, it is not ensured that the corruption can be detected in a reliable manner

Therefore, the stack configuration in most systems is only estimated, without having any real experience data. This may either lead to system crashes, if the memory is estimated too small or to a large memory consumption, if too much memory is assumed.

In the exercise, we will implement a simple (and not 100% precise) stack measurement tool, based on the memory pattern initialisation and corruption technique. In this approach, the stack memory region is filled with a magic pattern, e.g. 0xCC00FFEE. Then the program is executed and afterwards, it is checked how much of the memory is being corrupted – in other words has been used by the stack.

Although this approach sounds simple, it has a couple of challenges, as shown in the requirements.

Req-Id	Description
FR1	The system will be developed as "ERIKA system," supporting the analysis of the system stack as well as RTOS stacks, e.g. for the extended tasks.
NFR2	All status information and IDs of the analyzers will be stored in self-explanatory structs and described by self-explanatory enums.
NFR3	Use the concept of object oriented C as presented in the presemester class.
FR4	Provide an API to initialize the stack area with a magic pattern
FR5	Provide an API to check how much of the pattern has been corrupted
FR6	Provide an API to pront the stack usage as well as the boundaries of the stack region
FR7	Provide a configuration file containing references to the boundaries. If possible, the boundaries should be read from global system objects. Check ERIKA and the linker de-

	scription file for this.
NFR8	Note, that the functions of the stack analyzer might also use the stack by themselves. The impact should be as less intrinsic as possible.
FR9	The Stack analyzer may not overwrite stack memory, which is in use.
NFR10	The code should be portable, i.e. inline assembly should be avoided if possible.

9.4.1 Bare Metal

We will start with the implementation for the bare metal system stack. For this, we need to figure out where the stack typically is stored and how we can access this data. The starting point is to check the generated linker control file, which can be found in the Generated_Source folder and has the extension .ld. In our system architecture, the file is called cm3gcc.ld.

Simply search for the keyword stack and you will find the following interesting sections.

```
/* Provide fall-back values */
PROVIDE(__cy_heap_start = _end);
PROVIDE(__cy_region_num = (__cy_regions_end - __cy_regions) / 16);
PROVIDE(__cy_stack = ORIGIN(ram) + LENGTH(ram));
PROVIDE(__cy_heap_end = __cy_stack - 0x0800);
```

And

```
/* The .stack and .heap sections don't contain any symbols.
 * They are only used for linker to calculate RAM utilization.
 */
.heap (NOLOAD) :
{
  . = _end;
  . += 0x80;
  __cy_heap_limit = .;
} >ram

.stack (__cy_stack - 0x0800) (NOLOAD) :
{
  __cy_stack_limit = .;
  . += 0x0800;
} >ram

/* Check if data + heap + stack exceeds RAM limit */
ASSERT(__cy_stack_limit >= __cy_heap_limit, "region RAM overflowed
with stack")
```

Please note, that the magic values 0x80 for the heap and 0x0800 for the stack are configured in the PSOC creator and identify the max amount of heap and stack the system may occupy before running into a fault.

The symbols defined in the linker file can be used in the code files – they are visible as external symbol.

```
/**
 * Symbols from the linker file containing the stack boundaries
 * Note: The stack goes from high addresses to low addresses
 */
extern uint32_t const __cy_stack[];         //High address, stack start
extern uint32_t const __cy_stack_limit[];   //Low address, stack end
```

Using these addresses, we now can initialise the memory with a magic pattern and check for the corruption in a second API. Please note, that stack may be implemented as rising or falling memory. I.e. our system has to deal with starting addresses, which may be bigger or smaller than the end address.

9.4.2 Erika Extended Tasks

Erika extended tasks have a private task. Unfortunately, we do not know, where and how the RTOS creates the stack. A typical implementation pattern is a global variable (array) of the required stack size to which the stack pointer is set once the task is scheduled. However, also in this case, it is not specified, if the stack is rising or falling.

In order to investigate this a bit deeper, we declare 2 local variables, which are placed on the stack (e.g. by passing their address to the parameter of a function – this will in most cases prevent the variable to be placed in a register instead of the stack).

As shown on the picture below, we have declared 2 locals in the task, ev and delayInMs. In the debugger, we can check the addresses and compare them to the globals in the mapfile. Doing this, we find the variable EE_cortex_mx_stack_5, which also has the size of the stack set in the Erika configuration (x1fffa088 – x1fff9ef8 = x190 = 400). Very likely, this is the array we are looking for.

Comparing the memory addresses of both variables, we can see that the memory is falling. And last but not least we see, that the first variable is not set to the start address of the stack section (which would be A088), but has an offset of some bytes

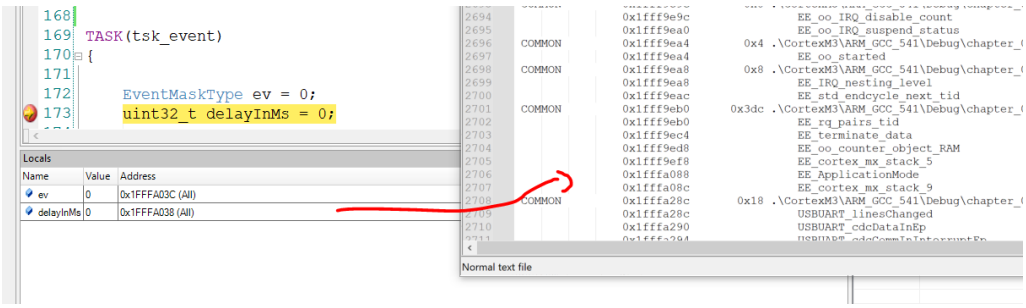


Figure 101 – Identifying stack arrays in Erika

Please note, that the measured values provide a very good approximation, but are also not 100% precise.

- We do not know if our tests really required most stack memory, or if we have another (untested) configuration, requiring even a few bytes more.
- The printf of the STACK API requires around 200bytes of stack
- We did not verify the behaviour on the boundaries, which might result in a +/-1 error
- We simply count the overwritten memory locations. In case the pattern persists between 2 variables, because a write operation was not performed, this is not counted as occupied, which is probably also not truly correct

But at least we have some data we can use to review our stack configuration.

9.4.3 Extension HEAP

A similar solution is possible for the heap memory, but as the heap is not a LIFO Buffer, interpreting the memory holes will be a bit more complex.

9.5 Hardware Faulthandler

Effort: 5h	Category - C
Exception handling, Hardware Faults, Fault Registers	

Until now, errors have been considered mostly in the sense of logical error. Wrong code (the bug) produced a wrong result (the fault) resulting in a wrong or exceptional behaviour of the system. The key goal of writing good code is to identify and handle such faults before a real problem occurs. E.g. by turning off an engine when detecting ^^a wrong calculation of the setvalue.

In addition to these software bugs, also hardware errors may occur which need to be handled. Some of these errors can be caused by buggy software, e.g. when attempting to write data into a read-only area of memory (e.g. flash). Other faults may be caused by the hardware itself, e.g. busfaults.

No matter what kind of a hardfault appeared – they have one aspect in common. The hardware on which the software is running cannot be trusted anymore. This means that the options for handling such a fault are limited

- Let's stop the system (by entering an endless loop)
- Let's try to store some post mortem information which supports the developer in finding the root cause for the problem

9.5.1 Catching Hardfaults

Hardware faults are handled in a similar way like interrupts. In case a hardfault appears, the system will check for an entry in the corresponding vector and will try to execute that code.

The following table shows the exception model of the PSOC 5LP.

Interrupt Number	Exception Type	Priority	Comment
1	Reset	-3 (highest) Not programmable	Reset
2	NMI	-2 Not programmable	Non-Maskable Interrupt
3	Hard Fault	-1 Not Programmable	All fault conditions if the corresponding handler is not enabled
4	Reserved	NA	–
5	Bus Fault	Programmable	Bus error occurs when AHB interface receives an error response from a bus slave (also called prefetch abort if it is an instruction fetch or data abort if it is a data access)
6	Usage Fault	Programmable	Exceptions due to program error
7	Reserved	NA	–
8	Reserved	NA	–
9	Reserved	NA	–
10	Reserved	NA	–
11	SVCall	Programmable	System Service Call
12	Debug Monitor	Programmable	Debug monitor (watchpoints, breakpoints, external debug request)
13	Reserved	NA	–
14	PendSV	Programmable	Pendable request for system device
15	SYSTICK	Programmable	System Tick Timer

Figure 102 – PSOC 5LP Exceptions [001-78426_PSoC5LP_Architecture_TRM.pdf]

Hardware faults depend on the architecture. In case of the Cortex M3, faults are generated by[17]:

- Bus error during an instruction fetch or vector table load or other a data access.
- Internally-detected error such as an undefined instruction.
- Attempting to execute an instruction from a memory region marked as Non-Executable
- If your device contains an MPU, a privilege violation or an attempt to access an unmanaged region causing an MPU fault.
- Unaligned data access
- Division by 0

We will start by creating a simple handler. To set up a handler, check the API

```
cyisraddress CyIntSetSysVector(uint8 number, cyisraddress address)
```

in `cyclib.c`.

For a first testrun, we will install the following handlers and initialise the vectors to find out which is called by different errors.

```
void FAULT_handler_simple1()
{
    asm("bkpt");
}

void FAULT_handler_simple2()
{
    asm("bkpt");
}

void FAULT_handler_simple3()
{
    asm("bkpt");
}

void FAULT_handler_simple4()
{
    asm("bkpt");
}
```

Initialise the vectors:

```
CyIntSetSysVector(CY_INT_HARD_FAULT_IRQN,FAULT_handler_simple1);
CyIntSetSysVector(CY_INT_MEM_MANAGE_IRQN,FAULT_handler_simple2);
CyIntSetSysVector(CY_INT_BUS_FAULT_IRQN,FAULT_handler_simple3);
CyIntSetSysVector(CY_INT_USAGE_FAULT_IRQN,FAULT_handler_simple4);
```

In order to test the vectors, we will create some testcases which hopefully will call the corresponding handlers.

[17] https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/fault-handling?lang=en

Note: The result of these tests has proven to be not 100% reproducable. Depending how the code is linked and how the memory is initialised, these tests may result in a hardfault or not.

For a first testcase, we will try an integer and a float division by 0.

Unfortunately, a simple function like

```
/*
 * Must be called with parameter 0
 */
int HFI_divideZero(int nonZero, int zero)
{
    return (nonZero/zero);
}
```

did not fool the compiler / controller. The (unexpected and wrong) result came as zero.

Same is true when using a float division. However, here the result is as expected because the IEEE standard defined for float operations has considered the division by 0 as a special case.

The next attempt is to call a non existing function and to execute some invalid opcode.

```
const static int InvalidOpcode = 0x12345678;

/*
 * Will call an invalid function through a function pointer
 */
void HFI_invalidFunction()
{
    typedef void (*fn_t)();
    fn_t foo = (fn_t)(&InvalidOpcode) ;
    foo() ;
}
```

Interestingly enough, when we place the code in an own function, no hardfault is injected. When we copy the same code into main, the hardfault will show up.

In the following screendump, one can see that the code jumped from FAUL_Test1() to the (invalid) flashdata of the variable InvalidOpcode and then ended in the installed hardfault handler.

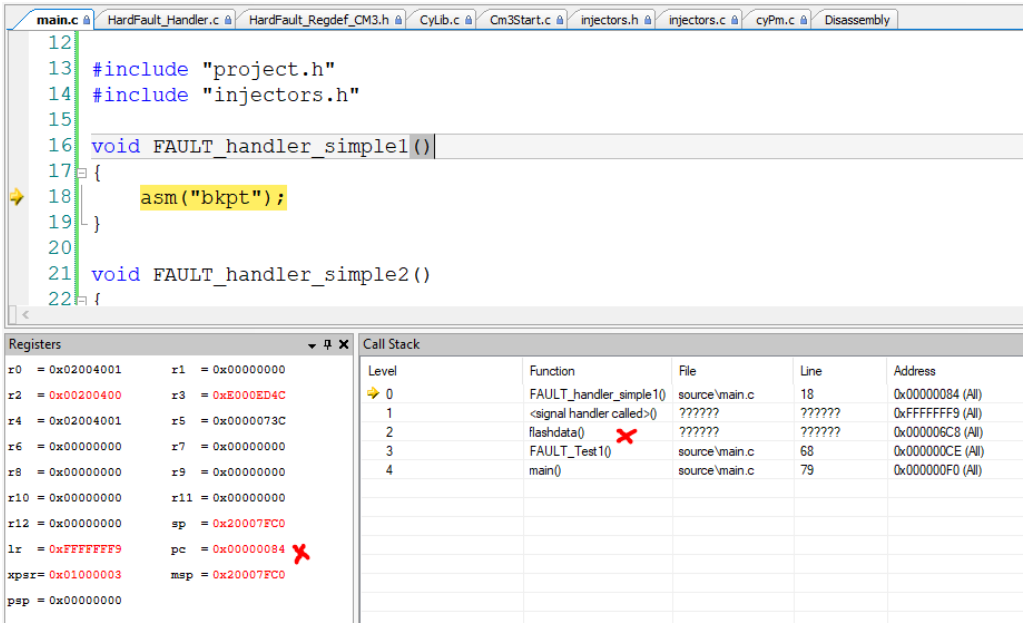


Figure 103 – Hardfault Handler Call

Try to create additional testcases causing hardfaults and check, which handler is being called.

9.5.2 Finding the root cause for a hardfault

The handler implementation so far only provided us a single piece of information: “Something went terribly wrong”. Unfortunately, we often cannot rely on debugger information any more at this point of time, because the controller hardware failed. The call stack might (or might not) still provide correct information and should be a first place to be checked.

In addition, most controllers provide post mortem registers, which are containing helpful information which have been stored at the moment of death. The job of the faulthandler will, to copy the content of these registers into a global data structure, which can be accessed by the debugger.

Which registers must be copied and wether the copy operation still can be done in C or (e.g. to avoid manipulation critical registers like stack or program counter) must be performed in assembly depends on the controller.

The following picture shows e.g. a data structure, which stores the address causing the problem – one of the most important pieces of information we need.

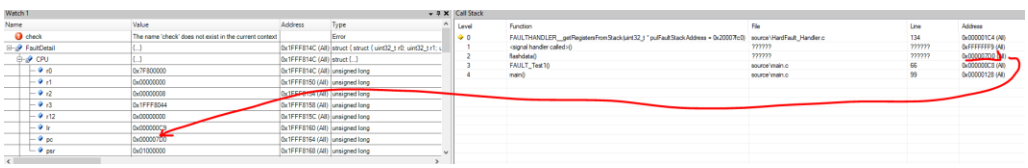


Figure 104 – Post Mortem Dump

Please note, that the address is shown in the non-corrupted callstack as well as in the data structure.

Other registers show possible root causes of the problem.

Watch 1

Name	Value	Address	Type
B	{...}	0x1FFF8172 (All)	struct {...}
UNDEFINSTR	0x00000000	0x1FFF8172 (All)	unsigned int
INVSTATE	0x00000000	0x1FFF8172 (All)	unsigned int
INVPC	0x00000000	0x1FFF8172 (All)	unsigned int
NOCP	0x00000000	0x1FFF8172 (All)	unsigned int
reserved_1	0x00000000	0x1FFF8172 (All)	unsigned int
UNALIGNED	0x00000001	0x1FFF8172 (All)	unsigned int
DIVBYZERO	0x00000000	0x1FFF8172 (All)	unsigned int
reserved_2	0x00000000	0x1FFF8172 (All)	unsigned int
HFSR	{...}	0x1FFF8174 (All)	union { unsigned int u32 : 32; struct { u
u32	0x40000000	0x1FFF8174 (All)	unsigned int

Figure 105 – Hardware Fault Root Cause

Please check the hardware manual of the controller for a detailed description of those register.

E.g. https://developer.arm.com/documentation/dui0552/a/the-cortex-m3-processor/fault-handling/fault-status-registers-and-fault-address-registers?lang=en

Here, you will also find additional information on the bug.

[8]	UNALIGNED	Unaligned access UsageFault: 0 = no unaligned access fault, or unaligned access trapping not enabled 1 = the processor has made an unaligned memory access. Enable trapping of unaligned accesses by setting the UNALIGN_TRP bit in the CCR to 1, see *Configuration and Control Register*. Unaligned `LDM`, `STM`, `LDRD`, and `STRD` instructions always fault irrespective of the setting of UNALIGN_TRP.

Figure 106 – Hardfault Documentation

It is a good idea to copy such information as a comment into your traphandler.

```
/** \brief [2] reserved*/
unsigned int reserved_1:1;

/** \brief [3] Enables unaligned access traps:
 *
 *  0 = do not trap unaligned halfword and word accesses
 *  1 = trap unaligned halfword and word accesses.
 * If this bit is set to 1, an unaligned access generates a UsageFault.
 * Unaligned LDM, STM, LDRD, and STRD instructions always fault irrespective of whether UNALIGN_TRP is set to 1.*/
unsigned int UNALIGN_TRP:1;

/** \brief [4] Enables faulting or halting when the processor executes an SDIV or UDIV instruction with a divisor of 0
 *
 *  0 = do not trap divide by 0
 *  1 = trap divide by 0.
 * When this bit is set to 0, a divide by zero returns a quotient of 0.*/
unsigned int DIV_0_TRP:1;
```

Figure 107 – Hardware Fault Comments

10 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_SWDCLK
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	SWDCLK
J3_05	P1.0	SWDIO

Figure 108 - PSOC Pin Header

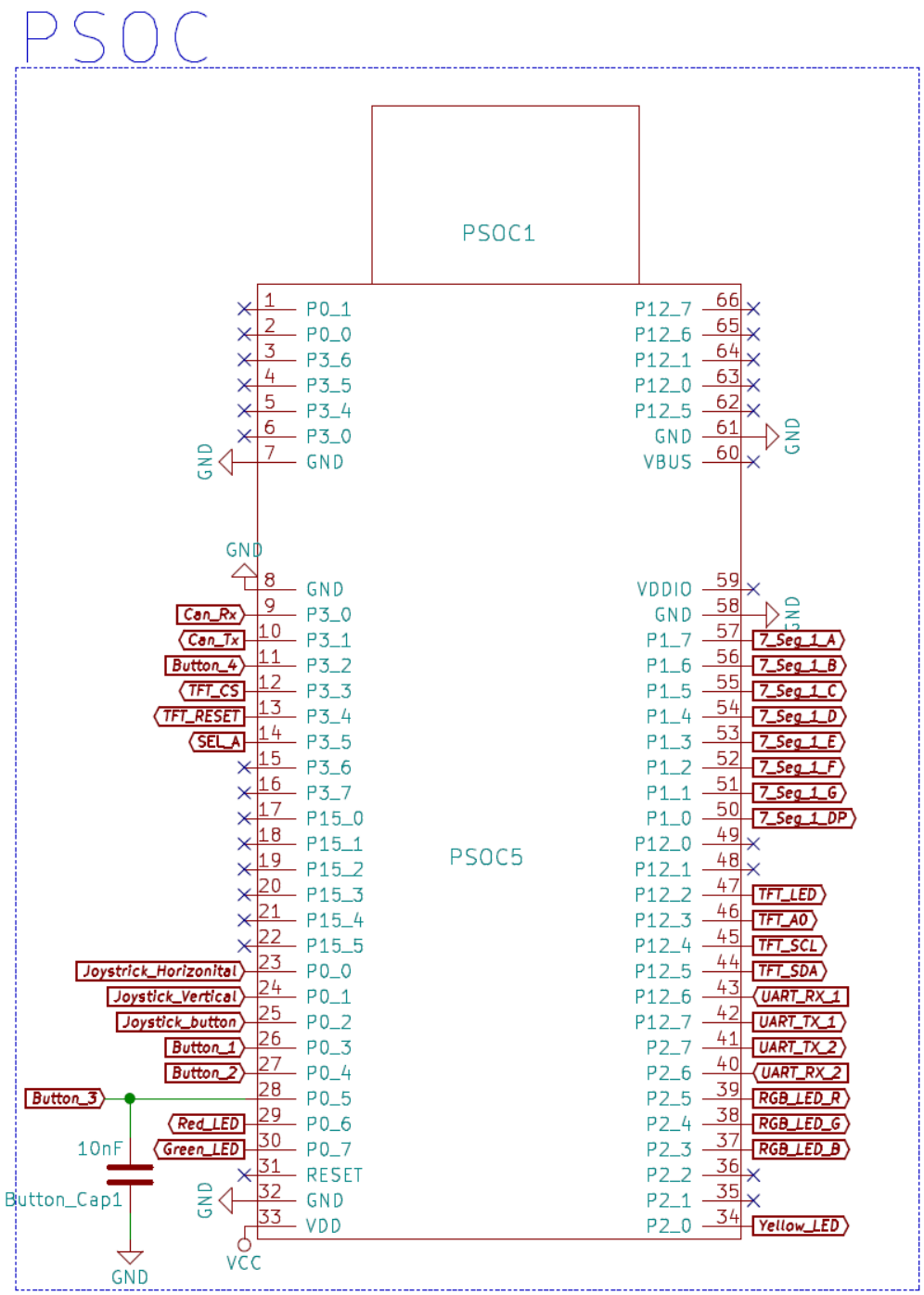


Figure 109 - LabBoard Connections

11 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 (availayble 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”

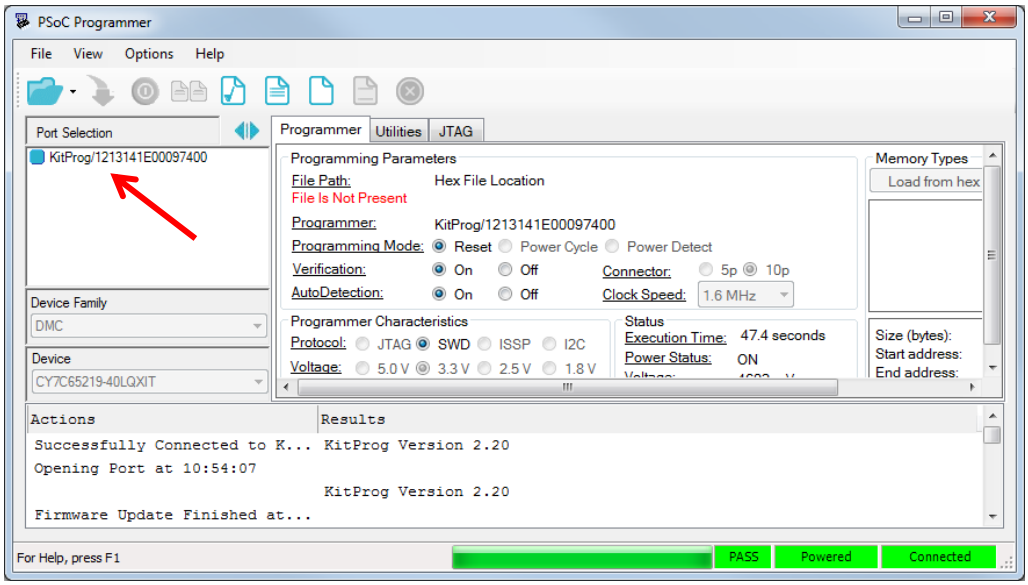


Figure 110 - Firmware Upgrade

12 Known Issues

The compiler creates an error message “unknown counter type”.	This happens when we create a counter, which is not used. Either connect the counter to an alarm or delete the counter.
The program hangs when using resources in an ISR2 context.	This seems to be a bug of the ERIKA generator. Try to deactivate the use resource flag in the ISR
OSEK hooks do not work	Also this seems to be an issue with the generator, which became visible when adding the tracealyser functionality. When you need hooks, please use a version prior to 2.5.1.
7 segment LEDs toggle when using the debugger	Design flaw of the Labboard. This is caused by multi functional pins. The pins used for debugging are also used for the 7 segment displays. When running the program without debugger it should be fine.

13 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.

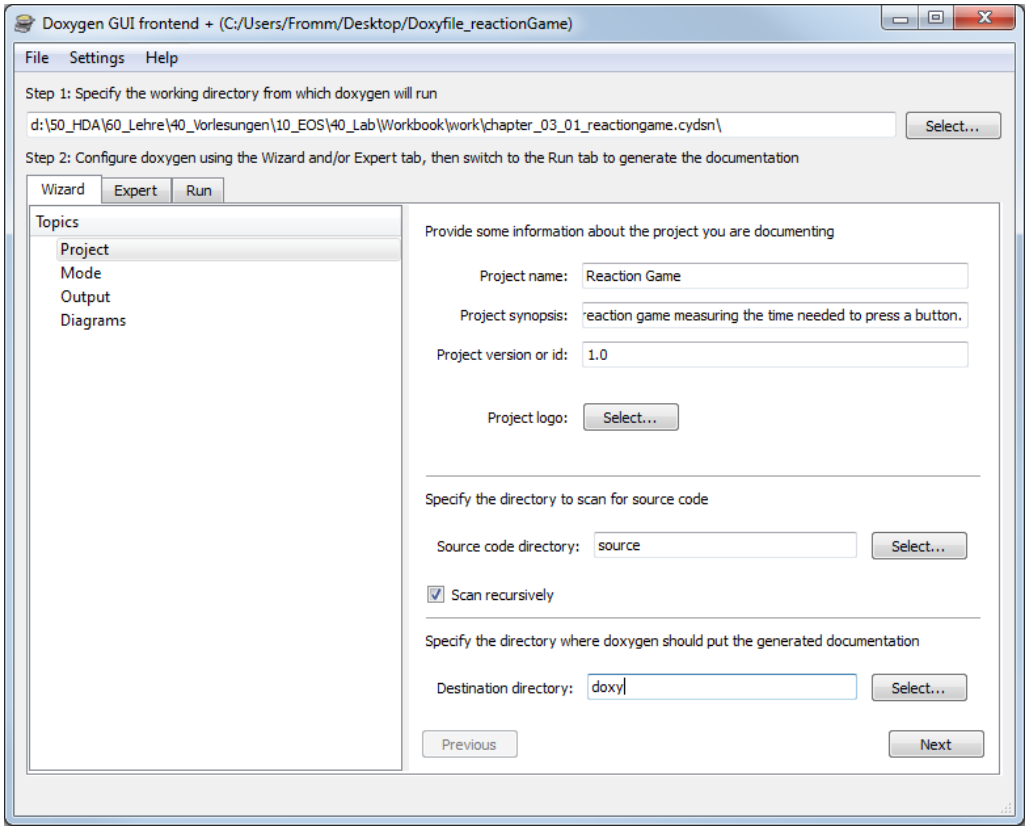


Figure 111 - Doxywizard Start Panel

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

The other fields a 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.

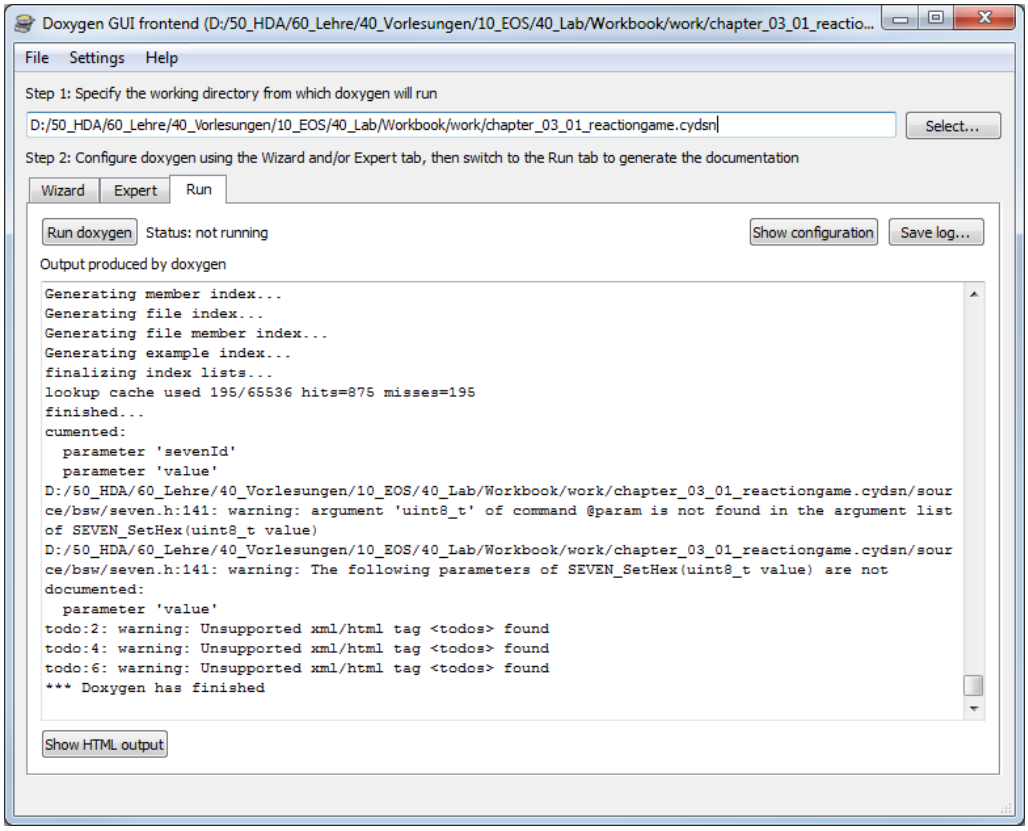


Figure 112 - Run Doxygen

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
```

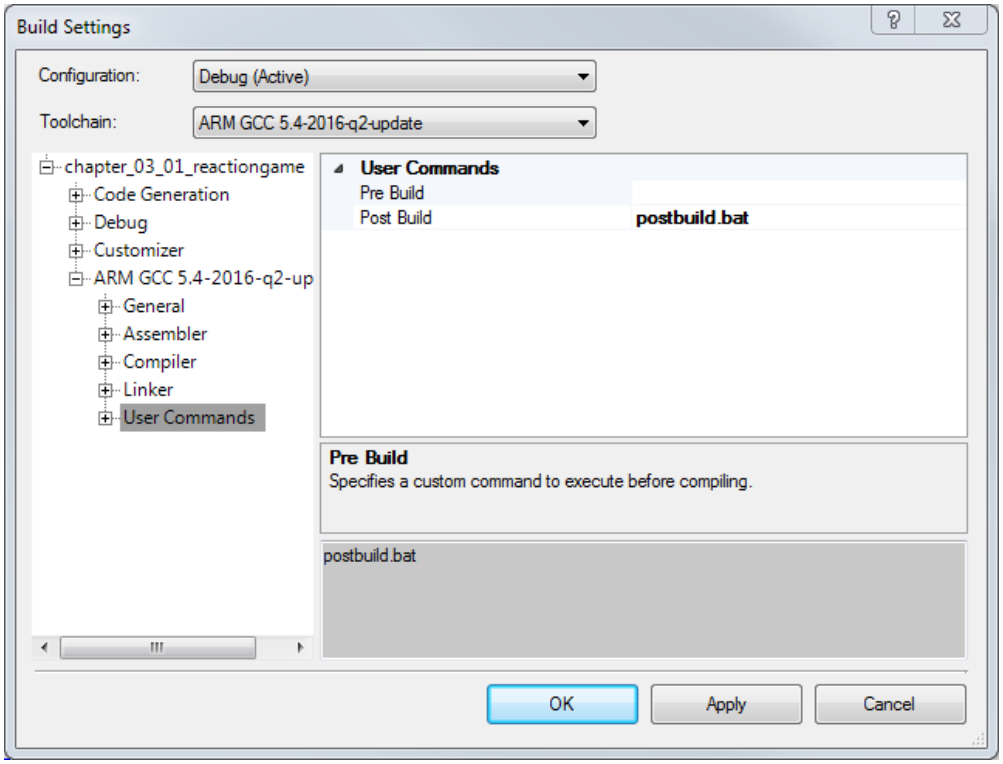


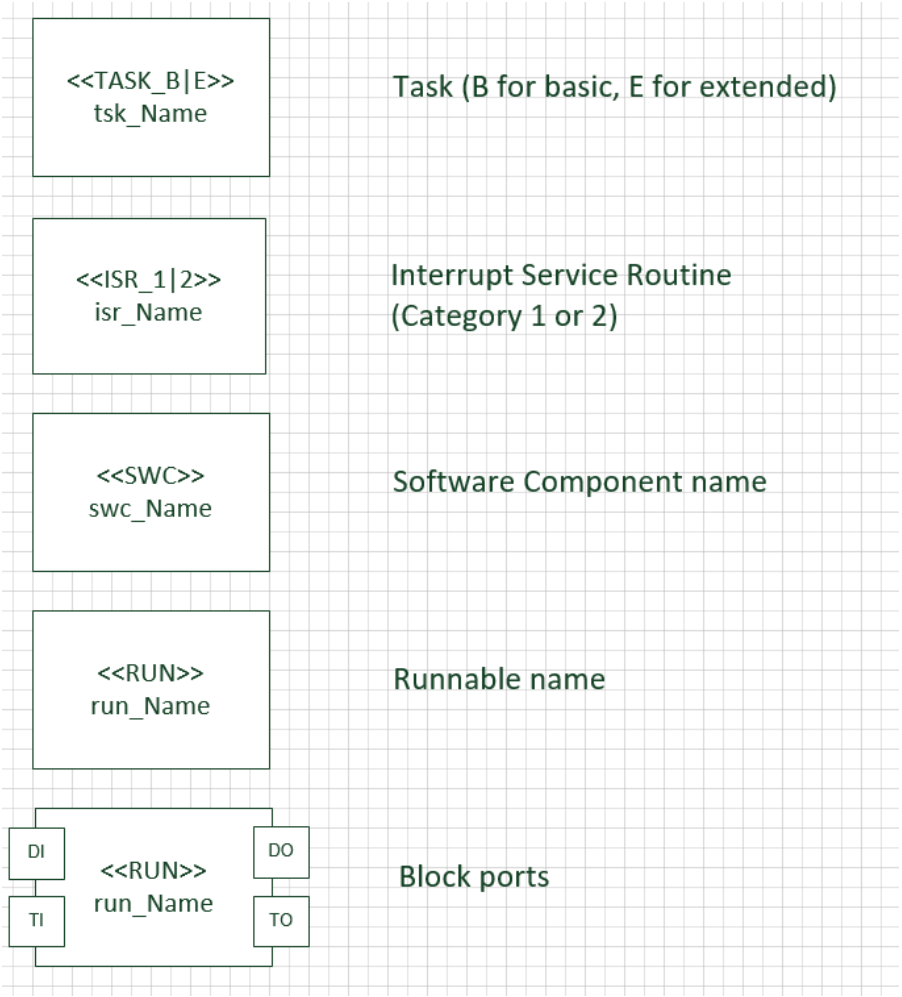
Figure 113 - Autogenerated documentation

14 Annex – Signal Flow Modelling Conventions

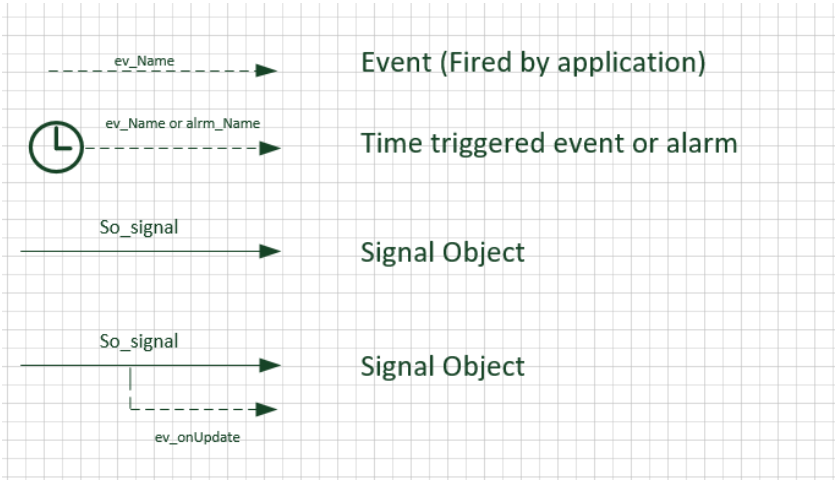
Blocks represent implementation units. The stereotype describes the type of the unit, e.g. task, isr, software component, runnable,…

Block communicate through ports. Trigger ports activate a runnable (e.g. cyclically or because a signal was updated or explicitly by firing a user event).

- DO – Data Out
- DI – Data In
- TO – Trigger Out
- TI – Trigger In



Signals are used to establish a communication between blocks. Normal signals transfer data, events are used to signal e.g. state changes. Signals and events can be combined using the ev_OnUpdate event. In this case, the receiving runnable will be triggered by an event in case the signal is updated.



In the following example, the isr_uart_rx receives data from a uart port and stores the data in the signal so_prot (e.g. a ringbuffer). Once the last byte of a protocol is received, the event ev_prot_avail is fired to the runnable run_parser. The runnable run-parser reads the complete protocol and translates it into a speed command, which is then passed on to the runnable run_engine. This runnably is activates every 100ms by a cyclic event.