

# Software for Embedded Systems Laboratory Course Description

© Prof. Dr. Volker Turau, Institute of Telematics



Institute of Telematics  
Hamburg University of Technology

**TUHH**  
Hamburg University of Technology



This document describes the lab of the master lecture *Software for Embedded Systems* from Technical University Hamburg. The goal of the lab is to convey techniques and methods taught in this lecture. The covered topics are employed in technical systems in various fields, such as automotive systems, production automation and control, and low power systems. The lab assumes a working knowledge with open-source software development tools such as the GNU compiler collection and the Eclipse IDE. In addition experience in the programming language C is expected. The lab teaches how to program embedded systems and their peripherals. It consist of six sections, each consisting of several tasks. All tasks are programmed for a board based on a 8-Bit microcontroller from ATMEL with many peripheral components. The hardware specification of the board is available on request.

This document is the joint effort of many of my former PhD students. I am thankful to: Christian Renner, Stefan Unterschütz, Andreas Weigel, Florian Kauer, Tobias Lübker and Florian Meyer. Special thanks to Jürgen Jessen for designing the board.

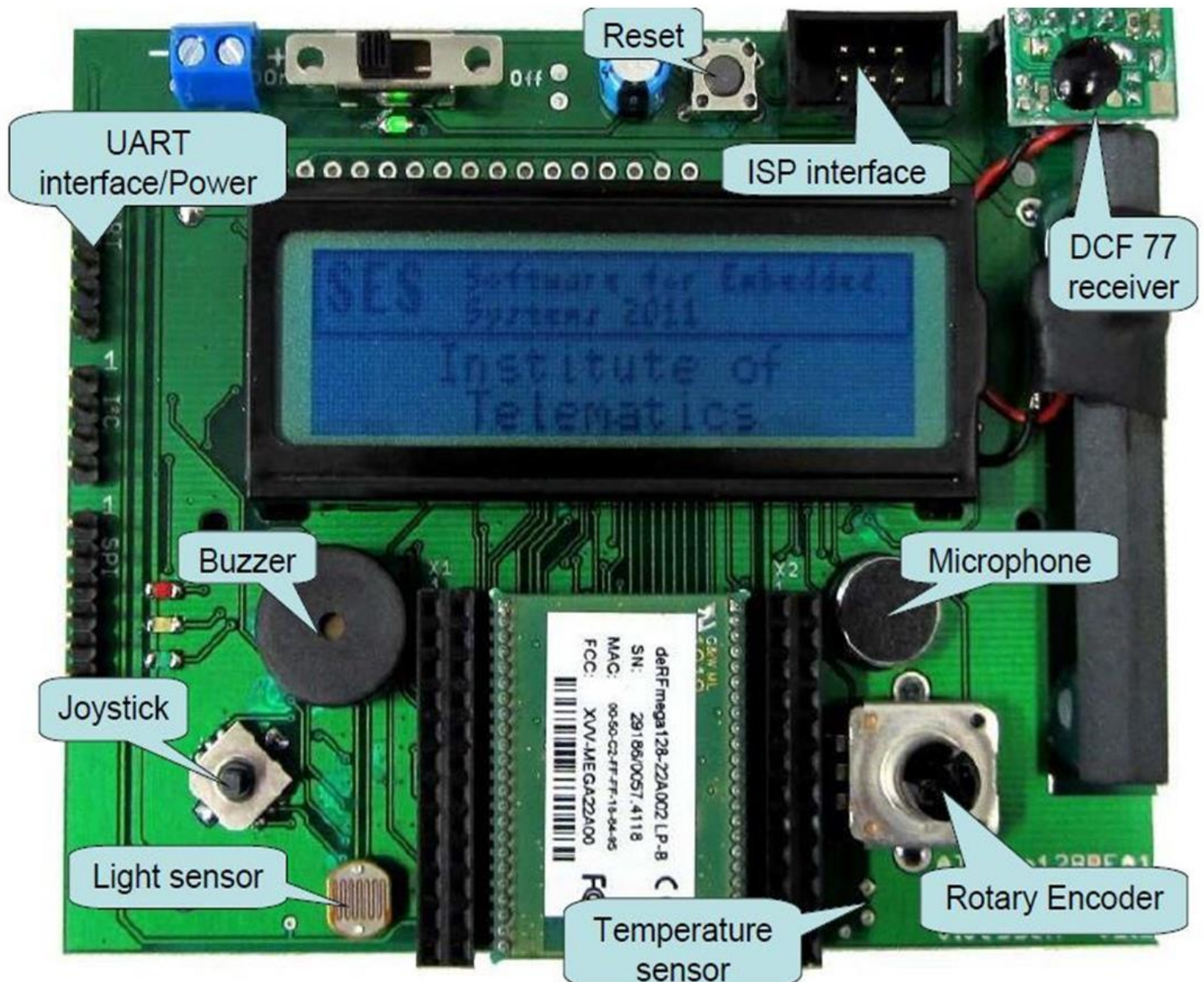
Copyright 2019 Prof. Dr. Volker Turau, Institute of Telematics  
Hamburg University of Technology  
turau@tuhh.de

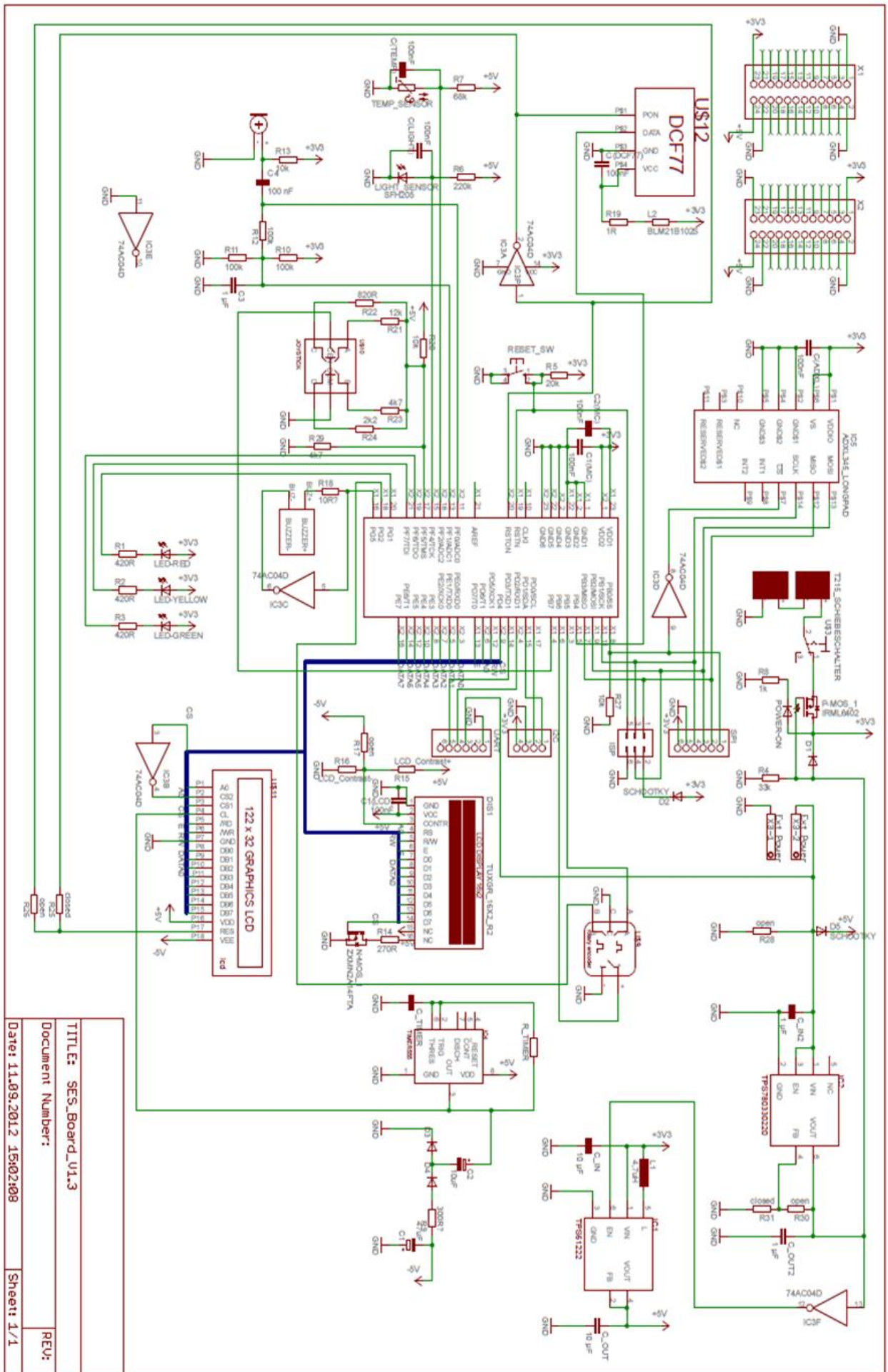


# Contents

1. Picture of the Hardware
2. Circuit Diagram of the Hardware
3. Exercise Sheet 0: Wanna Be Startin' Somethin'
4. Exercise Sheet 1: Tools & Tints
5. Exercise Sheet 2: Peripherals
6. Exercise Sheet 3: Interrupts and Timer
7. Exercise Sheet 4: Task Scheduler
8. Exercise Sheet 5: Motor Controller
9. Exercise Sheet 6: Alarm Clock

## SES Board





TITLE: SES_Board_V1.3	
Document Number:	
Date: 11.09.2012 15:02:08	Rev: 1
Sheet: 1/1	

# Wanna Be Startin' Somethin'

April 9<sup>th</sup>, 2019

**Please complete this exercise before the first lab!**

If you already have advanced knowledge of the C programming language, you may skip this exercise. In this (pre-)exercise you will set up a toolchain for compiling C applications for your PC. Moreover, in a small example we will show important language features which you have to understand before you should proceed to microcontroller programming.

Here are the main benefits of working with a native compiler (for your PC):

- It is a good starting point to learn C.
- You can easily test parts of the code, testing and debugging on the microcontroller is challenging.
- You can write entire libraries and port them to the AVR, e.g. an implementation of a list or an interpolation routine.

## Task 0.1 : C/C++ GNU Toolchain

If you are using Linux, you probably have a GNU C Compiler installed and thus you can skip this task. The following instruction is for the Windows programmers among you only.

First, you have to install a C toolchain (including compiler and libraries) to be able to program in C. In this lecture we use MinGW. MinGW, a contraction of "Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications.

Open <http://sourceforge.net/projects/mingw/files/> and click on *Installer*, *mingw-get-inst*, and *mingw-get-inst-\** (select latest version). Download and execute the file *mingw-get-inst-\*.exe*. Follow the installation, use default settings, don't use an installation path with whitespaces! As Compiler suite use the C Compiler (if you want you can also install C++). Finish the installation.

It is necessary to add the folder of MinGW's bin directory to the *Path* system variable of Windows. Go to *System Properties->Advanced->Environment Variables....* Add *MINGW\_ROOT\bin* (e.g. *C:\MinGW\bin*) to the Path system variable. Use ; as delimiter.

## Task 0.2 : Eclipse

(You can skip this section, if you have already installed Eclipse with CDT plugin.)

From the Eclipse homepage (<http://www.eclipse.org/downloads/>) download *Eclipse IDE for C/C++ Developers* for your operating system. Note that Eclipse requires an installed Java JRE. Afterwards you can extract (unzip) the downloaded file into a folder of your choice. Eclipse can be started by calling the file *eclipse*, which can be found in the main directory of the program.

## Task 0.3 : !!!Hello World!!!

Now let's finally start coding. Start Eclipse and select a workspace directory in which you want to store your projects. Press *File->new->Project...* and select *C/C++->C Project*. Use the template *Executable Hello World ANSI C Project* and select MinGW GCC or Linux GCC as toolchain (if this is not possible,

something went wrong during the installation). Open the source file (ending \*.c) that is located in the folder *src*. By pressing Ctrl + b the source code is compiled. If you make changes in the file do not forget to save it before you compile it. To run the program press the green play button in the toolbar and select Local C/C++ *application*. On the bottom the output of the execution is shown (!!!Hello World!!!).

Now, copy the content of the file *sheet0.c* into the existing source file (completely delete the existing code). Try to understand the program! You may also like to play around with it.

### Task 0.4 : A Bit of Masking and Shifting

Bit operations are performed in order to access various internal components and peripherals of the microcontroller. To get familiar with the use of them, solve the following tasks by writing to and reading from a variable `uint8_t var` with one C statement. You should do this by hand, afterwards you can check your solution by writing a small test program in C.

- Set bit 3
- Set bits 4 and 6 (with / without bit shifting)
- Clear bit 2
- Clear bits 2 and 7
- Toggle (invert) bit 3
- Set bit 2 and clear bits 5 and 7 at the same time
- Swap bits 3-5 and bits 0-2



For each operation, ensure that the remaining bits of `var` are not changed!



Note that the least significant bit (LSB) is bit 0 and most significant bit (MSB) is bit 7. Use the following bit operators:

& bitwise AND	bitwise OR	^ bitwise XOR
« left shift	» right shift	~ one's complement

### Task 0.5 : More Fun with Bits

Write a function that reverses the bit order of an 8 bit (use `uint8_t`) input! Given the input's bit representation  $\text{input} = (b_7b_6b_5b_4b_3b_2b_1b_0)$  the output becomes  $\text{output} = (b_0b_1b_2b_3b_4b_5b_6b_7)$ .

## Tools & Tints

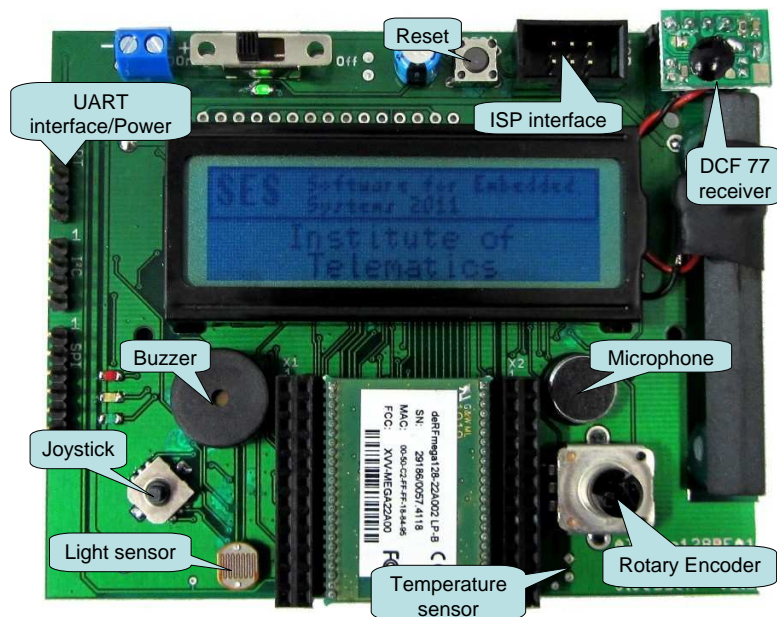
April 9<sup>th</sup>, 2019

⚡ **Successful participation in this exercise is required to continue the course. The interviews will take place the following Tuesday (2019-04-16).**

⚡ Please work on this sheet **on your own** using your own laptop. Work in teams of two will begin next week. Please prepare team suggestions for the interview.

In the following labs, we will put several components of the SES board into operation. In this lab however, we'll have an easy start with the LEDs. The SES board can be powered by 3 options: battery, external power supply or UART power supply. For ease of use we will use the UART power supply. The UART FTDI cable combines bidirectional serial communication and 5 V power supply. To power the board connect the 6-pin connector to the SES board, so that the black wire is connected to pin 1 of the UART pin header (upper pin on the left side of board). For flashing your program connect the ISP connector to the corresponding header of the board. The connector is polarity protected. It can remain connected, even while your program is running on the board.

⚡ Check polarity of the UART connector to avoid short circuits!



## Task 1.1 : Get the Party Started

Read the *Eclipse AVR Tutorial* (Stud.IP) and install the software needed for AVR programming. Execute the example given in the tutorial.

In that program, two global variables `PORTG` and `DDRG` are used. Explain their functionality!



Use the provided material *ATmega128RFA1 Datasheet.pdf*, which you can download from Stud.IP.



Make sure to have a working project with the code from the *Eclipse AVR Tutorial* prepared for the interview next week. During the interview we will ask you to toggle one of the three LEDs with a specific frequency. To find out how to toggle the different LEDs please have a look into the *SES Board Pinout* (Stud.IP)

## Task 1.2 : Looping Louie

On some occasions, e.g. the initialization of a display, delays have to be inserted in the program before performing the next operation. A very simple way of doing this is busy waiting, i.e. running a loop to burn processor cycles. You have already seen the function `_delay_ms`. In this exercise you have to make your own implementation of a waiting function. The function `void wait(uint16_t millis)` takes a 16 bit unsigned integer as the input parameter, corresponding to the delay of approximately `millis` milliseconds. You can test your function with the blinking program of the *Eclipse AVR Tutorial*.



For this task, assume a processor frequency of 16 MHz (16 million clock cycles per second). Furthermore an arbitrary busy waiting function is shown in the code snippet below. The challenge is to determine the required clock cycles for the C program. This can be achieved by compiling the source code whereupon a file with the ending `*.lss` is created. In this file the assembler code for each function is listed separately. A short description of the mnemonics and the required clock cycles can be found in the document *AVR Instruction Set Manual* which can be retrieved from Stud.IP.

```
#include <stdint.h>
#include <avr/io.h>

void shortDelay (void)
{
    uint16_t i; // 16 bit unsigned integer

    for (i = 0x0100; i > 0 ; i--) {
        //prevent code optimization by using inline assembler
        asm volatile ( "nop" ); // one cycle with no operation
    }
}
```

## Task 1.3 : Around the clock

To test the correctness of your delay loop implement a second LED program. After exactly 1 second toggle the LED. Check your program by synchronizing it with a watch and count the seconds for exactly 1 minute. If you have enough time left, you may also use the Logic Analyzer (which will be used in the next exercise anyway) to measure the timing more accurately. The tutorial can be found in the Stud.IP.

## Task 1.4 : Take a look at Atmel's ATmega128RFA1

In general, you should be aware of the resources and capabilities of your embedded system. For this purpose please try to find out more about the ATmega128RFA1 by reading the datasheet! Important parameters are:

- size of programmable flash
- size of internal EEPROM
- size of internal SRAM
- available ports
- available counters/timers
- peripherals (e.g., UART, ADC)

# Peripherals

## Building using Libraries for In- and Output

April 16<sup>th</sup>, 2019



The work on this sheet has to be finished within **two** lab sessions!

In this and the upcoming exercises, you will create several drivers and modules for the peripherals existent on the SES board. Those will be reused in later (also graded) exercises to create more and more complex systems. To enable reuse of all the basic functionality, you will put your drivers (starting with LED, Buttons and ADC in this exercise) into a library which will be referenced by upcoming projects as needed. Additionally, you will learn to include and use precompiled libraries for textual outputs using UART and LCD.



Even though this exercise is not graded, the created library is part of the later submission and the grading, so coding style is important!

### Task 2.1 : An SES library

As a first step, download and extract the *sheet2\_templates.zip* archive from the Stud.IP page. Create a new project named **ses**, as target choose "AVR cross target **static library**". Projects which use the library have to set their include path (to find the header files) and link the library created by this project. Copy all files from the *sheet2\_templates.zip* (*libuart.a*, *liblcd.a*, *ses\_button.h*, *ses\_common.h*, *ses\_uart.h*, *ses\_lcd.h*, *ses\_led.h*, and *ses\_led.c*) to your ses library project.

For the next lab sessions, you should upload your sources to the SVN. **The SVN will be set up on Wednesday after the group assignment, so it is not available during the first Tuesday session of this task!** Instructions can be found in the AVR Eclipse Tutorial. Please make sure that you only commit source and header files (NOT compiled object files, the library file or the whole Debug/Release folder). Also make sure that your driver library ends up at

<https://svn.ti5.tuhh.de/courses/ses/2019/teamX/ses> where X is your team's number.



**Keep the structure of your **ses** directory flat, i.e., do not use subdirectories there!**

### Task 2.2 : Setting up the Application Structure

Testing is an important part of the software development process. For this reason, you will write a very basic test application that will be filled with further functionality in the next tasks. Create a project for the this task to collaborate with your team member at

[https://svn.ti5.tuhh.de/courses/ses/2019/teamX/task\\_2](https://svn.ti5.tuhh.de/courses/ses/2019/teamX/task_2) where X is your team's number.

The new project has to know the location of the header files of your library and also needs to link to the compiled library. To achieve this with Eclipse, right click your new project and navigate to

**Properties->C/C++Build->Settings->Tool Settings;**

there, use AVR Compiler/Directories for the include paths and AVR Linker/Libraries for the linking:

- Under **AVR Compiler->Directories** click the add symbol, choose Workspace and select your library project's folder

- Under **AVR C Linker->Libraries** click the add symbol for **libraries** and type **ses**. Repeat the step for **uart** and **lcd**. Then click the add symbol for **library paths**, choose workspace and select the **ses** directory. Repeat the same for **ses/Debug**.

**Make sure that the settings are made for the configurations (Debug/Release) you currently use and remember this later if you change anything!** Additionally, to make code navigation and resolution of headers available, add your library project to the list of referenced projects (**Project Properties->Project References**) and to make sure changes in the library trigger a rebuild of projects using it, add the active configuration of your **ses** project to the referenced projects in **Project Properties->C/C++ General->Paths and Symbols->References** tab.



To avoid having to setup all this again and again, it is a good idea to create an empty template project with all those settings made and create a new project by copying and renaming this empty template project. Be careful to not copy a project which is under version control, because the versioning info will be copied, too, which is definitely not what you want!

### Task 2.3 : Start-up Messages

- In your new project, create a *main.c* file.
- Include *ses\_lcd.h* and *ses\_uart.h* in the *main.c*.
- Add the lines `uart_init(57600);` and `lcd_init();` to the initialization section of the main function, followed by an empty infinite while loop.
- To print a message to UART and LCD, use the statements `fprintf(uartout, "START\n");` and `fprintf(lcdout, "START");`, respectively. Don't forget a delay when using the outputs in the while loop.
- After this you can compile and flash to test the program. You can restart your program using the **Reset** button on the SES Development Board.

After compiling and flashing, the microcontroller sends the message via the UART to the PC. However, we need a program to receive the data. If not already done, you have to install the driver for the USB-FTDI cable. Have a look into the Tutorial for reference. Now, you can install a terminal program, e.g. *cutecom* for Ubuntu or *Termite* ([http://www.compuphase.com/software\\_termite.htm](http://www.compuphase.com/software_termite.htm)) for Windows. Start the terminal program, select a baudrate of 57600 and select the assigned COM port for the FTDI cable. All other initial settings can remain (8N1 no handshake). Try to connect to the port. If everything works you should see output messages when you press the reset button.






The UART and LCD libraries define FILE descriptors, which can be used together with `fprintf`. If you want to use `printf`, then add the line `stdout=uartout;` right after the initialization of the UART. Of course this only works for one of UART and LCD at the same time.

### Task 2.4 : Writing an LED Device Driver

In the Eclipse tutorial one LED was accessed using bit operations. Obviously, this is not very handy. Furthermore, this way of using LEDs is no good solution from a software-engineering point of view at all. As a result, we will develop a driver for using the LEDs. The driver consists of three parts: the header files *ses\_common.h*, *ses\_led.h* and the source file *ses\_led.c*. These files should be copied to your newly created **ses** library project.

The given header files already define all constants, e.g., the port and the LED pins. Furthermore, the header files contain all function declarations and the desired functionality of the functions. In this exercise you have to implement the given function prototypes! Empty functions are already present in `ses_led.c`.

-  Use the macro `DDR_REGISTER(x)` from `ses_common.h` to calculate the data direction register for a given port `x`. This macro helps to avoid redundancy.
-  The LEDs are *active low* (a logical *high* will turn them off)
-  For each operation ensure that all other bits remain unchanged!

## Task 2.5 : Buttons

The states of the two buttons on the SES board can be read as digital signals. The button of the rotary encoder is located on the lower right of the module and is connected to pin 6 of port B. The joystick button is located on the lower left and is connected to pin 7 of port B.



- In the library project, create a `ses_button.c` and provide the implementation of the functions declared in the `ses_button.h`:
- Create macros for the button wiring analogous to the ones in the `ses_led.c`.

```
void button_init(void)
```

- Configure the pin of each button as an input (write a `0` to the corresponding bit of the data direction register (`DDRB`))
- Activate the internal pull-up resistor for each of the buttons <sup>1</sup> (write a `1` to the corresponding bits of the buttons' port `PORTB`)

```
bool button_isJoystickPressed(void)
bool button_isRotaryPressed(void)
```

- When the button is pressed, the corresponding pin is grounded and the input can be read as logic *low* level via `PINB`

-  When setting or clearing bits in any register, be sure to leave all other bits unchanged!
-  You can use the `PIN_REGISTER(x)` and `DDR_REGISTER(x)` macro from `ses_common.h` to calculate the pin and data direction register for a given port `x`. These macros help to avoid redundancy.

## Task 2.6 : Superloop

Extend the `main.c` from the application project so that it conducts the following tasks:

- While the rotary button is pressed, light up the red LED
- While the joystick button is pressed, light up the green LED

<sup>1</sup> A pull-up resistor ensures a logic *high* level in the electric circuit, when a high-impedance device like an open switch or button is connected.

- Show the seconds since reset on the LCD



This subtask is mainly meant to illustrate the pain of superloops. Unless you provide a very clever implementation, the buttons will show bad responsiveness and/or the timings are not accurate. Do not bother yourself about this too much, you will learn about a much better approach in the next exercise.

## Analog to Digital Converter

The Analog-to-Digital Converter converts an analog input voltage to a digital value. The ATmega128RFA1 features an ADC with 10 bit resolution, yielding  $2^{10}$  distinct digital values which correspond to analog voltages<sup>2</sup>. The ADC is connected via a multiplexer to one of 8 external channels and a few internal channels. On the SES board, the following peripherals are connected to the ADC:

- A temperature sensor at pin 2 of port F
- A light sensor at pin 4 of port F
- The joystick at pin 5 of port F
- A microphone connected differentially to the pins 0 and 1 of port F<sup>3</sup>

All components output an analog voltage between 0 V and 1.6 V. More information is provided in the ATmega128RFA1 datasheet in chapter 27. *ADC - Analog to Digital Converter*.

## Task 2.7 : ADC (Initialization and reading)

Write an abstraction layer for the ADC of the board according to the following description!

The initialization of the ADC should be done in

```
void adc_init(void)
```

- Configure the data direction registers (temperature, light, microphone, joystick) and deactivate their internal pull-up resistors.
- Disable power reduction mode for the ADC module. This is done by clearing the PRADC bit in register PRR0.
- To define the reference voltage used, set the macro `ADC_VREF_SRC` in source file. The reference voltage should be set to 1.6 V.
- Use this `ADC_VREF_SRC` macro to configure the voltage reference in register `ADMUX`.
- Configure the `ADLAR` bit, which should set the ADC result right adjusted.
- The ADC operates on its own clock, which is derived from the CPU clock via a prescaler. It must be 2 MHz or less for full 10 bit resolution. Find an appropriate prescaler setting, which sets the operation within this range at fastest possible operation! Set the prescaler in the macro `ADC_PRESCALE` in the corresponding header file.
- Use the macro `ADC_PRESCALE` to configure the prescaler in register `ADCSRA`.
- Do not select auto triggering (`ADATE`).

<sup>2</sup>In this exercise, a reference voltage of 1.6 V is used, thus the ADC output corresponds to analog voltages from the interval  $[0 \text{ V}, 1.6 \text{ V} - \frac{1.6 \text{ V}}{2^{10}}]$

<sup>3</sup>The microphone is not used in this exercise, however

- Enable the ADC (ADEN).

The ADC should be ready for conversion now. The following function triggers the conversion:

```
uint16_t adc_read(uint8_t adc_channel)
```

`adc_channel` should contain the sensor type (enum `ADChannels`). By now the ADC is configured in a run once mode, which will be started for the configured channel in `ADMUX`. If an invalid channel is provided as parameter, `ADC_INVALID_CHANNEL` shall be returned.

A conversion can be started by setting `ADSC` in `ADCSRA`. The conversion will start in parallel to the program execution. When the conversion is finished, the microcontroller hardware will reset the `ADSC` bit to zero. So you can stop the program flow and use polling to check the `ADSC` bit.<sup>4</sup> The result is readable from the 16 bit register `ADC`, which is automatically combined from the 8 bit registers `ADCL` and `ADCH`.



If you use `ADCL` and `ADCH` to read out the ADC, make sure that you **always** read from `ADCH` after you have read from `ADCL`, otherwise the data register will be blocked and conversion results are lost.

## Task 2.8 : ADC Peripheral Abstractions

Usually the raw ADC value is not very useful, instead provide abstractions to calculate the corresponding physical measure for the joystick and the temperature sensor. For the light sensor, use the raw value. For testing purposes, write the results to the UART or the LCD.

```
uint8_t adc_getJoystickDirection(void)
```

The joystick on the SES Board is a 4-axis joystick with a button. The button press is evaluated as done before. The four directions are not connected to any digital channel, but to a single ADC channel (`ADC_JOYSTICK_CH`).

The RAW value of the ADC corresponds to the following directions:

- 200 ⇒ Right
- 400 ⇒ Up
- 600 ⇒ Left
- 800 ⇒ Down
- 1000 ⇒ No direction

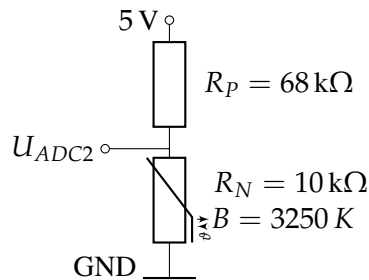


The RAW values are never exact, therefore choose an error tolerant mapping.

```
int16_t adc_getTemperature(void)
```

For the temperature sensor, the ADC values have to be converted to °C. The component used for temperature measurement is an NTC thermistor with a temperature dependent resistance  $R_T$ . It is connected to the ADC as given in the following circuit diagram.

<sup>4</sup>A better possibility would be to use interrupts to avoid stopping the program flow or to put the controller in noise reduction mode. The concept of interrupts will be introduced in the next exercise.



According to the voltage divider rule, the voltage  $U_{ADC2}$  at the ADC pin 2 is

$$U_{ADC2} = 5\text{ V} \cdot \frac{R_T}{R_T + R_P}.$$

Thereby, the current resistance of the thermistor can be calculated and used to get the current temperature in Kelvin as given by

$$T = \frac{B \cdot 298.15\text{ K}}{B + \ln\left(\frac{R_T}{R_N}\right) \cdot 298.15\text{ K}},$$

with  $B$  and  $R_N$  being characteristics of the thermistor as given in the schematic. Your task is to provide a function that converts the raw ADC value to the corresponding temperature. The unit of the return value `int16_t` shall be  $\frac{1}{10}^\circ\text{C}$ , i.e., a value of 10 represents  $1.0^\circ\text{C}$ .

However, since the computational power of the ATmega128RFA1 is limited and floating point operations are quite expensive, linear interpolation should be used instead of applying above formulas directly. For this, calculate the raw ADC values for two temperature values (e.g.  $40^\circ\text{C}$  and  $20^\circ\text{C}$ ) and use the following function to calculate interim values. A suitable `ADC_TEMP_FACTOR` is necessary to avoid rounding the slope to zero.

```
int16_t adc = adc_read(ADC_TEMP_CH);
int16_t slope = (ADC_TEMP_MAX - ADC_TEMP_MIN) / (ADC_TEMP_RAW_MAX - ADC_TEMP_RAW_MIN);
int16_t offset = ADC_TEMP_MAX - (ADC_TEMP_RAW_MAX * slope);
return (adc * slope + offset) / ADC_TEMP_FACTOR;
```

Extend your superloop with the following functionality:

- While the joystick is in the left position, light up the yellow LED
- Every 2500 milliseconds, read the values of the temperature and the light and show it on the LCD

## Task 2.9 : Logic Analyzer (optional)

In the previous task it was required to display new ADC values every 2.5 seconds. To achieve this when only using the `_delay_ms` functionality, the execution time of the ADC read and LCD display commands need to be incorporated. One approach to measure the execution time on an embedded device is to indicate start and stop of commands using output signals (e.g. LEDs) and then measure the time with an external device such as oscilloscopes or logic analyzers. Furthermore, these devices allow to assess unexpected behavior of input signals such as bouncing of buttons. In this lab simple logic analyzers are provided to analyze timings and behaviors of input signals.

Before using the device read the logic analyzer tutorial available in StudIP and setup the software!

Modify your code such that you indicate start and stop of the ADC and LCD operations (e.g. turn a LED on and off). Connect the logic analyzer to the corresponding output pin and do not forget to connect the ground. Start PulseView, select an adequate sampling configuration and measure the execution time of displaying the ADC values. Now you can update the execution delay in order to approximate a total execution interval of 2.5 seconds.

# Interrupts and Timer

April 30<sup>th</sup>, 2019



This exercise is not graded and has to be finished within two weeks! Even though this exercise is not graded, the created library is part of the later submission and the grading, so coding style is important! During the last exercise, you learned how to use buttons, the ADC, and delays to write simple applications. However, polling the status of peripherals and using blocking delay loops result in very unsatisfying timing as well as intertwined code. As a solution, the AVR microcontroller provides interrupts. That is, under certain conditions, such as pressing a button, the current program flow is stopped and a dedicated interrupt service routine (ISR) is executed before the normal program flow is continued.

In this exercise, you will learn how to use interrupts for buttons, and about a more elegant way for implementing timed actions. First, create a new project that has to end up at

[https://svn.ti5.tuhh.de/courses/ses/2019/teamX/task\\_3](https://svn.ti5.tuhh.de/courses/ses/2019/teamX/task_3) where X is your team's number.

The new project has to know the location of the header files of your library and also needs to link to the compiled library as described in the previous exercise. Furthermore, download and extract the *sheet3\_templates.zip* archive from the Stud.IP page to the project of your ses library.

## Task 3.1 : May I Interrupt you?

All buttons of the SES board trigger one pin change interrupt. To understand this functionality, please open the ATmega128RFA1 datasheet<sup>1</sup> and read section 16 carefully. Furthermore, C function pointers are used in this exercise. The use of function pointers was shown in the first exercise. If you need more information on function pointers, please ask Google, Bing, or any source of your personal trust.

Extend the header *ses\_buttons.h* with the following three lines

```
typedef void (*pButtonCallback)();  
void button_setRotaryButtonCallback(pButtonCallback callback);  
void button_setJoystickButtonCallback(pButtonCallback callback);
```

Open the library file *ses\_buttons.c*. Extend the `button_init` function that you wrote in the previous exercise with the following actions:

- To activate the pin change interrupt at all write a **1** to the corresponding pin in the `PCICR` register
- To enable triggering an interrupt if a button is pressed write a **1** to the corresponding position in the mask register `PCMSK0` (position is same as given in `BUTTON_ROTARY_PIN` and `BUTTON_JOYSTICK_PIN` definition)

Implement the interrupt service routine `ISR(PCINT0_vect)`:

- Check whether one of the button values changed
- Execute the appropriate button callback function
- Make sure that a button callback is only executed if a valid callback was set and the mask register contains a **1**

Now, implement the setter functions `button_setRotaryButtonCallback` and `button_setJoystickButtonCallback` and store the given function pointers in variables that can

<sup>1</sup><http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-8266-MCU-Wireless-ATmega128RFA1-Datasheet.pdf>

be accessed by the ISR. Document their usage in the header.

Having implemented our little button library, we actually want to test it. For this purpose, in the `task_3` project, write a program that first initializes the buttons and defines two functions for toggling the leds. Enable the buttons and pass the function pointers of the led toggle to the buttons.



When writing public libraries like the button driver, always consider that invalid parameters might be passed (such as null pointers)!



Don't forget to globally enable all interrupts by using the `sei()` function!



Do not forget the infinite *while* loop after the initialization!



Although the buttons are rather “good”, depending on the board you may observe mechanical bouncing effects. You may ignore these for this task.

### Task 3.2: Setting up a Hardware Timer

Instead of busy-waiting with the `_delay_ms` function, the timing should be done in parallel to other operations. The ATMEL ATmega128RFA1 has two 8-bit timers and four 16-bit timers. A timer is simply a special register in the microcontroller that can be incremented or decremented in hardware (e.g. by the clock signal) independently from other operations. The crucial benefit of timers is that they can trigger timer-interrupts once certain conditions are met. For instance, a timer can increment an 8-bit register and upon the transition from 255 to 0 an overflow interrupt flag is set. For that timer at 16 MHz 62500 interrupts are triggered per second resulting in a time of  $16 \mu\text{s}$  between two interrupts. The time can be extended by using prescalers, dividing the frequency of the clock signal by a fixed number. With a prescaler of e.g. 16 the time between two interrupts increases to  $256 \mu\text{s}$ . The time can be further extended by using a larger prescaler, using 16-bit timers or implementing a software counter based on the timer-interrupts. The set of prescalers is limited depending on the microcontroller and the timer used.

In this part of the exercise we will use the 8-bit timer2. It will be necessary to read parts of the ATmega128RFA1 datasheet chapter 21. Especially chapter 21.11 *Register Description* contains the most important information!

Open the files `ses_timer.h` and `ses_timer.c` and implement the given functions according to the following information:

- Use timer2
- Complete and use the macros given in `ses_timer.c`
- Use *Clear Timer on Compare Match (CTC)* mode operation (*chapter 21.5: Modes of Operation*)
- Select a prescaler of 64
- Set interrupt mask register for *Compare A*
- Clear the interrupt flag by setting an 1 in flag register for *Compare A*
- Set a value in register `OCR2A` in order to generate an interrupt every 1 ms



Do not use *magic numbers* in your code, instead use macros from the datasheet!

Implement the interrupt service routine. Only the callback function has to be invoked which was passed to `timer2_setCallback()`.

Now test your timer implementation by extending your `main()` function as well as providing a function `void softwareTimer(void)`, which will serve as callback. In `main()`, initialize the timer and pass the pointer to the function `softwareTimer()` to it. The function `softwareTimer()` should toggle the LED. Since the timer is fired each millisecond you will probably need some software counter to decrease the frequency.

### Task 3.3 : Button debouncing

In the first task, we directly triggered an interrupt on an edge in the signal caused by a button press. If you were lucky, your board had a very “good” button and you did not notice any bouncing effects. Normally, however, mechanical buttons exhibit some bouncing behavior (see Fig. 1) and directly triggering interrupts on undebounced buttons is considered bad practice. Thus, instead of using polling in a main loop or using direct interrupts which may lead to multiple button presses due to bouncing, we now steer middle ground by polling the button state within a timer interrupt.

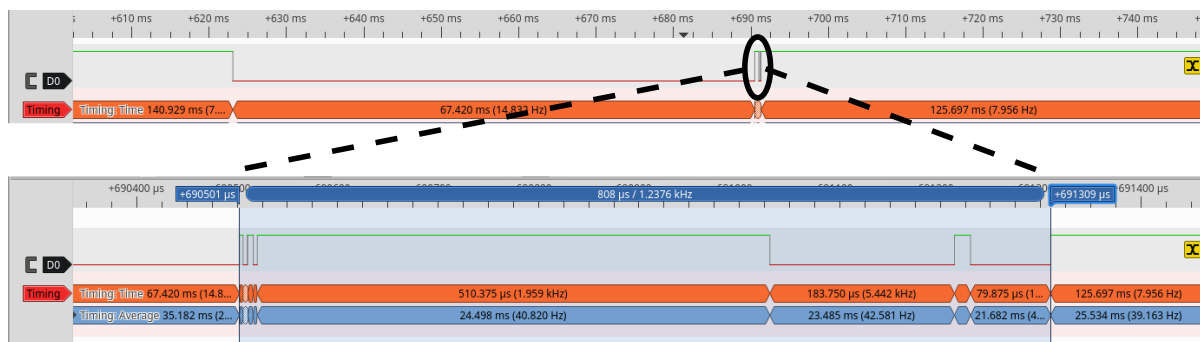


Figure 1: Bouncing of a button after press.

Implement the empty functions for timer 1:

- achieve an interval of 5 ms
- make sure to select the correct mode (value differs from timer 2)!
- use your simple test program to verify that the timer is configured correctly, too.

Now copy the following (incomplete) function to your `ses_button.c`:

```
void button_checkState() {
    static uint8_t state[BUTTON_NUM_DEBOUNCE_CHECKS] = {};
    static uint8_t index = 0;
    static uint8_t debouncedState = 0;
    uint8_t lastDebouncedState = debouncedState;

    // each bit in every state byte represents one button
    state[index] = 0;
    if(button_isJoystickPressed()) {
        state[index] |= 1;
    }
    if(button_isRotaryPressed()) {
        state[index] |= 2;
    }

    index++;
    if (index == BUTTON_NUM_DEBOUNCE_CHECKS) {
        index = 0;
    }

    // init compare value and compare with ALL reads, only if
    // we read BUTTON_NUM_DEBOUNCE_CHECKS consistent "1" in the state
    // array, the button at this position is considered pressed
    uint8_t j = 0xFF;
    for(uint8_t i = 0; i < BUTTON_NUM_DEBOUNCE_CHECKS; i++) {
        j = j & state[i];
    }
    debouncedState = j;

    // TODO extend function
}
```

Try to understand the purpose of the function and define the macro `BUTTON_NUM_DEBOUNCE_CHECKS` in your button driver. This function is meant to be called by the timer interrupt you just implemented, so it should be set as callback during the initialization. Extend the `button_init` function so that it takes a flag that specifies if the debouncing or the direct interrupt technique should be used:

```
void button_init(bool debouncing) {
    // TODO initialization for both techniques (e.g. setting up the DDR register)

    if(debouncing) {
        // TODO initialization for debouncing
        timer1_setCallback(button_checkState);
    }
    else {
        // TODO initialization for direct interrupts (e.g. setting up the PCICR register)
    }
}
```

Extend the `button_checkState` function, so that the corresponding button callback is called as soon as the `debouncedState` indicates a (debounced) button **press** (not release). Note, that though we are using it for two buttons only, this method could be used to debounce up to eight buttons in an efficient way.

Test your modified button driver with the main project from the first task!



A good value for `BUTTON_NUM_DEBOUNCE_CHECKS` and our buttons is 5, which introduces a delay of

up to 30 ms. This is fast enough to feel instantaneous for a human and provides reliable debouncing. Note, however, that individual buttons may exhibit different behavior. Some nice reading is “The Art of Designing Embedded Systems” by Jack Ganssle, where also the idea for the algorithm originates.

### Task 3.4 : The SEI Instruction (Challenge)



If you have no time left to finish this subtask during the current two week, please skip it to catch up in the following week.

The datasheet (page 17) states

When using the SEI instruction to enable interrupts, the instruction following SEI will be executed before any pending interrupts, [...].

Find out if this is an inherent property of the SEI instruction or if this is also valid for other ways of enabling the global interrupt enable bit.



Most C instructions result in more than one microcontroller instruction. For investigating such statements, better use (inline) assembly routines. Have a look at Chapter 34 of the datasheet. In the following, some useful routines are given.

The sei instruction in inline assembly:

```
asm volatile ("sei");
```

An alternative way of setting the SEI bit:

```
asm volatile ("in r16,0x3f \t\n\
               ori r16,128 \t\n\
               out 0x3f,r16");
```

Enabling the yellow LED in a single instruction (after initialization):

```
asm volatile ("cbi 0x11, 7");
```

What happens if the next instruction after SEI is CLI?

```
asm volatile ("cli");
```

# Task Scheduler

May 14<sup>th</sup>, 2019



**This is a graded exercise. For further information regarding the evaluation criteria read the document *GradedExercises.pdf* carefully. The submission deadline is 2019-05-24 23:59 (CEST). The interviews will be held on Tuesday after submission, on 2019-05-28.**

Your solution has to be committed to your team folder in the SVN repository by the specified time. We expect the following directories/files to be present in your root directory at <https://svn.ti5.tuhh.de/courses/ses/2019/teamX/> where X is your team's number.

- **ses** (dir; contains your current ses library's source/header files)
- **task\_4\_2** (dir; containing scheduler test code source/header files)
- **task\_4\_3** (dir; optional; if you decide to do the challenge)

Ignoring this directory structure will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present (you do NOT need to check in the compiled libraries!).



**Keep the structure of those directories flat, i.e., do not use subdirectories there!**

## Task 4.1 : Task Scheduler

Listing 1: `taskDescriptor` data structure

```
/**type of function pointer for tasks */
typedef void (*task_t)(void*);

/** Task data structure */
typedef struct taskDescriptor_s {
    task_t task;           ///< function pointer to call
    void * param;          ///< pointer, which is passed to task when executed
    uint16_t expire;       ///< time offset in ms, after which to call the task
    uint16_t period;       ///< period of the timer after firing; 0 means exec once
    uint8_t execute:1;     ///< for internal use
    uint8_t unused:7;      ///< unused
    struct taskDescriptor_s * next; ///< pointer to next taskDescriptor, internal use
} taskDescriptor;
```

So far you have implemented a hardware timer in the previous exercise sheet. If we needed more timers, we could use more hardware timers. But the number of hardware timers is limited to six for the ATmega128RFA1. Moreover, executing a time-consuming function inside an interrupt service routine blocks other interrupts. This should be avoided! To overcome the mentioned problems, a task scheduler<sup>1</sup> can be used. We need only one hardware timer for the scheduler, which allows the execution of tasks<sup>1</sup> after a given time period in a synchronous context (not in the interrupt service routine). In order to provide a synchronous execution of tasks, the scheduler periodically polls for executable tasks inside its `scheduler_run()` function.

Put the files `ses_scheduler.h`, `ses_scheduler.c` (from the provided ZIP file) into your ses library project. In this task, you have to implement the function skeletons. The function `scheduler_init` initializes the

<sup>1</sup>In this context, a task is a function, which terminates/returns after some time by itself

scheduler and timer2 (*ses\_timer.h*, *ses\_timer.c*). All tasks are represented by a data structure as shown in Listing 1.

Each task is described by a function pointer to the function to execute (`taskDescriptor.task`). A function is scheduled for execution after a fixed time period given by `taskDescriptor.expire`, which is also used by the scheduler as an individual counter, counting down the time (in milliseconds) till execution. Tasks can be scheduled for single execution (`taskDescriptor.period==0`) or periodic execution (`taskDescriptor.period>0`). “Single execution tasks” are removed after execution; periodic tasks remain in the scheduler and are repeated depending on their period. To schedule a task, those parameters have to be set, and a pointer to the `taskDescriptor` has to be provided to `scheduler_add()`. Additionally, the function to be executed takes a `void*` parameter to enable the passing of parameters to a task (`taskDescriptor.param`). Added tasks can later be removed from the scheduler with the function `scheduler_remove()`.

Within the scheduler, the scheduled tasks should be organized as a singly linked list, which is initially empty:

```
static taskDescriptor* taskList = NULL;
```

Clients of the task scheduler have to provide the memory to store the `taskDescriptor`. Thereby, the scheduler is not restricted to a fixed number of tasks.

The callback for the timer2 interrupt is used to update the scheduler every 1 ms by decreasing the expiry time of all tasks by 1 ms and mark expired tasks for execution. Additionally, the expiration time of periodic tasks should be reset here to the period.

The function `scheduler_run` executes the scheduler in a superloop (`while(1)`) and is usually called from the `main()` function. It executes the next task marked for execution (if any), resets its execute flag and/or removes it from the list, if it is non-periodic. Be careful to handle task execution, adding, removing and resetting correctly!

Note that any interrupt service routine that calls a scheduler function may interleave with the execution of a scheduler function from the main-loop, i.e., from a task. This can lead to an inconsistency of memory (a crash is possible!) if shared variables are accessed. Therefore, the access to the list of `taskDescriptors` has to be restricted by disabling the interrupts in critical sections. There are some macros defined in `util/atomic.h` of the AVR libraries which can be used, e.g.:

```
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    // this block is executed atomically
}
```



Read the comments in the source code carefully.



With the given structure, adding or removing the same `taskDescriptor` twice may lead to unwanted behavior. Make sure your code prevents this situation!

## Task 4.2: Using the Scheduler

In this exercise, you have to use your scheduler and implement a program which runs different tasks in parallel.

Create a new project with a file `scheduler_test.c` containing the `main` function. Then implement the following functionality by **adding** or **removing** tasks to the scheduler:

- Implement a task to toggle an LED and use the parameter to select the color, e.g. by an enum. Use

it to toggle the green led with a frequency of  $0.5\text{ s}^{-1}$ .

- Instead of using timer 1, use the scheduler for debouncing the buttons by calling `button_checkState` as task every 5 ms.
- When pressing the joystick button, turn on the yellow LED. Turn it off when pressing the joystick button again or after 5 seconds, whatever comes first.
- Implement a stop watch which starts and stops when pressing the rotary button. Use the LCD to show the current stop watch time in seconds and tenths of seconds. You do not need to implement a reset of the stop watch (you may use the reset button for this purpose).



Make use of the library created in previous exercises (LCD, LED, and Buttons).



Make sure, that the `taskDescriptors` you use are not local variables, which run out of scope after scheduling – this is a sure means of producing undefined behavior!



Do not use `_delay_ms()` (or **any** other busy waiting) for waiting multiple milliseconds!

### Task 4.3: Preemptive Multitasking (Challenge)

In this task you have to implement a preemptive multitasking execution model. The idea is that a number of predefined tasks are passed to the scheduler at the beginning. These tasks, which each have to run infinitely, should be scheduled using round-robin with a time slot length of 1 ms. At the beginning, you have to initialize a separate stack for each task and run the first task. Note that you can change the stack pointer of the AVR MCU. On a timer interrupt (each 1 ms), you have to store the context of the current task in its own stack and to restore the context of the next task.

The following example shows the intended usage of the preemptive multitasking scheduler. Provide the library required to run this program:

```
#include "pscheduler.h"

void taskA (void) {
    ...
    while (1) { ... }
}

void taskB (void) {
    ...
    while (1) { ... }
}

void taskC (void) {
    ...
    while (1) { ... }
}

task_t taskList[] = {taskA, taskB, taskC};
int main(void) {
    pscheduler_run(taskList,3);
    return 0;
}
```



The following hints may help you:

- Put the implementation of the preemptive scheduler into a separate project `task_4_3`, it won't be

re-used in coming lab classes.

- Make yourself familiar with preemptive multitasking, round-robin scheduling, context switches, and the way the MCU executes a program (stack, stack pointer, status register, etc.).
- The context switch has to be done using inline assembler, make yourself familiar with the concept of passing C variables to assembler programs.
- Read the file *Multitasking\_on\_an\_AVR.pdf* (StudIP), this actually contains everything you have to know. Also have a look into the lecture slides again, they also contain some useful hints.
- After compiling your code, open the *\*.lss* file which contains the assembler code. Use release mode for compiling since this makes the code more readable. Also check whether the compiler adds unintentional code. The *.lss* file is also a good place to identify potential problems, because the debugging possibilities are rather restricted.
- The source file *challenge.c* (in exercise ZIP file) might contain some useful code snippets.

# Motor Controller

May 28<sup>th</sup>, 2019

For this exercise, the SES boards will be equipped with a DC motor. Your task is to control and measure the speed of the motor.



**This is a graded exercise. For further information regarding the evaluation criteria read the document *gradedExercises.pdf* carefully. The submission deadline is 2019-06-14 17:00 (CEST). The interviews will be held on Tuesday after submission, on 2019-06-18.**

Your solution has to be committed to your team folder in the SVN repository by the specified time. We expect the following directories/files to be present in your root directory at <https://svn.ti5.tuhh.de/courses/ses/2019/teamX/> where X is your team's number.

- `ses` (contains your current ses library source/header files, including the new drivers)
- `task_5` (containing source/header files for the main project and optionally the challenge task)

Ignoring this directory structure will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present (you must NOT submit the compiled libraries!).



**Keep the structure of those directories flat, i.e., do not use subdirectories there!**

## Task 5.1 : PWM Speed Control

The current delivered to the motor can be controlled by a transistor whose basis is connected to `PORTG5`. This pin is also the `OC0B` pin, which can be changed by the timer 0 hardware, e.g. on compare match. Thus, a PWM signal with a fixed frequency, but adjustable duty cycle can be used to control the motor speed. In the SES library create a file `ses_pwm.h` with the following interface

```
void pwm_init(void);  
void pwm_setDutyCycle(uint8_t dutyCycle);
```

In the file `ses_pwm.c` implement the following

- Write a 0 to bit `PRTIM0` in `PRR0` to enable timer 0.
- Read Sect. 17.7.3 of the datasheet to understand the functionality of the fast PWM mode and select the respective mode in `TCCR0A` and `TCCR0B`.
- Disable the prescaler so that the timer is directly driven from the processor clock. Think about why a high prescaler is unsuitable.
- Configure the timer registers to set the `OC0B` pin when the counter reaches the value of `OCR0B`.
- Implement the `pwm_setDutyCycle` to set the value of `OCR0B`.

Test your code by creating a new project `task_5`.

- At initialization, the motor shall be stopped.
- After pressing the button, start the motor by configuring a `OCR0B` value of 170.
- A further press shall stop the motor again.



Verify that the resulting PWM output signal shows the desired behavior using the logic analyzer.

## Task 5.2: Frequency Measurement

Due to the three motor windings and the commutator, the motor current drops sharply 6 times per revolution<sup>1</sup>. This current signal is filtered in the circuitry and for every current drop, a spike is generated at the `PORTD0` pin that can be used as external interrupt `INT0`.

In the SES library create a file `ses_motorFrequency.h` to provide an interface to measure the revolutions of the motor in Hertz with the following functions.

```
void motorFrequency_init();
uint16_t motorFrequency_getRecent();
uint16_t motorFrequency_getMedian();
```

- Similar to the button interrupt pins, a signal edge at `INT0` can generate an interrupt. Read Chapter 16 and configure an interrupt for every rising edge and toggle the yellow LED.
- For calculating the frequency, timer 5 shall be used.
  - ◆ Implement an appropriate timer configuration that fulfills the requirements for the following functionality. Choose a correct prescaler to measure frequencies down to 10 Hz.
  - ◆ Use the interrupt service routine of `INT0` to measure the time required for one revolution.
  - ◆ Implement a functionality that allows to recognize a stopped motor, e.g. by using the CTC mode and the timer interrupt. While a stopped motor is recognized the green LED shall light up and the get functions shall return a frequency of 0 Hz.
- Implement `motorFrequency_getRecent()` to return the most recent measurement in Hertz.
- In the `task_5` project, use the scheduler to display the motor frequency in revolutions per minute (rpm) on the LCD every second.



The current value of the timer can be read out from `TCNT5`. The timer can be reset by `TCNT5 = 0`.

## Task 5.3: Median Calculation

As you might notice, the results of `motorFrequency_getRecent` are quite unstable and many erroneous measurements occur. Therefore, the measurements should be filtered with a median filter. Implement the function `motorFrequency_getMedian` that shall calculate the median of the last `N` interval measurements.

The interrupt service routine shall be used to store the last `N` interval measurements in a suitable data structure. When `motorFrequency_getMedian` is called, calculate the median over these measurements and return the inverse in Hertz. Show the result in rpm on the LCD together with the result of `motorFrequency_getRecent`.



Find a good `N` to allow for a stable, but responsive measurement.



Pay attention to prevent data inconsistencies if the external interrupt fires during the calculation. Though, avoid long atomic blocks, so wrapping the whole median calculation in a single atomic block has to be avoided.

<sup>1</sup>For details see <https://www.precisionmicrodrives.com/tech-blog/2011/06/08/using-dc-motor-commutation-spikes-to-measure-motor-speed-rpm>




**Task 5.4 : Challenge**

In task 5.2 you have used a fixed duty cycle to control the motor speed. However, due to production variances and different motor loads, this does not lead to a constant, predefined motor speed. In this challenge you shall implement a PID controller to maintain a given motor frequency  $f_{target}$  once the button is pressed, while measuring the current motor frequency  $f_{current}$ . As a first step, plot the trace of the frequency on the LCD to better see variations and oscillations. For this, the LCD library allows you to set single pixels of the LCD.

On a microcontroller a PID controller with anti-windup can be implemented by executing the following algorithm given as pseudo code in regular intervals.

$$\begin{aligned} e &:= f_{target} - f_{current} \\ e_{\Sigma} &:= \max(\min(e_{\Sigma} + e, A_w), -A_w) \\ u &:= K_p \cdot e + K_I \cdot e_{\Sigma} + K_D \cdot (e_{last} - e) \\ e_{last} &:= e \end{aligned}$$

The result  $u$  can be used to determine the duty cycle. Find appropriate values for  $K_p$ ,  $K_I$ ,  $K_D$  and  $A_w$  and initialize the controller properly.

-  The motor is responding very fast, so it is advised to start with a pure I controller ( $K_p = K_D = 0$ ) until a good, but slow control is maintained. Then adapt the values to get a faster response.
-  Do not use floating point operations. Thus, you have to scale the calculations.
-  Use the UART for debugging the course of the variables.

# Alarm Clock

June 18<sup>th</sup>, 2019



**This is a graded exercise. For further information regarding the evaluation criteria read the document *gradedExercises.pdf* carefully. The submission deadline is 2019-06-28 17:00 (CEST). The interviews will be held on Tuesday after submission, on 2019-07-02.**

Your solution has to be committed to your team folder in the SVN repository by the specified time. We expect the following directories/files to be present in your root directory at <https://svn.ti5.tuhh.de/courses/ses/2019/teamX/> where X is your team's number.

- **ses** (contains your current ses library source/header files, including updated scheduler)
- **task\_6\_3** (containing source/header files for task 6.3/6.4 (alarmClock))
- **task\_6\_5** (containing the source/header files for task 6.5 (bootloader))
- **fsm.<ext>** (a file containing a state chart of your alarm clock's state machine (see Task 6.2))

Ignoring this directory structure will lead to point deductions. Make sure that all **header** and **source** files necessary to build and link your solution are present (you do NOT need to check in the libraries!).



Keep the structure of those directories flat, i.e., do not use subdirectories there!

## Task 6.1 : Creating a Clock

In this exercise you have to implement a simplistic alarm clock.

The first task is to create a 24-hour Clock with a precision of 1 millisecond. For this task you have to extend your scheduler (*ses\_scheduler.c*, *ses\_scheduler.h*). Since the scheduler already uses a timer callback with a precision of 1 millisecond you can extend its ISR. Define a private 32-bit variable for storing the current time and increment it every millisecond.



For orientation, calculate the time until the timer overflows. However, you do not have to handle this event.

For reading or writing this value you have to add these two functions:

```
typedef uint32_t systemTime_t;
systemTime_t scheduler_getTime();
void scheduler_setTime(systemTime_t time);
```

For easier handling in your alarm clock, you should write some wrapper functions calculating the time in a more human-friendly way, using the structure `time_t`:

```
struct time_t {
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
    uint16_t milli;
};
```

## Task 6.2 : Finite-State Machine Model

The main task of this exercise sheet is the implementation of a simple but functional alarm clock. Before implementing the code, draw a **UML statechart** matching the description in the following paragraph. This drawing is part of the graded exercise and has to be present as digital file in your submission directory (.png, .pdf, .jpeg; you may also take a photo of a hand-drawn image as long as we can clearly read everything).

The following paragraph gives a description of the alarm clock in natural language:

When the microcontroller is powered up, the actual time is not known. The user has to set the time manually using the buttons. At this stage, the display shows an uninitialized clock using the format HH:MM. The second display line may show a request for the user to set time. First, the hour has to be set by repeatedly pressing the *Rotary Button*. After pressing the *Joystick Button*, the minutes have to be set via the *Rotary Button*. Pressing the *Joystick Button* again updates the system time and starts the clock. In this normal operation mode the time is shown in the format HH:MM:SS. In this state, the user can press the *Rotary Button* to enable the alarm or to disable the alarm. If the *Joystick Button* is pressed, the alarm time can be set. In this mode line 1 shows the alarm time instead of the current system time using the format HH:MM. If the alarm is enabled and the actual time matches the alarm time, the red LED shall toggle with 4 Hz. The alarm shall stop, if any button is pressed or 5 seconds have passed. The alarm must only be triggered if the clock is in its normal operating mode, i.e., not if the alarm time is being modified.

Additionally the LEDs are used for the following functionality:

- The green LED blinks synchronously with the counter of the seconds.
- The yellow LED is on, when the alarm is enabled.
- The red LED is flashing with 4 Hz during alarm.



The next chapter provides some instructions on how the more detailed aspects of the state machine shall be handled. You may want to read it first.

## Task 6.3 : Implementing the Alarm Clock

Implement the described alarm clock using the state machine designed in Task 7.2! Implement the FSM as a wholly **event-based** (not synchronous) one!

The Finite State Machine shall be implemented with pointers to functions. A state is represented by a pointer to an event handler (i.e., a function), which takes a pointer to the FSM itself and the event that occurred.

```
typedef struct fsm_s Fsm;    //< typedef for alarm clock state machine
typedef struct event_s Event; //< event type for alarm clock fsm

/** return values */
enum {
    RET_HANDLED,    //< event was handled
    RET_IGNORED,    //< event was ignored; not used in this implementation
    RET_TRANSITION  //< event was handled and a state transition occurred
};
typedef uint8_t fsmReturnStatus; //< typedef to be used with above enum

/** typedef for state event handler functions */
typedef fsmReturnStatus (*State)(Fsm *, const Event*);
```

The return value is used by the FSM “core” to decide about entry and exit actions, but more on that later. We define the struct `Fsm` with additional attributes besides (`state`). The member `isAlarmEnabled` can be used to store the actual alarm state (on/off) and `timeSet` to store the alarm-time or setup-time before normal clock operation.

```
struct fsm_s {
    State state;           //< current state, pointer to event handler
    bool isAlarmEnabled;   //< flag for the alarm status
    struct time_t timeSet; //< multi-purpose var for system time and alarm time
};
```

The `Event` struct contains possible events that may trigger state changes. Using `signal`, the type of the event can be determined by the event handler (e.g., *Joystick Button* pressed).

```
typedef struct event_s {
    uint8_t signal; //< identifies the type of event
} Event;
```



It is a probably a good idea to list the possible events within an enum.

For this exercise, we use the functions presented in the lecture, but extend `fsm_dispatch` to provide entry and exit actions as well, because these are useful for the alarm clock. Here, you can also see how the return status of the event handler is used to dispatch entry and exit actions.

```
/* dispatches events to state machine, called in application*/
inline static void fsm_dispatch(Fsm* fsm, const Event* event) {
    static Event entryEvent = {.signal = ENTRY};
    static Event exitEvent = {.signal = EXIT};
    State s = fsm->state;
    fsmReturnStatus r = fsm->state(fsm, event);
    if (r==RET_TRANSITION) {
        s(fsm, &exitEvent); //< call exit action of last state
        fsm->state(fsm, &entryEvent); //< call entry action of new state
    }
}
/* sets and calls initial state of state machine */
inline static void fsm_init(Fsm* fsm, State init) {
    //... other initialization
    Event entryEvent = {.signal = ENTRY};
    fsm->state = init;
    fsm->state(fsm, &entryEvent);
}
```

These functions can be used to control the state machine, e.g., to dispatch an event when a button was pressed. The following code snippet illustrates the intended usage:

```
#define TRANSITION(newState) (fsm->state = newState, RET_TRANSITION)

static void joystickPressedDispatch(void * param) {
    Event e = {.signal = JOYSTICK_PRESSED};
    fsm_dispatch(&theFsm, &e);
}

fsmReturnStatus running(Fsm * fsm, const Event* event) {
    switch(event->signal) {
        //... handling of other events
        case JOYSTICK_PRESSED:
```

```

        return TRANSITION(setHourAlarm);
    default:
        return RET_IGNORED;
    }
}

```

The shown TRANSITION macro can be used exclusively to change states to make sure that state changes are always the last operations within an event handler.



Be aware that operations regarding the state machine need a “run-to-completion” semantic, i.e., must not be interrupted by another operation on the FSM. One way to accomplish this, is to run the FSM completely within task context. Another would be to use atomic blocks, however, considering that the alarm clock involves several LCD-related operations, this approach should be discarded.

With the given information, it should be possible to implement the alarm clock in a very concise and elegant way. Below you will find some additional information (and, of course, the challenge task).

The shown code snippets can easily be extended to a more generic event processor, which could be used to implement arbitrary state machines. It is inspired heavily by the material presented in “Practical UML Statecharts - Event-Driven Programming for Embedded Systems” by Miro Samek. If you are interested, you can find much more information there! Note also that usually such an event processor would use an event-queue to hold incoming events. With our recommendation to execute the FSM in task context, we instead use the scheduler to queue the events for our alarm clock which is probably not the most clean, but – within the scope of the exercise – a very practical and functional solution.

## Task 6.4 : Rotary Encoder

To improve the usability of setting the time, the rotary encoder shall be used instead of the push button. The rotary encoder has two inputs A (connected to **PORTB 5**) and B (**PORTG 2**), which can be *high* or *low* (open/closed switch to GND=C). The encoding is given in the figure below, also showing the mechanical detent positions. Implement the functionality to detect a left or right turn of the incremental rotary encoder within a new library module `ses_rotary`. Use callbacks to increment or decrement the current hour or minute. Consider the given interface.

```

typedef void (*pTypeRotaryCallback)();
void rotary_init();
void rotary_setClockwiseCallback(pTypeRotaryCallback);
void rotary_setCounterClockwiseCallback(pTypeRotaryCallback);

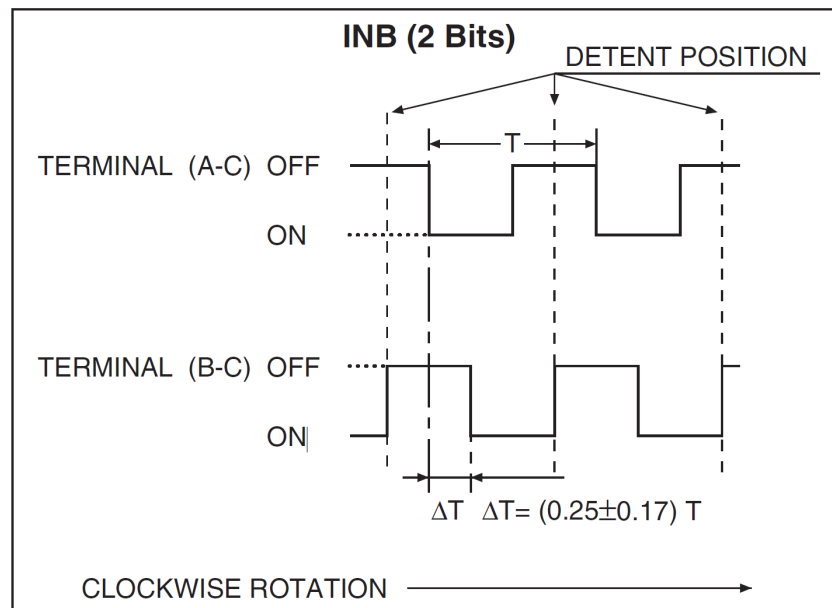
```



To get better understanding of the rotary button, plot the samples of both inputs on the LCD or use the logic analyzer. You may use the given function below, which plots the current samples once the rotary is turned until the display is full. Do not forget to initialize the inputs!



The rotary requires careful debouncing. Use a sampling approach similar to the button debouncing. Yet, occasionally skipping dents at fast turning is acceptable.



```

/*
 * Callback-procedure to plot the samples of the rotary pins on the LCD after first change
 */
void check_rotary() {
    static uint8_t p = 0;
    static bool sampling = false;

    bool a = PIN_REGISTER(A_ROTARY_PORT) & (1 << A_ROTARY_PIN);
    bool b = PIN_REGISTER(B_ROTARY_PORT) & (1 << B_ROTARY_PIN);

    if (a != b)
        sampling = true;

    if (sampling && p < 122) {
        lcd_setPixel( (a ? 0 : 1, p, true );
        lcd_setPixel( (b ? 4 : 5, p, true );
        p++;
    }
}

```

### Task 6.5: UART Bootloader (Challenge)

Until now, you have been using the Olimex programmer to flash the microcontroller. In this task you have to use the UART interface and a bootloader program to do the same. Once the bootloader is flashed into the microcontroller, it can be used to load new user code into the microcontroller via UART with a PC running avrdude. To be able to activate the bootloader only on demand, the joystick button shall be used (keep the joystick button pressed during reset to enter bootloader mode).

To communicate with avrdude, you have to implement the protocols for given in the [AVR109:Self Programming](#) application note (also available on StudIP). The bootloader must implement the AVRProg commands with ID = (T, A, g, B, e, b, t, P, L, E, S, s, V, a) given in the application note. All remaining commands and invalid commands return a question mark. Use 'Release' mode in Eclipse for building the project. Set the following fuse bits (AVRDude->Fuses):

- Boot Reset vector Enabled
- Boot Flash size=4096 words

Use the UART library provided with the Exercise 4 (initialise UART with baudrate of 57600). A joystick button press at startup shall be displayed by the red LED. The bootloader program is placed inside the boot section of the flash memory by **Settings-> AVR C Linker -> General -> Other Arguments** to `-Wl, -section-start=.text=0x1E000`.

To flash a new project to the microcontroller with the new bootloader, change the programmer settings to Atmel Butterfly Development Board, serial port of the FTDI cable, and baudrate of 57600. Reset the microcontroller and press the joystick button to start flashing.



Make sure that interrupts are disabled in the bootloader.



<avr/boot.h>, <avr/pgmspace.h> contains the required functions for the bootloader.