Peripherals

Building and using Libraries for In- and Output

May 5th, 2020

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

In this and the upcoming exercises, you will create several drivers and modules for the peripherals of 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 later submissions and the grading, so coding style is important!

Task 2.1 : A SES library

As a first step, download and extract 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*) archive from the Stud.IP page and copy the files to folder *./lib/ses/* in your workspace (please use the prepared workspace from your Git repository as described in Task 1.1).

For the next lab sessions, you should upload your sources to your Git repository. Instructions can be found in the Git Tutorial. Please make sure that you only commit source and header files.

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

Task 2.2 : Add a new Project to your Workspace

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. The applications should be located in own projects. You are free to decide to use one folder for this sheet (*task_2*) or one folder per task, e. g., *task_2-3* and *task_2-6*.

The `main`-function of each project must be in the file *main.c* in the folder *src*.

To create a new project you have to follow these steps:

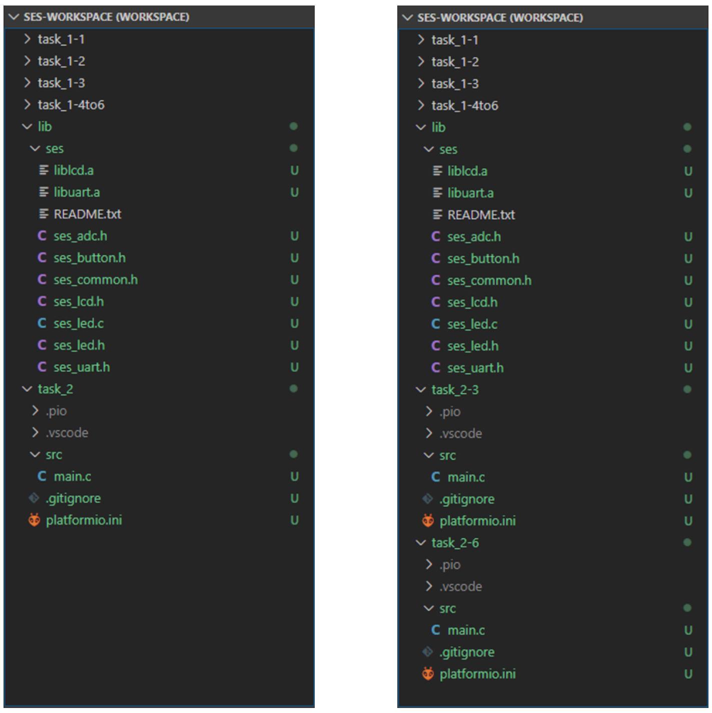
- Create a new folder in your workspace, e. g., *task_2*.
- Copy the file *platformio.ini* from a previous project to this folder.
- Add the folder *src* including the file *main.c* to the folder from the first step.
- Open Visual Studio Code, and add the folder from the first step to your workspace (`File -> Add Folder to Workspace..`).
- Reload Visual Studio Code. Afterwards, the folders *.pio* and *.vscode* should appear.

Figure 1 depicts two possible workspace structures.

The template folder (*sheet2_templates.zip*) contains two precompiled libraries (*libuart.a*, *liblcd.a*). To link these libraries extent the *platformio.ini* file to

```
[env:ses_avr]
platform = atmelavr
board_mcu = atmega128rfa1
board_f_cpu = 16000000L

lib_extra_dirs = ../lib
build_flags =
    -L../lib/ses/
    -l uart
    -l lcd
```



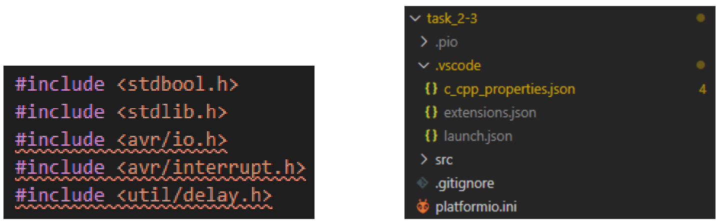
```
SES-WORKSPACE (WORKSPACE)
  > task_1-1
  > task_1-2
  > task_1-3
  > task_1-4to6
  v lib
    v ses
        liblcd.a          U
        libuart.a         U
        README.txt
        ses_adc.h         U
        ses_button.h      U
        ses_common.h      U
        ses_lcd.h         U
        ses_led.c         U
        ses_led.h         U
        ses_uart.h        U
  v task_2
    > .pio
    > .vscode
    v src
        main.c            U
      .gitignore          U
      platformio.ini      U
```

(a) Workspace with one project per sheet

```
SES-WORKSPACE (WORKSPACE)
  > task_1-1
  > task_1-2
  > task_1-3
  > task_1-4to6
  v lib
    v ses
        liblcd.a          U
        libuart.a         U
        README.txt
        ses_adc.h         U
        ses_button.h      U
        ses_common.h      U
        ses_lcd.h         U
        ses_led.c         U
        ses_led.h         U
        ses_uart.h        U
  v task_2-3
    > .pio
    > .vscode
    v src
        main.c            U
      .gitignore          U
      platformio.ini      U
  v task_2-6
    > .pio
    > .vscode
    v src
        main.c            U
      .gitignore          U
      platformio.ini      U
```

(b) Workspace with one project per task

Figure 1: Examples for possible workspace structures in Visual Studio Code.



```
#include <stdbool.h>
#include <stdlib.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
```

(a) The editor can not find included files.

```
v task_2-3
  > .pio
  v .vscode
      c_cpp_properties.json   4
      extensions.json
      launch.json
  > src
    .gitignore
    platformio.ini
```

(b) Warning that the folder *include* is missing.

Figure 2: Warnings produced by Visual Studio Code.

Is PlatformIO able to build your code, but the editor in Visual Studio Code is not able to find the included files in the SES folder (see Fig. 2a)?

In general, it is not necessary to solve this issue, because it is independent of the building process with PlatformIO. However, these messages are confusing and distract from actual errors. If you want to get rid of it, add a new folder *.vscode* to *./lib/ses/*. Afterwards create a new file with the name *c_cpp_properties.json* in the folder *./lib/ses/.vscode/* and add the following lines to this file.

For Linux-based operating systems:

```
{
    "configurations": [
        {
            "name": "Linux",
            "includePath": [
            ],
            "defines": [
                "F_CPU=16000000L",
                "AVR_ATmega128RFA1",
                ""
            ],
            "compilerPath": "~/.platformio/packages/toolchain-atmelavr/bin/avr-gcc",
            "cStandard": "c11",
            "cppStandard": "c++11",
            "intelliSenseMode": "clang-x64",
            "compilerArgs": [
                "-mmcu=atmega128rfa1",
                ""
            ]
        }
    ],
    "version": 4
}
```

For Windows:

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
            ],
            "defines": [
                "F_CPU=16000000L",
                "AVR_ATmega128RFA1",
                ""
            ],
            "compilerPath": "~/.platformio/packages/toolchain-atmelavr/bin/avr-gcc.exe",
            "cStandard": "c11",
            "cppStandard": "c++11",
            "intelliSenseMode": "clang-x64",
            "compilerArgs": [
                "-mmcu=atmega128rfa1",
                ""
            ]
        }
    ],
    "version": 4
}
```

A second warning in your regular PlatformIO project folder (e. g., *task_2-3* in Fig. 2b) could be, that it is not possible to find the folder *include*. To solve this problem, add the folder *include* to the project. To push this folder to your repository, you can add an empty file to this folder, e. g., *empty_folder.txt*.

Task 2.3 : Start-up Messages

- Open the *main.c* file in the folder *src* in your current project.
- Include *ses_lcd.h* and *ses_uart.h* in the *main.c*.

```
#include "ses_lcd.h"
#include "ses_uart.h"
```

- Add the lines

```
uart_init(57600);
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");
fprintf(lcdout, "START");
```

  Don't forget adding 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 (or the **Reset MCU** in the remote builder interface).

After compiling and flashing, the microcontroller sends the message via the UART to the PC. To see the serial output of your SES board, you have to press **Play** in the remote builder interface.

There is a small delay of the image from the webcam. You can switch on and off the light in the control environment to get a feeling for it.

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. Please note that you can either use the UART or the LCD with `printf`, but not both at the same time.

Task 2.4 : Writing an LED Device Driver

In the first exercise 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 file *ses_button.c* and provide the implementation of the functions declared in the *ses_button.h* (from template folder).
- Create macros for the button connection analogously to the ones in the *ses_led.c*.
- The function

  ```
  void button_init(void)
  ```

  initializes the button library.
  - 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 [1] (write a `1` to the corresponding bits of the buttons' port `PORTB`)
- When the button is pressed, the corresponding pin is grounded and the input can be read as logic *low* level via `PINB`. The functions

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

  return the status of the joystick and rotary button.

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 file *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
- Show the seconds since reset on the LCD

The buttons in the remote builder interface have the same behavior compared to the simple buttons. When the button is pressed, a relay in the lab is closed. This behavior enables to hold the button over a longer period.

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 with this too much, you will learn about a much better approach in future exercise.

[1] 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.

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[2]. 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 is connected differential to the pins 0 and 1 of port F.[3]

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 SES 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 used reference voltage, set the macro `ADC_VREF_SRC` (in the file *ses_adc.c*). The reference voltage should be set to 1.6 V.
- Use the `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`).
- Enable the ADC (`ADEN`).

After the initialization the ADC is ready for the voltage conversion. The following function triggers the conversion and returns the result:

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

`adc_channel` contains 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

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

[3] The microphone is not used in this exercise, however

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 program execution. When the conversion is finished, the microcontroller hardware will reset the `ADSC` bit to zero. You can interrupt the program flow and use polling to check the `ADSC` bit.[4] The result is readable from the 16 bit register `ADC`, which is automatically combined from the 8 bit registers `ADCL` and `ADCH`.

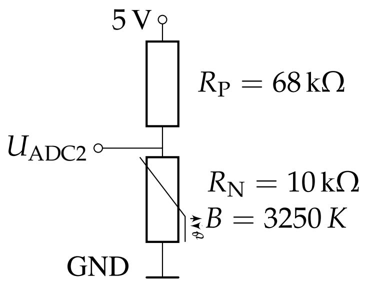
If you use `ADCL` and `ADCH` to read out the ADC, make sure that you **always** read from `ADCH` after you have read from `ADCL`, otherwise the data register will be blocked and conversion results are lost.

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 temperature sensor. For the light sensor, use the raw value. For testing purposes, write the results to the UART or the LCD.

```
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.



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

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

Thereby, the 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[5]. 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 $\frac{1}{10}$°C, i.e., a value of 10 represents 1.0 °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 purpose, calculate the raw ADC values for two temperature values (e.g. 40°C and 20°C) and use

[4] 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.

[5] R_T is the resistance of the thermistor at the temperature T (in K). R_N is the resistance at a reference temperature of $T = 298.15\,\text{K} = 25\,°\text{C}$.

the following code to calculate interpolated 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:

- Every 2500 milliseconds, read the values of the temperature and the light and write them to the UART.

✎ The light sensor has a low resolution. Do not worry about noisy raw values. As a practical example, the sensor could be used to turn the LCD light on and off.

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 with the function `_delay_ms`, the execution time of other functions (ADC read and LCD display) are neglected. However, these function has an execution time in the millisecond range.

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 an oscilloscopes or logic analyzers.

Due to the restrictions of the remote builder interface, the following code was flashed to a SES board (an hex image version is also available in Stud.IP).

```c
#include <stdint.h>
#include <util/delay.h>

#include "ses_lcd.h"
#include "ses_uart.h"
#include "ses_led.h"

#define BAUDRATE 57600

/**
 * Print the time since boot on the LCD
 * @timeBoot Time since boot in seconds
 */
void printTime(uint16_t timeBoot) {
    lcd_clear();
    // first row
    lcd_setCursor(0,0);
    fprintf(lcdout,"Time since reset:");
    // second row
    lcd_setCursor(0,1);
    fprintf(lcdout,"%4u seconds",timeBoot);
}

/**
 * Write the time since boot to the UART
 * @timeBoot Time since boot in seconds
 */
void sendTime(uint16_t timeBoot) {
    fprintf(uartout,"%4u seconds since reset.\n",timeBoot);
}

/**
 * main function
 */
int main (void) {
    led_redInit();
    led_greenInit();
    led_yellowInit();

    lcd_init();
    uart_init(BAUDRATE);

    uint16_t timeBoot = 0; // time since boot in seconds
    // set all LED outputs to a low level
    led_redOn();     // pin PG1
    led_yellowOn();  // pin PF7
    led_greenOn();   // pin PF6

    // superloop
    while (1) {
        // 1 second delay
        led_redOff();
        _delay_ms(1000);
        timeBoot++;
        led_redOn();
        // print on LCD
        led_yellowOff();
        printTime(timeBoot);
        led_yellowOn();
        // write to UART
        led_greenOff();
        sendTime(timeBoot);
        led_greenOn();
    }
    return 0;
}
```

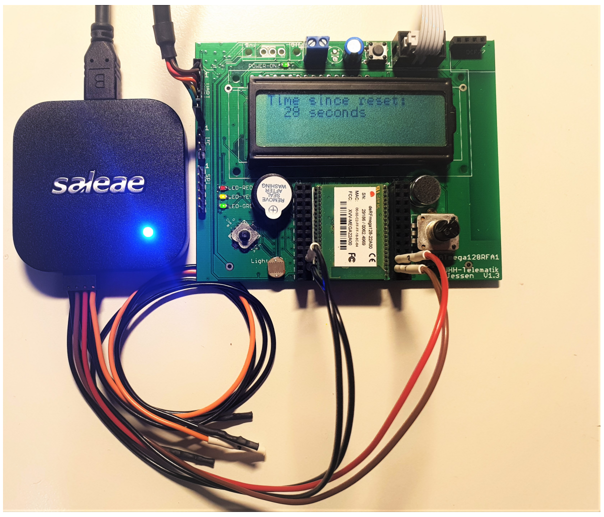


Figure 3: Setup to measure timings with the help of a logic analyzer.

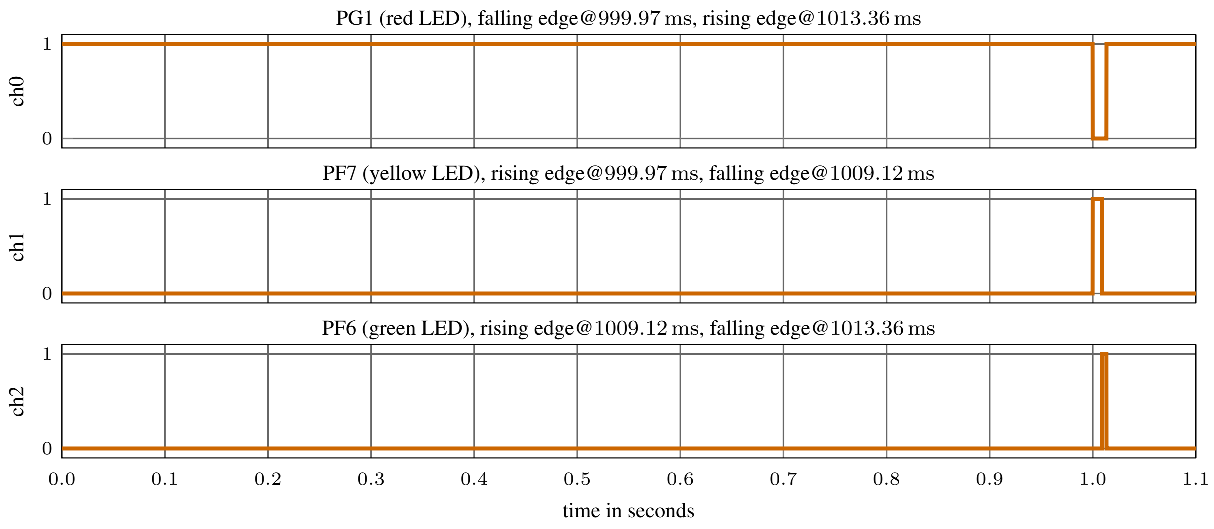


Figure 4: Recorded logic channels (trigger was set to a rising edge on channel 0).

In addition, a logic analyzer was connected to the LED pins (see Fig. 3) and recorded one cycle of the while-loop. The logic analyzer sampled with 10 MHz and the trigger was set to start the recording, when a rising edge on channel 0 (connected to pin PG1) was detected. Figure 4 depicts the measured signals.

Try to give an answer to the following points:

- What is the drift of the implemented timer?
- What happens, if you extent the number of tasks in the superloop?
- Think about a better implementation.