Exercise Sheet

Peripherals

Building and Using Libraries for In- and Output

April 23rd, 2025

You are supposed to work on this sheet for **two** lab sessions! Please refer to the *roadmap* (available in Stud.IP).

In this and the upcoming exercise sheets, you will create several drivers and modules for the peripherals of the SES board. Those drivers will be reused in later (also graded) sheets to create increasingly complex systems. To enable reuse of all the basic functionality while avoiding code duplication, you will put your drivers (starting with LED, Buttons, and ADC in this exercise) into a library, which will be used by upcoming exercise sheets as needed. Additionally, you will learn to include and use precompiled libraries for textual outputs using USB connection and display. The USB connection enables serial communication between the microcontroller and the PC. Especially in conjunction with `printf`, the USB connection is an excellent debugging facility that comes in handy in future tasks.

Even though this exercise is not graded, the created library is part of later submissions and their grading, so following our coding style is essential!

Task 2.1: An SES library

As a first step, download and extract the *sheet2_templates.zip* archive from Stud.IP and copy the files into the folder *./lib/ses/* in your workspace (please use the prepared workspace from your Git repository as described in Task 1.1).

For the upcoming lab sessions, upload all source files to your Git repository. The *Git Tutorial* outlines all instructions on how to do that. Please ensure you only commit source and header files, plus your workspace configuration.

Keep the structure of your *lib/ses* directory flat, i.e., do not use subdirectories! Do not upload any binary files, compiled images, etc. to your repository!

Task 2.2: Add a new Project to your Workspace

Testing is an integral part of the software development process. For this reason, you will write an elementary test application that will grow throughout subsequent tasks. All applications should be located in individual projects. You are free, however, to use a single 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 subfolder *src*.

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

1. Create a new folder in your workspace, e. g., *task_2*.

TUHH Hamburg University of Technology

Prof. Dr.-Ing. B. Renner
Institute for Autonomous Cyber-Physical Systems

2. Copy the file *platformio.ini* from a previous project to this folder.
3. Add the subfolder *src* to the folder from the first step.
4. Create (or copy) the file *main.c* to the subfolder *src*.
5. Open Visual Studio Code, and add the folder from the first step to your workspace (`File -> Add Folder to Workspace...`).
6. After *building* the created task the folders *.pio* and *.vscode* should appear automatically. If this is not the case, restart Visual Studio Code.

The template archive *sheet2_templates.zip* contains three precompiled libraries (*libdisplay.a*, *libLUFA.a*, and *libusbserial.a*) and a set of header files. We use the open source library LUFA[1] for USB communication. To link these libraries to your project (during compilation) and make the headers available to your project, extend the *platformio.ini* file of your new project to

```
[env:ses_avr]
platform = atmelavr
board = leonardo

board_upload.use_1200bps_touch = false
board_upload.wait_for_upload_port = false

lib_extra_dirs = ../lib

build_flags =
    -L ../lib/ses/
    -l usbserial
    -l LUFA
    -l display
```

If VSCode complains about missing folders in the configuration file *c_cpp_properties.json* create a (empty) subfolder named *include* in your project's main folder (e. g., *task_2*).

Task 2.3 : Start-up Messages

- Open the file *main.c* in the folder *src* in your newly created project.
- Include the headers *ses_usbserial.h* and *ses_display.h* in the *main.c*.

```
#include "ses_usbserial.h"
#include "ses_display.h"
```

- Add the lines

```
usbserial_init();
display_init();
sei(); // Activate interrupts. Required to use USB serial communication.
```

to the initialization section of the main function, followed by an empty infinite **`while`** loop.

[1] http://www.fourwalledcubicle.com/files/LUFA/Doc/210130/html/

- To print a message to USB serial communication and display, use the statements

```
fprintf(serialout, "Your USB output goes here\n");

display_setCursor(0,0); // Set cursor to column 0, row 0
fprintf(displayout, "Your display output goes here");
display_update();
```

  inside the **`while`** loop. 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.

After compiling and flashing, the microcontroller sends the message via the USB to the PC. To see the received messages, click on the *Serial Monitor* icon in the PlatformIO toolbar.

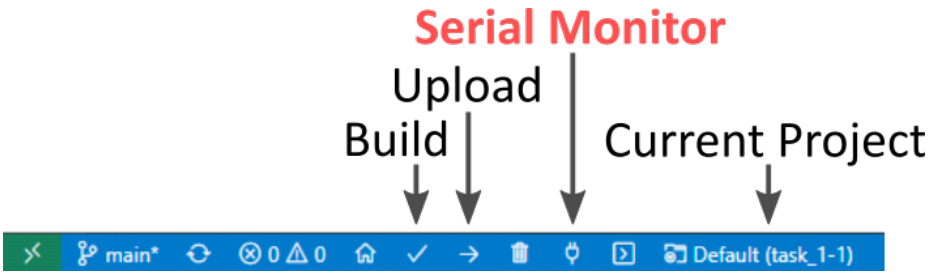


Figure 1: PlatformIO toolbar

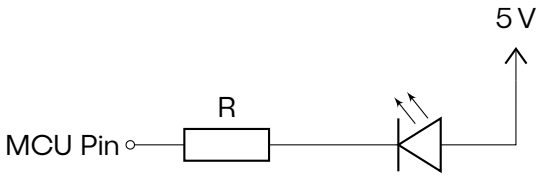
The USB serial and display libraries define the `FILE` descriptors `serialout` and `displayout`, which can be used with `fprintf()`. You can also use `printf()` instead, which can write **either** to the USB serial **or** the display. Therefore you need to set either `stdout=serialout;` or `stdout=displayout;` right after the initialization of the USB serial or display, respectively.

Task 2.4 : Writing an LED Device Driver

In the exercise of sheet 1, one LED was toggled using bit operations. Repeating these bit operations at many places in a program is not very convenient. Furthermore, this way of using LEDs leads to code duplication and requires changes at many locations if you want to change the LED pin, for example. A better approach is a driver for using the LEDs. The driver consists of two parts: the header file *ses_led.h* and the source file *ses_led.c*. Copy the files from the previously downloaded zip file to your newly created `ses` *library* project.

The provided source file already defines all constants, e. g., the port and the LED pins. Furthermore, the header file contains all function declarations and describes the desired functionality of the functions. In this exercise, you have to implement the functions! Use the provided function bodies already present in *ses_led.c*.

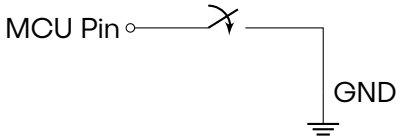
The LEDs are *active low* (a logic *high* will turn them off).



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 left of the module and is connected to pin 5 of port B. The push button is located on the lower right and is connected to pin 4 of port B. A pressed button pulls the input pin to GND:



Implement the library driver for both buttons:

- In the *library* project, create file *ses_button.c* and provide the implementation of the function prototypes declared in *ses_button.h* (from the template folder).
- Create macros for the button connection analogously to the ones in *ses_led.c*.
- The function

```
void button_init(void);
```

  initializes the button library.
  - Configure the pin of each button as an input (write a logic `0` to the corresponding bit of the data direction register `DDRB`).
  - Activate the internal pull-up resistor for each of the buttons [2] (write a logic `1` to the corresponding bits of the buttons' port `PORTB`).
- When a button is pressed, the corresponding pin is grounded, and the input can be read as logic *low* level via `PINB`. The functions

```
bool button_isPushButtonPressed(void);
bool button_isRotaryButtonPressed(void);
```

  return the logical state of the push and rotary buttons, i.e. true if button is pressed, false otherwise.

When setting or clearing bits in any register, be sure to leave all other bits unchanged!

[2] 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.

TUHH Hamburg University of Technology
Prof. Dr.-Ing. B. Renner
Institute for Autonomous Cyber-Physical Systems

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 push button is pressed, light up the green LED.
- Show the seconds since MCU reset on the display.

To obtain the time since boot, add a small delay after checking the buttons in the superloop and sum up the delays or loop cycles. The delay should be much longer than the ADC sampling times (next task), but short enough to make the application sufficiently responsive (500 ms is a good starting point).

This subtask illustrates the pain of superloops. Unless you provide a very clever implementation, the buttons will show impaired responsiveness and inaccurate timings. Do not bother yourself with this too much—you will learn about a much better approach in future exercises.

Analog to Digital Converter

The Analog-to-Digital Converter converts an analog input voltage to a digital value. The ATmega32U4 features an ADC with 10-bit resolution, yielding 2^{10} distinct digital values, which are linearly mapped to analog voltages[3]. The ADC is connected via a multiplexer to one of 12 external channels and a few internal channels.

On the SES board, the following peripherals are connected to the ADC:

- A potentiometer at pin 6 of port F.
- An analog temperature sensor at pin 7 of port F.
- A light sensor at pin 0 of port F.

All components output an analog voltage between 0 V and 3.3 V. More information is provided in the ATmega32U4 datasheet in chapter *24. Analog to Digital Converter — ADC*.

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 (potentiometer, temperature, light) and deactivate their internal pull-up resistors.
- Disable power reduction mode for the ADC module by clearing the `PRADC` bit in register `PRR0`.
- Select an external reference voltage (3.3 V is connected to the AREF pin) in register `ADMUX`

[3] In this exercise, a reference voltage of 3.3 V is used, thus the ADC output corresponds to analog voltages from the interval $[0\,V, 3.3\,V - \frac{3.3V}{2^{10}}]$

- Configure the ADLAR bit, which should set the ADC result right-adjusted.
- The ADC clock is derived from the CPU clock via a prescaler. It must be 200 kHz or less for the full 10-bit resolution. Find a prescaler setting that achieves the fastest operation within this range! 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 initialization, the ADC is ready for voltage conversions. Implement the following function that triggers the conversion and returns the result:

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

The parameter adc_channel contains the sensor type (enum ADChannels). By now, the ADC is configured in a run-once mode, which will start for the channel selected in ADMUX. If adc_channel is invalid, ADC_INVALID_CHANNEL shall be returned. Otherwise, the function should return the raw conversion result.

Setting ADSC in ADCSRA starts a single conversion, which will run in parallel to program execution, i. e., you have to make sure that you wait for the conversion to finish before you try to read the result. Once finished, the microcontroller hardware resets the ADSC bit to zero. We suggest using polling[4] (busy waiting) to check the ADSC bit and only continue once the conversion result is available

The result is available in the 16-bit register ADC, which combines 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

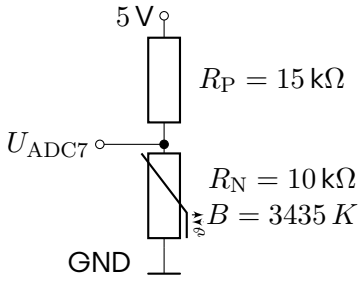
Usually, the raw ADC value is not very meaningful; conversion to the corresponding physical quantity is favorable. In this task, you will provide a function to read the temperature sensor's raw value and calculate the temperature in tenth degree Celsius ($\frac{1}{10}$°C):

```
int16_t adc_getTemperature(void);
```

For temperature measurement, we use an NTC thermistor with a temperature-dependent resistance R_T. The following circuit diagram shows its connection to the microcontroller.

[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 next exercise sheet introduces the concept of interrupts.

TUHH Hamburg University of Technology
Prof. Dr.-Ing. B. Renner
Institute for Autonomous Cyber-Physical Systems



According to the voltage divider rule, the voltage U_{ADC7} at ADC pin 2 is

$$U_{\mathrm{ADC7}} = 5\,\mathrm{V} \cdot \frac{R_\mathrm{T}}{R_\mathrm{T} + R_\mathrm{P}}. \tag{1}$$

A simplified equation for the temperature of the thermistor in Kelvin is given by

$$T = \frac{B \cdot 298.15\,\mathrm{K}}{B + \ln\left(\frac{R_\mathrm{T}}{R_\mathrm{N}}\right) \cdot 298.15\,\mathrm{K}}\,, \tag{2}$$

with B and R_N being characteristics[5] of the thermistor as given in the circuit diagram. Based on the above equations, provide a function that converts the raw ADC value to the corresponding temperature. The unit of the return value with type `int16_t` shall be $\frac{1}{10}$°C, i.e., a value of 10 represents 1 °C.

However, since the computational power of the ATmega32U4 is limited and floating-point operations are computationally expensive, linear interpolation should be used instead of applying the formulas directly. For this purpose, calculate the raw ADC values for two temperature values (e.g., 10 °C and 30 °C) and use the following code to calculate interpolated values. Make sure to define the constants as integers.

```
int32_t adc = adc_read(ADC_TEMP_CH);
return (int16_t)(((adc - ADC_TEMP_RAW_LOW) * (ADC_TEMP_HIGH - ADC_TEMP_LOW))
    / (ADC_TEMP_RAW_HIGH - ADC_TEMP_RAW_LOW)) + ADC_TEMP_LOW;
```

Please note that the underlying math is based on the linear interpolation equation

$$T \;=\; (r - R_\mathrm{low}) \cdot \frac{T_\mathrm{high} - T_\mathrm{low}}{R_\mathrm{high} - R_\mathrm{low}} + T_\mathrm{low} \;=\; (r - R_\mathrm{high}) \cdot \frac{T_\mathrm{high} - T_\mathrm{low}}{R_\mathrm{high} - R_\mathrm{low}} + T_\mathrm{high}\,, \tag{3}$$

where r is the current raw ADC value and R_low and R_high are the raw ADC values for the two temperature references T_high and T_low, respectively. Please note that R_low and R_high can be obtained for T_high and T_low from equations Eq. (2) and Eq. (1).

Extend the superloop with the following functionality:

- Every 500 ms, read the values of the potentiometer, the temperature, and the light sensor and show it on the display.

[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 $T = 298.15\,\mathrm{K} = 25\,^\circ\mathrm{C}$.