

## Grading System

The score will be 10 points per exercise (13 points with the challenge task):

- **6 points: Functionality**

Implement main program and library functions corresponding to their functional description.

- **3 points: Coding Style**

Continuously use **only one** coding style (do a web search on *c programming style guides*); avoid code redundancy, use proper names for functions and variables, do not use Magic Numbers, promote readability (white space, indentation).

- **1 point: Documentation**

Take care to a proper documentation of your code: description of the function behavior, parameters, and return values (it is recommend to use a JavaDoc compliant documentation); explanation of the file content; documentation of complex non-intuitive parts/ lines.

- **3 points: Solving the Challenge Task**

You think the exercises are too easy? Then solve the task marked as *Challenge* in order to earn up to 3 bonus points. Note that tasks marked as *Challenge* are not reused in later exercises.

In total 3 selected exercises will be graded, so a total number of 30 points can be achieved (up to 39 points with the challenge tasks). The results from these exercises will be combined with the result from the written examination at the end of the semester. The examination is weighted with 70 points, adding up to a total of 100 points (109 points with the challenge tasks, capped at 100).

Note that doing the exercise is not a prerequisite for joining the exam. Nevertheless a high score in the exercise will lead to a better final mark.

## Submission

All exercises are to be solved and submitted from a team consisting of two members. Submission is done via the git server which is provided for the course (see below). The directory structure used in the repository has to correspond *exactly* to the requirements stated on the exercise sheet, deviations will lead to deduction of points.

The deadline for submission will also be noted on the corresponding exercise sheet.

Git server:

[https://collaborating.tuhh.de/smartport/students/ses-2020/groups/group\\_X](https://collaborating.tuhh.de/smartport/students/ses-2020/groups/group_X)

(where X is your team number)

## Interview

In the first week after submission there will be an interview during lab hours. Here you have to explain your programming solution and we will ask questions about your code. Both team members have to be able to explain the **entire** solution, which includes library functions from earlier (non-graded) exercises which are referenced in the code for the graded exercise. Participation in those lab classes is *mandatory*. No points are granted for solutions which cannot be explained.



You can use the hardware beyond the exercise sessions, too. The remote access will be permanently available.



Plagiarism will be penalized with 0 points for all involved teams! This also includes partial copying of another program, e.g. with renamed variables.

Your submissions have to pass a plagiarism check. In addition to the actual submissions from your fellow students, the plagiarism check compares your code to the submissions from the previous semesters.

## Coding Style Examples

In addition to the functionality of your code, a good coding style is important. At first, the students are referred to the additional slides *SES Appendix: C for Embedded Systems*, which gives an extensive introduction. In addition to that, we decided to provide an (incomplete) list with examples. In all cases, the good (✓) and bad (✗) implementations work. However, the green code (✓) is a smarter implementation.

## Code Formatting

Whites spaces and indentations enhance the readability.

```
int main (void) {
    led_redInit();
    button_init();
    lcd_init();
    uint16_t timeBoot = 0; // time since boot in seconds
    while (1) {
        if (button_isRotaryPressed()) {
            led_redOn();
        }else{
            led_redOff();
        }
        printTime(timeBoot);
        _delay_ms(1000);
        timeBoot ++;
    }
    return 0;
}
```




```
int main (void) {
    led_redInit();
    button_init();
    lcd_init();

    uint16_t timeBoot = 0; // time since boot in seconds

    while (1) {
        if (button_isRotaryPressed()) {
            led_redOn();
        }else{
            led_redOff();
        }

        printTime(timeBoot);


        _delay_ms(1000);
        timeBoot ++;
    }
    return 0;
}
```




## Function and Variable Names

Proper names for functions and variables establish a better understanding of your code.

```
int16_t adc_getTemperature() {
    // adc value
    int16_t a = adc_read(ADC_TEMP_CH);
    // slope
    int16_t b = (ADC_TEMP_MAX - ADC_TEMP_MIN) / (ADC_TEMP_RAW_MAX - ADC_TEMP_RAW_MIN);
    // offset
    int16_t c = ADC_TEMP_MAX - (ADC_TEMP_RAW_MAX * b);
    // factor
    int16_t d = ADC_TEMP_FACTOR;
    // magic calculation..
    return (a * b + c) / d;
}
```



```
int16_t adc_getTemperature() {
    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;
}
```



## Return Boolean Values

Do not think too complicated in the case of boolean values. It is possible to return boolean statements.

```
bool button_isJoystickPressed(void) {  
    if (!(PIN_REGISTER(BUTTON_JOYSTICK_PORT) & (1 << BUTTON_JOYSTICK_PIN))) {  
        return true;  
    }else{  
        return false;  
    }  
}
```



```
bool button_isJoystickPressed(void) {  
  
    bool return_value = false;  
  
    if ((PIN_REGISTER(BUTTON_JOYSTICK_PORT) & (1 << BUTTON_JOYSTICK_PIN))) {  
        return_value = false;  
    }else{  
        return_value = true;  
    }  
    return return_value;  
}
```



```
bool button_isJoystickPressed(void) {  
    return !(PIN_REGISTER(BUTTON_JOYSTICK_PORT) & (1 << BUTTON_JOYSTICK_PIN));  
}
```



## Boolean Statements

When a function returns a boolean value, it is not necessary to compare it to true or false again.

```
if (button_isJoystickPressed() == true) {  
    // do something  
}
```



```
if (button_isJoystickPressed()) {  
    // do something  
}
```



## Smart Implementation

Also in this case, do not think too complicated..

```
// wait until conversion is completed
while(1){
    if ((ADCSRA & (1 << ADSC)) == 0) {
        break;
    }
}
```



```
// wait until conversion is completed
while(ADCSRA & (1 << ADSC));
```



The variable `adc_channel` can be directly used instead of the implementation of a complicated switch statement.

```
// select ADC channel
switch(adc_channel) {

    case ADC_TEMP_CH: // == 2
        ADMUX = (ADMUX & ~ADC_MUX_MASK) | (0x02 & ADC_MUX_MASK);
        break;

    case ADC_LIGHT_CH: // == 4
        ADMUX = (ADMUX & ~ADC_MUX_MASK) | (0x04 & ADC_MUX_MASK);
        break;

    default:
        break;
}
```



```
// select ADC channel
ADMUX = (ADMUX & ~ADC_MUX_MASK) | (adc_channel & ADC_MUX_MASK);
```



### Avoid Redundancy

The function `led_redOff()` can be used in `led_redInit()` to avoid redundancy.

```
void led_redInit(void) {  
    DDR_REGISTER(LED_RED_PORT) |= (1 << LED_RED_PIN);  
    LED_RED_PORT |= (1 << LED_RED_PIN);  
}
```

```
void led_redOff(void) {  
    LED_RED_PORT |= (1 << LED_RED_PIN);  
}
```



```
void led_redInit(void) {  
    DDR_REGISTER(LED_RED_PORT) |= (1 << LED_RED_PIN);  
    led_redOff();  
}
```

```
void led_redOff(void) {  
    LED_RED_PORT |= (1 << LED_RED_PIN);  
}
```



### Use AVR Definitions

Use the register and bit definitions from *avr/io.h*. The usage makes your code portable to other AVR platforms.

```
// Disable power reduction mode  
PRR0 &= ~(1 << 0); // PRADC
```



```
// Disable power reduction mode  
PRR0 &= ~(1 << PRADC);
```



## Useful Comments

Comments are important to understand the implementation. Indeed, comments should explain the background of your code. In this example is easy to see, that `PRADC` is cleared in `PRR0`. Based on that, it is not necessary to explain this behavior in a comment. More imported is, why do you clear the bit `PRADC` in the register `PRR0`?

```
// Clear bit PRADC in the register PRR0  
PRR0 &= ~(1 << PRADC);
```



```
// Disable power reduction mode  
PRR0 &= ~(1 << PRADC);
```



## Definitions Instead of Variables

Use definitions for previously defined values.

```
int16_t adc_getTemperature() {  
    int16_t ADC_TEMP_FACTOR = 0;  
    int16_t ADC_TEMP_RAW_MIN = 0;  
    int16_t ADC_TEMP_RAW_MAX = 0;  
    int16_t ADC_TEMP_MIN = 0;  
    int16_t ADC_TEMP_MAX = 0;  
  
    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;  
}
```



```
#define ADC_TEMP_FACTOR    0  
#define ADC_TEMP_RAW_MIN  0  
#define ADC_TEMP_RAW_MAX  0  
#define ADC_TEMP_MIN      0  
#define ADC_TEMP_MAX      0  
  
int16_t adc_getTemperature() {  
    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;  
}
```



## Smart Definitions and Marco Implementations

Definitions should enhance the changeability and readability of your implementation. But it is not a good style to redefine everything.

```
#define ADC_MUX_MASK      0x1F
#define SES_ADMUX         ADMUX
#define SES_ADCSRA        ADCSRA
#define SES_ADSC          ADSC

// select ADC channel
SES_ADMUX = (SES_ADMUX & ~ADC_MUX_MASK) | (adc_channel & ADC_MUX_MASK);

// start conversion
SES_ADCSRA |= (1 << SES_ADSC);
```



```
#define ADC_MUX_MASK      0x1F

// select ADC channel
ADMUX = (ADMUX & ~ADC_MUX_MASK) | (adc_channel & ADC_MUX_MASK);

// start conversion
ADCSRA |= (1 << ADSC);
```



However, in some cases too many macros can reduce the readability and comprehensibility.

```
#define ADD(a,b) ((a) + (b))
#define DIFF(a,b) ((a) - (b))
#define MUL(a,b) ((a) * (b))
#define DIV(a,b) ((a) / (b))

int16_t adc_getTemperature() {
    int16_t adc = adc_read(ADC_TEMP_CH);
    int16_t slope = DIV(DIFF(ADC_TEMP_MAX, ADC_TEMP_MIN), DIFF(ADC_TEMP_RAW_MAX, ↵
    ↵ ADC_TEMP_RAW_MIN));
    int16_t offset = DIFF(ADC_TEMP_MAX, MUL(ADC_TEMP_RAW_MAX, slope));
    return DIV(ADD(MUL(adc, slope), offset), ADC_TEMP_FACTOR);
}
```



```
int16_t adc_getTemperature() {
    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;
}
```

