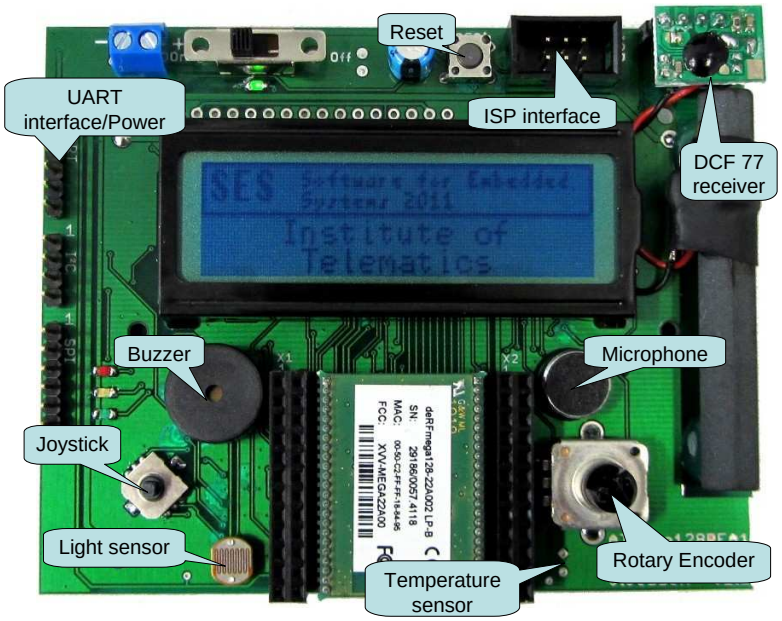
Tools & Tints

April 28th, 2020

Successful participation in this exercise is required to continue the course. Please push your code and Screenshots to your Git repository until Tuesday (05.05.2020).

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.



Preparation

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

Read the *Git Usage Guide* (Stud.IP) and make sure you have your project folder connected to your groups git repository. Commit and push your solutions to the repository regularly. We expect every group to have a working project pushed to their git repository until next weeks exercise session.

As a proof, that you got the toolchain running correctly, both team members must **individually** push a screenshot into the root folder of their Git repository showing the VSCode screen and the output of the compiler showing a *Success*. **This is mandatory to continue participation in the exercise**.

Since the exercise can not take place in person due to the Corona virus, you will be using a web interface to upload your programs to boards in our lab. The usage is described in our *Toolchain Tutorial* (Stud.IP).

Task 1.1 : Get the Party Started

Open a new VSCode window and click on *File* → *OpenWorkspace*. In the root of your Git repository, there is a file called `ses-workspace.code-workspace`. Select this.

In your workspace is already a folder for every exercise in this sheet. You will learn in the next exercise, how to extend your workspace with new projects. Have a look into the folder *task_1-2/src*. In there is now the simple blinker program from the toolchain directory.

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

/**Toggles the red LED of the SES-board*/
int main(void) {
    DDRG |= 0x02;

    while (1) {
        _delay_ms(1000);
        PORTG ^= 0x02;
    }
    return 0;
}
```

Here, 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.

To find out how to toggle the different LEDs please have a look at the *SES Board Circuit Diagram* (Stud.IP)

Make sure you can build this program. Push a screenshot in this Git repository showing the compiler output until next weeks exercise session.

Task 1.2 : Looping Louie

For this exercise, change to the next folder *task_1-2*. Always use a new folder for every exercise.

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 *Toolchain 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 inspecting the assembler code, that the compiler generates.

Displaying the assembler code is a bit tricky with PlatformIO. In the Menu bar at the top of the VSCode window, select *Terminal* → *New Terminal*. In the terminal, which appears, you now have to enter a command that takes the compiled .elf file and generates human readable assembler code from it. The `.elf` file is placed by the compiler to the same directory as the `.hex` file.

The program needed for translation is part of the toolchain installed by PlatformIO into your user directory. The toolchain exexcutables are by default placed to

`~/.platformio/packages/toolchain-atmelavr/avr/bin`.

In Linux, MAC and modern Windows systems—where PowerShell is available—the command is

`<toolchain-path>/objdump -h -S <elf-path>`

After calling this function, you see the assembler code for each function in the terminal window. 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 change the LED program to use your delay function instead of the `_delay_ms(int ms)` function. After exactly 1 second toggle the LED. Check your program by measuring the time with a watch, that it takes to blink exactly 60 times. The long observation time should compensate little delays introduced by the webcam-stream.

Task 1.4 : Practice C

During the semester, you will often have to manipulate single bits in registers. In this exercise, you can practice that. Make sure you have installed the VSCode extension *C/C++ Compile Run* as described in the *Toolchain Tutorial*. Switch to the folder *task_1-4* and open the file *bit_shifting.c* run it by pressing `F6`.

Try to understand the program. If you like, play around with it.

Task 1.5 : A Bit of Masking and Shifting

Bit operations are performed in oder 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 variabe `uint8_t var` with one C statement. You should do this by hand, afterwards you can check you 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 1.6 : 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 `input`$= (b_7b_6b_5b_4b_3b_2b_1b_0)$ the output becomes `output`$= (b_0b_1b_2b_3b_4b_5b_6b_7)$.

Task 1.7 : 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)