# Lab 2: Control of GPIO pins

### Learning objectives

After completing this lab you will be able to:

- Configure input/output ports of AVR using control registers
- Use ATmega328P manual and find information
- Understand the difference between header and source files
- Create your own library
- Understand how to call a function with pointer parameters

The purpose of this laboratory exercise is to learn how to create your own library in C. Specifically, it will be a library for controlling GPIO (General Purpose Input/Output) pins with help of control registers.

### Table of contents

### Components list

- Arduino Uno board, USB cable
- Breadboard
- 2 LEDs
- 1 two-color LED
- 4 resistors
- 1 push button
- Jumper wires
- Logic analyzer

## Pre-Lab preparation

1. Fill in the following table and enter the number of bits and numeric range for the selected data types defined by C.

| **Data type** | **Number of bits** | **Range** | **Description** |
|---|---|---|---|
| `uint8_t` | 8 | 0, 1, ..., 255 | Unsigned 8-bit integer |
| `int8_t` | | | |
| `uint16_t` | | | |
| `int16_t` | | | |

| Data type | Number of bits | Range | Description |
|---|---|---|---|
| `float` | | -3.4e+38, ..., 3.4e+38 | Single-precision floating-point |
| `void` | -- | -- | *Incomplete type that cannot be completed* |

2. Any function in C contains a declaration (function prototype), a definition (block of code, body of the function); each declared function can be executed (called). Study this article and complete the missing sections in the following user defined function declaration, definition, and call.

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

// Function declaration (prototype)
uint16_t calculate(uint8_t, ***    );

int main(void)
{
   uint8_t a = 210;
   uint8_t b = 15;
   uint16_t c;

   // Function call
   c = ***      (a, b);

   // Infinite loop
   while (1) ;

   // Will never reach this
   return 0;
}

// Function definition (body)
***      calculate(uint8_t x, uint8_t y)
{
   uint16_t result;    // result = x^2 + 2xy + y^2

   result = x*x;
   ***
   ***
   return result;
}
```

3. Find the difference between a variable and pointer in C. What mean notations `*variable` and `&variable`?

## Part 1: Synchronize repositories and create a new project

When you begin working, ensure that you synchronize the contents of your working folder and local repository with the remote version on GitHub. This practice ensures that none of your changes are lost.

1. In your working directory, use **Source Control (Ctrl+Shift+G)** in Visual Studio Code or Git Bash (on Windows) or Terminal (on Linux) to update the local repository.

   > **Help:** Useful bash and git commands are `cd` - Change working directory. `mkdir` - Create directory. `ls` - List information about files in the current directory. `pwd` - Print the name of the current working directory. `git status` - Get state of working directory and staging area. `git pull` - Update local repository and working folder.

2. In Visual Studio Code create a new PlatformIO project `lab2-gpio_library` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB2-GPIO_LIBRARY > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

## Part 2: GPIO control registers

AVR microcontroller associates pins into so-called ports, which are marked with the letters A, B, C, etc. Each of the pins is controlled separately and can function as an input (entry) or output (exit) point of the microcontroller. Control is possible exclusively by software via control registers.

There are exactly three control registers for each port: DDR, PORT and PIN, supplemented by the letter designation of the port. For port A these are registers DDRA, PORTA and PINA, for port B registers DDRB, PORTB, PINB, etc.

DDR (Data Direction Register) is used to set the input/output direction of port communication, PORT register is the output data port and PIN register works for reading input values from the port.

A detailed description of working with input/output ports can be found in ATmega328P datasheet in section I/O-Ports.

1. Copy/paste your solution with two LEDs from Lab1 to `LAB2-GPIO_LIBRARY > src > main.c` source file. Compile (build) the project and note the first-version size in bytes.

| Version | Size [B] |
|---|---|
| Ver. 1: Arduino-style | |
| Ver. 2: Registers | |
| Ver. 3: Library functions | |

2. Use the datasheet to find out the meaning of the DDRB and PORTB control register values and their combinations. (Let PUD (Pull-up Disable) bit in MCUCR (MCU Control Register) is 0 by default.)

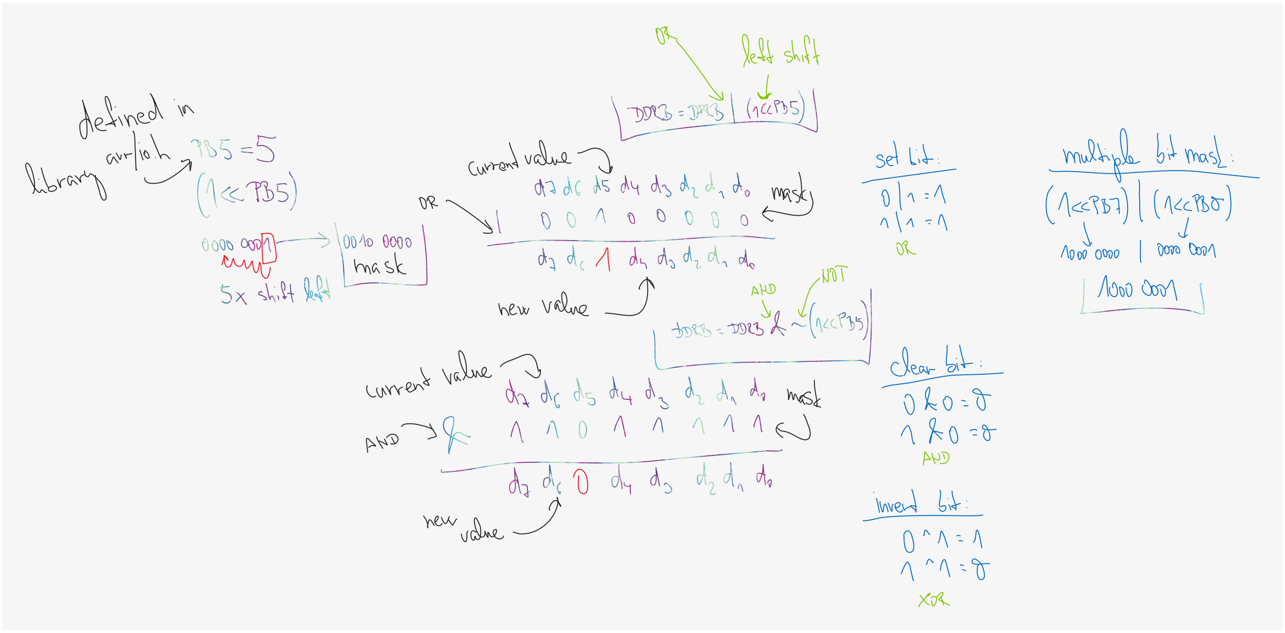
| DDRB | PORTB | Direction | Internal pull-up resistor | Description |
|---|---|---|---|---|
| 0 | 0 | input | no | Tri-state, high-impedance |
| 0 | 1 | | | |
| 1 | 0 | | | |

| DDRB | PORTB | Direction | Internal pull-up resistor | Description |
|---|---|---|---|---|
| 1 | 1 | | | |

3. To control individual bits, the following binary operations are used.

    1. `|` OR
    2. `&` AND
    3. `^` XOR
    4. `~` NOT
    5. `<<` binary shift to left

| b | a | b OR a | b AND a | b XOR a | NOT b |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 |



4. Comment Arduino-style defines and functions, use binary operations with control registers DDRB, PORTB and rewrite the application. Note the second-version size after the compilation.

```c
...
// #include "Arduino.h"
// #define PB5 13           // In Arduino world, PB5 is called "13"
// #define PB0 8

int main(void)
{
    uint8_t led_value = 0;  // Local variable to keep LED status

    // Set pins where LEDs are connected as output
    // Ver 1: Arduino style
    // pinMode(LED_GREEN, OUTPUT);
    // pinMode(LED_RED, OUTPUT);

    // Ver 2: Low-level (register) style
    DDRB = DDRB | (1<<LED_GREEN);
    ...
}
```

## Part 3: GPIO library files

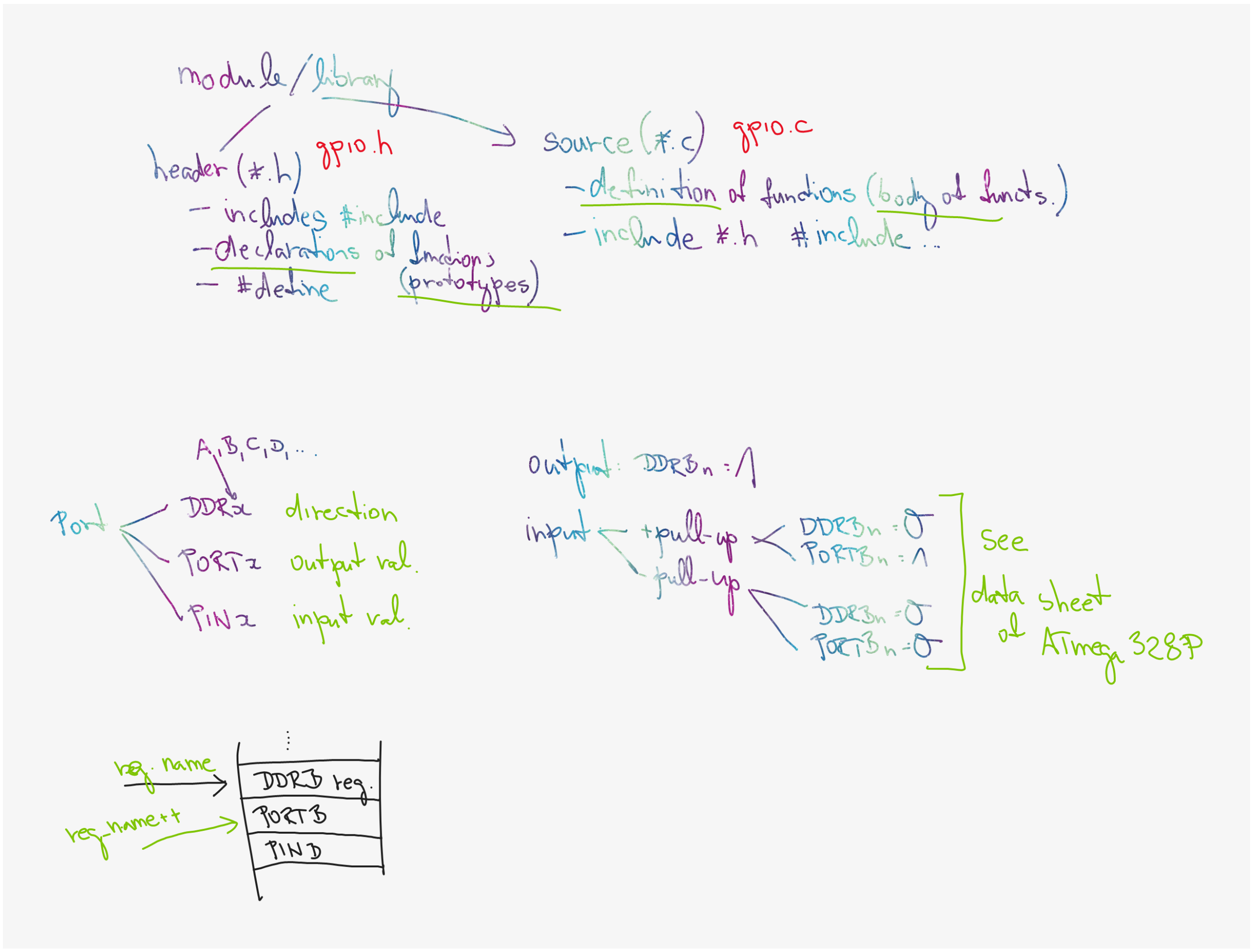
For clarity and efficiency of the code, the individual parts of the application in C are divided into two types of files: header files and source files. Note that together they form one module.

**Header file** is a file with extension `.h` and generally contains definitions of data types, function prototypes and C preprocessor commands. **Source file** has the extension `.c` and is used to implement the code. It is bad practice to mix usage of the two although it is possible.

C programs are highly dependent on functions. Functions are the basic building blocks of C programs and every C program is combination of one or more functions. There are two types of functions in C: **built-in functions** which are the part of C compiler and **user defined functions** which are written by programmers according to their requirement.

To use a user-defined function, there are three parts to consider:

- Function declaration or Function prototype (`*.h` file)
- Function definition (`*.c` file)
- Function call (`*.c` file)



*[A function prototype](https://www.programiz.com/c-programming/c-user-defined-functions) is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body. A **function prototype** gives information to the compiler that the function may later be used in the program.*

*A **function definition** contains the block of code to perform a specific task.*

*By **calling the function**, the control of the program is transferred to the function.*

A header file can be shared between several source files by including it with the C preprocessing directive `#include`. If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE_NAME         // Preprocessor directive allows for conditional compilation. If not defined.
# define HEADER_FILE_NAME        // Definition of constant within your source code.

// The body of entire header file

#endif                           // The #ifndef directive must be closed by an #endif
```

This construct is commonly known as a wrapper `#ifndef`. When the header is included again, the conditional will be false, because `HEADER_FILE_NAME` is already defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

1. In PlatformIO project, create a new folder `LAB2-GPIO_LIBRARY > lib > gpio`. Within this folder, create two new files `gpio.c` and `gpio.h`. See the project structure:

   ```
   LAB2-GPIO_LIBRARY   // PlatfomIO project
   ├── include         // No need this
   ├── lib             // Libraries
   │   └── gpio        // Our new GPIO library
   │       ├── gpio.c
   │       └── gpio.h
   ├── src             // Source file(s)
   │   └── main.c
   ├── test            // No need this
   └── platformio.ini  // Project Configuration File
   ```

   1. Copy/paste header file to `gpio.h`
   2. Copy/paste library source file to `gpio.c`
   3. Include header file to `src > main.c`:

      ```c
      #include <gpio.h>

      int main(void)
      {
          ...
      }
      ```

2. Go through both files and make sure you understand each line. The GPIO library defines the following functions.

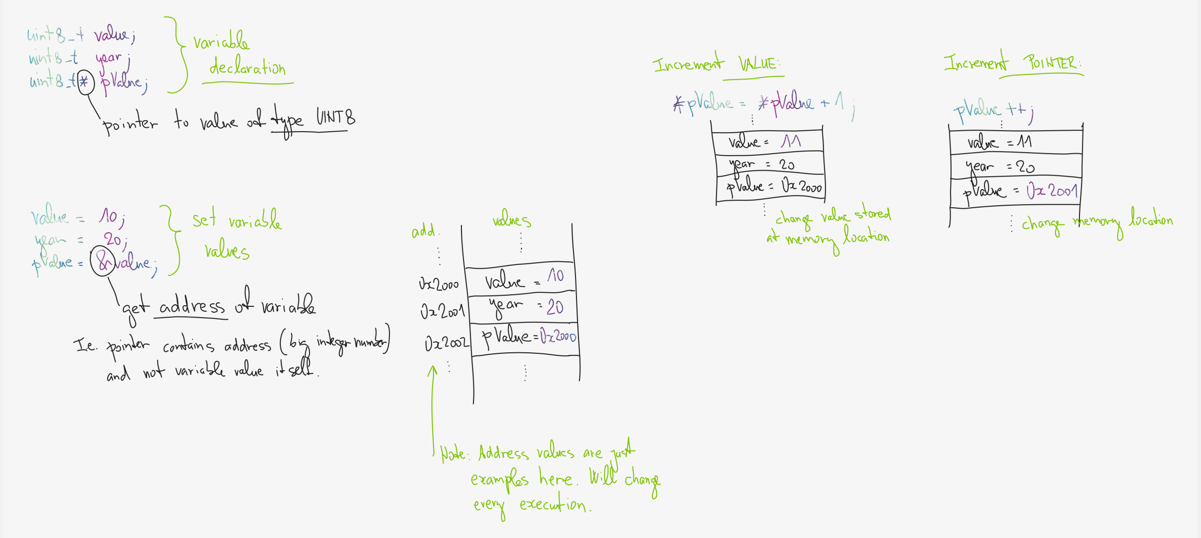
| **Return** | **Function name** | **Function parameters** | **Description** |
|---|---|---|---|
| `void` | `GPIO_mode_output` | `volatile uint8_t *reg, uint8_t pin` | Configure one output pin |
| `void` | `GPIO_mode_input_pullup` | `volatile uint8_t *reg, uint8_t pin` | Configure one input pin and enable pull-up resistor |
| `void` | `GPIO_write_low` | `volatile uint8_t *reg, uint8_t pin` | Write one pin to low value |
| `void` | `GPIO_write_high` | `volatile uint8_t *reg, uint8_t pin` | Write one pin to high value |

| Return | Function name | Function parameters | Description |
|---|---|---|---|
| `uint8_t` | `GPIO_read` | `volatile uint8_t *reg, uint8_t pin` | Read a value from input pin |

> **Note:** Suggestions for other features you can add are:
>
> - `void GPIO_mode_input_nopull(volatile uint8_t *reg, uint8_t pin)` Configure one input pin without pull-up resistor
> - `void GPIO_toggle(volatile uint8_t *reg, uint8_t pin)` Toggle one pin value

The register name parameter must be `volatile` to avoid a compiler warning. Note that the C notation `*variable` representing a pointer to memory location where the variable's **value** is stored. Notation `&variable` is address-of-operator and gives an **address** reference of variable.

```c
#include <gpio.h>  // Do not forget to include GPIO header file

int main(void)
{
    // Examples of various function calls
    GPIO_mode_output(&DDRB, LED_GREEN);  // Set output mode in DDRB
reg
    ...
    GPIO_write_low(&PORTB, LED_GREEN);   // Set output low in PORTB
reg
    ....
    temp = GPIO_read(&PIND, BTN);        // Read input value from PIND
reg
    ...
}
```



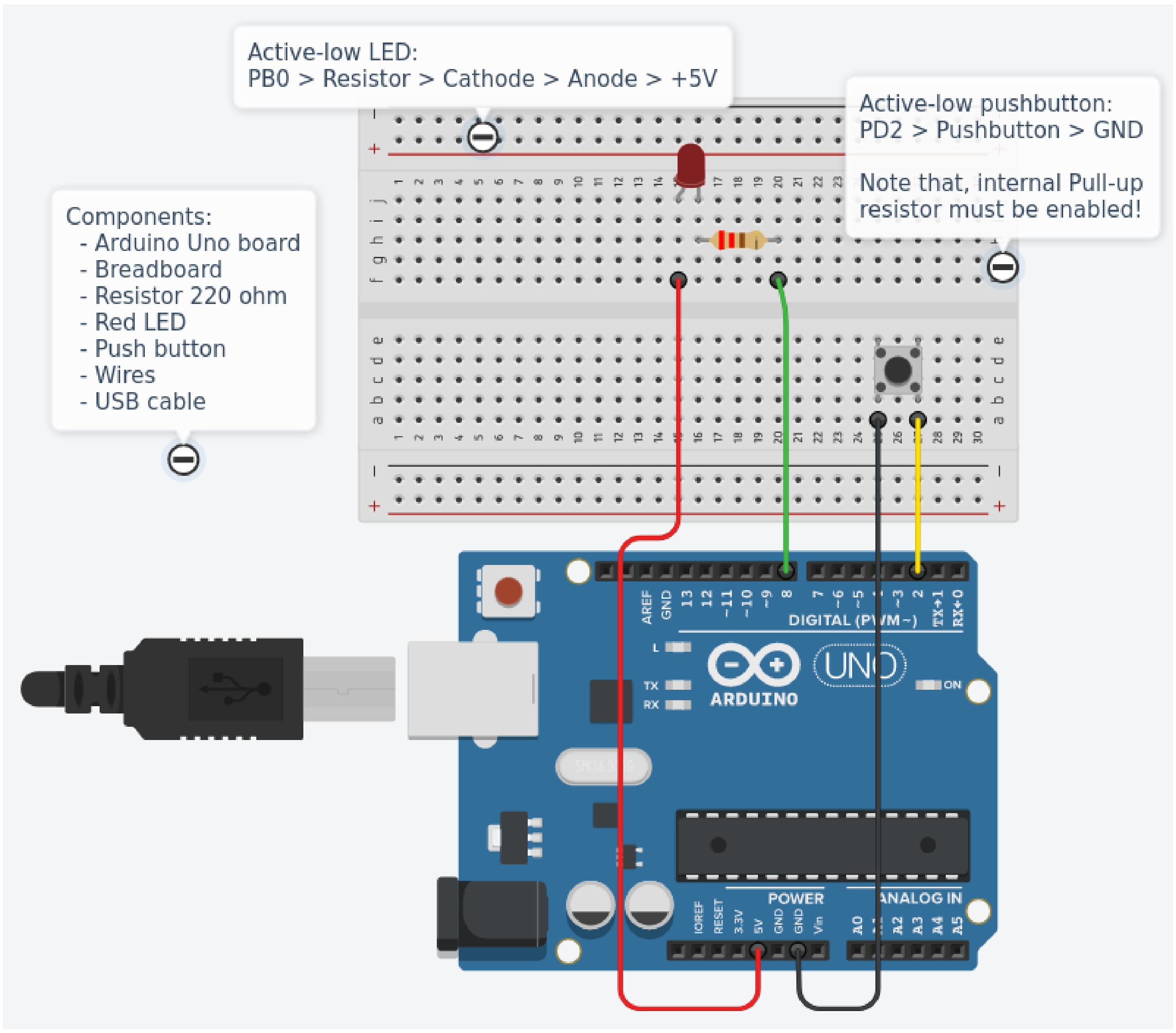
> **Note:** Understanding C Pointers: A Beginner's Guide is available here. Explanation of how to pass an IO port as a parameter to a function is given here.

3. In `main.c` comment binary operations with control registers (DDRB, PORTB) and rewrite the application with library functions. Note its size after the compilation as third-verion. Try to optimize code to the most effective way.

   ```c
   #include <gpio.h>

   int main(void)
   {
       uint8_t led_value = 0;  // Local variable to keep LED status

       // Set pins where LEDs are connected as output
       // Ver 1: Arduino style
       // pinMode(LED_GREEN, OUTPUT);
       // pinMode(LED_RED, OUTPUT);

       // Ver 2: Low-level (register) style
       // DDRB = DDRB | (1<<LED_GREEN);
       // DDRB = DDRB | (1<<LED_RED);

       // Ver 3: Library function style
       GPIO_mode_output(&DDRB, LED_GREEN);
       ...
   }
   ```

4. On a breadboard, connect a two-color LED (3-pin LED) and resistors to pins PB2 and PB3. Develop the code to achieve alternating blinking of two LEDs.

5. (Optional) On a breadboard, connect an active-low push button to pin PD2. In your code, activate the internal pull-up resistor on this pin. Make the LEDs blink only when the button is pressed.



6. After completing your work, ensure that you synchronize the contents of your working folder with both the local and remote repository versions. This practice guarantees that none of your changes are lost. You can achieve this by using **Source Control (Ctrl+Shift+G)** in Visual Studio Code or by utilizing Git commands.

   > **Help:** Useful git commands are `git status` - Get state of working directory and staging area. `git add` - Add new and modified files to the staging area. `git commit` - Record changes to the local repository. `git push` - Push changes to remote repository. `git pull` - Update local repository and working folder. Note that, a brief description of useful git commands can be found here and detailed description of all commands is here.

## (Optional) Experiments on your own

1. Complete declarations (`*.h`) and definitions (`*.c`) of GPIO suggested functions `GPIO_mode_input_nopull()` and `GPIO_toggle()`.

2. Connect at least five LEDs and one push button to the microcontroller and program an application in Knight Rider style. When you press and release a push button once, the LEDs starts to switched on and off; ensure that only one of LEDs is switched on at a time. Do not implement the blinking speed changing.

3. Simulate the Knight Rider application in SimulIDE.

4. Draw a schematic of Knight Rider application. The image can be drawn on a computer or by hand. Always name all components, their values and pin names!

5. Finish all experiments, upload them to your GitHub repository, and submit the project link via BUT e-learning. The deadline for submitting the assignment is the day prior to the next lab session, which is one week from now.

## References

1. Parewa Labs Pvt. Ltd. C User-defined functions

2. Understanding C Pointers: A Beginner's Guide

3. avr-libc. How do I pass an IO port as a parameter to a function?

4. Tomas Fryza. Useful Git commands

5. Goxygen commands

6. LEDnique. LED pinouts - 2, 3, 4-pin and more