Digitální elektronika 2

Laboratorní cvičení (v angličtině)

Garant:
doc. Ing. Tomáš Frýza, PhD.

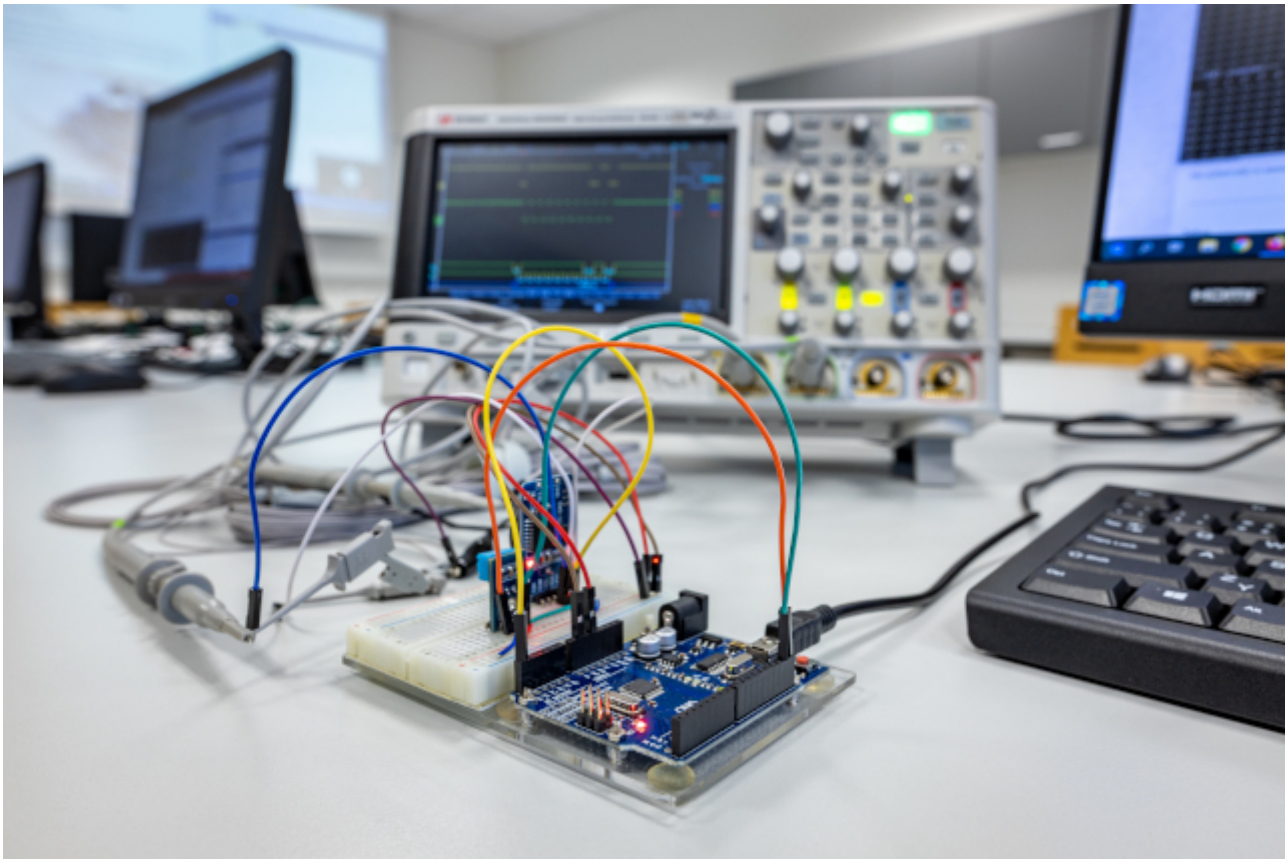
Autor:
doc. Ing. Tomáš Frýza, PhD.

Brno 15. 10. 2023

Vytvořeno za podpory RP182302002, PPSŘ 2023

Digital electronics 2

The repository contains AVR lab exercises for bachelor course *Digital Electronics 2* at Brno University of Technology, Czechia. Arduino Uno board and some shields are used as the main programming platform.



Exercises

1. Git version-control system, AVR tools
2. Control of GPIO pins
3. Timers
4. Liquid Crystal Display (LCD)
5. Analog-to-Digital Converter (ADC)
6. Universal Asynchronous Receiver-Transmitter (UART)
7. Inter-Integrated Circuits (I2C)
8. Assembly language and project documentation

List of examples

- Basic C template, blink a LED
- Project documentation with Doxygen
- Autonomous slot car

Components

The following hardware and software components are mainly used in the lab.

- Devices:
  - ATmega328P 8-bit microcontroller: AVR Instruction Set Manual
- Boards and shields:
  - Schematics
  - Arduino Uno board
  - LCD and keypad shield with LCD and five push buttons
  - Multi-function shield with four LEDs, three push buttons, four seven-segment displays, and others
- Sensors and modules:
  - DHT12 I2C humidity and temperature sensor: data sheet
  - DS3231 I2C real time clock: data sheet
  - HC-SR04 ultrasonic sensor
  - Analog joystick PS2
  - ESP8266 Wi-Fi module: AT commands
- Analyzers:
  - 24MHz 8-channel logic analyzer: software
  - Oscilloscope Keysight Technologies DSOX3034T (350 MHz, 4 analog channels), including 16 logic timing channels DSOXT3MSO and serial protocol triggering and decode options D3000BDLA
- Development tools:
  - Visual Studio Code
  - PlatformIO
  - Atmel Studio 7 (Microchip Studio 7)
  - GCC Compilers for AVR
- Other tools:
  - SimulIDE, real time electronic circuit simulator. With PIC, AVR and Arduino simulation
  - git

References

1. How to use AVR template with PlatformIO
2. How to use AVR template on Windows
3. How to use AVR template on Linux
4. Peter Fleury, AVR-GCC libraries
5. Wykys, Tools for development of AVR microcontrollers
6. Barr Group, Embedded C Coding Standard
7. 4Geeks. How to use Gitpod

Lab 1: Git version-control system, AVR tools

Learning objectives

After completing this lab you will be able to:

- Use markdown README files
- Create git repository
- Understand basic structure of C files
- Compile and download firmware to AVR device
- Use breadboard and connect electronic devices to AVR pins

The purpose of this laboratory exercise is to learn how to use the `git` versioning system, write the markdown readme file, learn the basic structure of C code, and how to use development tools to program ATmega328P microcontroller on the Arduino Uno board.

Table of contents

Components list

- Arduino Uno board, USB cable
- Breadboard
- 2 LEDs
- 2 resistors
- Jumper wires
- Logic analyzer

Pre-Lab preparation

1. If you don't have any, create a free account on GitHub.
2. For future synchronization of local folders with GitHub, download and install git. Git is free, open source, and available on Windows, Mac, and Linux platforms. Window users may also need to use the Git Bash application (installed automatically with git) for command line operations.
3. (Optional) Download and install SimulIDE electronic circuit simulator.
4. (Optional) If you have option to use Arduino Uno board and logic analyzer, also download and install Saleae Logic 1.

Part 1: GitHub

GitHub serves as a platform for hosting code, facilitating collaboration, and managing version control. It enables you and your collaborators to work together on projects, retain a history of all prior changes, create distinct branches, and offers a multitude of additional features.

1. In GitHub, create a new public repository titled **digital-electronics-2**. Initialize a README, C template `.gitignore`, and MIT license.

2. Use any available Git manuals, such as Markdown Guide, Basic Syntax and add the following sections to your README file.

   - Headers H1, H2, H3
   - Emphasis (*italics*, **bold**)
   - Lists (ordered, unordered)
   - Links
   - Table
   - Listing of C source code (with syntax highlighting)

3. Use your favorite file manager and run Git Bash (Windows) or Terminal (Linux) inside your home folder `Documents`.

4. With help of Git command, clone a local copy of your public repository.

   > **Important:** To avoid future problems, never use national characters (such as éščřèêö, ...) and spaces in folder- and file-names.
   >
   > **Help:** Useful git command is `git clone` - Create a local copy of remote repository. This command is executed just once; later synchronization between remote and local repositories is performed differently.
   >
   > Useful bash commands are `cd` - Change working directory. `mkdir` - Create directory. `ls` - List information about files in the current directory. `ls -a` - List information aout all files in the current directory. `pwd` - Print the name of the current working directory.

   ```
   ## Windows Git Bash or Linux:
   $ git clone https://github.com/your-github-account/digital-
   electronics-2
   $ cd digital-electronics-2/
   $ ls -a
   .gitignore  LICENSE  README.md
   ```

5. Set username and email for your repository (values will be associated with your later commits):

   ```
   $ git config user.name "your-git-user-name"
   $ git config user.email "your-email@address.com"
   ```

   You can verify that the changes were made correctly by:

```
$ git config --list
```

Part 2: Install and test AVR tools

1. Run Visual Studio Code, follow these instructions and install the PlatformIO plugin.

2. Create a new project `lab1-blink_arduino`, select `Arduino Uno` board, and change project location to your local repository folder `Documents/digital-electronics-2`. Copy/paste blink example code to your `LAB1-BLINK_ARDUINO > src > main.cpp` file.

3. IMPORTANT: Rename `LAB1-BLINK_ARDUINO > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

   The final project structure should look like this:

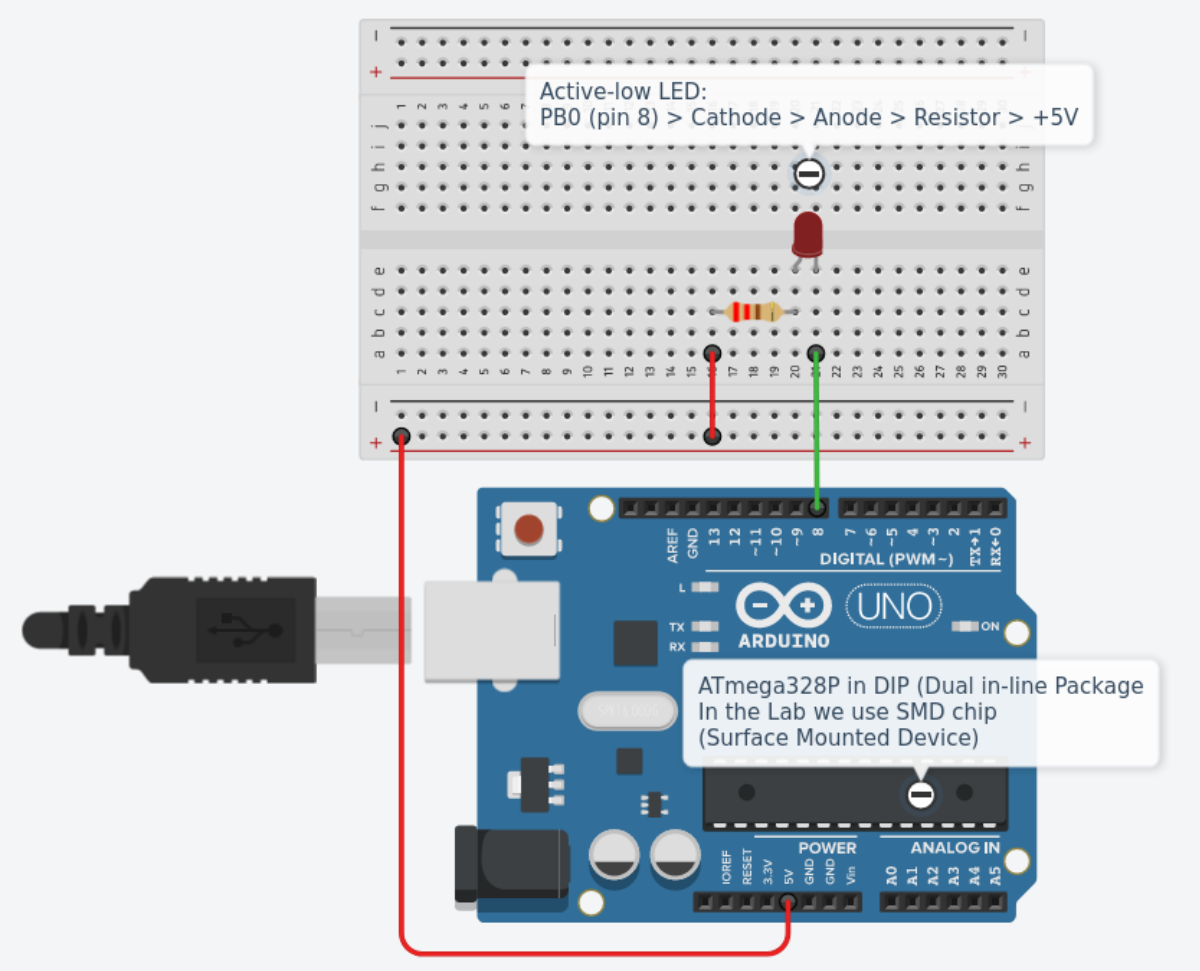
   ```
   LAB1-BLINK_ARDUINO  // PlatfomIO project
   ├── include         // Included files
   ├── lib             // Libraries
   ├── src             // Source file(s)
   │   └── main.c
   ├── test            // No need this
   └── platformio.ini  // Project Configuration File
   ```

4. Compile and download the firmware to target ATmega328P microcontroller. Go through all the lines of code and make sure you understand their function. Change the delay duration and observe the behavior of on-board LED.

   - See Arduino Uno pinout
   - See Arduino Docs for GPIO / Pin Management

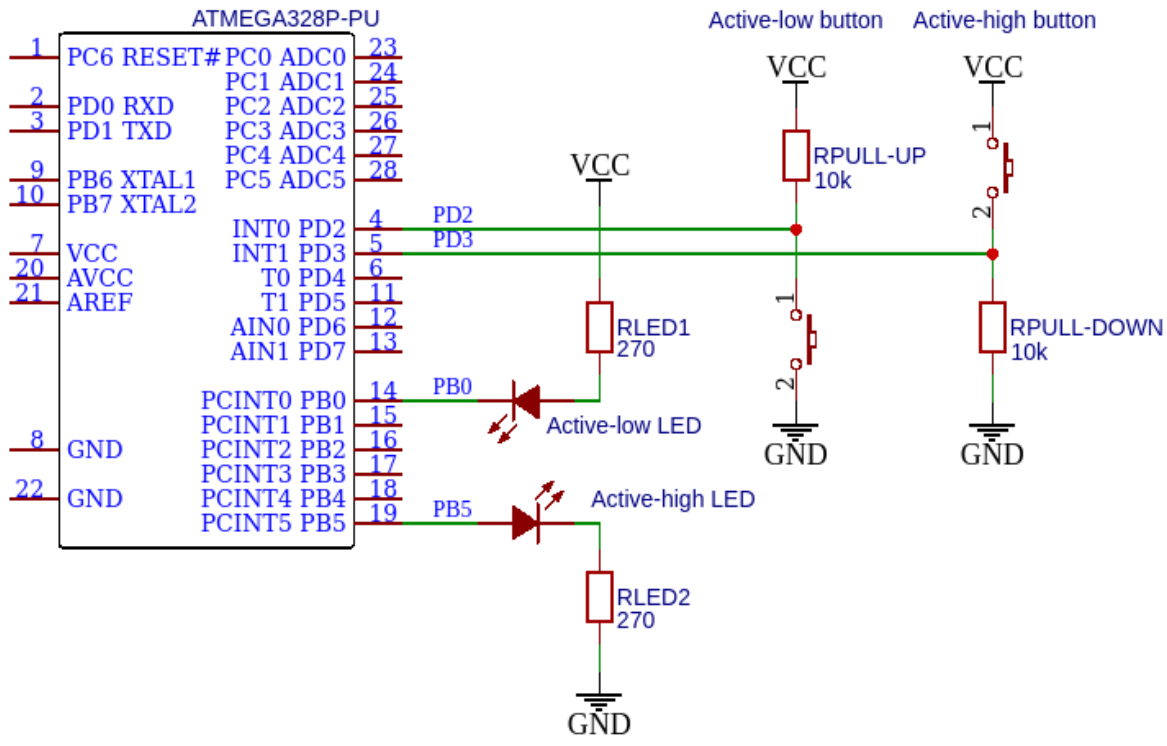
5. Use breadboard, wires, resistor, and a second LED. Connect it to a GPIO pin PB0 in active-low way and modify your code to blink both LEDs.

   - See this breadboard description or that one
   - See LED resistor value calculation
   - Connection of external LED in active-low way:



> **Note:** Picture was created by Autodesk Tinkercad.

- General connections of LEDs and push buttons in active-low and active-high way:

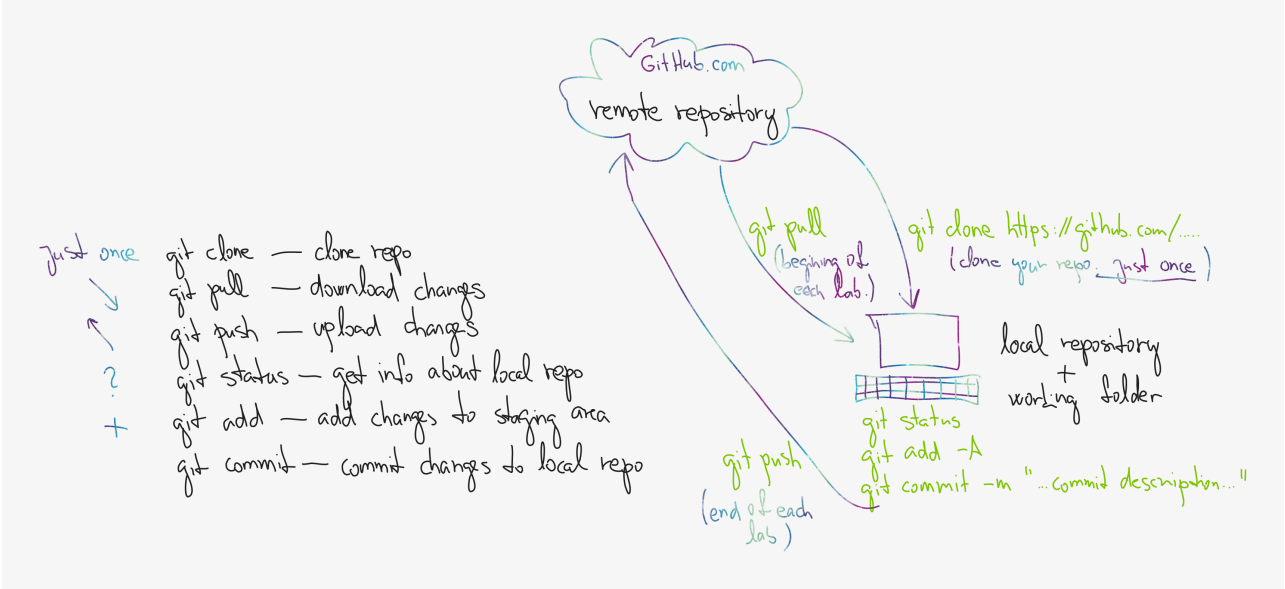


> **Note:** Picture was created by EasyEDA.

6. On breadboard, connect two LEDs and control them by a single output pin PB0. Is it possible to get all commbinations, i.e. ON+ON, ON+OFF, OFF+ON, and OFF+OFF?

7. 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 to add, commit, and push all local changes to your remote repository. Check GitHub web page for changes.

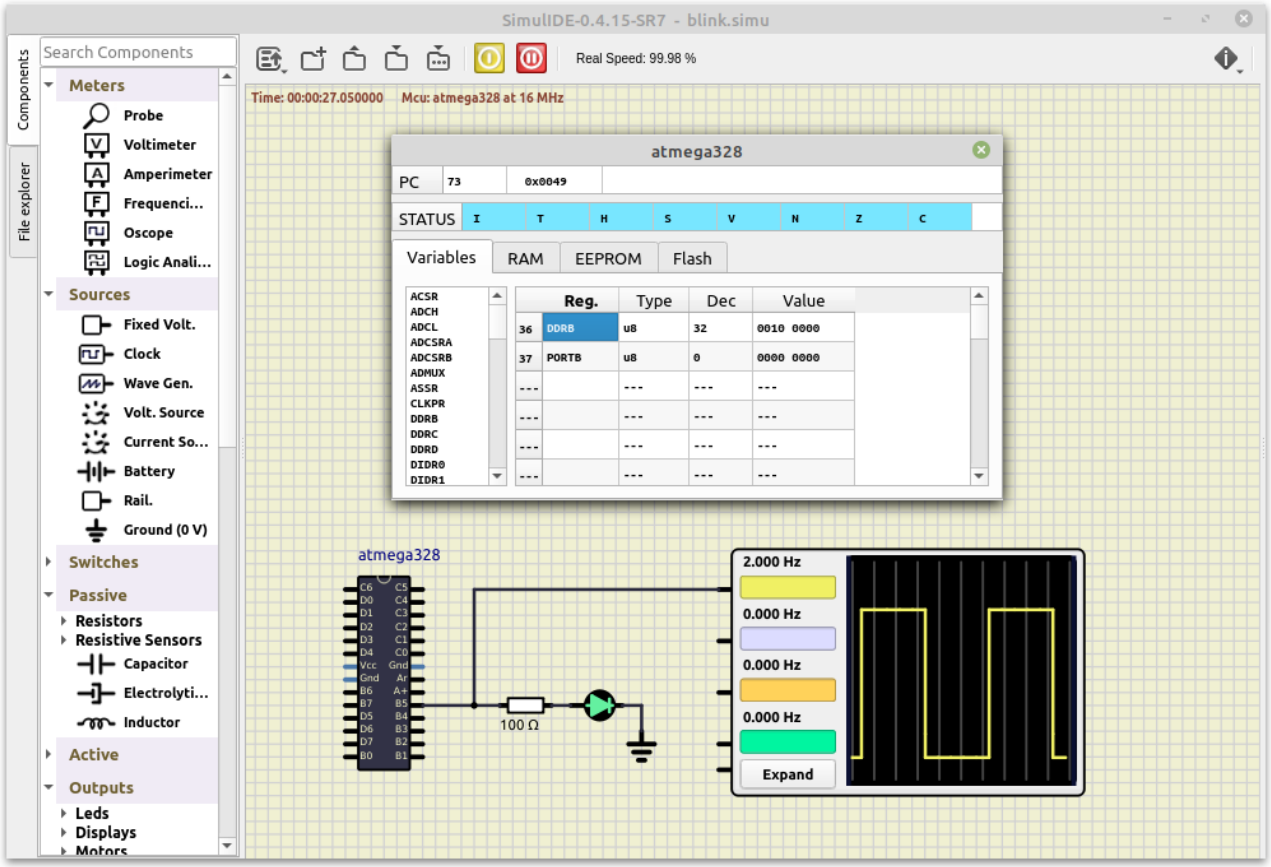
> **Help:** Useful git commands are `git status` - Get state of working directory and staging area. `git add` - Add new and modified files to the staging area. `git commit` - Record changes to the local repository. `git push` - Push changes to remote repository. `git pull` - Update local repository and working folder. Note that, a brief description of useful git commands can be found here and detailed description of all commands is here.
>
> ```
> ## Windows Git Bash or Linux:
> $ git status
> $ git add -A
> $ git status
> $ git commit -m "Creating lab1-blink program"
> $ git status
> $ git push
> $ git status
> ```



(Optional) Part 3: SimulIDE electronic circuit simulator

1. Run SimulIDE, use online tutorials, and create a circuit with ATmega328 AVR microcontroller.

2. All circuit and control elements are available in the **Components** tab. Use the following components: ATmega328 (**Micro > AVR > atmega > atmega328**), resistor (**Passive > Resistors > Resistor**), LED (**Outputs > Leds > Led**), and GND (**Sources > Ground (0 V)**) and connect them as shown.



3. Right-click on the ATmega package and select **Load firmware**. In your project folder, find the `*.hex` file that was created by the previous compilation process.

4. Right-click on the ATmega package and select **Open Mcu Monitor** to view internal registers and memory contents. Select **Variable** folder. In the **Reg.** column, type `DDRB` on the first line and `PORTB` on the second.

5. Click to **Power Circuit** button at the top of the window to simulate the project and monitor the LED status and register values. The simulation can be paused with the **Pause Simulation** button and stopped by pressing the **Power Circuit** button again.

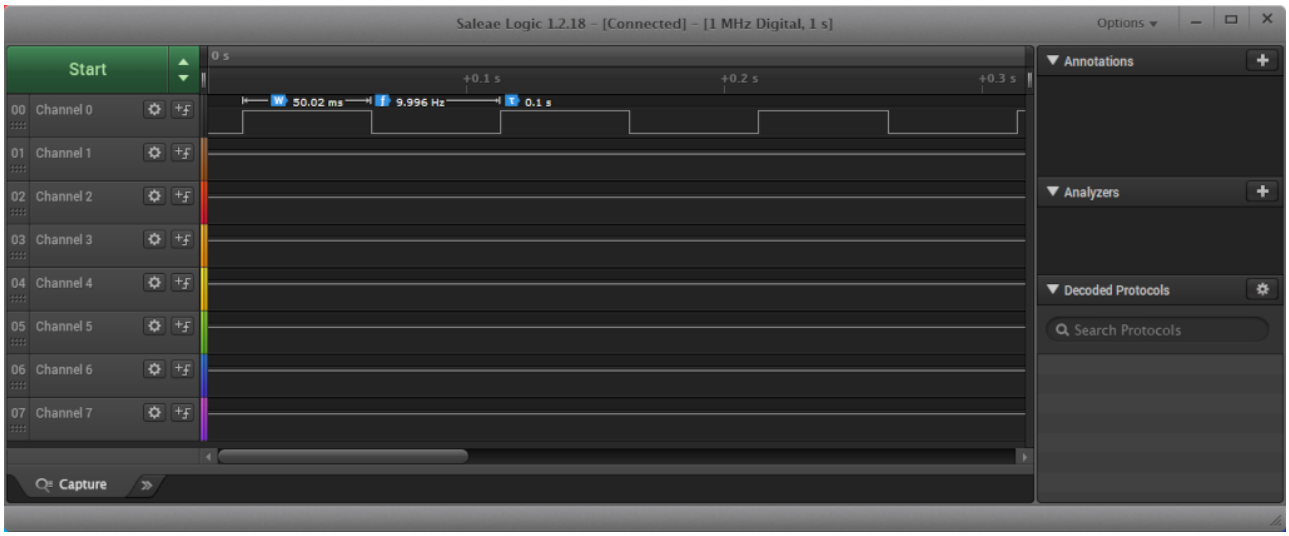
6. You can connect a probe (**Meters > Probe**), an oscilloscope (**Meters > Oscope**), a voltmeter (**Meters > Voltimeter**), or a frequency meter (**Meters > Frequencimeter**) to output B5 and observe the signal.

7. Properties of individual components can be found/changed by right-clicking on the component and selecting **Properties**.

(Optional) Part 4: Logic analyzer

1. Run Saleae Logic software, use wire and connect Channel 0 to Arduino board pin 13 (pin PB5 is connected here), and verify the duration of delay function.

2. To start sampling, press the green button with two arrows, set the sampling rate to 1 MS/s and the recording time to 1 second. Click the Start button.



(Optional) Experiments on your own

1. Install the AVR development tools on your computer.
2. Modify the code from `lab1-blink_arduino` example and build an application that will repeatedly trasnmit the string `PARIS` on a LED in the Morse code. Choose the duration of "dot" and "dash" so that they are visible during the simulation and/or implementation. Note that the proper Morse code timing is explained here.
3. Simulate the Morse code application in SimulIDE.
4. Draw a schematic of Morse code application, i.e. connection of AVR device, two LEDs (one in active-high, second in active-low way), two resistors, and supply voltage. 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. MIT license
2. Markdown Guide, Basic Syntax
3. GitHub, Inc. Mastering Markdown
4. Tomas Fryza. Useful Git commands
5. Joshua Hibbert. Git Commands
6. Stephen C. Phillips. Morse Code Timing
7. Science Buddies. How to Use a Breadboard for Electronics and Circuits

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	

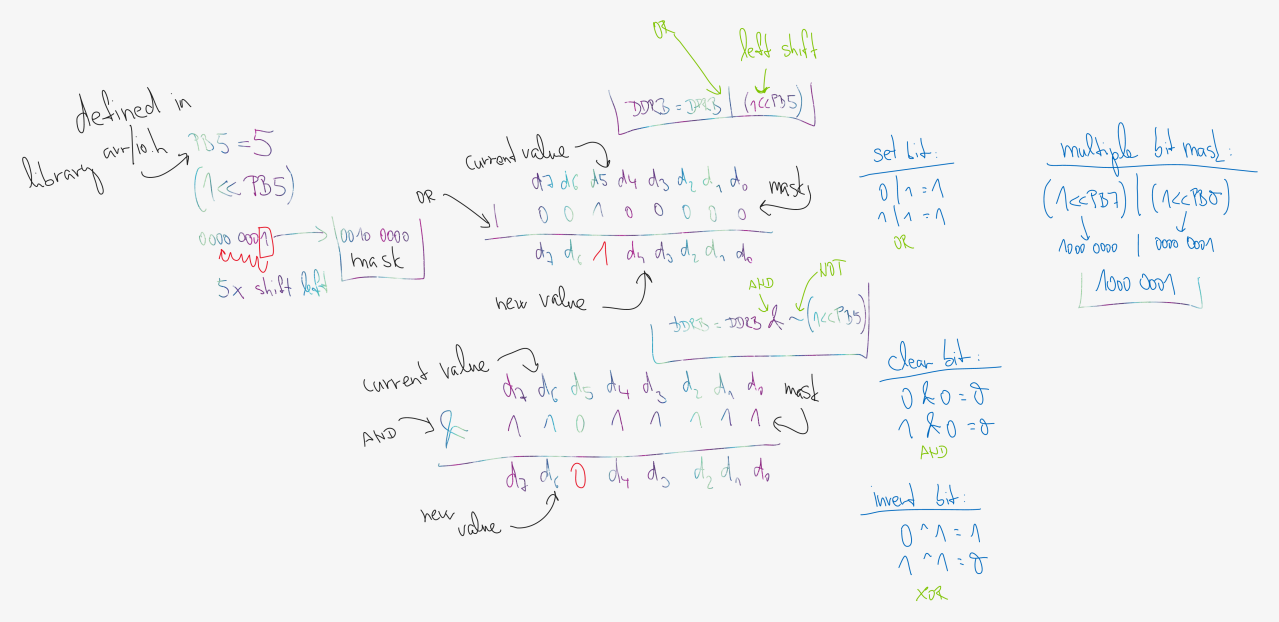
2. Use the datasheet to find out the meaning of the DDRB and PORTB control register values and their combinations. (Let PUD (Pull-up Disable) bit in MCUCR (MCU Control Register) is 0 by default.)

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.

```
...
// #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

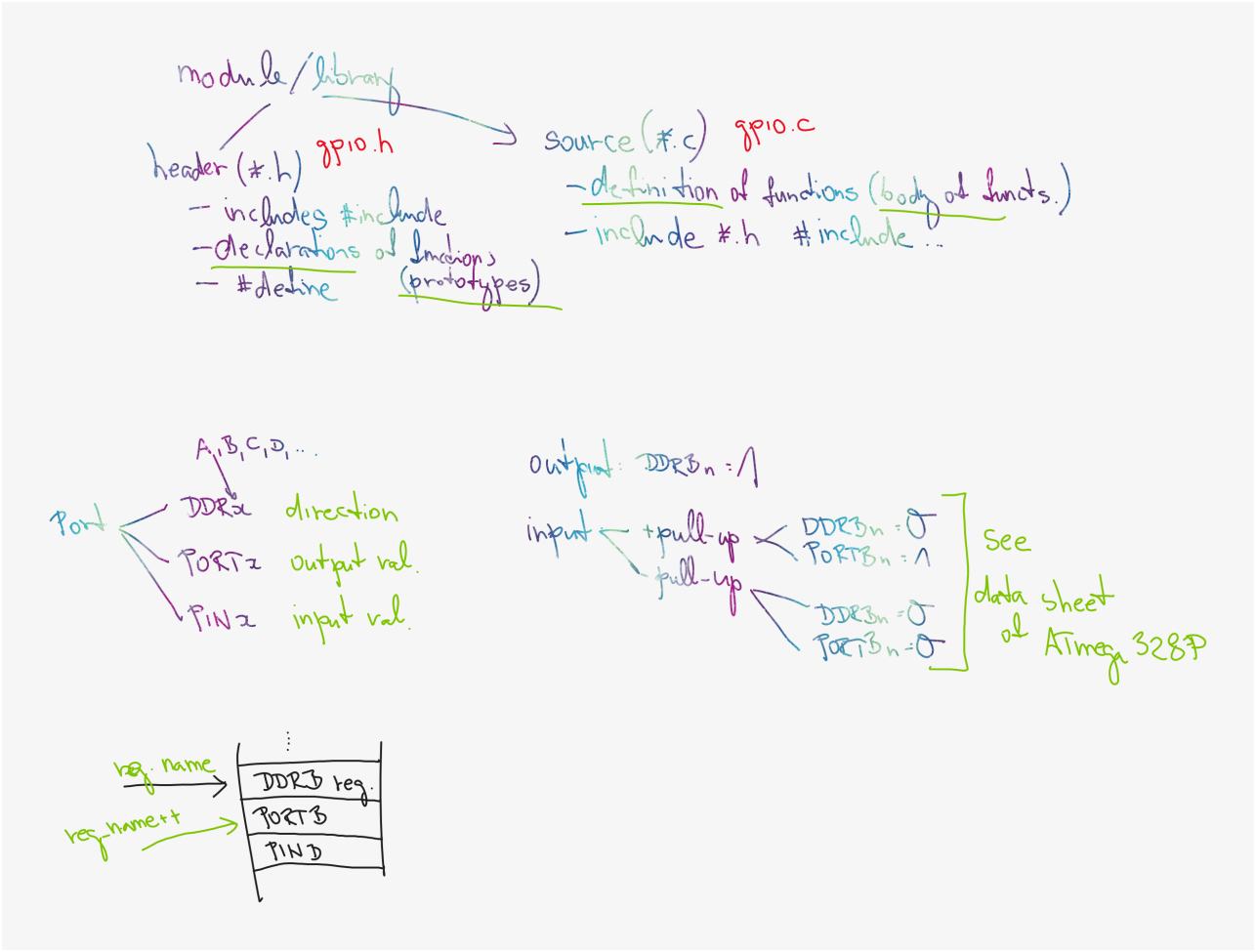
For clarity and efficiency of the code, the individual parts of the application in C are divided into two types of files: header files and source files. Note that together they form one module.

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 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`:

```
#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.

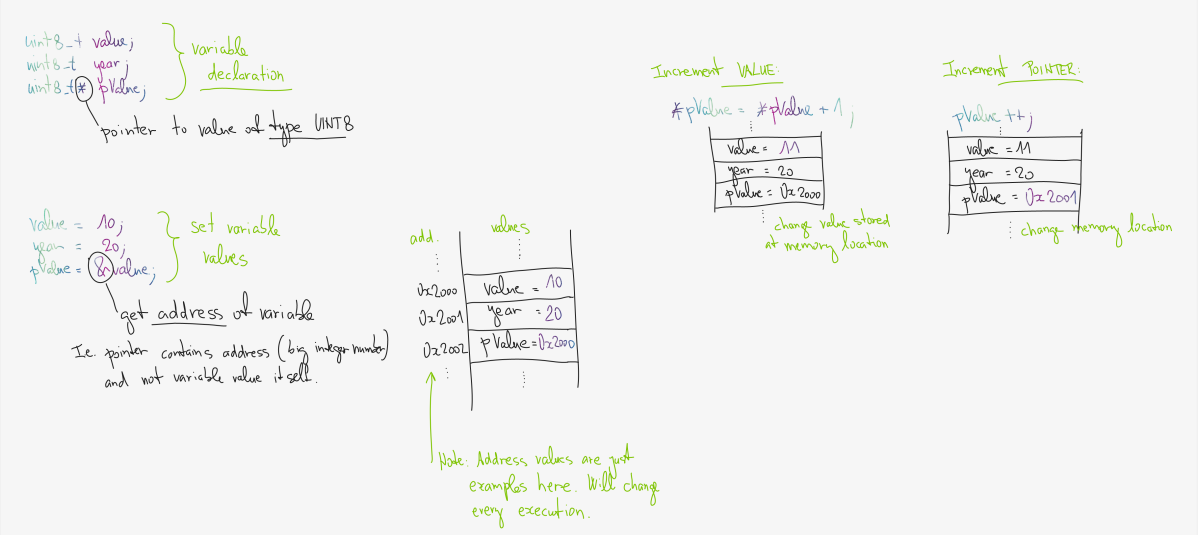
Return	Function name	Function parameters	Description
`void`	`GPIO_mode_output`	`volatile uint8_t *reg, uint8_t pin`	Configure one output pin
`void`	`GPIO_mode_input_pullup`	`volatile uint8_t *reg, uint8_t pin`	Configure one input pin and enable pull-up resistor
`void`	`GPIO_write_low`	`volatile uint8_t *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.

```
#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
    ...
}
```



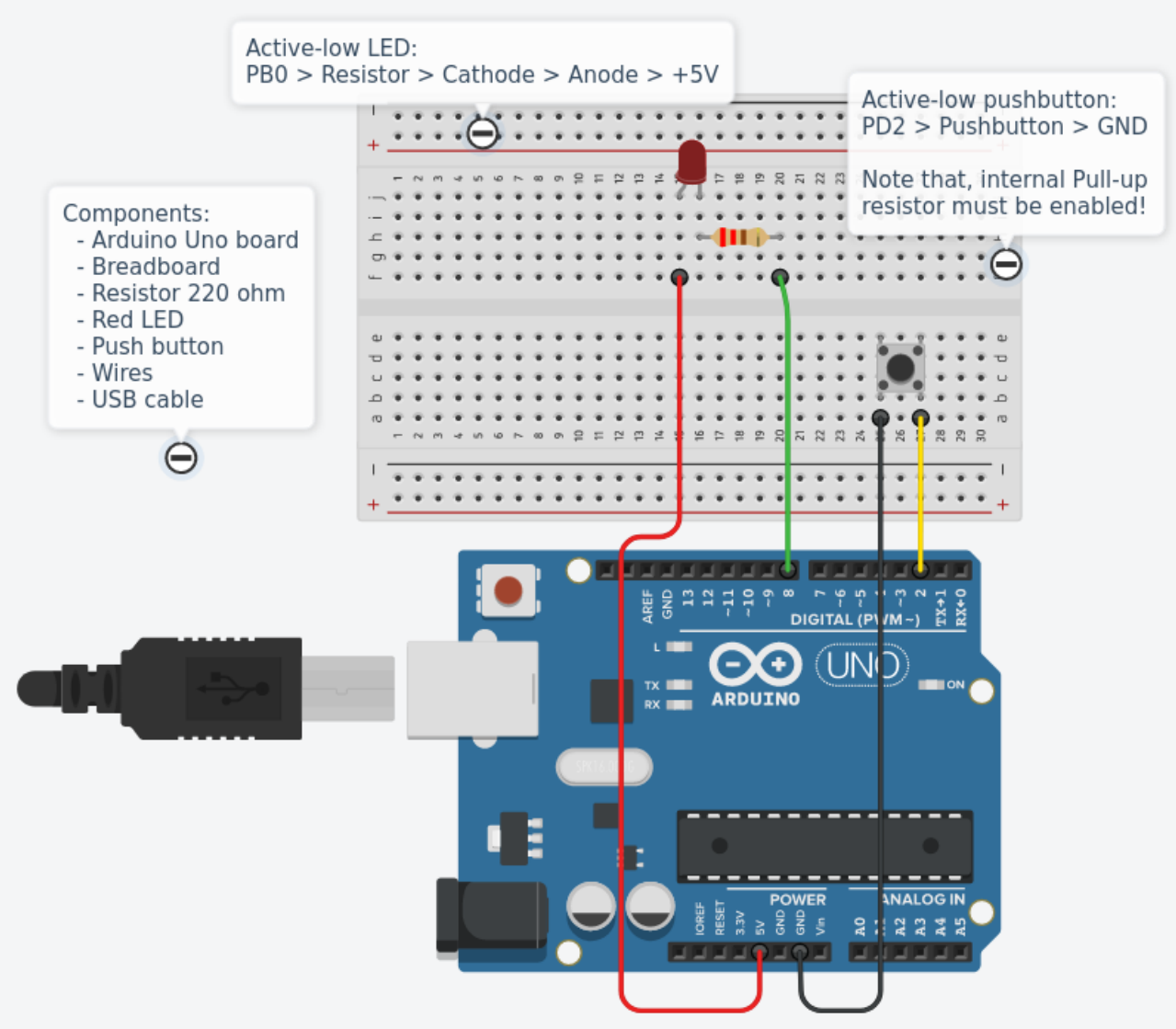
> **Note:** Understanding C Pointers: A Beginner's Guide is available here. Explanation of how to pass an IO port as a parameter to a function is given here.

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

Lab 3: Timers

Learning objectives

After completing this lab you will be able to:

- Use `#define` compiler directives
- Use internal microcontroller timers
- Understand overflow
- Combine different interrupts

The purpose of the laboratory exercise is to understand the function of the interrupt, interrupt service routine, and the functionality of timer units. Another goal is to practice finding information in the MCU manual; specifically setting timer control registers.

Table of contents

Components list

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

Pre-Lab preparation

Consider an *n*-bit number that we increment based on the clock signal. If we reach its maximum value and try to increase it, the value will be reset. We call this state an **overflow**. The overflow time depends on the frequency of the clock signal, the number of bits, and on the prescaler value:

$$t_{OVF} = \frac{1}{f_{CPU}} \cdot 2^{nbit} \cdot prescaler$$

> **Note:** The equation was generated by Online LaTeX Equation Editor using the following code.
>
> ```
> t_{OVF} = \frac{1}{f_{CPU}}\cdot 2^{nbit}\cdot prescaler
> ```

1. Calculate the overflow times for three Timer/Counter modules that contain ATmega328P if CPU clock frequency is 16 MHz. Complete the following table for given prescaler values. Note that, Timer/Counter2 is able to set 7 prescaler values, including 32 and 128 and other timers have only 5 prescaler values.

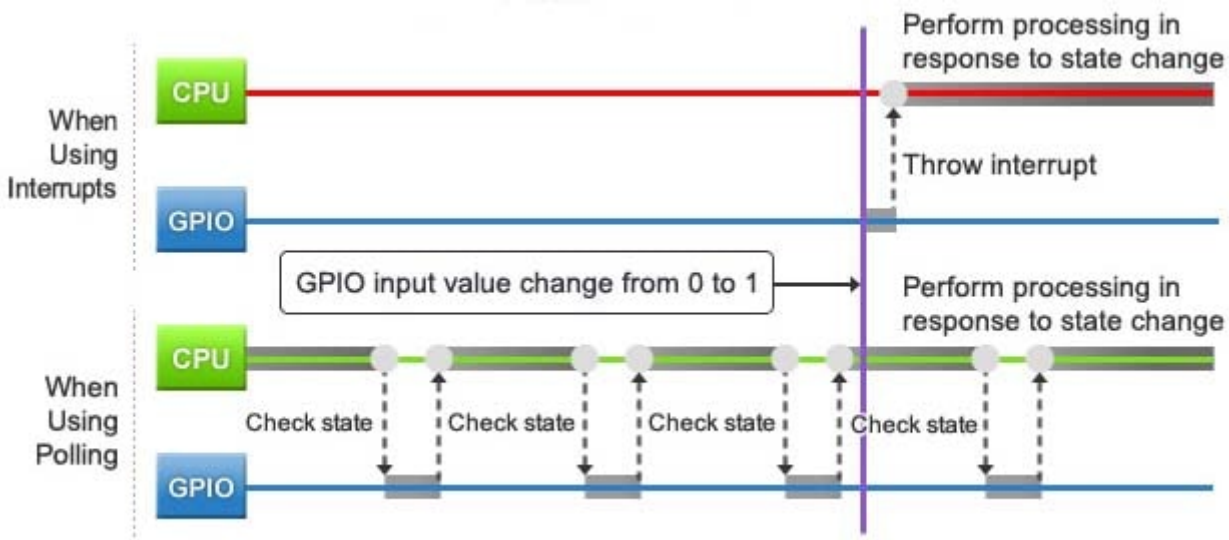
Module	Number of bits	1	8	32	64	128	256	1024
Timer/Counter0	8	16u	128u	--		--		
Timer/Counter1	16			--		--		
Timer/Counter2	8							

Part 1: Polling and interrupts

The state of continuous monitoring of any parameter is called **polling**. The microcontroller keeps checking the status of other devices; and while doing so, it does no other operation and consumes all its processing time for monitoring [3].

While polling is a straightforward method for monitoring state changes, it comes with a trade-off. If the polling interval is too long, there may be a significant delay between the occurrence and detection of a state change, potentially leading to missing the change entirely if the state reverts before the next check. On the other hand, a shorter interval provides quicker and more dependable detection, but it also consumes considerably more processing time and power, as there are more unsuccessful checks.

An alternative approach is to employ **interrupts**. In this approach, a state change triggers an interrupt signal that prompts the CPU to pause its current operation (while preserving its current state), execute the interrupt-related processing, and subsequently restore its prior state before resuming from where it had been interrupted.



An interrupt is a fundamental feature of a microcontroller. It represents a signal sent to the processor by hardware or software, signifying an event that requires immediate attention. When an interrupt is triggered, the controller finishes executing the current instruction and proceeds to execute an **Interrupt Service Routine (ISR)** or Interrupt Handler. ISR tells the processor or controller what to do when the interrupt occurs [4]. After the interrupt code is executed, the program continues exactly where it left off.

Interrupts can be set up for events such as a counter's value, a pin changing state, receiving data through serial communication, or when the Analog-to-Digital Converter has completed the conversion process.

See the ATmega328P datasheet (section **Interrupts > Interrupt Vectors in ATmega328 and ATmega328P**) for sources of interruptions that can occur on ATmega328P. Complete the selected interrupt sources in the following table. The names of the interrupt vectors in C can be found in C library manual.

Program address	Source	Vector name	Description
0x0000	RESET	--	Reset of the system
0x0002	INT0	`INT0_vect`	External interrupt request number 0
	INT1		
	PCINT0		
	PCINT1		
	PCINT2		
	WDT		
	TIMER2_OVF		
0x0018	TIMER1_COMPB	`TIMER1_COMPB_vect`	Compare match between Timer/Counter1 value and channel B compare value
0x001A	TIMER1_OVF	`TIMER1_OVF_vect`	Overflow of Timer/Counter1 value
	TIMER0_OVF		
	USART_RX		
	ADC		
	TWI		

All interrupts are disabled by default. If you want to use them, you must first enable them individually in specific control registers and then enable them centrally with the `sei()` command (Set interrupt). You can also centrally disable all interrupts with the `cli()` command (Clear interrupt).

Part 2: Synchronize repositories and create a new project

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 `lab3-timers` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB3-TIMERS > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: Timer overflow

A timer (or counter) is an integral hardware component in a microcontroller unit (MCU) designed for measuring time-based events. The ATmega328P MCU features three timers, designated as Timer/Counter0, Timer/Counter1, and Timer/Counter2. Timer0 and Timer2 are 8-bit timers, whereas Timer1 is a 16-bit timer.

The counter increments in alignment with the microcontroller clock, ranging from 0 to 255 for an 8-bit counter or 65,535 for a 16-bit counter. If counting continues, the timer value overflows to the default value of zero. Various clock sources can be designated for each timer by utilizing a CPU frequency divider equipped with predetermined prescaler values, including 8, 64, 256, 1024, and other options.

1. The timer modules can be configured with several special purpose registers. According to the ATmega328P datasheet (eg in the **8-bit Timer/Counter0 with PWM > Register Description** section), which I/O registers and which bits configure the timer operations?

Module	Operation	I/O register(s)	Bit(s)
Timer/Counter0	Prescaler 8-bit data value Overflow interrupt enable		
Timer/Counter1	Prescaler 16-bit data value Overflow interrupt enable	TCCR1B TCNT1H, TCNT1L TIMSK1	CS12, CS11, CS10 (000: stopped, 001: 1, 010: 8, 011: 64, 100: 256, 101: 1024) TCNT1[15:0] TOIE1 (1: enable, 0: disable)
Timer/Counter2	Prescaler 8-bit data value Overflow interrupt enable		

2. Copy/paste template code to `LAB3-TIMERS > src > main.c` source file.

3. In PlatformIO project, create a new folder `LAB3-TIMERS > lib > gpio`. Copy your GPIO library files `gpio.c` and `gpio.h` from the previous lab to this folder.

4. In PlatformIO project, create a new file `LAB3-TIMERS > include > timer.h`. Copy/paste header file to `timer.h`. See the final project structure:

```
LAB3-TIMERS          // PlatfomIO project
├── include          // Included file(s)
│   └── timer.h
├── lib              // Libraries
│   └── gpio         // Your GPIO library
│       ├── gpio.c
│       └── gpio.h
├── src              // Source file(s)
│   └── main.c
├── test             // No need this
└── platformio.ini   // Project Configuration File
```

To simplify the configuration of control registers, we defined Timer/Counter1 macros with meaningful names in the `timer.h` file. Because we only define macros and not function bodies, the `timer.c` source file is **not needed** this time!

5. Go through the files and make sure you understand each line. Build and upload the code to Arduino Uno board. Note that `src > main.c` file contains the following:

```
#include <avr/interrupt.h>  // Interrupts standard C library for AVR-
GCC
#include <gpio.h>           // GPIO library for AVR-GCC
#include "timer.h"          // Timer library for AVR-GCC

int main(void)
{
    ...
    // Enable overflow interrupt
    TIM1_OVF_ENABLE
    ...
    // Enables interrupts by setting the global interrupt mask
    sei();
    ...
}

// Interrupt service routines
ISR(TIMER1_OVF_vect)
{
    ...
}
```

6. In `timer.h` header file, define similar macros also for Timer/Counter0 and Timer/Counter2. On a breadboard, connect a two-color LED (3-pin LED) or two LEDs and resistors to pins PB2 and PB3. Modify `main.c` file, and use three interrupts for controlling all three LEDs (one on-board and two off-board). Build and upload the code into ATmega328P and verify its functionality.

7. (Optional) Consider an active-low push button with internal pull-up resistor on the PD2 pin. Use Timer0 4-ms overflow to read button status. If the push button is pressed, turn on `LED_RED`; turn the

LED off after releasing the button. Note: Within the Timer0 interrupt service routine, use a read function from your GPIO library to get the button status.

Part 4: Extend the overflow

1. Use Timer/Counter0 16-ms overflow and toggle `LED_RED` value approximately every 100 ms (6 overflows x 16 ms = 100 ms).

   FYI: Use static variables declared in functions that use them for even better isolation or use volatile for all variables used in both Interrupt routines and main code loop. According to [7] the declaration line `static uint8_t no_of_overflows = 0;` is only executed the first time, but the variable value is updated/stored each time the ISR is called.

```
ISR(TIMER0_OVF_vect)
{
    static uint8_t no_of_overflows = 0;

    no_of_overflows++;
    if (no_of_overflows >= 6)
    {
        // Do this every 6 x 16 ms = 100 ms
        no_of_overflows = 0;
        ...
    }
    // Else do nothing and exit the ISR
}
```

2. Reduce the overflow time by storing a non-zero value in the Timer/Counter0 data register TCNT0 after each overflow.

```
ISR(TIMER0_OVF_vect)
{
    static uint8_t no_of_overflows = 0;

    no_of_overflows++;
    if (no_of_overflows >= 6)
    {
        no_of_overflows = 0;
        ...
    }
    // Change 8-bit timer value anytime it overflows
    TCNT0 = 128;
    // Overflow time: t_ovf = 1/f_cpu * (2^bit-init) * prescaler
    // Normal counting:
    // TCNT0 = 0, 1, 2, ...., 128, 129, ...., 254, 255, 0, 1
    //         |--------------------------------------|
    //                          16 ms
    // t_ovf = 1/16e6 * 256 * 1024 = 16 ms
    //
```

```
    // Shortened counting:
    // TCNT0 = 0, 128, 129, ...., 254, 255, 0, 128, ....
    //        |--------------------------|
    //                    8 ms
    // t_ovf = 1/16e6 * (256-128) * 1024 = 8 ms
}
```

3. 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. In `timer.h` header file, complete macros for all three timers.
2. Enhance the current application to control four LEDs in the Knight Rider style. Avoid using the delay library and instead, implement this functionality using a single Timer/Counter.
3. Use the ATmega328P datasheet (section **8-bit Timer/Counter0 with PWM > Modes of Operation**) to find the main differences between:
    - Normal mode,
    - Clear Timer on Compare mode,
    - Fast PWM mode, and
    - Phase Correct PWM Mode.
4. 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. Tomas Fryza. Schematic of Arduino Uno board
2. Microchip Technology Inc. ATmega328P datasheet
3. Renesas Electronics Corporation. Essentials of Microcontroller Use Learning about Peripherals: Interrupts
4. Tutorials Point. Embedded Systems - Interrupts
5. C library manual
6. norwega. Knight Rider style chaser

7. StackOverflow. Static variables inside interrupts
8. Tutorials Point. Arduino - Pulse Width Modulation
9. Tomas Fryza. Useful Git commands
10. Goxygen commands

Lab 4: Liquid crystal display (LCD)

Learning objectives

After completing this lab you will be able to:

- Use alphanumeric LCD
- Understand the digital communication between MCU and HD44780
- Understand the ASCII table
- Use library functions for LCD
- Generate custom characters on LCD

The purpose of the laboratory exercise is to understand the serial control of Hitachi HD44780-based LCD character display and how to define custom characters. Another goal is to learn how to read documentation for library functions and use them in your own project.

Table of contents

Components list

- Arduino Uno board, USB cable
- LCD keypad shield

Pre-Lab preparation

1. Use schematic of the LCD keypad shield and find out the connection of LCD display. What data and control signals are used? What is the meaning of these signals?

LCD signal(s)	AVR pin(s)	Description
RS	PB0	Register selection signal. Selection between Instruction register (RS=0) and Data register (RS=1)
R/W		
E		
D[3:0]		
D[7:4]		
K		

2. What is the ASCII table? What are the codes/values for uppercase letters `A` to `E`, lowercase letters `a` to `e`, and numbers `0` to `4` in this table?

Char	Decimal	Hexadecimal
A	65	0x41
B		
...		
a	97	0x61
b		
...		
0	48	0x30
1		
...		

Part 1: LCD screen module

LCD (Liquid Crystal Display) is an electronic device which is used for display any ASCII text. There are many different screen sizes e.g. 16x1, 16x2, 16x4, 20x4, 40x4 characters and each character is made of 5x8 matrix pixel dots. LCD displays have different LED back-light in yellow-green, white and blue color. LCD modules are mostly available in COB (Chip-On-Board) type. With this method, the controller IC chip or driver (here: HD44780) is directly mounted on the backside of the LCD module itself.

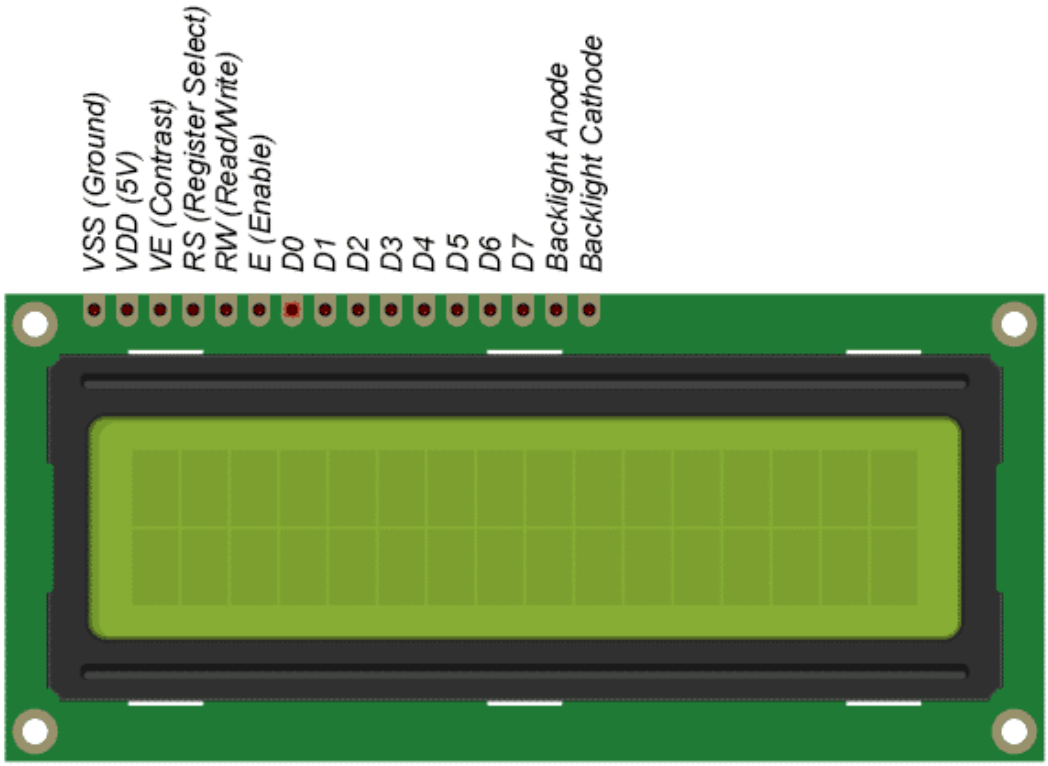
The control is based on the Hitachi HD44780 chipset (or a compatible), which is found on most text-based LCDs, and hence the driving software remains the same even for different screen sizes. The driver contains instruction set, character set, and in addition you can also generate your own characters.

The HD44780 is capable of operating in 8-bit mode i.e. faster, but 11 microcontroller pins (8 data + 3 control) are needed. Because the speed is not really that important as the amount of data needed to drive the display is low, the 4-bit mode is more appropriate for microcontrollers since only 4+2=6 (or 4+3=7) pins are needed.

In 8-bit mode we send the command/data to the LCD using eight data lines (D0-D7), while in 4-bit mode we use four data lines (D4-D7) to send commands and data. Inside the HD44780 there is still an 8-bit operation so for 4-bit mode, two writes to get 8-bit quantity inside the chip are made (first high four bits and then low four bits with an E clock pulse).

In the lab, the LCD1602 display module is used. The display consists of 2 rows of 16 characters each. It has an LED back-light and it communicates through a parallel interface with only 6 wires (+ 1 signal for backglight control):

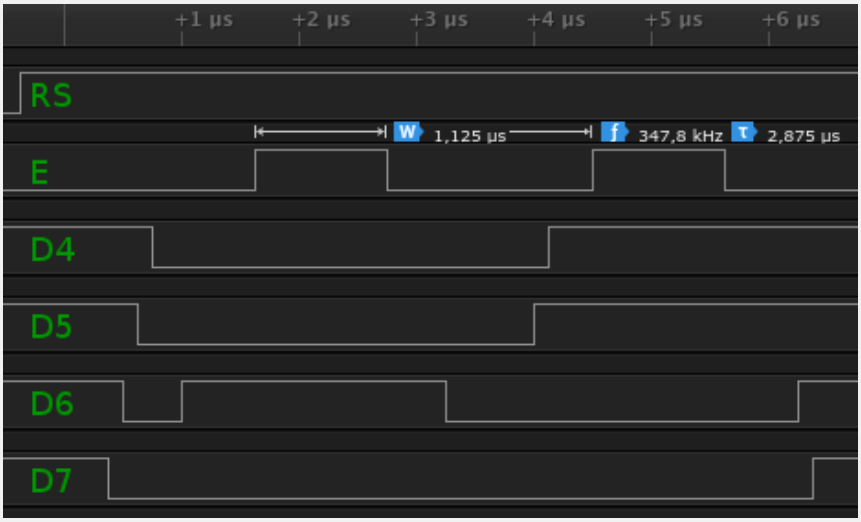
- RS - register select. Selects the data or instruction register inside the HD44780
- E - enable. This loads the data into the HD44780 on the falling edge
- (at LCD keypad shield, R/W pin is permanently connected to GND)
- D7:4 - Upper nibble used in 4-bit mode
- K - Back-light cathode



When a command is given to LCD, the command register (RS = 0) is selected and when data is sent to LCD, the data register (RS = 1) is selected. A **command** is an instruction entered on the LCD in order to perform the required function. In order to display textual information, **data** is send to LCD.

Example of HD44780 communication

Question: Let the following image shows the communication between ATmega328P and LCD display in 4-bit mode. How does HD44780 chipset understand the sequence of these signals?



Answer: The following signals are read on the first falling edge of the enable, therefore: `RS = 1` (data register) and higher four data bits `D7:4 = 0100`. On the second falling edge of enable, the lower four data bits are `D7:4 = 0011`. The whole byte transmitted to the LCD is therefore `0100_0011` (0x43) and according to the ASCII (American Standard Code for Information Interchange) table, it represents letter `C`.

The Hitachi HD44780 has many commands, the most useful for initialization, xy location settings, and print [1].

Table 6 Instructions

Instruction	Code RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Clear display	0	0	0	0	0	0	0	0	0	1	Clears entire display and sets DDRAM address 0 in address counter.	
Return home	0	0	0	0	0	0	0	0	1	—	Sets DDRAM address 0 in address counter. Also returns display from being shifted to original position. DDRAM contents remain unchanged.	1.52 ms
Entry mode set	0	0	0	0	0	0	0	1	I/D	S	Sets cursor move direction and specifies display shift. These operations are performed during data write and read.	37 µs
Display on/off control	0	0	0	0	0	0	1	D	C	B	Sets entire display (D) on/off, cursor on/off (C), and blinking of cursor position character (B).	37 µs
Cursor or display shift	0	0	0	0	0	1	S/C	R/L	—	—	Moves cursor and shifts display without changing DDRAM contents.	37 µs
Function set	0	0	0	0	1	DL	N	F	—	—	Sets interface data length (DL), number of display lines (N), and character font (F).	37 µs
Set CGRAM address	0	0	0	1	ACG	ACG	ACG	ACG	ACG	ACG	Sets CGRAM address. CGRAM data is sent and received after this setting.	37 µs
Set DDRAM address	0	0	1	ADD	ADD	ADD	ADD	ADD	ADD	ADD	Sets DDRAM address. DDRAM data is sent and received after this setting.	37 µs
Read busy flag & address	0	1	BF	AC	AC	AC	AC	AC	AC	AC	Reads busy flag (BF) indicating internal operation is being performed and reads address counter contents.	0 µs

Table 6 Instructions (cont)

Instruction	Code RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0	Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs t_{ADD} = 4 µs*
	I/D = 1: Increment I/D = 0: Decrement S = 1: Accompanies display shift S/C = 1: Display shift S/C = 0: Cursor move R/L = 1: Shift to the right R/L = 0: Shift to the left DL = 1: 8 bits, DL = 0: 4 bits N = 1: 2 lines, N = 0: 1 line F = 1: 5 × 10 dots, F = 0: 5 × 8 dots BF = 1: Internally operating BF = 0: Instructions acceptable										DDRAM: Display data RAM CGRAM: Character generator RAM ACG: CGRAM address ADD: DDRAM address (corresponds to cursor address) AC: Address counter used for both DD and CGRAM addresses	Execution time changes when frequency changes Example: When f_{cp} or f_{OSC} is 250 kHz, $37\ \mu s \times \frac{270}{250} = 40\ \mu s$

Note: — indicates no effect.

* After execution of the CGRAM/DDRAM data write or read instruction, the RAM address counter is incremented or decremented by 1. The RAM address counter is updated after the busy flag turns off. In Figure 10, t_{ADD} is the time elapsed after the busy flag turns off until the address counter is updated.

If you are an advanced programmer and would like to create your own library for interfacing your microcontroller with an LCD module then you have to understand those instructions and commands which can be found its datasheet.

Part 2: Synchronize repositories and create a new project

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 `lab4-lcd` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB4-LCD > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: Library for HD44780 based LCDs

In the lab, we are using LCD library for HD44780 based LCDs developed by Peter Fleury.

1. Use the online manual of LCD library (generated by Doxygen tool) and add input parameters and description of the following functions.

Function name	Function parameters	Description	Example
`lcd_init`	`LCD_DISP_OFF` `LCD_DISP_ON` `LCD_DISP_ON_CURSOR` `LCD_DISP_ON_CURSOR_BLINK`	Display off	`lcd_init(LCD_DISP_OFF);` ... ...
`lcd_clrscr`			`lcd_clrscr();`
`lcd_gotoxy`			
`lcd_putc`			
`lcd_puts`			

2. Copy/paste template code to `LAB4-LCD > src > main.c` source file.

3. In PlatformIO project, create a new folder `LAB4-LCD > lib > gpio`. Copy your GPIO library files `gpio.c` and `gpio.h` from the previous lab to this folder.

4. In PlatformIO project, create a new file `LAB4-LCD > include > timer.h`. Copy/paste header file from the previous lab to this file.

5. In PlatformIO project, create a new folder `LAB4-LCD > lib > lcd`. Within this folder, create three new files `lcd.c`, `lcd.h`, and `lcd_definitions.h`. The final project structure should look like this:

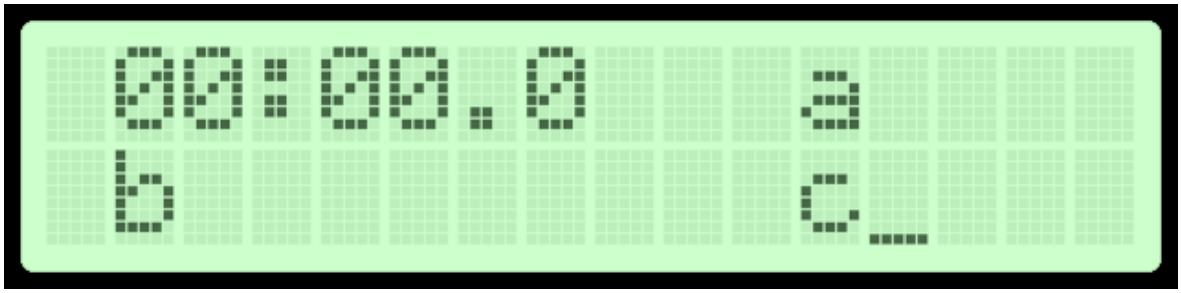
```
LAB4-LCD            // PlatfomIO project
├── include         // Included file(s)
│   └── timer.h
├── lib             // Libraries
│   └── gpio        // Your GPIO library
│   │   ├── gpio.c
│   │   └── gpio.h
```

```
|   └── lcd          // Peter Fleury's LCD library
|       ├── lcd.c
|       ├── lcd.h
|       └── lcd_definitions.h
├── src              // Source file(s)
|   └── main.c
├── test             // No need this
└── platformio.ini   // Project Configuration File
```

1. Copy/paste header file to `lcd.h`
2. Copy/paste header file to `lcd_definitions.h`
3. Copy/paste library source file to `lcd.c`

6. Go through the `lcd_definitions.h` and `main.c` files and make sure you understand each line. Build and upload the code to Arduino Uno board.

7. Use library functions `lcd_gotoxy()`, `lcd_puts()`, `lcd_putc()` and display strings/characters on the LCD as shown in the figure bellow. Explanation: You will later display the square of seconds at position "a", the process bar at "b", and the rotating text at position "c". Note, there is a non-blinking cursor after letter "c".

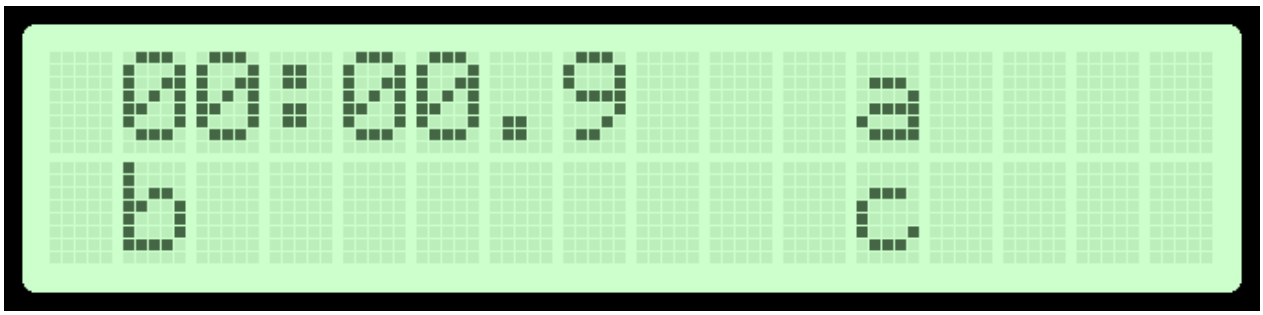


> **Note:** The figure above was created by online LCD Display Screenshot Generator.

8. Use the PB2 pin to control the back-light of the LCD screen. (Optionally: Create a new library function for this purpose.)

Part 4: Stopwatch

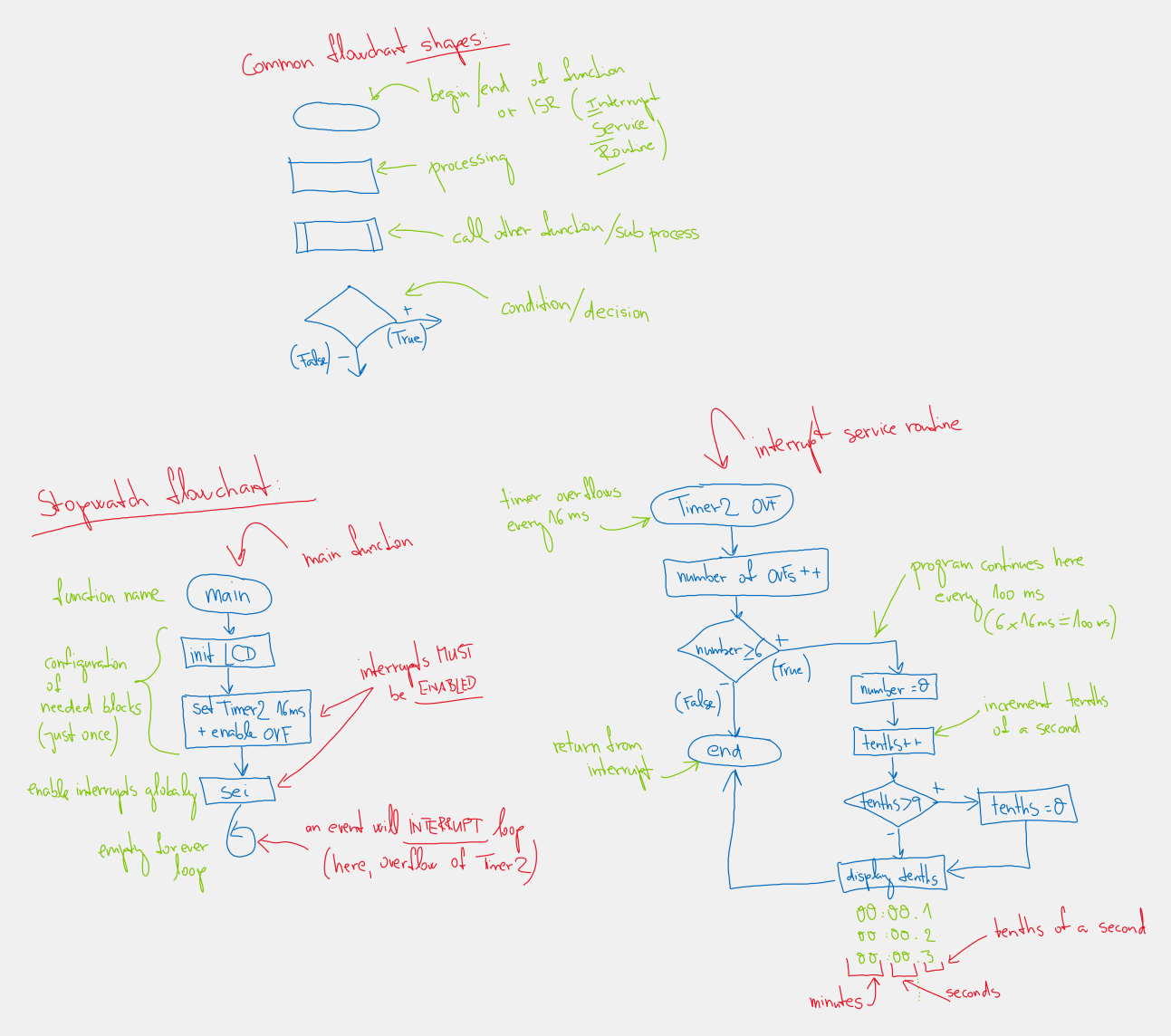
1. Use Timer/Counter2 16-ms overflow and update the stopwatch LCD value approximately every 100 ms (6 x 16 ms = 100 ms) as explained in the previous lab. Display tenths of a second only in the form `00:00.tenths`, ie let the stopwatch counts from `00:00.0` to `00:00.9` and then starts again.



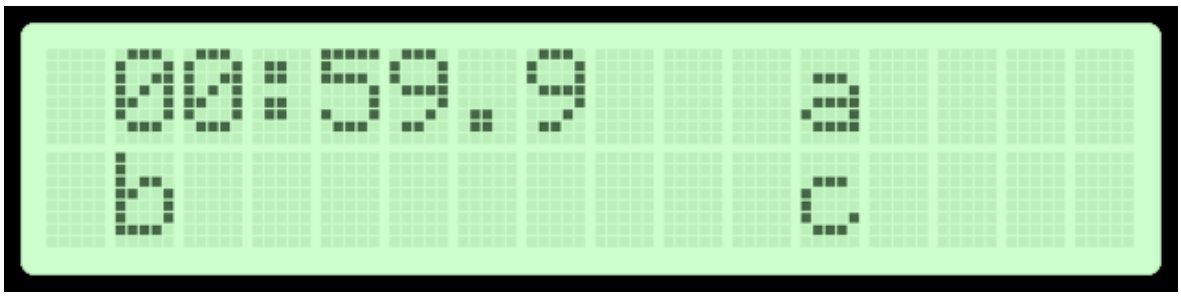
IMPORTANT: Because library functions only allow to display a string (`lcd_puts`) or individual characters (`lcd_putc`), the variables' number values need to be converted to such strings. To do this, use the `itoa(number, string, num_base)` function from the standard `stdlib.h` library. The `num_base` parameter allows you to display the `number` in decimal, hexadecimal, or binary.

```
#include <stdlib.h>         // C library. Needed for number conversions
...

ISR(TIMER2_OVF_vect)
{
    static uint8_t no_of_overflows = 0;
    static uint8_t tenths = 0;  // Tenths of a second
    char string[2];             // String for converted numbers by itoa()

    no_of_overflows++;
    if (no_of_overflows >= 6)
    {
        // Do this every 6 x 16 ms = 100 ms
        no_of_overflows = 0;

        // Count tenth of seconds 0, 1, ..., 9, 0, 1, ...
        ...

        itoa(tenths, string, 10);  // Convert decimal value to string
        // Display "00:00.tenths"
        lcd_gotoxy(7, 0);
        lcd_puts(string);
    }
    // Else do nothing and exit the ISR
}
```

2. A flowchart is a visual representation of a certain process or flow of instructions of an algorithm that helps to understand it. A flowchart basically uses rectangles, diamonds, ovals and various other shapes to make the problem easier to understand.



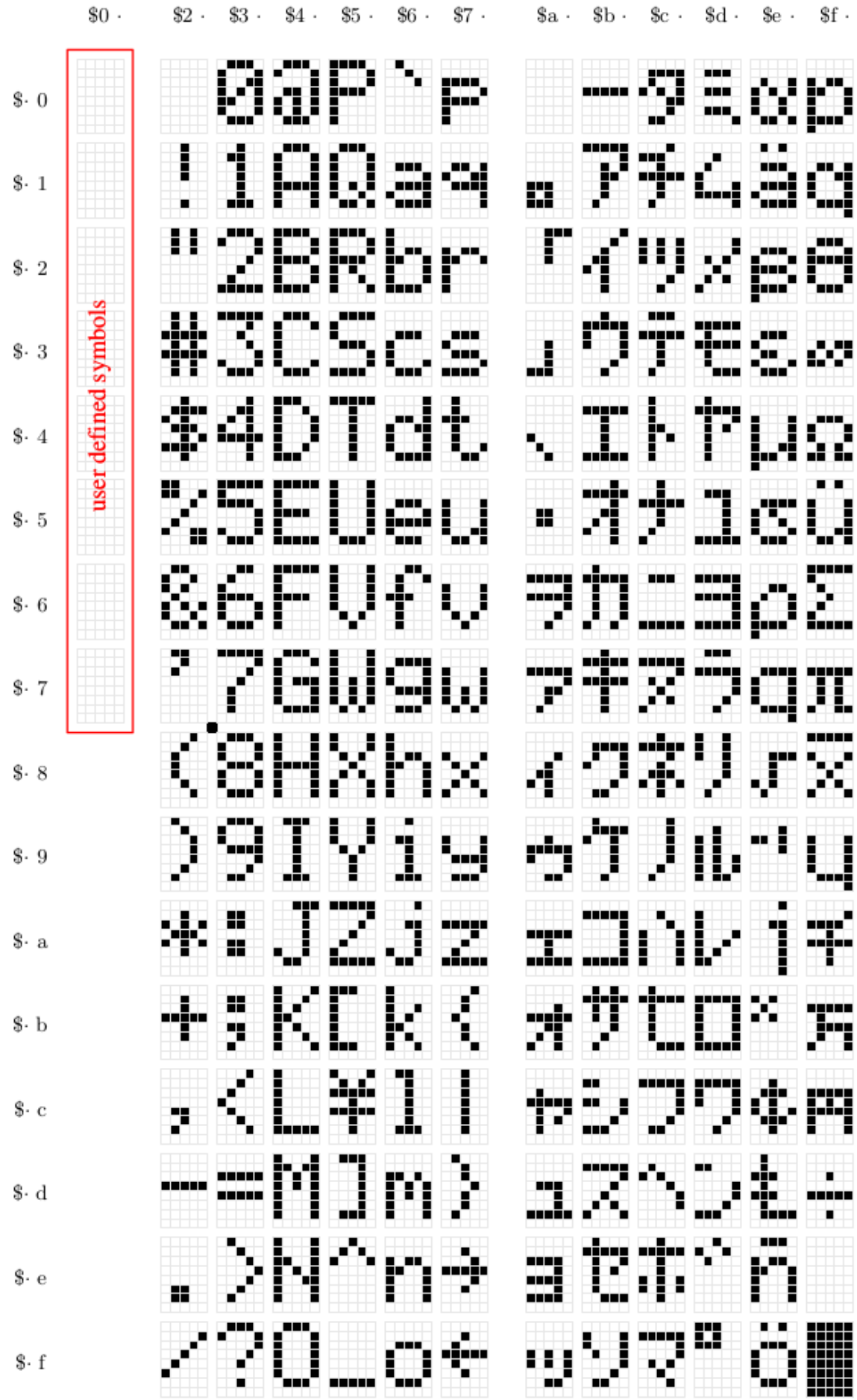
3. Complete the stopwatch flowchart of the Timer/Counter2 interrupt service routine with seconds. According to the flowchart, program the `ISR()` source code. Let the stopwatch counts from `00:00.0` to `00:59.9` and then starts again.



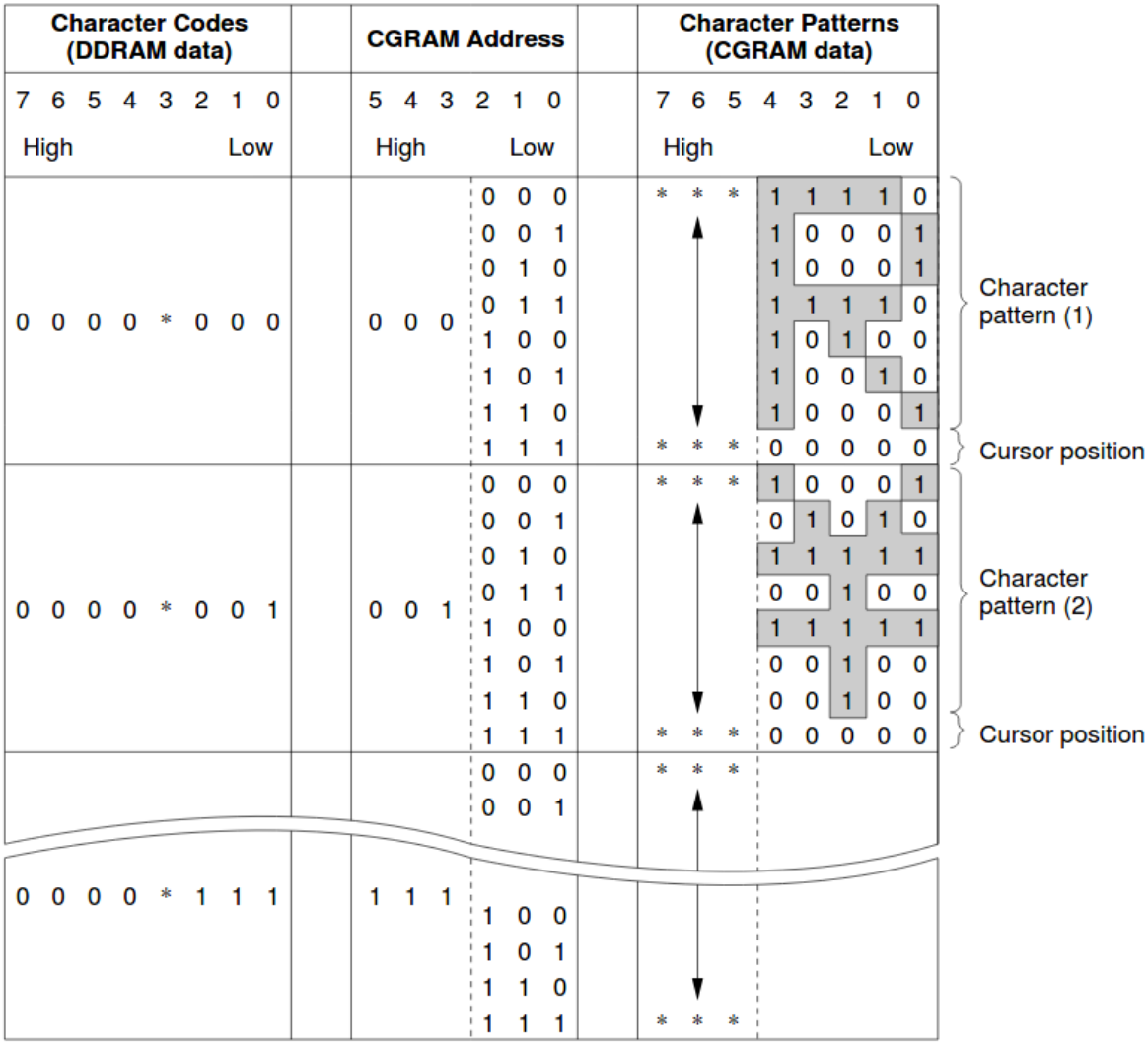
Part 5: Custom characters

All LCD displays based on the Hitachi HD44780 controller have two types of memory that store defined characters: CGROM and CGRAM (Character Generator ROM & RAM). The CGROM memory is non-volatile and cannot be modified, while the CGRAM memory is volatile and can be modified at any time [4].

CGROM memory is used to store all permanent fonts that can be displayed using their ASCII code. For example, if we write 0x43, then we get the character "C" on the display. In total, it can generate 192 5x8 character patterns.



CGRAM is another memory that can be used for storing user defined characters. This RAM is limited to 64 bytes. Meaning, for 5x8 pixel based LCD, up to 8 user-defined characters can be stored in the CGRAM. It is useful if you want to use a character that is not part of the standard 127-character ASCII table.



Notes:
1. Character code bits 0 to 2 correspond to CGRAM address bits 3 to 5 (3 bits: 8 types).
2. CGRAM address bits 0 to 2 designate the character pattern line position. The 8th line is the cursor position and its display is formed by a logical OR with the cursor.
   Maintain the 8th line data, corresponding to the cursor display position, at 0 as the cursor display.
   If the 8th line data is 1, 1 bits will light up the 8th line regardless of the cursor presence.
3. Character pattern row positions correspond to CGRAM data bits 0 to 4 (bit 4 being at the left).
4. As shown Table 5, CGRAM character patterns are selected when character code bits 4 to 7 are all 0. However, since character code bit 3 has no effect, the R display example above can be selected by either character code 00H or 08H.
5. 1 for CGRAM data corresponds to display selection and 0 to non-selection.

* Indicates no effect.

A custom character is an array of 8 bytes. Each byte (only 5 bits are considered) in the array defines one row of the character in the 5x8 matrix. Whereas, the zeros and ones in the byte indicate which pixels in the row should be on and which ones should be off.

1. To design a new custom character, store it in CGRAM according to the following code.

```
...
#define N_CHARS 1  // Number of new custom characters

int main(void)
{
    // Custom character definition using https://omerk.github.io/lcdchargen/
    uint8_t customChar[N_CHARS*8] = {
```

```
        0b00111,
        0b01110,
        0b11100,
        0b11000,
        0b11100,
        0b01110,
        0b00111,
        0b00011
    };

    // Initialize display
    lcd_init(LCD_DISP_ON);

    lcd_command(1<<LCD_CGRAM);        // Set addressing to CGRAM (Character
Generator RAM)
                                      // ie to individual lines of character
patterns
    for (uint8_t i = 0; i < N_CHARS*8; i++)  // Copy new character patterns
line by line to CGRAM
        lcd_data(customChar[i]);
    lcd_command(1<<LCD_DDRAM);        // Set addressing back to DDRAM (Display
Data RAM)
                                      // ie to character codes

    // Display symbol with Character code 0
    lcd_putc(0x00);
    ...
```

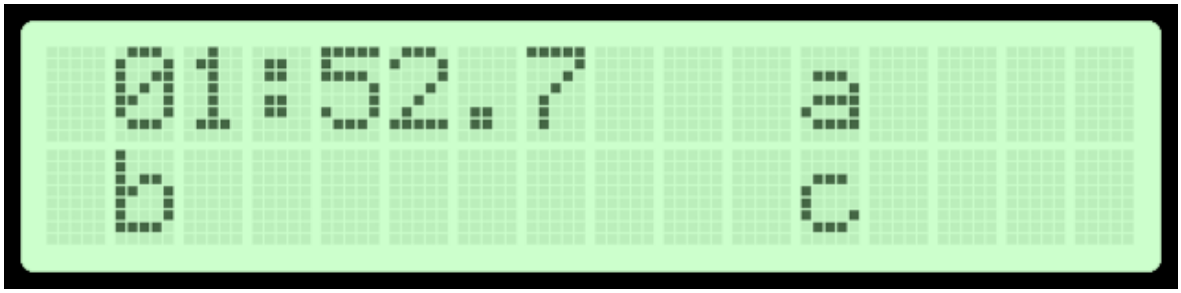
2. Design at least one more custom character, store it in CGRAM memory according to the previous code, and display all new characters on the LCD screen.

3. 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.

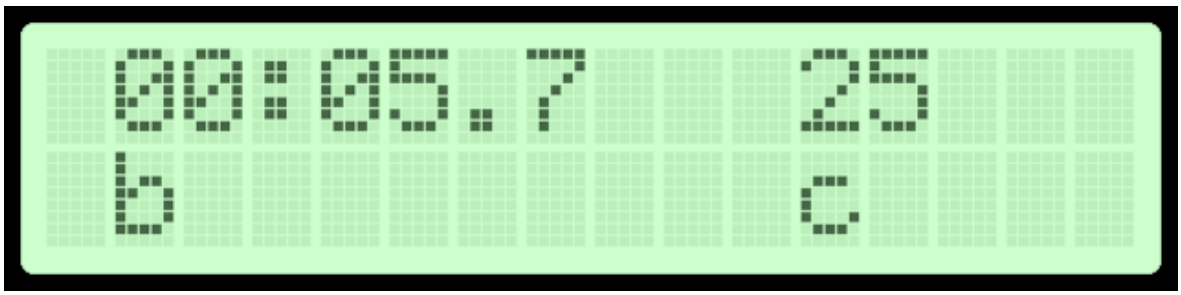
(Optional) Experiments on your own

1. Complete the `TIMER2_OVF_vect` interrupt routine with stopwatch code and display `minutes:seconds.tenths`.

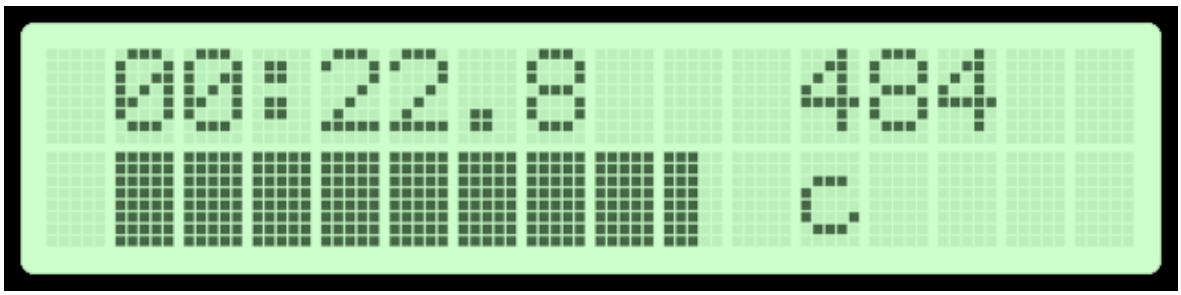


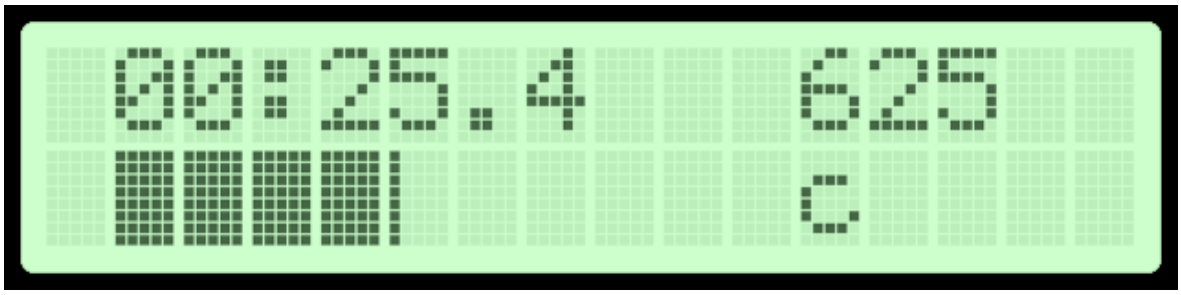
2. In `lcd.h` and `lcd.c` files create a new library function to turn on/off LCD's backlight.

3. Display the square value of the `seconds` at LCD position "a".



4. Use new characters and create a progress bar at LCD position "b". Let the full bar state corresponds to one second.



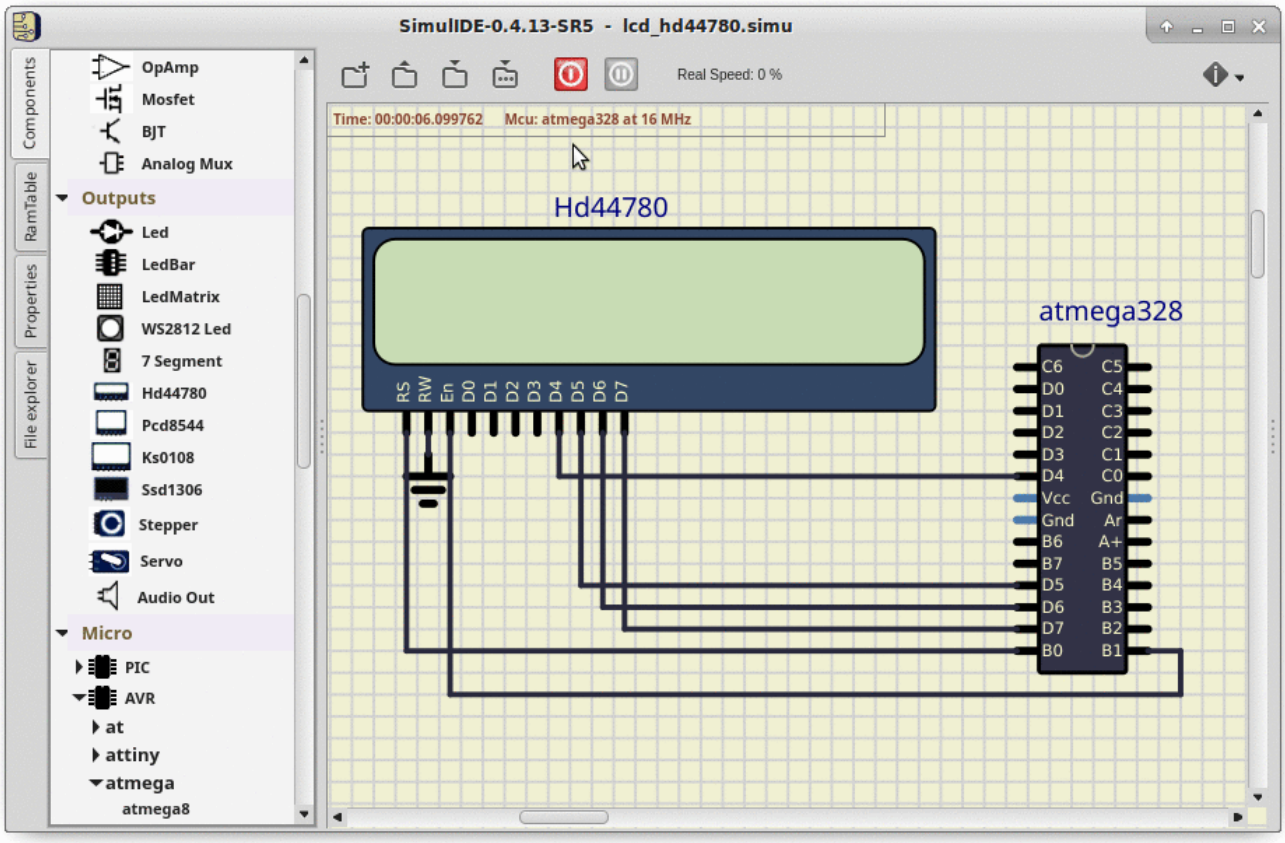


Hint: Use Timer/Counter0 with 16ms overflow and change custom characters at specific display position.

```
/* Variables -----------------------------------------------------------*/
// Custom character definition using https://omerk.github.io/lcdchargen/
uint8_t customChar[] = {
    // addr 0: .....
    0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000, 0b00000,
    // addr 1: |....
    0b10000, 0b10000, 0b10000, 0b10000, 0b10000, 0b10000, 0b10000, 0b10000,
    ...
};
...
/*---------------------------------------------------------------------*/
/**
 * ISR starts when Timer/Counter0 overflows. Update the progress bar on
 * LCD display every 16 ms.
 */
ISR(TIMER0_OVF_vect)
{
    static uint8_t symbol = 0;
    static uint8_t position = 0;

    lcd_gotoxy(1+position, 1);
    lcd_putc(symbol);
```

```
        // WRITE YOUR CODE HERE
    }
```

5. From LCD position "c", displays running text, ie text that moves characters to the left twice per second. Hint: Use Timer/Counter1 with an 262ms prescaler and every 2nd overflow move the auxiliary variable along the defined string, such as `uint8_t running_text[] = " I like Digital electronics!\n";`.



6. Draw a flowchart for `TIMER2_OVF_vect` interrupt service routine which overflows every 16 ms but it updates the stopwatch LCD screen approximately every 100 ms (6 x 16 ms = 100 ms). Display tenths of a second, seconds, and minutes and let the stopwatch counts from `00:00.0` to `59:59.9` and then starts again. The image can be drawn on a computer or by hand. Use clear description of individual algorithm steps.

7. Finish all (or several) 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. HITACHI. HD44780U, Dot Matrix Liquid Crystal Display Controller/Driver
2. Peter Fleury. LCD library for HD44780 based LCDs
3. avtanski.net. LCD Display Screenshot Generator
4. LastMinuteEngineers.com. Interfacing 16×2 Character LCD Module with Arduino
5. omerk.github.io. Custom Character Generator for HD44780 LCD Modules
6. Protostack Pty Ltd. HD44780 Character LCD Displays – Part 1
7. Circuit Basics

8. CGRAM display memory
9. Tomas Fryza. Useful Git commands
10. Tomas Fryza. Schematic of LCD Keypad shield
11. Goxygen commands

Lab 5: Analog-to-Digital Converter (ADC)

Learning objectives

After completing this lab you will be able to:

- Calculate the voltage divider
- Understand the analog-to-digital conversion process
- Configure and use internal ADC unit

The purpose of the laboratory exercise is to understand analog-to-digital conversion and the use of an internal 8-channel 10-bit analog-to-digital converter.

Table of contents

Components list

- Arduino Uno board, USB cable
- LCD keypad shield

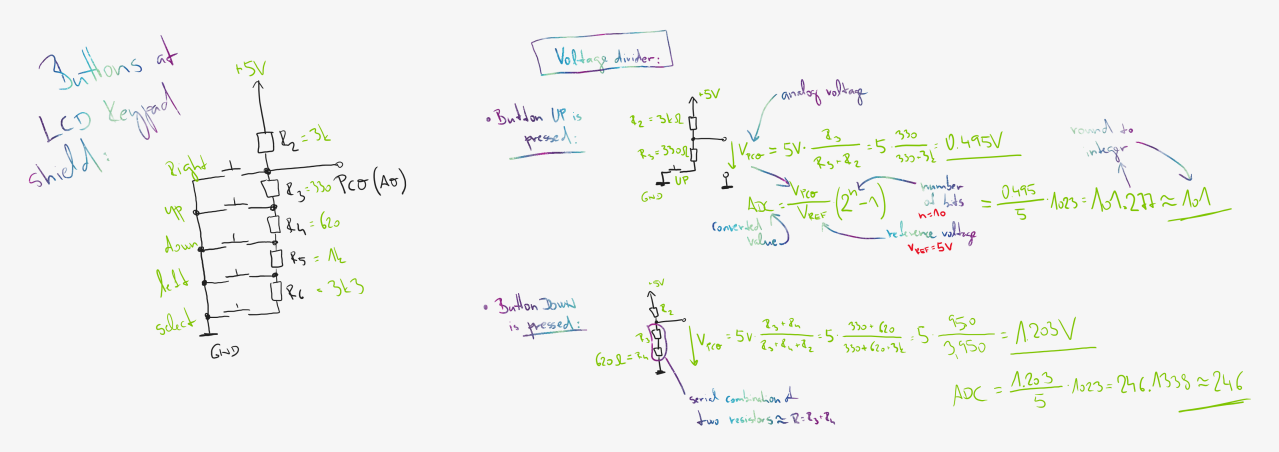
Pre-Lab preparation

1. Use schematic of the LCD keypad shield and find out the connection of five push buttons: Select, Left, Up, Down, and Right.

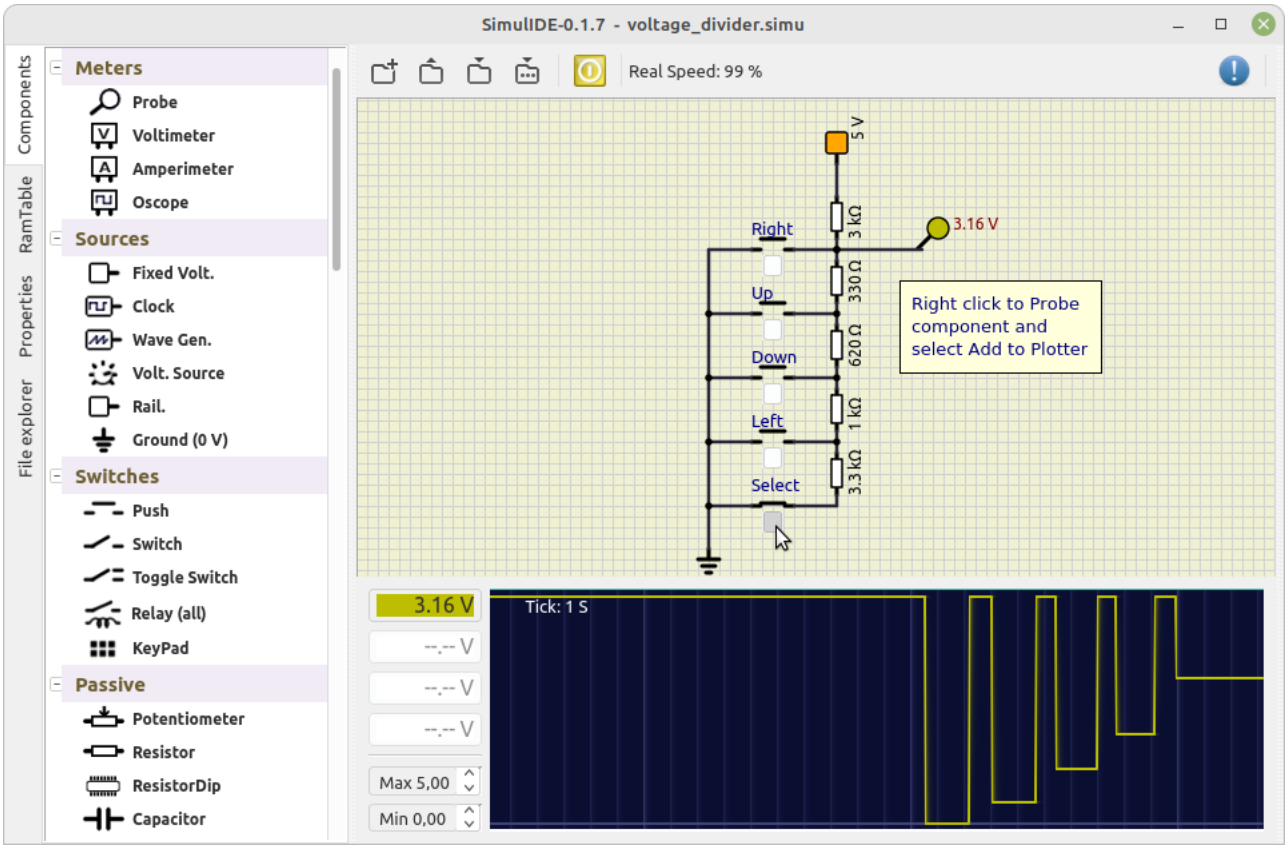
2. Remind yourself, what a voltage divider is.

Part 1: Voltage divider

1. According to the LCD keypad shield connection, calculate the voltage values on pin PC0 for each pressed buttons. In this case, the voltage on the pin is given by the voltage divider, where resistors R3, R4, R5 and R6 are applied successively. Write your values to the **PC0 voltage** column in the following table.



Push button	PC0 voltage	ADC value (calculated)	ADC value (measured)	ADC value (measured, hex)
Right	0 V	0	0	0
Up	0.495 V	101		
Down	1.203 V	246		
Left				
Select				
none				



Part 2: Synchronize repositories and create a new project

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 `lab5-adc` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB5-ADC > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: Analog-to-Digital Conversion

We live in an analog world, but surrounded by digital devices. Everything we see, feel or measure is analog in nature such as light, temperature, speed, pressure etc. It is obvious that we need something that could convert these analog parameters to digital value for a microcontroller or micro-processor to understand it.

An Analog to Digital Converter (ADC) is a circuit that converts a continuous voltage value (analog) to a binary value (digital). These ADC circuits can be found as an individual ADC ICs by themselves or embedded into a modern microcontroller.

The internal ADC module of ATmega328P can be used in relatively slow and not extremely accurate data acquisitions. But it is a good choice in most situations, like reading sensor data or push button signals.

AVR's ADC module has 10-bit resolution with +/-2LSB accuracy. It means it returns a 10-bit integer value, i.e. a range of 0 to 1023. It can convert data at up to 76.9 kSPS, which goes down when higher resolution is used. We mentioned that there are 8 ADC channels available on pins, but there are also three internal channels that can be selected with the multiplexer decoder. These are temperature sensor (channel 8), bandgap reference (1.1V) and GND (0V) [4].

1. Convert the voltages from the previous part according to the following equation. Note that reference is Vref=5V and number of bits for analog to digital conversion is n=10. Write the values to **ADC value (calculated)** column in the table from Part 2.1.

$$ADC = \left\lfloor \frac{V_{PC0}}{V_{REF}} \cdot (2^{nbit} - 1) \right\rfloor$$

2. The operation with the AD converter is performed through ADMUX, ADCSRA, ADCL+ADCH, ADCSRB, and DIDR0 registers. See ATmega328P datasheet (**Analog-to-Digital Converter > Register Description**) and complete the following table.

Operation	Register(s)	Bit(s)	Description
Voltage reference	ADMUX	REFS1:0	00: ..., 01: AVcc voltage reference (5V), ...
Input channel	ADMUX	MUX3:0	0000: ADC0, 0001: ADC1, ...
ADC enable	ADCSRA		

Operation	Register(s)	Bit(s)	Description
Start conversion			
ADC interrupt enable			
ADC clock prescaler		ADPS2:0	000: Division factor 2, 001: 2, 010: 4, ...
ADC 10-bit result			

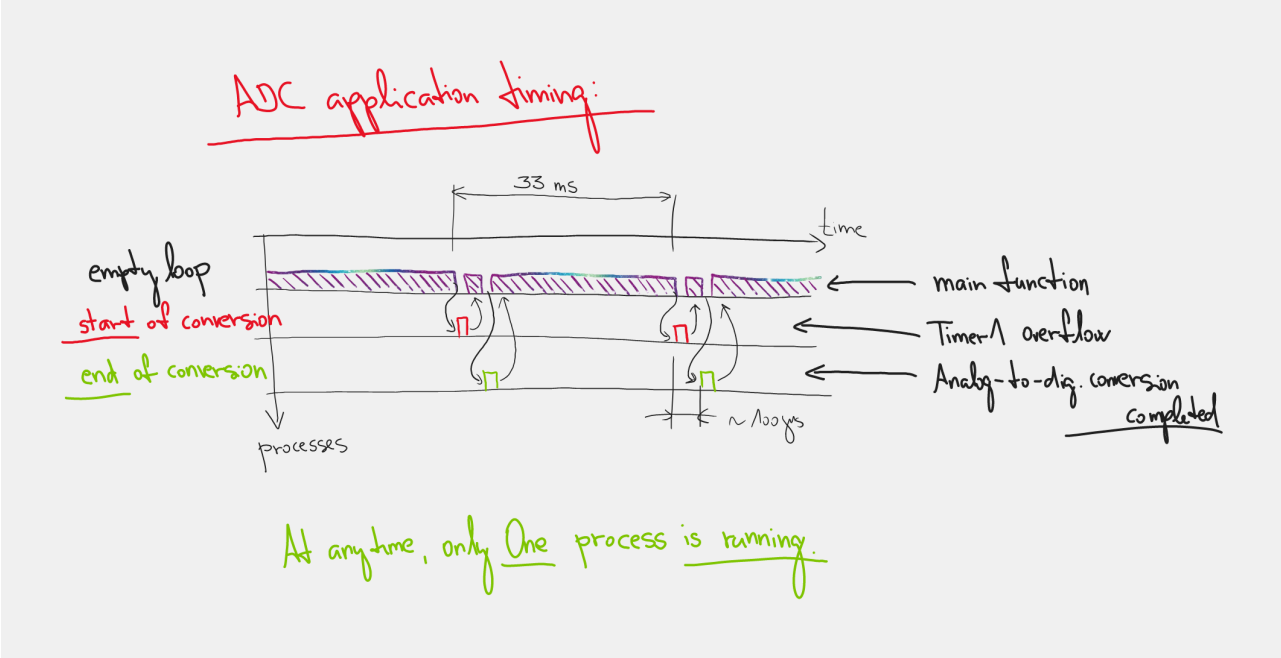
3. Copy/paste template code to `LAB5-ADC > src > main.c` source file.

4. Use your favorite file manager and copy `timer` and `lcd` libraries from the previous lab to the proper locations within the `LAB5-ADC` project. The final project structure should look like this:

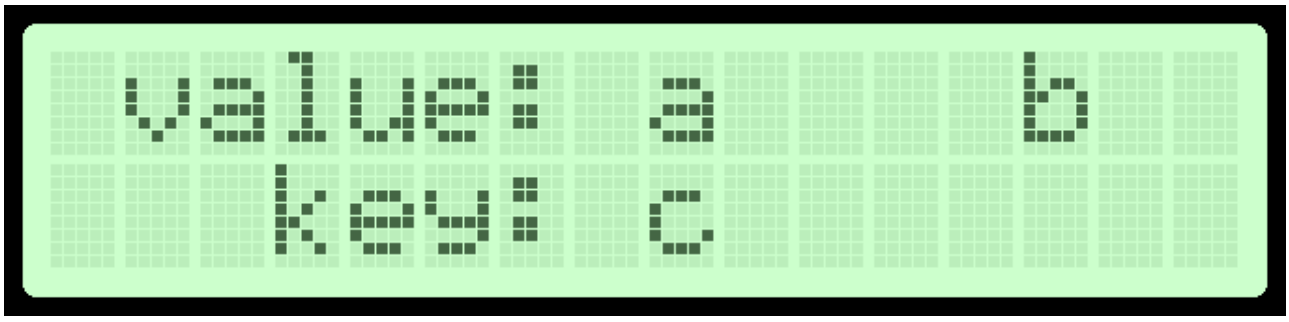
```
LAB5-ADC            // PlatfomIO project
├── include         // Included file(s)
│   └── timer.h
├── lib             // Libraries
│   └── lcd         // Peter Fleury's LCD library
│       ├── lcd.c
│       ├── lcd.h
│       └── lcd_definitions.h
├── src             // Source file(s)
│   └── main.c
├── test            // No need this
└── platformio.ini  // Project Configuration File
```

5. Go through the `main.c` file and make sure you understand each line. Use ATmega328P datasheet and complete the ADC configuration as follows:

    - voltage reference: AVcc with external capacitor at AREF pin
    - input channel: ADC0
    - enable ADC module
    - enable interrupt
    - clock prescaler: 128

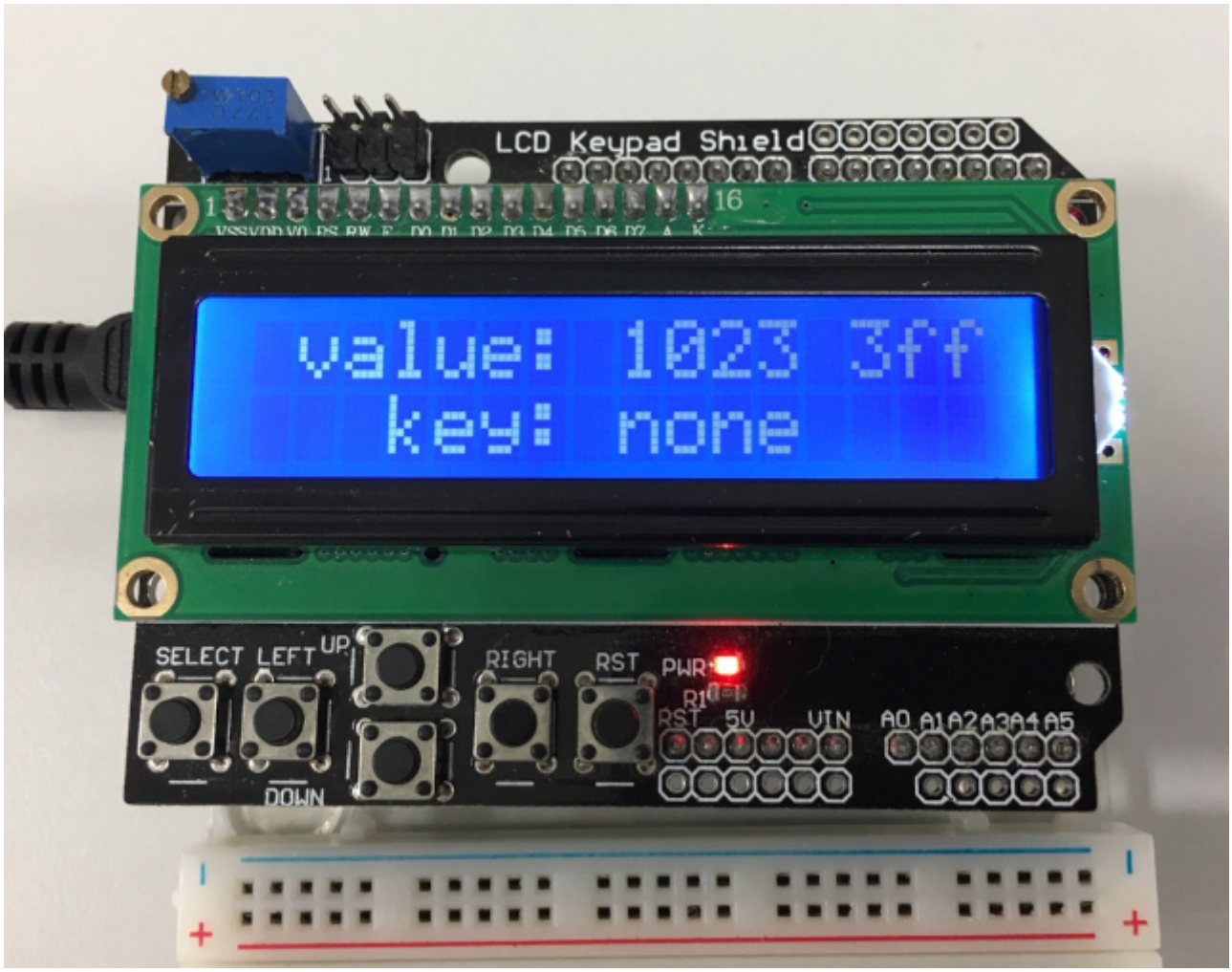


Use single conversion mode and start the AD conversion every 33 ms by Timer/Counter1 overflow. When analog to digital conversion is finished, read the voltage level on push buttons and display it in decimal at LCD display position a. Display the same value in hexadecimal at position b. Note that you can use the 16-bit ADC variable (declared in the AVR library) to read the value from both converter registers ADCH:L.




```
/*********************************************************************
*
 * Function: ADC complete interrupt
 * Purpose:  Display converted value on LCD screen.

*********************************************************************
/
ISR(ADC_vect)
{
    uint16_t value;
    char string[4];  // String for converted numbers by itoa()

    // Read converted value. Note that, register pair ADCH and ADCL
    // can be read as a 16-bit value ADC
    value = ADC;
    // Convert "value" to "string" and display it
}
```

Use standard C library and display converted value as string. Build the application and upload it to Arduino Uno board.



6. Test all push buttons and write converted values to the **ADC value (measured)** column in the table from Part 1.1.

7. Apply the "extending" method from past labs and start the ADC conversion not every 33 milliseconds but every 100 milliseconds.

8. Based on the converted values, distinguish which push button was pressed and display the information at LCD position `c`.

9. 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. In your application, try to recalculate the input voltage values in mV. *Hint: Use integer data types only; the absolute accuracy of the calculation is not important here.*

2. Create a library for Analog-to-Digital Converter. Create new files `adc.h` and `adc.c`, suggest function names, their parameters, and program their bodies.

3. Consider an application for temperature measurement. Use analog temperature sensor TC1046, LCD, and a LED. Every 30 seconds, the temperature is measured and the value is displayed on LCD screen. When the temperature is above the threshold, turn on the LED.

    1. Draw a schematic of temperature meter. The image can be drawn on a computer or by hand. Always name all components, their values and pin names!

    2. Draw two flowcharts of temperature meter: `TIMER1_OVF_vect` (which overflows every 1 sec) and `ADC_vect` interrupt handlers. The image can be drawn on a computer or by hand. Use clear description of individual algorithm steps.

4. Finish all (or several) 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. Tomas Fryza. Schematic of LCD Keypad shield

2. EETech Media, LLC. Voltage Divider Calculator

3. Components101. Introduction to Analog to Digital Converters (ADC Converters)

4. Embedds. ADC on Atmega328. Part 1

5. Microchip Technology Inc. ATmega328P datasheet

6. Tomas Fryza. Useful Git commands

Lab 6: Universal Asynchronous Receiver-Transmitter (UART)

Learning objectives

After completing this lab you will be able to:

- Understand the UART communication
- Decode UART frames
- Use functions from UART library
- Use logic analyzer

The purpose of the laboratory exercise is to understand serial asynchronous communication, data frame structure and communication options using an internal USART unit.

Table of contents

Components list

- Arduino Uno board, USB cable
- Logic analyzer

Pre-Lab preparation

1. Use schematic of Arduino Uno board and find out on which Arduino Uno pins the UART transmitter (Tx) and receiver (Rx) are located.

2. Remind yourself, what an ASCII table is. What codes are defined for control characters `Esc`, `Space`, `Tab`, and `Enter`?

Part 1: Basics of UART communication

The UART (Universal Asynchronous Receiver-Transmitter) is not a communication protocol like SPI and I2C, but a physical circuit in a microcontroller, or a stand-alone integrated circuit, that translates communicated data between serial and parallel forms. It is one of the simplest and easiest method for implement and understanding.

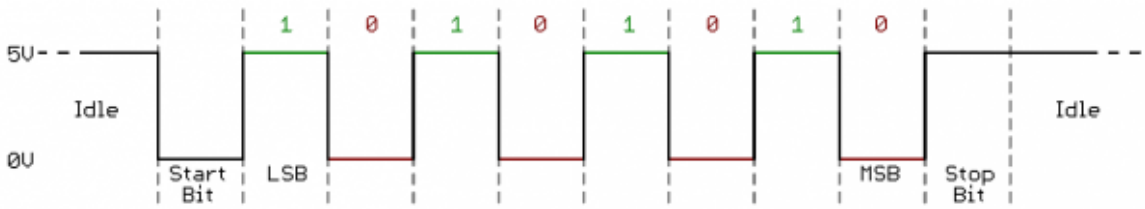
In UART communication, two UARTs communicate directly with each other. The transmitting UART converts parallel data from a CPU into serial form, transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. Only two wires are needed to transmit data between two UARTs. Data flows from the Tx pin of the transmitting UART to the Rx pin of the receiving UART [3], [4].

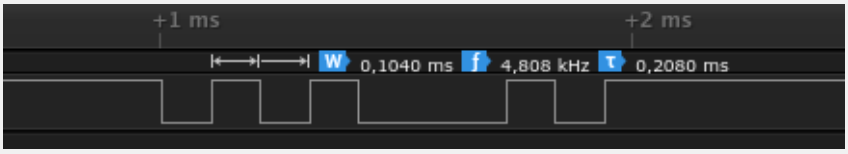
UARTs transmit data asynchronously, which means there is no external clock signal to synchronize the output of bits from the transmitting UART. Instead, timing is agreed upon in advance between both units, and special **Start** (log. 0) and 1 or 2 **Stop** (log. 1) bits are added to each data package. These bits define the beginning and end of the data packet so the receiving UART knows when to start reading the bits. In addition to the start and stop bits, the packet/frame also contains data bits and optional parity.

The amount of **data** in each packet can be set from 5 to 9 bits. If it is not otherwise stated, data is transferred least-significant bit (LSB) first.

Parity is a form of very simple, low-level error checking and can be Even or Odd. To produce the parity bit, add all 5-9 data bits and extend them to an even or odd number. For example, assuming parity is set to even and was added to a data byte `0110_1010`, which has an even number of 1's (4), the parity bit would be set to 0. Conversely, if the parity mode was set to odd, the parity bit would be 1.

One of the most common UART formats is called **9600 8N1**, which means 8 data bits, no parity, 1 stop bit and a symbol rate of 9600 Bd.



> Example of UART communication
>
> **Question:** Let the following image shows one frame of UART communication transmitting from the ATmega328P in 8N1 mode. What ASCII code/character does it represent? According to bit period, estimate the symbol rate.
>
> 
>
>
> **Answer:** 8N1 means that 8 data bits are transmitted, no parity is used, and the number of stop bits is one. Because the frame always starts with a low level start bit and the order of the data bits is from LSB to MSB, the data transmitted bu UART is therefore `0100_0101` (0x45) and according to the ASCII (American Standard Code for Information Interchange) table, it represents the letter `E`.
>
> The figure further shows that the bit period, i.e. the duration of one bit, is 104 us. The symbol rate of the communication is thus 1/104e-6 = 9615, i.e. approximately 9600 Bd.

Part 2: Synchronize repositories and create a new project

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 `lab6-uart` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB6-UART > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: Communication between Arduino board and computer

In the lab, we are using UART library developed by Peter Fleury.

1. Use online manual of UART library and add input parameters and description of the following functions.

Function name	Function parameter(s)	Description	Example
`uart_init`	`UART_BAUD_SELECT(9600, F_CPU)`	Initialize UART to 8N1 and set baudrate to 9600 Bd	`uart_init(UART_BAUD_SELECT(9600, F_CPU));`
`uart_getc`			
`uart_putc`			
`uart_puts`			

2. Copy/paste template code to `LAB6-UART > src > main.c` source file.

3. Use your favorite file manager and copy `timer.h` file from the previous labs to `LAB6-UART > include` folder.

4. In PlatformIO project, create a new folder `LAB6-UART > lib > uart`. Within this folder, create two new files `uart.c` and `uart.h`.

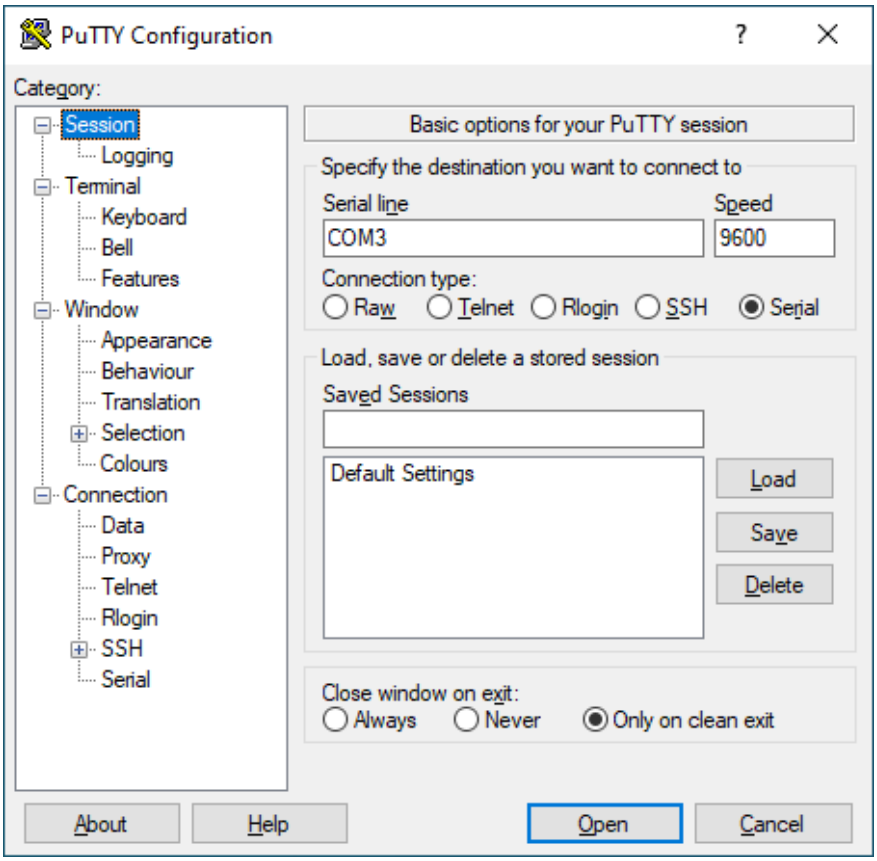
    1. Copy/paste library source file to `uart.c`
    2. Copy/paste header file to `uart.h`

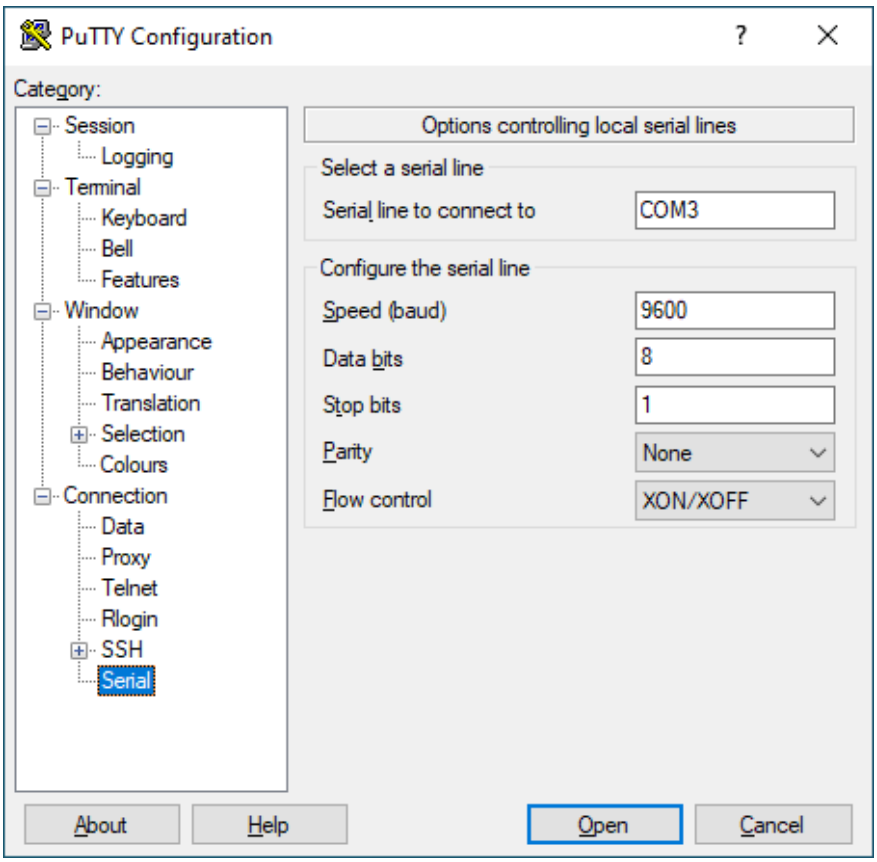
    The final project structure should look like this:

```
LAB6-UART           // PlatfomIO project
├── include         // Included file(s)
│   └── timer.h
├── lib             // Libraries
│   └── uart        // Peter Fleury's UART library
│       ├── uart.c
│       └── uart.h
├── src             // Source file(s)
│   └── main.c
├── test            // No need this
└── platformio.ini  // Project Configuration File
```

5. Go through the `main.c` file and make sure you understand each line. Build and upload the code to Arduino Uno board. What is the meaning of ASCII control characters `\r`, `\n`, and `\t`?

Use **PlatformIO: Serial Monitor** or **PuTTY application** to receive values from Arduino board. In PuTTY, set connection type to `Serial` and check that configuration is the same as in the ATmega328P application, ie. 9600 8N1 mode. Note that, **serial line** (here COM3 on Windows) could be different. In Linux, use `dmesg` command to verify your port (such as `/dev/ttyUSB0`).

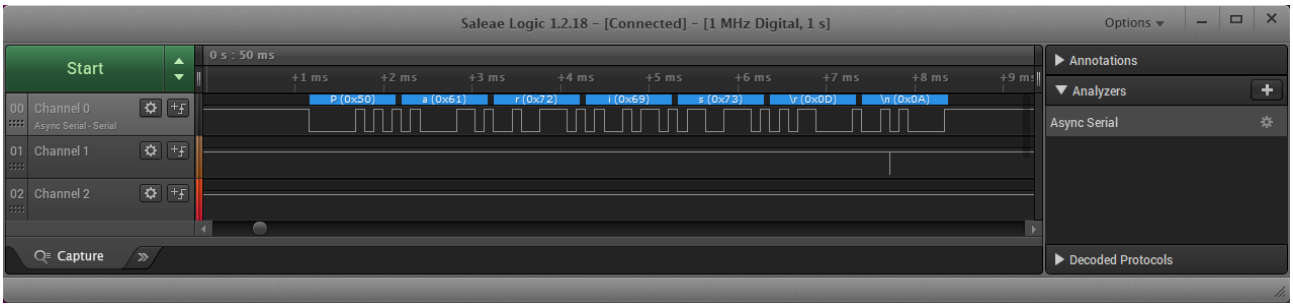




> **Warning:** Before Arduino board re-programming process, PuTTY app must be closed!

> In SimulIDE, right click to ATmega328 package and select **Open Serial Monitor**. In this window you can receive data from the microcontroller, but also send them back.

6. Configure Timer1 to overflow four times per second and transmit UART string `Paris`. Use Logic Analyzer to visualize and decode transmitting strings.



> **Note:** You have to have a Saleae logic analyzer or similar, and to download and install Saleae Logic 1 software on your computer.
>
> Tutorial about using a logic analyzer is available in this video.

7. Use `uart_getc` function and display the ASCII code of received character in decimal, hexadecimal, and binary. You can use Timer1 overflow handler to perform such receiver. Fill the table with selected keys.

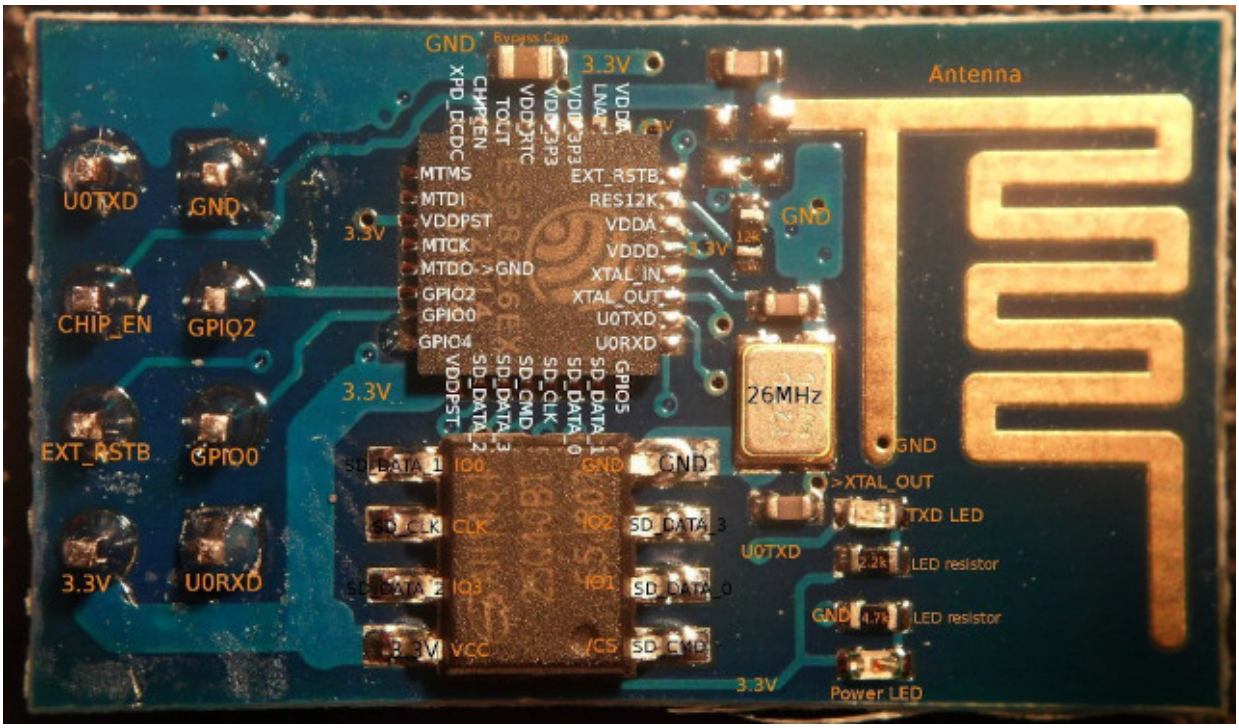
```
ISR(TIMER1_OVF_vect)
{
    uint8_t value;
    char string[8];  // String for converted numbers by itoa()

    value = uart_getc();
    if (value != '\0') {  // Data available from UART
        // Display ASCII code of received character
        // WRITE YOUR CODE HERE
    }
}
```

Char	Decimal	Hexadecimal	Binary
a	97	0x61	0b0110_0001
b			
c			
0	48	0x30	0b0011_0000
1			
2			
Esc			
Space			
Tab			

Char	Decimal	Hexadecimal	Binary
Backspace			
Enter			

8. (Optional) Verify basic AT commands of Wi-Fi module ESP8266 ESP-01. Connect Wi-Fi module to Arduino Uno board according to the following instructions.

ESP-01 pin	Arduino Uno pin	ESP-01 pin	Arduino Uno pin
U0TXD	Tx (pin 1)	GND	GND
CHIP_EN	3.3V	GPIO2	Not connected
EXT_RSTB	Not connected	GPIO0	Not connected
3.3V	3.3V	U0RXD	Rx (pin 0)



In your code, disable interruptions by commenting `// sei();` function. The reason is the micro controller will not affect UART lines and whole communication will be done between Serial Monitor and Wi-Fi module. To use PlatfomIO Serial Monitor, add the following command to `platformio.ini` project configuration file: `monitor_speed = 115200`. Compile and upload the application. Test the following AT commands and see the module's responses. If needed, use Logic analyzer to read the response of Wi-Fi module.

- `AT` - Check the communication with module
- `AT+CWMODE=1` - Set the module mode
- `AT+GMR` - Get the module version
- `AT+CWLAPOPT=1,6` - Limit the list to `rssi` and `ssid` parameters only
- `AT+CWLAP` - List `ssid` and `rssi` parameters of available Wi-Fi APs (takes few seconds)

The complete list and description of all AT commands are available here. To avoid a conflict with Wi-Fi module, remove the Tx and Rx wires when uploading the firmware and put them back after the upload is

complete.

9. 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

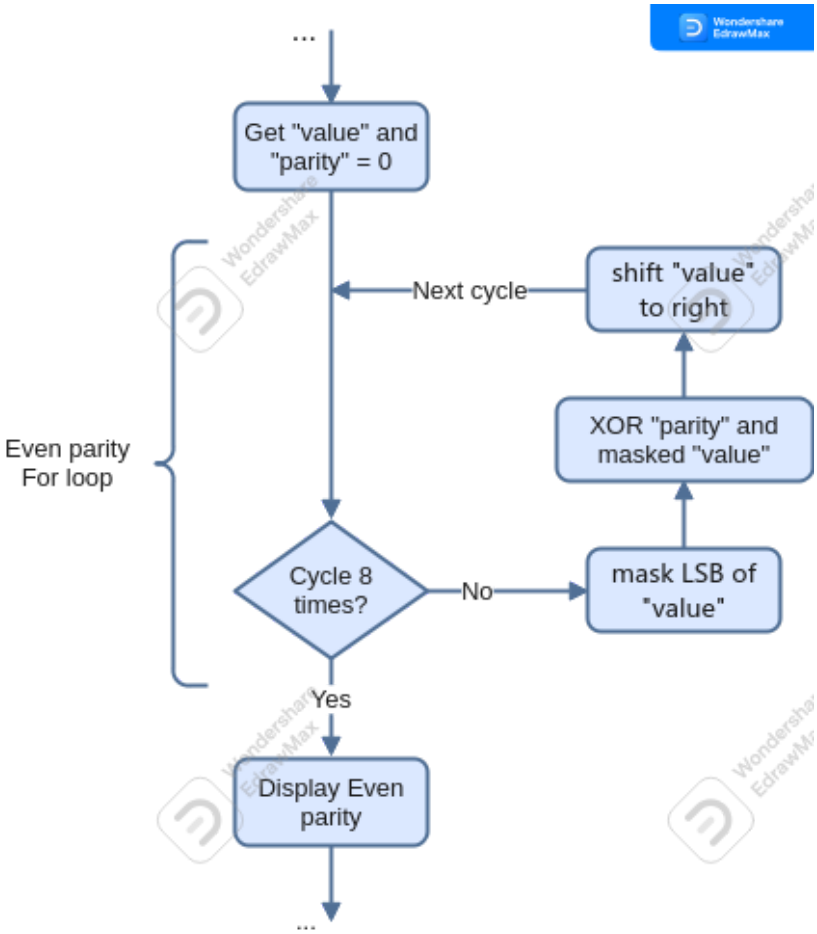
1. Use ANSI Escape Sequences and modify color and format of transmitted strings according to the following code. Try other formatting styles.

```
/*
 * Color/formatting sequence is prefixed with `Escape` (`\x1b` in
hexadecimal,
 * \033, etc), followed by opening square bracket `[`, commands delimited
by
 * semi colon `;` and ended by `m` character.
 *
 * Examples:
 *   \x1b[1;31m  - Set style to bold, red foreground
 *   \x1b[4,32m  - Set underline style, green foreground
 *   \x1b[0m     - Reset all attributes
 */
uart_puts("\x1b[4;32m");  // 4: underline style; 32: green foreground
uart_puts("This is all Green and Underlined\r\n");
uart_puts("\x1b[0m");     // 0: reset all attributes
uart_puts("This is Normal text again\r\n");
```

   To enable ANSI color codes in PlatformOI serial monitor, add the following line to `platformio.ini` project configuration file.

```
# Enable ANSI color codes in serial monitor
monitor_raw = yes
```

2. Program a piece of code to calculate even parity bit from received value.



3. Draw a timing diagram of the output from UART/USART when transmitting three character data `De2` in 4800 7O2 mode (7 data bits, odd parity, 2 stop bits, 4800 Bd). The image can be drawn on a computer (by WaveDrom for example) or by hand. Name all parts of timing diagram.

4. Program a software UART transmitter (emulated UART) that will be able to generate UART data on any output pin of the ATmega328P microcontroller. Let the bit rate be approximately 9600 Bd and do not use the delay library. Also consider the possibility of calculating the parity bit. Verify the UART communication with logic analyzer or oscilloscope.

5. Program an interactive console that communicates between ATmega328P and the computer (PuTTY application) via UART. Let the main screen of the console is as follows:

```
--- Interactive UART console ---
1: read current Timer/counter1 value
2: reset Timer/counter1
>
```

After pressing the '1' key on computer keyboard, ATmega328P receives ASCII code of the key and sends the current Timer1 value back to PuTTY. After pressing the '2' key, ATmega328P resets Timer1 value, etc. Use ANSI escape sequences to highlight information in PuTTY console.

```
uint8_t value;
...
value = uart_getc();
if (value != '\0') {     // Data available from UART
```

```
    if (value == `1`) {  // Key `1` received
        ...
    }
}
...
```

> **Warning:** Keep UART strings as short as possible. But if you need to transmit a larger amount of data, it is necessary to increase the size of the transmit/receive buffer in the `uart.h` file, eg to 128.
>
> ```
> /** @brief  Size of the circular receive buffer, must be power of 2
> *
> *  You may need to adapt this constant to your target and your
> application by adding
> *  CDEFS += -DUART_RX_BUFFER_SIZE=nn to your Makefile.
> */
> #ifndef UART_RX_BUFFER_SIZE
> # define UART_RX_BUFFER_SIZE 128
> #endif
>
> /** @brief  Size of the circular transmit buffer, must be power of 2
> *
> *  You may need to adapt this constant to your target and your
> application by adding
> *  CDEFS += -DUART_TX_BUFFER_SIZE=nn to your Makefile.
> */
> #ifndef UART_TX_BUFFER_SIZE
> # define UART_TX_BUFFER_SIZE 128
> #endif
> ```

6. Finish all (or several) 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. Tomas Fryza. Schematic of Arduino Uno board
2. ASCII Table
3. Circuit Basics. Basics of UART Communication
4. Eric Peña, Mary Grace Legaspi. UART: A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter
5. Peter Fleury. UART library
6. Kolban Technical Tutorials. ESP32 Technical Tutorials: Using a logic analyzer
7. Tomas Fryza. Useful Git commands
8. Christian Petersen. ANSI Escape Sequences

Lab 7: Inter-Integrated Circuits (I2C)

Learning objectives

After completing this lab you will be able to:

- Understand the I2C communication
- Use functions from I2C library
- Perform data transfers between I2C devices and MCU

The purpose of the laboratory exercise is to understand serial synchronous communication using the I2C (Inter-Integrated Circuit) bus, as well as the structure of the address and data frame and the possibilities of communication using the internal TWI (Two Wire Interface) unit.

Table of contents

Components list

- Arduino Uno board, USB cable
- Breadboard
- DHT12 humidity/temperature sensor
- RTC DS3231 and AT24C32 EEPROM memory module
- GY-521 module with MPU-6050 microelectromechanical systems
- Jumper wires

Pre-Lab preparation

1. Use schematic of the Arduino Uno board and find out on which Arduino Uno pins the SDA and SCL signals are located.

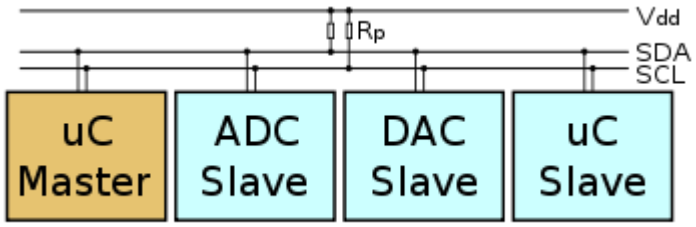
2. Remind yourself, what the general structure of I2C address and data frame is.

Part 1: I2C bus

I2C (Inter-Integrated Circuit) is a serial protocol for two-wire interface to connect low-speed devices like microcontrollers, EEPROMs, A/D and D/A converters, I/O interfaces and other similar peripherals in embedded systems. It was invented by Philips and now it is used by almost all major IC manufacturers.

I2C uses only two wires: SCL (serial clock) and SDA (serial data). Both need to be pulled up with a resistor to +Vdd. There are also I2C level shifters which can be used to connect to two I2C buses with different

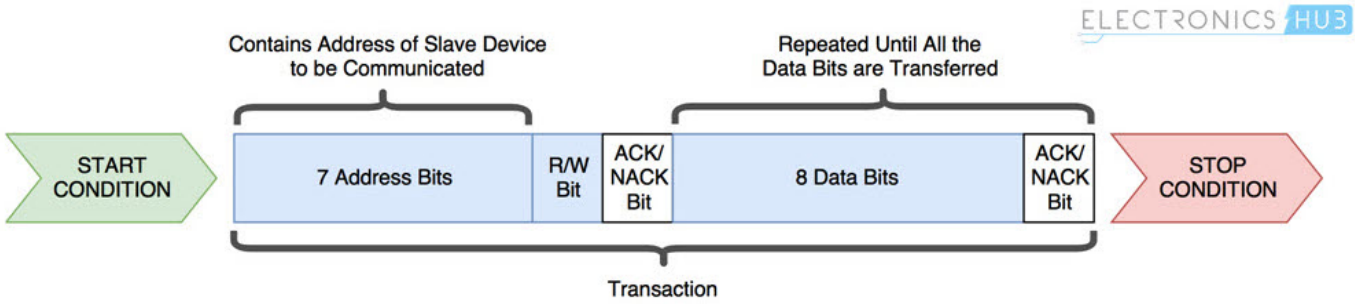
voltages. On I2C bus, there is always one Master and one or several Slave devices. Each Slave device has a unique address [2].



The initial I2C specifications defined maximum clock frequency of 100 kHz. This was later increased to 400 kHz as Fast mode. There is also a High speed mode which can go up to 3.4 MHz and there is also a 5 MHz ultra-fast mode.

In normal state both lines (SCL and SDA) are high. The communication is initiated by the master device. It generates the Start condition (S) followed by the address of the slave device (SLA). If the bit 0 of the address byte was set to 0 the master device will write to the slave device (SLA+W). Otherwise, the next byte will be read from the slave device (SLA+R). Each byte is supplemented by an ACK (low level) or NACK (high level) acknowledgment bit, which is always transmitted by the device receiving the previous byte.

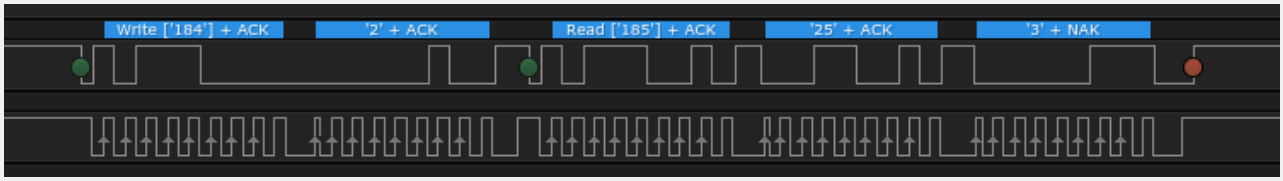
The address byte is followed by one or more data bytes, where each contains 8 bits and is again terminated by ACK/NACK. Once all bytes are read or written the master device generates Stop condition (P). This means that the master device switches the SDA line from low voltage level to high voltage level before the SCL line switches from high to low [3].



Note that, most I2C devices support repeated start condition. This means that before the communication ends with a stop condition, master device can repeat start condition with address byte and change the mode from writing to reading.

Example of I2C communication

Question: Let the following image shows several frames of I2C communication between ATmega328P and a slave device. What circuit is it and what information was sent over the bus?



Answer: This communication example contains a total of five frames. After the start condition, which is initiated by the master, the address frame is always sent. It contains a 7-bit address of the slave device, supplemented by information on whether the data will be written to the slave or read from it to the master. The ninth bit of the address frame is an acknowledgment provided by the receiving side.

Here, the address is 184 (decimal), i.e. `101_1100-0` in binary including R/W=0. The slave address is therefore 101_1100 (0x5c) and master will write data to the slave. The slave has acknowledged the address reception, so that the communication can continue.

According to the list of I2C addresses the device could be humidity/temp or pressure sensor. The signals were really recorded when communicating with the humidity and temperature sensor.

The data frame always follows the address one and contains eight data bits from the MSB to the LSB and is again terminated by an acknowledgment from the receiving side. Here, number `2` was written to the sensor. According to the DHT12 sensor manual, this is the address of register, to which the integer part of measured temperature is stored. (The following register contains its decimal part.)

Register address	Description
0x00	Humidity integer part
0x01	Humidity decimal part
0x02	Temperature integer part
0x03	Temperature decimal part
0x04	Checksum

After the repeated start, the same circuit address is sent on the I2C bus, but this time with the read bit R/W=1 (185, `1011100_1`). Subsequently, data frames are sent from the slave to the master until the last of them is confirmed by the NACK value. Then the master generates a stop condition on the bus and the communication is terminated.

The communication in the picture therefore records the temperature transfer from the sensor, when the measured temperature is 25.3 degrees Celsius.

Frame #	Description
1	Address frame with SLA+W = 184 (0x5c<<1 + 0)
2	Data frame sent to the Slave represents the ID of internal register
3	Address frame with SLA+R = 185 (0x5c<<1 + 1)
4	Data frame with integer part of temperature read from Slave
5	Data frame with decimal part of temperature read from Slave

Part 2: Synchronize repositories and create a new project

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 `lab7-i2c` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB7-I2C > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 3: I2C scanner

The goal of this task is to create a program that will verify the presence of unknown devices connected to the I2C bus by sequentially trying all address combinations.

1. Copy/paste template code to `LAB7-I2C > src > main.c` source file.

2. Use your favorite file manager and copy `timer` and `uart` libraries from the previous lab to the proper locations within the `LAB7-IC2` project.

3. In PlatformIO project, create a new folder `LAB7-I2C > lib > twi`. Within this folder, create two new files `twi.c` and `twi.h`.

   1. Copy/paste library source file to `twi.c`
   2. Copy/paste header file to `twi.h`

   The final project structure should look like this:

   ```
   LAB7-I2C            // PlatfomIO project
   ├── include         // Included file(s)
   │   └── timer.h
   ├── lib             // Libraries
   │   ├── twi         // Tomas Fryza's TWI/I2C library
   │   │   ├── twi.c
   │   │   └── twi.h
   │   └── uart        // Peter Fleury's UART library
   │       ├── uart.c
   │       └── uart.h
   ├── src             // Source file(s)
   │   └── main.c
   ├── test            // No need this
   └── platformio.ini  // Project Configuration File
   ```

4. In the lab, we are using I2C/TWI library developed by Tomas Fryza according to Microchip Atmel ATmega16 and ATmega328P manuals. Use the `twi.h` header file and add input parameters and description of the following functions.

Function name	Function parameters	Description	Example
`twi_init`	None	Initialize TWI unit, enable internal pull-up resistors, and set SCL frequency	`twi_init();`
`twi_start`			
`twi_write`			`twi_write((sla<<1) | TWI_WRITE);`
`twi_read`			
`twi_stop`			`twi_stop();`

5. Use breadboard and connect available I2C modules to Arduino Uno board, such as humidity/temperature DHT12 digital sensor, combined module with RTC DS3231 (Real Time Clock) and AT24C32 EEPROM memory, or GY-521 module (MPU-6050 Microelectromechanical systems that features a 3-axis gyroscope, a 3-axis accelerometer, a digital motion processor (DMP), and a temperature sensor). Instead of external pull-up resistors on the SDA and SCL pins, the internal ones will be used.

> **Important:** Connect the components on the breadboard only when the supply voltage/USB is disconnected!

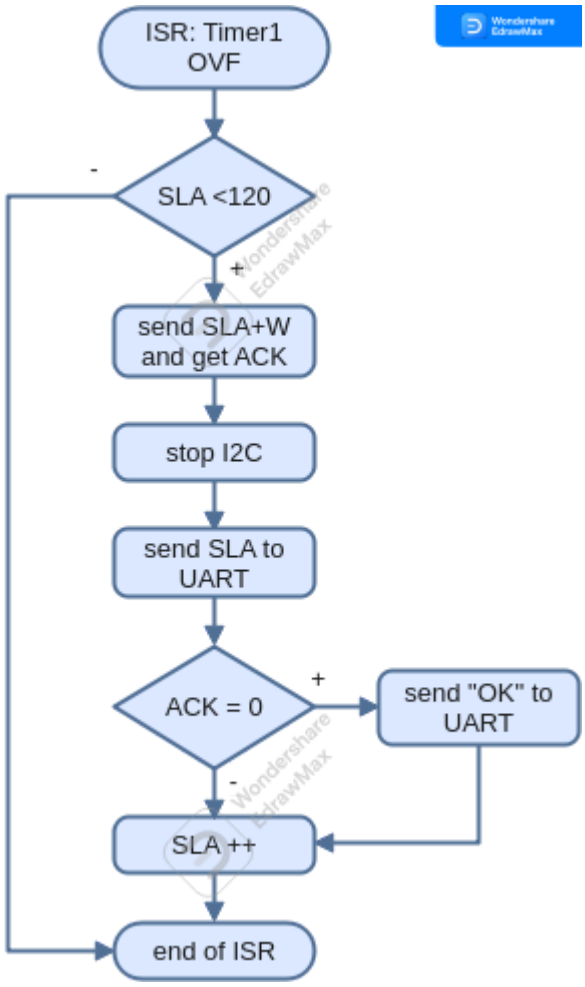
DHT12 pin	Arduino Uno pin
+	5V (or 3.3V)
SDA	SDA
-	GND
SCL	SCL

RTC+EEPROM pin	Arduino Uno pin
32K (reference clock - output)	Not connected
SQW (square-wave - output)	Not connected
SCL	SCL
SDA	SDA
VCC	5V (or 3.3V)
GND	GND

GY-521 pin	Arduino Uno pin
VCC	5V (or 3.3V)
GND	GND

GY-521 pin	Arduino Uno pin
SCL	SCL
SDA	SDA
XDA	Not connected
XCL	Not connected
ADO	Not connected
INT	Not connected

6. Go through the `main.c` file and make sure you understand each line. Build and upload the code to Arduino Uno board. Use **PlatformIO: Serial Monitor** to receive values from Arduino board. Complete the Timer1 overflow handler and test all Slave addresses from the range 8 to 119. If Slave device address is detected, send the information via UART. What Slave addresses were detected?



> **Important:** If the received characters are not displayed on the Serial Monitor, exit and restart the monitor again.

Part 4: Communication with I2C devices

1. Program an application which reads data from humidity/temperature DHT12 sensor and sends them periodically via UART to Serial Monitor or PuTTY SSH Client. Use Timer/Counter1 with a suitable

overflow time. Note that, according to the DHT12 manual, the internal DHT12 data registers have the following structure.

Register address	Description
0x00	Humidity integer part
0x01	Humidity decimal part
0x02	Temperature integer part
0x03	Temperature decimal part
0x04	Checksum

Note that a structured variable in C can be used for read values.

```
/* Global variables ---------------------------------------------------
*/
// Declaration of "dht12" variable with structure
"DHT_values_structure"
struct DHT_values_structure {
    uint8_t humInt;
    uint8_t humDec;
    uint8_t temInt;
    uint8_t temDec;
    uint8_t checksum;
} dht12;

...
dht12.humidInt = twi_read(TWI_ACK);  // Store one byte to structured
variable
```

2. (Optional) Find out how checksum byte value is calculated.

3. Program an application which reads data from RTC DS3231 chip and sends them periodically via UART to Serial Monitor or PuTTY SSH Client. Note that, according to the DS3231 manual, the internal RTC registers have the following structure.

Address	Bit 7	Bits 6:4	Bits 3:0
0x00	0	10 Seconds	Seconds
0x01	0	10 Minutes	Minutes
0x02	0	12/24 AM/PM 10 Hour	Hour
...	...	...	...

4. (Optional) Verify the I2C communication with logic analyzer.

5. (Optional) Program an application which reads data from GY-521 module. It consists of MPU-6050 Microelectromechanical systems that features a 3-axis gyroscope, a 3-axis accelerometer, a digital

motion processor (DMP), and a temperature sensor.

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. Form the UART output of I2C scanner application to a hexadecimal table such as the following figure. Note that, the designation RA represents I2C addresses that are reserved and cannot be used for slave circuits.

```
Scan I2C-bus for slave devices:

     .0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .a .b .c .d .e .f
0x0.: RA RA RA RA RA RA RA RA -- -- -- -- -- -- -- --
0x1.: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x2.: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x3.: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x4.: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x5.: -- -- -- -- -- -- -- 57 -- -- -- -- -- -- -- --
0x6.: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
0x7.: -- -- -- -- -- -- -- -- RA RA RA RA RA RA RA RA

Number of detected devices: 1
```

2. Program functions that will be able to read only one value from the RTC DS3231 at a time.

Function name	Function parameters	Description	Example
`rtc_read_seconds`	None	Read seconds from RTC	`rtc.secs = rtc_read_seconds();`
`rtc_read_minutes`	None	Read minutes from RTC	`rtc.mins = rtc_read_minutes();`
`rtc_read_hours`	None	Read hours from RTC	`rtc.hours = rtc_read_hours();`

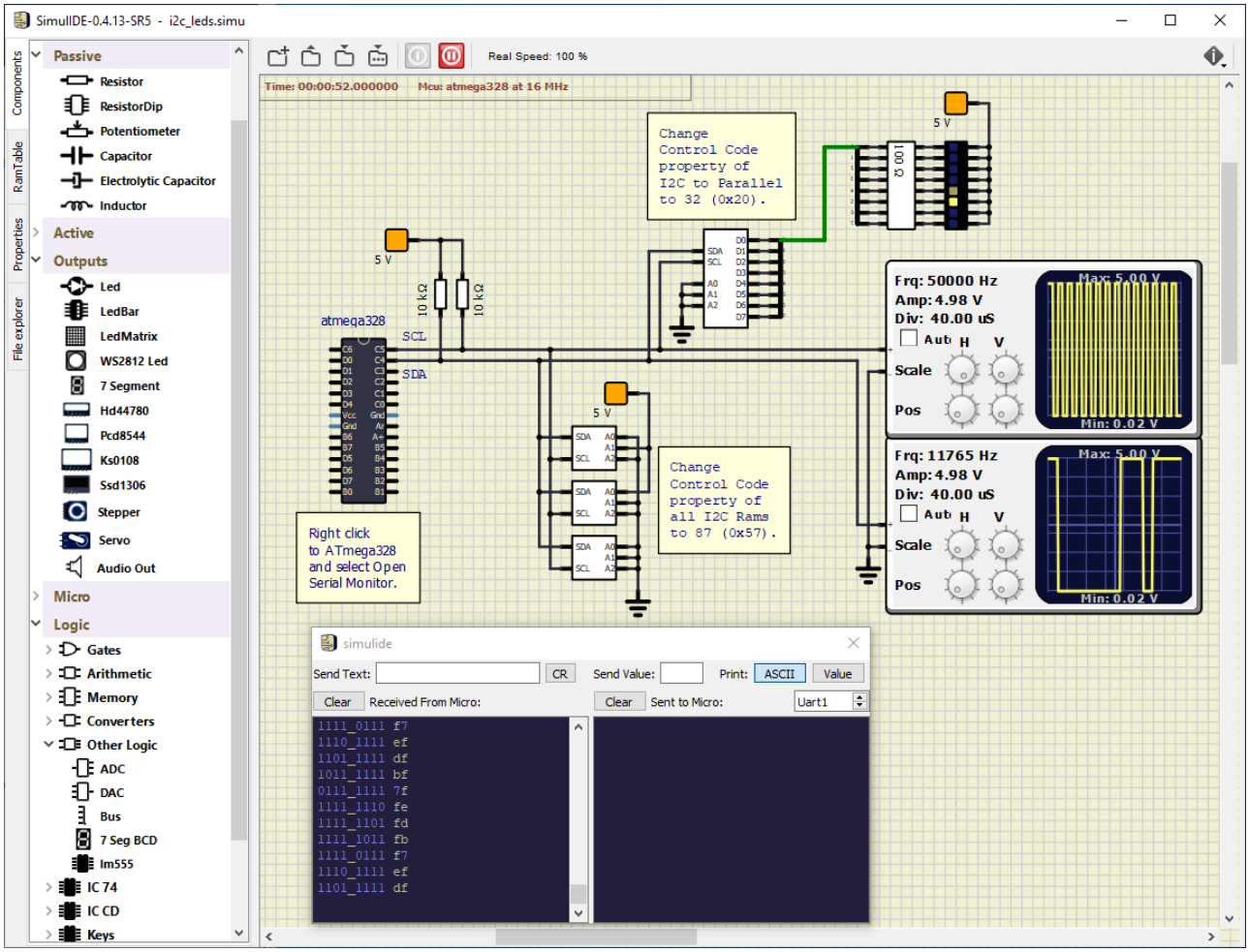
3. Program the functions that will be able to save the current time values to the RTC DS3231.

4. Consider an application for temperature and humidity measurements. Use sensor DHT12, real time clock DS3231, LCD, and one LED. Every minute, the temperature, humidity, and time is requested from Slave devices and values are displayed on LCD screen. When the temperature is above the threshold, turn on the LED.

   Draw a flowchart of `TIMER1_OVF_vect` (which overflows every 1 sec) for such Meteo station. The image can be drawn on a computer or by hand. Use clear description of individual algorithm steps.

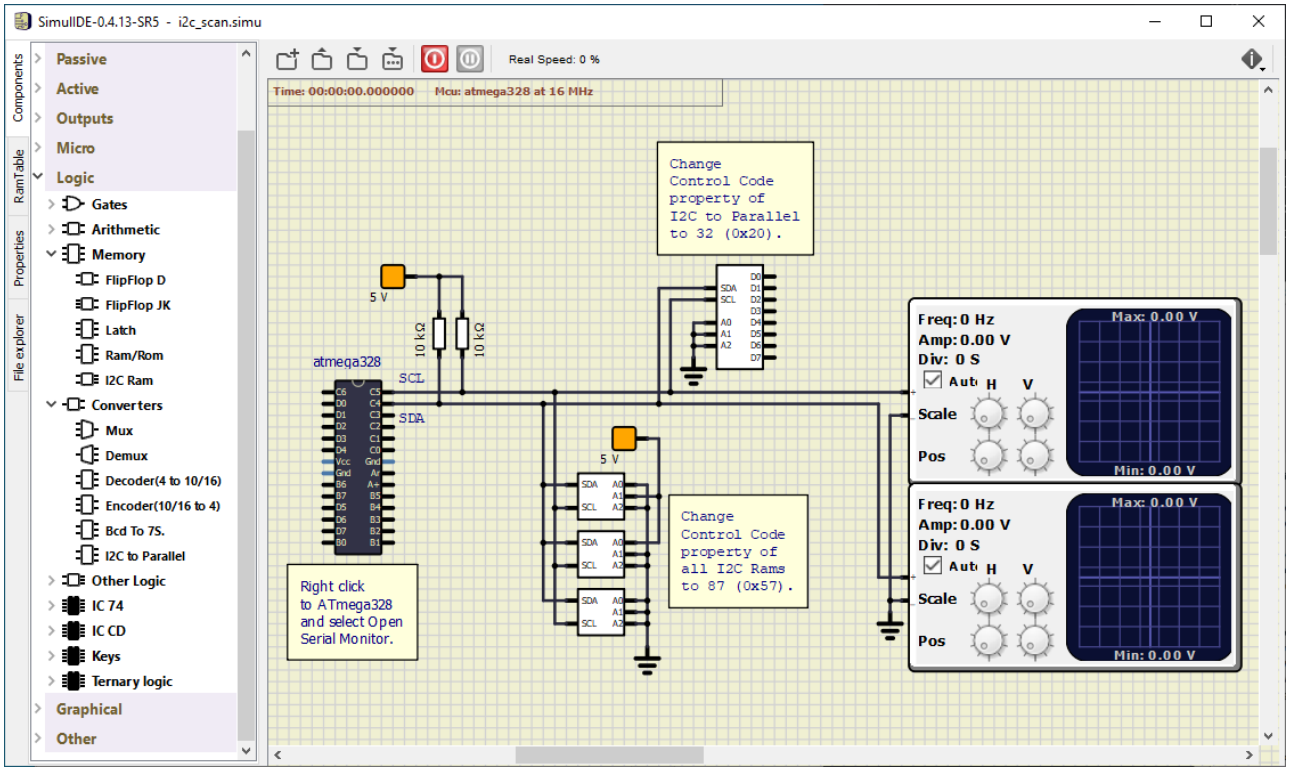
5. Draw a timing diagram of I2C signals when calling function `rtc_read_years()`. Let this function reads one byte-value from RTC DS3231 address `06h` (see RTC datasheet) in the range `00` to `99`. Specify when the SDA line is controlled by the Master device and when by the Slave device. Draw the whole request/receive process, from Start to Stop condition. The image can be drawn on a computer (by WaveDrom for example) or by hand. Name all parts of timing diagram.

6. In the SimulIDE application, create the circuit with eight active-low LEDs connected to I2C to Parallel expander. You can use individual components (ie. 8 resistors and 8 LEDs) or single **Passive > ResistorDip** and **Outputs > LedBar** according to the following figure. Several signals can form a bus **Logic > Other Logic > Bus**, as well.



Create an application that sequentially turns on one of the eight LEDs. ie first LED0, then LED1 and finally LED7, then start again from the beginning. Use Timer/Counter1 and change the value every 262 ms. Send the status of the LEDs to the UART. Try to complement the LED controls according to the Knight Rider style, ie light the LEDs in one direction and then in the opposite one.

7. In the SimulIDE application, use the following components: I2C Ram (**Components > Logic > Memory > I2C Ram**), I2C to Parallel (**Components > Logic > Converters > I2C to Parallel**) and create a schematic according to the following figure. Also, change **Control Code** property of all I2C devices. These codes represent the I2C addresses of the Slave circuits. Pins A2, A1, A0 allow you to specify part of the device address. Thus, up to 8 (2^3 = 8) identical devices can be connected and it will be possible to distinguish them. External pull-up resistors on SDA and SCL signals must be used for correct simulation.



Program an application that communicates with memory modules using the I2C bus. Store random data in the first ten address positions of the first and second memory modules. Then copy 5 values from the first memory to the third and another 5 values from the second memory to the third one. Send the first ten values from each memory module to the UART.

8. Finish all (or several) 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. Tomas Fryza. Schematic of Arduino Uno board
2. Ezoic. I2C Info - I2C Bus, Interface and Protocol
3. Electronicshub.org. Basics of I2C Communication | Hardware, Data Transfer, Configuration
4. Adafruit. List of I2C addresses
5. Aosong. Digital temperature DHT12
6. NXP. I2C-bus specification and user manual

7. Maxim Integrated. DS3231, Extremely accurate I2C-Integrated RTC/TCXO/Crystal

8. LastMinuteEngineers. Interface DS3231 Precision RTC Module with Arduino

9. Tomas Fryza. Useful Git commands

Lab 8: Assembly language and project documentation

Learning objectives

After completing this lab you will be able to:

- Use basic AVR instructions
- Convert AVR instruction to hexadecimal machine code
- Pass parameters from C code to assembly and vice versa

The purpose of the laboratory exercise is to understand the AVR instruction set and how the individual instructions are translated into machine code. The main goal is to learn to combine higher and lower programming language in one project.

Table of contents

Components list

- Arduino Uno board, USB cable

Pre-Lab preparation

1. Use **AVR® Instruction Set Manual** from Microchip Online Technical Documentation, find the description of selected instructions, and complete the table.

Instruction	Operation	Description	Cycles
`add Rd, Rr`			
`andi Rd, K`	Rd = Rd and K	Logical AND between register Rd and 8-bit constant K	1
`bld Rd, b`			
`bst Rd, b`			
`com Rd`			
`eor Rd, Rr`			
`mul Rd, Rr`			
`pop Rd`			

Instruction	Operation	Description	Cycles
`push Rr`			
`ret`			
`rol Rd`			
`ror Rd`			

Part 1: Synchronize repositories and create a new project

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 `lab8-asm` for `Arduino Uno` board and change project location to your local repository folder `Documents/digital-electronics-2`.

3. IMPORTANT: Rename `LAB8-ASM > src > main.cpp` file to `main.c`, ie change the extension to `.c`.

Part 2: Assembly language

Any program is just a series of instructions, that fetch and manipulate data. In most applications, this means reading the inputs, checking their status, switching on the outputs accordingly, or transferring data to another device, such as a display or serial line.

A number of simple binary instructions are used to perform these basic tasks, and each has an equivalent assembly language instruction that people can understand. Using assembly language allows you to understand much more about how the micro controller works and how it is put together. It also produces very small and therefore fast code. The disadvantage is that you as a programmer have to do everything, including memory management and program structure, which can be very time consuming.

To avoid this, higher-level languages are more often used to write programs for micro controllers, especially C but also Basic and Java. A high level means that each line of C (or other language) can be translated into one or many lines of assembly language.

The compiler also deals with program structure and memory management, making writing code much easier. Commonly used routines, such as delays, can also be stored in libraries and easily reused. In addition, the C compiler makes it easier to work with numbers larger than one byte.

For time- or memory space-critical applications, it can often be desirable to combine C code (for easy maintenance) and assembly code (for maximal speed or minimal code size) together. To allow a program written in C to call a subroutine written in assembly language, you must be familiar with the register usage convention of the C compiler [2].

Parameters between C and assembly may be passed via registers and/or the Stack memory. Using the register way, parameters are passed via R25:R8 (18 regs, first function parameter is stored in R25:24, second in R23:22, etc.). If the parameters require more memory, then the Stack is used to pass additional parameters. Return value is placed in the same registers, ie. an 8-bit value gets returned in R24, an 16-bit value in two registers R25:24, an 32-bit value gets returned in four registers R25:22, and an 64-bit value gets returned in R25:18 [3].

1. Copy/paste template code to `LAB8-ASM > src > main.c` source file.
2. Use your favorite file manager and copy `timer` and `uart` libraries from the previous labs to the proper locations within the `LAB8-ASM` project.
3. In PlatformIO project, create two new files `lfsr.S` and `mac.S` within `LAB8-ASM > src` source folder.

    1. Copy/paste assembly Multiply–and-Accumulate file to `mac.S`
    2. Copy/paste assembly LFSR generator to `lfsr.S`

    The final project structure should look like this:

```
LAB8-ASM            // PlatfomIO project
├── include         // Included file(s)
│   └── timer.h
├── lib             // Libraries
│   └── uart        // Peter Fleury's UART library
│       ├── uart.c
│       └── uart.h
├── src             // Source file(s)
│   ├── lfsr.S      // Assembly implementation of LFSR-based generator
│   ├── mac.S       // Assembly example of Multiply-and-Accumulate
│   └── main.c
├── test            // No need this
└── platformio.ini  // Project Configuration File
```

4. Go through the `main.c` file and make sure you understand each line. Use **AVR® Instruction Set Manual** from Microchip Online Technical Documentation, find the description of instructions used in `mac.S`, and complete the table.

Instruction	Operation	Description	Cycles
`add Rd, Rr`			
`mul Rd, Rr`			
`ret`			

5. Use manual's 16-bit Opcodes and convert used instructions to hexadecimal.

Instruction	Binary opcode	Hex opcode	Compiler Hex opcode
`add r24, r0`			

Instruction	Binary opcode	Hex opcode	Compiler Hex opcode
`mul r22, r20`			
`ret`	`1001_0101_0000_1000`	`95 08`	

6. Build and upload the code to Arduino Uno board. Use **PlatformIO: Serial Monitor** to receive values from Arduino board.

7. In Visual Studio Code select **Terminal > New Terminal Ctrl+Shift+;** and run the following command to generate the listing file:

```
# Windows:
C:\Users\YOUR-LOGIN\.platformio\packages\toolchain-atmelavr\bin\avr-objdump -S -d -m avr .pio/build/uno/firmware.elf > firmware.lst

# Linux:
~/.platformio/packages/toolchain-atmelavr/bin/avr-objdump -S -d -m avr .pio/build/uno/firmware.elf > firmware.lst
```

From the project root folder, open the generated liting file `firmware.lst`. Compare your conversion from previous table with the compiler's.

> **Note:** By default, there is no highlighting mode for `*.lst` listing file. You can select the Language mode by clicking on the **Plain Text** identifier in the lower right corner of VS Code. Select **Assembly** mode, or if you have installed the `AVR Support` extension, choose **AVR Assembler** mode.

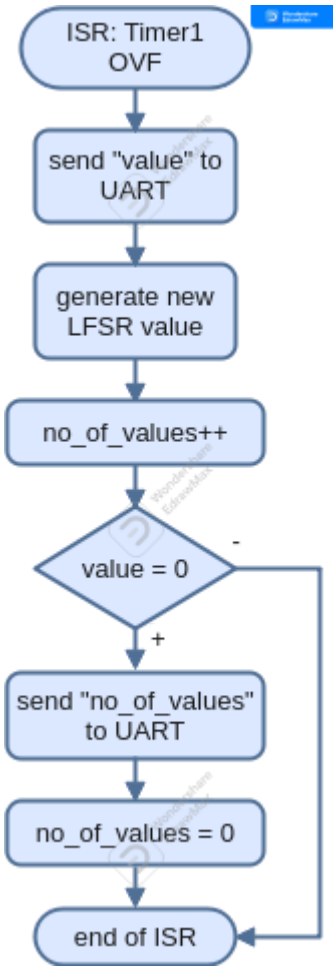
Part 3: LFSR-based pseudo random generator

A linear-feedback shift register (LFSR) is a shift register whose input bit is a linear function of its previous state. The bit positions that affect the next state are called the taps. We can use this type of functions in many application such as counters, crypto, CRC generation, scrambling/descrambling algorithm, etc.

There are two different (but equivalent) types of LFSR implementation the Fibonacci and the Galois. The LFSR can be implemented using XOR or XNOR primitive functions [4].

A maximum-length LFSR produces an m-sequence i.e. it cycles through all possible 2^N−1 states which look like pseudo-random values. If XOR gates are used, the illegal state is all zeros because this case will never change. A state with all ones is illegal when using an XNOR feedback, because the counter would remain locked-up in this state.

1. Consider a 4-bit shift register whose input (LSB bit) is formed by an XNOR gate with taps [4, 3] and the initial value is 0000 [5]. Explore LFSR algorithm within `lfsr4_fibonacci_asm` assembly function, complete Timer1 overflow handler and generate 4-bit pseudo-random sequences for different Tap positions. How many states are generated for every settings?



Tap position	Generated values	Length
4, 3		
4, 2		
4, 1		

2. Change LFSR tap positions in `lfsr4_fibonacci_asm` function and generate 5-, 6-, and 7-bit versions of pseudorandom sequence. Do not forget to change the binary mask used to clear unused bits in input/output register.

Tap position	Length

Part 4: Generate documentation from source code

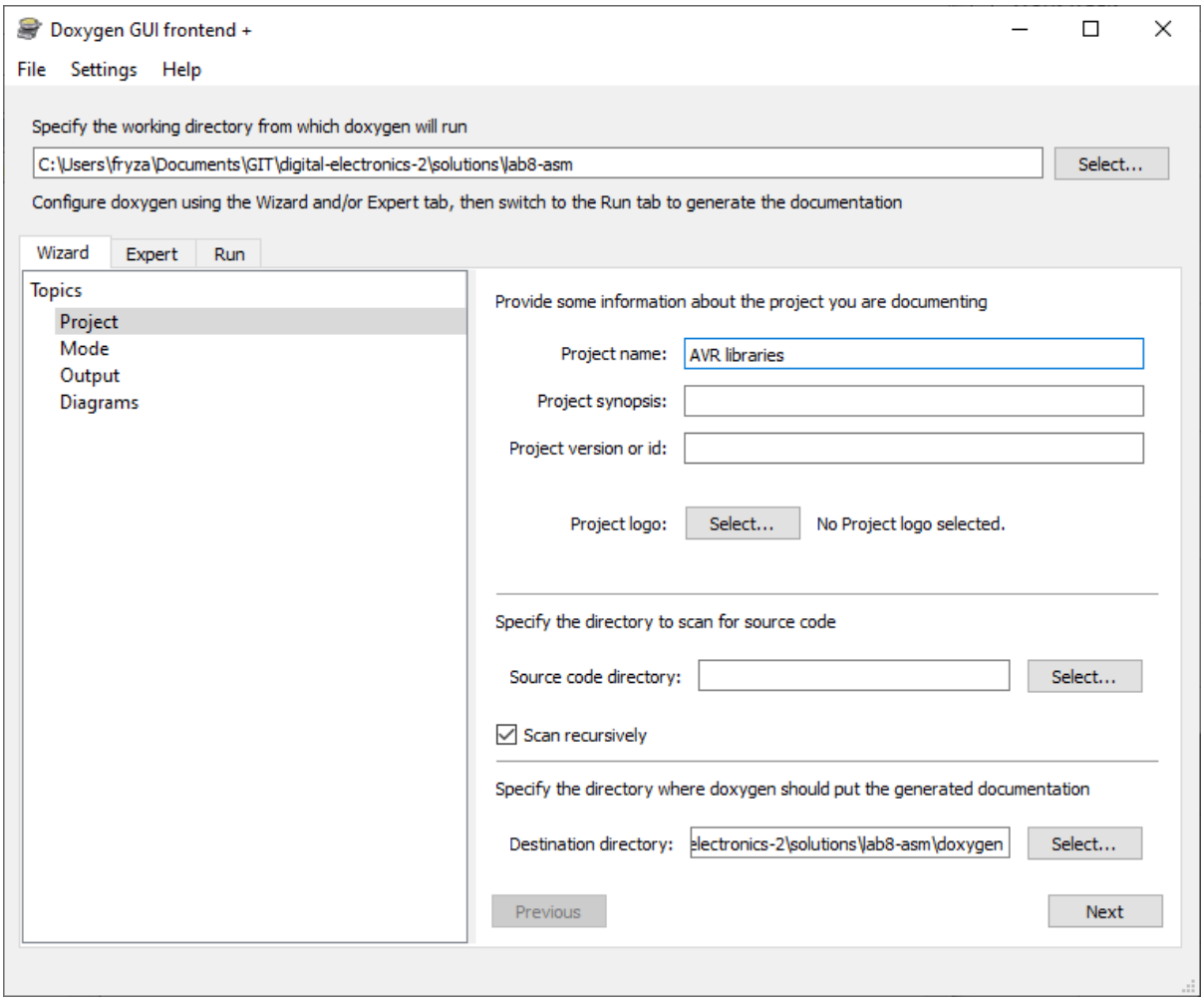
Doxygen is a free, multiplatform (Linux, Windows, Mac, ...) tool for easy generation of program manuals. It supports popular programming languages such as C++, C, C#, PHP, Java, Python, Fortran. Doxygen also supports the hardware description language VHDL. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix

man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.

Doxygen uses several keywords that are inserted into your block comments. For C, these comments must begin with a triple character with two asterisks:

```
/**
 * Doxygen will search this block
 */

/*
 * Classic C block comment; Doxygen will not search it
 */
```

1. Open Doxywizard and set the basic settings as follows:
    1. Select working directory with your project `Documents\digital-electronics-2\lab8-asm`
    2. In **Wizard > Project** set **Project name**
    3. In **Wizard > Project** check **Scan recursively**
    4. In **Wizard > Project** select **Destination directory** to new folder within your project `Documents\digital-electronics-2\lab8-asm\doxygen`
    5. In **Wizard > Mode** select programming language to **Optimized for C or PHP output**
    6. In **Wizard > Output > HTML** unselect **With search function**
    7. In **Wizard > Output** unselect LaTeX and keep just HTML generation
    8. In **Run** click to button **Run doxygen** and then **Show HTML output**



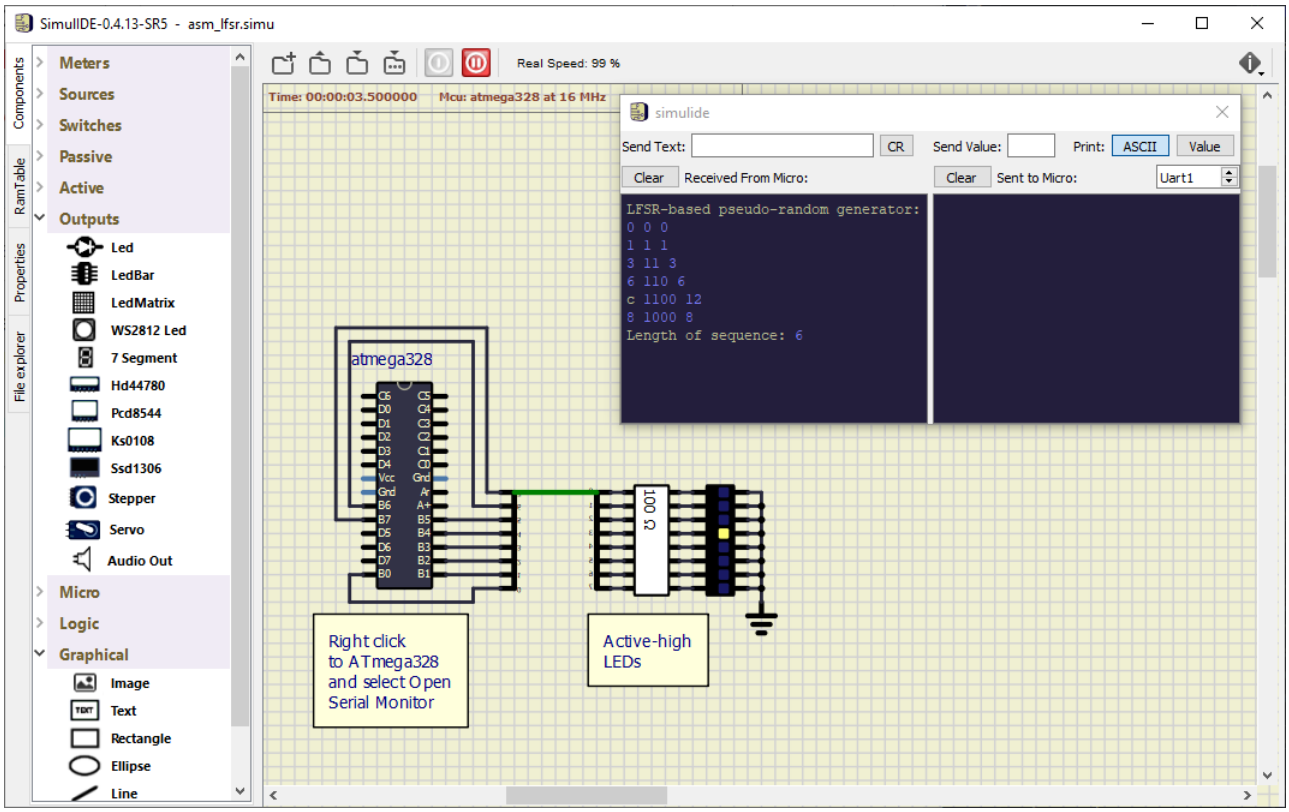
> **Note:** Complete guide on using Doxygen to document C source code is here.

2. 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. In `lfsr.S` file, program the assembly function `uint8_t lfsr8_fibonacci_asm(uint8_t value)`, which generates a 8-bit LFSR sequence with Tap positions 8, 6, 5, 4. What is the sequence length? What is the duration of the function in CPU cycles? Use eight LEDs and display each generated LFSR value. Simulate the application in SimulIDE.



2. In `main.c` file, program the C function `uint8_t lfsr4_fibonacci_c(uint8_t value)`, which generates a 4-bit LFSR sequence with a maximum length. In the `.lst` file compare both functions, in assembly and your C-realization. What is the duration of both functions in CPU cycles?

Function	Number of instructions	Total number of CPU cycles
`lfsr4_fibonacci_asm`		
`lfsr4_fibonacci_c`		

3. Program a 16-bit LFSR-based pseudo-random generator in assembly language and display values at UART. What LFSR taps provide the maximum length of generated sequence?

4. In assembly, program a function `void burst_asm(uint8_t length)` to generate a variable number of short pulses at output pin. Let the pulse width be the shortest one. Write the same function `void burst_c(uint8_t length)` in C and compare duration of both functions. Use a logic analyzer, verify the pulse width and calculate the CPU frequency accordingly.

5. Draw a flowchart of function `void burst_c(uint8_t number)` which generates a variable `number` of short pulses (ie. combination of high and low levels) at output pin PB5. Let the pulse width be the shortest one without any delay. The image can be drawn on a computer or by hand. Use clear descriptions of the individual steps of the algorithms.

6. In assembly, program your own delay function with one parameter that specifies the delay time in microseconds. Use a logic analyzer or oscilloscope to verify the correct function when generating pulses on the ATmega328P output pin. Use this function to generate the following acoustic tones: C2, D2, E2, F2, G2, and A2.

7. In assembly, program an interrupt service routine for Timer/Counter1 overflow.

8. In assembly, program the `uint8_t sop_asm(*uint8_t a, *uint8_t b, uint8_t length)` function to calculate the sum of the products of two integer vectors `a` and `b`, which have the same number of elements `length`. Transmit the SoP result via UART. For simplicity, consider only 8-bit sum and multiplication operations.

   Write the same function `uint8_t sop_c(*uint8_t a, *uint8_t b, uint8_t length)` in C language and compare the duration of both functions using the file `.lss`.

9. Finish all (or several) 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. Microchip Atmel. AVR® Instruction Set Manual
2. William Barnekow. Mixing C and assembly language programs
3. Chris Taylor. Mixing C and Assembly
4. Surf-VHDL. How to implement an LFSR in VHDL
5. Clive Maxfield. Tutorial: Linear Feedback Shift Registers (LFSRs) – Part 1
6. Doxygen tool
7. Embedded Inventor. Complete Guide On Using Doxygen To Document C Source Code..!!
8. Tomas Fryza. Useful Git commands
9. B. H. Suits. Physics of Music - Notes

Project C

Topics

Topics will be presented one week before the project starts.

Instructions

The objective of this group project is to collaborate within small teams, explore a chosen topic, develop innovative solutions, simulate and execute these solutions, generate project documentation, and present the final outcomes. Team members are responsible for organizing and assigning roles and tasks among themselves.

- Students will work on a project **in the labs** during the 9th to 13th weeks of the semester, with the practical demonstration scheduled for the last week.
- Using BUT e-learning, students should submit a link to the GitHub repository containing the C project, required images, documents, and a descriptive README file. The submission deadline is the day before the demonstration."
- The AVR code must be written in C and/or Assembly and must be implementable on an Arduino Uno board using the toolchains provided during the semester, specifically PlatformIO. The use of Arduino frameworks/libraries or any other development tools is not permitted!
- Create your own libraries for new components.
- Physical implementation on AVR is required, not just computer simulation.

Recommended README.md file structure

Team members

- Member 1 (responsible for ...)
- Member 2 (responsible for ...)
- Member 3 (responsible for ...)

Theoretical description and explanation

Enter a description of the problem and how to solve it.

Hardware description of demo application

Insert descriptive text and schematic(s) of your implementation.

Software description

Put flowchats of your algorithm(s) and direct links to source files in `src` or `lib` folders.

Instructions

Write an instruction manual for your application, including photos or a link to a video.

References

1. Write your text here.
2. ...