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.

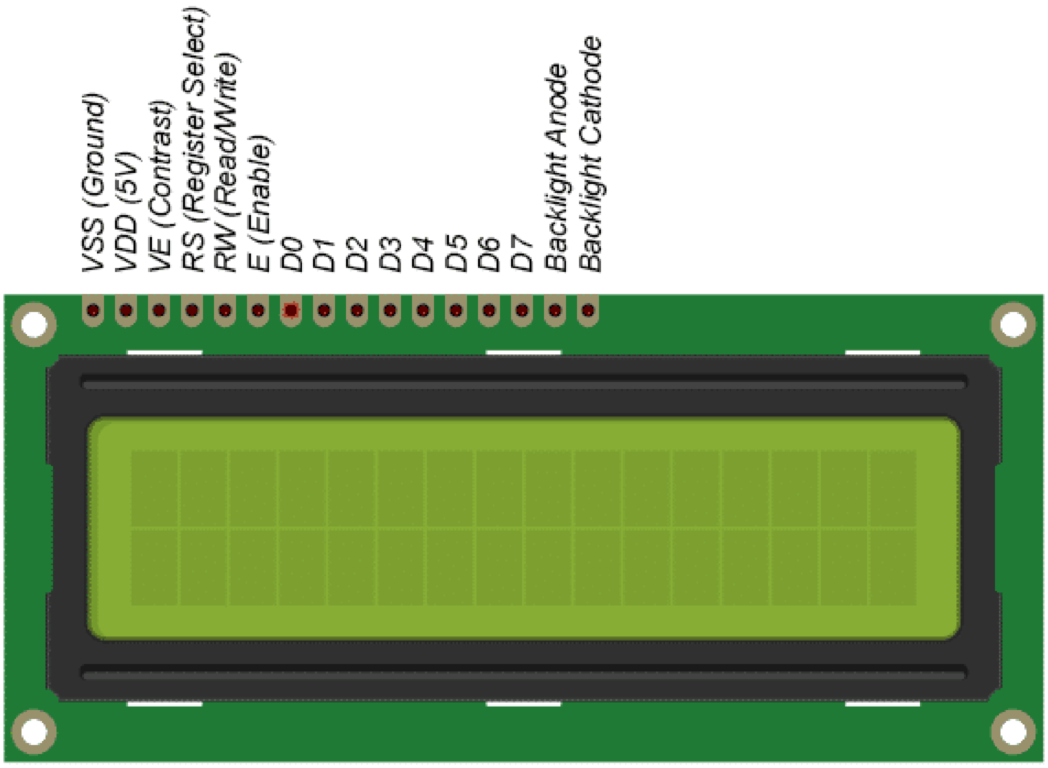
The control is based on the Hitachi HD44780 chipset (or a compatible), which is found on most text-based LCDs, and hence the driving software remains the same even for different screen sizes. The driver contains instruction set, character set, and in addition you can also generate your own characters.

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):

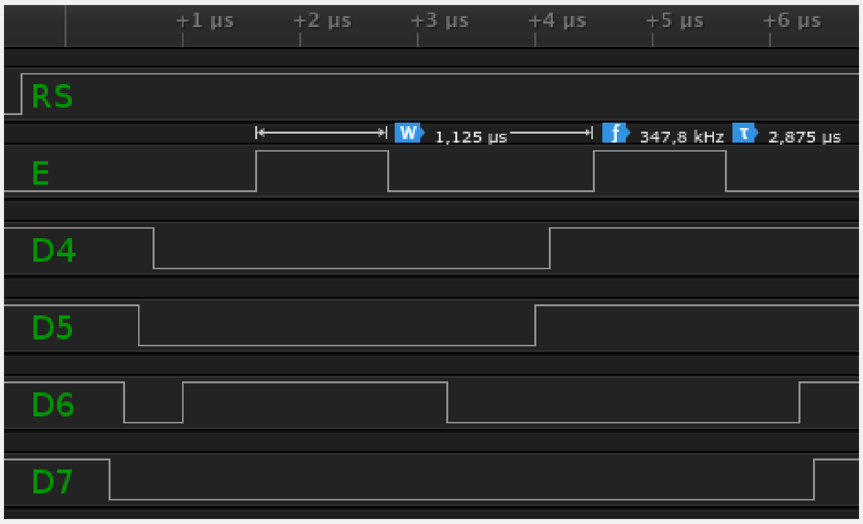
- RS - register select. Selects the data or instruction register inside the HD44780
- E - enable. This loads the data into the HD44780 on the falling edge
- (at LCD keypad shield, R/W pin is permanently connected to GND)
- D7:4 - Upper nibble used in 4-bit mode
- K - Back-light cathode



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].

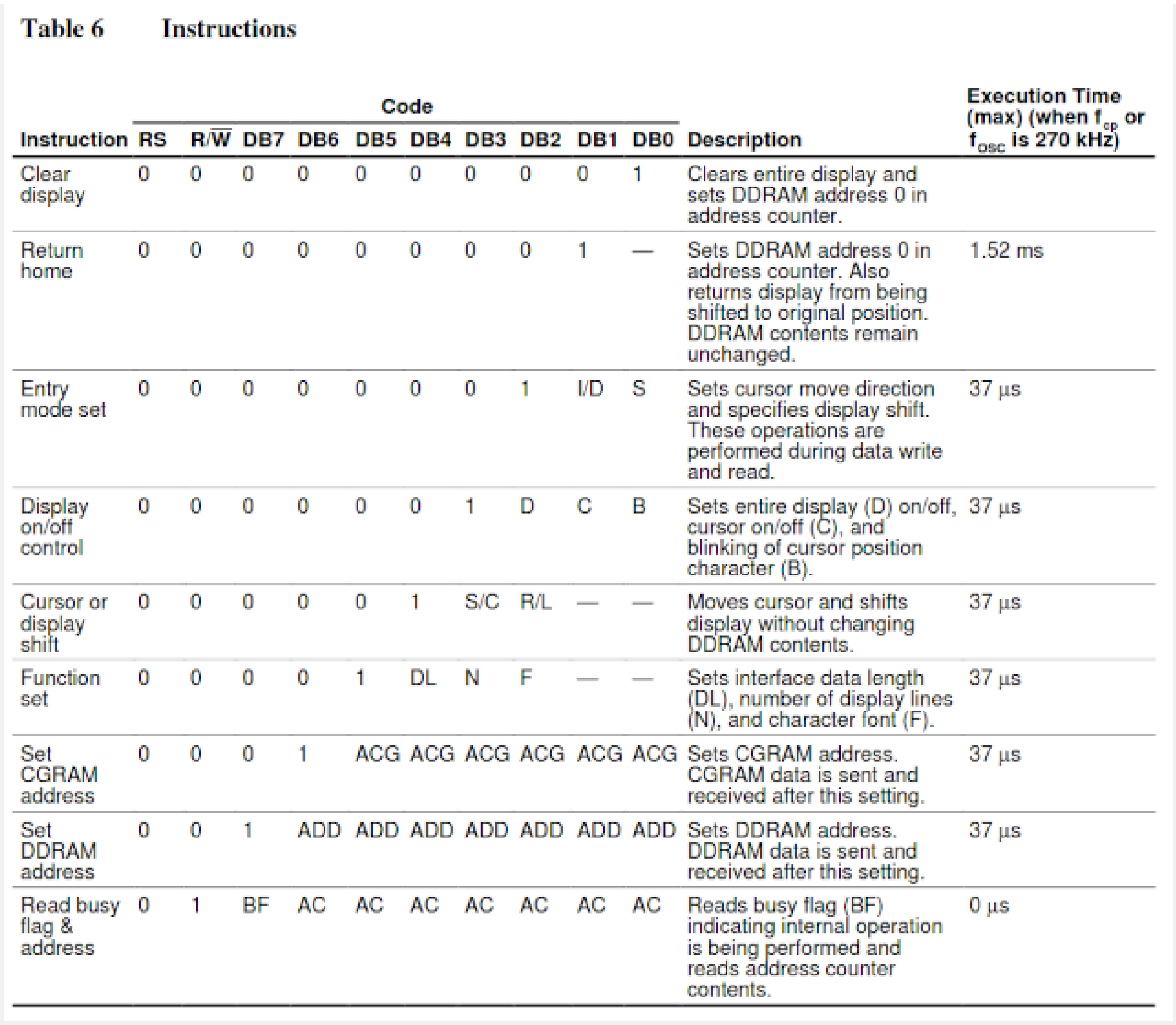


Table 6 Instructions

Instruction	Code										Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
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

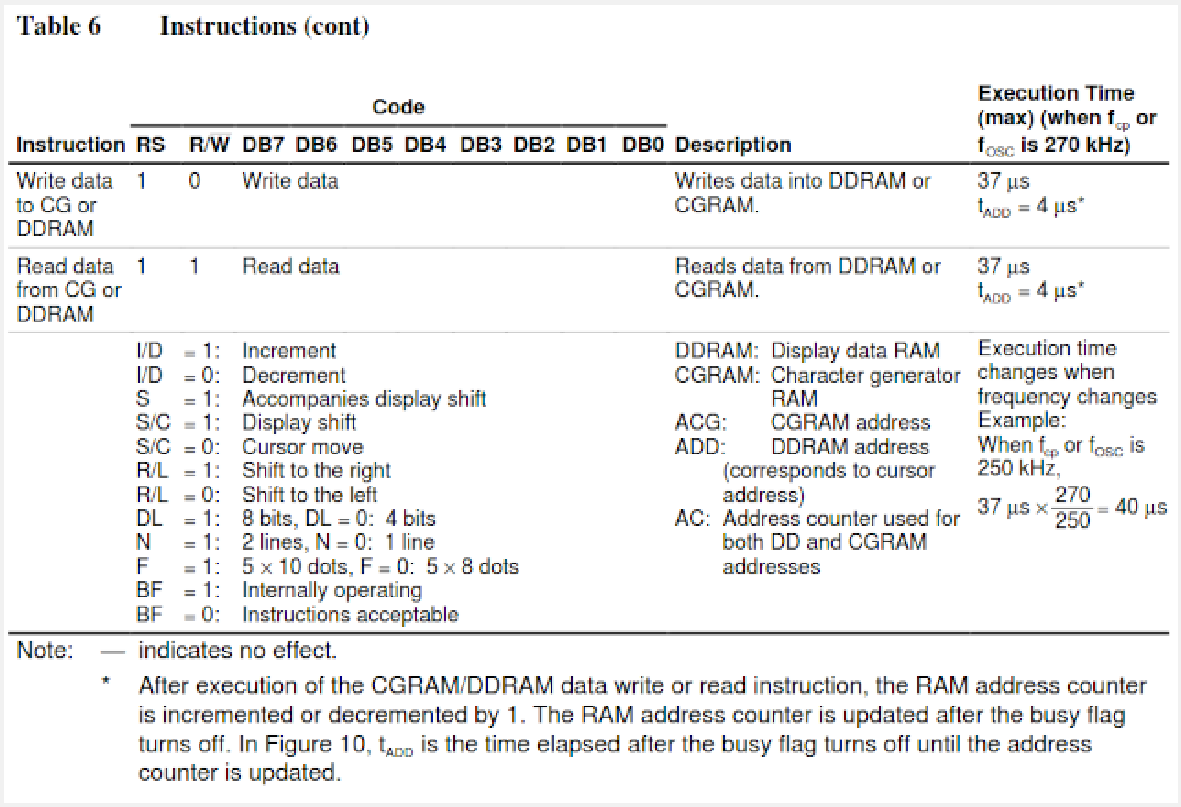


Table 6 Instructions (cont)

Instruction	Code										Description	Execution Time (max) (when f_{cp} or f_{OSC} is 270 kHz)
	RS	R/$\overline{W}$	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
Write data to CG or DDRAM	1	0	Write data								Writes data into DDRAM or CGRAM.	37 µs $t_{ADD} = 4\ \mu s$*
Read data from CG or DDRAM	1	1	Read data								Reads data from DDRAM or CGRAM.	37 µs $t_{ADD} = 4\ \mu s$*
	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.

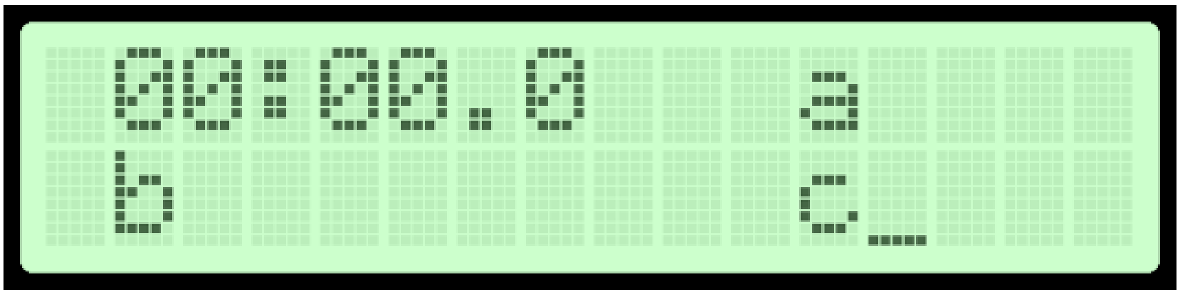
5. In PlatformIO project, create a new folder `LAB4-LCD > lib > lcd`. Within this folder, create three new files `lcd.c`, `lcd.h`, and `lcd_definitions.h`. The final project structure should look like this:

```
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".

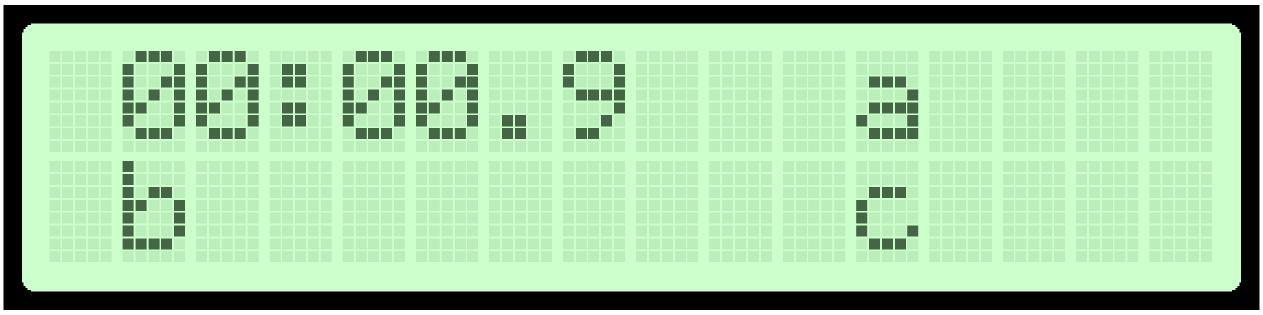


> **Note:** The figure above was created by online LCD Display Screenshot Generator.

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

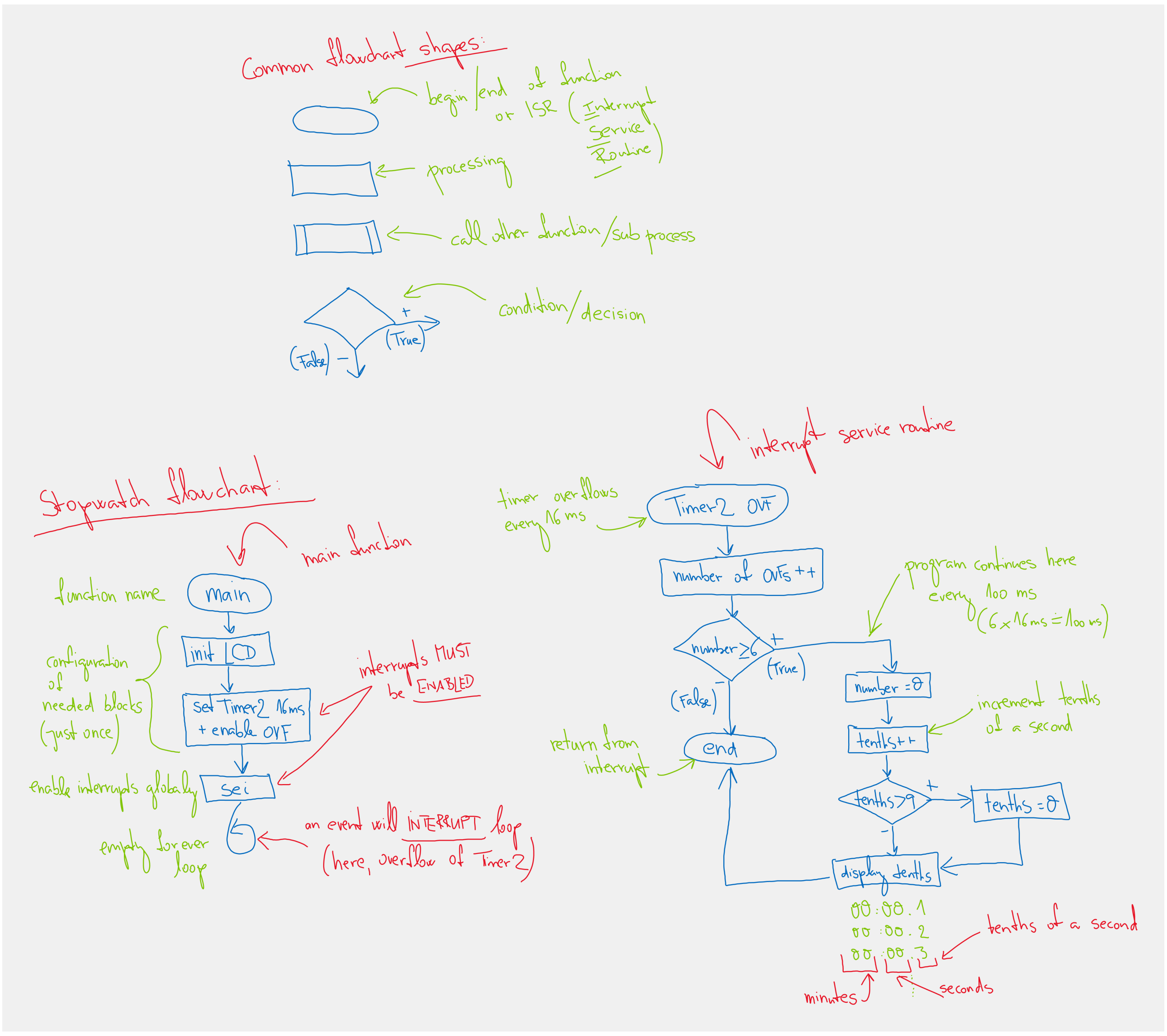
1. Use Timer/Counter2 16-ms overflow and update the stopwatch LCD value approximately every 100 ms (6 x 16 ms = 100 ms) as explained in the previous lab. Display tenths of a second only in the form `00:00.tenths`, ie let the stopwatch counts from `00:00.0` to `00:00.9` and then starts again.



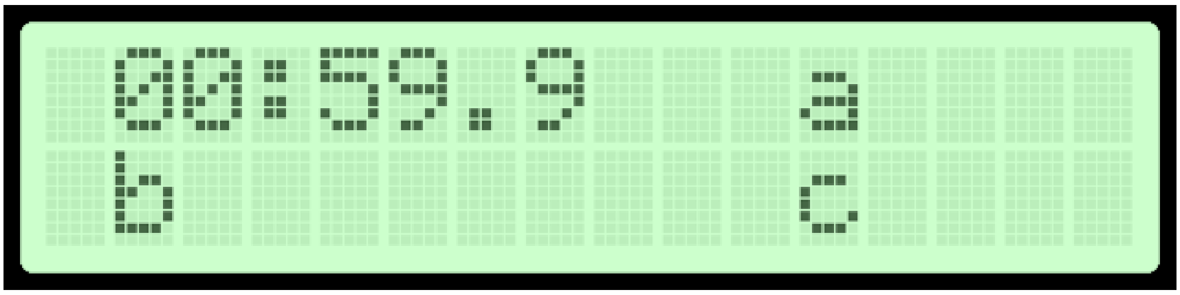
IMPORTANT: Because library functions only allow to display a string (`lcd_puts`) or individual characters (`lcd_putc`), the variables' number values need to be converted to such strings. To do this, use the `itoa(number, string, num_base)` function from the standard `stdlib.h` library. The `num_base`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

```c
#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.


Common flowchart shapes:
begin/end of function or ISR (Interrupt Service Routine)
processing
call other function/sub process
condition/decision
+ (True)
(False) -
Stopwatch flowchart:
main function
function name
main
configuration of needed blocks (just once)
init LCD
Set Timer2 16ms + enable OVF
interrupts MUST be ENABLED
enable interrupts globally
sei
empty forever loop
an event will INTERRUPT loop (here, overflow of Timer2)
interrupt service routine
timer overflows every 16 ms
Timer2 OVF
number of OVFs++
number≥6
+
(True)
(False)
program continues here every 100 ms (6×16ms ≐ 100 ms)
number=0
increment tenths of a second
tenths++
return from interrupt
end
tenths>9
+
-
tenths=0
display tenths
00:00.1
00:00.2
00:00.3
tenths of a second
minutes
seconds

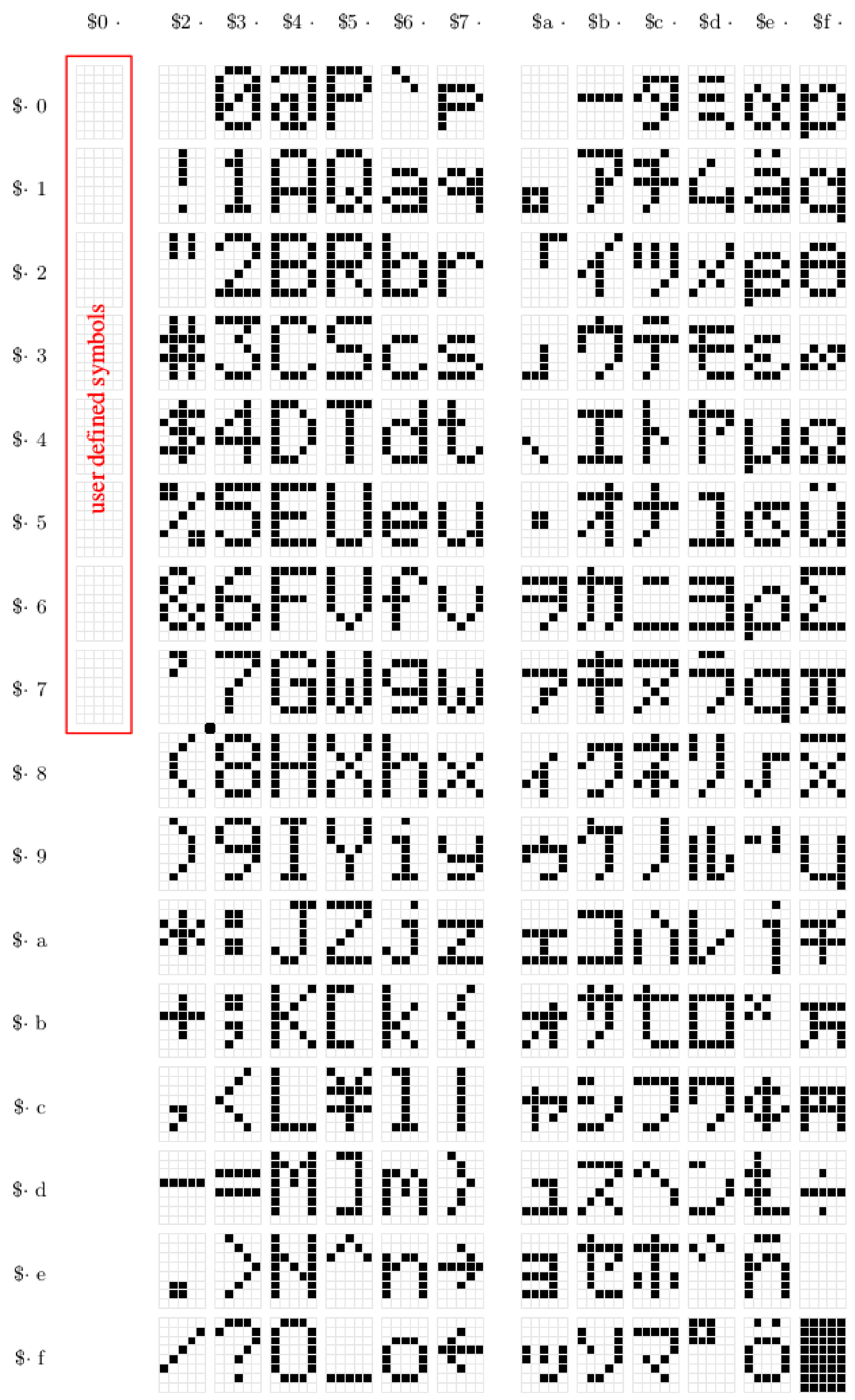
3. Complete the stopwatch flowchart of the Timer/Counter2 interrupt service routine with seconds. According to the flowchart, program the `ISR()` source code. Let the stopwatch counts from `00:00.0` to `00:59.9` and then starts again.



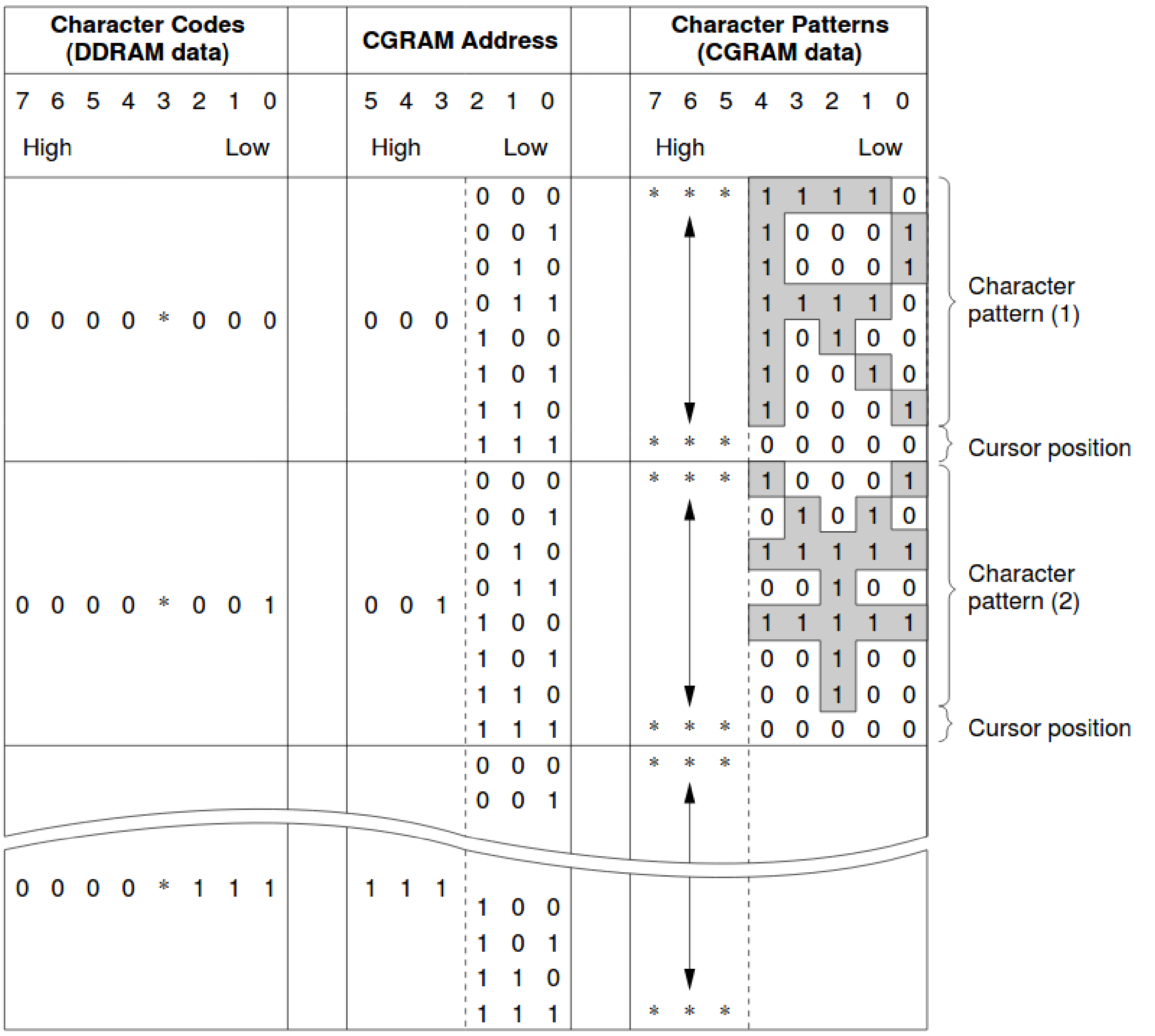
Part 5: Custom characters

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.


$0 ·
$2 ·
$3 ·
$4 ·
$5 ·
$6 ·
$7 ·
$a ·
$b ·
$c ·
$d ·
$e ·
$f ·
$· 0
$· 1
$· 2
$· 3
$· 4
$· 5
$· 6
$· 7
$· 8
$· 9
$· a
$· b
$· c
$· d
$· e
$· f
user defined symbols

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.



Character Codes (DDRAM data)	CGRAM Address		Character Patterns (CGRAM data)		
7 6 5 4 3 2 1 0	5 4 3	2 1 0	7 6 5	4 3 2 1 0	
High Low	High Low		High Low		
0 0 0 0 * 0 0 0	0 0 0	0 0 0	* * *	1 1 1 1 0	Character pattern (1)
		0 0 1		1 0 0 0 1	
		0 1 0		1 0 0 0 1	
		0 1 1		1 1 1 1 0	
		1 0 0		1 0 1 0 0	
		1 0 1		1 0 0 1 0	
		1 1 0		1 0 0 0 1	
		1 1 1	* * *	0 0 0 0 0	Cursor position
0 0 0 0 * 0 0 1	0 0 1	0 0 0	* * *	1 0 0 0 1	Character pattern (2)
		0 0 1		0 1 0 1 0	
		0 1 0		1 1 1 1 1	
		0 1 1		0 0 1 0 0	
		1 0 0		1 1 1 1 1	
		1 0 1		0 0 1 0 0	
		1 1 0		0 0 1 0 0	
		1 1 1	* * *	0 0 0 0 0	Cursor position
		0 0 0	* * *		
		0 0 1			
0 0 0 0 * 1 1 1	1 1 1	1 0 0			
		1 0 1			
		1 1 0			
		1 1 1	* * *		

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);
    ...
```

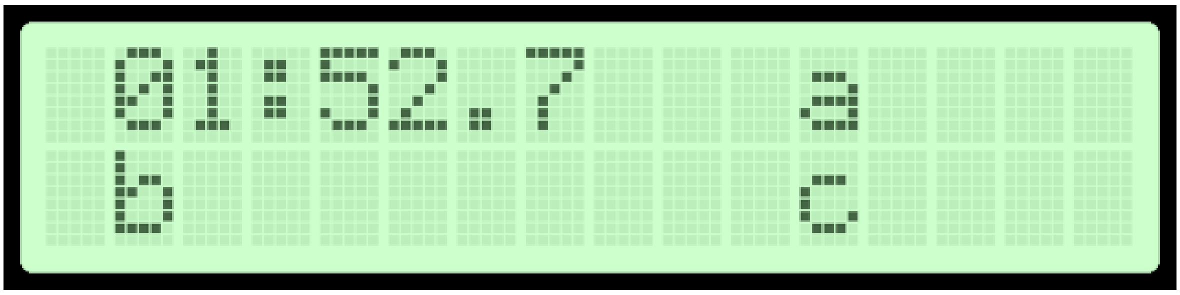
2. Design at least one more custom character, store it in CGRAM memory according to the previous code, and display all new characters on the LCD screen.

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.

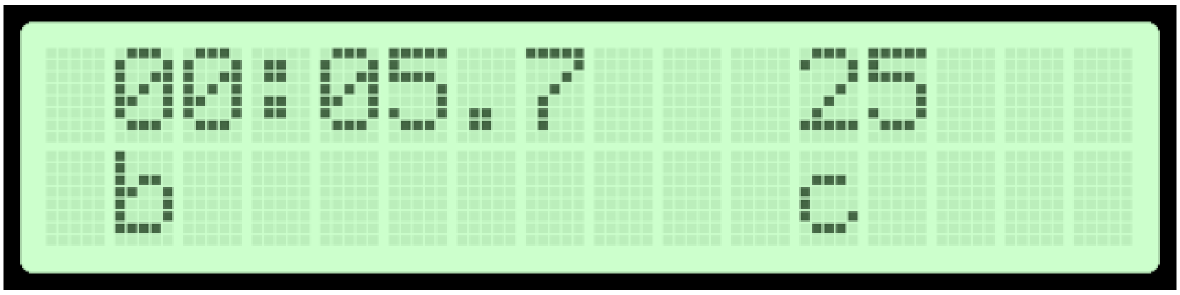
(Optional) Experiments on your own

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



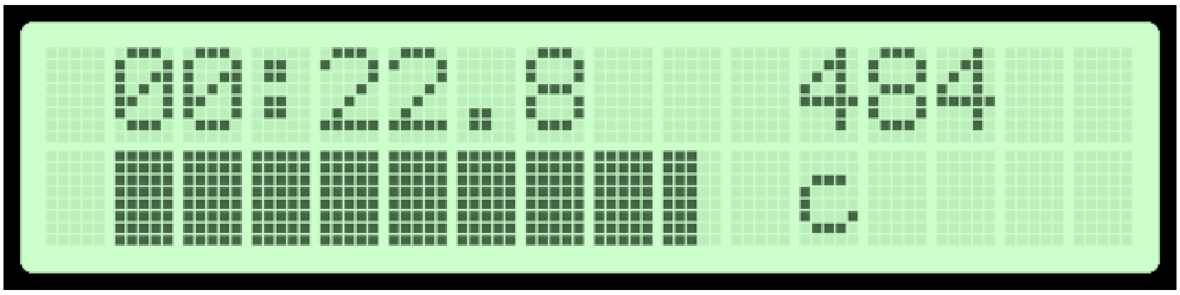
2. In `lcd.h` and `lcd.c` files create a new library function to turn on/off LCD's backlight.

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

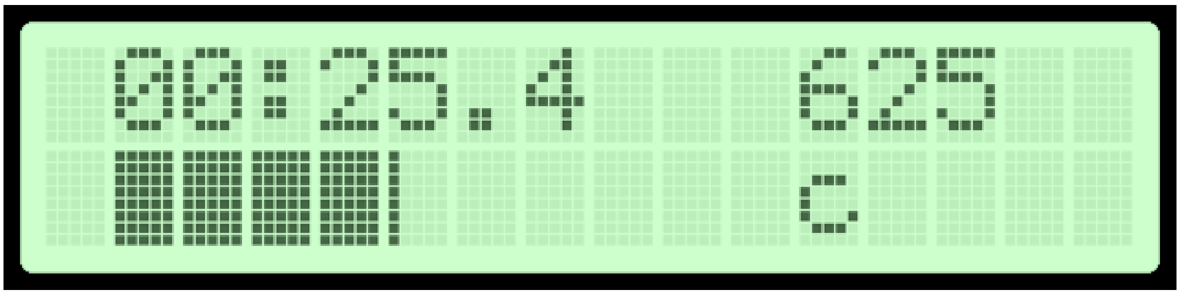


```
 00:05.7   25
 b         c
```

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



```
 00:22.8   484
 ████████▌ c
```



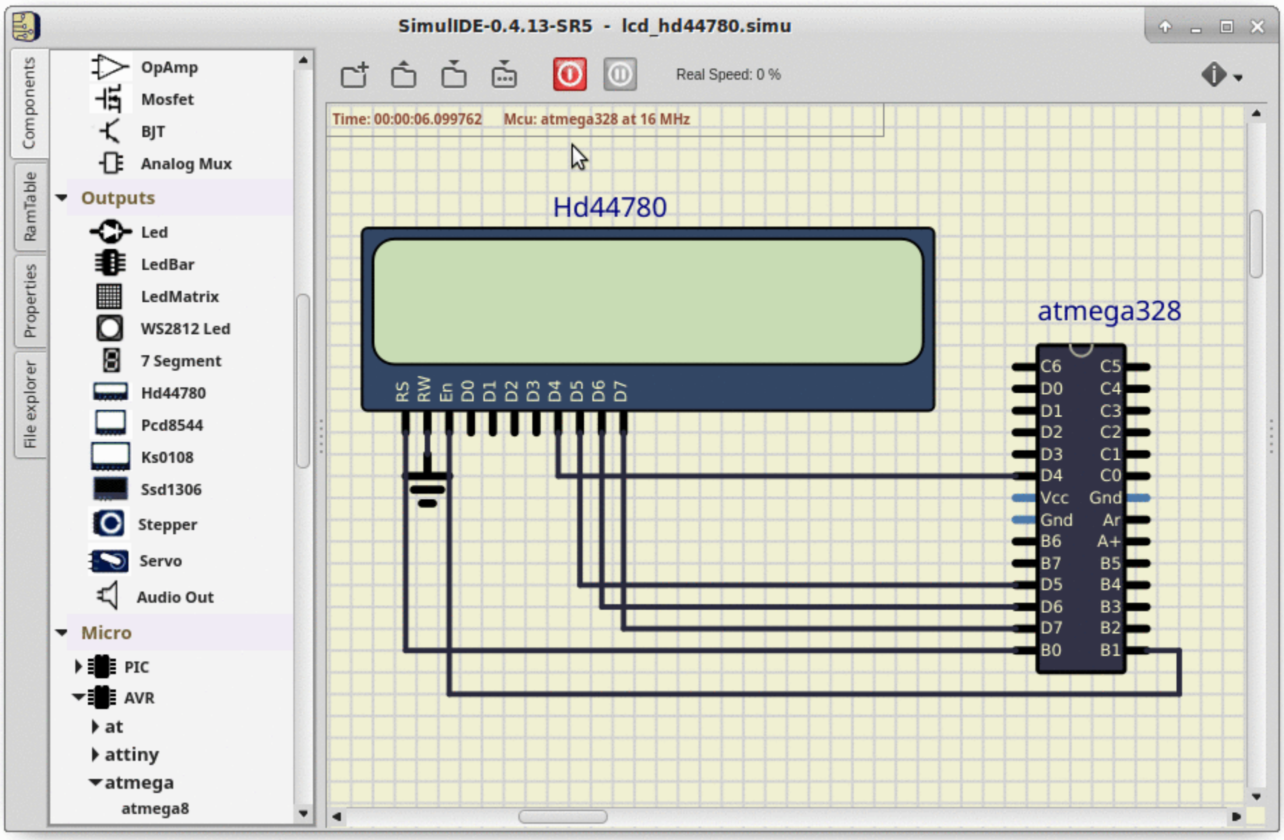
```
 00:25.4   625
 ████▏     c
```

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

```c
/* 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