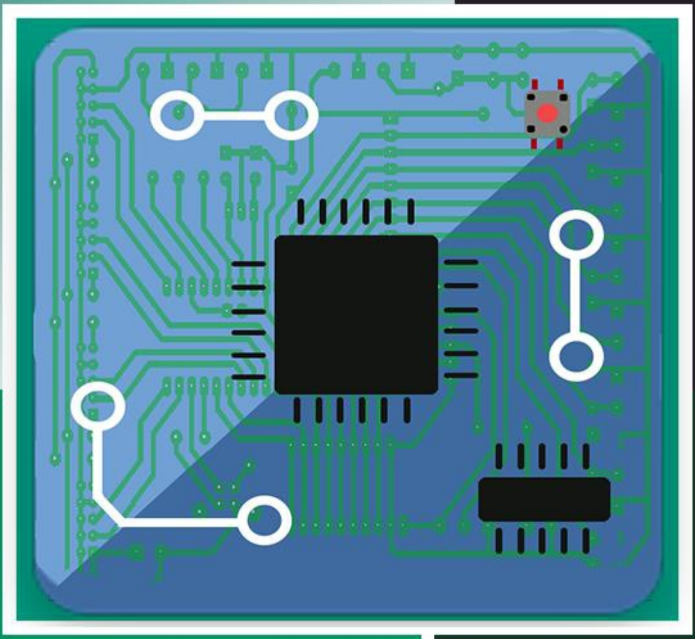
Radu Dănescu
Mircea Paul Mureșan
Razvan Itu
Tiberiu Marița



Design with Microprocessors

Laboratory Guide

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

UTPRESS
Cluj-Napoca, 2018
ISBN 978-606-737-337-0

Editura U.T. PRESS
Str. Observatorului nr. 34
C.P. 42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999
e-mail: utpress@biblio.utcluj.ro
http://biblioteca.utcluj.ro/editura

Director: Ing. Călin D. Câmpean

Recenzia: Conf.dr.ing. Florin Oniga
Prof.dr.ing. Sergiu Nedevschi

ISBN 978-606-737-337-0

TABLE OF CONTENTS

Introduction...2
I. Laboratory 1 – Introduction to the Arduino boards...3
1.1. First example: Using the Arduino development board...4
1.2. 2-nd example: Digital input pins and the serial interface...6
1.3. 3-rd example: The use of the LED Pmod as an output device. The use of ports……10
1.4. Individual work...13
II. Laboratory 2 – Applications with simple I/O modules...14
2.1. The 2x7 segment display...14
2.2. The usage of microcontroller port for Input/Output operations.................................16
2.3. Connecting the Seven Segments display to the Arduino board.................................17
2.4. The Arduino Learning Shield..20
2.5. Individual work...23
III. Laboratory 3 – Working with the LCD shield and the interrupt system..................24
3.1. Working with the LCD shield 24
3.2. Working with the interrupt system...30
3.3. Individual work...35
IV. Laboratory 4 – Usage of timers...36
4.1. Timer based interrupts...36
4.2. Tones generation...40
4.3. PWM signal generation using Arduino...42
4.4. Individual work...43
Appendix A – Lab.4 (contents of „pitches.h” file)...44
V. Laboratory 5 – Communication Interfaces...46
5.1. Individual work...56
VI. Laboratory 6 – Working with the AVR Assembly language.....................................57
6.1. Assembly and C...57
6.2. Using the Arduino IDE...62
6.3. Using Atmel Studio..67
6.4. Individual work...76
VII. Laboratory 7 – Analog signals processing..77
7.1. Individual work...85
VIII. Laboratory 8 – Arduino and WiFi – IoT applications...86
8.1. Individual work...90
IX. Laboratory 9 – Use of DC motors and servo motors. The experimental robot.........91
9.1. DC motors..91
9.2. Servo motors...94
9.3. The experimental robot..95
9.4. Example code...98
9.5. Project activity rules...100
Appendix 1– Using the Processing IDE...102
Appendix 2 – Robotics Operating System (ROS) and Arduino…………..……………111

Introduction

The microprocessor is the central element of a modern computing system. It includes, in a single integrated circuit, all the functions of a central processing unit (CPU). The CPU's main role is to execute programs, which are sequences of machine code instructions stored in a memory. The execution of an instruction usually takes four steps: reading the instruction from memory (*fetch*), decoding it (*decode*), arithmetic/logic execution (*execute*), and writing the results (*write back*).

Although they are highly complex circuits, the microprocessors cannot work without other essential systems, such as memories that store instructions and data, the input and output data interfaces, or peripherals for sending and receiving data, all connected to the microprocessor by means of buses. Together with the microprocessor, these components form microprocessor based systems. Such a microprocessor based system is the PC motherboard.

Unlike microprocessors, which are only tasked with executing instructions, the microcontroller has its own memory for storing programs and data, ports for input and output (GPIO - General Purpose Input/Output), and other components such as timers, communication interfaces, converters, etc. The microcontroller is therefore a single chip microprocessor based system, which can be used to control multiple processes or devices. Due to the small size and low production cost, the microcontrollers are often used in embedded systems, where reducing the resource demand is important.

The Design with Microprocessors subject, from the Computer Science 3rd year curriculum, aims to acquaint the students with the specifics of the microprocessor based systems. We also aim to provide the students, and all the interested learners, the knowledge and skills needed to design and implement their own microprocessor or microcontroller based systems.

This laboratory guide includes step by step tutorials for making the students familiar with the AVR 8 bit microcontroller family, the Arduino Mega and Arduino Uno development boards, the use of input/output ports and the use of serial interfaces, and also with the use of multiple external modules for generating and displaying data, with the use of sensors and actuators. For writing and debugging the programs the students will learn to use the software development tools Arduino IDE, Atmel Studio and Processing.

We wish these laboratory works, which contain detailed explanations and examples, will help the students assimilate the theoretical and practical notions of the subject, so that they will get a good mark in the exam, but we also hope that the practical skills they acquire will inspire them to turn their own original ideas into reality.

THE AUTHORS

I. Laboratory 1 – Introduction to the Arduino boards

The set of Arduino development tools include μC (microcontroller) boards, accessories (peripheral modules, components etc.) and open source software tools which allows users to implement projects using a high-level unified approach, which is μC-model independent. The Arduino boards are mainly equipped with AVR (Atmel) MicroController Units (MCUs), but there are variants equipped with ARM or x86 MCUs. Besides the genuine versions of the Arduino boards there are a lot of third party ones (XDRuino, Freeduino, Funduino etc.) which are low cost and fully compatible (although their reliability could be questionable).

The board used for the lab is Arduino Mega 2560, featuring the 8-bit Atmel ATMega 2560 MCU (8 bit due to the size of the internal registers). The board gives you access to 54 digital pins of the MCU for I/O operations and 16 pins for analog signals reading. Some pins can have multiple functions (i.e. providing additional communication facilities: UART, SPI, I2C etc.) – see the printed material on your desks. The MCU operates at a 16 MHz clock. The board can be power through the USB interface (common usage) or through an external power source at 7 … 12 DC V (minimum current of 0.25A). The second option is necessary to power current hungry peripherals (i.e. motors, GSM shields etc.)

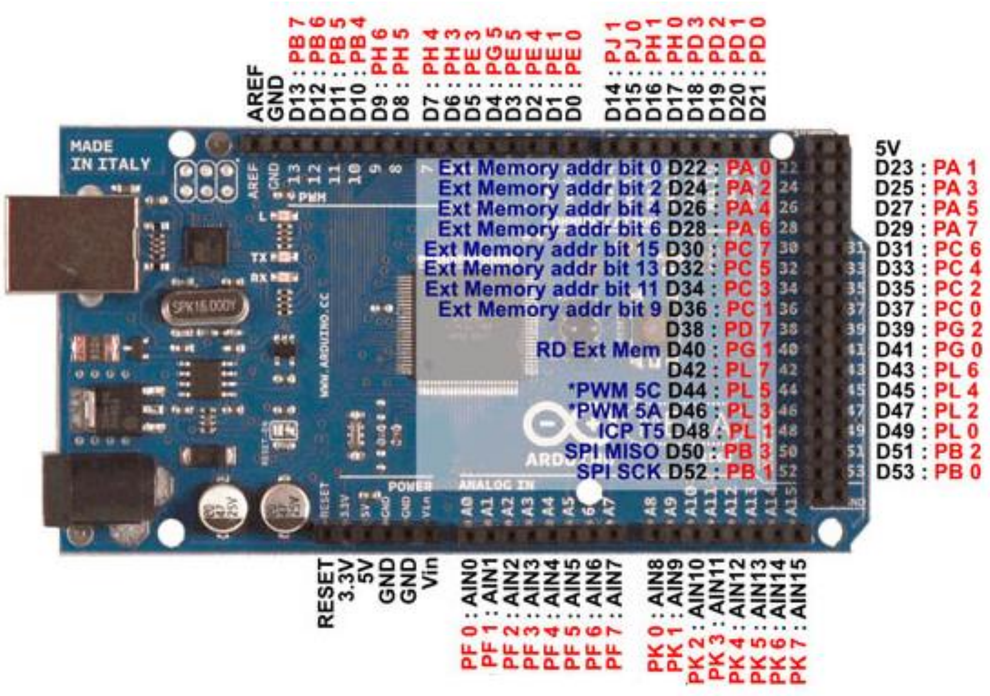


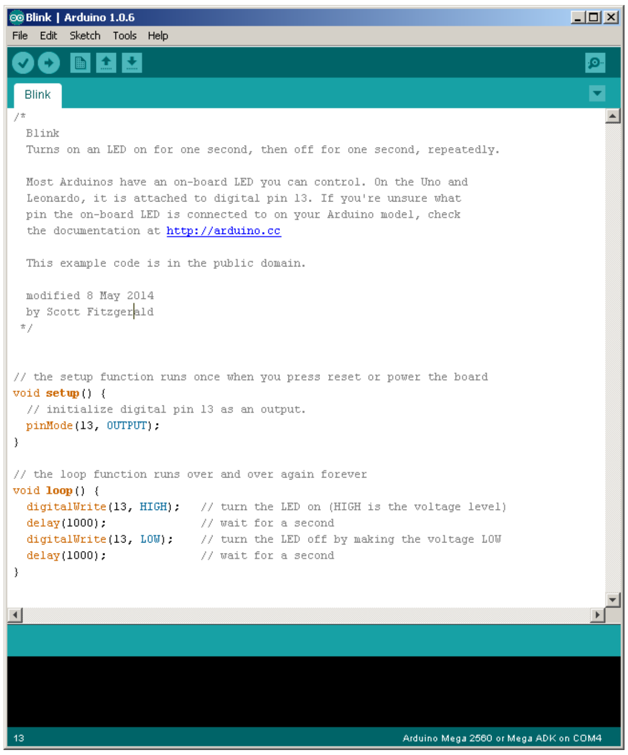
Fig.I.1 Arduino Mega pinout (source: http://electrobist.com/product/arduino-mega-2560-r3/)

1.1. First example: Using the Arduino development board

For a safer usage the board is mounted on an acrylic base along with the breadboard. For the beginning you should load the most basic example "Blink", available in the installation directory of the Arduino IDE (usually *C:\Program Files\Arduino\examples\01.Basics\Blink*). Copy the "*Blink*" folder in your working folder (compulsory **in D:\Studenti\Grupa30xxx, otherwise it will be periodically DELETED**). Check that after the copy operation you have for the "Blink" folder and its contained "*blink.ino*" file write permissions! (not Read Only).

Rule: every Arduino project (even if it has only one source file) should be placed in a folder with the same name as the source file (Ex: *D:\Studenti\Grupa30xxx\Nume_Prenume\***Blink******Blink.ino**).

After these preparations are done, launch the Arduino IDE either by double-clicking on the *.ino file or by launching the IDE from the start menu/shortcut followed by an explicit *Open* operation on the *.ino file. The program window should look like this:



```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedly.

  Most Arduinos have an on-board LED you can control. On the Uno and
  Leonardo, it is attached to digital pin 13. If you're unsure what
  pin the on-board LED is connected to on your Arduino model, check
  the documentation at http://arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */


// the setup function runs once when you press reset or power the board
void setup() {
  // initialize digital pin 13 as an output.
  pinMode(13, OUTPUT);
}

// the loop function runs over and over again forever
void loop() {
  digitalWrite(13, HIGH);   // turn the LED on (HIGH is the voltage level)
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // turn the LED off by making the voltage LOW
  delay(1000);              // wait for a second
}
```

```
13                                   Arduino Mega 2560 or Mega ADK on COM4
```

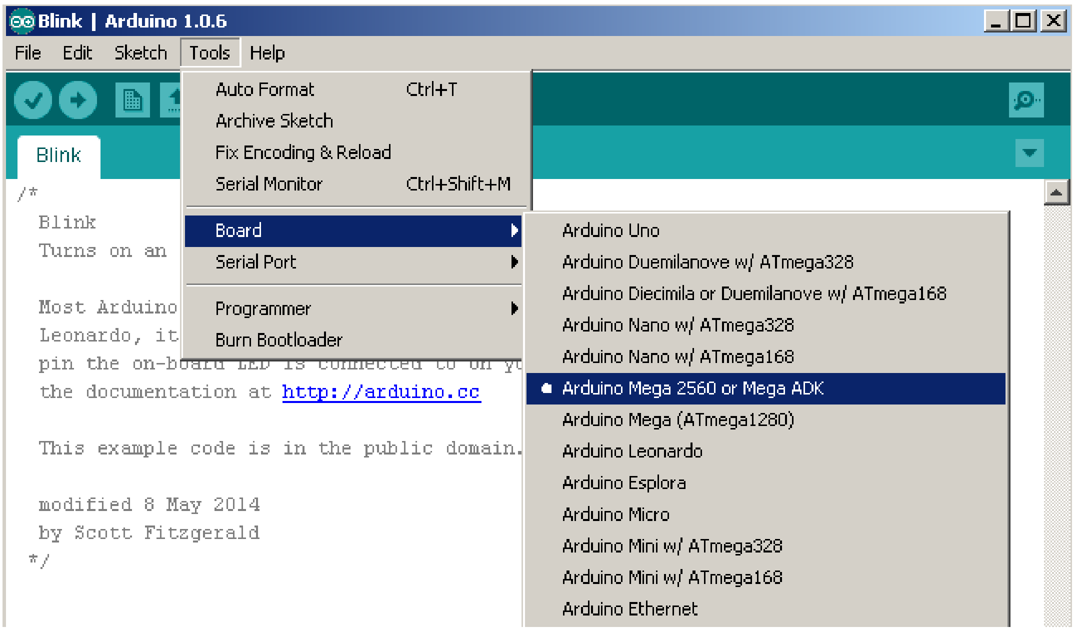
Fig.I.2. The Arduino IDE main window

If the IDE does not start, the cause may be a missing association between the IDE and the .ino file type. You can start the program from the Start Menu / Programs, and then use the File → Open menu command to open the source file.

At this point you can connect the Arduino Mega 2560 board to the PC through the USB cable. The operating system may ask you for the installation of a driver (in that case specify the following path: “*C:\Program Files\Arduino\drivers*”). If you encounter difficulties, or you do not have enough user rights, ask the professor for help.

If the driver is correctly installed and the board is functioning (a green LED on the board is on) you can go ahead.

Before programming the board, you should check if the IDE is configured properly in the menu ***Tools* → *Board*:**


Blink | Arduino 1.0.6
File
Edit
Sketch
Tools
Help
Auto Format Ctrl+T
Archive Sketch
Fix Encoding & Reload
Serial Monitor Ctrl+Shift+M
Board
Serial Port
Programmer
Burn Bootloader
Blink
Arduino Uno
Arduino Duemilanove w/ ATmega328
Arduino Diecimila or Duemilanove w/ ATmega168
Arduino Nano w/ ATmega328
Arduino Nano w/ ATmega168
● Arduino Mega 2560 or Mega ADK
Arduino Mega (ATmega1280)
Arduino Leonardo
Arduino Esplora
Arduino Micro
Arduino Mini w/ ATmega328
Arduino Mini w/ ATmega168
Arduino Ethernet

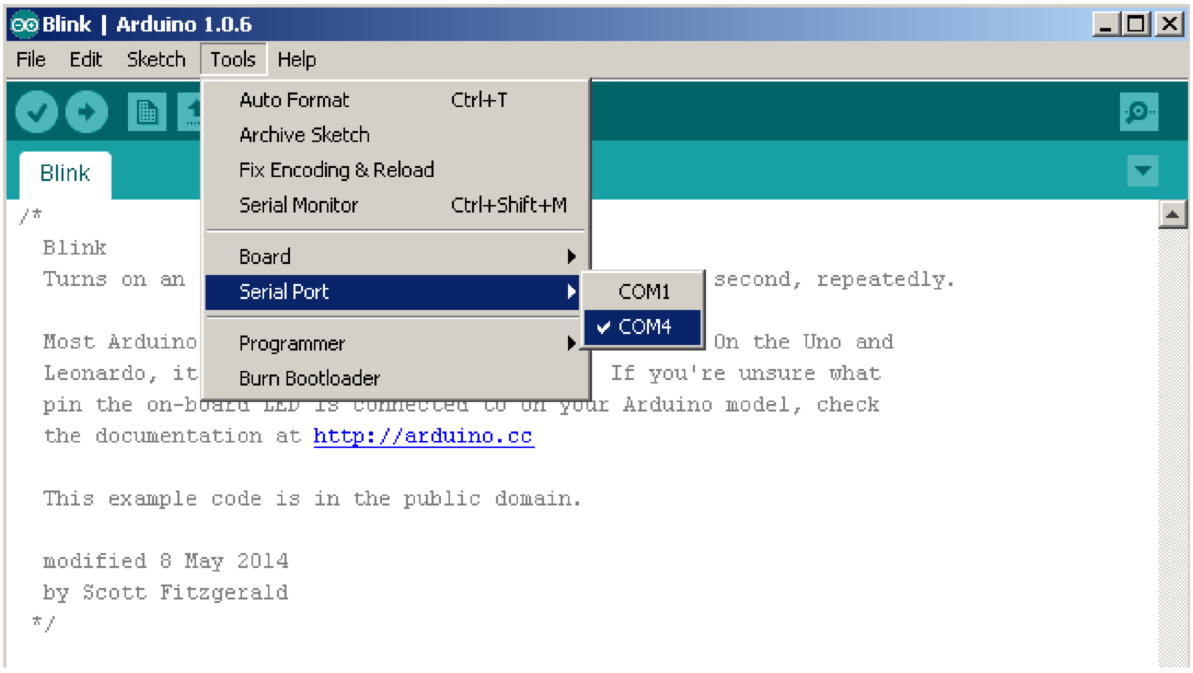
```
/*
  Blink
  Turns on an

  Most Arduino
  Leonardo, it
  pin the on-board LED is connected to on yo
  the documentation at http://arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */
```

Fig.I.3. Selection of the development board type

Alternatively, if you are using other development board (i.e. UNO you should make the settings accordingly).

Also the serial port use to communicate with the board should be properly configured. The USB driver/interface of the board will install a virtual serial port (COMx). COM 1 and 2 are usually the serial ports of the PC. The virtual serial ports are assigned to higher numbers (above 3).

Configure the serial port in the IDE ***Tools* → *Serial*** menu as in the figure below:



```
/*
  Blink
  Turns on an                                         second, repeatedly.

  Most Arduino                                        On the Uno and
  Leonardo, it                                    If you're unsure what
  pin the on-board LED is connected to on your Arduino model, check
  the documentation at http://arduino.cc

  This example code is in the public domain.

  modified 8 May 2014
  by Scott Fitzgerald
 */
```

Fig.I.4. Selection of the serial port

After the configuration is complete, you can compile & upload the first example using the button “Upload” (as shown in the figure below). If all the steps were completed successfully, the binary program will run on the board by blinking the on-board LED (which is connected to the digital pin 13.

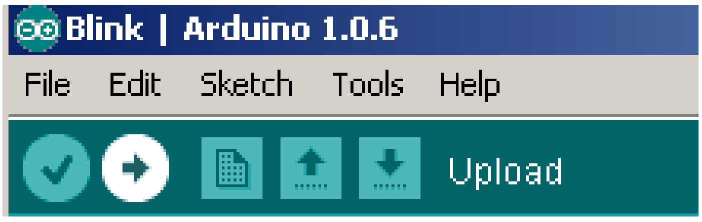


Fig.I.4. The upload button

1.2. 2-nd example: Digital input pins and the serial interface

As basic input device we’ll use the button block shown below:

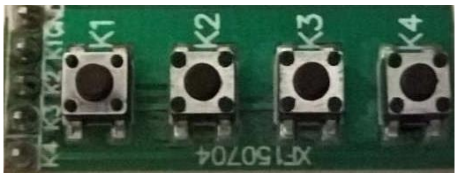


Fig.I.6. The button block

The module has 5 pins:
- GND pin
- 4 data pins (K1, K2, K3 and K4), indicating the button status (logic 0 = button pressed)

The button names appear on the pins and near each button.

The general solutions for connecting a button to a microcontroller are shown below. The first schematic shows the use of a Pull-Down resistor, and the second schematic shows the use of a Pull-Up resistor.

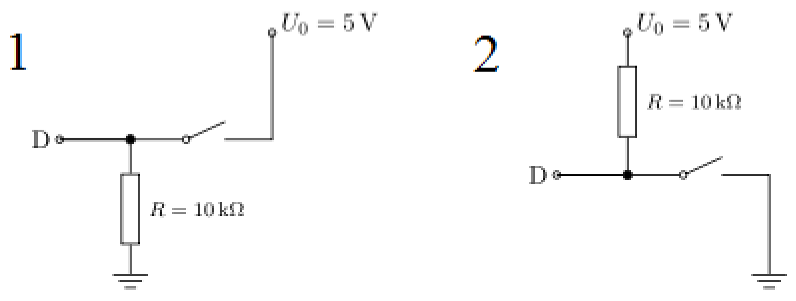


Fig.I.7, Fig.I.8 – The use of pull-down and pull-up resistors (source: https://de.wikibooks.org/wiki/Mikrocontroller/_Digitale_und_analoge_Signale)

We shall focus on the use of Pull-Up resistors, as they are more common. The operation principles for the two types of resistors are similar, the difference being that the Pull-Up resistors are connected to 5V (VCC), while the Pull-Down resistors are connected to the ground (GND). When using a Pull-Up resistor, and the button is not pressed, the value of the input pin D is HIGH, or logic 1. A small current flows between VCC and the input pin. When the button is pressed, the input pin is connected directly to the ground, and the current flows through the resistor to the ground, while the input pin reads LOW, or logic 0. If no resistor is used, pressing the button would connect VCC directly to GND, resulting in a short-circuit (which is not desirable, and may be dangerous).

The button block does not have Pull-Up (or Pull-Down) resistors. This means that, in order to use it, we have to either:
- Attach external resistors to each button pin
- We use the Pull-Up resistors of the microcontroller

In this lab work we'll use the internal resistors of the microcontroller on the Arduino board. These resistors are activated by using the INPUT_PULLUP pin configuration option.

As visualization output, the serial interface will be used (allows to monitor the output of the Arduino board on the PC). **All the connections between the peripheral modules, breadboard and MCu board will be done with the USB cable disconnected from the PC !!!**

To connect the button block to the Arduino board, the breadboard will be used:
- The 4 data pins will be connected to the digital pins 4, 5, 6, 7 of the Arduino board
- GND pin of the button module will be connected to the GND of the Arduino board

In order to avoid the situation of the wires breaking loose at every small move, leading to a very unstable setup, we shall use the breadboard to strengthen the prototype.

The breadboard has electrically connected pins in groups of 5 (a half column) as shown in the figure below:

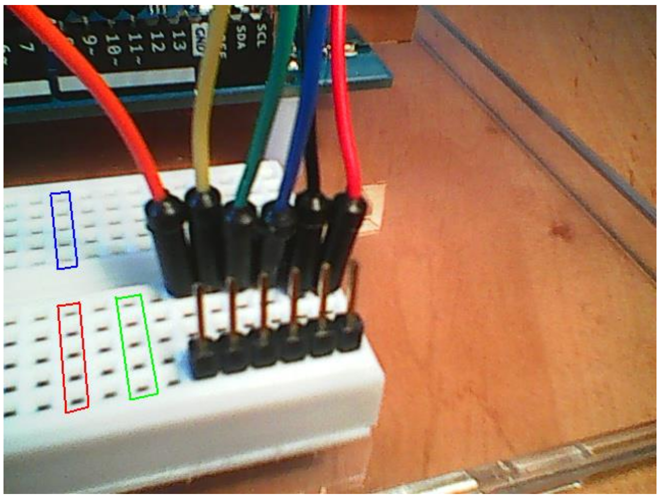


Fig.I.9. The electrical connections on a breadboard

Place the button module above the breadboard, and press firmly so that the pins enter the breadboard's holes. Then, insert a wire into a hole of each half-column, corresponding to a button block pin, as shown in the figure below. Use a black wire for the ground, and different colors for each button pin (if possible).

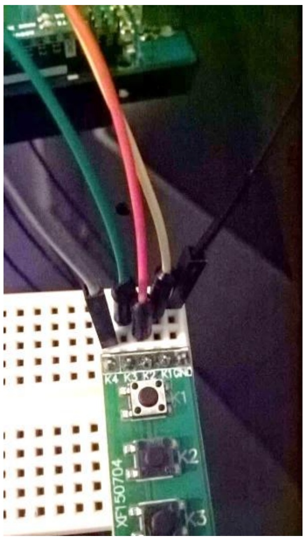


Fig.I.10. Connecting the buttons to the breadboard

The other ends of the wires will be connected to the Arduino board:

- The signal wires (for the button pins) to the digital pins 4, 5, 6, 7:
- The black wire to the ground (GND)

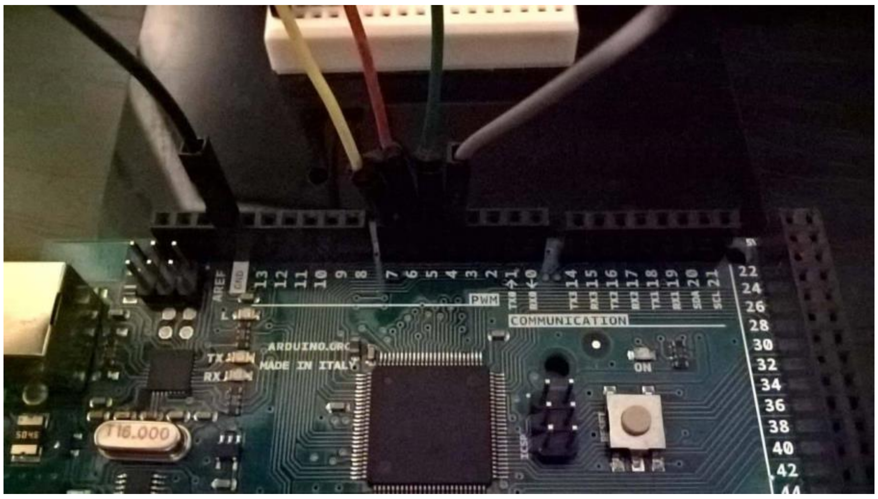
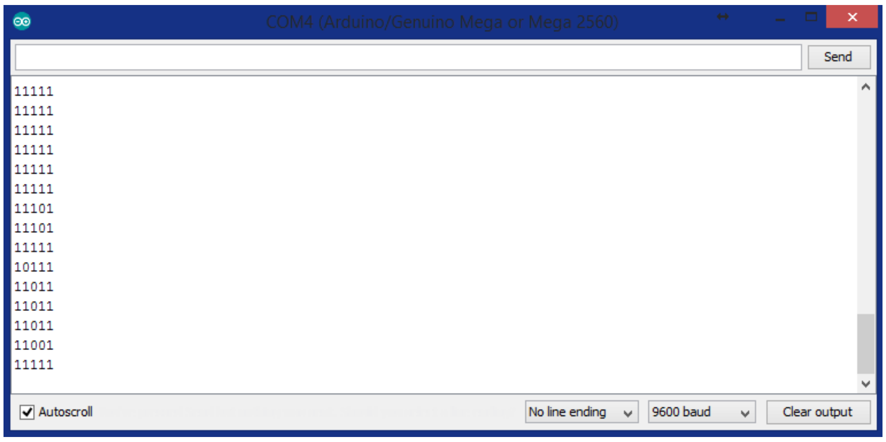


Fig.I.11. Connecting the wires to the Arduino board

Now the physical setup is ready. In the following, open the Arduino IDE and create a new program (*File* → *New*), which will contain the following code:

```
//  Read  status  of  buttons  and  send  it  to  the  PC  over  the
//serial connection
// Variables for reading the status of the buttons connected
//to the digital pins 4, 5, 6, 7
int b1;
int b2;
int b3;
int b4;

// variable for the transmitted status (as a decimal number)
int stat = 0;

void setup() {
    // configure digital pins as inputs
    pinMode(4, INPUT_PULLUP);
    pinMode(5, INPUT_PULLUP);
    pinMode(6, INPUT_PULLUP);
    pinMode(7, INPUT_PULLUP);
    // activate serial communication for displaying the result
   // on the PC
    Serial.begin(9600);
}

void loop() {
    // read BTNs status
    b1 = digitalRead(4);
    b2 = digitalRead(5);
    b3 = digitalRead(6);
    b4 = digitalRead(7);
    // combine the bits in a decimal number (stat)
    stat = 10000 + b4 * 1000 + b3 * 100 + b2 * 10 + b1;
    //transmit status
    Serial.println(stat);
    // delay 0.5 sec
    delay(500);
}
```

Connect the Arduino board to the PC and “Upload” the program. To visualize the result, open the “Serial Monitor” utility from the *Tools* → *Serial Monitor* menu. At every 0.5 sec the value of the stat variable will be displayed as a 5 digit number (1XXXX), the list significant 4 digits being the status of the buttons. When a button is pressed, its corresponding digit turns from 1 to 0.



```
11111
11111
11111
11111
11111
11111
11101
11101
11111
10111
11011
11011
11011
11001
11111
```

Fig.I.12. The output in the Serial Monitor

Attention: The Serial Monitor utility should be closed before disconnecting the Arduino board form the PC. Otherwise, it is possible the hang the virtual serial port in a blocked status and further communication with the board will be anymore possible only by restarting the PC.

1.3. 3-rd example: The use of the LED Pmod as an output device. The use of ports.

For this example, we’ll use a block of LEDs. This module has 7 pins, one being the ground and the other 6 are signals for the 6 LEDs (a logic 1 means the LED is lit). The LEDs have current limiting resistors to prevent damage.

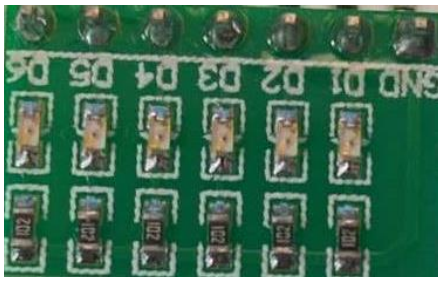


Fig.I.13. The LED block

In order to control the LEDs faster, we will use the ports of the ATMega 2560 microcontroller.

First, we have to prepare seven positions on the breadboard, for the 7 pins of the LED block (D6… D1, GND). We'll also need 7 wires.

We'll insert the LED block into the breadboard, as shown below. Press firmly, but non-violently.

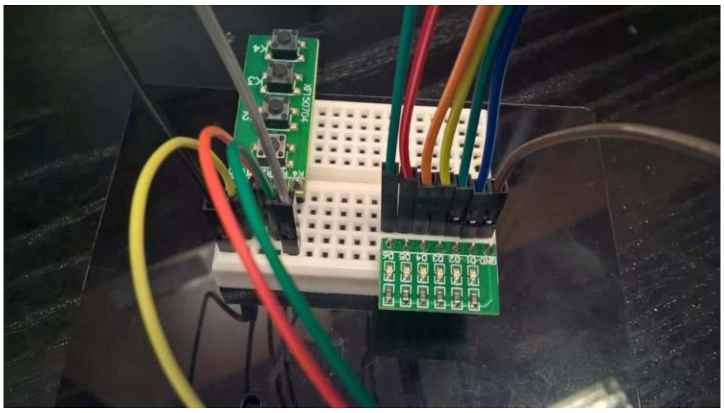


Fig.I.14. Connecting the LED block to the breadboard

We'll connect the signal wires to the digital pins 22, 23, 24 25, 26 and 27 of the Arduino board, which correspond to the bits PA5, PA4, PA3, PA2, PA1, and PA0 of port A, as shown below:

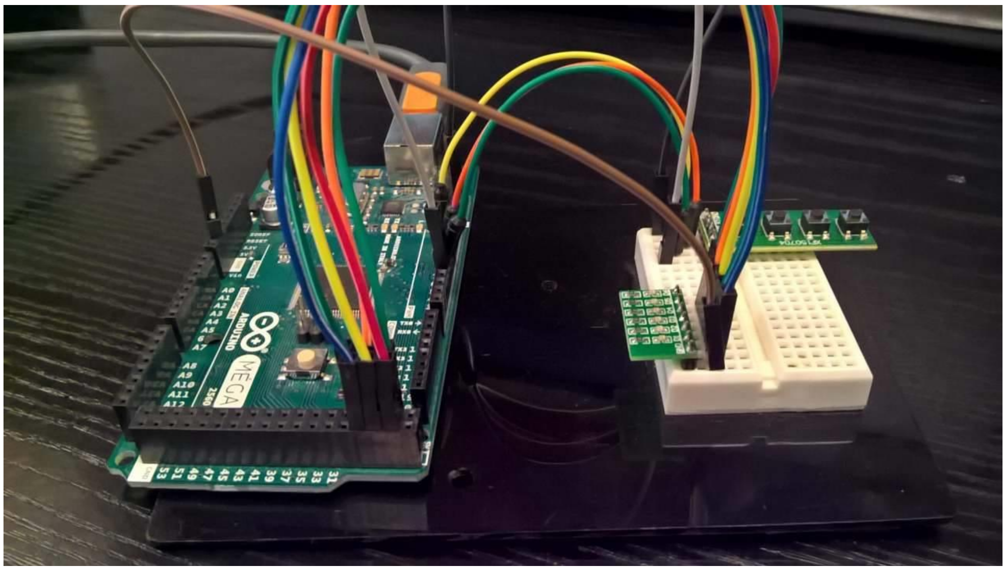


Fig.I.15. Putting it all together

The correspondence between the Arduino board pins and the ATMega 2560 MCU pins (ports) is described in the "ATmega2560-Arduino Pin Mapping" document http://arduino.cc/en/Hacking/PinMapping2560 or in the first figure or in the printed schematic). For port A, we have:

PA7 (AD7)	Digital pin 29
PA6 (AD6)	Digital pin 28
PA5 (AD5)	Digital pin 27
PA4 (AD4)	Digital pin 26
PA3 (AD3)	Digital pin 25
PA2 (AD2)	Digital pin 24
PA1 (AD1)	Digital pin 23
PA0 (AD0)	Digital pin 22

Fig.I.16. The correspondence between digital pins and the AVR ports

We shall also re-use the buttons set up in the previous example.

Note: In the future, when we may use multiple components, we will try to identify common signals, and group them on the breadboard in a single column. In our example, the ground signals for the two modules can be grouped, and a single ground wire used.

After creating the physical setup, create a new Arduino project and add the following code:

```
// Read status of buttons and display it on LEDs connected to
//PORTA

// Variables for reading the status of the buttons connected
//to the
// digital pins 4, 5, 6, 7
int b1;
int b2;
int b3;
int b4;

// variable for the LED status
unsigned char stat = 0;

void setup() {
    // configure digital pins as inputs
    pinMode(4, INPUT_PULLUP);
    pinMode(5, INPUT_PULLUP);
    pinMode(6, INPUT_PULLUP);
    pinMode(7, INPUT_PULLUP);
        // activate PORTA, as output,
        DDRA = 0b11111111;
}

void loop() {
    // read BTNs status
    b1 = digitalRead(4);
    b2 = digitalRead(5);
    b3 = digitalRead(6);
    b4 = digitalRead(7);
    // combine result: each LED is controlled by a button.
    //Some buttons are //duplicated
    stat = (b4<<5) | (b3<<4) | (b4<<3) | (b3<<2) | (b2<<1) |
    b1;
    // Display status on the LEDs connected to port A
    PORTA = stat;
    // delay 50 ms
    delay(50);
}
```

1.4. Individual work:

1. Compute the value of the Pull-Up resistor, if you choose to use an external one, and the maximum current intensity should be limited to 1 mA.
2. Run examples 1 and 2.
3. Modify example 2, in order to transmit to the PC various information according to the pressed button. Ex: one button press will transmit the number of ms from the start of the program (call milis() function), another button press you can transmit the number of sec (millis()/1000), another button wil transmit a text etc.
4. Run example 3.
5. Modify example 3 in order to display some animations on the LEDs (1 walking LED lit-on from right-left or left-right). The walking direction depends on the status of one button.

References:

1. Additional references:
http://users.utcluj.ro/~rdanescu/teaching_pmp.html

2.“ATmega2560-Arduino Pin Mapping” document :
https://www.arduino.cc/en/Hacking/PinMapping2560

II. Laboratory 2 – Applications with simple I/O modules

2.1. The 2x7 segment display

The PmodSSD (Seven Segment Display) from the figure below provides the possibility of displaying two characters. Each segment is a LED, which gives light if there is the right combination of voltages on the segments anode and cathode (anode High, cathode LOW).

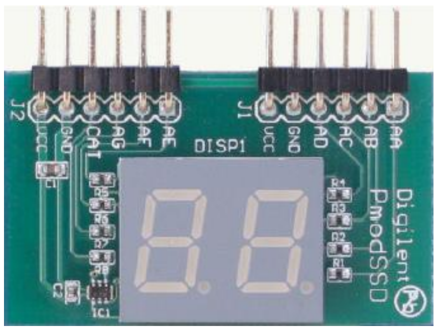


Fig.II.1. Seven segment display (SSD)

In order to reduce the number of pins necessary, the two digits cannot be lit simultaneously. The selection between the two digits is done by the CAT signal (the C on the schema below). If this signal is 0 the segments corresponding to the unities are giving light and if the CAT signal is 1 the segments corresponding to the digit on the left are lit.

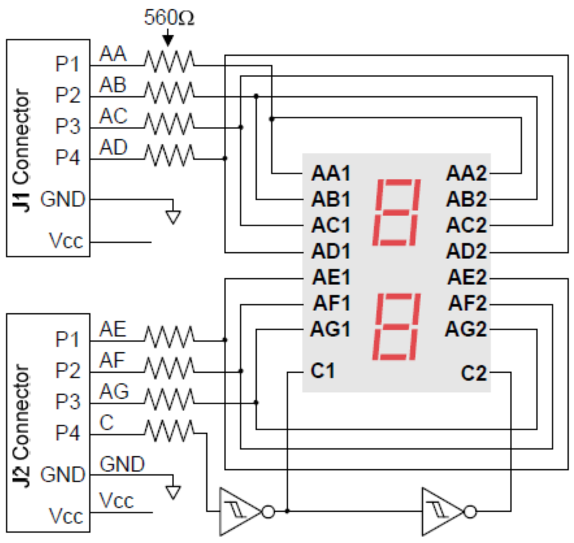


Fig.II.2. The internal structure of the SSD (source: https://store.digilentinc.com/pmod-ssd-seven-segment-display/)

The signs denoted by the letters AA…AC correspond to the anodes of the segment LEDs. The correspondence of these signals is indicated by the figure below:

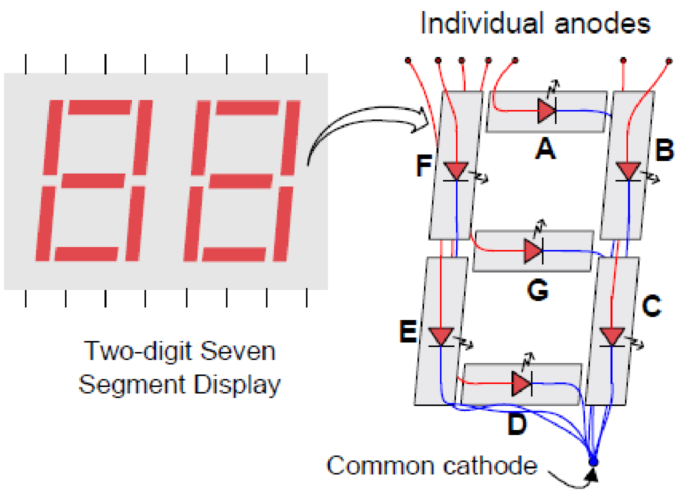


Fig.II.3. The internal structure of a digit (source: https://store.digilentinc.com/pmod-ssd-seven-segment-display/)

As an example, for displaying the digit 3 we will need the segments to receive the following logical levels:

G	F	E	D	C	B	A
1	0	0	1	1	1	1

Fig.II.4. Example of bit activation for displaying the digit ‘3’

In order to show a number of two digits we need to switch (multiplex) very fast between the two blocks using the CAT signal. The following time diagram illustrates the process:

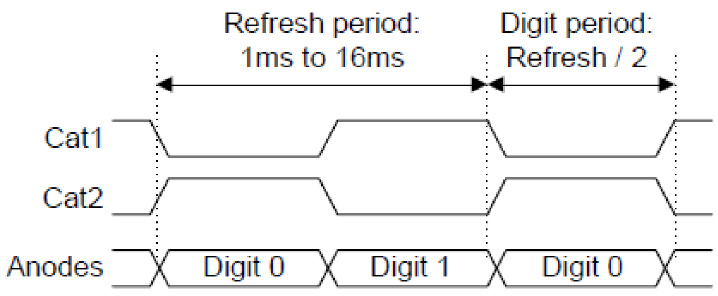


Fig.II.5. Timing diagram for the SSD refresh cycle (source: https://store.digilentinc.com/pmod-ssd-seven-segment-display/)

To realize the functionality exemplified in the above diagram we use the following pseudo code:

```
Loop:
   AA…AG = unit_digit_code
   CAT=0
   Delay()
   AA…AG=tens_digit_code
   CAT=1
   Delay()
Goto loop
```

2.2. The usage of microcontroller port for Input / Output operations

Arduino, through its digitalRead / digitalWrite functions, hides the mechanism through which the microcontroller does these operations. Furthermore these functions induce delays which may be significant when we need to transfer data on several bits.

Any microcontroller is connected to the outside through ports of input / output. The AVR Atmega 2560 is an 8-bit microcontroller on so it has its 8-bit ports. Each port is associated with three registers (x will be replaced with A, B, C, D, ..., depending on the port used):

– DDRx – direction reister
– PINx – register for input data
– PORTx – output register

The direction register DDRx

DDRx (Data Direction Register) configures the data direction of the port pins (if a bit of a port will be used for input or output). A written on a bit of 0 makes the pin DDRx port corresponding to the input pin, and a bit set to 1 makes the corresponding pin to be output pin.

Examples:

- In order to configure all the bits port A as input pins
  DDRA = 0b00000000;
- To configure all the bits of port A as output
  DDRA = 0b11111111;
- To configure the lower half of port B as output and the upper half as input
  DDRB = 0b00001111;

The input register PINx

PINx (Port IN) is used to read data from the set the set of pins configured as input. To read data, these pins must be set as input pins, setting all of DDRx bits to zero.
Example: reading data from port A:

DDRA = 0;
char a = PINA;

The output register PORTx

Register PORTx is used to transmit data to the microcontroller peripherals connected to the port x pins. For data output to be visible, corresponding bits of DDRx register must be set the direction of value 1.
Example: switching on half of the 8 LEDs connected to Port A
DDRA = 0xFF
PORTA = 0b10101010

2.3. Connecting the Seven Segments display to the Arduino board

The assembly is done with the board unplugged from the PC. The first step is the introduction of the display into the breadboard, gently without hitting or damaging the system. Then we will connect the power wires in the half-columns corresponding to GND and VCC for the second connector (J2) of the display. The other end of the powering wires will be connected to the 5V and GND pins of the Arduino board. We are not required to apply extra voltage to the J1 connector.

For the wires corresponding to the anodes of the display AA…AG and for the cathode CAT, we will connect successively wires that we will introduce into Arduinos digital pins starting from position 22, 23,…29, corresponding to bits 0…7 of port A.

The integral assembly:

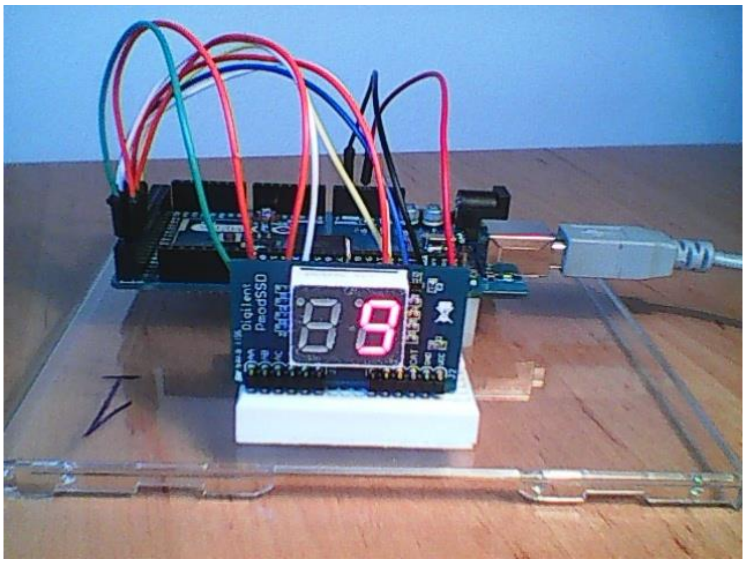


Fig.II.6. The Arduino-SSD assembly

Details of how the wires are connected:

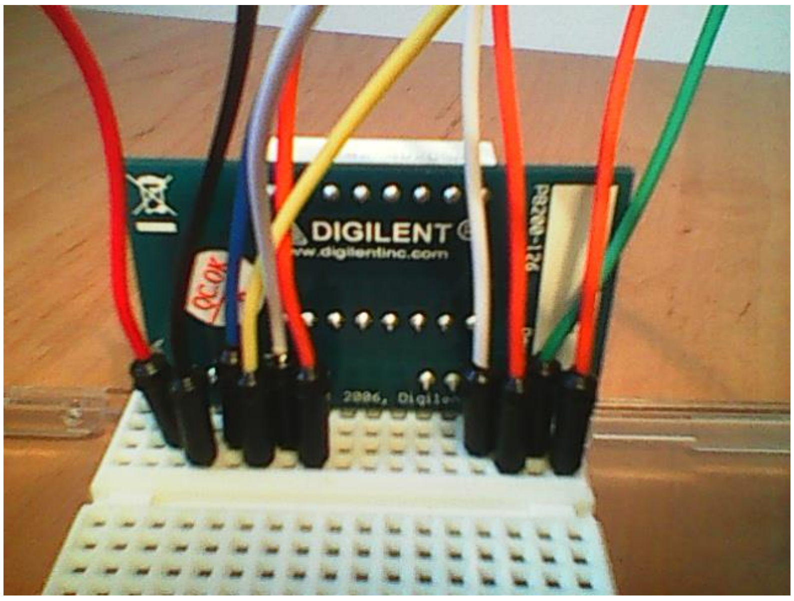


Fig.II.7. Connecting the wires with the SSD on the breadboard

The connection of the signal wires:

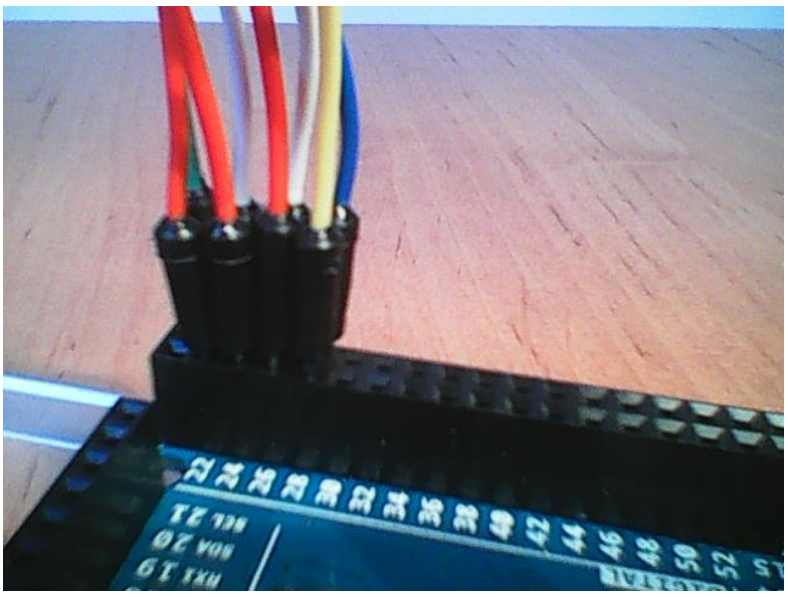


Fig.II.8. Connecting the SSD wires to Arduino

Please Note : The first two pins of the connector from the Arduino board are power pins. The digital pins start from the second pair. Pay much attention of their position.

Create a new project (sketch) in Arduino, and copy the following code:

```
// Display on SSD
// connected at PORTA

// Table of values, or look up table (LUT) with the BCD codes
//for every digit from 0 to 9. Every bit corresponds to a LED,
//1 means the LED is lit and 0 means it is not giving light.

const  unsigned  char  ssdlut[]  =  {0b00111111,  0b00000110,
0b01011011,  0b01001111,  0b01100110,  0b01101101,  0b01111101,
0b00000111, 0b01111111, 0b01101111};
// the size of the lut
const int lutsize = 10;

int cpos = 0; // current position
int cdigit = 0; // first digit from the two
unsigned char outvalue = 0;

void setup() {
    // setting port A as output
    DDRA = 0b11111111;
}

void loop() {

    outvalue = cdigit>0 ? 0x80 : 0;
    // which cathode are we choosing ? (00000000 sau 10000000)
    // the cathode is wired to bit7 from port A, through this
    //operation we are setting bit 7 on
    //logical 1 or
    //0, alternatively, the following bits will be attached
    //through a logical OR operation in the
    //following
    //line of code
    PORTA = (ssdlut[cpos] | outvalue); // we make an OR
    //between the value from the LUT
   // and the selected cathode

    cpos++; // we increment the current position

    if (cpos>=lutsize) { // if we reached the final position
        cpos = 0; // we come back at 0
        cdigit^=1; // if the previus digit was 0 we make it a
        //1 and vice versa
    }

    // wait 0.5 sec
    delay(500);
}
```

This small program will display the digits from 0 to 9 on the first element and then do the same on the second.

2.4. The Arduino Learning Shield

The „shields" are PCBs (Printed Circuit Boards) that can be placed over the Arduino boards, extending their capabilities. There is a large variety of shields, such as XBee shield, SD-Shield, H-Bridge shield, etc. The mapping between the Arduino pins and the devices present on the shields should be carefully taken into account in when programming for proper mapping and conflicts avoidance (the pins provided/used by the shield are specified in the product datasheet).

In this lab work we will use a learning shield, as seen in the figure below. The shield provides the following resources:

- 1 seven segment display, with 4 digits
- 3 buttons (+ one reset button)
- 4 Leds
- 1 potentiometer for generating an analog signal
- 1 sound generator (buzzer)

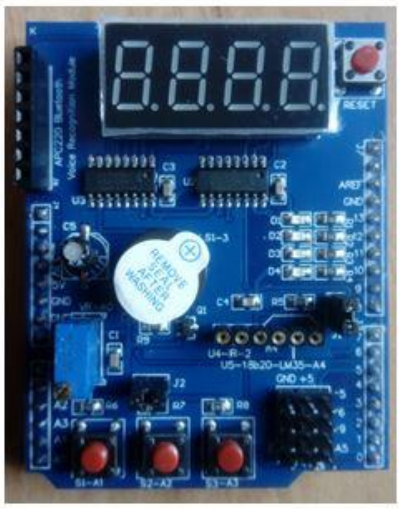
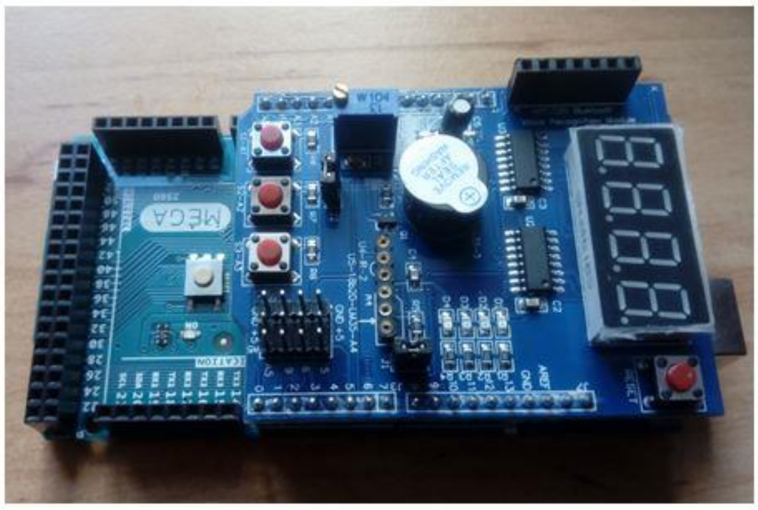



Fig.II.9, Fig.II.10 – The learning Shield and Arduino

The electric schematic for this shield can be downloaded from:
https://ardushop.ro/en/index.php?controller=attachment&id_attachment=40

Accessing the shield's resources:

The buttons are connected to pins A1, A2 and A3. These buttons have Pull-Up resistors on the shield, and therefore their released state will generate a logic 1, and the pressed state will generate a logic 0. There is no need for internal Pull-Up resistor activation.

The Leds are connected with the anode to VCC, and with the **cathodes to pins 10, 11, 12 and 13**. Thus, for lighting an LED a logic 0 must be written to the corresponding pin.

The 4x7 segments display is organized in a common anode configuration, with individual cathodes for each digit segment, unlike the PMod SSD, where the segments have a common cathode and separate anodes. Therefore, for lighting a digit we need to set the digit's anode to logic 1, and the cathodes for the segments to be lit to zero.

For driving the four digits we'll thus need 4 signals for the anodes, and 8 signals for the cathodes (7 segments and the dot). These signals are not available for the programmer, but are connected to two shift registers, as shown in the schematic below. The shift registers are connected in series, the output of the anode register being connected with the input for the cathode register. As a consequence, for writing a full configuration (selection of active digit + selection of segments to light), **only three signals are needed:**

- SDI – serial data in, connected to **pin 8**
- SFTCLK – shift clock, the clock signal for the data shifting, connected to **pin 7**
- LCHCLK – latch clock pin, the signal that allows reading the data, connected to **pin 4**.

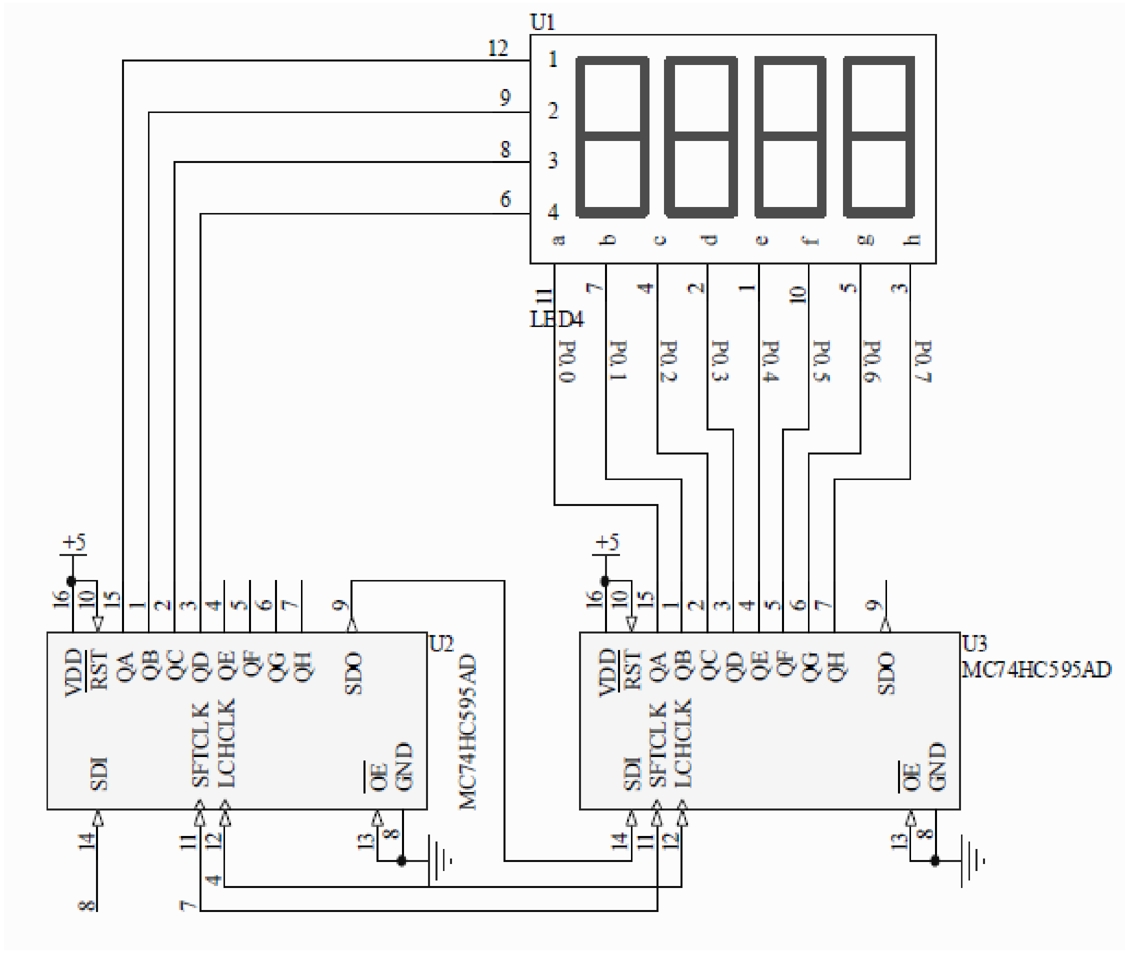


Fig.II.11. The internal structure of the learning shield SSD system (source: https://ardushop.ro/en/index.php?controller=attachment&id_attachment=40)

For writing a digit, first the 8 bit cathode configuration must be transmitted, followed by 8 bits for the anode configuration (with only one bit set to 1, for the active digit).

The following code is an example for using the 4x7 segment display:

```
int latchPin = 4;
int clockPin =7;
int dataPin = 8; // SSD pins
```

```
const  unsigned  char  ssdlut[]  =  {0b00111111,  0b00000110,
0b01011011,  0b01001111,  0b01100110,  0b01101101,  0b01111101,
0b00000111, 0b01111111, 0b01101111};
const unsigned char anodelut[]    = {0b00000001, 0b00000010,
0b00000100, 0b00001000};

const unsigned char digits[] = {1,2,3,4}; // The number to be
//displayed is 1234. You can change it.

void setup ()
{

    pinMode(latchPin,OUTPUT);
    pinMode(clockPin,OUTPUT);
    pinMode(dataPin,OUTPUT); // The three pins connected to
   // the shift register must be output pins

}

void loop()
{

    for(char i=0; i<=3; i++) // For each of the 4 digits
    {

        unsigned char digit = digits[i];  // the current digit
        unsigned char cathodes = ~ssdlut[digit]; // The
        //cathodes of the current digit, we'll
        //negate the value from the original LUT

        digitalWrite(latchPin,LOW); // Activate the latch to
        //allow writing
        shiftOut(dataPin,clockPin,MSBFIRST, cathodes); //
        //shift out the cathode byte
        shiftOut(dataPin,clockPin,MSBFIRST, anodelut [i] );
        // shift out the anode byte
        digitalWrite(latchPin,HIGH); // De-activate the latch
        //signal
        delay(2);   // Short wait
    }

}
```

Analyzing the code:

We will use the same LUT as in the previous example, but we have to complement the bits, as now the bits activate the cathodes, on logic 0.

Arduino provides the function shiftOut, which transmits serially a byte on the dataPin, generating a clock signal on clockPin. This function can be used with any two digital pins.

There are two ways of shifting out a byte: starting from bit 7 (MSBFIRST), or starting from bit 0 (LSBFIRST). In our case, the MSBFIRST mode must be chosen, as it is important that the bits of the cathodes correspond to the segments of the digits.

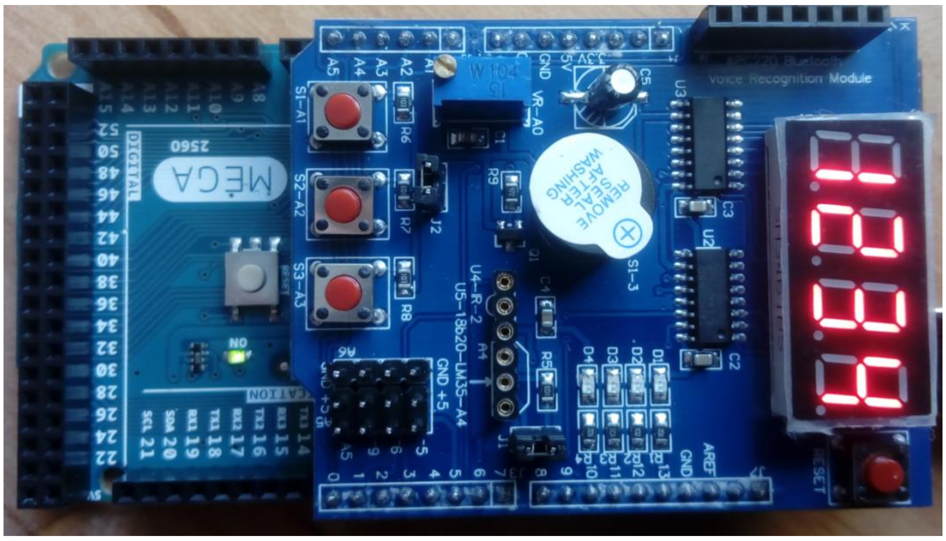


Fig.II.12. Results of running the program

2.5. Individual Work

1. Build and run the first example from the document.
2. Fill in the LUT with the corresponding values for hexadecimal numbers (A,B,C,D,E). Test the program with these kind of numbers.
3. Change the program so that you can show any number from 0 to 99 on the display, in a decimal format. Take into account that it is not the same as its representation as digits on half of byte (this thing is true in the case of hexadecimal numbers). For example, the number 16 is represented binary as 00010000, which will lead to the display of number 10. For correctly displaying the decimal numbers the following steps will need to be realized:

   -find the tens digit = the quotient from the division with 10

   -find the unit digit = the remainder from the division with 10

   CZ = nr div 10

   CU = nr mod 10 = nr – (CZ*10)

4. Run the second example, using the Learning Shield.
5. Change the second example, to display any 4-digit decimal number. The number is provided as **int**, not as individual digits. Make the number increment periodically.
6. Use the buttons on the Learning Shield. Use one button to increment the displayed number, and another to decrement it.
7. Use the 4 digit display, and the buttons (on the Learning Shield) to make a chronometer that counts seconds (2 digits) and hundredths of seconds (2 digits). Use one button for start, another for stop, and another for clear. **Hint: use the function millis()**

III. Laboratory 3 – Working with the LCD shield and the interrupt system

3.1. Working with the LCD shield

The „shields" are PCBs (Printed Circuit Boards) that can be placed over the Arduino boards, extending their capabilities. The mapping between the Arduino pins and the devices present on the shields should be carefully taken into account in when programming for proper mapping and conflicts avoidance (the pins provided/used by the shield are specified in the product datasheet). In the figure below such shield examples are shown: an LCD shield, an SD Card shield and a voice recognition EasyVR) shield.

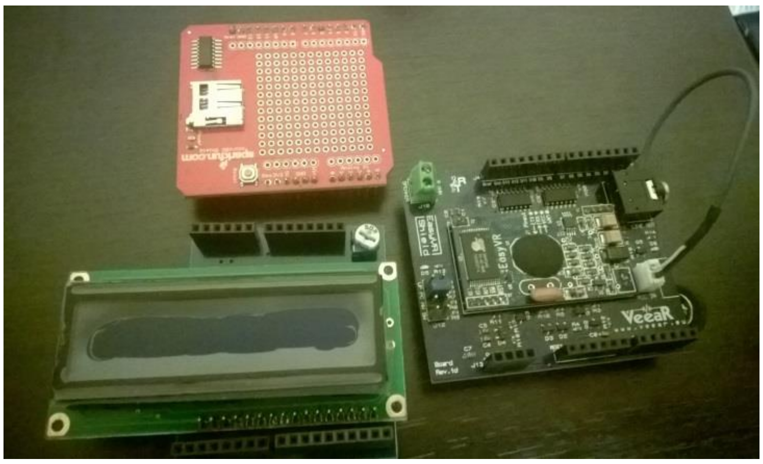


Fig.III.1. Various shields

Since the Arduino boards are very popular, many competitors are using the same design layout (see the figure below: left - a Fez Domino board with ARM processor; right- a CipKit board with PIC processor) being also able to use the Arduino compatible shields.

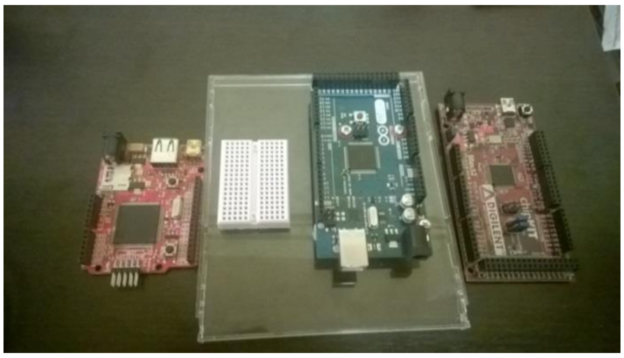


Fig.III.2. Multiple development boards with Arduino-like shape

In this laboratory an LCD (Liquid Crystal Display) shield will be used. It contains the display and a potentiometer for tuning its brightness.

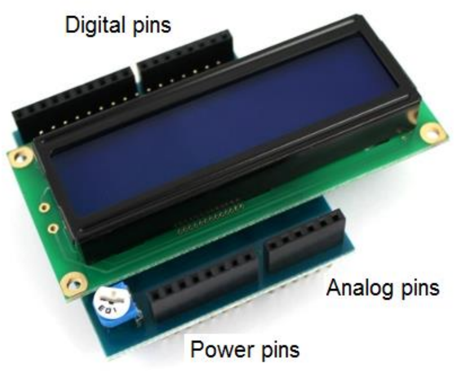


Fig.III.3, Fig.III.4 – The LCD shield

The shield is placed over the Arduino board in such a way that the longer pin bars (10 + 8) correspond to the digital pins (see the figure below).

DO NOT remove the shield from the Arduino board (because you will damage its pins) !!!

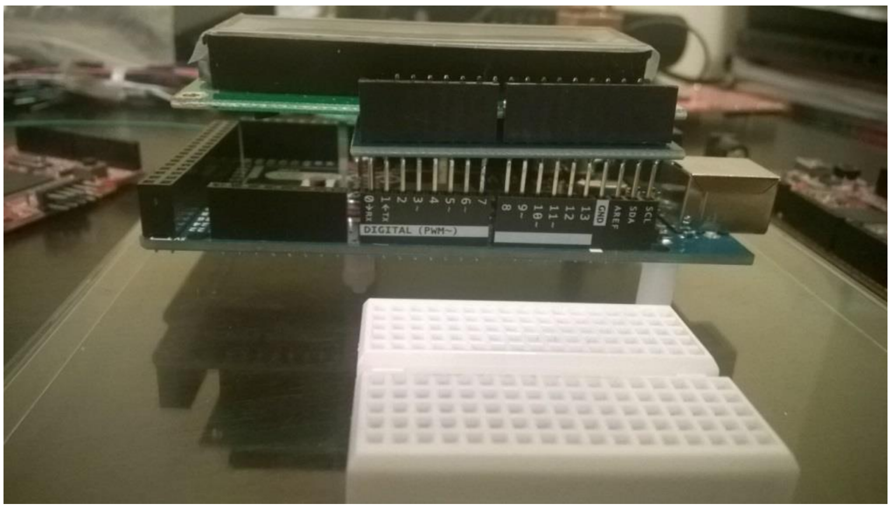


Fig.III.5. Connecting the LCD shield to Arduino

The LCD uses the digital pins 2 … 7: 7 – RS, 6 – En, 5 – DB4, 4 – Db5, 3 – DB6, 2 – DB7 (their mining is in the table below).

The shield uses the classic LCD controller Hitachi HD44780. One such controller can be used for LCDs with max. 80 characters and 4 lines. The pinout of the shield is shown below:

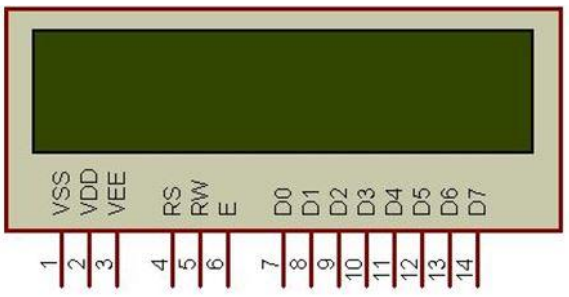


Fig.III.6. The LCD pins (source: https://www.pantechsolutions.net/interface-cards-tutorials/user-manual-for-lcd-interface-card)

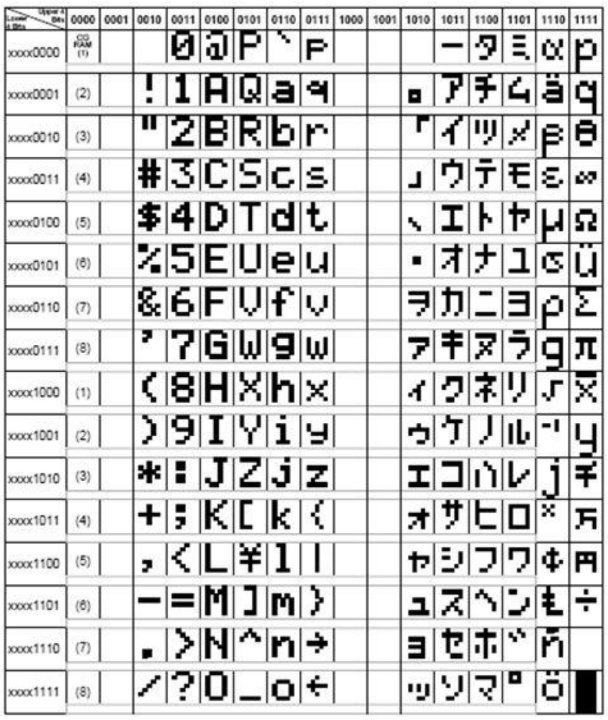
Pin No.	**Name**	**Description**
1	**VSS**	Power supply (GND)
2	**VCC**	Power supply (+5V)
3	**VEE**	Contrast adjust
4	**RS**	Register Select: 0 = Instruction input; 1 = Data input
5	**R/W**	0 = Write to LCD module; 1 = Read from LCD module
6	**EN**	Enable signal
7	**D0**	Data bus line 0 (LSB)
8	**D1**	Data bus line 1
9	**D2**	Data bus line 2
10	**D3**	Data bus line 3
11	**D4**	Data bus line 4
12	**D5**	Data bus line 5
13	**D6**	Data bus line 6
14	**D7**	Data bus line 7(MSB)

Fig.III.7. LCD pins description

The HD44780 contains 2x 8 bits registers: a Data register and an Instruction register. The instruction register is used to send command to the LCD (i.e. shift, clear etc.). The data register is used to buffer the displayed data. When the EN signal is activated, the Data present on the data pins are transferred firstly in the Data register and then moved in the DDRAM (Display Data RAM) and displayed on the LCD. The Data register is used also to transfer data to the CGRAM (Character Generator RAM) – the memory used to store the user defined characters.

The DDRAM (Display Data RAM) stores the data to be displayed, represented as 8 bit characters. Its total capacity is 80x8 bits (80 characters). The remaining free DDRAM memory can be used as a generic RAM (our LCD uses only 2x16=32 characters.

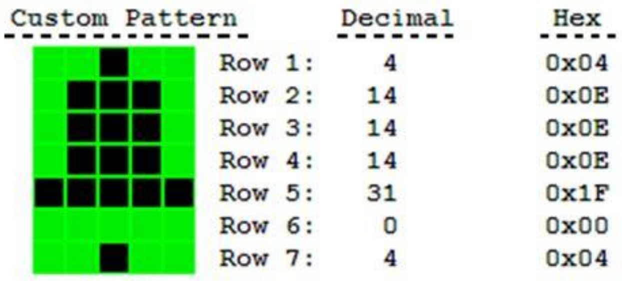
The graphical shape of the characters is stored in the CGROM (Character Generator Read Only Memory). This memory contains the matrices of points for every character (5x8 or 5x10 – depending on the device).



Lower 4 Bits \ Upper 4 Bits	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
xxxx0000	CG RAM (1)			0	@	P	`	p				ー	タ	ミ	α	p
xxxx0001	(2)		!	1	A	Q	a	q			。	ア	チ	ム	ä	q
xxxx0010	(3)		"	2	B	R	b	r			「	イ	ツ	メ	β	θ
xxxx0011	(4)		#	3	C	S	c	s			」	ウ	テ	モ	ε	∞
xxxx0100	(5)		$	4	D	T	d	t			、	エ	ト	ヤ	μ	Ω
xxxx0101	(6)		%	5	E	U	e	u			・	オ	ナ	ユ	σ	ü
xxxx0110	(7)		&	6	F	V	f	v			ヲ	カ	ニ	ヨ	ρ	Σ
xxxx0111	(8)		'	7	G	W	g	w			ァ	キ	ヌ	ラ	g	π
xxxx1000	(1)		(	8	H	X	h	x			ィ	ク	ネ	リ	√	x̄
xxxx1001	(2)		)	9	I	Y	i	y			ゥ	ケ	ノ	ル	⁻¹	y
xxxx1010	(3)		*	:	J	Z	j	z			ェ	コ	ハ	レ	j	千
xxxx1011	(4)		+	;	K	[	k	{			ォ	サ	ヒ	ロ	ˣ	万
xxxx1100	(5)		,	<	L	¥	l	|			ャ	シ	フ	ワ	¢	円
xxxx1101	(6)		-	=	M	]	m	}			ュ	ス	ヘ	ン	£	÷
xxxx1110	(7)		.	>	N	^	n	→			ョ	セ	ホ	゛	ñ	
xxxx1111	(8)		/	?	O	_	o	←			ッ	ソ	マ	゜	ö	█

Fig.III.8. The character codes for 5 x 8 points chars (source: https://www.sparkfun.com/datasheets/LCD/HD44780.pdf)

The shape of the characters with codes between 0x00 and 0x07 can be defined by the user. For each character an 8 byte matrix will be specified (one byte for each row). The least significant 5 bits from each row will specify which pixels will be turned on or off (see the example below).



Custom Pattern		Decimal	Hex
	Row 1:	4	0x04
	Row 2:	14	0x0E
	Row 3:	14	0x0E
	Row 4:	14	0x0E
	Row 5:	31	0x1F
	Row 6:	0	0x00
	Row 7:	4	0x04

Fig.III.9. Example of a user defined character (source: https://omerk.github.io/lcdchargen/)

The LCD displays can communicate with the microcontroller on 8 or 4 bits. Usually the 8 bit connection is preferred (to spare the number of pins). The LCD shield used in the lab is configured (by default) with a 4 bit data interface (and uses only 7 pins from the Arduino board).

Example: display character strings and numerical values on the LCD

In this first example a string of characters will be displayed on the first line of the LCD while on the second line a number representing the elapsed time (seconds) from the program start will be displayed.

```
//includes the LCD library
#include <LiquidCrystal.h>
/* LCD RS - pin 7
 * LCD Enable - pinl 6
 * LCD D4 - pin 5
 * LCD D5 - pin 4
 * LCD D6 - pin 3
* LCD D7 - pin 2
The 7-th pin of the LCD is connected to the display brightness
control potentiometer!  */
// Init the LCD with the stated pin numbers
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
unsigned long time;

void setup()
{
    // Sets the no. of rows and columns of the LCD
    lcd.begin(16, 2);
}

void loop()
{
    // Read the number of elapsed seconds from the program
    //start
    time = millis() / 1000;
    // Set the cursor on col 0,  row 0 (first row)
    lcd.setCursor(0, 0);
    // Write a string of characters
    lcd.print("Hello Children");
    // Move the cursor in the middle of the second row (row 1)
    lcd.setCursor(7, 1);
    // Display the elapsed time
    lcd.print(time);
}
```

In the figure below the result is shown:

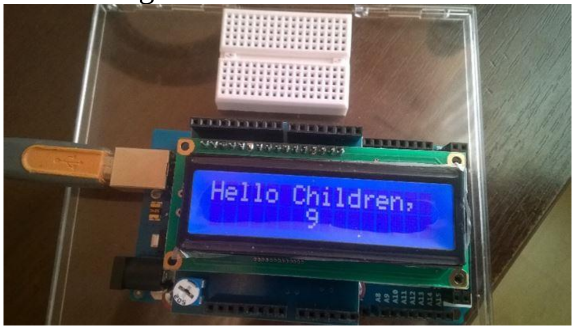


Fig.III.10. The result of the display operation

Example: generation of user defined characters
This example shows the user defined character generation and usage procedure

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

// Character matrix for the first character: every line is a
//row of pixels of the character
byte happy[8] = {
    B00000,
    B11011,
    B11011,
    B00000,
    B00000,
    B10001,
    B01110,
};

// Matrix for the second character
byte sad[8] = {
    B00000,
    B11011,
    B11011,
    B00000,
    B00000,
    B01110,
    B10001,
};

void setup() {
    lcd.begin(16, 2);
    // The 2 character are stored in the CGROM, user defined
    //area, pos. 0 and 1
    lcd.createChar(0, happy);
    lcd.createChar(1, sad);
    // Display the first line: a string followed by the 1-
    //stuser defined char
    lcd.setCursor(0, 0);
    lcd.print("Happy ");
    lcd.write(byte(0)); // See the difference between print
    //and write
    /* When you are referring the "0" user defined char you
    must write a cast to the "byte"
     type, otherwise the compiler throws an error. Exception
     is the case when you are referring a varaiable:
        byte zero = 0;
        lcd.write(zero);
    */

    // Display the second line
    lcd.setCursor(0, 1);
    lcd.print("Sad ");
    lcd.write(1); // when you are referring other characters
    //then "0" it is not necessary to cast
}

void loop()
{ }
```

3.2. Working with the interrupt system

Interrupts are events that require immediate attention from the microcontroller. As a response to such an event, the microcontroller stops its normal instruction execution sequence and “serves” the interrupt (it executes am Interrupt Service Routine (ISR) – a routine attached to that specific interrupt event). In order to respond to interrupt events, the microcontroller should have the Global Interrupt Enable bit and the Interrupt specific bit enabled. The following issues are essential in order to have proper interrupt system usage:

- The interrupt should be activated on its specific bit
- The Global Interrupt Enable bit (bit I in the Status Register – SREG) should be set
- The stack should be initialized (Arduino IDE does it by default)
- Every ISR ends with an RETI instruction (which resumes the normal program execution flow) – added by default by the Arduino IDE compiler

Interrupts can by of 2 types: internal and external. The internal ones are associated with events related to the microcontroller peripherals (ex. Timer/Counter, Analog Comparator etc.). External interrupts are triggered on external pins (ex. Buttons connected to these pins).

Each AVR MCU has an interrupt list which includes the type of the event that triggers the interrupt. When the interrupts are enabled and such an event occurs, the microprocessor will jump to a specific address in the program memory (referred by the entry in the Interrupt Vector Table (IVT) corresponding to the interrupt number). By writing an ISR (Interrupt Service Routine) and attaching its address in the IVT (Interrupt Vector Table) the system will be determined to execute a specific action as a response to a particular interrupt event.

In the first example external interrupts triggered by buttons will be used and corresponding messages will be displayed on the LCD. To run this example you need the

ATmega 2560 board with the LCD shield and a PModBtn peripheral module. The ATmega 2560 has 6 external interrupt pins (the pin correspondence is available here: http://arduino.cc/en/Hacking/PinMapping2560.

The buttons will be connected to the pins corresponding to the interrupts INT0 and INT1 (digital pins 20 and 21). In the figure below are shown the necessary connections are shown followed by the corresponding code.

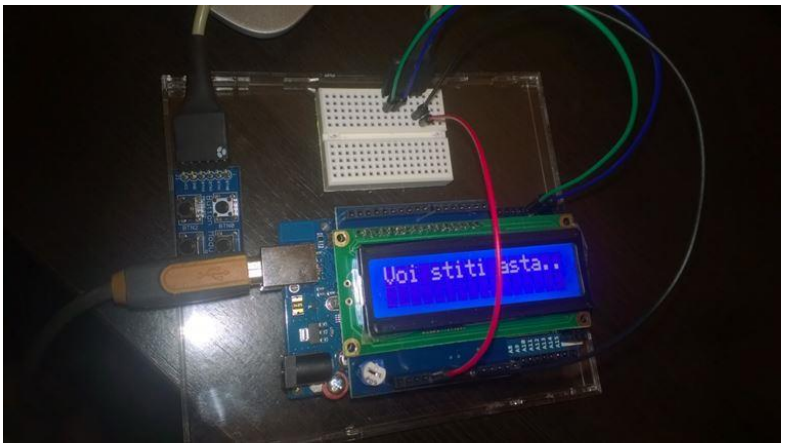


Fig.III.11. Connections for the 1-st example

```
// Include the header for the avr interrupt system
#include "avr/interrupt.h"
// Include the LCD library
#include <LiquidCrystal.h>

// Init the LCD
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
volatile int buttonVariable; //public variable that can be
//modified by the ISR

void setup(void)
{
    buttonVariable = 0; // Init the variable shared between
    the ISR and the main program

    // Set the LCD row and col number
    lcd.begin(16, 2);
    lcd.print("Incepe experimentul");
    delay(1000); // perform a 1 sec delay to display this
    //massage on the LCD

    // Set pin 21 as input (the pin corresponding to INT0)
    pinMode(21 ,INPUT);
    // Set pin 20 as input (the pin corresponding to INT1)
    pinMode(20, INPUT);

    pinMode(13, OUTPUT); // Set pin 13 as output
    digitalWrite(13, HIGH); // Lit up the onboard LED

    delay(1000);

    EIMSK |= (1 << INT0); // Activate INT0
    EIMSK |= (1 << INT1); // Activate INT1

    EICRA |= (1 << ISC01); // Specify INT0 triggering
    //behavior: falling edge of the
    EICRA |= (1 << ISC11); // Same behavior for INT1
    sei(); // Global interrupt system activation
    digitalWrite(13, LOW); // Turn off the onboard LED

    lcd.clear(); // Erase the LCD screen
}

void loop()
{
    // If an interrupt was triggered/executed the LCD has to
    //be erased and the main massage
    //displayed
    if(buttonVariable == 1)
    {
        lcd.clear(); // Erase the LCD
        buttonVariable = 0; // Global variable re-initialized
    }

    delay(1000);
    lcd.setCursor(0,0); // Set the LCD cursor
    lcd.print("Liniste…"); // Display a message
}

// ISR for INT0 ("INT0_vect" is a predefined name (address)
//for INT0 ISR
ISR(INT0_vect)
{
    digitalWrite(13, !digitalRead(13)); // Change the status
    //of pin 13
    lcd.setCursor(0,0); // Move the LCD cursor in the top-left
    //corner
    lcd.print("Intrerupem");// Display message
    lcd.setCursor(0,1);
    lcd.print("ptr stirea zilei");
    buttonVariable = 1;
}

// ISR for INT1
ISR(INT1_vect)
{
    digitalWrite(13, !digitalRead(13));
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Stirea Doi");
    buttonVariable = 1;
}
```

Delay() and millis() function does not work during an ISR (they are based on interrupts but the interrupt system is deactivated during the execution of an ISR). In generally, during the execution of an ISR the system does not respond to any other interrupt request. Therefore it is recommended that the execution time of an ISR to be as short as possible.

If a variable is modified inside an ISR and if you want to access its modified value globally, declare the variable as volatile. Hence the compiler knows that the **variable** can be modified any time, discarding any code optimization and placing the variable in the RAM.

Arduino allows the usage of the interrupt system without microcontroller specific knowledge by using generic functions. The first one is **attachInterrupt()**. It attaches an ISR to an external interrupt, replacing any previous attachment:

attachInterrupt(interrupt, ISR, mode)

The first parameter is the interrupt number but does not match neither the external interrupt number of the Atmega2560 nor the digital pin number of the board. It is a simple index. The table below shows the difference between these numbers:

attachInterrupt	Name	Pin on chip (TQFP)	Pin on board
0	INT4	6	D2
1	INT5	7	D3
2	INT0	43	D21
3	INT1	44	D20
4	INT2	45	D19
5	INT3	46	D18

Fig.III.12. The interrupt numbers vs. pins (source: http://www.gammon.com.au/interrupts)

Therefore the use of the **digitalPinToInterrupt**(pin) function is recommended – it will return the interrupt number attached to the digital pin (if exists):

The second parameter is the name (address) of the ISR.

The third parameter is the trigger response behavior: **RISING** edge, **FALLING** edge, **LOW** level or **HIGH** level or level toggle (**CHANGE**).

Function **noInterrupts()** disables the interrupts. They can be reactivated using: **interrupts()**.

The **detachInterrupt(interrupt)** function disables the interrupt with the number specified in the parameter.

The previous example was implemented using the AVR configuration registers, being dependent on the microcontroller specifications. In the following example the same functionality can be achieved using Arduino specific functions:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(7,6,5,4,3,2);
volatile int buttonVariable;

void setup()
{
    buttonVariable = 0;
    lcd.begin(16,2);
    lcd.print("A inceput");
    lcd.setCursor(0,1);
    lcd.print("din nou");
    delay(1000);

    // The 2 interrupt pins 21 and 20 declared as inputs with
    //pull-up resistors activated
    pinMode(20 ,INPUT);
    pinMode(21 ,INPUT);
    digitalWrite(20, HIGH);
    digitalWrite(21, HIGH);
    // Atach ISRs to the interrupts corresponding to pins 21
    //and 20 (INT0 and INT1)
    attachInterrupt(digitalPinToInterrupt(20), functieUnu,
    RISING);
    attachInterrupt(digitalPinToInterrupt(21), functieDoi,
    CHANGE);
}

void loop()
{
    // Insert here task for normal program flow ….
    lcd.print("Programul principal");
    delay(1000);
}

// First ISR function
void functieUnu()
{
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Functia Unu");
}

// 2-nd ISR function
void functieDoi()
{
    lcd.clear();
    lcd.setCursor(0,0);
    lcd.print("Functia Doi");
}
```

3.3. Individual Work

1. Run the presented examples.
2. Using the character generator, create an animation that indicates the time elapsed (a clock with a rotating arm, an hourglass etc.).
3. Create a timer using the interrupts. One button starts/stops the timer while the other resets the timer.
4. Combine 2 + 3: the animation runs as long the timer is running and is toped when the timer is stopped.
5. Bonus: use an Arduino board for generating signals of different frequencies and duty cycles (amount of time the signal is 1), by changing the state of a pin (using digitalWrite) at regular time intervals, controlled by delay() or millis(). Use the external interrupts of another board to measure the parameters of this signal, and display the parameters on the LCD.

References:

1.https://www.arduino.cc/en/Reference/LiquidCrystal
2.https://www.arduino.cc/en/Reference/AttachInterrupt

IV. Laboratory 4 – Usage of timers

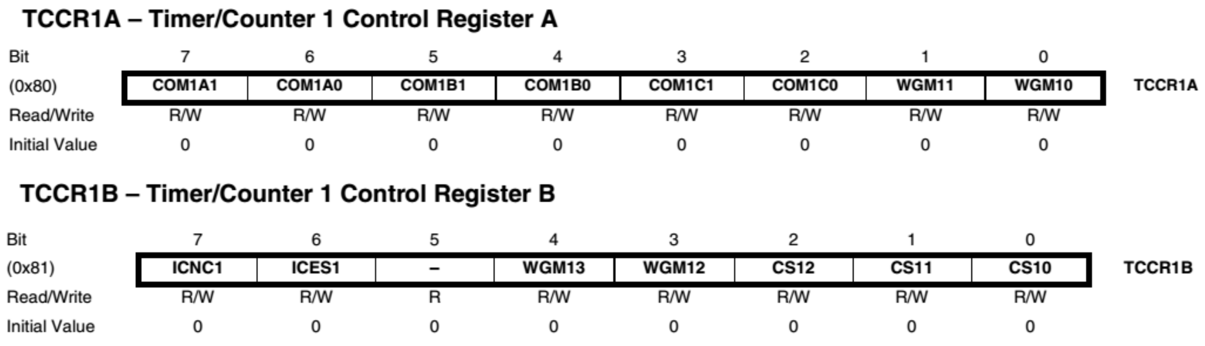
4.1. Timer based interrupts

Beside external interrupt, the MCU responds to internal ones which are triggered by external events (on the external pins). The source of the internal interrupts are hardware components inside the MCU (like the Timers).

Using the timer/counter interrupts, you can trigger events at specific time intervals (without using functions like *delay* and *millis*). Because these interrupts have an asynchronous behavior, the main program flow can be executed normally and only when a temporal threshold is reached a specific ISR is executed. The timer increments a *counter* register, and when its maximum counting capacity is reached an *overflow* bit (*flag*) is set. The flag can be manually checked or it can trigger an interrupt request. At the ISR execution the fag is reset.

Each timer needs a clock source, which most commonly is the development board's oscillator; its frequency can be divided by an auxiliary counter (*prescaler*).

Arduino Mega has 5 counters (2 / 8 bits and 3 / 16bits). The Arduino UNO has 2 / 8 bits and 1 / 16bits counters. To use the counters the configuration registers must be properly set. Two of them are the Timer Counter Control Registers TCCRxA and TCCRxB (x denotes the timer number). To start the timer the most important bits are those that set the clock frequency division (*prescaler*) (and consequently the counting speed): CSn2, CSn1 and CSn0.



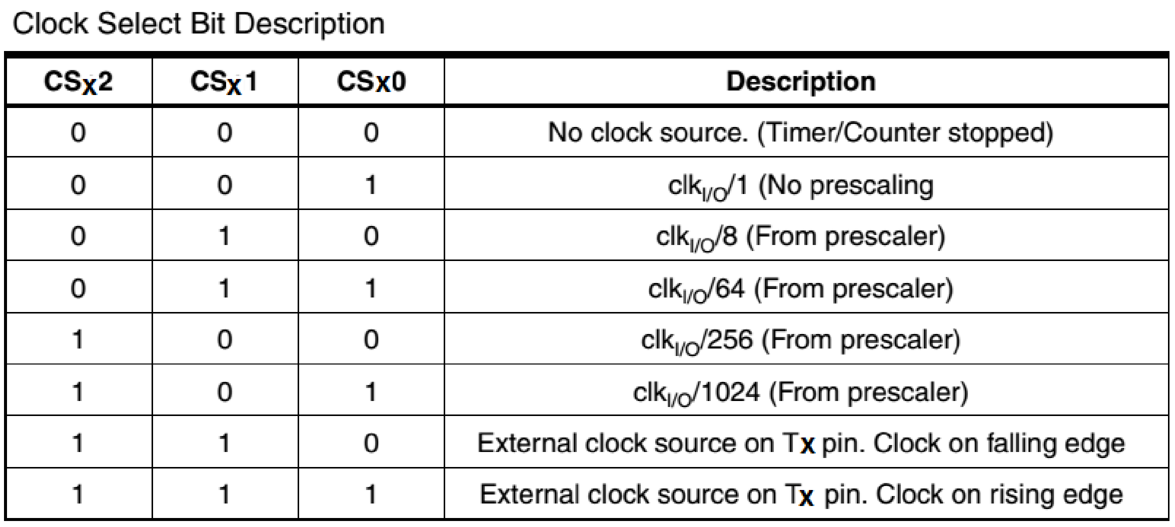
TCCR1A – Timer/Counter 1 Control Register A

Bit	7	6	5	4	3	2	1	0	
(0x80)	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	TCCR1A
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

TCCR1B – Timer/Counter 1 Control Register B

Bit	7	6	5	4	3	2	1	0	
(0x81)	ICNC1	ICES1	–	WGM13	WGM12	CS12	CS11	CS10	TCCR1B
Read/Write	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig.IV.1. Timer 1 control registers (source: Atmega2560 datasheet)



Clock Select Bit Description

CSx2	CSx1	CSx0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	$clk_{I/O}$/1 (No prescaling
0	1	0	$clk_{I/O}$/8 (From prescaler)
0	1	1	$clk_{I/O}$/64 (From prescaler)
1	0	0	$clk_{I/O}$/256 (From prescaler)
1	0	1	$clk_{I/O}$/1024 (From prescaler)
1	1	0	External clock source on Tx pin. Clock on falling edge
1	1	1	External clock source on Tx pin. Clock on rising edge

Fig.IV.2. Clock source selection for Timer1 (source: source: Atmega2560 datasheet)

By default CSx2, CSx1 and CSx0 are 0, meaning that the timer does not have a clock signal and therefore is stopped. A valid configuration for the prescaler means starting the timer. If the selected clock source is external, the timer will work only if such an external source is connected to the board (which is not the case at the lab – such settings will be avoided!!!).

Below is a summary of the most important registers used with the timers:

- TCCRx - Timer/Counter Control Register: here you can set the clock source.
- TCNTx - Timer/Counter Register: stores the current value of the timer's counter.
- OCRx - Output Compare Register: value set by the user and compared with the TCNTx to generate waveforms or trigger events.
- ICRx - Input Capture Register: used to measure time intervals between external events (only for 16 bit timers).
- TIMSKx - Timer/Counter Interrupt Mask Register: to enable/disable timer interrupts.
- TIFRx - Timer/Counter Interrupt Flag Register: signals the existence of an interrupt request.

In the following example we will increment a variable when TIMER1 overflows. The value of the incremented variable will be displayed on the LCD.

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <LiquidCrystal.h>

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
volatile int myVar;

void setup()
{
    myVar = 0;
    //init Timer1
    cli(); // disable the global interrupts system in order to
    //setup the timers
    TCCR1A = 0; // SET TCCR1A and B to 0
    TCCR1B = 0; // timer set to Normal mode (WGMx3:0 = 0)
    TIMSK1 = (1 << TOIE1); //timer overflow interrupt enable
    //for timer 1
    //Set the prescaler to 1024
    // CPU freq. is 16 MHZ and Timer1 is on 16 bits
    // Counter increment at every dt = 1024 / (16 * 10^6) sec
    // Overflow event occurs at every: dt * 2^16 = 4.194 sec.
    TCCR1B |= (1 << CS10);
    TCCR1B |= (1 << CS12);
    lcd.begin(16, 2);
    lcd.print("Timers");
    sei(); // enable the global interrupts system
}

void loop()
{
    lcd.setCursor(0,1);
    lcd.print(myVar);
    lcd.setCursor(5, 1);
    lcd.print(TCNT1);
}

ISR(TIMER1_OVF_vect)
{
    myVar = myVar + 1;
}
```

In order to trigger the timer interrupt at customized time intervals (not only on timer overflow), the CTC (Clear Timer on Compare match) mode must be used. This way, the value of the TCNTx register will be compared with the value of the OCRx register (set by the user) and at equality the value of TCNTx is reset (becomes zero).

In order to obtain a T time interval between consecutive interrupt requests, the value pf OCRx register must be set by the user:

$$\mathbf{OCRx + 1 = T / (P / (16 * 10\char`^6))} \qquad \textbf{(1)}$$

Where:

T – the time interval between consecutive events (ex. 1 sec)
P – prescaler value (ex. 1024) (+ 1 is added to OCRx because the timer reset takes 1 clock cycle)

Applying eq. 1 for T=1 sec, we get OCRx =15624.

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

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
volatile int myVar;
void setup()
{
    // Initialize Timer1
     cli();// disable the global interrupts system in order to
     //setup the timers
    TCCR1A = 0; // SET TCCR1A and B to 0
    TCCR1B = 0;
    // Set the OCR1A with the desired value:
    OCR1A = 15624;
    // Active CTC mode:
    TCCR1B |= (1 << WGM12);
    // Set the prescaler to 1024:
    TCCR1B |= (1 << CS10);
    TCCR1B |= (1 << CS12);
     // Enable the Output Compare interrupt (by setting the
     //mask bit)
    TIMSK1 |= (1 << OCIE1A);

    lcd.begin(16, 2);
    lcd.print("Timere cu CTC");

    sei(); // enable global interrupts
}

void loop()
{
    lcd.setCursor(0,1);
    lcd.print(myVar);
    lcd.setCursor(5, 1);
    lcd.print(TCNT1);
}

ISR(TIMER1_COMPA_vect)
{
     myVar = myVar + 1;
}
```

In contrast to the previous example, when the variable is incremented at overflow (about every 4 seconds), in the current example we have more control over the incrementing period of the variable (once per second).

In the following a much simpler method for using the timers, based on the TimerOne library, is presented. The TimerOne library can be downloaded from the following location: http://playground.arduino.cc/Code/Timer1, where detailed installation and usage instructions are given.
In the example below a routine is called every second:

```
#include <TimerOne.h>
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

volatile int myVar;

void setup(void)
{
    Timer1.initialize(1000000);  // init the timing interval
    //for event triggering (1s = 10^-6s)
    Timer1.attachInterrupt(ShowMessage);  // The function is
    //called at the preset time interval
}

void ShowMessage(void)
{
    lcd.setCursor(0,0);
    lcd.print(myVar);
    myVar++;
}

void loop(void)
{
    // Do something else …
}
```

4.2. Tones generation

For that purpose Arduino offers the **tone** function, which generates specified frequency pulses with 50% fill factor. To call the function the length of the signal must be specified (otherwise the tone will be generated until the **noTone()** function is called. Minimum tone frequency is 31 HZ. In order to generate tones on different pins, **noTone()** must be called before switching the output pin.

The format of the tone function is:

tone (pin, frequency, duration)
or
tone (pin, frequency, duration)

In order to run the following example, the following connection must be made: connect **the red (signal) pin** of the speaker to the **digital pin 8 of the Arduino**. The **black pin** of the speaker must be connected to the ground (**GND**).

Afterwards, create a file named pitches.h. To create a new file inside a project, press the top-right button from the IDE and select the "New Tab" option (see figure below):

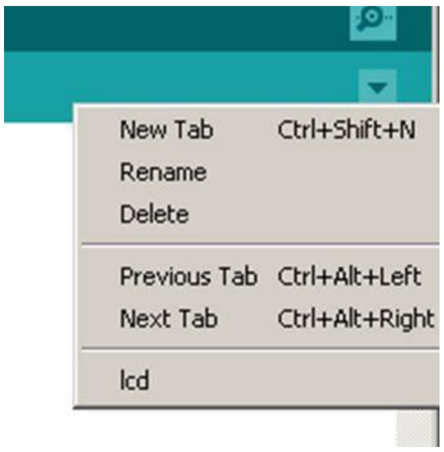


Fig.IV.3. Creating a new file

Rename the new file *pitches.h* and insert the tones defined in “Appendix A”. Save the document and go back to the principal tab, where enter the following code:

```
#include "pitches.h" // contains the frequency values for all
//the notes

// notes in the melody - constant values defining frequency
for each used note
int melody[] = {
    NOTE_C4,  NOTE_G3,NOTE_G3,  NOTE_A3,  NOTE_G3,0,  NOTE_B3,
NOTE_C4};

// note durations: 4 = quarter note, 8 = eighth note, etc.:
int noteDurations[] = { 4,8,8,4,4,4,4,4 };

void setup()
{
    for (int thisNote = 0; thisNote < 8; thisNote++) {
        // iterate over the notes of the melody:
        // to calculate the note duration, take one second
        //divided by the note type.
        // e.g. quarter note = 1000 / 4, eighth note = 1000/8,
        //etc.
        int noteDuration = 1000/noteDurations[thisNote];
        tone(8, melody[thisNote],noteDuration);
        // to distinguish the notes, set a minimum time
        between them: note's duration + 30%
        int pauseBetweenNotes = noteDuration * 1.30;
        delay(pauseBetweenNotes);
        noTone(8); // stop the tone playing for current note
    }
}
void loop() { }   // no need to repeat the melody
```

4.3. PWM signal generation using Arduino

PWM (Pulse Width Modulation) is a technique used to obtain an analog signal from a digital one by alternating periodically the output between "1" and "0". The fraction of time for which the signal is high ("1") is called duty cycle. With Arduino the PWM signal can be generated in 3 ways:

- using the PWM working modes of the AVR timers
- using the analogWrite function
- by varying in software the amount of time at which a pin's output is logic high

In the following, the **analogWrite**(duty-cycle) function will be used. The possible values for the parameter duty_cycle are between 0 .. 255 (0 means no signal is generated; 255 means a continuous signal of 5 V is generated). Any value X in between means a signal with a duty cycle of X/255. The figure below illustrates some examples:

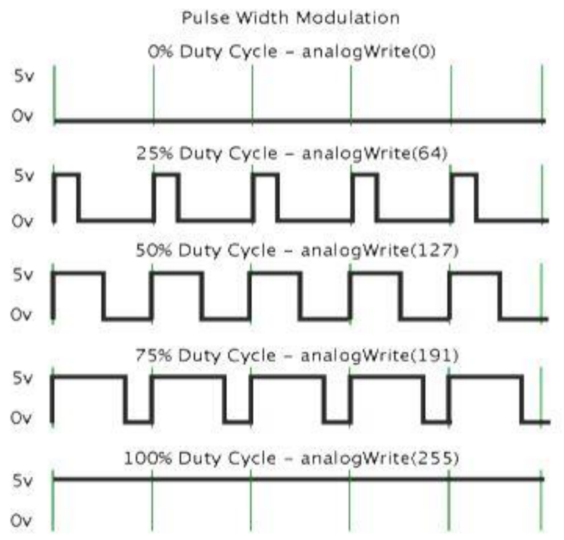


Fig.IV.4. PWM signals with several duty cycles (source: https://www.arduino.cc/en/Tutorial/PWM)

In the following example we will generate a PWM signal for the piezoelectric speaker and a PWM signal with the same duty cycle for the on-board LED. The signal pin of the piezoelectric speaker is (red wire) is connected to digital pin 8 and the black wire to the ground.

```
int  buzerPin  =  8;  //  pin  for  attaching  the  piezoelectric
//speaker
int puls = 0;     // duty cycle, initially 0
int step = 10;     // duty cycle increment step
int ledPin = 13; // on-board LED

void setup()  {
    // Pin  direction setup
    pinMode(buzerPin, OUTPUT);
    pinMode(ledPin, OUTPUT);
}

void loop()  {
    // set the duty cycle for both the buzzer and the LED
    analogWrite(buzerPin, puls);
    analogWrite(ledPin, puls);
    // increment the duty cycle
    puls = puls + step;

    // change the increment direction at the end of the
    //interval
    if (puls <= 0 || puls >= 255) {
        step = -step ;
    }
    // small delay to sense the effects
    delay(30);
}
```

4.4. Individual work

1. Implement all the examples. Ask the teacher for any question related to theoretical or practical aspects.
2. Using the interrupts, play the melody in the background while the main program displays an animation on the LCD (ex. walking character). The animation speed should be variable to demonstrate the asynchronous playing of the melody relative the main program.
3. Using the button block, implement a mini-piano with 4 notes. Be sure to end the sound generation when the button is released.
4. Using the PWM technique and an LED block, implement an animation through which the intensity of each LED is varied continuously

References:

1.Datasheet ATMega 2560 http://www.atmel.com/images/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf
2. Timer 1 library http://playground.arduino.cc/Code/Timer1
3. analogWrite functions https://www.arduino.cc/en/Reference/AnalogWrite
4. Tones generation: https://www.arduino.cc/en/Reference/Tone

Appendix A

(contents of “pitches.h” file, from: https://www.arduino.cc/en/Tutorial/toneMelody)

```
#define NOTE_B0  31
#define NOTE_C1  33
#define NOTE_CS1 35
#define NOTE_D1  37
#define NOTE_DS1 39
#define NOTE_E1  41
#define NOTE_F1  44
#define NOTE_FS1 46
#define NOTE_G1  49
#define NOTE_GS1 52
#define NOTE_A1  55
#define NOTE_AS1 58
#define NOTE_B1  62
#define NOTE_C2  65
#define NOTE_CS2 69
#define NOTE_D2  73
#define NOTE_DS2 78
#define NOTE_E2  82
#define NOTE_F2  87
#define NOTE_FS2 93
#define NOTE_G2  98
#define NOTE_GS2 104
#define NOTE_A2  110
#define NOTE_AS2 117
#define NOTE_B2  123
#define NOTE_C3  131
#define NOTE_CS3 139
#define NOTE_D3  147
#define NOTE_DS3 156
#define NOTE_E3  165
#define NOTE_F3  175
#define NOTE_FS3 185
#define NOTE_G3  196
#define NOTE_GS3 208
#define NOTE_A3  220
#define NOTE_AS3 233
#define NOTE_B3  247
#define NOTE_C4  262
#define NOTE_CS4 277
#define NOTE_D4  294
#define NOTE_DS4 311
#define NOTE_E4  330
#define NOTE_F4  349
#define NOTE_FS4 370
```

```
#define NOTE_G4  392
#define NOTE_GS4 415
#define NOTE_A4  440
#define NOTE_AS4 466
#define NOTE_B4  494
#define NOTE_C5  523
#define NOTE_CS5 554
#define NOTE_D5  587
#define NOTE_DS5 622
#define NOTE_E5  659
#define NOTE_F5  698
#define NOTE_FS5 740
#define NOTE_G5  784
#define NOTE_GS5 831
#define NOTE_A5  880
#define NOTE_AS5 932
#define NOTE_B5  988
#define NOTE_C6  1047
#define NOTE_CS6 1109
#define NOTE_D6  1175
#define NOTE_DS6 1245
#define NOTE_E6  1319
#define NOTE_F6  1397
#define NOTE_FS6 1480
#define NOTE_G6  1568
#define NOTE_GS6 1661
#define NOTE_A6  1760
#define NOTE_AS6 1865
#define NOTE_B6  1976
#define NOTE_C7  2093
#define NOTE_CS7 2217
#define NOTE_D7  2349
#define NOTE_DS7 2489
#define NOTE_E7  2637
#define NOTE_F7  2794
#define NOTE_FS7 2960
#define NOTE_G7  3136
#define NOTE_GS7 3322
#define NOTE_A7  3520
#define NOTE_AS7 3729
#define NOTE_B7  3951
#define NOTE_C8  4186
#define NOTE_CS8 4435
#define NOTE_D8  4699
#define NOTE_DS8 4978
```

V. Laboratory 5 – Communication Interfaces

Embedded electronics refers to the interconnection of circuits (micro-processors or other integrated circuits) with the goal of creating a unified system. In order to transfer information, these circuits need to have a common communication mechanism. These mechanics can be categorized in two main classes: **serial and parallel** communication schemes.

Parallel interfaces transfer more bits at a time. They usually require busses to transmit over 8, 16 or more lines. The transmitted and received data represent massive flows of 1s and 0s. In figure V.1, one can observe a simple 8-bit data bus, controlled by a clock signal, that can transmit one byte at each clock cycle (9 wires are used).

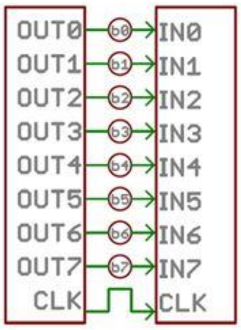


Fig.V.1. Parallel Transmission (source: https://learn.sparkfun.com/tutorials/serial-communication/all)

Serial interfaces send the information bit by bit. These interfaces can operate on a single wire and usually do not need more than 4 wires (minimum 1, maximum 4 wires).

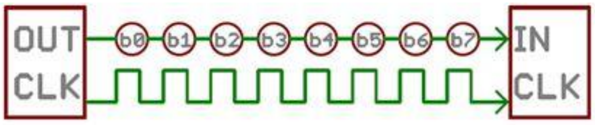


Fig.V.2. Serial Transmission (source: https://learn.sparkfun.com/tutorials/serial-communication/all)

In Figure V.2, one can observe an example of a serial interface that transmits one bit at each clock cycle (2 wires are used). Although the parallel interfaces have their strengths, they require a large number of pins on the used development platform and taking into account that the number of pins on the Arduino UNO/Mega is limited, we will focus on serial communication interfaces.

Another classification used for communication interfaces refers to the communication mechanism: **synchronous or asynchronous**. A synchronous communication interface uses a clock signal at both ends of the communication link (transmitter and receiver). This kind of communication is usually faster, but it requires an extra wire between the communication devices. Examples of such communications are SPI and I2C. Asynchronous communication refers to the fact that data is being transferred without the support of a clock signal. In this manner there is no need for a clock signal, but special attention should be given to the synchronization of the transferred data.

Rules of asynchronous serial transfer – UART

Asynchronous serial transfer employs a mechanism to ensure a robust error free transmission. This mechanism contains the following:

- Baud Rate or transfer rate
- The Data Packet (Data Frame)
- Data Bits – character (data chunk)
- Synchronization Bits
- Parity Bits
- Data line (**in idle state has the logical level "1"**)

The critical part is to ensure that the devices that communicate using the serial bus obey the same communication protocol.

Baud Rate

In serial communication Baud Rate defines how fast is the data transmitted over the serial line. This is expressed in number of bits per second. Baud rate examples: 1200, 2400, 4800, 19200, 38400, 57600, 115200.

Data Packet (Data Frame)

Each block of data (usually a Byte) that is to be transmitted is embedded into a data packet (frame). The data packet is formed by adding synchronization and parity bits to the data that is to be transmitted. Figure V.3 illustrates the form of a data packet.

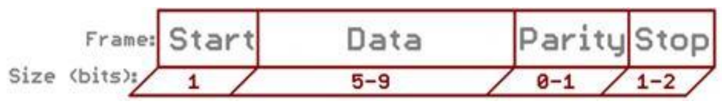


Fig.V.3. Data Packet – Data Frame (source: https://learn.sparkfun.com/tutorials/serial-communication/all)

Data chunk

The most important part of each packet is represented by the data that a packet contains. This part is also named the data chunk, since the data size is not always the same. It is usually called a character. The data size can be between 5 and 9 bits – standard data size is 8 bits. After the data size is chosen, the communication devices must also agree on the **endianness** (which bit is transmitted first, the most significant bit MSB or the least significant bit LSB).

Synchronization Bits

The synchronization bits are special bits transmitted with every character. These are the **start** and **stop** bits, marking the beginning and ending of a data packet. The start bit is always 0. There can exist more than one stop bit. The stop bit is always 1.

Parity Bit

The parity bit assures a very primitive error control mechanism. The parity bit can represent odd parity, even parity, or it can be skipped from the data packet. For producing the parity bit (Figure V.4) all the bits from the data chunk are aggregated using the “exclusive or” operator. Parity checking is optional, not used so much in practice. It is useful to use the parity bit when transmitting through noisy medium.

$$
\begin{aligned}
P_{even} &= d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0 \\
P_{odd} &= d_{n-1} \oplus \ldots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1
\end{aligned}
$$

Fig.V.4. Parity bit Computation (source: https://learn.sparkfun.com/tutorials/serial-communication/all)

An asynchronous serial bus contains only 2 wires – one for transmitting the data and one for receiving. So, the components that communicate in serial mode must have 2 pins: receive pin (**RX**) and transmit pin (**TX**), as in Figure V.5.

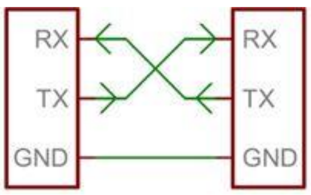


Fig.V.5. Serial Connection Scheme (source: https://learn.sparkfun.com/tutorials/serial-communication/all)

All the Arduino development boards contain at least a **Serial** port (known as UART or USART). Serial communication can be realized through pins 0 (RX) and 1 (TX), but also through the USB interface (the USB interface uses pins 0 and 1 to communicate with the microcontroller). This is why digital pins 0 and 1 must not be used for applications, because losing control on this pins means losing control for board programming.

The Arduino Mega development board contains 3 additional serial ports Serial1 – on pins 19 (RX) and 18 (TX), Serial2 – on pins 17 (RX) and 16 (TX) and Serial 3 – on pins 15 (RX) and 14 (TX).

In our first example in this laboratory we will implement the serial communication between the Arduino board and the PC; displaying the received message on the PC. For this example, we will use the LCD shield mounted on the development boards. The information is sent from the PC and displayed on the LCD. There exist a variety of functions for serial data manipulation. In the previous laboratories we have used a serial communication between the Arduino board and the PC for showing the state of the pressed buttons. The most used serial functions are shown in Figure V.6 (https://www.arduino.cc/en/Reference/Serial).

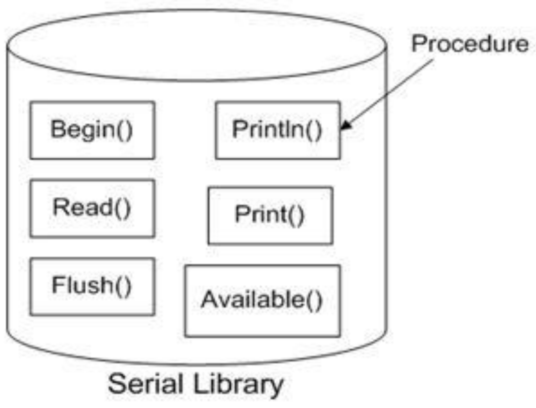


Fig.V.6. Asynchronous Serial Communication Functions (source: https://www.arduino.cc/en/Reference/Serial)

Functions **print()** and **println()** from the Serial class send data through the serial port. The difference is that println() sends an additional new line character ('\n') and a "carriage return" character ('\r') at the end of the transmitted message. For the transmitted numbers you can also specify the format of the data (HEX, DEC, etc.).

Function **begin()** is used to initialize the baud rate of the serial communication. For serial communication with the PC the following baud rates are usually used: 300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200. Optionally, you can add an additional parameter for configuring the number of bits sent, parity and number of stop bits. The following values are implicit: **8 data bits, no parity and 1 stop bit**.

Function **read()** is used to read the date received from the serial interface. The syntax is the following: **IncomingByte = Serial.read();**

Function **write()** sends a series of bytes over the serial line. For sending numbers it is recommended to use the function **print()**.

The **flush()** instruction waits for the serial transmission to complete.

Function **available()** returns the number of bytes that can be read from the serial port. These data have already been received and are available in the serial receive buffer.

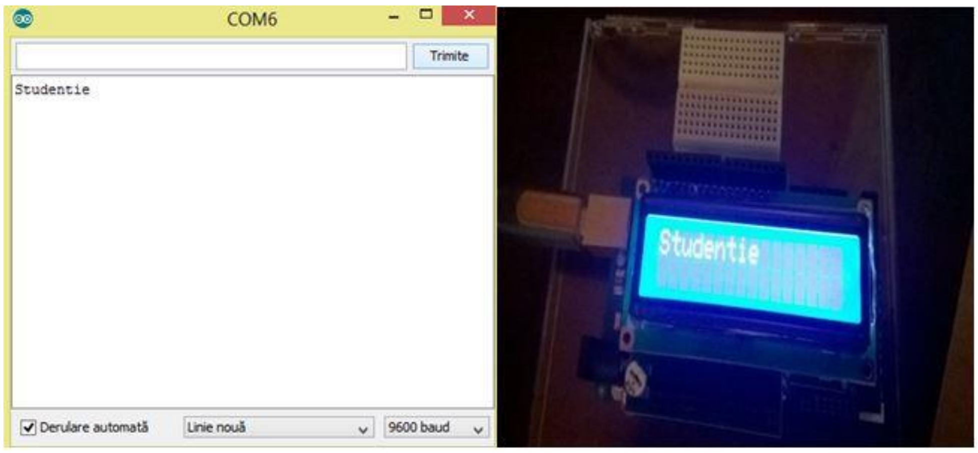
A useful function is **serialEvent().** The function is defined by the user and will be automatically called when new data is ready to be read from the serial receive buffer.

The following example illustrates receiving data from the serial monitor and displaying it on the LCD.

```
//include LCD library
#include <LiquidCrystal.h>
String inputString = "";// create an empty string to hold the
//incoming serial data
```

```
// Boolean flag to test whether a complete string has been
//received (enter pressed in serial monitor)
boolean stringComplete = false;
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

void setup() {
    // initialize serial interface
    Serial.begin(9600); // implicit serial frame format
    //initialize lcd
    lcd.begin(16, 2);
    // reserve 200 bytes for the string
    inputString.reserve(200);
}

void loop() {
    // display the string when new line is received
    if (stringComplete) {
        lcd.setCursor(0, 0);
        lcd.print(inputString);
        Serial.println(inputString);
        // empty the string
        inputString = "";
        // reset the flag
        stringComplete = false;
    }
}

/*
SerialEvent is called when new data is received on the RX port
This function is automatically called when the loop is called.
If we add delays in loop we will also delay the showing of the
result. */
void serialEvent() {
    while (Serial.available()) {
        // read the new received byte
        char inChar = (char)Serial.read();
        // check if new line character has been received; if
        //not, add it to the string
        // we do not add new line in input string since it
        //will show an extra character in the LCD
        if (inChar != '\n')
        inputString += inChar;
        // dif the receive character is new line, set the flag
        //so that the loop will know to display
        //the received data
        if (inChar == '\n') {
            stringComplete = true;
        }
    }
}
```

For transmitting data to the Arduino board, use the Serial Monitor, opened from the Tools menu.



```
Studentie
```

☑ Derulare automată   Linie nouă   9600 baud

Fig.V.7. Serial Data transmission in Serial Monitor

Inter-Integrated Circuit (I2C) protocol

The **Inter Integrated Circuit (I2C)** protocol was created for connecting more “slave” integrated circuits with one or more “master” circuit. This communication protocol is intended for short distance communications and similarly to the UART protocol or RS232 requires two wires for sending / receiving information.

Unlike UART, I2C allows communication between more than two devices. One device must be *master*, and it will communicate with a *slave*, but these roles can subsequently change.

The I2C bus is composed of 2 signals: SCL and SDA. SCL represents the clock signal; SDA the data signal. The clock signal is always generated by the *master*. (Some *slave* components can force the clock signal to low in order to signal the *master* to introduce a delay in the data transmission – this is called clock stretching).

Unlike other serial communication protocols, I2C is an “open drain” bus, that is a signal line can be pulled to 0 logic level (“low”) but it cannot be pulled to 1 logical level (“high”). In this manner “bus contentions” are eliminated – a device tries to pull a signal to “high” while another devices tries to pull the signal to “low”. Each signal has a pull-up resistor in order for the device to be able to set the signal to “high” when no other device is trying to pull the signal to “low”.

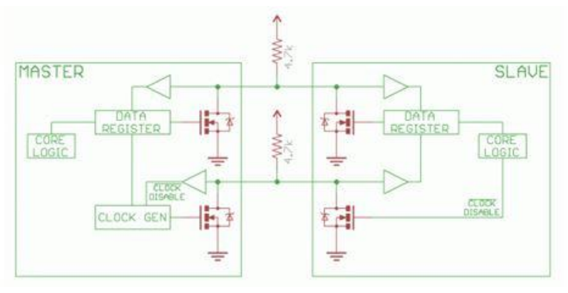


Fig.V.8. I2C connection scheme. There is a resistor on each signal (source: https://learn.sparkfun.com/tutorials/i2c/all)

The selection for the resistor varies with the devices that use the bus, but as a rule start with 4.7k resistors and decrease them if necessary.

Protocol Description:

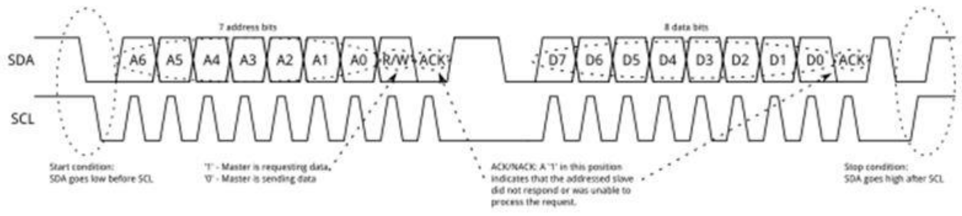

7 address bits
8 data bits
SDA
SCL
A6
A5
A4
A3
A2
A1
A0
R/W
ACK
D7
D6
D5
D4
D3
D2
D1
D0
ACK
Start condition: SDA goes low before SCL
'1' - Master is requesting data, '0' - Master is sending data
ACK/NACK: A '1' in this position indicates that the addressed slave did not respond or was unable to process the request.
Stop condition: SDA goes high after SCL

Fig.V.9. I2C Transmission Protocol (source: https://learn.sparkfun.com/tutorials/i2c/all)

I2C message are split in 2 types of frames: address frames (where the *master* selects the *slave* to which the data will be sent) and data frames that contain 8 bits of data to be passes from *master* to *slave* or from *slave* to *master*. The data are put on the SDA line after SCL is “low” and are sampled when SCL reaches the “high” logical level.

Start Condition

In order to initiate the address frame, the *master* leaves SCL to high and pulls SDA low. This prepares all the *slaves* for transmission. If 2 *master* devices want to send data over the I2C bus, the one that pulls SDA low first is granted control of the bus.

Address Frame

The address frame is always the first in message in the I2C communication. The address bits are transmitted first, MSB to LSB, followed by the R/W bit, showing if the operation is a read (1) or a write (0). The 9th bit of the address frame is the NACK / ACK bit. After the first 8 bits of the address frame are transmitted the receiving device gets control of the SDA line.

If the receiving device does not pull the SDA line to 0 logic before the 9th clock cycle, it can be inferred that the receiving device either did not receive the message or did not correctly interpret the message. In this case the communication is stopped and the *master* decides what to do next.

Data Frames

After the address frame has been sent, the useful information can be transmitted on the I2C bus. The *master* will continue to generate the clock signal, at a regular interval, and the data will be set on the SDA line, either by the *master* or by the *slave* (bit R/W indicates if it is a read or write operation). The number of data frames is arbitrary.

Stop condition

As soon as all the data frames have been transmitted, the *master* will generate a stop conditions. Stop conditions are defined as transitions from low to high (0 → 1) on the SDA line, after a 0 → 1 transition on SCL with SCL remaining high. During the write operations the value on SDA should not change when SCL is high, in order to avoid false stop conditions.

Example:

For testing the I2C protocol functionality recreate the mounting from Figure V.10 or Figure V.11 (depending on the used boards).

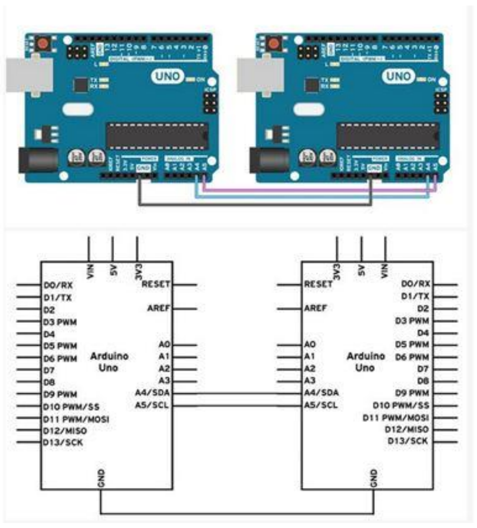


Fig.V.10. Connecting 2 Arduino Uno Boards (source: https://www.arduino.cc/en/Tutorial/MasterWriter)

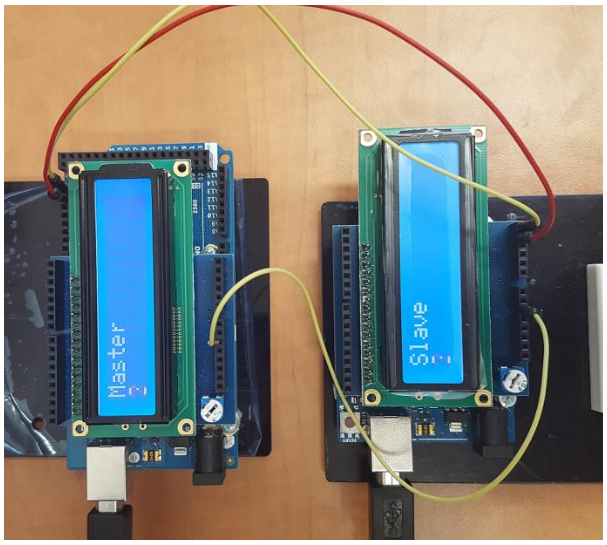


Fig.V.11. Connecting an Arduino Mega Board with an Arduino Uno Board

If you are using an Arduino UNO / dumilanove / mini board connect pins A4 and A5 of one board to the same pins on the other board. Also connect the GND signal from the two boards.

Do not connect the voltage pins from the two boards together and be sure the boards are powered up at the same voltage level.

In order to write the code we will use the Wire library from Arduino environment (https://www.arduino.cc/en/Reference/Wire).

In the following table you can see the I2C pin locations on different Arduino boards.

Board	**I2C**
UNO, Ethernet	A4 (SDA), A5 (SCL)
MEGA 2560	20 (SDA), 21 (SCL)
Leonardo	2 (SDA), 3 (SCL)
Due	20 (SDA), 21 (SCL), SDA1, SCL1

Fig.V.12. Location of the I2C signals on different Arduino boards

Cod for the *slave*:

```
#include <LiquidCrystal.h>

// include I2C library
#include <Wire.h>
```

```
int x = 0;

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);

void setup() {
    // Start i2C slave at address 9
    Wire.begin(9);
    // attach a function to be called when we receive
    //something on the I2C bus
    Wire.onReceive(receiveEvent);

    lcd.begin(16,2);
    lcd.print("Slave");
}

void receiveEvent(int bytes) {
    x = Wire.read();    //read I2C received character
}

void loop() {
    lcd.setCursor(0,1); // display received character
    lcd.print(x);
}
```

Code for the *master*:

```
#include <LiquidCrystal.h>

// include I2C library
#include <Wire.h>

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
int x = 0;

void setup() {
    // Open I2C bus as master
    Wire.begin();
    lcd.begin(16,2);
    lcd.print("Master");
}

void loop() {
    Wire.beginTransmission(9); // transmit to device #9
    Wire.write(x);              // transmit x
    Wire.endTransmission();    // stop transmission

    lcd.setCursor(0,1);  // display sent character on LCD
    lcd.print(x);

    x++; // increment x
    if (x > 5) x = 0; // reset x when it reaches 6
    delay(500);
}
```

5.1. Individual Work

1. Test the examples from the laboratory. Ask your TA if you have any problems in connecting the wires.

2. Implement a communication system between 2 PCs by using the Arduino boards. The boards are connecting to the PC via USB and between them through I2C. The text written in Serial Monitor on the PC connected to the *master* will be written in the Serial Monitor of the *slave* PC.

3. Implement the same setup as in problem 2, but bi-directional. For transmitting from *slave* to *master* read: https://www.arduino.cc/en/Tutorial/MasterReader

4. Implement a network with one *master* and 2 *slaves*. The *master* is connected to the PC and receives messages through the serial interface of the following type:

    a. s1-hello
    b. s2-goodbye

According to the number after the s character, the message after the '-' will be sent to the appropriate *slave*. The *slave* boards will display on LCD only the message addressed to them.

VI. Laboratory 6 - Working with the AVR Assembly language

6.1. Assembly and C

Why work in Assembly?

The assembly code directly translates into AVR instructions to be executed by the microcontroller, without compiler or environment overhead. While performant C/C++ compilers may sometimes produce very efficient code in terms of speed or memory requirements, the assembly code gives the programmer full control over the binary code that is produced. Most of the time the assembly code is smaller, faster, more predictable in terms of time and memory requirements, and easier to debug.

Assembly directives

The assembly directives are not part of the assembly language (which is made by the opcodes for the instructions that can be directly executed by the target microprocessor), but they instruct the compiler in the process of code generation.

Examples of uses:

- adjust the location of the program in memory
- define macros
- initialize memory

Some examples:

.byte	Reserve byte to a variable
.comm	declares a common symbol
.data	specifies the Data section of a program.
.ifdef	tells the assembler to include the following code if the condition is satisfied.
.else	tells the assembler to include the following code if the if condition is false
.include	include supporting files
#include	include supporting files
.file	start of a new logical file
.text	assemble what follows after this directive, defines the code section
.global	defines or acknowledges a global variable or subroutine, often used with a variable specified by .comm, if the variable is used in more than one file
.extern	treats all undefined symbols as external
.space	allocates space (for an array)
.equ	define constants for use in a program
.set	define (local) variables used in a program

Fig.VI.1. Example of assembly directives

Remembering functions

If we want to combine the C/C++ language with the assembly language, we can do this by means of functions (procedures). Functions must be declared (in the function's prototype) and defined (in the function's body). The functions may or may not have inputs or outputs. Generally, if a variable is passed as an argument to a function, that variable is expected to remain unchanged when the function is executed. Global Variables may be changed inside a function.

Calling conventions determine where the arguments and the return values are stored. When a function is called, it needs to know where to look for the arguments, and where to store the return value.

The CALL of a function pushes the return address onto the stack, and the arguments and the return value are passed in accordance to the calling convention (in registers, on the stack, etc). If the function uses global variables, they need to be declared as such, and initialized with the proper values.

After the function is executed, it finishes with the RET instruction, which pops the return address off the stack.

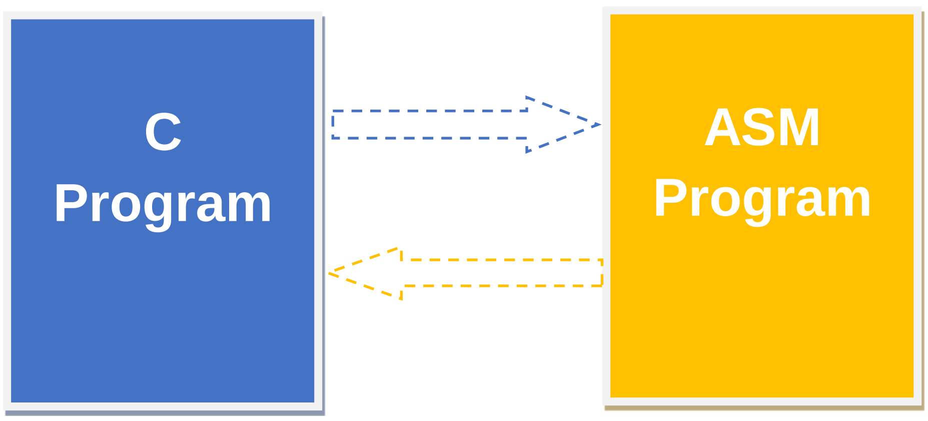


Fig.VI.2. Values are passed based on GCC convention.
C functions can only "return" one value.

Functions and the stack

When a function is called, an activation record is "pushed" on the stack. When the function returns, that activation record is "popped" from the stack Activation record is whatever data needed to be remembered to resume the process when the function returns. It typically consists of local variables (which are usually stored in registers) and the return address.
Stack is a memory area in SRAM pointed to by the 16 bit stack pointer (SP). At the AVR microcontrollers, the stack pointer is decreased when data is pushed on the stack, and therefore the stack must be initialized to the highest available data memory address (which at AtMega2560 is 0x21FF).

Global variable use

The global variables are accessible anywhere in your code, no matter if the code is C or assembly.

Global variables can be declared in the .ino file, but it is better to declare them in a header file. The globals should have equivalents in the .S assembly file (declared with the assembly directives .comm and .global).

Example declaration of the global variables in the .ino file, or in a header file:

```
extern "C" int8_t var8b;
extern "C" int16_t var16b;
extern "C" uint32_t var32b;
```

The declaration of these global variables in the assembly (.S) file:

```
.data
.comm var8b, 1
.global var8b
.comm var16b, 2
.global var16b
.comm var32b, 4
.global var32b
```

In C (Arduino) a global variable is just used as it is, whereas in ASM you may have to access one byte at a time.

Example:
Arduino:

```
void setup()
{
    longvar = 0xAABBCCDD;
    func1();
}
```

Assembly:

```
.align 2
.comm longvar, 4
.global longvar
.text
.global func1 ;
func1:
lds r18, longvar
lds r19, longvar+1
lds r20, longvar+2
lds r21, longvar+3
...
ret
```

When working with more than 1 byte variables, the C/C++ compiler usually expect them to be aligned in memory. For example, a 2 byte variable must start at an address multiple of 2, while a 4 byte variable (such as the longvar above) must start at an address multiple of 4. The .align compiler directive, which receives as argument a power of 2 (2^2 = 4 in the example above) , ensures the alignment.

Parameter calling convention

The instructions call (two-word, jump farther) and rcall (one-word, jump shorter) cause the content of the PC register to become the address of the function being called. A call instruction also pushes the return address (the address of the next instruction after the call) on the stack. The instruction ret pops the return address off the stack and places it into the PC. The function parameters are stored in registers r25 … r8, the first byte being stored in r24. If the function has more parameters, and these registers are not enough, they are placed on the stack before the call is executed. The code inside the function must read them from the stack, and the caller code must remove them from the stack after the procedure is executed.

Function Prototype

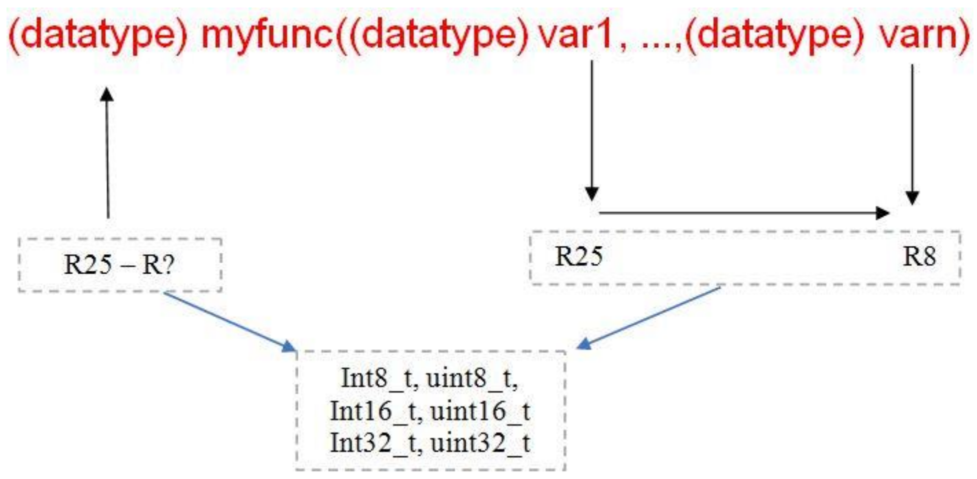


Fig.VI.3. The function prototype – register based parameter transfer and result return

To access the parameters in the stack frame (activation record), you need to copy the stack pointer SP to the Y pointer register:

```
in r28, SPL
in r29, SPH
```

If we have a function with 11 1-byte arguments, the first nine will be passed in the even-numbered registers from r24 down to r8, and the last two will be passed on the stack. If we use the Y pointer for accessing these values, we must save it first, so the beginning of our function should look like:

```
push r28
push r29
in r28, SPL
in r29, SPH
```

You can access arguments 10 and 11 by:

```
ldd r7, Y+5
ldd r7, Y+6
```

The use of registers inside functions

The 32 registers of the AVR microcontroller are given various roles and treatment by the gcc compiler. If your project contains pure asm code, you can use the registers as you like. If you combine asm and C, you have to follow the rules described in the following table:

0x00	R0	“free” register, can be changed freely without need of restore
0x01	R1	Must always holds the value of 0, do not change.
0x02	R2	These must be left unchanged by a function or saved and restored before return.
…	…	
0x0D	R13	
0x0E	R14	
0x0F	R15	
0x10	R16	
0x11	R17	
0x12	R18	R18 to R27 are freely available for use in functions. You are to expect their values to be changed in a function.
…	…	
0x1A(XL)	R26	The X pointer, freely available to use in functions, not required to be saved.
0x1B(XH)	R27	
0x1C(YL)	R28	Frame pointer, Y. Can be used inside a function, but must be saved and restored before exiting.
0x1D(YH)	R29	
0x1E(ZL)	R30	The Z pointer, freely available to use in functions, not required to be saved.
0x1F(ZH)	R31	

Fig.VI.4. The use of registers inside functions

Return values

The return value is passed in r25-r18, depending on the size of the return value (maximum return value size: 8 bytes). If the return value is 1 byte, it is placed in r24. r25 is either all 0’s (positive return value) or all 1’s (negative return value).

Declared Output	**Output location**
Byte, Boolean, int8_t, uint8_t	r24 (r25 = 00,FF)
int, uint, short, char, unsigned char int16_t, uint16_t	r25:r24
long, ulong, int32_t, uint23_t	r25:r22

Fig.VI.5. The return value types of the functions and their register use

Development tools

For developing assembly code that works with our Arduino Mega boards, we have the following options:

1. Using the Arduino IDE

   a. “Inline” Assembly – small pieces of assembly code inserted in the C++ code
   b. Assembly source files containing functions called from the .ino main file

2. Using Atmel Studio IDE

In this lab we will explore solutions 1.b and 2.

6.2. Using the Arduino IDE

a. A simple blink

Combining assembly and C code using the Arduino IDE is pretty trivial. In the first example, we’ll replicate the functionality of the first Arduino program, “Blink”, by using assembly language functions for port bits manipulation.

Open the Arduino IDE and paste the following code:

```
extern "C" void setpin();
extern "C" void turnon();
extern "C" void turnoff();

void setup() {
    setpin();
}

void loop() {
    turnon();
    delay(1000);
    turnsoff(0);
    delay(1000);
}
```

In the above code snippet, the functions setpin (which will configure pin 13 as output), turnon (which will turn the LED on), and turnoff (turn LED off) will be implemented in assembly.

To include assembly code, create a .S file by clicking in the upper right arrow in your IDE window, name it as you like, but don’t forget to set its extension to “.S”. Use CAPITAL S, not lowercase s, otherwise the compiler will not see the file.

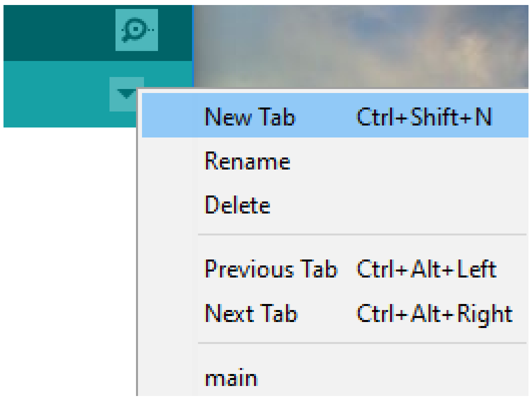


Fig.VI.6. Creation of a new file

Name this tab asm_functions.S. Paste the code snippet given below in this new file. The .S and .ino files have to be in the same folder.

```
#include "avr/io.h"

.global setpin

setpin:
    sbi _SFR_IO_ADDR(DDRB), 7 ; sets bit 7 of DDRB to 1 - output
    ret

.global turnon
turnon:
    sbi _SFR_IO_ADDR(PORTB), 7  ; sets bit 7 of PORTB to 1
    ret

.global turnoff
turnoff:
    cbi _SFR_IO_ADDR(PORTB), 7  ; sets bit 7 of PORTB to 0
    ret
```

The above code manipulates the value of bit 7 of port B, which is connected to digital pin 13 of the Arduino Mega board (see https://www.arduino.cc/en/Hacking/PinMapping2560 for other pin to port correspondences). First, the bit must be set to output by writing a ‘1’ to the corresponding position in DDRB, and then its value will be set by changing the bit in PORTB.

Warning: the Arduino compiler assumes that each port name/symbol refers to the Data Memory space address, and not to the I/O address. For example, for port B, we have from the datasheet two addresses: 0x05 in the I/O space, and 0x25 in the Memory space. In order to make the compiler use the I/O address, use the _SFR_IO_ADDR macro.

PORTB – Port B Data Register

Bit	7	6	5	4	3	2	1	0	
0x05 (0x25)	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig.VI.7. Port B addressing (source: atmega256 datasheet)

Compile the program and upload it to the board. It should perform the blinking function.

b. Using a function with parameters

We'll try to achieve the same behavior as in the first example, but instead of using two functions, one for turning the LED on, and another for turning it off, we'll use a single function and have the LED state passed as a parameter.

The c++ code will be changed to this:

```
extern "C" void setpin();
extern "C" char turnspecified(char c);

void setup() {
    setpin();
}

void loop() {
    turnspecified(1);
    delay(1000);
    turnspecified(0);
    delay(1000);
}
```

And the assembly code to this:

```
#include "avr/io.h"

.global setpin

setpin:
    sbi _SFR_IO_ADDR(DDRB), 7 ; sets bit 7 of DDRB to 1 - output
    ret

.global turnspecified
turnspecified:
    tst r24              ; r24 will hold the parameter of the
    ;function, test it for zero
    breq set0                 ; if zero, go set the pin to 0
    sbi _SFR_IO_ADDR(PORTB), 7     ; otherwise set it to 1
    rjmp finish
```

```
set0:
    cbi _SFR_IO_ADDR(PORTB), 7      ; set to zero
finish:
    ret
```

The parameters of an assembly function will be passed in registers r25 down to r8. **A 8 bit parameter will be passed in register r24**, a 16 bit parameter in registers r25:r24, and so on.

c. Using the serial interface in Assembly

This example will display a message via the Serial interface. The message will be stored in the program flash memory as a null-terminated string of characters.

The Arduino c++ code is the following:

```
extern "C" void Serial_Setup();
extern "C" void Print_Hello();

void setup() {
    Serial_Setup();
}

void loop() {
    Print_Hello();
    delay(500);
}
```

The assembly code is this:

```
#include "avr/io.h"

.global Serial_Setup
Serial_Setup:

    ; Configure the parameters of serial interface 0
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0  ; enable Rx & Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01  ; asynchronous, no
    ;parity, 1 stop, 8 bits
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

.global Print_Hello
Print_Hello:
   ; load the starting address of the string in the Z pointer
    ldi ZL, lo8(the_message)         ; r30
    ldi ZH, hi8(the_message)         ; r31
    lpm r18, Z+                ; Load the first character of the
    ;string in r18

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0        ; test the data buffer if data can be
    ;transmitted
    rjmp  Loop
    sts   UDR0, r18         ; send data contained in r18
    lpm r18, Z+             ; load the next character
    tst r18                 ; check if 0 - the string ends
    brne Loop
    ret

the_message: ; the message itself, followed by LF and CR and 0
    .ascii "Assembly is fun"
    .byte 10, 13,0
```

The first assembly function configures the parameters of the UART0 interface (the Serial interface of Arduino), and the second function sends a message stored in the program memory via this interface. Open the Serial Monitor tool to see the message being displayed.

Check the AVR ATMega2560 datasheet (http://ww1.microchip.com/downloads/en/devicedoc/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf) and the slides of Lecture 6, in order to understand the settings and operation of the UART interface.

d. Using C arrays in assembly functions

In this example we will write a function in assembly that adds the elements of an array, which is declared in the main Arduino file. The function is called form the Arduino code. Create three files: sum_array.ino, external_functions.h, arsum.S. The contents of these files are detailed below.

.ino file

```
#include "external_functions.h"

void setup() {
    compute();
    uint8_t val = result;
    Serial.begin(9600);
    Serial.println(val);
}
void loop() { }
```

external_functions.h file

```
#include <stdint.h>
extern "C" uint8_t result;
extern "C" void compute(void);
extern "C" uint8_t myarray[10]={1, 30, 3, 4, 5, 6, 7, 8, 10, 11};
```

arsum.S file

```
.file "arsum.S"
.data
.comm result, 1
.global result

.text
.global compute

compute:
    ldi r30, lo8(myarray)
    ldi r31, hi8(myarray)
    ldi r18, 0
    ldi r21, 0

looptest:
    ld r22, z+
    add r21, r22
    inc r18
    cpi r18, 10
    brlo looptest

out:
    sts result, r21
    ret
```

6.3. Using Atmel Studio

Setting up Atmel Studio

Atmel Studio 7 is the integrated development platform (IDP) for developing and debugging multiple microcontroller applications (including AVR). The Atmel Studio 7 IDP gives you an easy-to-use environment to write, build and debug your applications written in C/C++ or assembly code. It also connects together the debuggers, programmers and development kits that support AVR devices.

First of all, we have to set up the environment, in order to be able to upload the code on our Atmega2560 board.

1. Open up Atmel Studio 7
2. Go to the **Tools** menu
3. Select **External Tools**

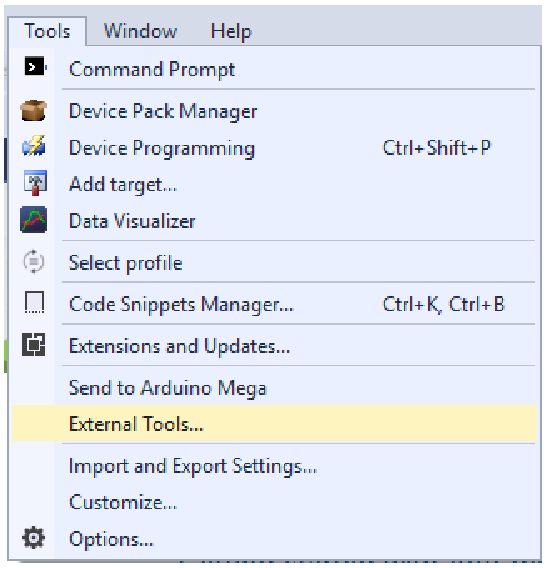


Fig.VI.8. The external tools menu

4. Click the Add button from the window

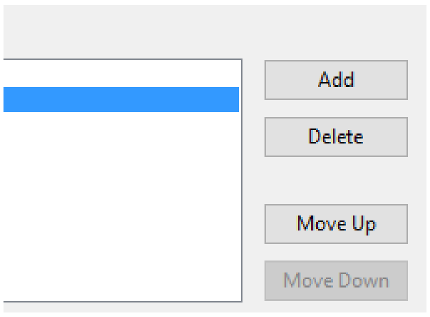


Fig.VI.9. Creation of a new external tool

5. A new entry will appear. Fill in the following data in the corresponding input text boxes.

Title: Send to Arduino Mega
Command : C:\Program Files (x86)\Arduino\hardware\tools\avr\bin\avrdude.exe
Arguments: -v -C"C:\Program Files (x86)\Arduino\hardware\tools\avr\etc\avrdude.conf" -p atmega2560 -c wiring -P COM5 -b 115200 -D -U flash:w:$(TargetDir)$(TargetName).hex:i

Please take note that this setting assumes that the Arduino tools are installed in C:\Program Files (x86). If Arduino is installed somewhere else, replace this folder with the correct path. If you have installed Arduino as a Windows 10 app, uninstall it and install it normally, otherwise Atmel Studio will not work with it.

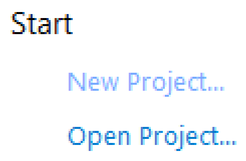
Please note that in the Arguments string the COM port is written as a constant. Please check the port of your ATMega board, and replace the above red highlighted text with the correct COM port.

Please ensure that the “Use Output Window” check box is checked, press Apply and Ok.
After completing the above steps, a menu option “Send to Arduino Mega” will appear in the Tools menu.

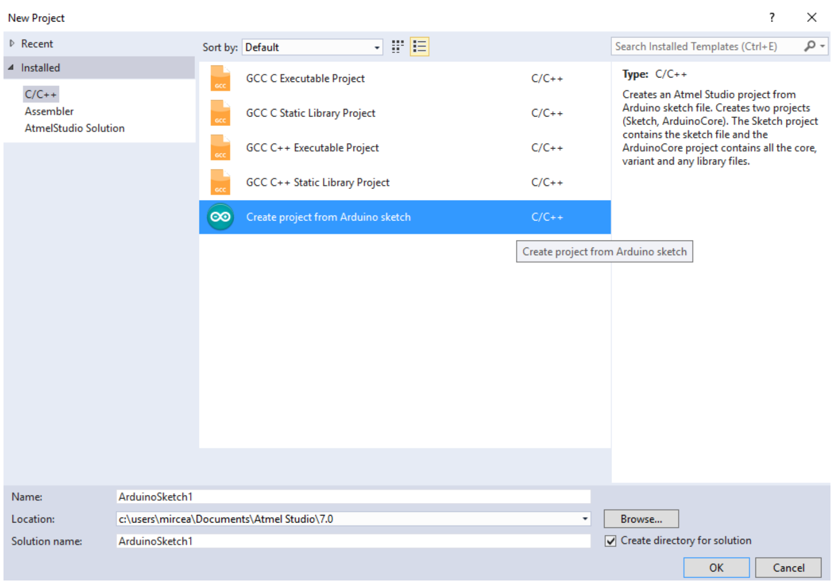
Creating a project from an Arduino sketch

Atmel Studio allows us to transform an Arduino project (sketch) into a Studio project. You have to perform the following steps:

1. In Atmel Studio click on new project.



2. Click on the Create project from Arduino sketch and fill in the desired name, location and solution name.


New Project
Recent
Installed
C/C++
Assembler
AtmelStudio Solution
Sort by: Default
Search Installed Templates (Ctrl+E)
GCC C Executable Project C/C++
GCC C Static Library Project C/C++
GCC C++ Executable Project C/C++
GCC C++ Static Library Project C/C++
Create project from Arduino sketch C/C++
Create project from Arduino sketch
Browse...
OK
Cancel

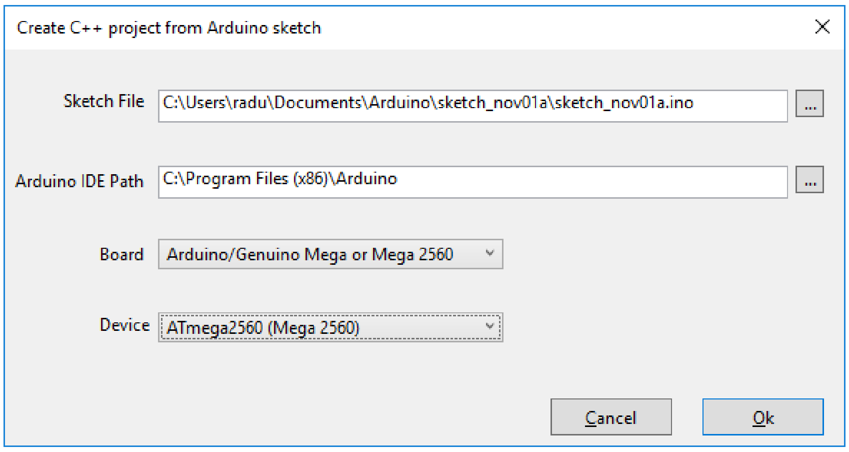
Type: C/C++

Creates an Atmel Studio project from Arduino sketch file. Creates two projects (Sketch, ArduinoCore). The Sketch project contains the sketch file and the ArduinoCore project contains all the core, variant and any library files.

Name: ArduinoSketch1
Location: c:\users\mircea\Documents\Atmel Studio\7.0
Solution name: ArduinoSketch1
☑ Create directory for solution

Fig.VI.10. Creating a project from an Arduino sketch

3. In the new window browse to the location of your Arduino sketch, select the path of your IDE and the development board used and press Ok.

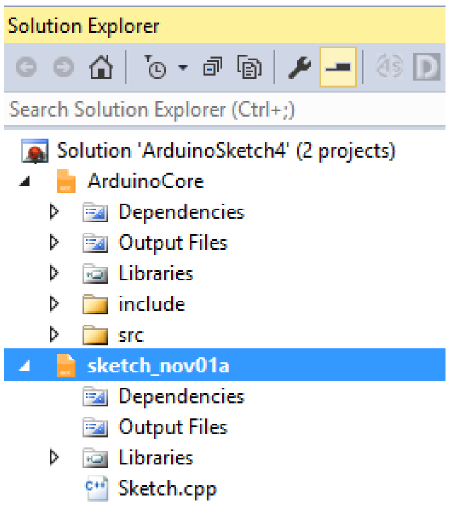


Sketch File: C:\Users\radu\Documents\Arduino\sketch_nov01a\sketch_nov01a.ino
Arduino IDE Path: C:\Program Files (x86)\Arduino
Board: Arduino/Genuino Mega or Mega 2560
Device: ATmega2560 (Mega 2560)

Fig.VI.11. Configuring the new project.

In this example, we will use the second blink program written in assembly and c, which uses an asm function to set the state of the LED pin 13.

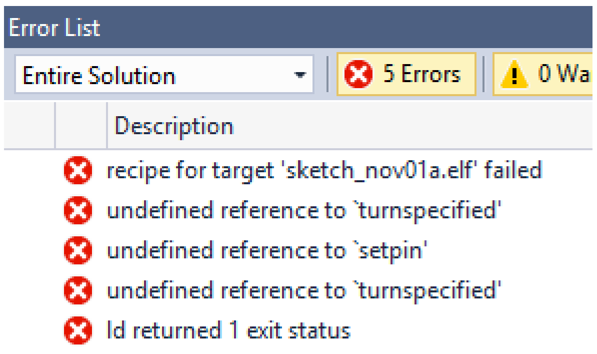
After importing the Arduino sketch the solution tree looks like in the image below:



```
Solution 'ArduinoSketch4' (2 projects)
  ArduinoCore
    Dependencies
    Output Files
    Libraries
    include
    src
  sketch_nov01a
    Dependencies
    Output Files
    Libraries
    Sketch.cpp
```

Fig.VI.12. The new project's solution tree

If we will compile the program we will see that we get the following errors.



```
recipe for target 'sketch_nov01a.elf' failed
undefined reference to `turnspecified'
undefined reference to `setpin'
undefined reference to `turnspecified'
ld returned 1 exit status
```

Fig.VI.13. Build errors

This happens due to the fact that Atmel Studio does not add automatically the references to any external library. To solve this issue right click on the project name and select Add Existing Item from the menu.

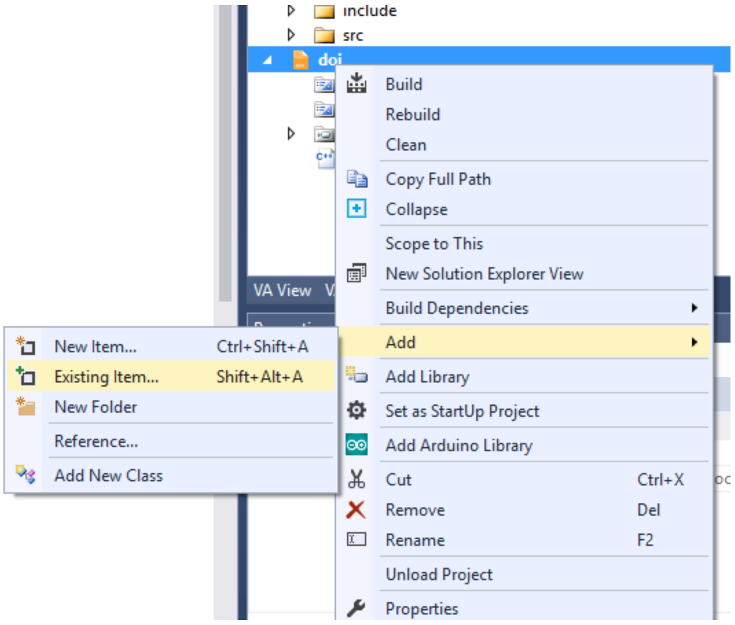


Fig.VI.14. Adding the missing dependencies

Browse to the location of the assembly file asm_functions.S in the Arduino folder where you have created them, and add the file to the solution.

Rebuild the solution and observe that the build succeeds this time.

In order to upload the program to the board click on the Tools menu and select Send to Arduino Mega. If you have implemented the above steps correctly the program should upload on the Arduino mega board without any issue. You will see the following message in the output window in case the program is uploaded successfully.

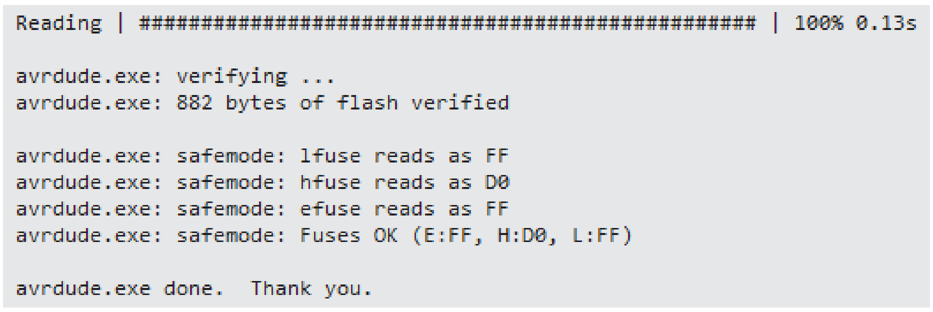

```
Reading | ################################################## | 100% 0.13s

avrdude.exe: verifying ...
avrdude.exe: 882 bytes of flash verified

avrdude.exe: safemode: lfuse reads as FF
avrdude.exe: safemode: hfuse reads as D0
avrdude.exe: safemode: efuse reads as FF
avrdude.exe: safemode: Fuses OK (E:FF, H:D0, L:FF)

avrdude.exe done.  Thank you.
```

Fig.VI.15. Build success

Debugging using Atmel Studio

A powerful feature of Atmel Studio is the simulation-based debugger. This debugging mechanism allows us to analyze the program behavior, monitor the registers, ports and memory, even without having a development board near us.

To set up the debugging environment first select the **Debug** option from the roll down menu and the **ATmega2560** board from the main menu strip.

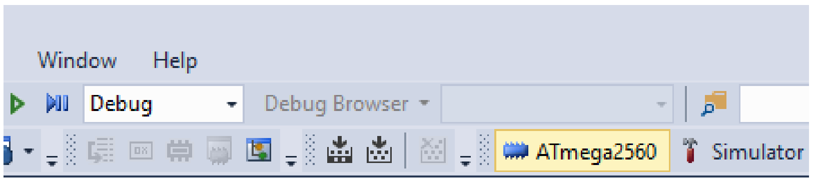
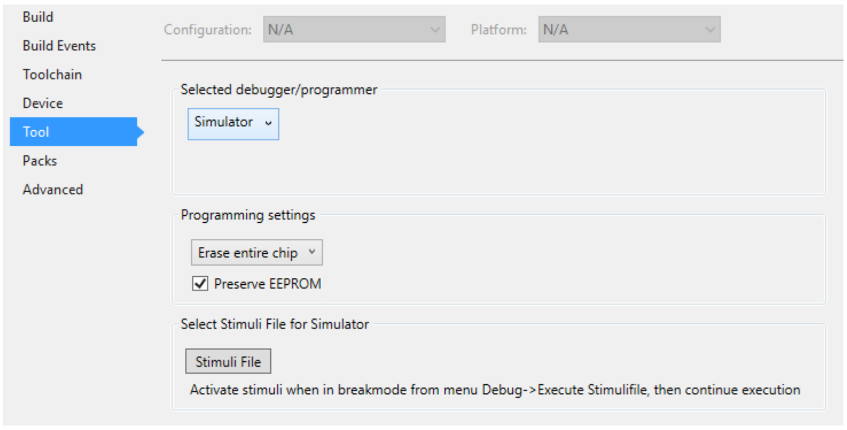


Fig.VI.16. Setting up the debugging target device

After pressing the ATmega2560 option, a new window will appear. Select the **Tool** option and from the **Select debugger / programmer** option select the **Simulator** option.


Build
Build Events
Toolchain
Device
Tool
Packs
Advanced

Configuration: N/A
Platform: N/A

Selected debugger/programmer
Simulator

Programming settings
Erase entire chip
☑ Preserve EEPROM

Select Stimuli File for Simulator
Stimuli File
Activate stimuli when in breakmode from menu Debug->Execute Stimulifile, then continue execution

Fig.VI.17. Setting up debugging options

After setting up the specified options, come back to the main program by clicking the program sketch tab. We can use the debugger to analyze the behavior of the blinking program that we have already imported from Arduino. For debugging, it is recommended that you comment out the delays in the code, as they will take a very long time in simulation.

You can set breakpoints by clicking on the left gray part near the line where you want to include the breakpoint or by positioning on a line and from the **Debug** window select the **Toggle Breakpoint** option (or by pressing F9). You can set breakpoints in the c++ file, or in the assembly file.

To start debugging press the button near the Debug drop down menu or press the Alt+F5 button combination then press F5 once. To view the available AVR input/output registers (ports), from the **Debug** menu select **Windows** and then **I/O.** From the same Debug/Windows menu you can select to view other information, such as Registers (content of the 32 registers), Memory (content of the program flash, data memories, EEPROM), Disassembly, and so on.

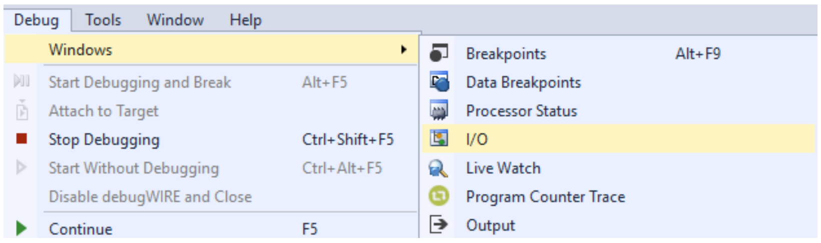
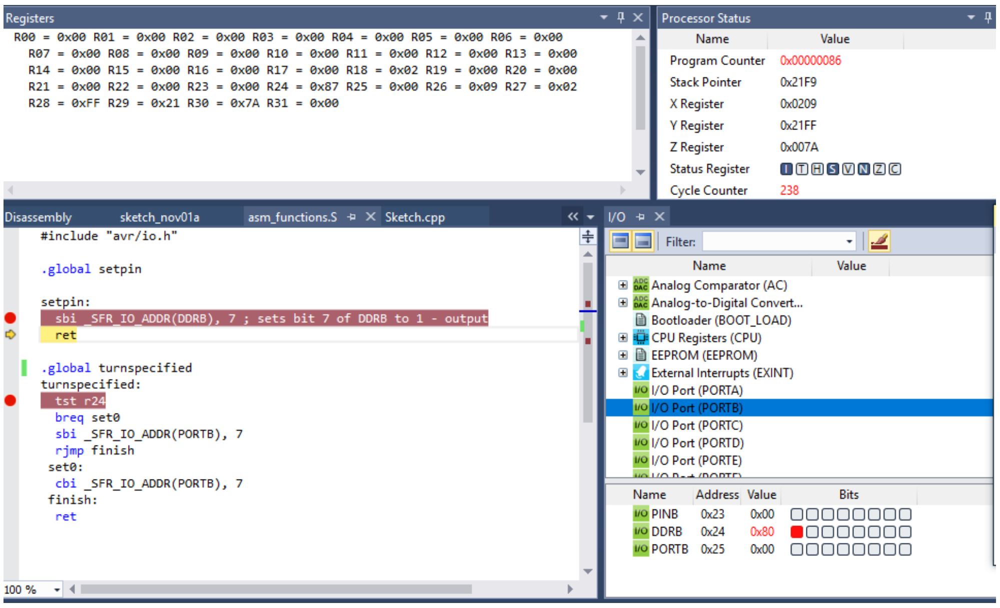


Fig.VI.18. Selecting the I/O information window

From the I/O window, select PORTB. You will see, at each step of the program, the contents of the associated registers, DDRB, PORTB, and the input PINB.



```
Registers
R00 = 0x00 R01 = 0x00 R02 = 0x00 R03 = 0x00 R04 = 0x00 R05 = 0x00 R06 = 0x00
  R07 = 0x00 R08 = 0x00 R09 = 0x00 R10 = 0x00 R11 = 0x00 R12 = 0x00 R13 = 0x00
  R14 = 0x00 R15 = 0x00 R16 = 0x00 R17 = 0x00 R18 = 0x02 R19 = 0x00 R20 = 0x00
  R21 = 0x00 R22 = 0x00 R23 = 0x00 R24 = 0x87 R25 = 0x00 R26 = 0x09 R27 = 0x02
  R28 = 0xFF R29 = 0x21 R30 = 0x7A R31 = 0x00
```

Processor Status

Name	Value
Program Counter	0x00000086
Stack Pointer	0x21F9
X Register	0x0209
Y Register	0x21FF
Z Register	0x007A
Status Register	I T H S V N Z C
Cycle Counter	238

```
#include "avr/io.h"

.global setpin

setpin:
  sbi _SFR_IO_ADDR(DDRB), 7 ; sets bit 7 of DDRB to 1 - output
  ret

.global turnspecified
turnspecified:
  tst r24
  breq set0
  sbi _SFR_IO_ADDR(PORTB), 7
  rjmp finish
 set0:
  cbi _SFR_IO_ADDR(PORTB), 7
 finish:
  ret
```

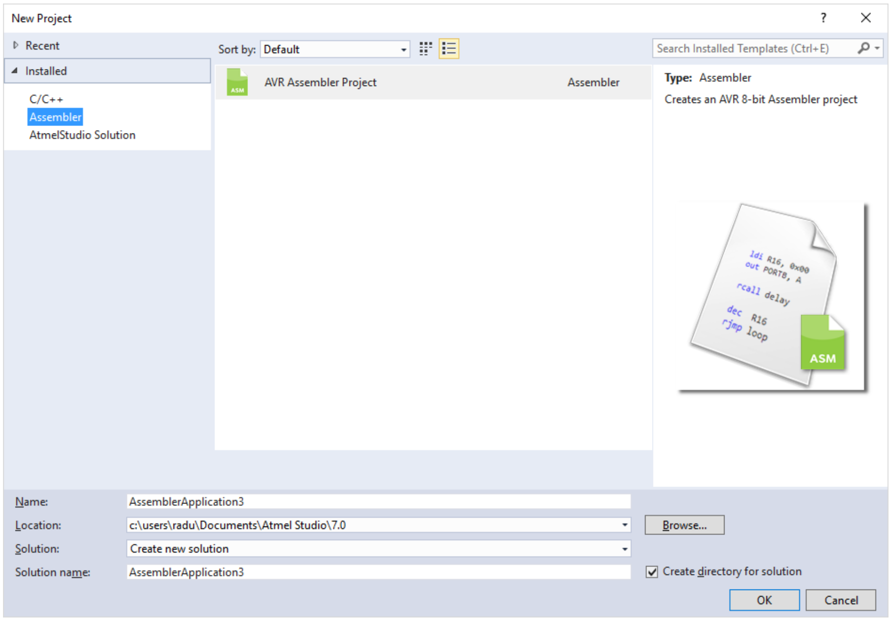
Name	Value
Analog Comparator (AC)	
Analog-to-Digital Convert...	
Bootloader (BOOT_LOAD)	
CPU Registers (CPU)	
EEPROM (EEPROM)	
External Interrupts (EXINT)	
I/O Port (PORTA)	
I/O Port (PORTB)	
I/O Port (PORTC)	
I/O Port (PORTD)	
I/O Port (PORTE)	

Name	Address	Value	Bits
PINB	0x23	0x00	☐☐☐☐☐☐☐☐
DDRB	0x24	0x80	🟥☐☐☐☐☐☐☐
PORTB	0x25	0x00	☐☐☐☐☐☐☐☐

Fig.VI.19. Displaying the state of PortB in the I/O window

Working with assembly-only projects

Atmel Studio allows you to write assembly-only solutions. From the **File** menu, select **New Project**. When the New Project window opens, select from the left panel the "Assembler" option, as shown in the figure below. Name your project, and click OK.



AVR Assembler Project   Assembler

Type: Assembler

Creates an AVR 8-bit Assembler project

Name:	AssemblerApplication3
Location:	c:\users\radu\Documents\Atmel Studio\7.0
Solution:	Create new solution
Solution name:	AssemblerApplication3

[✓] Create directory for solution

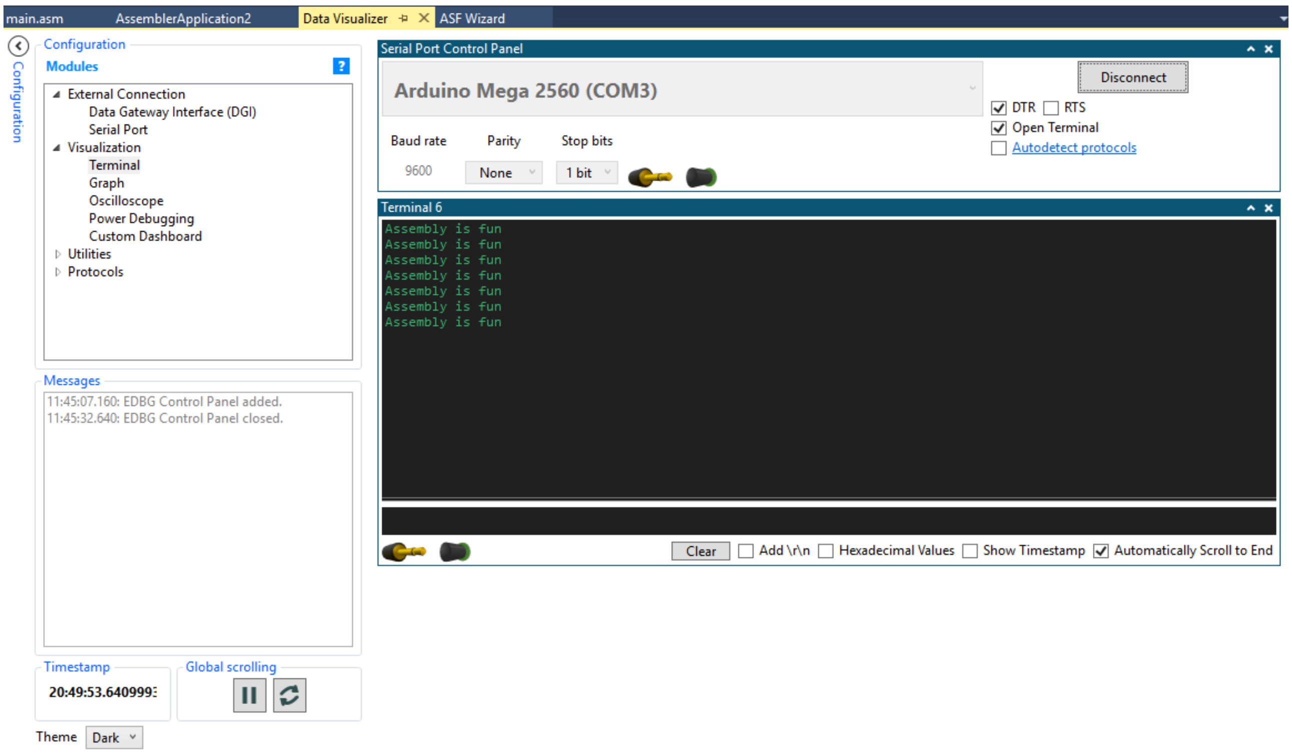
Fig.VI.20. Creating a new project using assembly language only

The environment will generate a main.asm file, with a dummy asm code. Replace this code with the one below, which will send to the serial interface the message “Assembly is fun”:

```
; Main program
main:
    rcall asm_setup

main_loop:
    rcall asm_loop
    rjmp main_loop


asm_setup:
; Init the serial interface
    clr r0
    sts   UCSR0A, r0
    ldi   r24, 1<<RXEN0 | 1 << TXEN0     ; enable Rx & Tx
    sts   UCSR0B, r24
    ldi   r24, 1 << UCSZ00 | 1 << UCSZ01 ; asynchronous, no
    ;parity, 1 stop, 8 bits
    sts   UBRR0H, r0
    ldi   r24, 103
    sts   UBRR0L, r24
    ret

asm_loop:
    ; print and wait
    rcall Print_Hello
    rcall wait
    ret


Print_Hello:

    ; loading address and size of array
    ldi ZL, LOW(2*array)          ; r30
    ldi ZH, HIGH(2*array)         ; r31
    lpm r16, Z+                   ; Load the character pointed by Z
    ;registers (r30/r31)

Loop:
    lds   r17, UCSR0A
    sbrs  r17, UDRE0           ; test the data buffer if data can
    ;be transmitted
    rjmp  Loop
    sts   UDR0, r16            ; send data contained in r16
    lpm r16, Z+                ; point to the next character
    tst r16               ; check for string end - 0
    brne Loop
    ret
```

```
; simple function to wait for aprox 1 second by idle counting
wait:

    ldi  R17, 0x53
    LOOP0:  ldi  R18, 0xFB
    LOOP1:  ldi  R19, 0xFF
    LOOP2:  dec  R19
    brne LOOP2
    dec  R18
    brne LOOP1
    dec  R17
    brne LOOP0
    ret

; string to be written, stored in the program memory
array:
.db "Assembly is fun",13,10,0
```

Build your solution using the Build menu, and send it to the Arduino Mega board. For seeing the serial output, you can use either the serial monitor of Arduino, or you can use the terminal of Atmel Studio. For this, from the Tools menu, select Data Visualizer. From the left panel of the tool, select Visualization/Terminal, and from the central panel select the serial port of your board and click Connect. The terminal should open and display your message, as shown in the following figure.



Serial Port Control Panel: Arduino Mega 2560 (COM3)

Baud rate 9600 Parity None Stop bits 1 bit

☑ DTR ☐ RTS ☑ Open Terminal

```
Assembly is fun
Assembly is fun
Assembly is fun
Assembly is fun
Assembly is fun
Assembly is fun
Assembly is fun
```

☐ Add \r\n ☐ Hexadecimal Values ☐ Show Timestamp ☑ Automatically Scroll to End

```
11:45:07.160: EDBG Control Panel added.
11:45:32.640: EDBG Control Panel closed.
```

Timestamp 20:49:53.640999

Theme Dark

Fig.VI.21. The Atmel Studio terminal

You can also use the debugger to analyze the step by step execution of assembly programs. The steps to be performed are the same as in the case of c/c++ projects.

6.4. Individual work:

1. Implement the examples provided in this lab. Use the debugger as often as possible, to see the behavior of the program. Can you see the message string in the program flash memory?
2. Using the datasheet and the information contained in Lecture 6, write a document explaining the settings and the operation of the serial interface as shown in the third example.
3. Write an assembly function that will return a value (the return values of functions start with registers r25:r24). Write the .ino program that will call this function and use its output. *Hint: you can read a port.*
4. Modify the example that displays the “Arduino is fun” message to display any string (char array, null terminated) declared in the c++ program. *Hint: the string is stored in the data memory*.
5. Analyze the assembly code for serial communication of the Arduino project, and compare it to the assembly code of the pure assembly project. Describe the differences and similarities.

References:

1. https://docslide.us/documents/lecture-12-5600350816ac8.html
2. https://forum.arduino.cc/index.php?topic=490065.0
3. https://www.youtube.com/watch?v=8yAOTUY9t10

VII. Laboratory 7 – Analog signals processing

An analog signals is variable voltage over time and is usually the output of a sensor that monitors the environment. Such a signal can be processed and interpreted by a microcontroller using an analog to digital converter (ADC), which is a device that converts a voltage into a digital number that can be “understood” by the microcontroller.

The pins of the Arduino boards that have ADC capabilities have the prefix “A”. In Figure VII.1 (Arduino Mega), the pins are highlighted with red color, while in Figure VII.2 they have the group name “ANALOG IN”.

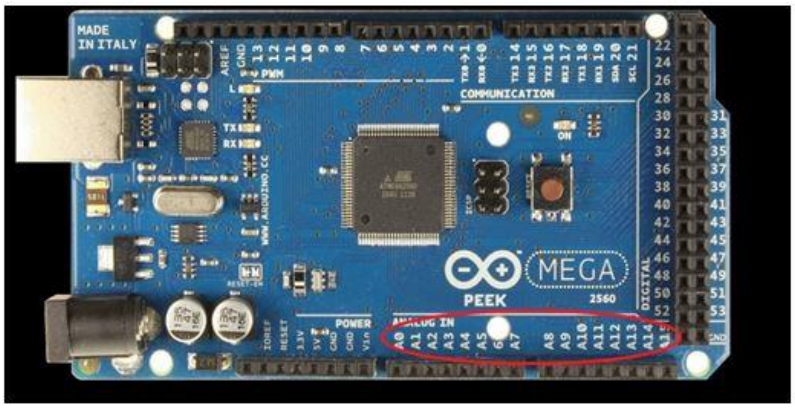


Fig.VII.1. Analog pins of Arduino Mega. (source: https://store.arduino.cc/arduino-mega-2560-rev3)

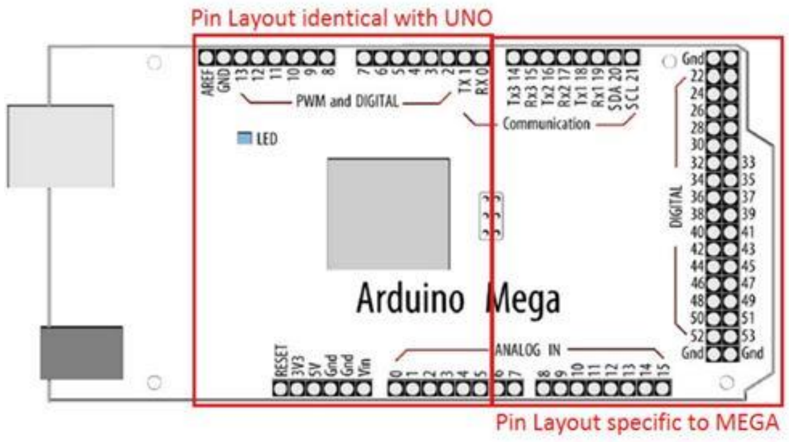


Fig.VII.2. Arduino UNO and Mega pin layout

ADCs can have different specifications among different microcontrollers. For example the Mega has 10 bits resolution ADCs (can provide a digital value between 0 .. 2^{10}-1 / 0 .. 1023). There are also ADCs with 8 bit or 16 bits resolution. The digital value provided by the ADC (ADC_reading) can be computed with the formula below:

$$\frac{\mathbf{Vref}}{\mathbf{1023}} = \frac{\mathbf{ADC\_reading}}{\mathbf{Voltage\_measured}} \qquad (2)$$

The analog pins of the Arduino Board can be also used as GPIO pins (as any digital pin, having also Pull-Up resistors).

To read the value provided by the ADC (ADC_reading) use the **analogRead(pin_no)** function.
It takes about 100 microseconds (0.0001 s) to read an analog input (max reading rate is therefore 10000 values / second).

If you must switch between readings from several analog inputs, use a short delay between the switching (otherwise the reading from the next input is noisy).

Note: the function does not provide correct values if the pin is programmed as output or if the Pull-Up logic is activated!!!

The value of the reference voltage Vref (2) can be configured using the function **analogReference(type)** (i.e. the value used as the top of the input RANGE).

Type - reference to use:

- **DEFAULT**: the default analog reference of **5 volts** (for UNO & MEGA)
- INTERNAL: a built-in reference, equal to 1.1 volts on UNO (not available on the Arduino Mega)
- INTERNAL1V1: a built-in 1.1V reference (Arduino Mega only)
- INTERNAL2V56: a built-in 2.56V reference (Arduino Mega only)
- EXTERNAL: the voltage applied to the AREF pin (**0 to 5V only**) is used as the reference.

Notes:
1. After changing the analog reference, the first few readings from analogRead() may not be accurate !!!
2. Don't use anything less than 0V or more than 5V for external reference voltage on the AREF pin! If you're using an external reference on the AREF pin, you must set the analog reference to EXTERNAL before calling analogRead(). Otherwise, you will short together the active reference voltage (internally generated) and the AREF pin, possibly damaging the microcontroller on your Arduino board !!!

Example 1: reading a potentiometer

In the following example, the value provided by a linear potentiometer will be read and displayed on the LCD. The circuit is illustrated in Figure VII.3 (connect VCC and GND pins of the potentiometer to +5V and GND pins of the board and the output signal/pin of the potentiometer to an analog pin of the board – A1).

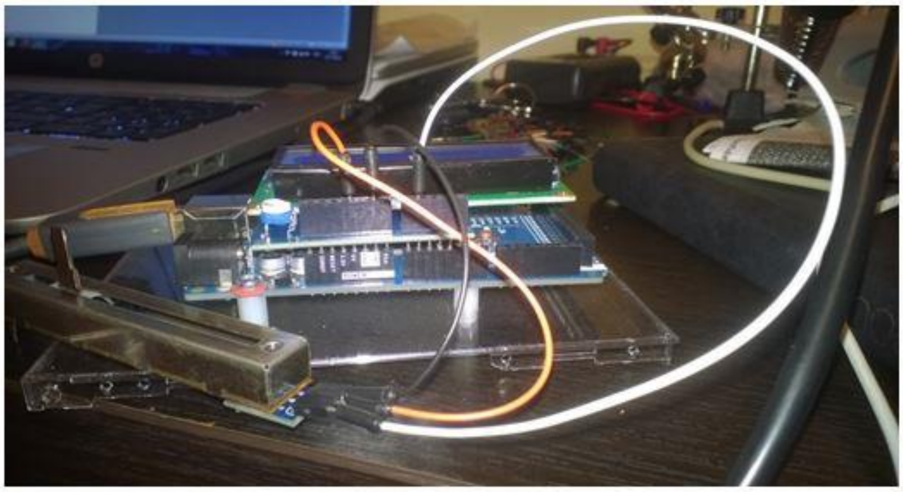


Fig.VII.3. Potentiometer interfacing example

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
void setup()
{
    analogReference(DEFAULT); //set the reference voltage to the
    //default value (optional)
    lcd.begin(16, 2); //init. LCD
    lcd.setCursor(0,0);
    lcd.print("Read sensor");
    pinMode(A1, INPUT); // Set pin A1 as input (make sure that it
    //was not set as output)
}
void loop()
{
    int val = analogRead(A1); //reading the analog value
    lcd.setCursor(0,1);
    lcd.print(val);
}
```

Note: If you cannot use the linear potentiometer (not enough potentiometers available), you can use the **Learning Shield**. This shield has a rotary potentiometer connected to analog input **A0**. When using the Learning Shield, you cannot use the LCD shield, but you can display the number on the shield's SSD (see Lab 2), or you can use the Serial interface.

Example 2: Reading the temperature

The used temperature sensor is LM50 http://www.ti.com/lit/ds/symlink/lm50.pdf

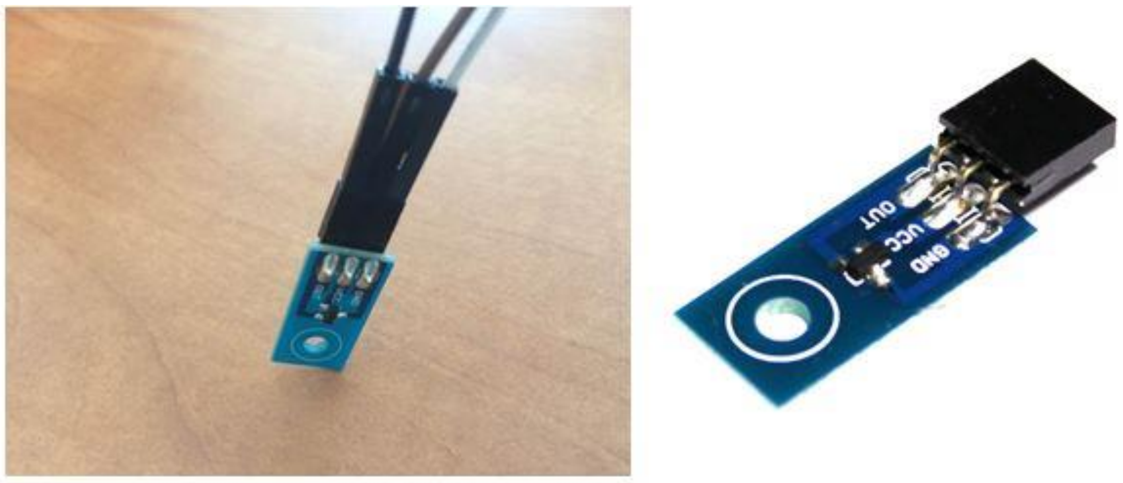


Fig.VII.4. The temperature sensor

Sensor specifications:

- Linear output: +10.0 mV/°C = 0.01V/°C
- Temperature range: −40°C ... +125°C
- +500 mV offset for negative temperatures reading

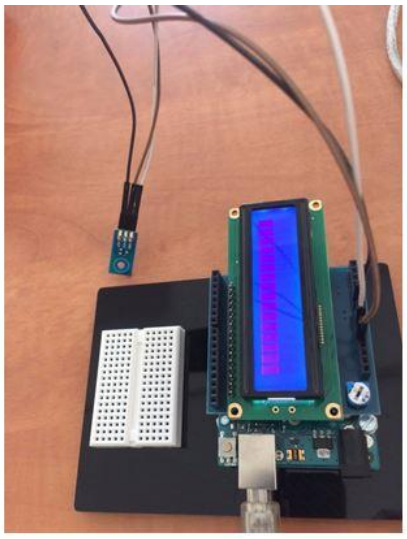


Fig.VII.5. Design layout for example 2

Example 2 – read temperature, performs an average of 10 consecutive readings and sends the result to the PC over the serial connection. Connect the sensor (sensor output connected to A0 pin) and test the code below:

```
float resolutionADC = .0049 ; // default ADC resolution for the 5V
//reference = 0.049 [V] / unit
float resolutionSensor = .01 ; // sensor resolution  = 0.01V/°C
```

```
void setup() {
    Serial.begin(9600);
}
void loop(){
    Serial.print("Temp [C]: ");
    float temp = readTempInCelsius(10, 0); // read temp. 10 times
    //and returns the average
    Serial.println(temp); // display the result
    delay(200);
}
float readTempInCelsius(int count, int pin) {
    // read temp. count times from the analog pin
    float sumTemp = 0;
    for (int i =0; i < count; i++) {
        int reading = analogRead(pin);
        float voltage = reading * resolutionADC;
        // subtract the DC offset and converts the value in
        //degrees (C)
        float tempCelsius = (voltage - 0.5) / resolutionSensor ;
        sumTemp = sumTemp + tempCelsius; // accumulates the
        //readings
    }
    return sumTemp / (float)count; // return the average value
}
```

Using the AVR’s ADC registers

The ATmega 2560 MCU contains a single 10 bits ADC with a maximum sampling frequency of 15kS/s at the maximum resolution. The sampling method used is the successive approximation.

The MCU offers the possibility to select among 16 input analog pins. A pin is selected through a multiplexing process. Also, differential input voltages can be used on 4 independent differential channels. A simplified schematic of the ADC is shown in Figure VII.6 :

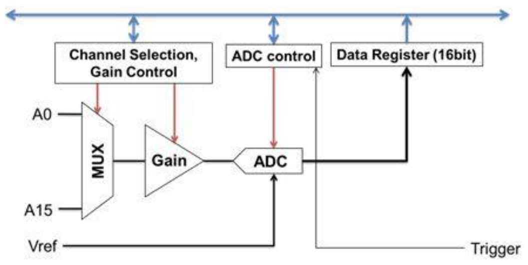


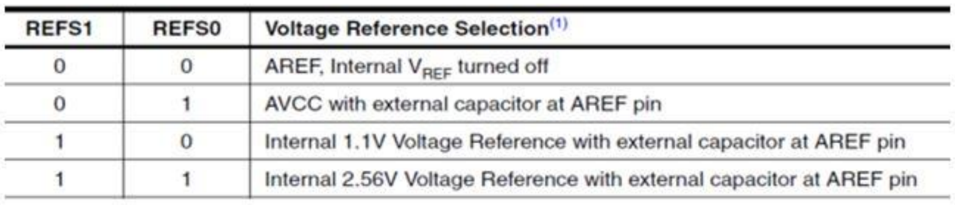
Fig.VII.6. Simplified view of the ADC. (source: https://bennthomsen.wordpress.com/arduino/peripherals/analogue-input/)

The ADC can be used in the following 3 modes:

1. Single conversion: started by writing a logic 1 to the start conversion bit
2. Triggered conversion: the conversion starts on the rising edge of the trigger signal

3. Free running: next conversion starts after the current one is finished

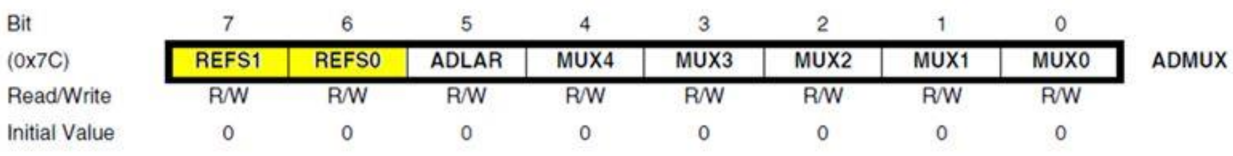
Before using the ADC, the reference voltage and the clock freq. should be set. The options for the reference voltage are shown in Figure VII.7 (they are set by bits 6 and 7 from the ADMUX register, as shown in Figure VII.8):



REFS1	REFS0	Voltage Reference Selection[1]
0	0	AREF, Internal V_{REF} turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Internal 1.1V Voltage Reference with external capacitor at AREF pin
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Fig.VII.7. Reference voltage settings (source: atmega2560 datasheet)

To set the reference voltage to Vcc one must write: ADMUX = (1<<REFS0)

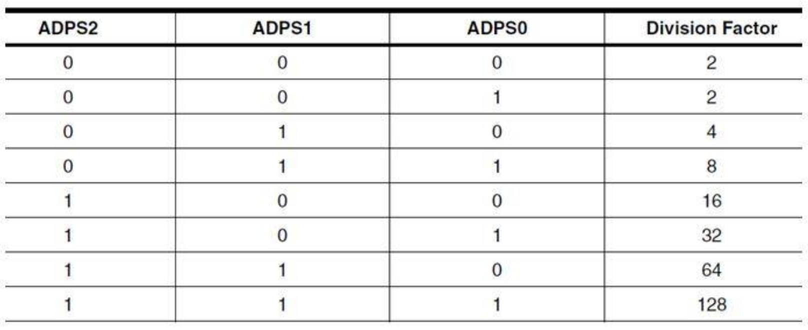


Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig.VII.8. ADMUX register. (source: atmega 2560 datasheet)

- **Bit 5 – ADLAR – ADC Left Adjust Result** – set to one to adjust left the result
- **Bits 4:0 – MUX4:0 – Analog Channel and Gain Selection Bits** – input channel select

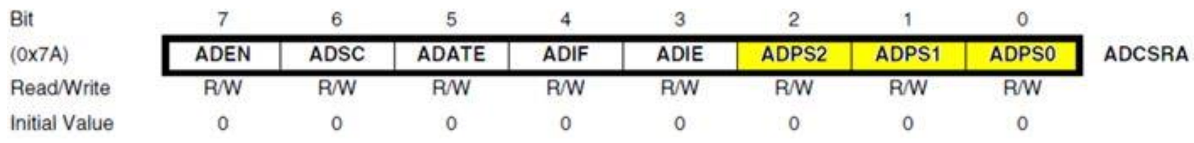
The prescaler controls the input clock frequency of the ADC (usually between 50 and 200 kHz). The conversion process requires 13-14 clock cycles. The prescaler options are shown in Figure VII.9:



ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Fig.VII.9. Prescaler options. (source: atmega 2560 datasheet)

These bits can be set through the ADC Control and Status Register (ADCSRA) – see Fig. VII.10:



Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Fig.VII.10. ADCSRA Register. (source: atmega2560 datasheet)

Meaning of the ADCSRA bits:

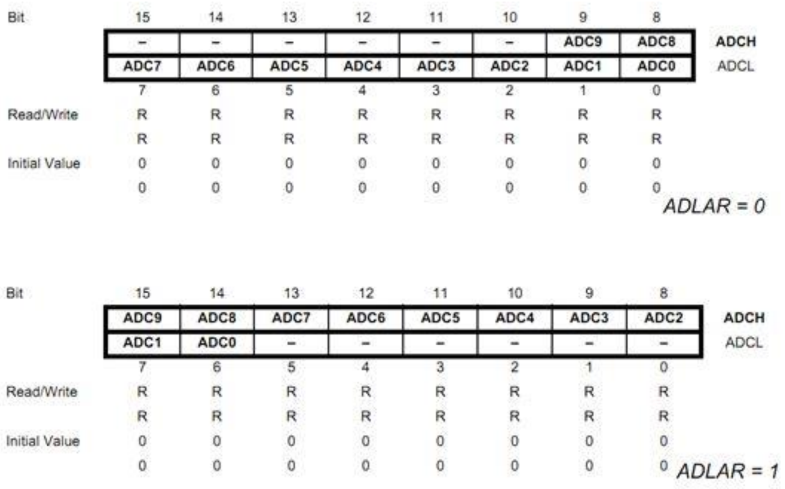
- **Bit 7 – ADEN – ADC Enable** – Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off (pins function become GPIO). Turning the ADC off while a conversion is in progress, this will terminate the conversion.
- **Bit 6 – ADSC – ADC Start Conversion** – Write this bit to one starts the first conversion. The first conversion will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC. ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.
- **Bit 5 – ADATE – ADC Auto Trigger Enable** – When this bit is written to one, Auto Triggering of the ADC is enabled. The ADC will start a conversion on a positive edge of the selected trigger signal. The trigger source is selected by setting the ADC Trigger Select bits, ADTS in ADCSRB.
- **Bit 4 – ADIF – ADC Interrupt Flag** – This bit is set when an ADC conversion completes and the Data Registers are updated. The ADC Conversion Complete Interrupt is executed if the ADIE bit and the I-bit in SREG are set. ADIF is cleared by hardware when executing the corresponding interrupt handling vector. Alternatively, ADIF is cleared by writing a logical one to the flag.
- **Bit 3 – ADIE – ADC Interrupt Enable** – When this bit is written to one and the I-bit in SREG is set, the ADC Conversion Complete Interrupt is activated.
- **Bits 2:0 – ADPS2:0 – ADC Prescaler Select Bits** – These bits determine the division factor between the MCU frequency and the input clock to the ADC (Table 2).

ADCL and ADCH – ADC Data Registers

The conversion result is stored in these registers (Fig. VII.11). If ADLAR is set, the result is left adjusted. If ADLAR is cleared (default), the result is right adjusted.

When ADCL is read, the ADC Data Register is not updated until ADCH is read. If the result is left adjusted and no more than 8-bit precision is required, it is sufficient to read ADCH. Otherwise, ADCL must be read first, then ADCH.

The 2 registers combined can be referred by the ADCW name:



Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Fig.VII.11. ADCL, ADCH registers contents depending on the ADLAR bit. (source: atmega 2560 datasheet)

```
#include <LiquidCrystal.h>
LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
void setup()
{
    // set the ADC clock to 16MHz/128 = 125kHz
    ADCSRA |= ((1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0));
    ADMUX |= (1<<REFS0);        //Set the ref. voltage to Vcc (5v)
    ADCSRA |= (1<<ADEN);        //Activate the ADC
    ADCSRA |= (1<<ADSC);
}

void loop()
{
    int val = read_adc(0); //read the value
    lcd.setCursor(0,1);
    lcd.print(val);
}


uint16_t read_adc(uint8_t channel)
{
    ADMUX &= 0xE0;           // delete MUX0-4 bits
    ADMUX |= channel&0x07;   //Sets in MUX0-2  the value of the
    //new channel to be read
    ADCSRB = channel&(1<<3); // Set MUX5 value
    ADCSRA |= (1<<ADSC);      // start conversion
    while(ADCSRA & (1<<ADSC));  //Wait for the conversion to
    //finish

     return ADCW;
}
```

7.1. Individual work

1. Test the examples from the laboratory. Ask your TA if you have any problems in connecting the wires.
2. Using a **light sensor**, implement a night light functionality: adjust the brightness of the LED (using PWM), so that it increases when the ambient light decreases.
3. Using the micros() function, compare the conversion speed of the analogRead(0 function and read_adc() given in example 3. Change prescaler values, using the ADPS bits, and observe the effect on the measured time.
4. Measure the temperature using the analog sensor and the digital sensor from the previous Lab. Using the digital sensor as a reference, make an automatic calibration procedure (compute the offset between sensors), which will run continuously if a button is pressed. When the button is released, the system will use the computed offset to correct the analog value, and display it side by side with the digital one.

VIII. Laboratory 8 - Arduino and WiFi - IoT applications

IoT - Internet of Things is a recent trend that refers to connecting smart appliances and electronics such as microcontrollers and sensors to the internet. In this laboratory we will be using an Arduino and an ESP8266 WiFi module. We will control the built-in LED from Arduino directly from a webpage that is hosted on the ESP8266 module. We will also be able to display data from Arduino on the webpage.
ESP 8266 is a standalone WiFi module that can be programmed using the Arduino IDE. It can be programmed via a FTDI programmer or by using an Arduino board with the ATMega328 chip removed. **In this laboratory we will be configuring the ESP8266 board using the serial communication, by issuing AT commands.**

Connecting the WiFi module

We will connect to the ESP board to Arduino by using the UART interface Serial1. Connect the RX pin of the ESP8266 to **pin 18 (TX1)** on Arduino and the TX pin from ESP8266 to **pin 19 (RX1)** on Arduino. Connect the GND pin to Arduino's GND and the VCC/3V3 pin to the Arduino **3.3V pin**. You must also connect the **EN pin** of the ESP8266 to the **3.3V power source**.

Consult the schematic below for connecting the WiFi module to Arduino:

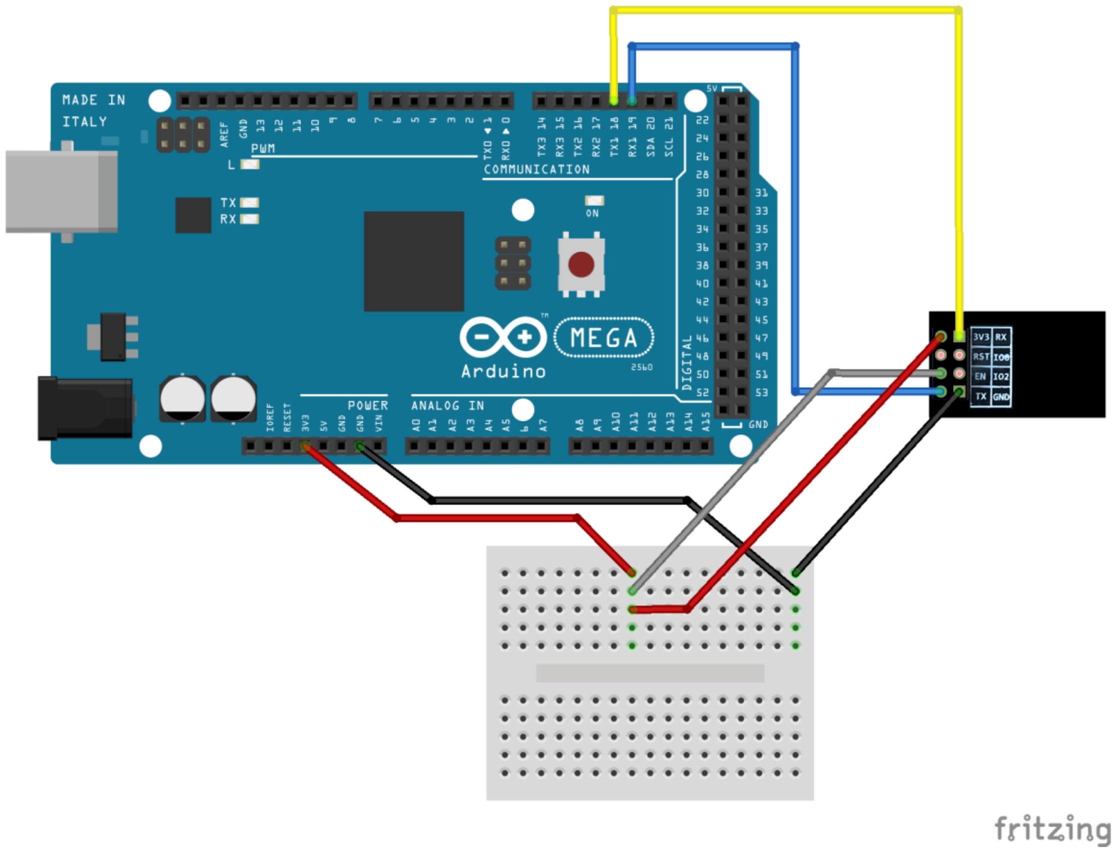


Fig.VIII.1. Connecting the wifi module to Arduino via breadboard

The Arduino program must issue AT commands to reset the WiFi module (*"AT+RST"*). The next step is to configure it as an access point (*"AT+CWMODE=2"*). After that, we get the IP address: 192.168.4.1 using the command: *"AT+CIFSR"* which also prints the MAC address of the module. Then we query to get the SSID info (*"AT+CWSAP?"*): the actual **WiFi network name** and password (default is not set) and then we configure for multiple connections (*"AT+CIPMUX=1"*) and turn on the web server on port 80 (*"AT+CIPSERVER=1,80"*).
Each AT command must end with carriage return and newline (*"\r\n"*).

More information regarding the ESP8266 AT commands can be found here: https://www.espressif.com/sites/default/files/documentation/4a-esp8266_at_instruction_set_en.pdf and here: https://github.com/espressif/esp8266_at/wiki.

The commands are sent via the serial interface that was setup from Arduino ("**Serial1**"), by using the "*print*" method. The module's response to an AT command is read and saved in a string, and then displayed in the serial monitor of the connection between the PC and Arduino ("**Serial**"). Please consult the "**sendData()"** method in the example below.

In the loop we check if data is available on the Serial1 interface and check if this data is data from the network (it will include the *"+IPD"* substring). We first read the connection id as it is required when sending data using the command: *"AT+CIPSEND"*. A simple webpage is constructed as a string and sent to the ESP8266 module. The webpage includes some text to display, two buttons for user input, and below the buttons is another text for displaying data from Arduino. After the AT command for the webpage is sent, we must close the connection by using: *"AT+CIPCLOSE"*.

The mechanism used for controlling the led on Arduino is built using the buttons on the webpage and the URLs they point to. The first button points to "*/l0*" and the second one to "*/l1*". By clicking them, the webpage tries to redirect to those addresses and it issues a request on the webserver. On Arduino, we will receive the request response by reading the ESP8266 response (in "*sendData()*" method) and we check if it contains the "*/l0*" or "*/l1*" substrings in the response string (ex: "*response.indexOf("/l0") != -1*").

Data from Arduino is displayed on the web page by adding the result from "*readSensor()*" method to the webpage string and sending it using AT commands to the WiFi module. In the example below the read sensor method will actually display the result of "*millis()*".

ESP 8266 example code:

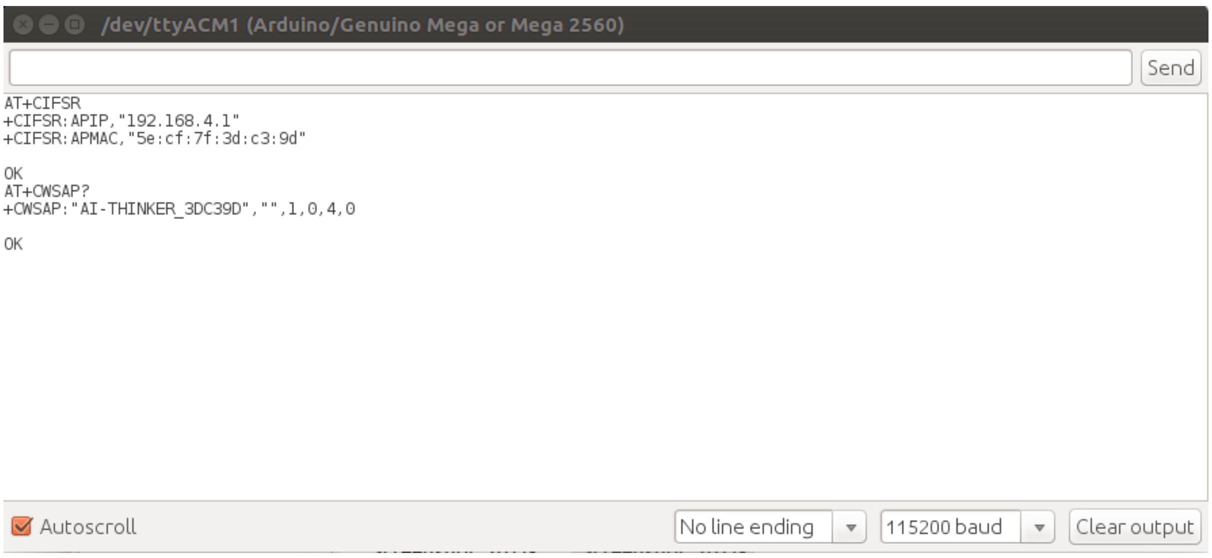
```
#define DEBUG true

void setup() {
    Serial.begin(115200);
    Serial1.begin(115200);
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
    sendData("AT+RST\r\n", 2000, false); // reset module
    sendData("AT+CWMODE=2\r\n", 1000, false); // configure as
    //access point
    sendData("AT+CIFSR\r\n", 1000, DEBUG); // get ip address
    sendData("AT+CWSAP?\r\n", 2000, DEBUG); // get SSID info
    //(network name)
    sendData("AT+CIPMUX=1\r\n", 1000, false); // configure for
    //multiple connections
    sendData("AT+CIPSERVER=1,80\r\n", 1000, false); // turn on
    //server on port 80
}

void loop() {
    if (Serial1.available()) {
        if (Serial1.find("+IPD,")) {
            delay(500);
            int connectionId = Serial1.read() - 48; // read()
            //function returns
            // ASCII decimal value and 0 (the first decimal
            number) starts at 48
            String webpage = "<h1>Hello World!</h1><a
            href=\"/l0\"><button>ON</button></a>";
            String cipSend = "AT+CIPSEND=";
            cipSend += connectionId;
            cipSend += ",";
            webpage += "<a href=\"/l1\"><button>OFF</button></a>";

            if (readSensor() > 0) {
                webpage += "<h2>Millis:</h2>";
                webpage += readSensor();

            }

            cipSend += webpage.length();
            cipSend += "\r\n";
            sendData(cipSend, 100, DEBUG);
            sendData(webpage, 150, DEBUG);

            String closeCommand = "AT+CIPCLOSE=";
            closeCommand += connectionId; // append connection id
            closeCommand += "\r\n";
            sendData(closeCommand, 300, DEBUG);
        }
    }
}

String sendData(String command, const int timeout, boolean debug)
{
    String response = "";
    Serial1.print(command); // send command to the esp8266
    long int time = millis();
    while ((time + timeout) > millis()) {
        while (Serial1.available()) {
            char c = Serial1.read(); // read next char
            response += c;
        }
    }
```

```
  if (response.indexOf("/l0") != -1) {
      digitalWrite(LED_BUILTIN, HIGH);
  }
  if (response.indexOf("/l1") != -1) {
      digitalWrite(LED_BUILTIN, LOW);
  }
  if (debug) {
        Serial.print(response);
    }
    return response;
}


unsigned long readSensor() {
    return millis();
}
```

Running the program

Upload the code on Arduino. Make sure the ESP8266 module is powered on the **3.3V** and that the EN pin is also connected to the **3.3V power source!**
Do not connect any of these pins to 5V, or the adapter will be destroyed!

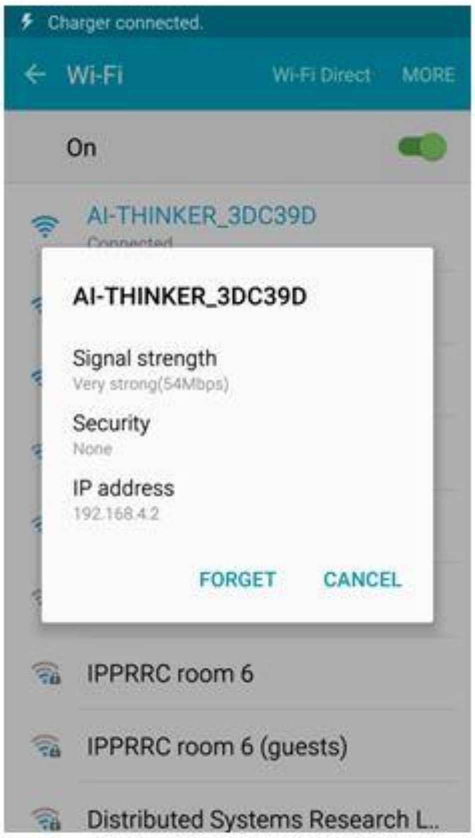
For convenience, use your smartphone and scan for WiFi networks. You should find the network's name displayed on the Serial Monitor.



```
AT+CIFSR
+CIFSR:APIP,"192.168.4.1"
+CIFSR:APMAC,"5e:cf:7f:3d:c3:9d"

OK
AT+CWSAP?
+CWSAP:"AI-THINKER_3DC39D","",1,0,4,0

OK
```

Fig.VIII.2. Looking up the network name

Do not forget to change **the baud rate in serial monitor to 115200!** As you can see, the network name for this module is "AT-THINKER_3DC39D" - with no password. The same network should be visible on your mobile device:



AI-THINKER_3DC39D
Connected

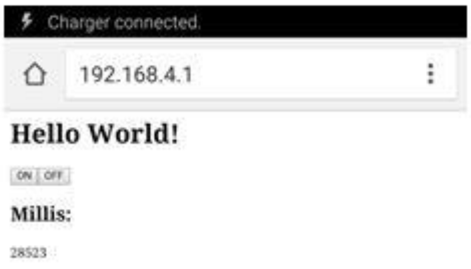
AI-THINKER_3DC39D

Signal strength
Very strong(54Mbps)

Security
None

IP address
192.168.4.2

IPPRRC room 6
IPPRRC room 6 (guests)
Distributed Systems Research L..



192.168.4.1

Hello World!

Millis:

28523

Fig.VIII.3. Name of the module as wifi access point

Fig.VIII.4. The web page

Note: Each adapter has its unique network name! Please connect only to your own network adapter, not to your colleagues' networks !

After you connect to the WiFi network, open a browser and enter the IP address of the web server: **192.168.4.1**. See the above screenshot of the resulting webpage on the mobile browser.

8.1. Individual work:

1. Run the example. Make sure the connections to the WiFi module are correct and the module is powered at 3.3 V.

2. Modify the example to display additional data in the web browser. Connect a sensor to Arduino and use the webpage buttons to select between displaying the time (millis()) and the sensor data. Make sure that the webpage tells the user what kind of information is displayed.

3. Using the WiFi module's manual, change the settings of the wireless network: change the SSID, add a password and an encryption mode.

IX. Laboratory 9 - Use of DC motors and servo motors. The experimental robot.

This lab work presents the use of DC motors, and the use of servo motors.

9.1. DC motors

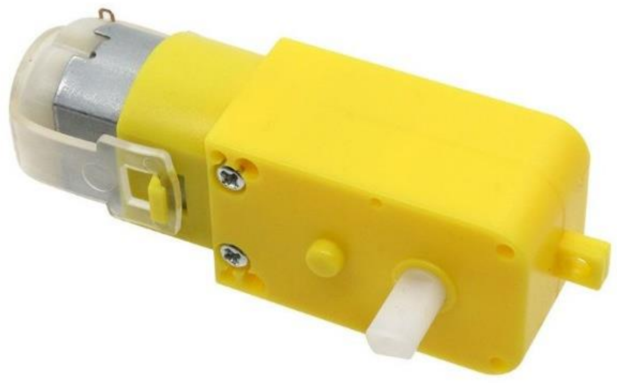


Fig.IX.1. DC Motor with 1:48 rotation rate reduction (source: https://ardushop.ro/en/electronics/64-dc-motor-3v-6v-gear-148.html)

The classic DC motors convert electrical energy into mechanical work. The rotation rate of a motor is proportional to the input voltage, and the rotation direction (clockwise or counter-clockwise) depends on the polarity of the voltage (connecting the two wires of the motor to Vcc and GND, or vice versa). The motors have a gear box (reduction mechanism) with a 1:48 reduction rate, which means that for a complete rotation of the external shaft 48 rotations of the motor shaft are needed. The reduction mechanism amplifies the force (torque), at the cost of speed.

Due to the fact that the motors require a significant current intensity to produce movement, they cannot be connected directly to the output pins of a microcontroller. A separation between the command signals and the power circuit is required, and this is achieved by using H bridges. The H bridged are circuits based on four switches (usually transistors), S1, S2, S3 and S4, as seen in Figure IX.2.

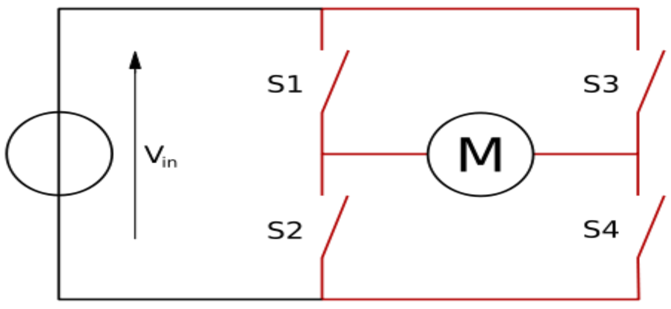


Fig.IX.2. The H bridge: S1-S4 are the switches, and M is the motor (source: https://en.wikipedia.org/wiki/H_bridge)

The name of “H bridge” comes from the shape of the power circuit. The upper left and lower right switches are usually connected to a common signal “A”, and the bottom left with the upper right switches are connected to a common signal “B”. Signals A and B must be exclusive, each one causing the motor to rotate in one direction. Activating both signals is forbidden, as it will short-circuit the power source.

The two allowed states of the H bridge’s switches are shown in Figure IX.3:

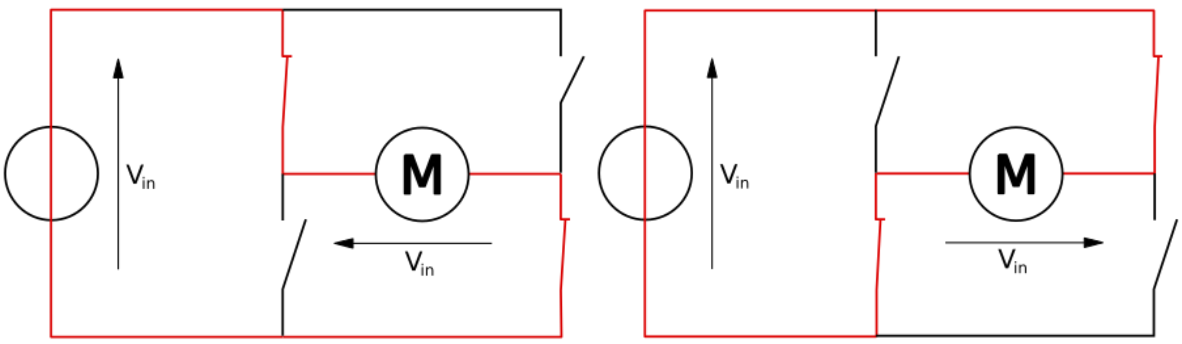


Fig.IX.3. The allowed states of the switches (source: https://en.wikipedia.org/wiki/H_bridge)

By activating the switches S1 and S4 the motor will rotate in one direction, and by activating S2 and S3 the motor will rotate in the opposite direction.

In this laboratory we will use the double bridge L298N Dual H-Bridge, capable of driving two DC motors in the same time.

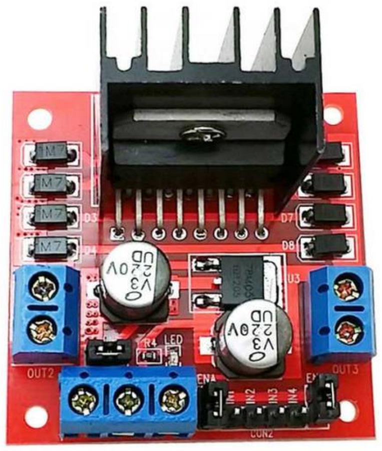


Fig.IX.4. L298N Dual H-Bridge (source: https://ardushop.ro/en/electronics/84-dual-h-bridge-for-dc-and-stepper-motors.html?search_query=L298&results=2)

Circuit specifications:

- The motor driving voltage (the pin marked +12V) Vs: 5~35V; If we want to use the same source for powering the Arduino board, a voltage between 7 and 35V must be connected, to allow the integrated voltage regulator to generate 5 V on the +5V pin.
- Maximum current for the motor driving circuit: 2A

- Logic circuit voltage (the pin marked +5V) Vss: 5 - 7V (can be connected to Arduino + 5V pin, for powering the microcontroller board)
- Maximum current for the logic circuit: 36mA
- Logic control signal levels: logic 0, $-0.3 \le Vin \le 1.5V$, logic 1, $2.3V \le Vin \le Vss$
- Maximum power: 20W

The schematic is presented in Figure IX.5:

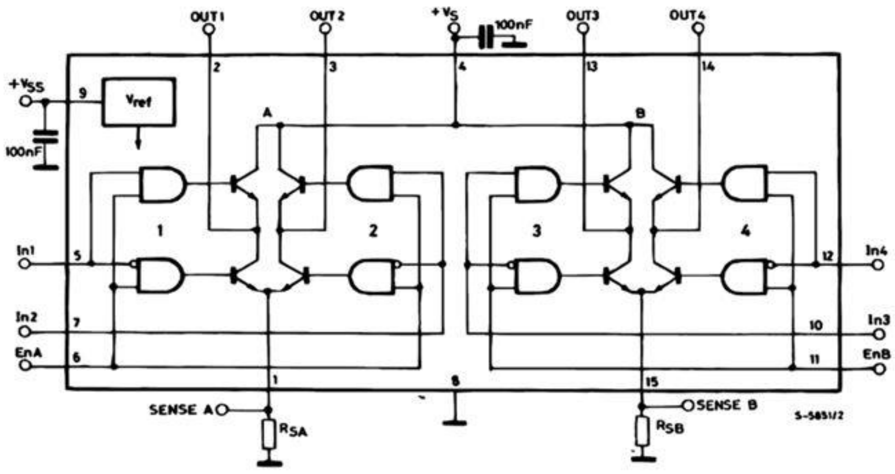


Fig.IX.5. Schematic of the L298N driver circuit (source: L298 datasheet)

For each motor there are three control pins. The first motor is connected to pins EnA, In1 and In2 and the second motor to pins EnB, In3 and In4. The En pins are connected to logic 1 using jumpers, and thus the user program will only control the In pins. The following combinations are available:

In1	**In2**	**Effect**
0	0	Motor 1 stopped (brake)
0	1	Motor 1 on – forward
1	0	Motor 1 on – reverse
1	1	Motor 1 stopped (brake)

In3	**In4**	**Effect**
0	0	Motor 2 stopped (brake)
0	1	Motor 2 on – forward
1	0	Motor 2 on – reverse
1	1	Motor 2 stop (brake)

Fig.IX.6. Motor commands

The motors can be driven simultaneously. Do not change the motor’s direction of rotation without stopping it first for several milliseconds.

The rotation rate of a motor is controlled by the voltage applied to its power pins. Since a microcontroller can only generate 0 or 5 V voltage levels, for changing the rotation rate a PWM signal on pins In1, In2, In3 or In4 can be used.

9.2. Servo motors

Unlike the DC motors, which rotate continuously as long as they are connected to a power source, the servo motors are used to achieve partial, stable and controlled rotations, for small amplitude but high precision operations: locking mechanisms, positioning of sensors, gestures, etc.

Fig.IX.7. A servo motor (source: https://www.indiamart.com/proddetail/sg90-9g-servo-motor-14077708788.html)

The servo motors have three wires, of different colors (depending on the manufacturer). The red color usually denotes Vcc (5V), while GND is usually black or brown. Besides the power wires, the command wire is usually yellow, orange or white. Figure IX.8 shows several color schemes.

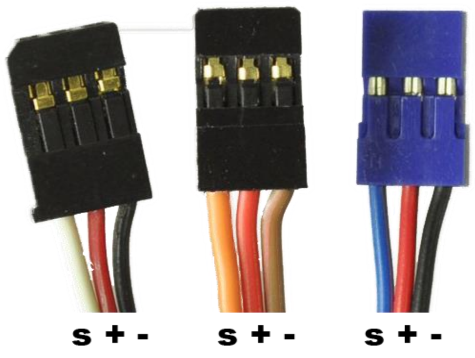


Fig.IX.8. Color schemes used for servo motor wires (source: http://www.pitch-play.nl/tips-and-tricks/servo-wires/)

The servo motor will not (usually) execute a complete rotation, but will deviate from a neutral position with an angle controlled by the voltage applied to the signal pin. Using a PWM signal on this pin, we can control the motor's angle of rotation.

The simplest way of controlling the servo motors is by the use of the Servo library. With this library, up to 48 servos can be controlled by Arduino Mega. If more than 12 servos are used, the library will disable PWM on pins 11 and 12. On Arduino Uno, this library will disable PWM on pins 9 and 10, regardless of the number of servos used.

The methods of the Servo class are the following:

servo.attach(pin) / servo.attach(pin, min, max) – attaches the Servo object to pins

- servo: a Servo class object
- pin: number of digital pin for the servo control signal
- min (optional): width of the pulse, in microseconds, for the minimum angle (0 degrees) of the servo motor (implicitly 544)
- max (optional): width of the pulse, in microseconds, for the maximum angle (180 degrees) of the servo motor (implicitly 2400)

servo.detach() – detaches the Servo object from the pin.
boolean val servo.attached() – checks whether the Servo object is attached to a pin.

servo.write (angle) – writes a value (0 .. 180) to the servo, controlling its motion:

- For standard servos ⇒ sets the angle in degrees, causing the servo to orient in the specified position.
- For continuously rotating servos ⇒ configures the rotation speed (0: maximum speed in one direction; 180: maximum speed in the opposite direction; ≈ 90: stopped)

int val servo.read() – reads the current angle of the servo, set by the last call of **write**().

9.3. The experimental robot

For the project activities, experimental robots have been assembled, with the following components:

1. Arduino Uno compatible microcontroller board
2. L298N Dual H-Bridge motor driver
3. 2x DC Motor
4. 1 Servo motor
5. Battery case 4xAA (R6)
6. 2 wheels connected to the motors, 1 balance wheel
7. Acrylic support
8. 2x Breadboard
9. 1 sonar sensor (not connected)

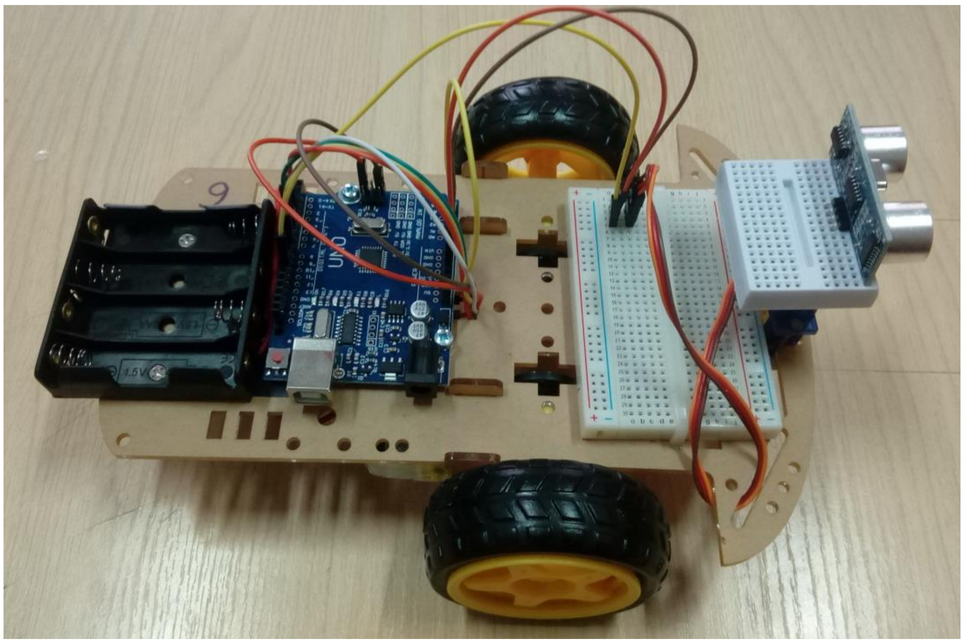


Fig.IX.9. The experimental robot – top view – control elements

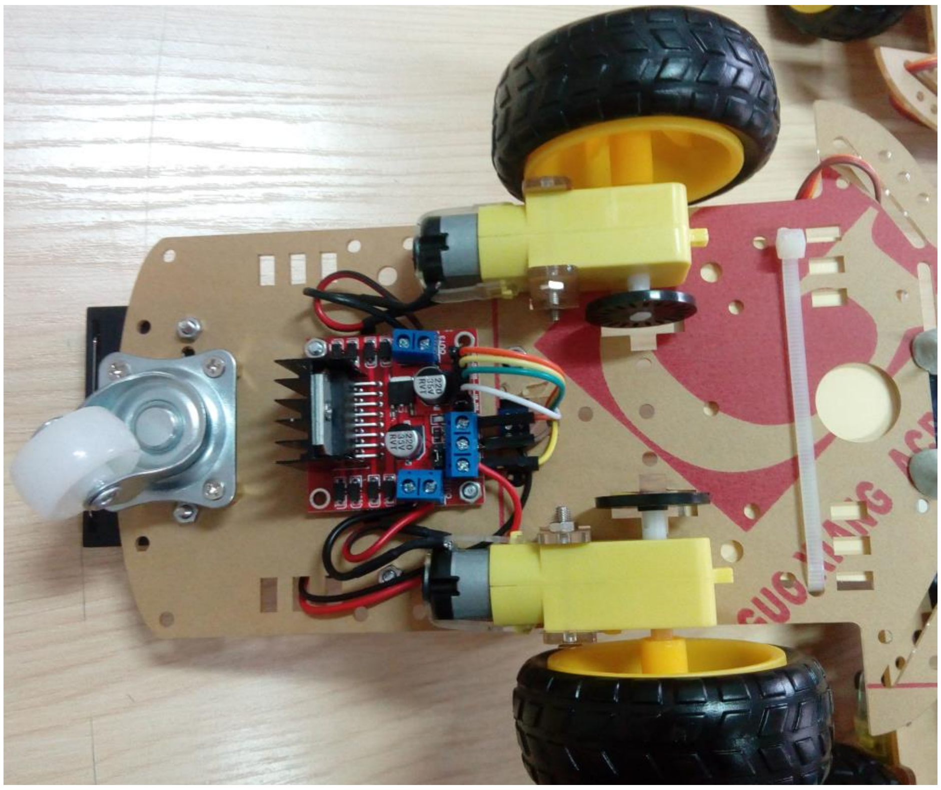


Fig.IX.10. The experimental robot – bottom view – power circuit

The schematic of the connections between the elements of the robot is shown in Figure IX.11:

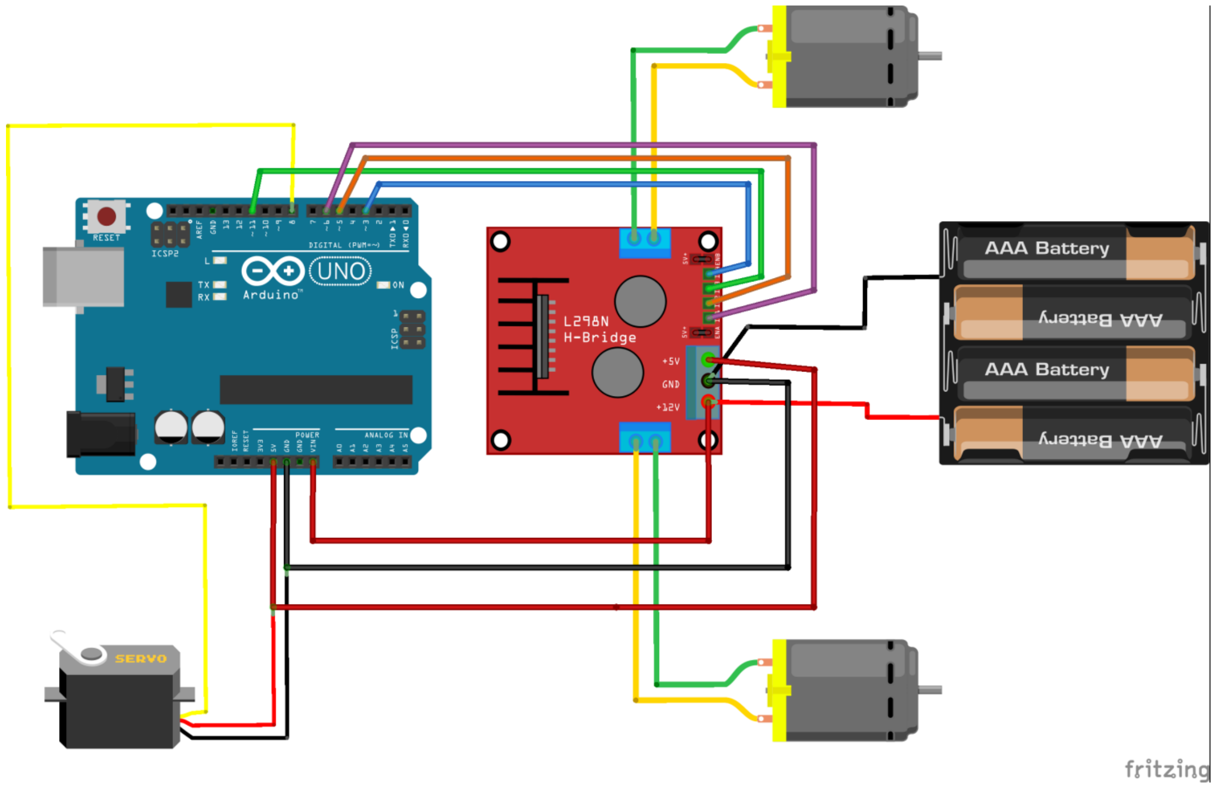


Fig.IX.11. Electric schematic of the robot

Important features:

- The motors can be powered by batteries (not optimal, according to the L298N specifications, but offers mobility), or by an external power supply connected to the Arduino jack power connector. The Vin pin of Arduino is connected internally to the jack connector and also to the +12V pin of L298N.
- Do not use an external power supply along with the batteries, as they will be exposes to the voltage of this source!
- The +5V output of L298N is connected to the +5V power pin of the Arduino board. For saving pins, the ICSP connector has been used.
- The control pins for the motors have been connected to Arduino pins 3 and 11 (Motor 1) and 5 and 6 (Motor 2), pins capable of PWM signal generation.
- The servo control pin is connected to Arduino pin 8.

Safety advice:

- Do not let the robot start its motors while connected to the PC via the programming cable. The example program shown on the next page there is a built in warning time (the LED on pin 13 will blink, at first slower, then faster, before the motors are started). Program the robot, then disconnect the USB cable and connect the external power supply, or the batteries.
- Check the polarity of the power supply connector (the positive electrode must be in the center of the connector) and its set voltage (recommended 7.5V).
- Do not let the robot start the motors while the wheels are on the table! It will move suddenly, and may cause breaking of cables, or it may fall from the table. Hold the robot in hand, or

mount it on a stand to keep the wheels away from the table. When you want to test the robot's motion, place it on the floor and leave enough space around it.

9.4. Example code

The following code is already programmed in the robot's microcontroller. This code will test all motors, after the execution of the warning code.

```
#include <Servo.h>

// Pins of motor 1
#define mpin00 5
#define mpin01 6

// Pins of motor 2
#define mpin10 3
#define mpin11 11

Servo srv;

void setup() {
    // configuration of motor pins as output, initially 0
    digitalWrite(mpin00, 0);
    digitalWrite(mpin01, 0);
    digitalWrite(mpin10, 0);
    digitalWrite(mpin11, 0);

    pinMode (mpin00, OUTPUT);
    pinMode (mpin01, OUTPUT);
    pinMode (mpin10, OUTPUT);
    pinMode (mpin11, OUTPUT);

    // LED pin
    pinMode(13, OUTPUT);
}

// Function to control a motor
// Input: pins m1 and m2, direction and speed
void StartMotor (int m1, int m2, int forward, int speed)
{
    if (speed==0) // stop
    {
        digitalWrite(m1, 0);
        digitalWrite(m2, 0);
    }
    else
    {
        if (forward)
        {
            digitalWrite(m2, 0);
            analogWrite(m1, speed); // use PWM
        }
        else
        {
            digitalWrite(m1, 0);
            analogWrite(m2, speed);
         }
    }
}

// Safety function
// Commands motors to stop, then delays
void delayStopped(int ms)
{
    StartMotor (mpin00, mpin01, 0, 0);
    StartMotor (mpin10, mpin11, 0, 0);
    delay(ms);
}

// Use of servo
// Set three angles
// When finished, the servo remains in the middle (90 degrees)
void playWithServo(int pin)
{
    srv.attach(pin);
    srv.write(0);
    delay(1000);
    srv.write(180);
    delay(1000);
    srv.write(90);
    delay(1000);
    srv.detach();

}

void loop() {

    // Warning code
    // Slow blink
    for (int i=0; i<10; i++)
    {
        digitalWrite(13, 1);
        delay(200);
        digitalWrite(13, 0);
        delay(200);
    }

    // Fast blink. Remove the USB cable!!!!
    for (int i=0; i<10; i++)
    {
        digitalWrite(13, 1);
        delay(100);
        digitalWrite(13, 0);
        delay(100);
    }

    digitalWrite(13, 1);

    // Start the servo motor
    playWithServo(8);

    // Now start the DC motors
    StartMotor (mpin00, mpin01, 0, 128);
    StartMotor (mpin10, mpin11, 0, 128);

    delay (500); // How long the motors are on
    delayStopped(500); // How long the motors are off

    StartMotor (mpin00, mpin01, 1, 128);
    StartMotor (mpin10, mpin11, 1, 128);

    delay (500);
    delayStopped(500);

    StartMotor (mpin00, mpin01, 0, 128);
    StartMotor (mpin10, mpin11, 1, 128);

    delay (500);
    delayStopped(500);

    StartMotor (mpin00, mpin01, 1, 128);
    StartMotor (mpin10, mpin11, 0, 128);

    delay (500);
    delayStopped(500);
}
```

9.5. Project activity rules:

1. Each team of students (1 or 2 students) will receive a robot. The lab assistant will write down in the attendance sheet the number of the robot for each student. In the following weeks, the students will use the same robot, if possible.
2. Check the connections for compliance with the documentation. Run the test program, check for all motors to work correctly (each motor must perform rotations both ways). If defects are found, they will be identified and you will attempt to fix them.

Step 2 is mandatory for each lab session!

3. The teams will add functions to the robot, by changing the program and adding components. Avoid permanent changes, leave the robots in the same state as you found them.
4. Each team will keep their code confidential. Each student is responsible for ensuring that no code remains on the workstation, and that their progress is securely saved.
5. In the last lab session, the project will be evaluated based on functionality, complexity and originality. The students must hand out a short documentation (5-10 pages).

Each damaged component will be replaced by the person that damaged it!

Ideas for additional functions (may be cumulative):

- Implement a precise control of the motors to ensure movement in a straight line. Implies measuring the rotation speed and tuning the PWM pulse to ensure equal rotation for both motors.
- Detecting and avoiding obstacles. Implies using the sonar sensor along with the servo motor to determine distance and angle to obstacles, and to compute free space.
- Mapping the environment. Implies detecting the obstacles and keeping track of the own movement, for placing the obstacles on the map.
- Use of the EEPROM memory to remember a path previously traveled.
- Remote control, using the mobile phone. Implies using the WiFi module.
- Communication between robots using gestures.
- Following a leader robot.

It is possible that the additional functions may require additional hardware to be acquired. If the components are acquired by the students, they remain in their property. It is recommended that each team buys at least 4 alkaline batteries.

Recommendation: you can take advantage of the features of the Mega boards without removing the UNO boards from the robot. Connect the boards by I2C, and use UNO only for motor control, and Mega for the other functions. Connect together the 5V and GND pins of the two boards, for sharing power.

Additional references:

1. Photoelectric sensor for using along with the code wheel, for measuring rotation speed:
https://ardushop.ro/ro/home/146-senzor-de-intrerupere-infrarosu.html?search_query=fotoelectric&results=1
or
https://www.robofun.ro/senzor-ir-break-beam-led-3mm

2. Specifications of the sonar sensor for distance measurement:
https://cdn.sparkfun.com/datasheets/Sensors/Proximity/HCSR04.pdf

3. Using the Sharp distance sensors:
http://www.robotshop.com/letsmakerobots/files/IR_SHARP.doc

4. Use of the AVR built in EEPROM:
https://www.arduino.cc/en/Reference/EEPROM

Appendix 1 – Using the Processing IDE

1. Introducing Processing

For better visualization of the data sent by a microcontroller, data sent to the PC via the UART serial interface, several software tools can be used. One of those products is Processing (https://processing.org/). Using Processing we can create graphical user interfaces to display data as graphs or tables.

Processing is an open source solution, like Arduino, having a very similar graphical user interface.

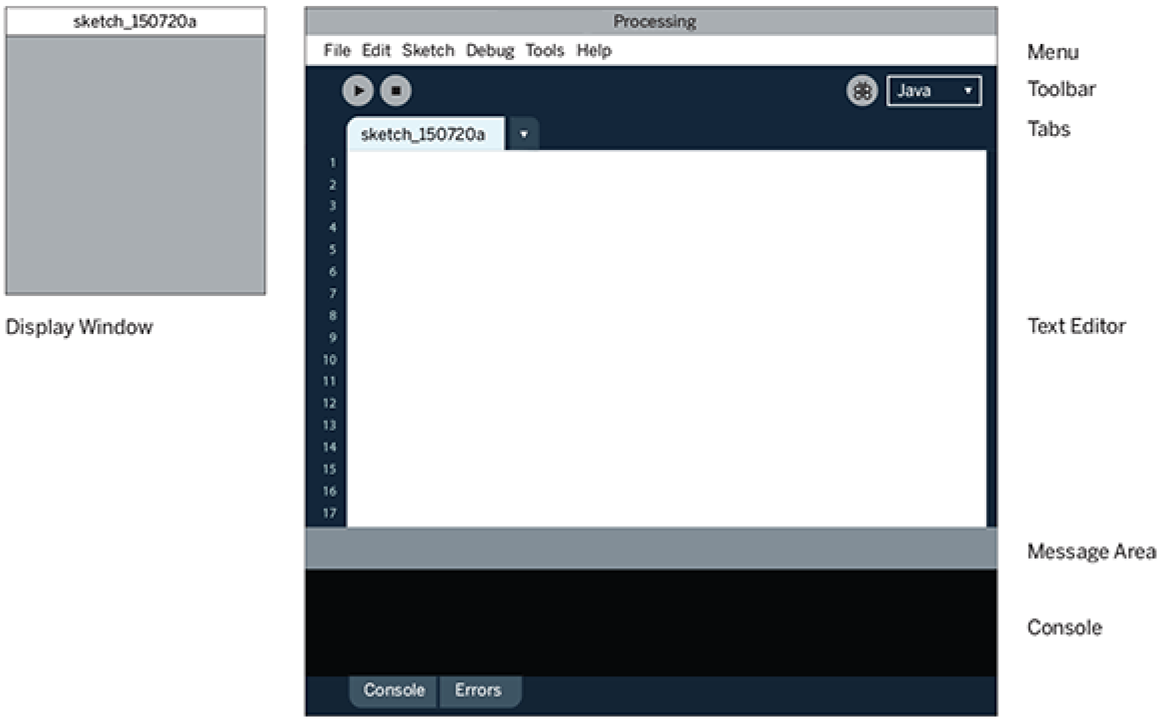


Fig.A1.1. Processing IDE

In the text editor window we'll write the programs. The "Message Area" and the "Console" areas will provide feedback as the program is compiled or executed.

Similar to the Arduino programs, the Processing programs have an initialization section, and a loop execution section. The initialization is achieved by executing the "setup()" function, and everything contained in the "draw()" function is executed continuously (equivalent to the Arduino "loop()" function).

Example 1 - Drawing a line and an ellipse in Processing:

```
void setup() {
    size(500, 300);
}

void draw() {
    line(15, 15, 15, 50);
    ellipse(150, 150, 80, 80);
}
```

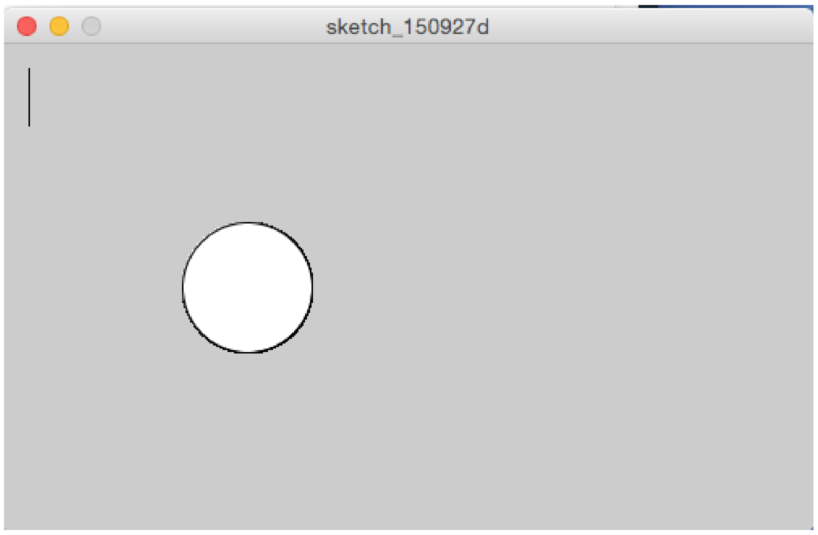


Fig.A1.2. Drawing graphical primives in processing

In “setup()” we’ll define the size of the window. In this example the window is 500 pixels wide, and 300 pixels tall. The “draw()” function includes graphical commands that will draw inside the window multiple graphical shapes: ellipse, rectangle, line, square, etc. The background color of the window can be specified in the “setup()” function:

```
void setup() {
    size(500, 300);
    background(0, 0, 0);
}
```

The colors are defined by values between 0 and 255 for each color channel (Red, Green, Blue - RGB), therefore for a window of white background we call *background(255, 255, 255)*, or simply *background(255)*. The colors can be also specified by the web format: *background(#ffffff)*.

The Processing coordinates system originates from the upper left corner. A line is defined by its endpoints A(x1, y1) și B(x2, y2).

A rectangle is defined as: *rect(x, y, width, height)*, where (x, y) is the position of the upper left corner of the rectangle.

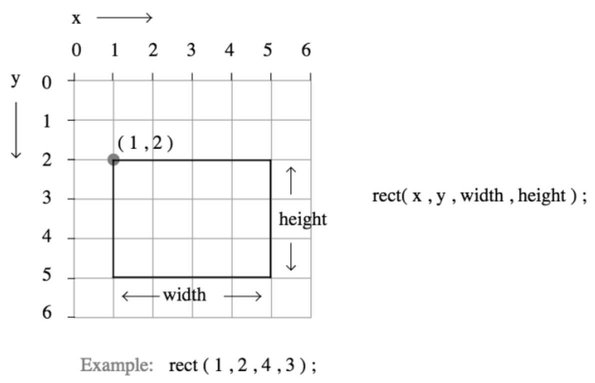

x
0 1 2 3 4 5 6
y
0
1
2
3
4
5
6
(1 , 2)
height
width
rect(x , y , width , height) ;
Example: rect (1 , 2 , 4 , 3) ;

Fig.A1.3. Coordinates for drawing a rectangle.
(source: https://processing.org/tutorials/drawing/)

A rectangle's position can also be specified by the position of its center, if *rectMode(CENTER)* is called before drawing. Another way to specify the rectangle is to specify its upper left and lower right corners, using the mode *rectMode(CORNERS)*.

For more information and more examples, go to : https://processing.org/tutorials/drawing/.

2. Using sensor data with Processing

The data read by Arduino from sensors can be processed and displayed much easier using the Processing IDE. We will use the temperature reading example from lab 7 (**that you will modify to send only the numerical value via the Serial port!**) and we will send via the serial interface the value of the temperature. In Processing we will read these values, and use them to generate a graph of the temperature's variation with time.

Example 2 - Reading data from the Serial interface using Processing.

```
import processing.serial.*;

Serial myPort;
int xPos = 1;        // the index on the X axis of the graph
int lastxPos=1;
int lastheight=0;

void setup() {
    size(400, 300);
    println(Serial.list());
    int lastport = Serial.list().length;
    String portName = Serial.list()[lastport-1]; // last serial
    //port of the PC - the Arduino port
    myPort = new Serial(this, portName, 9600);
    myPort.bufferUntil('\n');
    background(0);
}

void draw() {
    String inString = myPort.readStringUntil('\n');
    if (inString != null) {
        inString = trim(inString);
        float inByte = float(inString);
        println(inString);
        inByte = map(inByte, -5, 45, 0, height); // map the
        //temperature range to the window height

        stroke(231, 76, 60); // line color (RGB color components)
        strokeWeight(2);// line thickness
        line(lastxPos, lastheight, xPos, height - inByte); // draw
        //line
        lastxPos= xPos;
        lastheight= int(height-inByte);
        xPos++;
    }
}
```

For reading the data we need to use the Serial library of Processing. The function “Serial.list()” will return a list of all serial ports of the PC, and will help us in selecting the correct port in the “setup()” phase.

The serial port is initialized for reading as:

myPort = new Serial(this, portName, 9600);

The data is read until the “newline” character is met:

myPort.bufferUntil(‘\n’);

We can read one string at the time by using:

myPort.readStringUntil(‘\n’);

If data is received, the string is converted to float, and the value is used to draw lines on the screen so that we can generate a temperature graph:

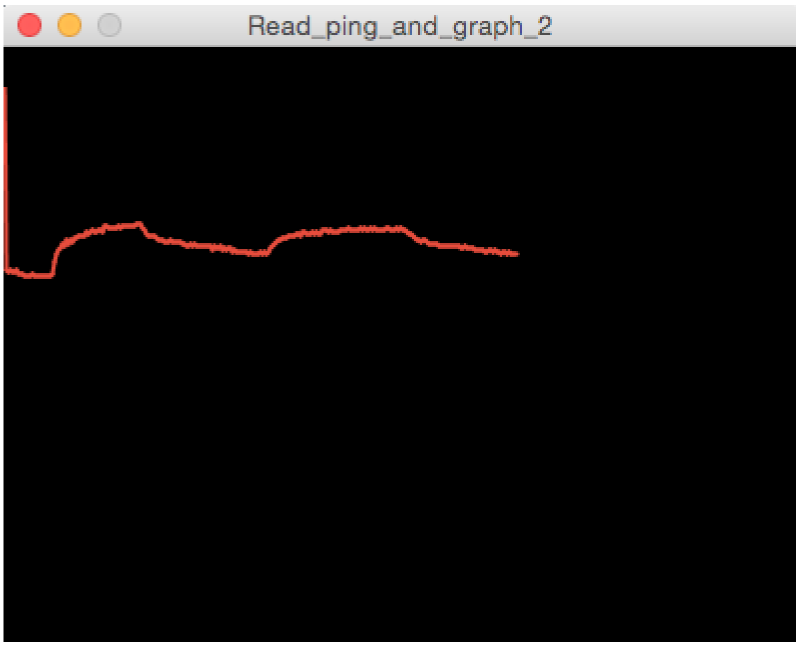


Fig.A1.4. Ploting a graph from sensor data

The lines are drawn by connecting the current coordinates with the previous ones, so that the graph will look continuous.

3. Writing data in in Processing

The serial communication library of Processing allows, besides reading data, to send data via the serial interface. You can send characters by calling the *write* function.

```
import processing.serial.*;

Serial myPort;

// Display the available serial ports
printArray(Serial.list());

myPort = new  Serial(this,  Serial.list()[Serial.list().length-1],
9600);

// Write “A” on the last serial port
myPort.write(65);
```

Source: https://processing.org/reference/libraries/serial/Serial_write_.html

A character string can be sent by a single call of the *write* function:

```
myPort.write(“Hello”);
```

4. Reading the data sent by Processing in Arduino

The data sent by Processing can be read in Arduino using the serial interface and the Serial library. Reading a string of characters can be done by calling:

string dataFromSerial = Serial.readStringUntil('\n');

Example 3: reading data in Arduino – Arduino code:

```
#include <LiquidCrystal.h>

LiquidCrystal lcd(7, 6, 5, 4, 3, 2);
String inputString = "";          // string for input data
boolean stringComplete = false;   // indicates full string

void setup() {
    Serial.begin(9600);
    lcd.begin(16, 2);
    lcd.print("Read Processing:");     // display message on LCD
    inputString.reserve(16); // reserve 16 bytes for inputString
}

void loop() {
    if (stringComplete) {
        lcd.setCursor(0, 1);
        lcd.print(inputString);
        inputString = "";
        stringComplete = false;
    }
}

void serialEvent() {
    while (Serial.available()) {
        inputString = Serial.readStringUntil('\n');
        stringComplete = true;
    }
}
```

The above code uses the LCD shield to display the message read from the serial interface.

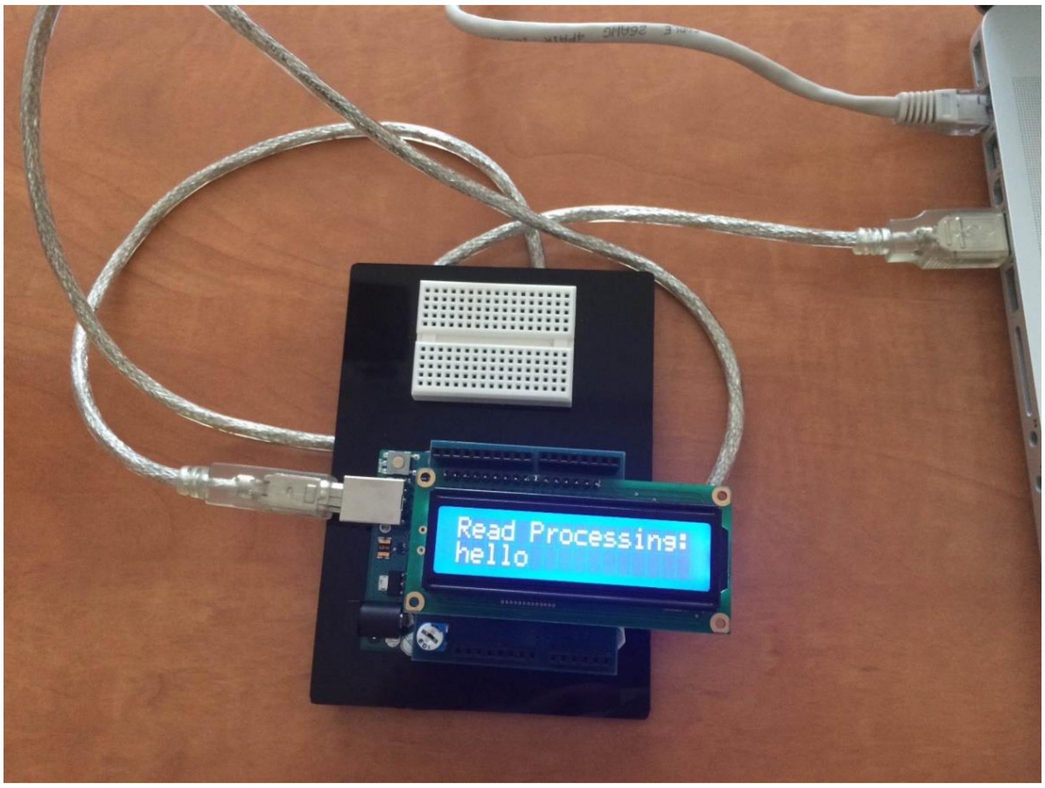


Fig.A1.5. Example of reading a character string sent by Processing using the serial interface)

In Processing we’ll create a simple interface, with a single button.

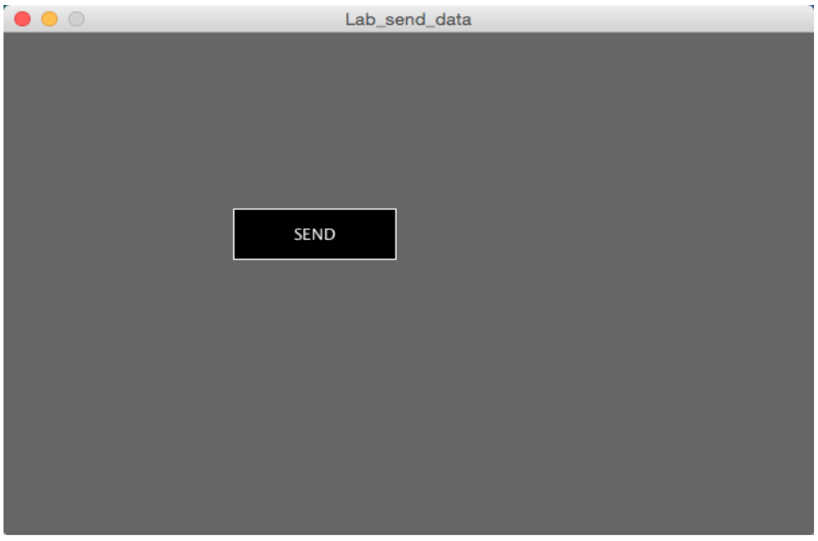


Fig.A1.6. The minimal Graphical Interface in Processing with a single button

Example 3 - Code for Processing:

```
import processing.serial.*;

int rectX, rectY;
int rWidth = 120;
int rHeight = 40;
color rectColor;
color rectHighlight;
boolean rectOver = false;

int c = 0;

Serial myPort;

void setup() {
    size(600, 400); // window size
    rectColor = color(0); // initial color of the button
    rectHighlight = color(51); // color of the button on mouse
    //hover

    // button position
    rectX = width/2-rWidth-10;
    rectY = height/2-rWidth/2;

    printArray(Serial.list());
    myPort = new Serial(this, Serial.list()[Serial.list().length-
    1], 9600);
}

void draw() {
    update(mouseX, mouseY); // for changing the button color
                          // when mouse is over it

    // change the color of the button
    if (rectOver) {
        fill(rectHighlight);
    }
    else {
        fill(rectColor);
    }

    // draw the contour of the button
    stroke(255);
    rect(rectX, rectY, rWidth, rHeight);

    fill(255);
    text("SEND", rectX + rHeight + 5, rectY + rHeight/2 + 5);
}

void update(int x, int y) {
    rectOver = overRect(rectX, rectY, rWidth, rHeight);
}

// action when the mouse button is pressed
void mousePressed() {
    if (rectOver) {
        myPort.write("hello"+c);
        c++;
    }
}

// returns true if the mouse is above the button
boolean overRect(int x, int y, int width, int height) {
    if (mouseX >= x && mouseX <= x+width &&
    mouseY >= y && mouseY <= y+height) {
        return true;
    }
    else {
        return false;
    }
}
```

Individual work:

1. Run the examples in this lab work.
2. Change example 2 to reset the graph once the the end of the screen is reached.
3. **Optional, instead of 2:** change example 2 so that the graph will scroll continuously towards the left when it becomes full.
4. Add the temperature graph text information. Use the function **text (text_to_display, x, y)**, where x is the horizontal coordinate, and y the vertical coordinate. Display the text information at a time interval that will ensure the texts will not overlap.
5. Change Example 3 (Arduino and Processing code) in order to control the state of some LEDs from the graphical user interface. Connect to Arduino a LED module. In Processing, add four buttons to the graphical interface. When pressing a button in the Processing program, the state of the corresponding LED connected to Arduino will change.
6. Use the “keyboard” functions of Processing to read keys. Display the pressed key in the Processing console, but also on the Arduino LCD.

References:

1.https://www.arduino.cc/en/Reference/AnalogRead
2.https://www.arduino.cc/en/Reference/AnalogReference
3.https://processing.org/
4.https://processing.org/examples/keyboard.html

Appendix 2 - Robotic Operating System (ROS) and Arduino

What is ROS and where can we use it

ROS is not an actual operating system, but a collection of software development frameworks designed with an architecture for inter-process / inter-machine communication. This robotics middleware has a programming language and hardware independent architecture which provides services such as hardware abstraction, low level device control, message passing between processes, package management and others. ROS is not designed to replace an actual operating system but work beside it. Mainly due to the fact that ROS uses a lot of open source libraries the best software support is available for Linux based systems.

The robotic operating system can be used in applications where distributing computing, reuse and rapid testing are required. For example, in a distributed computing application many different robots can rely on software that runs on different computers and spans on different processes. ROS is a good option when one wants to reuse code that runs on a robot, for example, functions regarding mapping or path planning.

Organization in ROS

The ROS file system is organized at two levels:

- Packages
  They are the lowest level of organization and are intended for a single functionality.
  There is also a file, called a manifest, that is responsible for two things: first it contains the description of the package and then it defines dependencies between packages.
- Stack
  These are collections of packages forming higher level libraries. There is a manifest file in the stack as well, having the same purpose as the one for packages

ROS communication platform

ROS nodes are responsible for specific computations. They represent processes distributed across a ROS network that can give or get data from the network. Some examples of tasks that can be performed by nodes are:

- Control robot wheels
- Acquire images from camera
- Provide graphical visualization of the system

ROS core is composed by three programs that are necessary for the ROS runtime:

1. **ROS master**
   The ROS master is the core node of ROS and behaves like a DNS server. The ROS master is a centralized RPC server that negotiates communication connections and registers and looks up names for ROS graph resources.
   It stores topics and services registration information for ROS nodes and makes callbacks to nodes when registration information changes. This master node also allows nodes to dynamically make connections as new nodes are run.
2. **Parameter Server**

It runs in the ROS master and it is responsible for storing persistent configuration parameters and other arbitrary data. While the core is running, data is stored in the parameter server. This server is not designed for high performance and hence it is better suited for configuration parameters. ROS naming convention prevents data conflicts (by using namespaces).

3. **Rosout**
   It is basically a network based standard out for human readable messages.

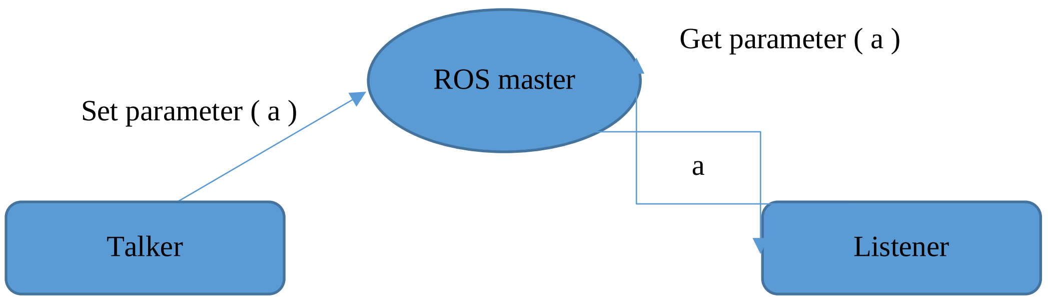


Fig.A2.1. Communication in ROS

Messages
Nodes communicate with each other by passing messages which are a data structure consisting of a type field. Messages can be routed either by using topics or by using services. Standard or primitive data types supported are: int, float, array[], etc.

Topics and services
Topics and services are methods of communication among nodes. Services rely on a query made from the terminal or by a node, retrieving the response that is provided by another node that is offering the service. Topics require a subscription to a node which will broadcast the information. Another difference between topics and services is that topics are asynchronous communication streams with multiple sources and destinations, while services are synchronous single-source communication functions with multiple destinations.

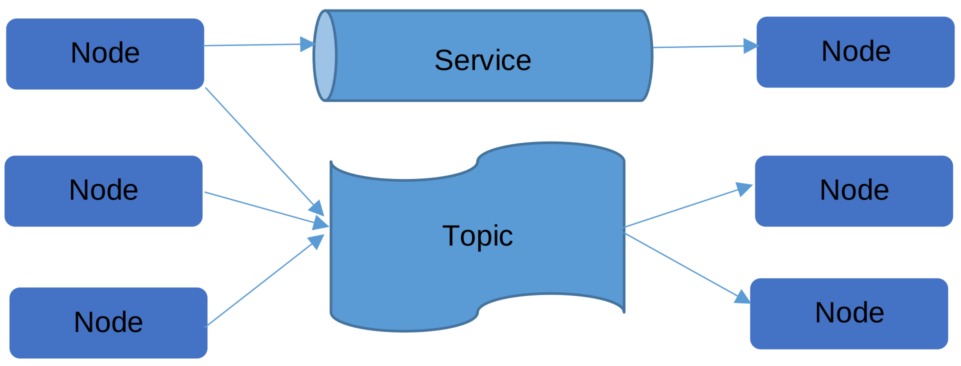


Fig.A2.2. Services and Topics in ROS

Picture after the image provided in ROS Tutorial book, Robotics Operating System, Antonio Marin-Hernandez, October 31, 2014

The publish subscribe model is a flexible paradigm where publishers and subscribers are generally unaware of each others existance. A single node may publish and subscribe to multiple topics. A node sends out a message by publishing it to a specific topic and a node that needs a specific type of data must subscribe to an appropriate topic. The topic type is defined by the message types that the nodes publish on it. For the publish – subscriber "game" there is no order of execution required.

Services can not use the publish – subscribe paradigm. They implement a request reply functionality (by using a pair of message structures i.e. one for request and one for reply). In general, there is a node provider which offers a service under a specific name. The client node consumes the service by sending a request message and awaits the reply.

In the following examples we will see three simple examples of programs using rosserial. The version of ROS used was ROS Indigo Igloo (Indigo Igloo is the 8th official ROS release). The microcontroller used was an atmega 2560. The setup for the IDE will not be discussed since it can be found on the Internet.

ROS environment quick setup guide

To set up the ROS environment either refer to the official ROS website and follow the step by step instructions or use a virtual machine with ROS preinstalled. We will use the preinstalled ROS indigo version but we encourage the reader to also check the step by step guide and explore even newer versions of the ROS framework.

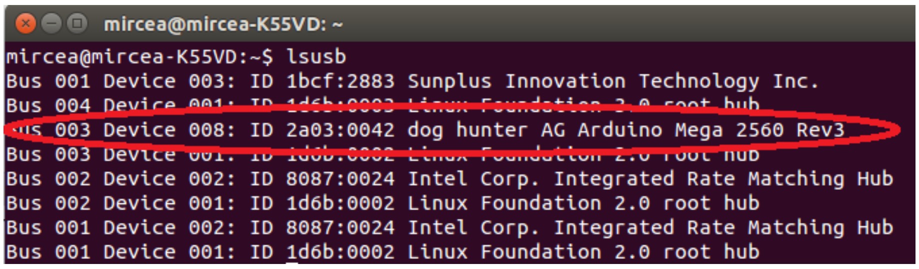
The link to the 32 and 64 bits VMs is given in reference number 7. The hypervisor we have used is Virtual Box. For this hypervisor it is **essential** to download and install the extension pack for the specific hypervisor version you use, so that the USB ports will also be visible and available inside the virtual machine.

Once VirtualBox is installed you need to make a few changes inside the virtual machine. Click the settings button and navigate to the USB section. Check the check box "Enable USB controller" and the check box "Enable USB 2.0 (EHCI) Controller".

An important step that has to be done if the host computer is also using Linux is to add virtual box users to the host machine user group. This can be done using the command:

```
sudo adduser $USER vboxusers
```

To see whether the Arduino board is recognized by the host machine (under linux) open a terminal and type lsusb. If the board is recognized you will see an entry with the name of the Arduino board.

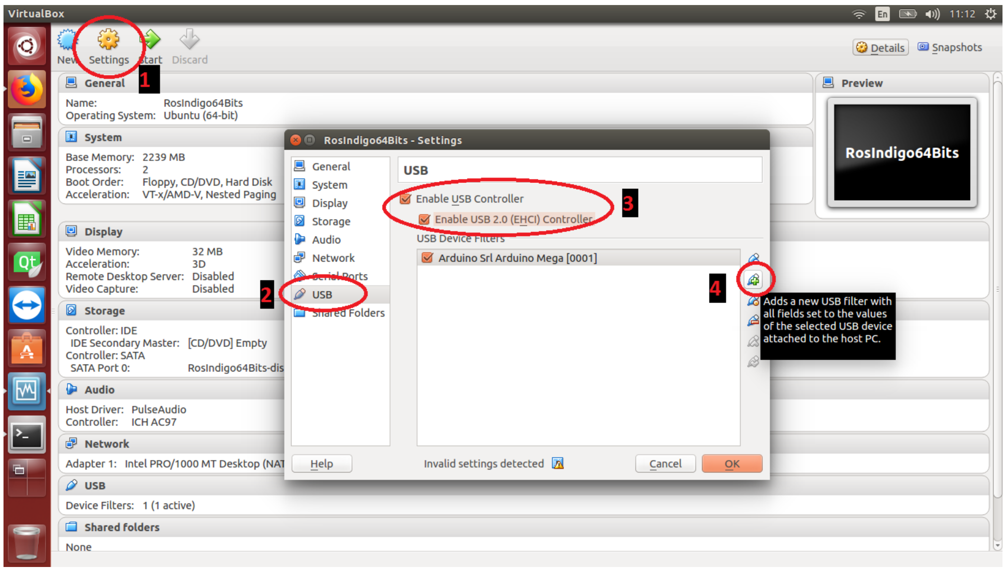


```
mircea@mircea-K55VD:~$ lsusb
Bus 001 Device 003: ID 1bcf:2883 Sunplus Innovation Technology Inc.
Bus 004 Device 001: ID 1d6b:0003 Linux Foundation 3.0 root hub
Bus 003 Device 008: ID 2a03:0042 dog hunter AG Arduino Mega 2560 Rev3
Bus 003 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 002 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 002: ID 8087:0024 Intel Corp. Integrated Rate Matching Hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Fig.A2.3. USB port connected to the Arduino board

After the check boxes have been validated, the virtual box users are in the same group as the host machine users and the Arduino board is recognized by the host computer, add a new

USB filter with all the fields set to the values of the selected USB device attached to the host PC. The steps mentioned above are depicted graphically in the figure below:



General
Name: RosIndigo64Bits
Operating System: Ubuntu (64-bit)

System
Base Memory: 2239 MB
Processors: 2
Boot Order: Floppy, CD/DVD, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging

Display
Video Memory: 32 MB
Acceleration: 3D
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: IDE
IDE Secondary Master: [CD/DVD] Empty
Controller: SATA
SATA Port 0: RosIndigo64Bits-dis

Audio
Host Driver: PulseAudio
Controller: ICH AC97

Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT

USB
Device Filters: 1 (1 active)

Shared folders
None

RosIndigo64Bits - Settings
General
System
Display
Storage
Audio
Network
Serial Ports
USB
Shared Folders

USB
☑ Enable USB Controller
☑ Enable USB 2.0 (EHCI) Controller
USB Device Filters
☑ Arduino Srl Arduino Mega [0001]

Adds a new USB filter with all fields set to the values of the selected USB device attached to the host PC.

Help
Invalid settings detected
Cancel
OK

Fig.A2.4. Steps for visualizing the Arduino board in the ROS virtual machine

Now, you can start the virtual machine by pressing the green arrow start. The default user name and password is “viki”.

After opening the virtual machine you need to install the Arduino IDE. Using the rosserial_arduino package you can use ROS with Arduino. This package works over the Arduino UART and it provides the a ROS communication considering Arduino a ROS node, which can have the same functionality as any other ROS node. As any library the ros_lib has to be copied into the Arduino libraries to be visible in the Arduino IDE.

To install the library on the ROS VM you need to use the following instructions:

```
sudo apt-get install ros-indigo-rosserial-arduino
sudo apt-get install ros-indigo-rosserial
```

This step creates the ros_lib which has to be copied into the Arduino build environment. You have to delete libraries/ros_lib in order to regenerate as its existence causes an error. The sketchbook is the directory where the Linux Arduino saves the sketches.

```
cd <sketchbook>/libraries
rm -rf ros_lib
rosrun rosserial_arduino make_libraries.py .
```

After these steps the ros_lib should appear in the examples section of the Arduino IDE.

Example 1: Publisher

In this program we will create a publisher on an atmega 2560 microcontroller and we will show the messages that the AVR sends to a topic in a console window. For this first example we will need an Arduino board and a USB cable to connect the board to the computer on which ROS is installed.

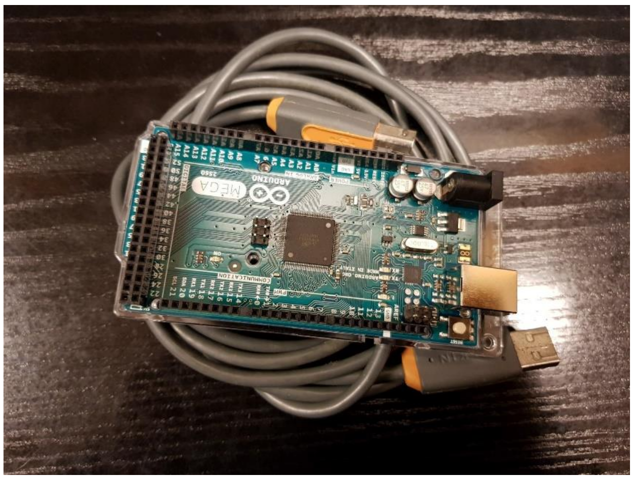
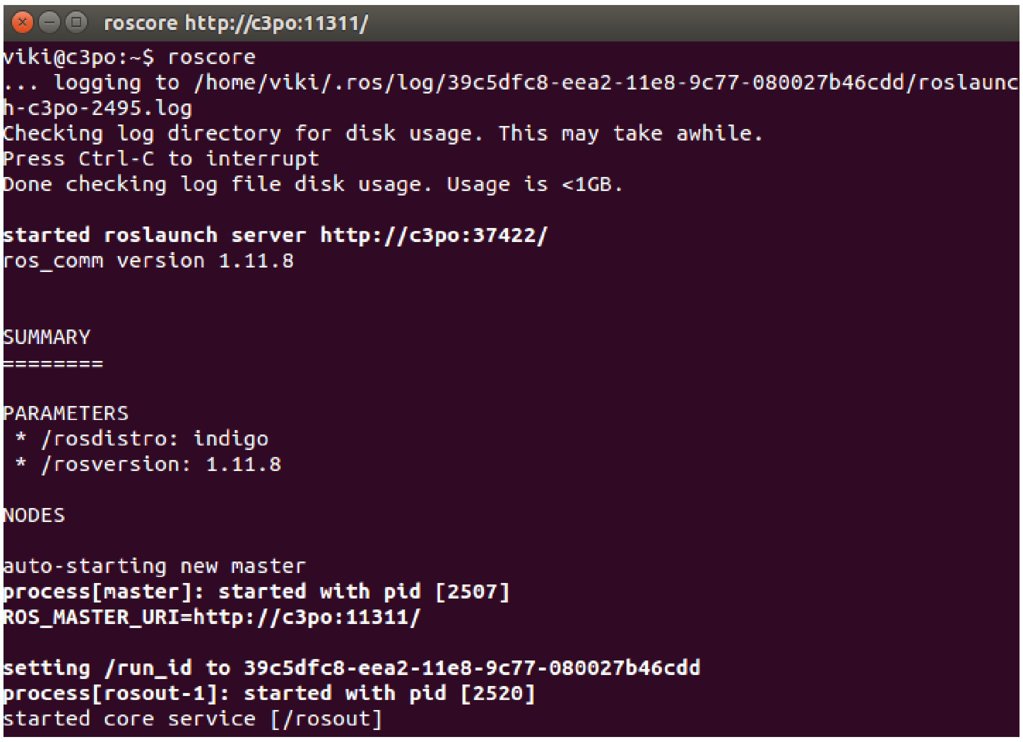


Fig.A2.5. Arduino board and USB cable

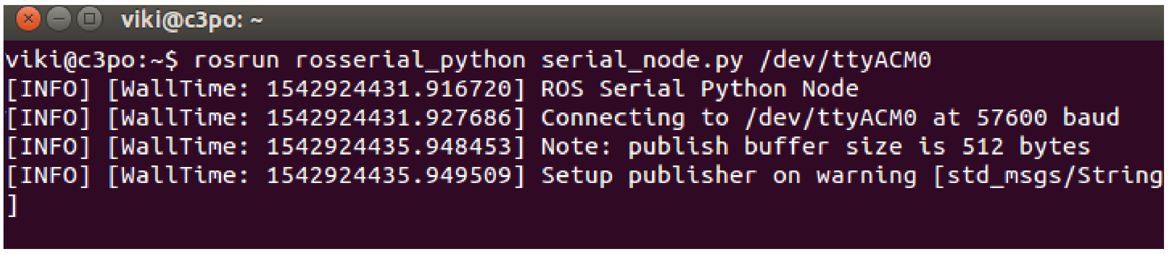
```
//In order to use the rosserial libraries in your own code you
//must first put ros.h heading
//prior to any other header file.
#include <ros.h>
#include <std_msgs/String.h>

//we need to instantiate the node handle, which allows our
//program to create publishers
// and subscribers. The node handle also takes care of serial
//port communications
ros::NodeHandle nodeHandle;
std_msgs::String msg;

//We need to instantiate the publishers and subscribers that
//we will be using.
//Here we instantiate a Publisher with a topic name of "
//warning ". The second
//parameter to Publisher is a reference to the message
//instance to be used.
ros::Publisher pub(“warning”, &msg);

char warning[26] = “This is your last warning!";

void setup() {
    //you then need to initialize your ROS node handle
    nodeHandle.initNode();
    // advertise any topics being published
    nodeHandle.advertise(pub);
}
void loop()
{
    //the node publishes the message "This is your last
    //warning!"
    msg.data= warning;
    pub.publish(&msg);
    //we call the spinOnce() function where all of the ROS
    //communication callbacks are handled
    nodeHandle.spinOnce();
    delay(1000);
}
```

In the first two lines we include the important headers that we are going to use in our program. The ros.h header must always be the first line. After this we will create our node handler. With this handler we will create publishers and subscribers to our node. (ros::NodeHandlenodeHandle). Next we create a publisher with a topic name of “warning”. In the setup part we will advertise the topic we want to publish. The initNode initializes the node handle. In the loop function the publisher publishes his message. On our Ubuntu machine we launch 3 terminals. In the first terminal we start the roscore with the command **roscore**. Next we run the **rosserial client** that forwards the messages to the rest of ROS nodes. And finally we show the message published to the topic with the command **rostopic echo warning.**



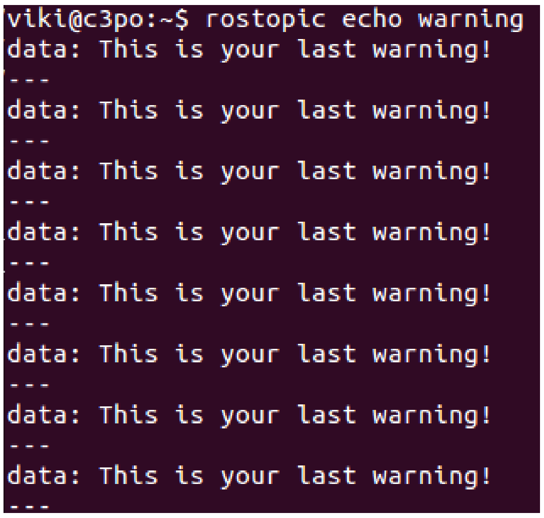
```
roscore http://c3po:11311/
viki@c3po:~$ roscore
... logging to /home/viki/.ros/log/39c5dfc8-eea2-11e8-9c77-080027b46cdd/roslaunc
h-c3po-2495.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://c3po:37422/
ros_comm version 1.11.8


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [2507]
ROS_MASTER_URI=http://c3po:11311/

setting /run_id to 39c5dfc8-eea2-11e8-9c77-080027b46cdd
process[rosout-1]: started with pid [2520]
started core service [/rosout]
```

Fig.A2.6. Running the roscore command in the linux terminal



```
viki@c3po: ~
viki@c3po:~$ rosrun rosserial_python serial_node.py /dev/ttyACM0
[INFO] [WallTime: 1542924431.916720] ROS Serial Python Node
[INFO] [WallTime: 1542924431.927686] Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [WallTime: 1542924435.948453] Note: publish buffer size is 512 bytes
[INFO] [WallTime: 1542924435.949509] Setup publisher on warning [std_msgs/String
]
```

Fig.A2.7. Instruction for running a ROS client in linux



```
viki@c3po:~$ rostopic echo warning
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
data: This is your last warning!
---
```

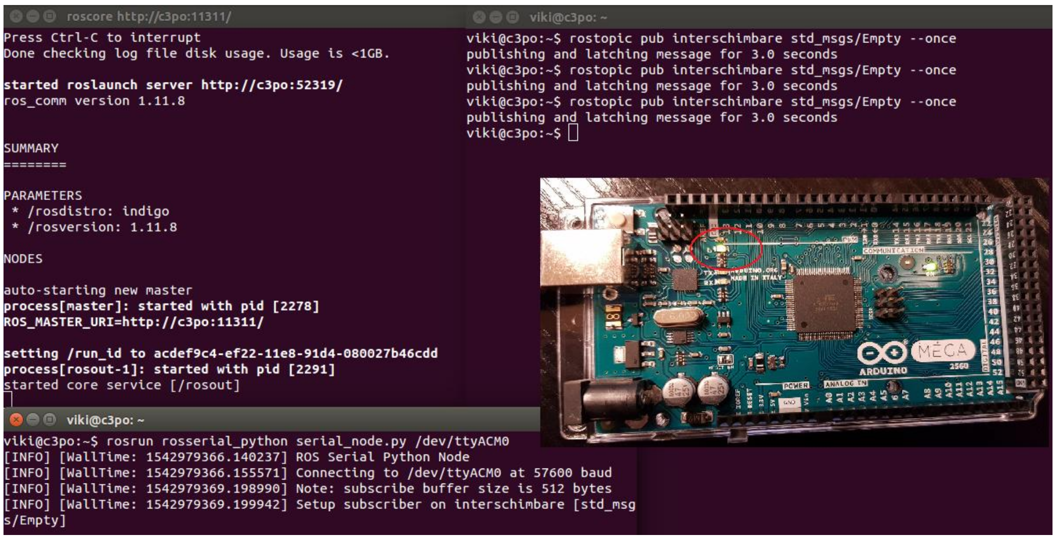
Fig.A2.8. Displaying the message from a topic

Example 2: Subscriber

In the subscriber example we will make a simple subscriber example which will toggle a led each time it will receive a message from a publisher. This example can also be found on the official Arduino ROS website and in the examples of the ros_lib Arduino library.

```
#include <ros.h>
#include <std_msgs/Empty.h>

ros::NodeHandle nh;

//We create the callback function for our subscriber. The call
//back function must take a
//constant reference of a message as its argument. In our
//callback callbackFunction,
//the type of message is std_msgs::Empty and the message name
//will be toggle_msg.
void callbackFunction( const std_msgs::Empty& toggle_msg)
{
    digitalWrite(13, HIGH-digitalRead(13));
}

//Here we instantiate a Subscriber with a topic name of
//"interchange " and type
//std_msgs::Empty. With Subscribers, you must remember to
//template the subscriber
//upon the message. Its two arguments are the topic it will be
//subscribing to and the
//callback function it will be using.
ros::Subscriber<std_msgs::Empty> sub(“interchange”,
&callbackFunction);

void setup()
{
    pinMode(13, OUTPUT);
    nh.initNode();
    //we subscribe to the topics we wish to listen to
    nh.subscribe(sub);
}

void loop()
{
    nh.spinOnce();
    delay(1);
}
```

A callback function is a function that is passed as an argument to another function. This function will be automatically called when an event is triggered. This function can be called from a low level software layer. The call back function must take a constant reference of a message as its argument.



```
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://c3po:52319/
ros_comm version 1.11.8


SUMMARY
========

PARAMETERS
 * /rosdistro: indigo
 * /rosversion: 1.11.8

NODES

auto-starting new master
process[master]: started with pid [2278]
ROS_MASTER_URI=http://c3po:11311/

setting /run_id to acdef9c4-ef22-11e8-91d4-080027b46cdd
process[rosout-1]: started with pid [2291]
started core service [/rosout]
```

```
viki@c3po:~$ rostopic pub interschimbare std_msgs/Empty --once
publishing and latching message for 3.0 seconds
viki@c3po:~$ rostopic pub interschimbare std_msgs/Empty --once
publishing and latching message for 3.0 seconds
viki@c3po:~$ rostopic pub interschimbare std_msgs/Empty --once
publishing and latching message for 3.0 seconds
viki@c3po:~$
```

```
viki@c3po:~$ rosrun rosserial_python serial_node.py /dev/ttyACM0
[INFO] [WallTime: 1542979366.140237] ROS Serial Python Node
[INFO] [WallTime: 1542979366.155571] Connecting to /dev/ttyACM0 at 57600 baud
[INFO] [WallTime: 1542979369.198990] Note: subscribe buffer size is 512 bytes
[INFO] [WallTime: 1542979369.199942] Setup subscriber on interschimbare [std_msg
s/Empty]
```

Fig.A2.9. Commands and result used in the subscriber example

Example 3: Publisher IR

In the last example we will attach a sharp IR distance sensor to our microcontroller and plot a graph corresponding to the range using rqt_plot. For this project we will need a sharp IR sensor, an Arduino board and an USB cable. This is a modified version of the usage example of IR sensor from the ros_lib examples.

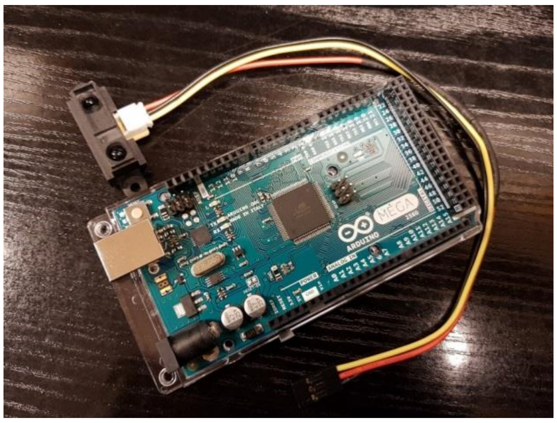


Fig.A2.10. Arduino board and IR distance sensor used in the publisher example

```
//the code begins by including the appropriate message headers
and ros.h
//from the rosserial library and then instantiating the
publisher
#include <ros.h>
#include <ros/time.h>
#include <sensor_msgs/Range.h>

ros::NodeHandle node;
sensor_msgs::Range msg;
ros::Publisher pub_range( “Sharp", &range_msg);
const int analog_pin = 0;
unsigned long range_timer;

//we write a function that gets the distance from a Sharp
//sensor connected to an analog
//pin pin_num and then filters it, by making an average over
//the acquired number
//of measurements
float getRange(int pin_num)
{
    int count = 10;
    int sum = 0;
    for (int i = 0; i<count; i++)
    {
        float volts = analogRead(pin_num) * ((float) 5 /
1024);
        float distance = 65 * pow(volts, -1.10);
        sum = sum + distance;
        delay(5);
    }
    sum = (int) (sum/count);
    return (sum -1)/100;
}
//a global variable for the sensors frame id string. It is
//important to make this string
//global so it will be alive for as long as the message will
//be in use.
char frameid[] = "/ir_ranger";

void setup()
{
    //we initialize the node handle
    node.initNode();
    node.advertise(pub_range);
    //we fill in the message fields
    range_msg.radiation_type = sensor_msgs::Range::INFRARED;
    range_msg.header.frame_id = frameid;
    //sensor specification data
    range_msg.field_of_view = 0.8;
    range_msg.min_range = 1;
    range_msg.max_range = 8;
}
void loop()
{
    //we publish the readings every 50 ms
    if ( (millis()-range_timer) > 50)
    {
        range_msg.range = getRange(analog_pin);
        range_msg.header.stamp = node.now();
        pub_range.publish(&range_msg);
        range_timer = millis();
    }
    node.spinOnce();
}
```

The application is run as in the previous examples:

1. roscore
2. rosrun rosserial_python serial_node.py _port:=/dev/ttyACM0

We plot the data that we receive using the function rqt_plot:

rqt_plot Sharp/rang

In case there is no object in the sensors range the value from the sensor fluctuates very much whereas if there is an object we get a smooth value in a range 0 and 1(the value we receive from the sensor has been scaled). We have varied the distance to the sharp sensor, obtaining the plot bellow.

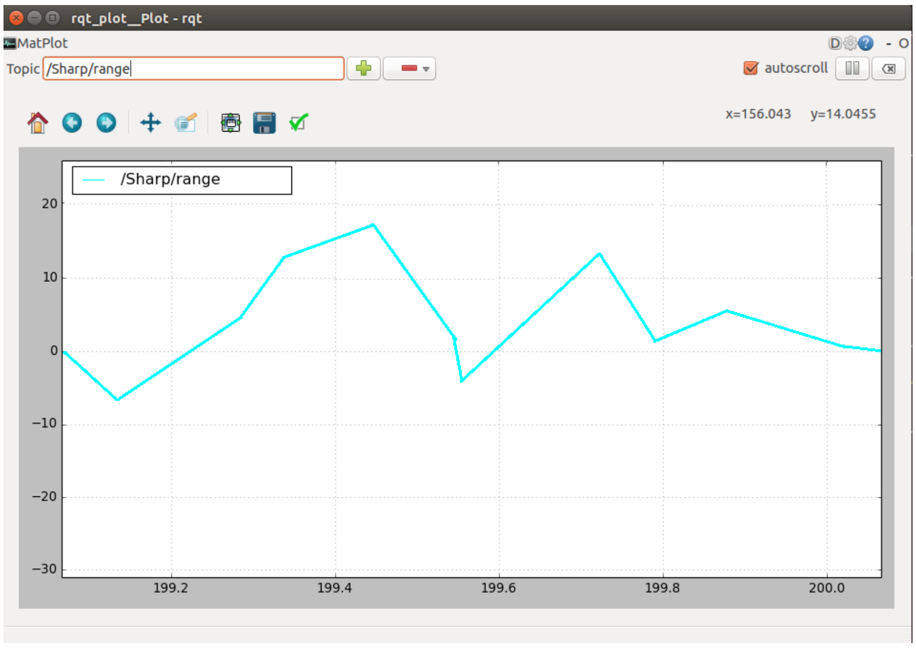


Fig.A2.11. Graph ploted using the data from the IR sensor

Individual Work

1. Test and run the examples provided

2. For more examples regarding the ROS library reffer to http://wiki.ros.org/ROS/Tutorials

3. Implement an intruder detection system by processing the information from the IR distance sensor. The detector should be activated and de-activated from a publisher, and should publish an alarm message when the intruder is detected.

Bibliography:

1. JasonM. O'Kane, "A Gentle Introduction to ROS"
2. Jonathan Bohren, "ROS Crash-Course (Part I) ", The Johns Hopkins University
3. Murilo Fernandes Martins, "PhD", Department of Electrical Engineering, FEI University Centre
4. Action and Perception (RAP) Group, "Robotics Operation System"**,** Universidad Veracruzana, Research Center on Artificial Intelligence LAAS-CNRSRobotics
5. "ROS 0-60A Comprehensive tutorial ofthe Robot Operating System"
6.http://wiki.ros.org/
7.https://nootrix.com/downloads/?fbclid=IwAR18YLs05vOQ5OG4DZcDIMK5oSDASWb1BtGlfhanjlKwICXBOaL3KGGt8p4#RosVM
8.http://wiki.ros.org/rosserial_arduino/Tutorials/Arduino%20IDE%20Setup