Ministerul Educaţiei şi Tineretului al Republicii Moldova

Universitatea Tehnică a Moldovei

Departament „Informatica aplicată”

RAPORT

Lucrarea de laborator nr.2

LA DISCIPLINA”Programarea aplicațiilor incorporate și independente de platformă ”

A efectuat:

St. gr. FAF 141 Bega Valeria

A verificat:

Lect . supp Bragarenco Andrei

Chişinău 2017

Topic: General Purpose Input/Output registers on AVR.

Objectives:

- Learn and understand GPIO
- Connecting LED
- Connecting Button

Task:

Create a LED blinking program for AVR Atmega32 Microcontroler using Proteus and Atmel Studio

Overview of the work:

1. GPIO

General-purpose input/output (GPIO) is a generic pin on an integrated circuit or computer board whose behavior—including whether it is an input or output pin—is controllable by the user at run time.

AVR is 8 bit microcontroller. All its ports are 8 bit wide. Every port has 3 registers associated with it each one with 8 bits. Every bit in those registers configure pins of particular port. Bit0 of these registers is associated with Pin0 of the port, Bit1 of these registers is associated with Pin1 of the port, …. and like wise for other bits.

These three registers are as follows : (x can be replaced by A,B,C,D as per the AVR you are using)

- DDRx register
- PORTx register
- PINx register

DDRx register

DDRx (Data Direction Register) configures data direction of port pins. Means its setting determines whether port pins will be used for input or output. Writing 0 to a bit in DDRx makes corresponding port pin as input, while writing 1 to a bit in DDRx makes corresponding port pin as output.

Example:

to make all pins of port A as input pins :
DDRA = 0b00000000;

to make all pins of port A as output pins :
DDRA = 0b11111111;

to make lower nibble of port B as output and higher nibble as input :
DDRB = 0b00001111;

PINx register

PINx (Port IN) used to read data from port pins. In order to read the data from port pin, first you have to change port's data direction to input. This is done by setting bits in DDRx to zero. If port is made output, then reading PINx register will give you data that has been output on port pins.

Now there are two input modes. Either you can use port pins as tri stated inputs or you can activate internal pull up. It will be explained shortly.

To read data from port A.

DDRA = 0x00 //set port for input

X = PINA // input

PORTx register

PORTx is used for two purposes.

1) To output data : when port is configured as output

2) To activate/deactivate pull up resistors – when port is configures as input

To output data

When you set bits in DDRx to 1, corresponding pins becomes output pins. Now you can write data into respective bits in PORTx register. This will immediately change state of output pins according to data you have written.

In other words to output data on to port pins, you have to write it into PORTx register. However do not forget to set data direction as output.

Example :

To output 0xFF data on port b

```
DDRB = 0b11111111;      //set all pins of port b as outputs
PORTB = 0xFF;           //write data on port
```

To output data in variable x on port a

```
DDRA = 0xFF;            //make port A as output
PORTA = x;              //output variable on port
```

To output data on only 0th bit of port c

```
DDRC.0 = 1;       //set only 0th pin of port c as output
PORTC.0 = 1;1     //make it high.
```

To activate/deactivate pull up resistors

When you set bits in DDRx to 0, i.e. make port pins as inputs, then corresponding bits in PORTx register are used to activate/deactivate pull-up registers associated

with that pin. In order to activate pull-up resister, set bit in PORTx to 1, and to deactivate (i.e to make port pin tri stated) set it to 0.

In input mode, when pull1-up is enabled, default state of pin becomes '1'. So even if you don't connect anything to pin and if you try to read it, it will read as 1. Now, when you externally drive that pin to zero(i.e. connect to ground / or pull-down), only then it will be read as 0.

However, if you configure pin as tri state. Then pin goes into state of high impedance. We can say, it is now simply connected to input of some OpAmp inside the uC and no other circuit is driving it from uC. Thus pin has very high impedance. In this case, if pin is left floating (i.e. kept unconnected) then even small static charge present on surrounding objects can change logic state of pin. If you try to read corresponding bit in pin register, its state cannot be predicted. This may cause your program to go haywire, if it depends on input from that particular pin.

Thus while, taking inputs from pins / using micro-switches to take input, always enable pull-up resistors on input pins.

NOTE : while using on chip ADC, ADC port pins must be configured as tri stated input.

Example :

To make port a as input with pull-ups enabled and read data from port a

```
DDRA = 0x00;       //make port a as input
PORTA = 0xFF;      //enable all pull-ups
y = PINA;          //read data from port a pins
```

To make port b as tri stated input

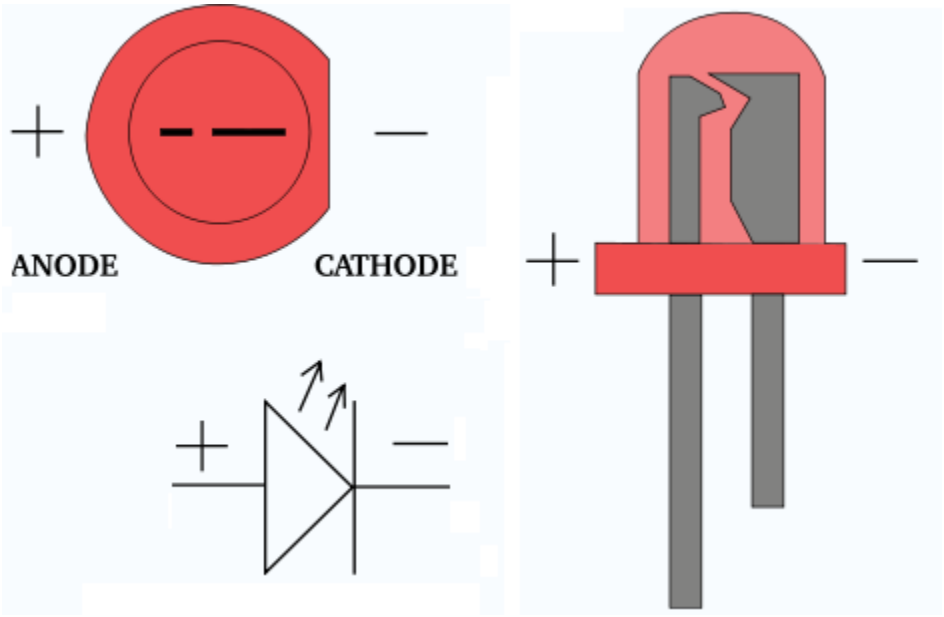
```
DDRB  = 0x00;      //make port b as input
PORTB = 0x00;      //disable pull-ups and make it tri state
```

To make lower nibble of port a as output, higher nibble as input with pull-ups enabled

```
DDRA  = 0x0F;      //lower nib> output, higher nib> input
PORTA = 0xF0;      //lower nib> set output pins to 0, higher nib> enable pull-ups
```

2. LED

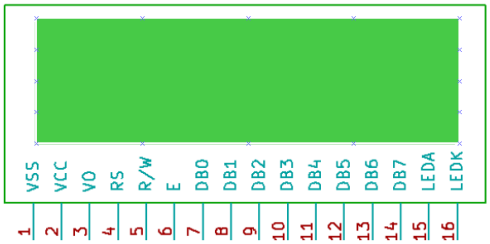
A light-emitting diode (LED) is a two-lead semiconductor light source. It is a p–n junction diode, which emits light when activated.[4] When a suitable voltage is applied to the leads, electrons are able to recombine with electron holes within the device, releasing energy in the form of photons. This effect is called electroluminescence, and the color of the light (corresponding to the energy of the photon) is determined by the energy band gap of the semiconductor.

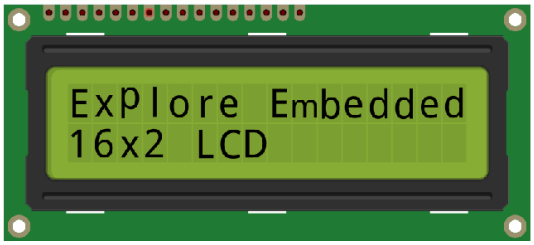


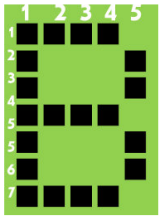
3. LCD display LM016L

The most commonly used LCDs found in the market today are 1 Line, 2 Line or 4 Line LCDs which have only 1 controller and support at most of 80 characters, whereas LCDs supporting more than 80 characters make use of 2 HD44780 controllers.

Most LCDs with 1 controller has 14 Pins and LCDs with 2 controller has 16 Pins (two pins are extra in both for back-light LED connections). Pin description is shown in the table below.







Pin description:

• VCC - Digital supply voltage.

• GND - Ground.

• Port A (PA7..PA0) - Port A serves as the analog inputs to the A/D Converter. Port A also serves as an 8-bit bi-directional I/O port, if the A/D Converter is not used. Port pins can provide internal pull-up resistors (selected for each bit). The Port A output buffers have symmetrical drive characteristics with both high sink and source capability. When pins PA0 to PA7 are used as inputs and are externally pulled low, they will source current if the internal pull-up resistors are activated. The Port A pins are tri-stated when a reset condition becomes active,even if the clock is not running.

• Port B (PB7..PB0) - Port B is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port B output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port B pins that are externally pulled low will source current if the pull-up resistors are activated. The Port B pins are tri-stated when a reset condition becomes active, even if the clock is not running.

• Port C (PC7..PC0) - Port C is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port C output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port C pins that are externally pulled low will source current if the pull-up resistors are activated. The Port C pins are tri-stated when a reset condition becomes active, even if the clock is not running. If the JTAG interface is enabled, the pull-up resistors on pins PC5(TDI), PC3(TMS) and PC2(TCK) will be activated even if a reset occurs. The TD0 pin is tri-stated unless TAP states that shift out data are entered.

• Port D (PD7..PD0) - Port D is an 8-bit bi-directional I/O port with internal pull-up resistors (selected for each bit). The Port D output buffers have symmetrical drive characteristics with both high sink and source capability. As inputs, Port D pins that are externally pulled low will source current if the pull-up resistors are activated. The Port D pins are tri-stated when a reset condition becomes active, even if the clock is not running.

• RESET - Reset Input. A low level on this pin for longer than the minimum pulse length will generate a reset, even if the clock is not running.

• XTAL1 - Input to the inverting Oscillator amplifier and input to the internal clock operating circuit.

• XTAL2 - Output from the inverting Oscillator amplifier.

• AVCC - AVCC is the supply voltage pin for Port A and the A/D Converter. It should be externally connected to VCC, even if the ADC is not used. If the ADC is used, it should be connected to VCC through a low-pass filter.

• AREF AREF is the analog reference pin for the A/D Converter.

Solution:

We should define drivers for a button and for LED, since we work with them in this laboratory work.In order to use the standard functions like printf() from C we have to implement 3 basic function ourselves. The first one is to initialize the terminal there's another to put a character to the terminal and one to get a character from the terminal. In the next code are 3 functions. They allow the user to interact with the terminal. To better understand how the code works I will present several code flow diagrams. Below are the flow diagrams of the main.c file. These are the 3 main functions.

LED Driver

LED driver has dependencies on:

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

It has MACRO definition for registers which makes our driver to work on and other devices.

```
struct LedDevice {
	uint8_t pinIndex;
	volatile uint8_t *ddr;
	volatile uint8_t *port;
};
```

This structure contains needed data for driver(configuration),, in order to work with an LED Device. This was done for using more than one LED on our Embedded System.

```
void LedInit(struct LedDevice *device);
void LedOn(struct LedDevice *device);
void LedOff(struct LedDevice *device);
```

Each function from this list has it own responsibility :

- Initialization
- Turn on LED
- Turn off LED

Every function has parameter of LED configuration.

Button Driver

Button driver has dependencies on.

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

It has MACRO definition for registers which makes our driver to work on and other devices.

```
struct ButtonDevice {
	uint8_t pinIndex;
	volatile uint8_t *pin;
	volatile uint8_t *ddr;
};
```

This structure contains needed data for driver(configuration), in order to work with an Button Device. This was done for using more than one LED on our Embedded System.

```
void ButtonInit(struct ButtonDevice *device);
char ButtonPressed(struct ButtonDevice *device);
```

Each function from this list has it own responsibility :

- Initialization
- Checking button state

Every function has parameter of Button configuration.

LCD Driver

LCD driver is more complex and was downloaded from extremeelectronics.co.in.

For more information please check author site.

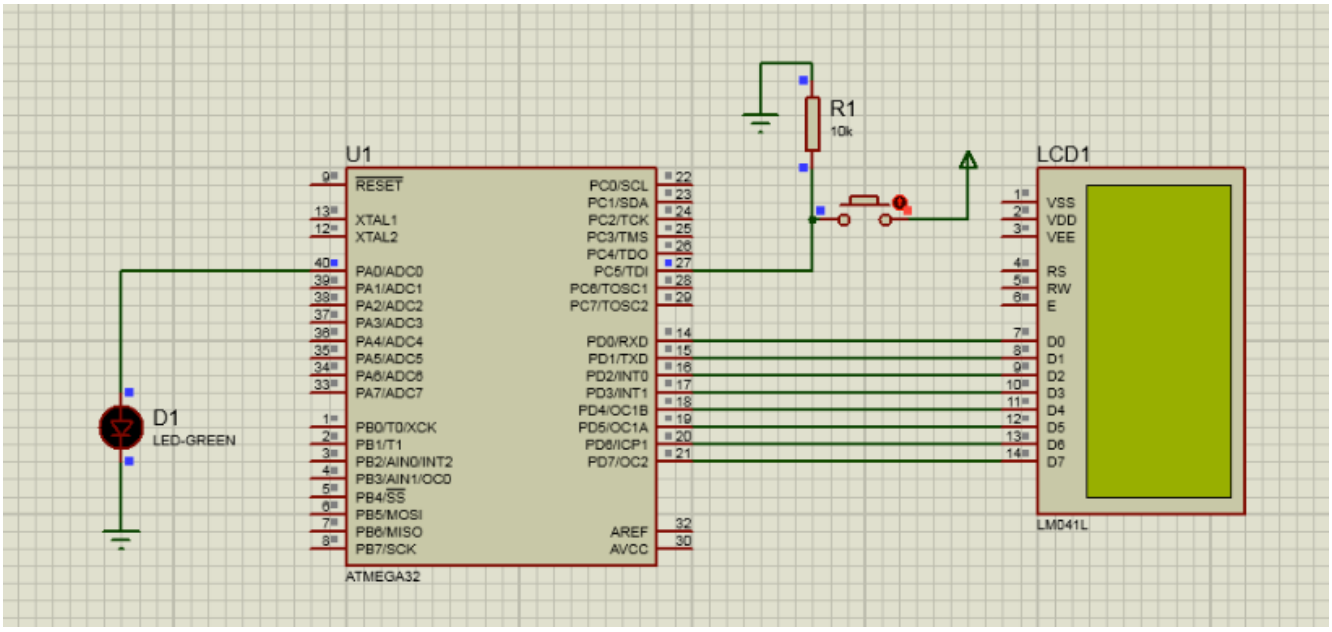
Main Program

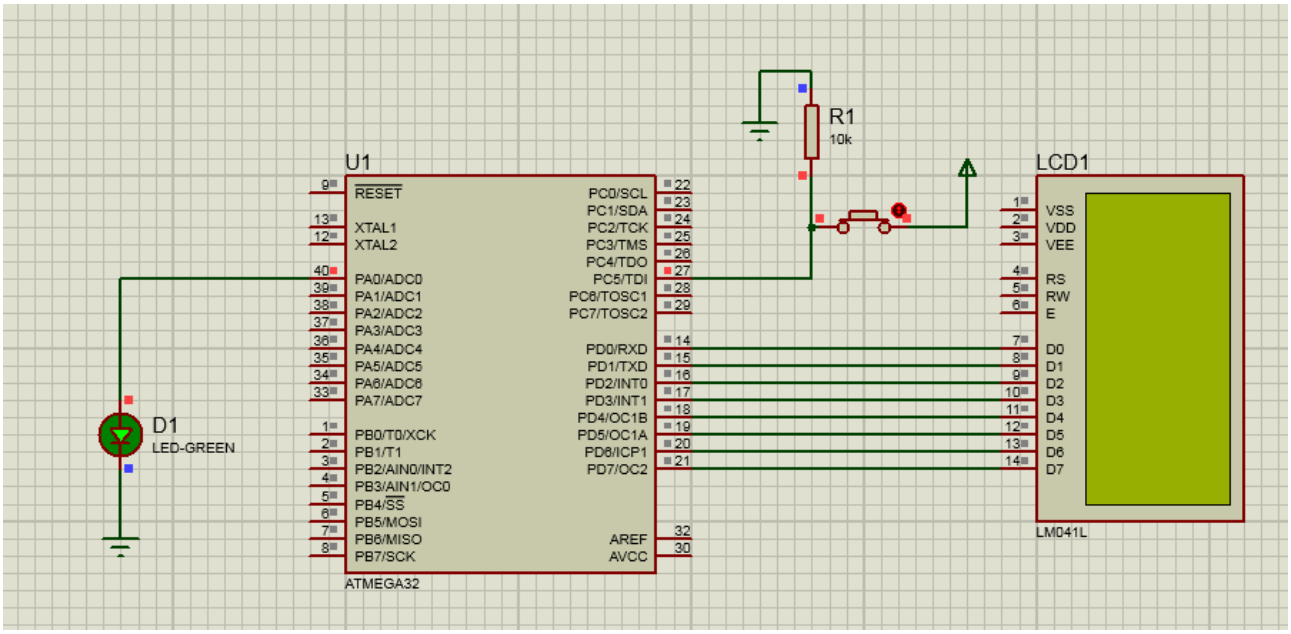
Basically our main program uses High Level Drivers.

- Initializes LED
- Initializes Button
- Start infinite loop
    a. Check if button is pressed
        -PRESSED – LED ON
        -NOT PRESSED – LED OFF
    b. sleep for 100 ms

Sleep is done with **avr/delay.h** library, which has method procedure **_delay_ms(duration)** defined.After code implementation, we should now **Build Hex** which will be written to MCU ROM.

After we finish work with implementation of the idea, we go off to Proteus, where we have to work with schemes.





Conclusion:

In this laboratory work I learned how to organize libraries in Atmel Studio, how to construct electrical schemes for MCU in Proteus and how to connect I/O device such as virtual terminal to scheme. I learned about GPIO and its purpose. The main useful thing is that we tried to connect a pheripheral to MC using ports and how to send and receive data through PIN and PORT registers.

Also, we got to practice more with schemes in Proteus. This time, there was a more difficult one. We had to know some basic physics stuff in order to make a scheme(we used resistances and stuff).

Appendix:

main.c

```
#include "led.h"

#include "uart_studio.h"

#include "button.h"

#include <avr/delay.h>
int main() {
```

```
	initButton();
	initLed();


	while(1) {

		_delay_ms(100);
		if(isPressed()) {

			ledOn();

		} else {

			ledOff();

		}

	}


	return 0;

}
```

button.h

```
#ifndef BUTTON_H_

#define BUTTON_H_

#include <avr/io.h>


int isPressed();
void initButton();


#endif /* BUTTON_H_ */
```

button.c

```
#include "button.h"
void initButton() {

	DDRC &= ~(1 << PORTC5) ;

}


int isPressed() {

	return PINC & (1<<PORTC5);

}
```

led.h

```
#ifndef LED_H_
```

```
#define LED_H_

#include <avr/io.h>


void initLed();
void ledOn();
void ledOff();


#endif /* LED_H_ */
```

led.c

```
#include "led.h"
void initLed() {

	DDRA |= (1 << PORTA0);

}


void ledOn() {

	PORTA |= (1 << PORTA0);

}


void ledOff() {

	PORTA &= ~(1 << PORTA0);

}
```