Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

RAPORT

la Programarea aplicațiilor incorporate și independente de platformă

Lucrare de laborator Nr. 2

Tema: Utilizarea regiștrilor pentru Intrare/Iesire

A efectuat st. gr. FAF-141: Oxana Dunav

A verificat: Andrei Bragarenco

Chişinău 2016

Topic

General Purpose Input/Output registers on AVR.

Purpose

- Understanding GPIO
- Connecting LED
- Connecting Button

Task

Write a C program and schematics for Microcontroller Unit (MCU) using led which will be turned on by pushing on button and turned off when button is released.

Domain

➔ GPIO

Every micro-controller has GPIO ports. GPIO stands for general purpose input output. These GPIO ports are used to take input on a micro-controller pin or output a value on micro-controller pin.

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
```

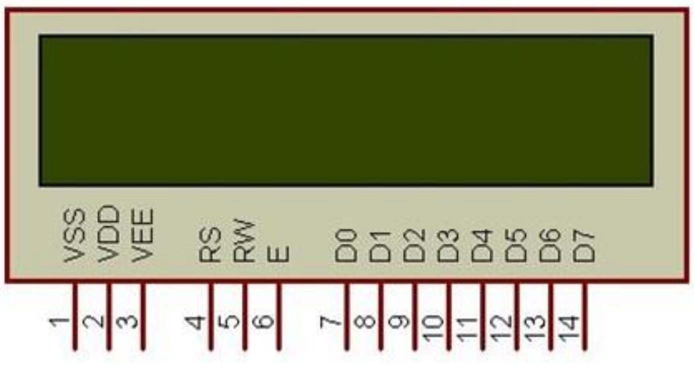
To make lower nibble of port a as output, higher nibble as input with pull-ups enabled

```
DDRA  = 0x0F;       //lower nib> output, higher nib> input

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

Used Resources

LCD Display LM016L



Dimensions: 16 character x 2 lines

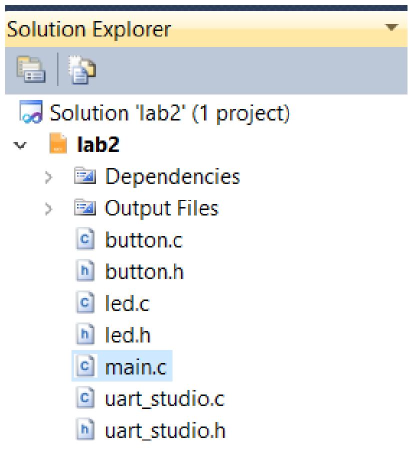
Power: +5V

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

Solution

In order to use LED and Button we should define drivers for each of them.

The *Project Structure* looks in this way


```
Solution Explorer
Solution 'lab2' (1 project)
  lab2
    Dependencies
    Output Files
    button.c
    button.h
    led.c
    led.h
    main.c
    uart_studio.c
    uart_studio.h
```

LED Driver

LED driver has dependencies on.

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

There are 3 functions for using the led:

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

Each function from this list has it own responsibility :

- Initialization
- Turn on
- Turn off

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.

There are 2 functions for using the button:

```
int isPressed();
void initButton();
```

Main Program

Our program has the following structure:

1. Initializes button
2. Initializes led
3. Start infinite loop
    a. Checks if button is pressed
        i. Pressed – Turns led on
        ii. Unpressed – Turns led off
    b. Sleep for 100 ms

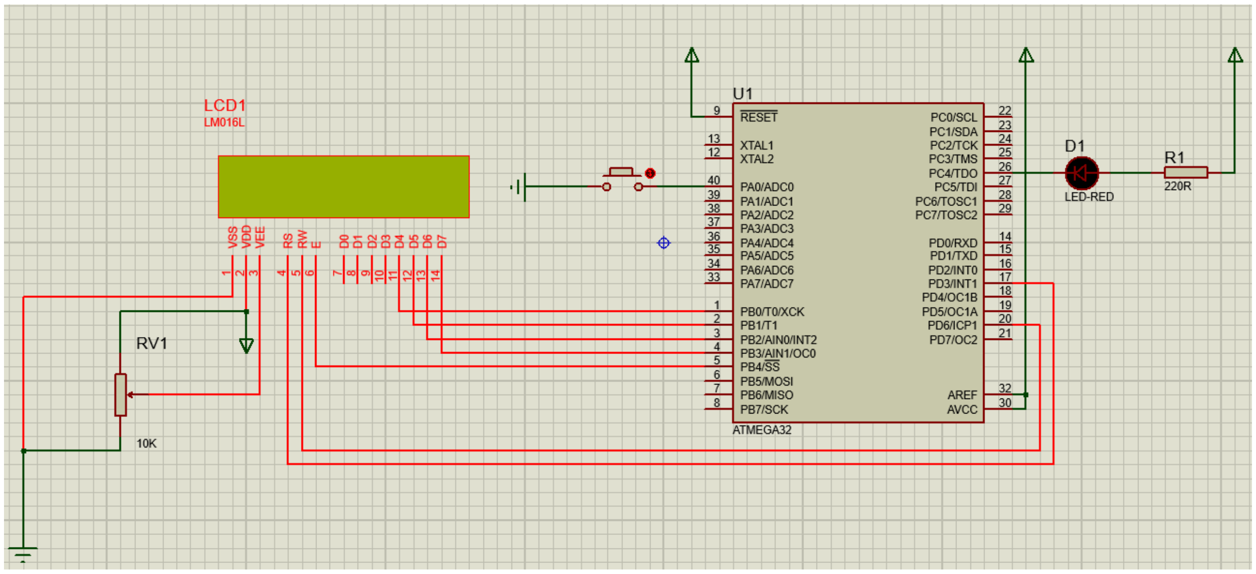
Sleep is done with **avr/delay.h** library, which has method procedure **_delay_ms(duration)** defined.

After code implementation, we should now **Build Hex** which will be written to MCU ROM.

Schematics

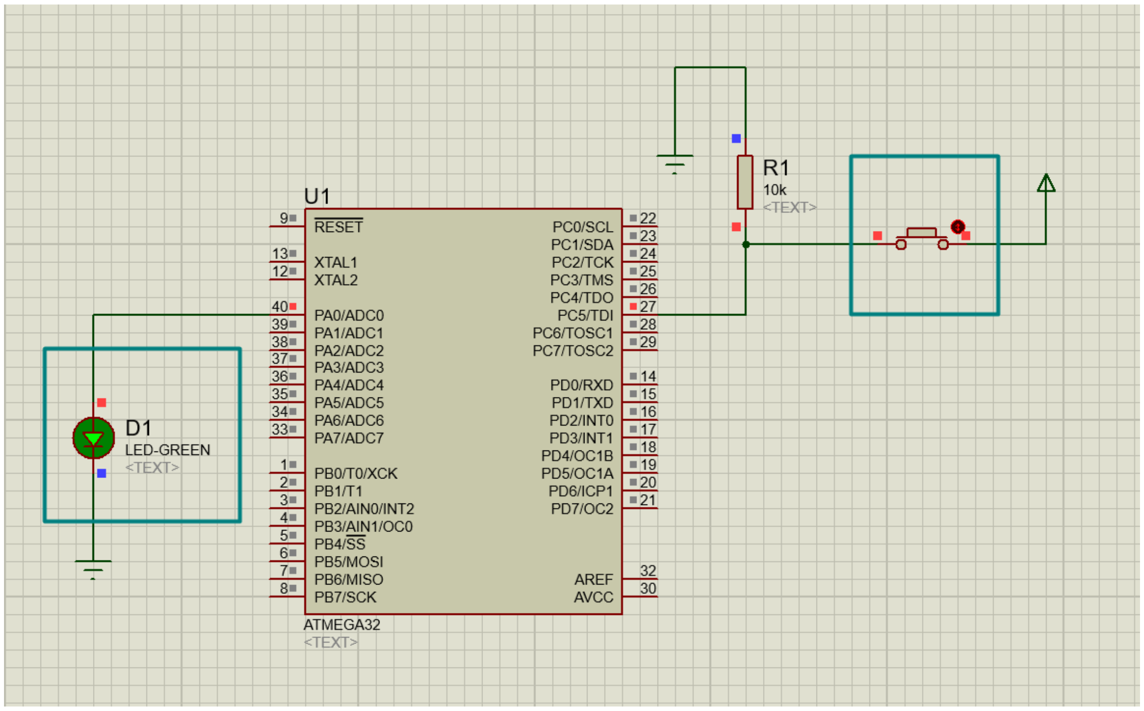
For our laboratory work we need

1. ATMega32 MCU
2. LED
3. Push button
4. LCD LM016L (16x2)

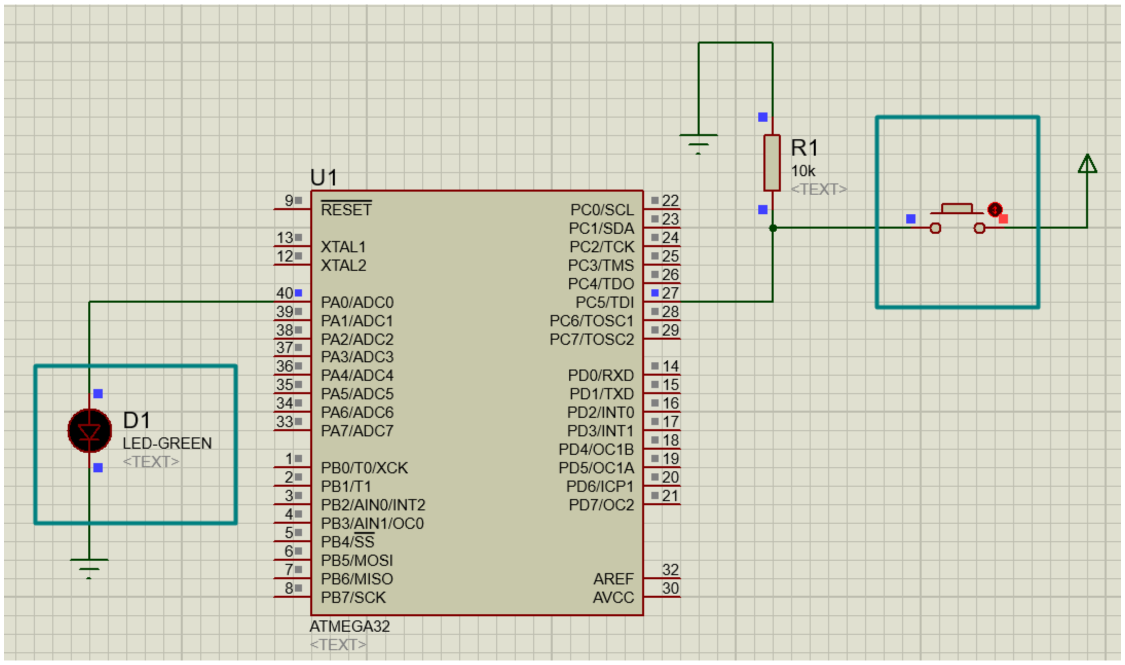


Simulation Result

Button Pressed



Button Released



Conclusion

During this laboratory work I learned about GPIO, how to connect a peripheral to MC using the ports and how to receive and send data through PIN and PORT registers.

Also, I got more practice in Proteus, as we had to use more components in the schematics. I connected a resistance to the LED because our power source is 5v, but we need 2.5V only for the led, so a resistance does this work.

By doing this laboratory work, I understand better about embedded systems, as the practice was doing a real problem and it was very interesting. It is better to understand by doing and so it helped to have a better overview about what it is.

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

```c
#ifndef LED_H_
#define LED_H_
#include <avr/io.h>

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

#endif /* LED_H_ */
```

led.c

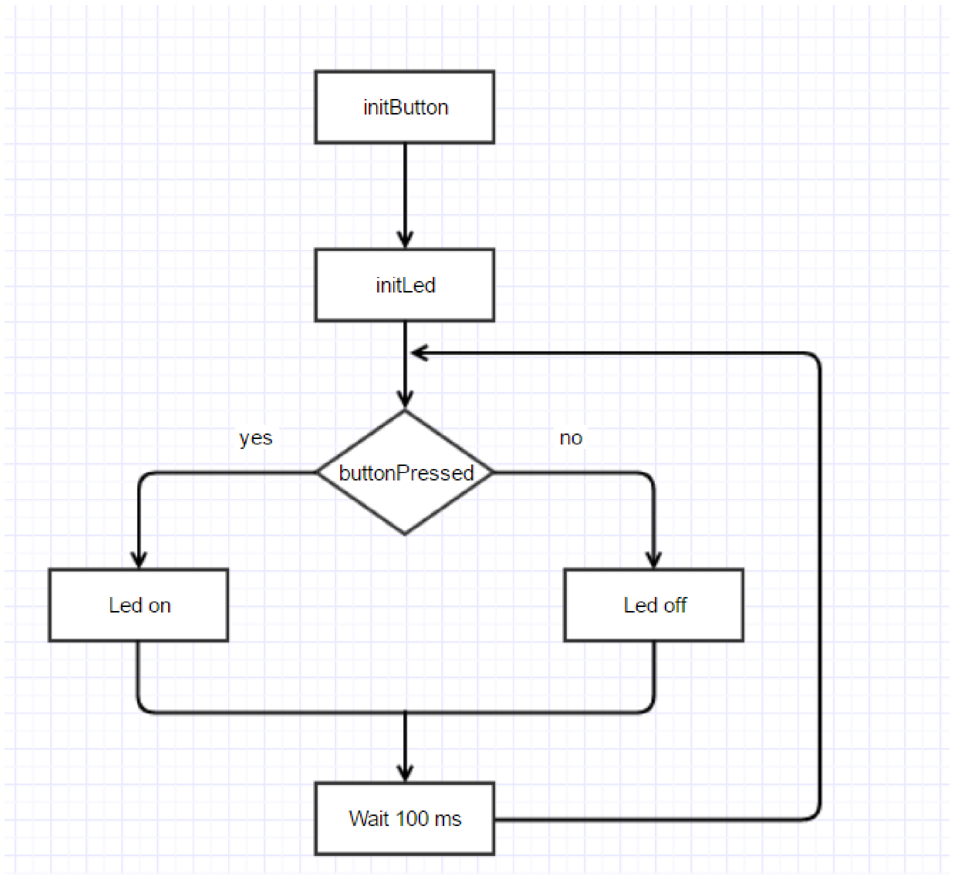
```c
#include "led.h"

void initLed() {
	DDRA |= (1 << PORTA0);
}

void ledOn() {
	PORTA |= (1 << PORTA0);
}

void ledOff() {
	PORTA &= ~(1 << PORTA0);
}
```

FlowChart


initButton
initLed
yes
buttonPressed
no
Led on
Led off
Wait 100 ms