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 registrilor pentru Intrare/Iesire”

A efectuat st. gr. FAF-141: Mihai Trofim

A verificat: Andrei Bragarenco

Chişinău 2017

Topic

General Purpose Input/Output registers on AVR.

Task

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

Additionally use LCD Display which will display current state of led.

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
```

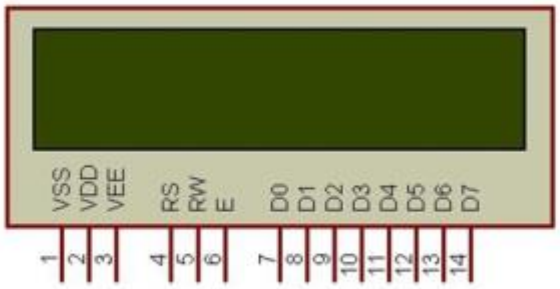
To make lower nibble of port a as output, higher nibble as input with pull-ups enabled

```
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 charracter x 2 lines

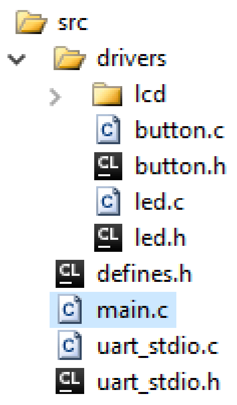
Power: +5V

In this laboratory work I used driver which works with only 4 data lines.

Solution

First of all, to make use LED,Button we should define drivers for each of them. So we will have 3 new drivers: Button,Led,LCD Display.

Generally, **Project Structure** looks in this way



```
src
  drivers
    lcd
    button.c
    button.h
    led.c
    led.h
  defines.h
  main.c
  uart_stdio.c
  uart_stdio.h
```

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.

```c
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.

```c
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

- Turn off

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

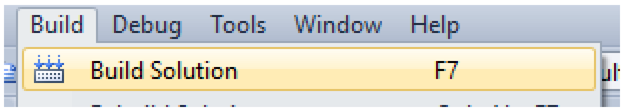
Basically our main program uses High Level Drivers.

1. Defines configuration for Led (LedDevice)
2. Defines configuration for Button (ButtonDevice)
3. Initializes led
4. Initializes button
5. Initializes LCD Display

6. Start infinite loop
    a. Checks if button is pressed
        i. Pressed – Turning on led and displaying “ON” message to LCD Display
        ii. Unpressed – Turning off led and displaying “OFF” message to LCD Display
    b. Sleep for 100 ms

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

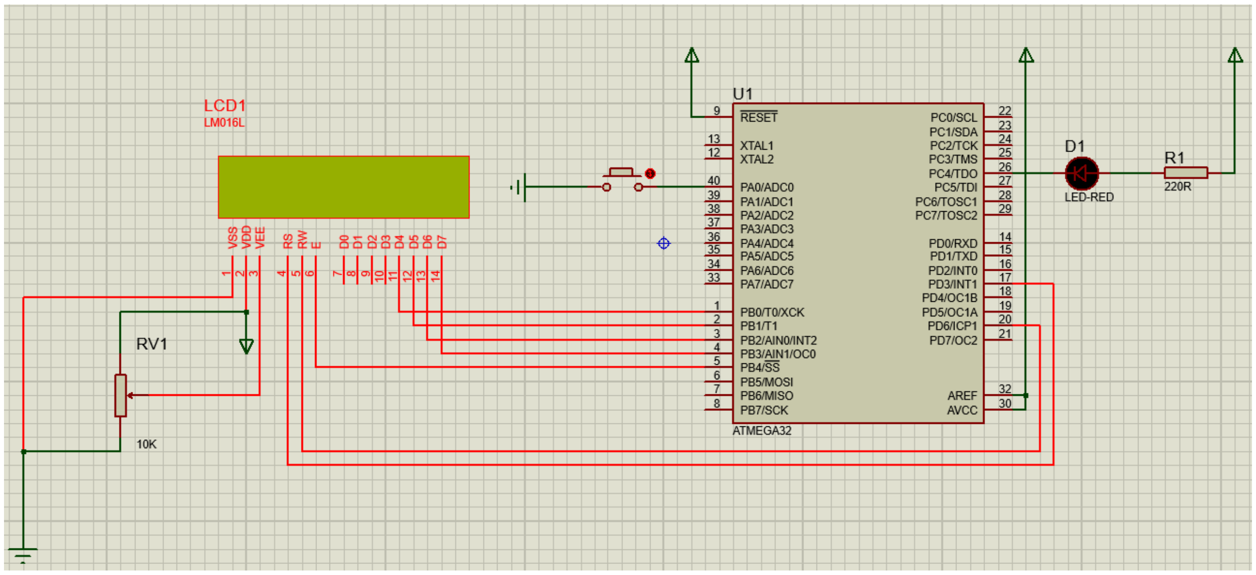
After code implementation, we should now **Build Hex** which will be written to MCU ROM.



Schematics

For our laboratory work we need

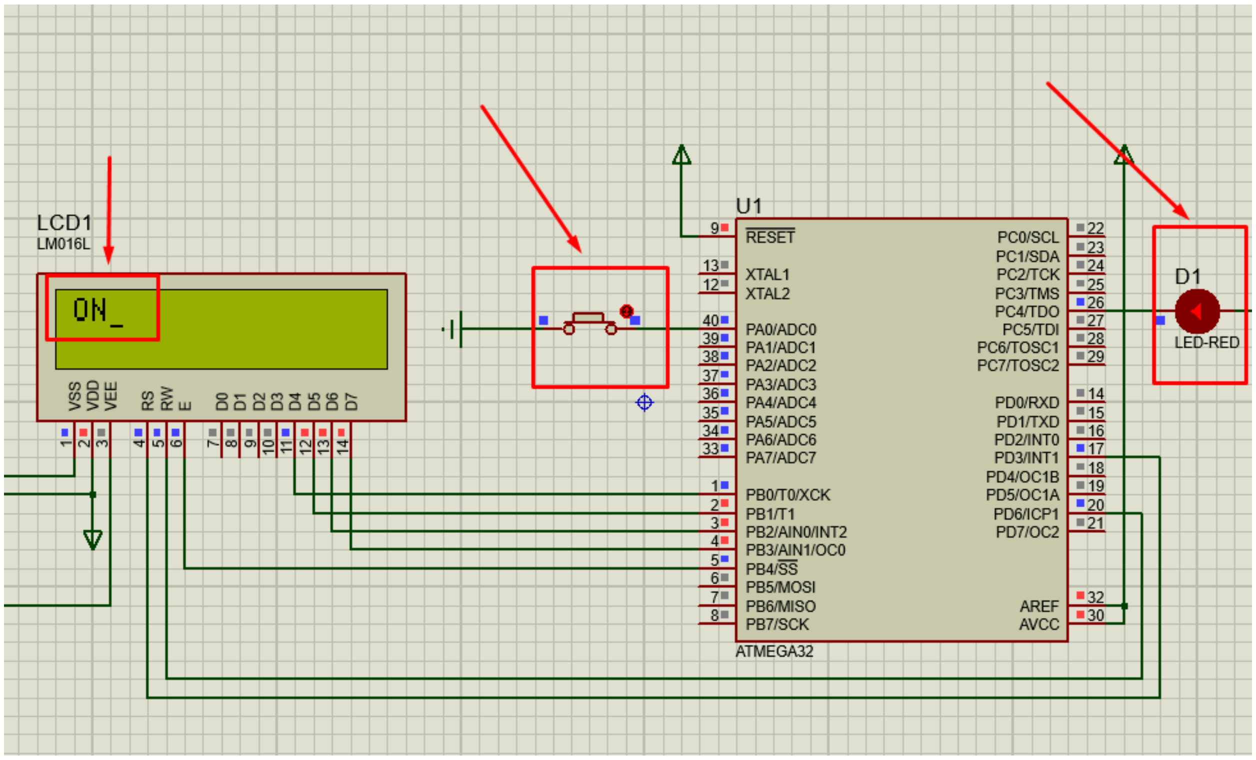
1. ATMega32 MCU
2. LED
3. Push button
4. LCD LM016L (16x2)



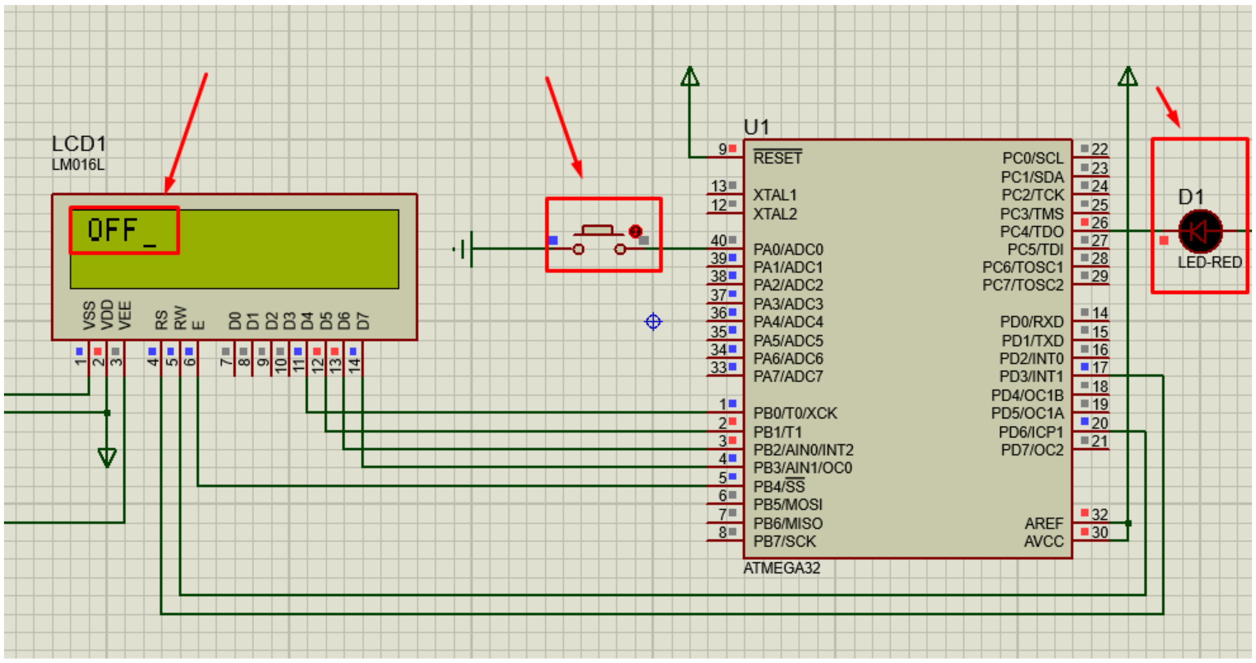
MCU is transmitter and peripheral is receiver. No vice versa connection because we don’t need it in our laboratory work.

Simulation Result

Button Pressed



Button Released



Conclusion

This laboratory work gave us basic knowledge about GPIO. We gained knowledge on following:

- how to connect a peripheral to MC using 4 Atmega Ports
- how to configure device for Input or output
- how to receive/send data through PIN/PORT registers.

Also there was some key point in this laboratory work like, connecting resistance to LED. As our power source is 5v , but we need 2.5V only for our RED Led.

Also, implementing button/led with Configuration Structure, has given us possibility to easy export these drivers and use where you want, with configuration you want.

Appendix

Main.c

```c
#include <avr/io.h>
#include <util/delay.h>
#include "drivers/button.h"
#include "drivers/led.h"
#include "drivers/lcd/lcd.h"

uint8_t i = 0;
struct ButtonDevice buttonDevice;
struct LedDevice ledDevice;

void turnOn(){
	LCDWriteString("ON");
	LedOn(&ledDevice);
}

void turnOff(){
	LCDWriteString("OFF");
	LedOff(&ledDevice);
}

void main() {
	buttonDevice.pin = &PINA;
	buttonDevice.pinIndex = 0;
	buttonDevice.ddr = &DDRA;

	ledDevice.ddr = &DDRC;
	ledDevice.pinIndex = 4;
	ledDevice.port = &PORTC;

	ButtonInit(&buttonDevice);
	LedInit(&ledDevice);
	LCDInit(LS_ULINE);

	while(1){
		LCDClear();
		if(ButtonPressed(&buttonDevice)){
			turnOn();
		} else {
			turnOff();
		}
		_delay_ms(100);
	}
}
```

Button.h

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

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

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

#endif /* BUTTON_H_ */
```

Button.c

```
#include "button.h"


void ButtonInit(struct ButtonDevice *device){
	*(device->ddr) = ~(1<<device->pinIndex);
}

char ButtonPressed(struct ButtonDevice *device){
	if(~(*(device->pin))&(1<<device->pinIndex))
		return 1;
	else
		return 0;
}
```

Led.h

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

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

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

#endif /* LED_H_ */
```

Led.c

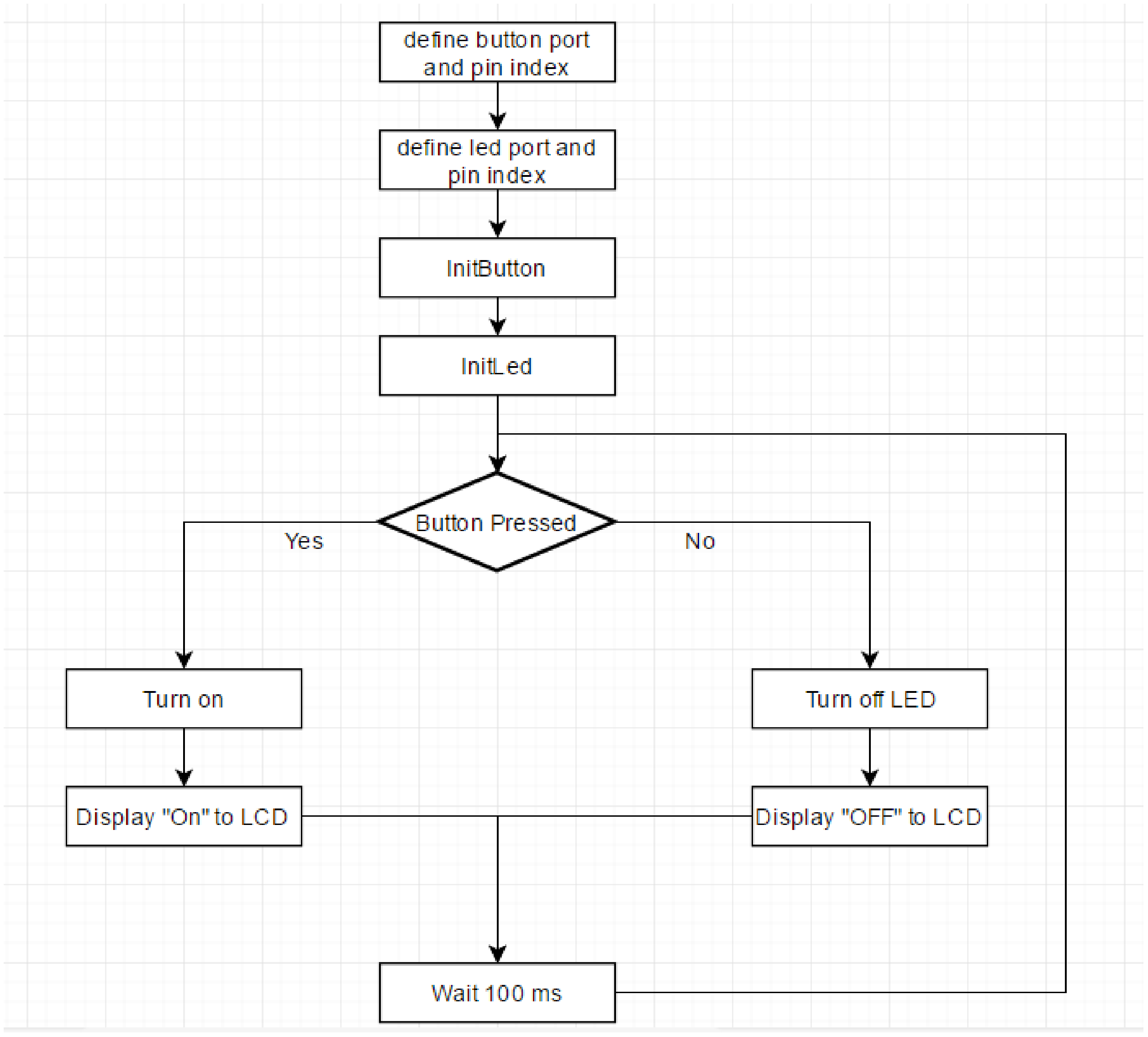
```
#include "led.h"

void LedInit(struct LedDevice *device){
       *(device->ddr) |= 1 << device->pinIndex;
}

void LedOn(struct LedDevice *device){
       *(device->port) &= ~(1<< device->pinIndex);
}

void LedOff(struct LedDevice *device){
       *(device->port) |= (1<< device->pinIndex);
}
```

FlowChart


define button port and pin index
define led port and pin index
InitButton
InitLed
Button Pressed
Yes
No
Turn on
Turn off LED
Display "On" to LCD
Display "OFF" to LCD
Wait 100 ms