Ministerul Educatiei al Republicii Moldova
Universitatea Tehnica a Moldovei
Filiera Anglofona

Report

Laboratory Nr.2

Embeded Systems

Performed By: Daniel Ciobanu

Verified By: Andrei Bragarenco

Chisinau 2017

Topic : Iput/Output registers. Work with LED and buttons.

Task: Write a simple program that connects a LED to the MCU and a button. When press the button the LED need to turn on and vice versa.

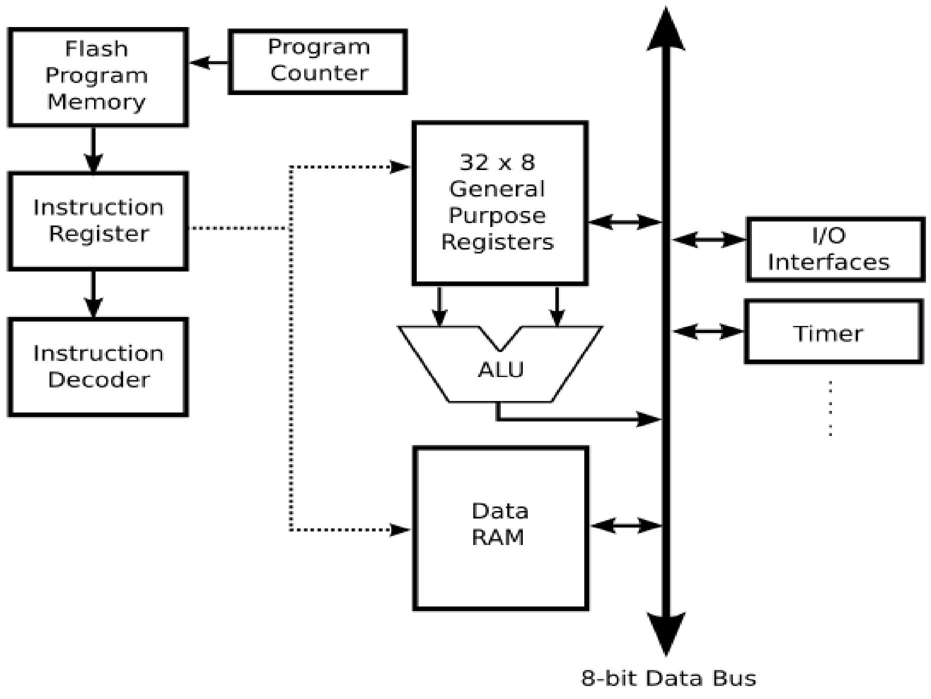
Theory:

A **microcontroller** is a self-contained system with peripherals, memory and a processor that can be used as an embedded system. Most programmable microcontrollers that are used today are embedded in other consumer products or machinery including phones, peripherals, automobiles and household appliances for computer systems. Due to that, another name for a microcontroller is "embedded controller." Some embedded systems are more sophisticated, while others have minimal requirements for memory and programming length and a low software complexity. Input and output devices include solenoids, LCD displays, relays, switches and sensors for data like humidity, temperature or light level, amongst others.

Applications for Microcontrollers:

Programmable microcontrollers are designed to be used for embedded applications, unlike microprocessors that can be found in PCs. Microcontrollers are used in automatically controlled devices including power tools, toys, implantable medical devices, office machines, engine control systems, appliances, remote controls and other types of embedded systems.

MCU Architecture:



Atmega32 AVR microcontroller :

ATmega32 is an 8-bit high performance microcontroller of Atmel's Mega AVR family. Atmega32 is based on enhanced RISC (Reduced Instruction Set Computing) architecture with 131 powerful instructions. Most of the instructions execute in one machine cycle. Atmega32 can work on a maximum frequency of 16MHz.

Atmega32 GPIO:

Registers

Atmel 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.

Example :

- *to read data from port A.*
  ```
  DDRA = 0x00;     //Set port a as input
  x = PINA;        //Read contents of port a
  ```

PORTx register

PORTx is used for two purposes.

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

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 |= 0b00000001;        //set only 0th pin of port c as output
  PORTC |= 0b00000001;       //make it high.
  ```

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

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 pull-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*

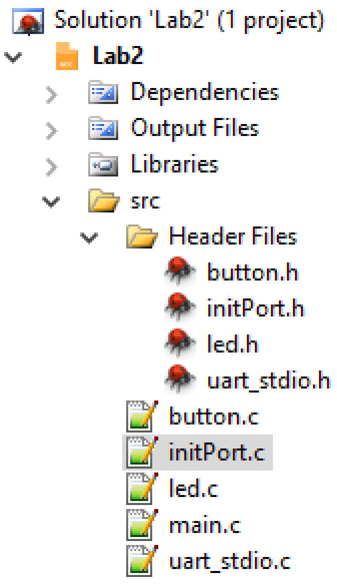
  ```
  DDRB  = 0x00;        //make port b as input
  PORTB = 0x00;        //disable pull-ups and make it tri state
  ```

- *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
  ```

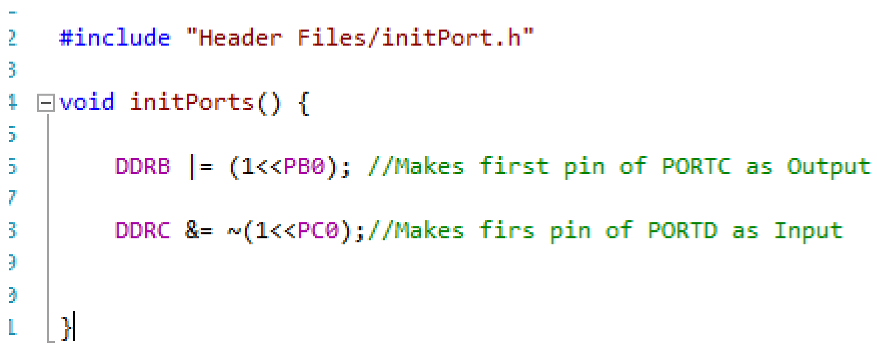
Implementation:

The structure of the laboratory 2 looks like this:



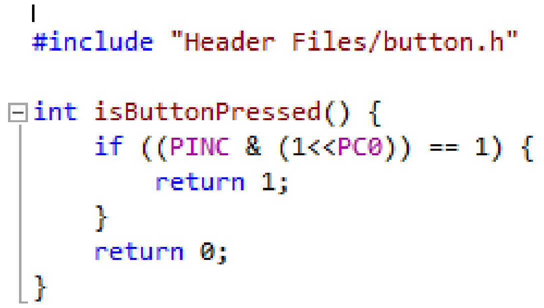
```
Solution 'Lab2' (1 project)
  Lab2
    Dependencies
    Output Files
    Libraries
    src
      Header Files
        button.h
        initPort.h
        led.h
        uart_stdio.h
      button.c
      initPort.c
      led.c
      main.c
      uart_stdio.c
```

Now first thing we do, we initialize ports, that means we need to set what port are for input and which are for output:

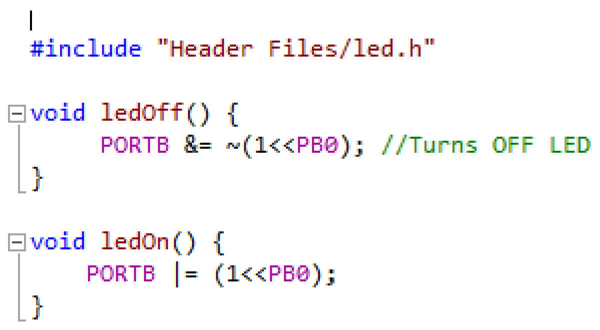


```c
#include "Header Files/initPort.h"

void initPorts() {

    DDRB |= (1<<PB0); //Makes first pin of PORTC as Output

    DDRC &= ~(1<<PC0);//Makes firs pin of PORTD as Input


}
```

Next we need to create the functions for pressing the button and for turning the led On/Off.

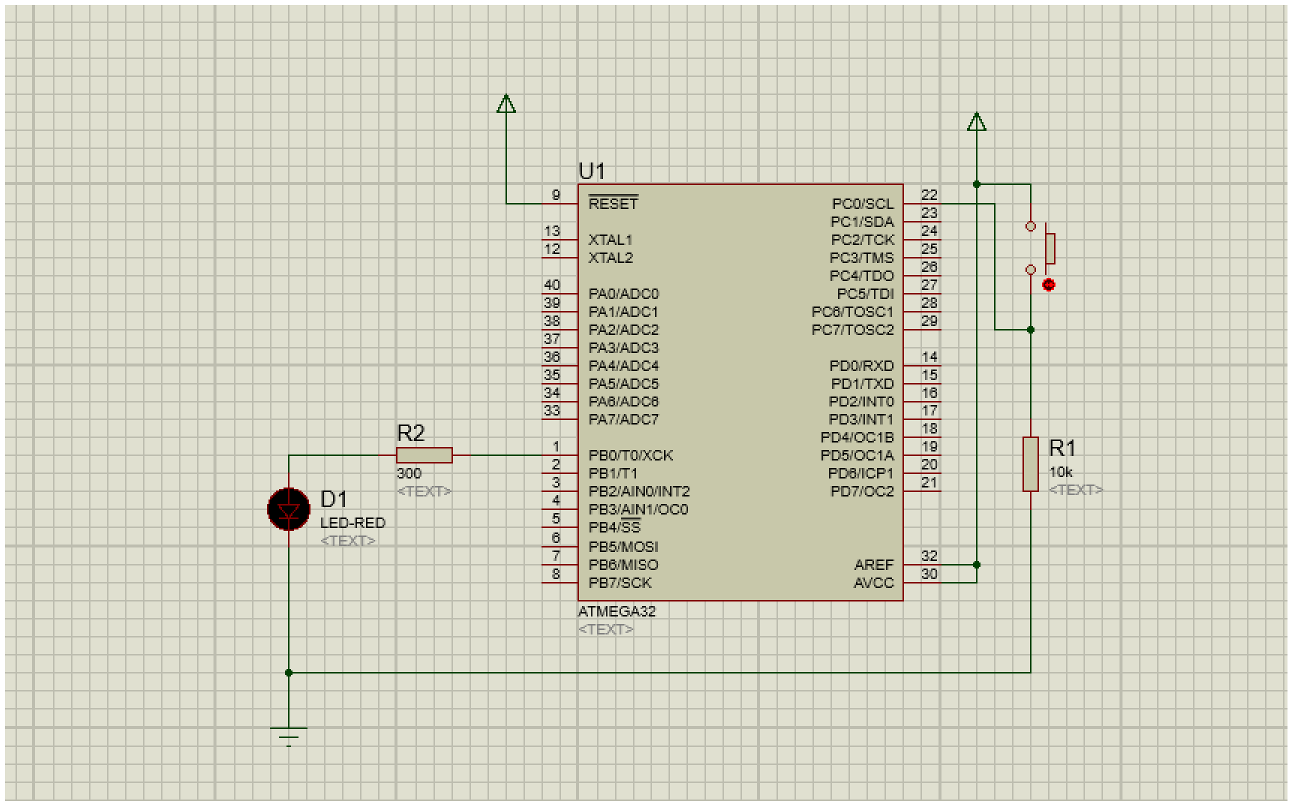


```c
#include "Header Files/button.h"

int isButtonPressed() {
    if ((PINC & (1<<PC0)) == 1) {
        return 1;
    }
    return 0;
}
```

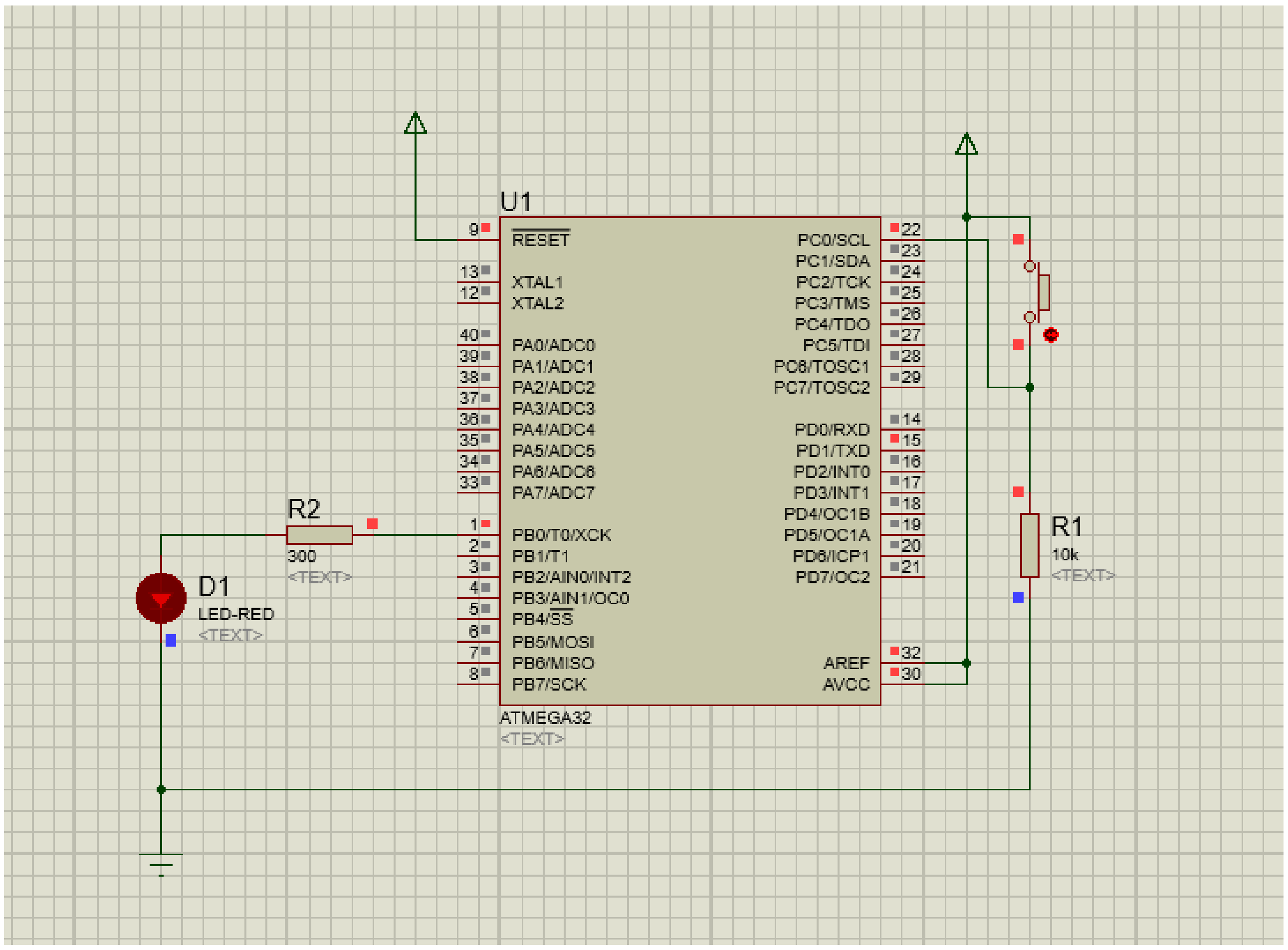


```
#include "Header Files/led.h"

void ledOff() {
     PORTB &= ~(1<<PB0); //Turns OFF LED
}

void ledOn() {
    PORTB |= (1<<PB0);
}
```

The scheme in Proteus:



The Result:



U1
RESET
XTAL1
XTAL2
PA0/ADC0
PA1/ADC1
PA2/ADC2
PA3/ADC3
PA4/ADC4
PA5/ADC5
PA6/ADC6
PA7/ADC7
PB0/T0/XCK
PB1/T1
PB2/AIN0/INT2
PB3/AIN1/OC0
PB4/SS
PB5/MOSI
PB6/MISO
PB7/SCK
PC0/SCL
PC1/SDA
PC2/TCK
PC3/TMS
PC4/TDO
PC5/TDI
PC6/TOSC1
PC7/TOSC2
PD0/RXD
PD1/TXD
PD2/INT0
PD3/INT1
PD4/OC1B
PD5/OC1A
PD6/ICP1
PD7/OC2
AREF
AVCC
ATMEGA32
<TEXT>
R2
300
<TEXT>
D1
LED-RED
<TEXT>
R1
10k
<TEXT>

Conclusion :

This laboratory work provides us with the knowledge of the input/output registers of the Atmega32 MCU. Also we used different components like LED's and buttons and how to connect them to the MCU to work properly.

Appendix:

uart_stdio.h

```
#ifndef UART_STDIO_H_
#define UART_STDIO_H_

#define F_CPU 1000000UL
#include <stdio.h>

void uart_stdio_Init(void);
int uart_PutChar(char c, FILE *stream);
char uart_ReadChar();

#endif /* UART_STDIO_H_ */
```

uart_stdio.c

```
#include "Header Files/uart_stdio.h"

#define UART_BAUD  9600

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

FILE uart_output = FDEV_SETUP_STREAM(uart_PutChar, NULL,
_FDEV_SETUP_WRITE);
FILE uart_input = FDEV_SETUP_STREAM(NULL, uart_ReadChar,
_FDEV_SETUP_READ);

void uart_stdio_Init(void) {
      stdout = &uart_output;
      stdin = &uart_input;

      #if F_CPU < 2000000UL && defined(U2X)
      UCSRA = _BV(U2X);             /* improve baud rate error by using
2x clk */
      UBRRL = (F_CPU / (8UL * UART_BAUD)) - 1;
```

```
	#else
	UBRRL = (F_CPU / (16UL * UART_BAUD)) - 1;
	#endif
	UCSRB = _BV(TXEN) | _BV(RXEN); /* tx/rx enable */
}

int uart_PutChar(char c, FILE *stream) {
	if (c == '\n')
	uart_PutChar('\r', stream);

	while (~UCSRA & (1 << UDRE));
	UDR = c;


	return 0;
}

char uart_ReadChar() {
	//Wait untill a data is available
	while(!(UCSRA & (1<<RXC)))
	{
		//Do nothing
	}
	//Now USART has got data from host
	//and is available is buffer

	return UDR;
}
```

main.c

```
#include "Header Files/uart_stdio.h"
#include "Header Files/button.h"
#include "Header Files/initPort.h"
#include "Header Files/led.h"
#include <util/delay.h>


int main(void) {

	uart_stdio_Init();
	initPorts();

	while(1){
		if (isButtonPressed() == 1) {
			ledOn();
			_delay_ms(500);
```

```
            } else {
                ledOff();
            }
        }
        return 0;
}
```

Led.h

```
#ifndef LED_H_
#define LED_H_

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

void ledOn();
void ledOff();

#endif /* LED_H_ */
```

Led.c

```
#include "Header Files/led.h"

void ledOff() {
         PORTB &= ~(1<<PB0); //Turns OFF LED
}

void ledOn() {
        PORTB |= (1<<PB0);
}
```

initPort.h

```
#ifndef INITPORT_H_
#define INITPORT_H_

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

void initPorts();

#endif /* INITPORT_H_ */
```

initPort.c

```
#include "Header Files/initPort.h"
```

```
void initPorts() {

      DDRB |= (1<<PB0); //Makes first pin of PORTC as Output

      DDRC &= ~(1<<PC0);//Makes firs pin of PORTD as Input

}
```

Button.h

```
#ifndef BUTTON_H_
#define BUTTON_H_

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

int isButtonPressed();

#endif /* BUTTON_H_ */
```

Button.c

```
#include "Header Files/button.h"

int isButtonPressed() {
      if ((PINC & (1<<PC0)) == 1) {
            return 1;
      }
      return 0;
}
```