Ministerul Educatiei al Republicii Moldova
Universitatea Tehnica a Moldovei
Filiera Anglofona

Report

Laboratory Nr.3

Embeded Systems

Performed By: Daniel Ciobanu

Verified By: Andrei Bragarenco

Chisinau 2017

Topic : Analog to Digital Converter of AVR MCU. Temperature measurement using LM20 Sensor.

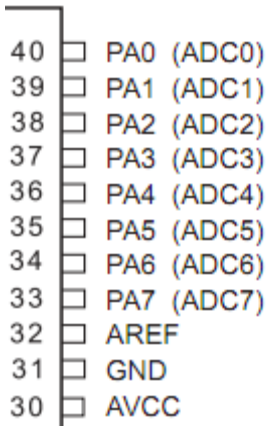
Task: Retrieve the data from a temperature sensor(analog value that has to be converted). The default value to be displayed will be in °**C**. There will be two buttons, used to switch from °**C** to **°F.** The temperature has to be displayed on an LCD interface or virtual terminal.

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.

Interfacing Sensors:

In general, sensors provide with analog output, but a MCU is a digital one. Hence we need to use ADC. For simple circuits, comparator op-amps can be used. But even this won't be required if we use a MCU. We can straightaway use the inbuilt ADC of the MCU. In ATMEGA16/32, PORTA contains the ADC pins.



Apart from this, the other things that we need to know about the AVR ADC are:

- ADC Prescaler
- ADC Registers – ADMUX, ADCSRA, ADCH, ADCL and SFIOR

ADC Prescaler:

The ADC of the AVR converts analog signal into digital signal at some regular interval. This interval is determined by the clock frequency. In general, the ADC operates within a frequency range of 50kHz to 200kHz. But the CPU clock frequency is much higher (in the order of MHz). So to achieve it, frequency division must take place. The prescaler acts as this division factor. It produces desired frequency from the external higher frequency. There are some predefined division factors – 2, 4, 8, 16, 32, 64, and 128. For example, a prescaler of 64 implies F_ADC = F_CPU/64. For F_CPU = 16MHz, F_ADC = 16M/64 = 250kHz.

Now, the major question is… which frequency to select? Out of the 50kHz-200kHz range of frequencies, which one do we need? Well, the answer lies in your need. **There is a trade-off between frequency and accuracy**. Greater the frequency, lesser the accuracy and vice-versa. So, if your application is not sophisticated and doesn't require much accuracy, you could go for higher frequencies.

ADC Registers:

ADMUX – ADC Multiplexer Selection Register

The ADMUX register is as follows.

Bit	7	6	5	4	3	2	1	0	
	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	

The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

Thus, to initialize ADMUX, we write

```
ADMUX = (1<<REFS0);
```

Bits 7:6 – REFS1:0 – Reference Selection Bits – These bits are used to choose the reference voltage.

Bit 5 – ADLAR – ADC Left Adjust Result – Make it ‘1’ to Left Adjust the ADC Result. We will discuss about this a bit later.

Bits 4:0 – MUX4:0 – Analog Channel and Gain Selection Bits – There are 8 ADC channels (PA0…PA7). Which one do we choose? Choose any one! It doesn’t matter. How to choose? You can choose it by setting these bits. Since there are 5 bits, it consists of 2^5 = 32 different conditions as follows. However, we are concerned only with the first 8 conditions. Initially, all the bits are set to zero.

ADCSRA – ADC Control and Status Register A

The ADCSRA register is as follows.

Bit	7	6	5	4	3	2	1	0	
	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	

Thus, we initialize ADCSRA as follows.

```
ADCSRA = (1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
// prescaler = 128
```

Bit 7 – ADEN – ADC Enable – As the name says, it enables the ADC feature. Unless this is enabled, ADC operations cannot take place across PORTA i.e. PORTA will behave as GPIO pins.

Bit 6 – ADSC – ADC Start Conversion – Write this to ‘1’ before starting any conversion. This 1 is written as long as the conversion is in progress, after which it returns to zero. Normally it takes 13 ADC clock pulses for this operation. But when you call it for the first time, it takes 25 as it performs the initialization together with it.

Bit 5 – ADATE – ADC Auto Trigger Enable – Setting it to ‘1’ enables auto-triggering of ADC. ADC is triggered automatically at every rising edge of clock pulse. View the SFIOR register for more details.

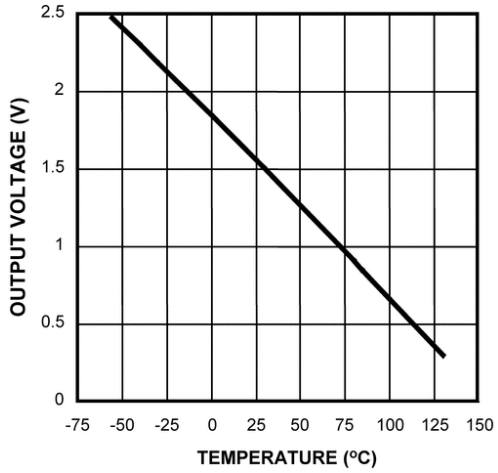
Bit 4 – ADIF – ADC Interrupt Flag – Whenever a conversion is finished and the registers are updated, this bit is set to ‘1’ automatically. Thus, this is used to check whether the conversion is complete or not.

Bit 3 – ADIE – ADC Interrupt Enable – When this bit is set to ‘1’, the ADC interrupt is enabled. This is used in the case of interrupt-driven ADC.

Bits 2:0 – ADPS2:0 – ADC Prescaler Select Bits – The prescaler (division factor between XTAL frequency and the ADC clock frequency) is determined by selecting the proper combination from the following.

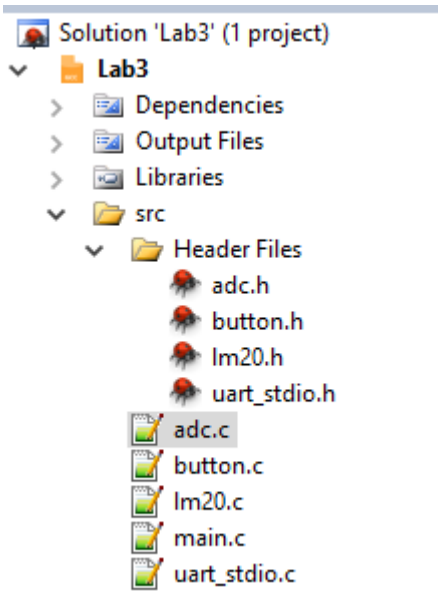
LM20 sensor:

The LM20 is a precision analog output CMOS integrated-circuit temperature sensor that operates over −55°C to 130°C. The power supply operating range is 2.4 V to 5.5 V. The transfer function of LM20 is predominately linear, yet has a slight predictable parabolic curvature. The accuracy of the LM20 when specified to a parabolic transfer function is ±1.5°C at an ambient temperature of 30°C. The temperature error increases linearly and reaches a maximum of ±2.5°C at the temperature range extremes. The temperature range is affected by the power supply voltage. At a power supply voltage of 2.7 V to 5.5 V, the temperature range extremes are 130°C and −55°C.



Implementation:

The structure of the laboratory 3 looks like this:



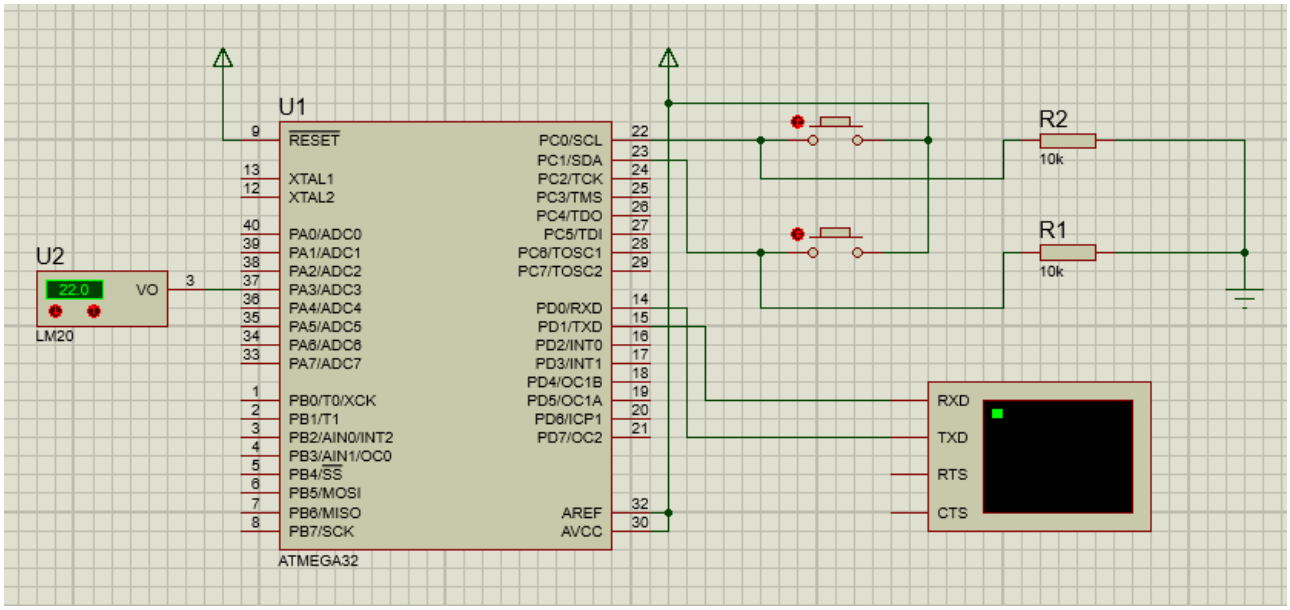
Now first thing we do, we initialize the adc registers according to the datasheet then we need to read ADC value:

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

void initADC(){
    ADMUX = (1<<REFS0);
    ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

int readADC(int channel) {
    int adcData = 0;
    while (ADCSRA & 1 << ADSC); //wait until ADC is busy
    channel &= 0x07;
    ADMUX = (ADMUX & (0x07)) | channel; //apply the channel to the ADMUX with
    ADCSRA |= (1 << ADSC); //start conversion
    while (ADCSRA & (1 << ADSC)); //wait until conversion is complete
    adcData = ADC;
    return adcData;
}
```

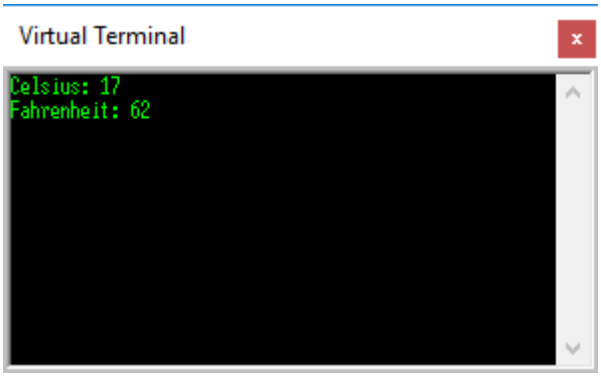
Then we use the ADC value to convert the the voltage from sensor to temperature.

```
void lm20Init() {
    initADC();
}

int lm20GetTemp() {
    return  (382 - readADC(3)) / 3;

}

int farenheitFromCelsius ( int temp ) {
    return temp * 1.8 + 32;
}
```

The scheme in Proteus:



The Result:


Virtual Terminal
x
Celsius: 17
Fahrenheit: 62

Conclusion :

This laboratory work provides us with the knowledge how to use analog to digital convertor and for what is used. Also we analized how lm20 sensor works and what needs to be done in order to get data from it and print it on the virtual terminal.

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/lm20.h"
#include "Header Files/button.h"
#include <avr/io.h>
#include <util/delay.h>


int main(void) {

	initButton1();
	initButton2();
	lm20Init();
	uart_stdio_Init();

	while(1){

		_delay_ms(500);

		if (button1Pressed()) {
```

```
                printf("Celsius: %d \r", lm20GetTemp());
          }

          if (button2Pressed()) {
                printf("Fahrenheit: %d \r",
farenheitFromCelsius(lm20GetTemp()));
          }

     }
     return 0;
}
```

Lm20.h

```
#ifndef LM20_H_
#define LM20_H_

void lm20Init();
int lm20GetTemp();
int farenheitFromCelsius(int temp);

#endif /* LM20_H_ */
```

Lm20.c

```
#include "Header Files/lm20.h"
#include "Header Files/adc.h"
#include "Header Files/uart_stdio.h"
#include <avr/io.h>


void lm20Init() {
     initADC();
}

int lm20GetTemp() {
     return  (382 - readADC(3)) / 3;

}

int farenheitFromCelsius ( int temp ) {
     return temp * 1.8 + 32;
}
```

adc.h

```
#ifndef ADC_H_
```

```
#define ADC_H_

#include <avr/io.h>

void initADC();
int readADC(int channel);

#endif /* ADC_H_ */
```

adc.c

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

void initADC(){
	ADMUX = (1<<REFS0);
	ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}

int readADC(int channel) {
	int adcData = 0;
	while (ADCSRA & 1 << ADSC); //wait until ADC is busy
	channel &= 0x07;
	ADMUX = (ADMUX & (0x07)) | channel; //apply the channel to the
ADMUX with protection of configuration bits
	ADCSRA |= (1 << ADSC); //start conversion
	while (ADCSRA & (1 << ADSC)); //wait until conversion is complete
	adcData = ADC;
	return adcData;
}
```

Button.h

```
#ifndef BUTTON_H_
#define BUTTON_H_

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

void initButton1();
void initButton2();

int button1Pressed();
int button2Pressed();


#endif /* BUTTON_H_ */
```

Button.c

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

void initButton1() {
	DDRC &= ~(1 << PORTC0) ;
}

void initButton2() {
	DDRC &= ~(1 << PORTC1);
}

int button1Pressed() {
	return PINC & (1 << PORTC0);
}

int button2Pressed() {
	return PINC & (1 << PORTC1);
}
```