Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

# RAPORT

la Programarea aplicațiilor incorporate și independente de platformă

Lucrare de laborator Nr. 3

Tema: : Transformarea semnalului Analog în Digital. Conectarea unui sensor de temperatură.

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

A verificat: Andrei Bragarenco

Chişinău 2016

# Topic

Converting Analog to Digital signal. Connecting temperature sensor to MCU and display temperature to display.

# Purpose

- ADC of the AVR
- Analog to Digital Conversion
- Connecting Temperature Sensor to MCU

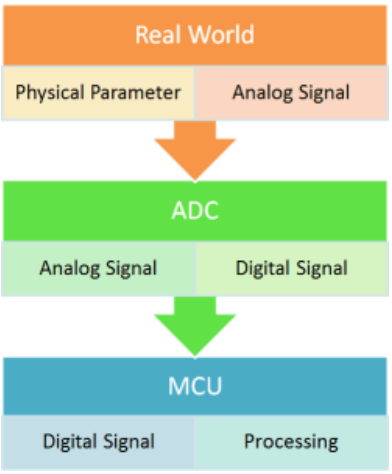
# Task

Write a driver for ADC and LM20 Temperature sensor. ADC will transform Analog to Digital data. LM20 driver will use data from ADC to transform to temperature regarding to this sensor parameters. Also use push button to switch between metrics Celsius, Fahrenheit and Kelvin.

# Domain

➔ Analog to Digital Conversion

Most real world data is analog. Whether it be temperature, pressure, voltage, etc, their variation is always analog in nature. For example, the temperature inside a boiler is around 800°C. During its light-up, the temperature never approaches directly to 800°C. If the ambient temperature is 400°C, it will start increasing gradually to 450°C, 500°C and thus reaches 800°C over a period of time. This is an analog data.\




Now, we must process the data that we have received. But analog signal processing is quite inefficient in terms of accuracy, speed and desired output. Hence, we convert them to digital form using an Analog to Digital Converter (ADC).

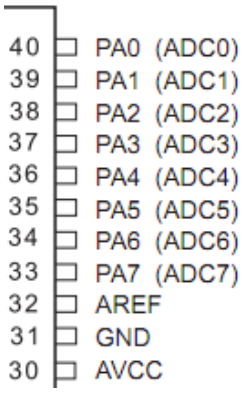
➔ Signal Acquisition Process

In general, the signal (or data) acquisition process has 3 steps.

In the Real World, a sensor senses any physical parameter and converts into an equivalent analog electrical signal.

For efficient and ease of signal processing, this analog signal is converted into a digital signal using an **Analog to Digital Converter (ADC).**

This digital signal is then fed to the **Microcontroller (MCU)** and is processed accordingly.




➔ 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.

➔ The ADC of the AVR

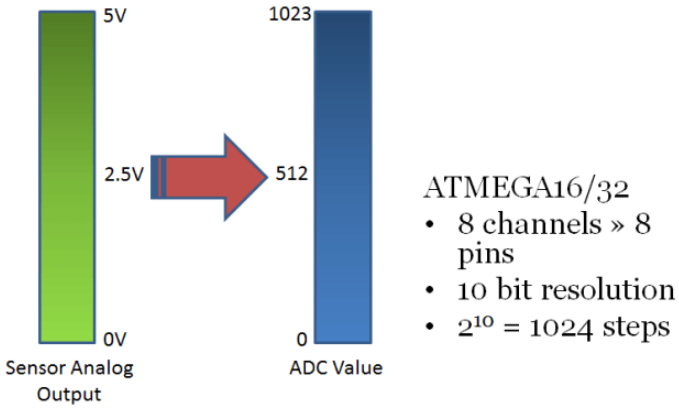
The AVR features inbuilt ADC in almost all its MCU. In ATMEGA16/32, PORTA contains the ADC pins. Some other features of the ADC are as follows:

- **10-bit Resolution**
- **0.5 LSB Integral Non-linearity**
- **±2 LSB Absolute Accuracy**
- **13 - 260 µs Conversion Time**
- **Up to 15 kSPS at Maximum Resolution**
- **8 Multiplexed Single Ended Input Channels**
- **7 Differential Input Channels**
- **2 Differential Input Channels with Optional Gain of 10x and 200x**
- **Optional Left adjustment for ADC Result Readout**
- **0 - $V_{CC}$ ADC Input Voltage Range**
- **Selectable 2.56V ADC Reference Voltage**
- **Free Running or Single Conversion Mode**
- **ADC Start Conversion by Auto Triggering on Interrupt Sources**
- **Interrupt on ADC Conversion Complete**
- **Sleep Mode Noise Canceler**

Right now, we are concerned about the **8 channel 10 bit resolution** feature.

**8 channel** implies that there are 8 ADC pins are multiplexed together. You can easily see that these pins are located across PORTA (PA0…PA7).

**10 bit resolution** implies that there are 2^10 = 1024 steps (as described below).




Suppose we use a 5V reference. In this case, any analog value in between 0 and 5V is converted into its equivalent ADC value as shown above. The 0-5V range is divided into 2^10 = 1024 steps. Thus, a 0V input will give an ADC output of 0, 5V input will give an ADC output of 1023, whereas a 2.5V input will give an ADC output of around 512. This is the basic concept of ADC.

To those whom it might concern, the type of ADC implemented inside the AVR MCU is of Successive Approximation type.

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 | |

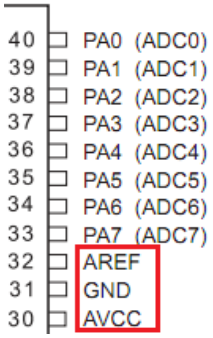
The bits that are highlighted are of interest to us. In any case, we will discuss all the bits one by one.

- **Bits 7:6 – REFS1:0 – Reference Selection Bits** — These bits are used to choose the reference voltage. The following combinations are used.

| REFS1 | REFS0 | Voltage Reference Selection |
|---|---|---|
| 0 | 0 | AREF, Internal Vref turned off |
| 0 | 1 | AVCC with external capacitor at AREF pin |
| 1 | 0 | Reserved |
| 1 | 1 | Internal 2.56V Voltage Reference with external capacitor at AREF pin |

The ADC needs a reference voltage to work upon. For this we have a three pins AREF, AVCC and GND. We can supply our own reference voltage across AREF and GND. For this, choose the first option. Apart from this case, you can either connect a capacitor across AREF pin and ground it to prevent from noise, or you may choose to leave it unconnected. If you want to use the VCC (+5V), choose the second option. Or else, choose the last option for internal Vref.

Let's choose the second option for Vcc = 5V.



- **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 | |

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

- **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.

### ADCL and ADCH – ADC Data Registers

The result of the ADC conversion is stored here. Since the ADC has a resolution of 10 bits, it requires 10 bits to store the result. Hence one single 8 bit register is not sufficient. We need two registers – ADCL and ADCH (ADC Low byte and ADC High byte) as follows. The two can be called together as ADC.

## Used Resources

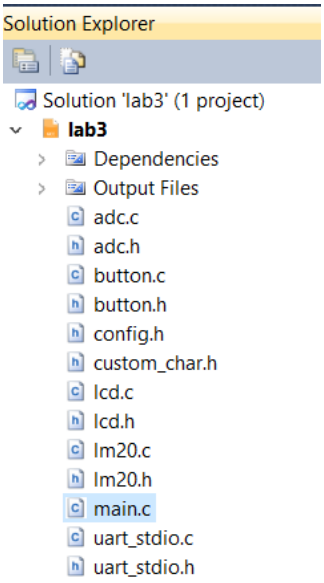
### *LM20 –Temperature Sensor*

Characteristics


- Rated for −55°C to 130°C Range
- Suitable for Remote Applications
- Accuracy at 30°C ±1.5 to ±4°C (Maximum)
- Accuracy at 130°C and −55°C ±2.5 to ±5°C
- Power Supply Voltage Range 2.4 V to 5.5 V
- Current Drain 10 µA (Maximum)
- Output Impedance 160 Ω (Maximum)

# Solution

The *Project Structure* looks in this way




### adc.h / adc.c

Contains declaration and implementation of ADC Driver. It has 3 methods:

```
void initADC(); // Initializes driver

int getData(); // Get data from initialized ADC. It's a 10 bit value, which can give us value from range 0..1023

void toVoltage(int t); //converts temperature to voltage
```

### lm20.h /lm20.c

Contains declaration and implementation of LM20 Driver. It depends on ADC driver. It has 3 methods:

```
void initLM(); // Initializes driver
int getTemp(); // Uses ADC driver to get digital value
int convertCelsiusToKelvin(int temp); //converts digital value to temperature
```

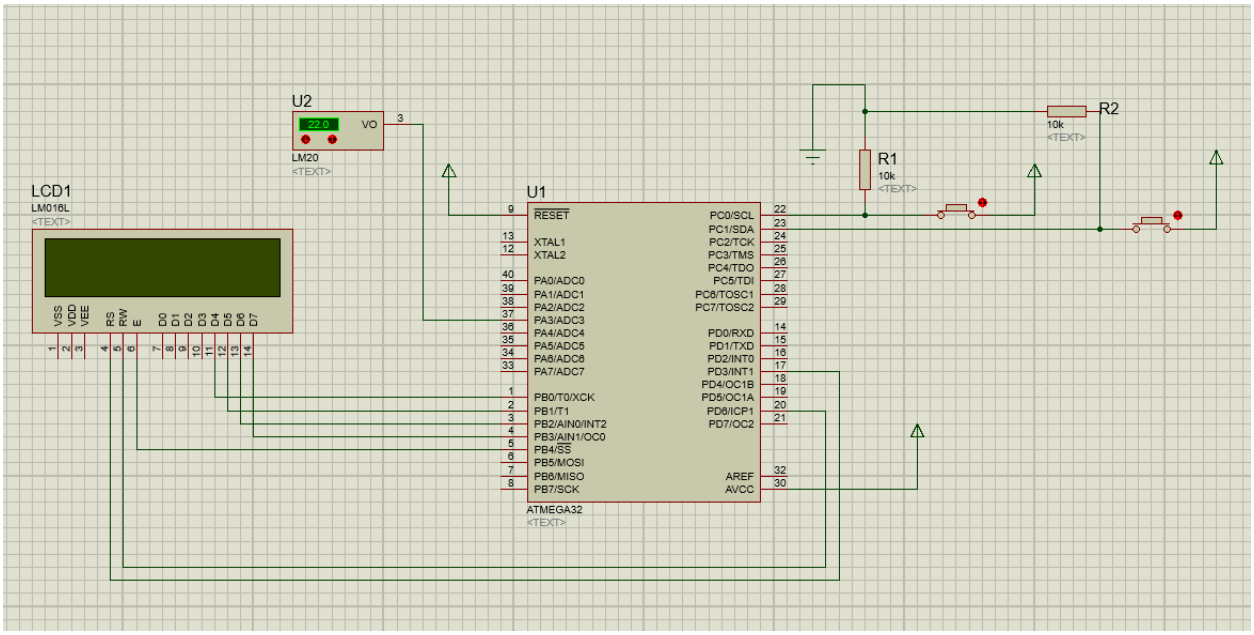
### Main Program

Main Program is responsible for :

- Initialization of 2 buttons , which switch from Celsius to Fahrenheit and Kelvin
- Initialization of UART , for displaying temperature
- Initialization of LM20 Sensor
- program receives temperature from Sensor and converts it to needed conversion, which is decided by button push

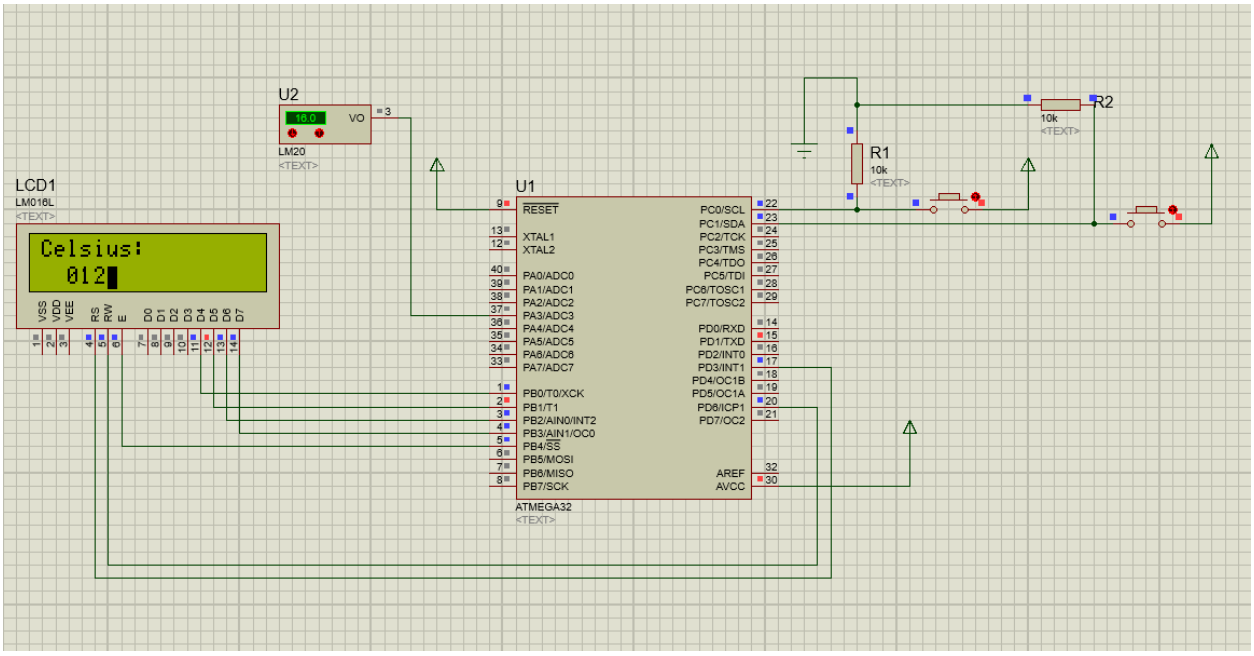
## Schematics

For our laboratory work we need ATMega32 MCU, 2 Push Buttons, LM20 sensor and LCD.

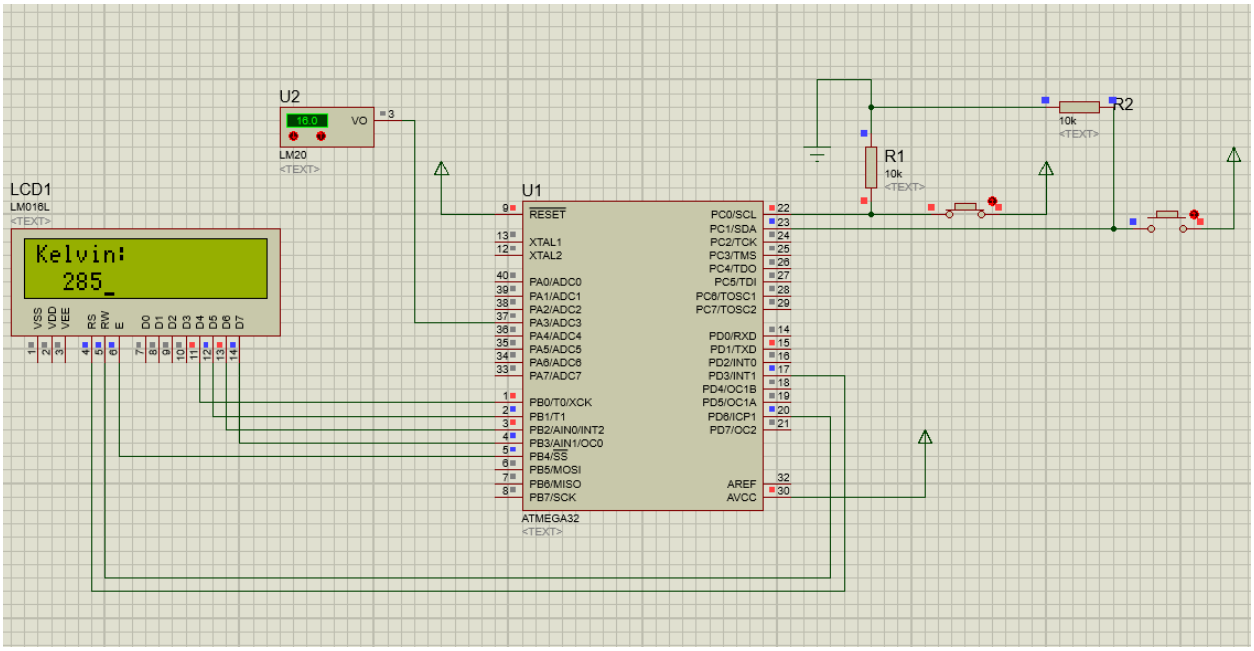



## Simulation Result

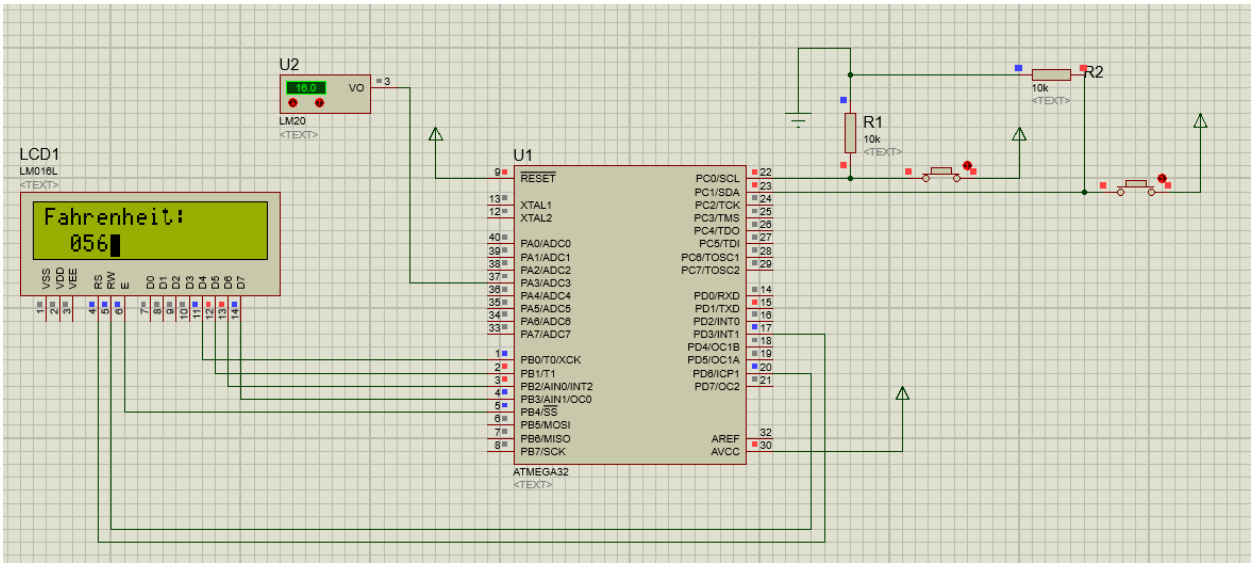
**Temperature in Celsius**




**Temperature in Kelvin**




**Temperature in Fahrenheit**




# Conclusion

During this laboratory work I learned about ADC, connecting sensor to MCU, writing driver for ADC and LM20, getting analog data from sensor and converting it to digital data.

Getting data from sensors is important in real life, because a lot of applications have to interact with the outside environment and then do the program with the data.

# Appendix

## main.c

```
#include <avr/io.h>
#include "button.h"
#include "lm20.h"
#include "lcd.h"
#include <avr/delay.h>

int main(void) {

	initButtonOne();
	initButtonTwo();
	initLM();
	uart_stdio_Init();

	//Initialize LCD module
   LCDInit(LS_BLINK|LS_ULINE);

   //Clear the screen
   LCDClear();


    while(1) {
		_delay_ms(1000);

		if(isButtonOnePressed()) {
			if(isButtonTwoPressed()) {
				LCDClear();
				LCDWriteString("Fahrenheit:");
				LCDWriteIntXY(1, 1, convertCelsiusToFahrenheit(getTemp()),3);
				printf("Fahrenheit: %d\n",
convertCelsiusToFahrenheit(getTemp()));
			}else {
				LCDClear();
				LCDWriteString("Kelvin:");
				LCDWriteIntXY(1, 1, convertCelsiusToKelvin(getTemp()),3);
				printf("Kelvin: %d\n", convertCelsiusToKelvin(getTemp()));
			}

		} else {
			LCDClear();
			LCDWriteString("Celsius:");
			LCDWriteIntXY(1, 1, getTemp(),3);
			printf("Celsius : %d\n", getTemp());
		}
    }
}
```

## adc.h

```
#ifndef ADC_H_
#define ADC_H_

void initADC();
int getData();
void toVoltage(int t);
```

```
#endif /* ADC_H_ */
```

### adc.c

```
#include "adc.h"
#include <avr/io.h>
int data;

void initADC() {

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

int getData() {

	int adcData = 0;
	int port = 3;

	// wait for conversion to complete
	// ADSC becomes '0' again, till then, run loop continuously
	while(ADCSRA & 1 << ADSC);

	// select the corresponding channel 0~7
    // ANDing with '7' will always keep the value between 0 and 7
	port &= 0x07;

	ADMUX = (ADMUX & ~(0x07)) | port; //clears the bits
	ADCSRA |= (1<<ADSC); // start single convertion write '1' to ADSC
	while (ADCSRA & (1<<ADSC));
	adcData = ADC;
	return adcData;

}
```

### lm20.h

```
#ifndef LM20_H_
#define LM20_H_

#include "adc.h"

void initLM();
int getTemp();
int convertCelsiusToKelvin(int temp);

#endif /* LM20_H_ */
```

### lm20.c

```
#include "lm20.h"

int temp = 0;

void initLM() {
```

```
	initADC();
}

int getTemp() {
	temp = (382 - getData()) / 3;
	return temp;
}

int convertCelsiusToKelvin(int temp) {
	return temp + 273;
}

int convertCelsiusToFahrenheit(int temp) {
	return temp * 2 + 32;
}
```

## FlowChart