Ministerul Educației al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare,Informatică și Microelectronică

Laboratory work nr.3

REPORT

At Embedded systems

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

st. gr. FAF-141: Cristea Victor

Verified by: Andrei Bragarenco

Chişinău 2016

Goal of the work:

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

Condition:

Write 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.

Theory

A **sensor** is a device whose purpose is to detect events or changes in its environment, and then provide a corresponding output.
Data transfer from the sensor to the CPU can be either CPU-initiated (*polling*) or sensor-initiated (via *interrupt*). In case it is CPU-initiated, the CPU has to keep checking whether the sensor is ready by reading a status line in a loop. This is much more time consuming than the alternative of a sensor-initiated data transfer, which requires the availability of an interrupt line. The sensor signals via an interrupt that data is ready, and the CPU can react immediately to this request.

Sensor Output	Sample Application
Binary signal (0 or 1)	Tactile sensor
Analog signal (e.g. 0..5V)	Inclinometer
Timing signal (e.g. PWM)	Gyroscope
Serial link (RS232 or USB)	GPS module
Parallel link	Digital camera

There is the following sensors classification:

- from a robot's point of view, it is more important to distinguish:
    - **Local** or **on-board** sensors *(mounted on the robot)*
    - **Global** sensors *(mounted outside the robot in its environment and transmitting sensor data back to the robot)*
- for mobile robot systems it is also important to distinguish:
    - **Internal** sensors (monitoring the robot's internal state)
    - **External** sensors (monitoring the robot's environment)
- from the point of view of sensor's activity:
    - **Passive** sensors (monitor the environment without disturbing it, for example digital camera, gyroscope)
    - **Active** sensors (stimulate the environment for their measurement, for example sonar sensor, laser scanner, infrared sensor)

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).**

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:

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).

ADC Prescaler

The ADC needs a clock pulse to do its conversion. This clock generated by system clock by dividing it to get smaller frequency. The ADC requires a frequency between 50KHz to 200KHz. At higher frequency the conversion is fast while a lower frequency the conversion is more accurate. As the system frequency can be set to any value by the user (using internal or externals oscillators). **So the Prescaler is provided to <u>produces acceptable frequency for ADC</u> from any system clock frequency. System** clock can be divided by 2, 4, 16, 32, 64, 128 by setting the Prescaler.

ADC Channels

The ADC in ATmega32 has 8 channels that means you can take samples from eight different terminal. You can connect up to 8 different sensors and get their values separately.

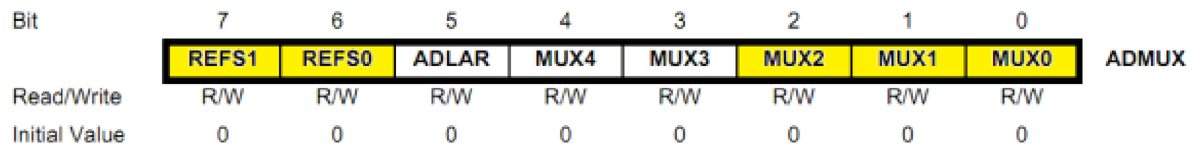
ADC Registers

The registers related to any particular peripheral module (like ADC, Timer, USART etc.) provides the communication link between the CPU and that peripheral. You configure the ADC according to need using these registers and you also get the conversion result also using appropriate registers. The ADC has only four registers.

1. **ADC Multiplexer Selection Register (ADMUX)** – for selecting the reference voltage and the input channel
2. **ADC Control and Status Register A (ADCSRA)** – as the name says it has the status of ADC and is also used for controlling it
3. **ADC Data Register (ADCL and ADCH)** – the final result of conversion is here

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.

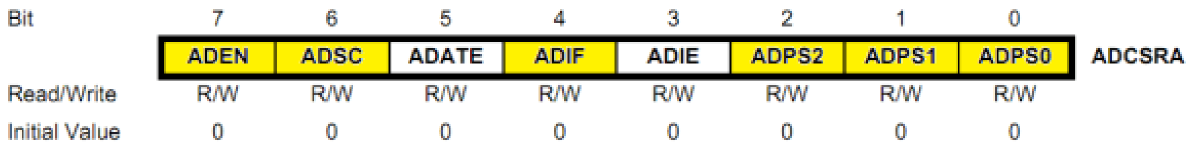
The ADC needs a reference voltage to work upon. For this we have a three pins AREF, AVCC and GND. We can supply our own reference voltage across AREF and GND. For this, choose the first option. Apart from this case, you can either connect a capacitor across AREF pin and ground it to prevent from noise, 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.



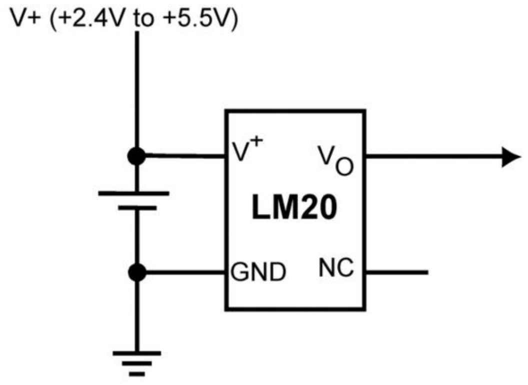
Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

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.



Scheme 1: *Simplified schematic from datasheet*

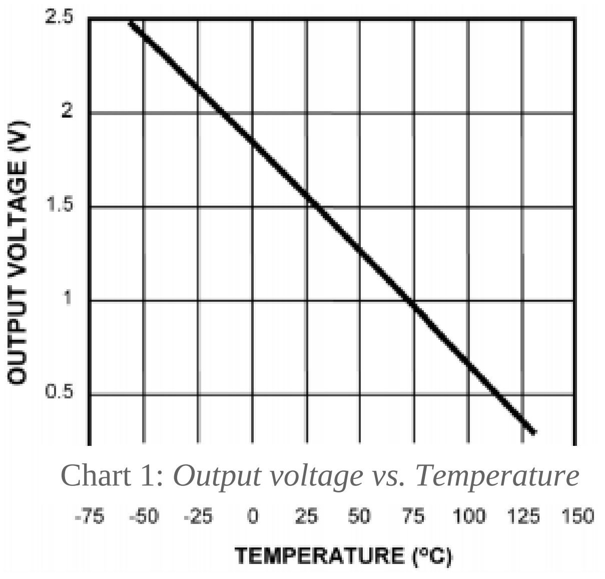


Chart 1: *Output voltage vs. Temperature*

ADC

An **A/D converter** translates an analog signal into a digital value. The characteristics of an A/D converter include:

- *Accuracy* – expressed in the number of digits it produces per value (for example 10bit A/D converter)
- *Speed* – expressed in maximum conversions per second (for example 500 conversions per second)
- *Measurement range* – expressed in volts (for example 0..5V)

A/D converters come in many variations. The output format also varies. Typical are either a parallel interface (for example up to 8 bits of accuracy) or a synchronous serial interface. The latter has the advantage that it does not impose any limitations on the number of bits per measurement, for example 10 or 12bits of accuracy.

.

Resources

LM20 –Temperature Sensor

In this laboratory work I use a LM20 temperature sensor for reading environment temperature.
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 2.4V to 5.5V. 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 ambient temperature of 30°C.

Characteristics

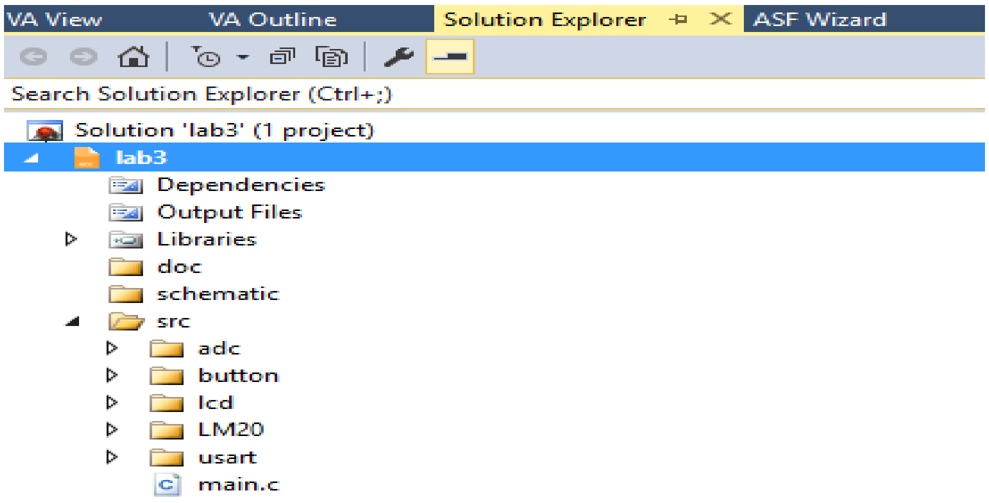
- Accuracy at 130°C and −55°C ±2.5 to ±5°C
- Accuracy at 30°C ±1.5 to ±4°C (Maximum)
- Current Drain 10 µA (Maximum)
- Output Impedance 160 Ω (Maximum)
- Power Supply Voltage Range 2.4 V to 5.5 V
- Rated for −55°C to 130°C Range
- Suitable for Remote Applications

Applications

- Appliances
- Battery Management
- Cellular Phones
- Computers
- Disk Drives
- FAX Machines
- HVAC
- Power Supply Modules
- Printers

Solving

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. **Project Structure** looks in this way



```
Solution 'lab3' (1 project)
  lab3
      Dependencies
      Output Files
    Libraries
      doc
      schematic
    src
      adc
      button
      lcd
      LM20
      usart
      main.c
```

Button

In order to make the program efficient and elegant, I have chosen to represent each connected device to a port of the MCU with a button struct:

```
struct Button {
    uint8_t pinNr;
        volatile uint8_t *ddr;
        volatile uint8_t *ioReg;
};
```

pinNr - is the index of the pin at some specific port(A, B, C or D)

ddr - configuration on input or output of the whole port

ioReg- pin or port - in dependence of the configuration (input or output)

LCD

For LCD interfacing I used a library found on internet - written by **eXtreme Electronics India.** For more info, check the link Extreme Elecrtonics.

ADC

For LCD interfacing I used a library found on internet - written by **eXtreme Electronics India.** For more info, check the link Extreme Elecrtonics.

LM20

For LCD interfacing I used a library found on internet - written by **eXtreme Electronics India.** For more info, check the link Extreme Elecrtonics.

Main Program

Main function is the entry point of the program. It works in the following way:

- Initializes the lcd, button and LM20 :

  initButtons();

  USARTInit();

  LM20_Init();

  LCDInit(LS_NONE);

- Initializes the LCD:

  LCDInit(LS_BLINK);

- Enters the infinite while loop:

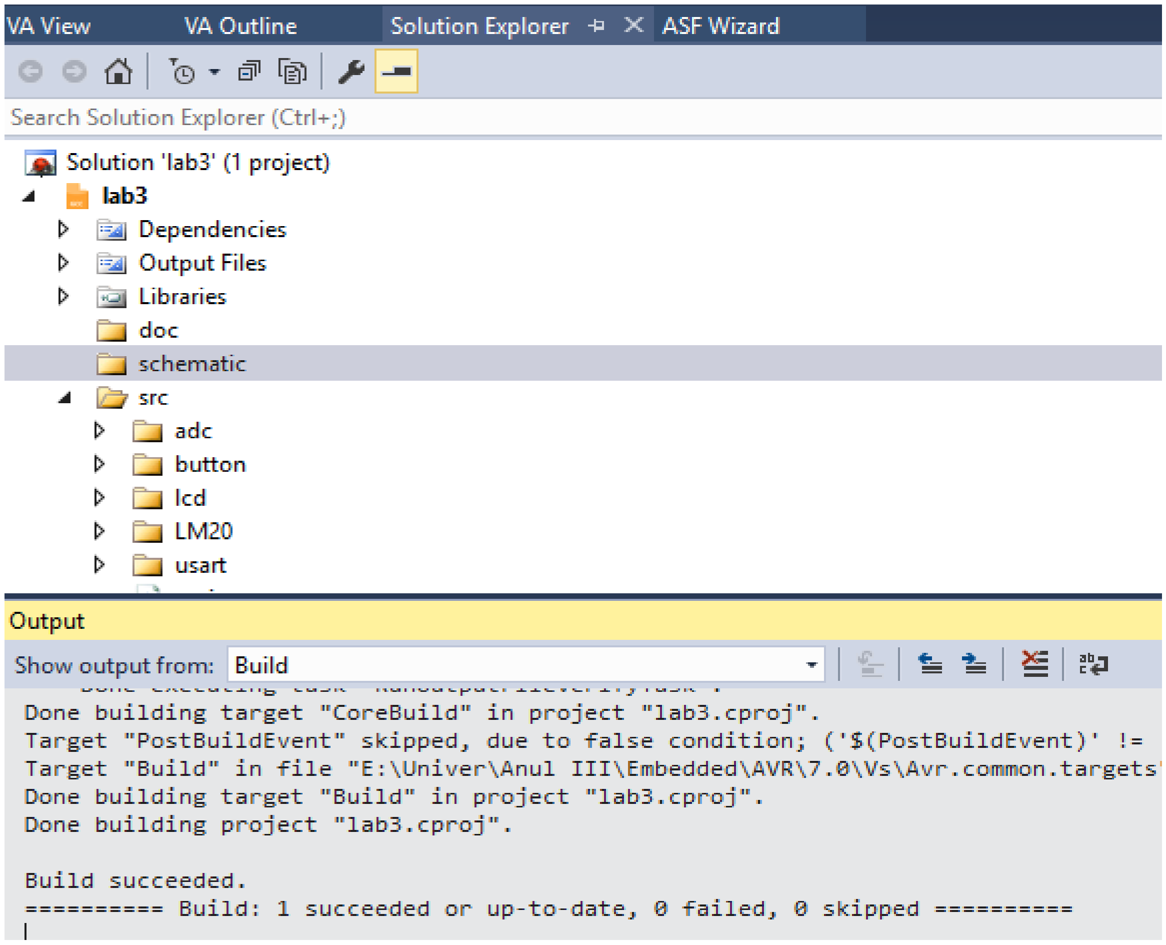
  With a frequency of 50 ms (_delay_ms(50);)

  (a) Gets the celsius value from ADC:
  value = LM20_GetCelsiusValue(value);

  (b) Converts it to display on LCD:
  itoa(value, buffer, 10);

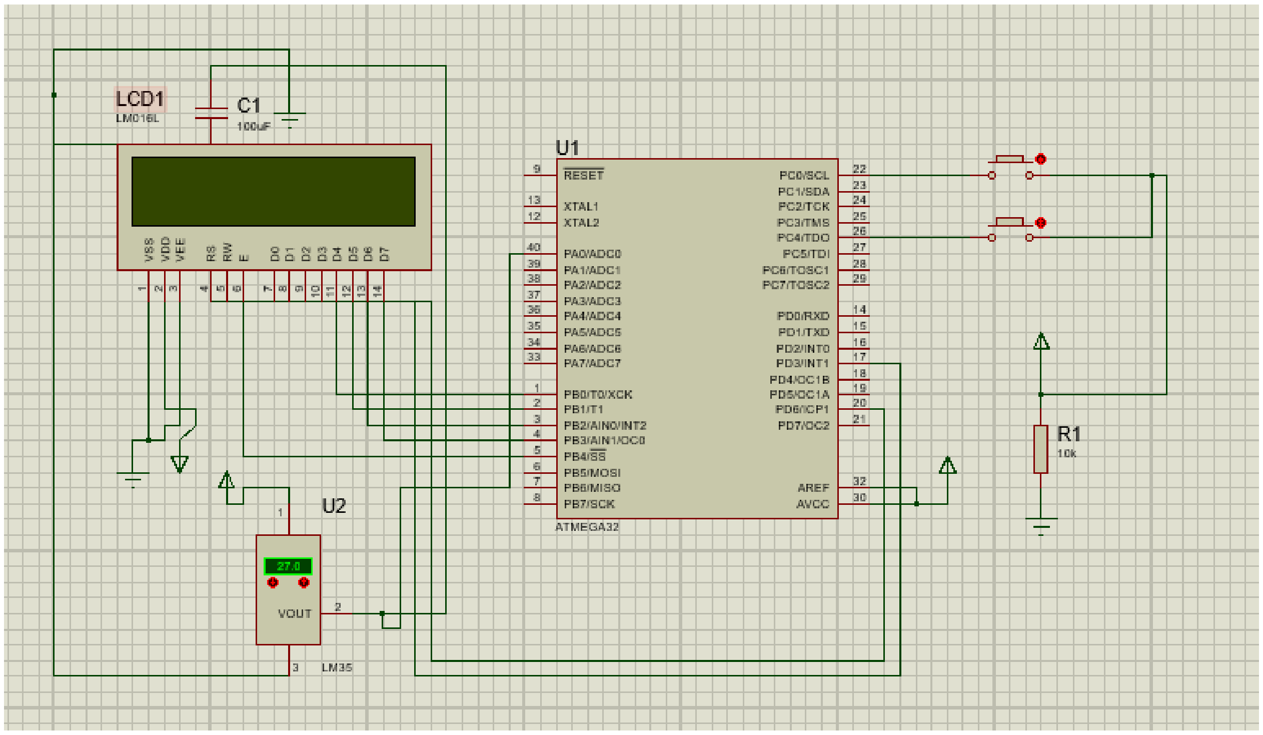
  (c) Checks whether the convert to Fahrenheit or Kelvin conversion buttons were clicked.
  checkTConversion(buffer, value);

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



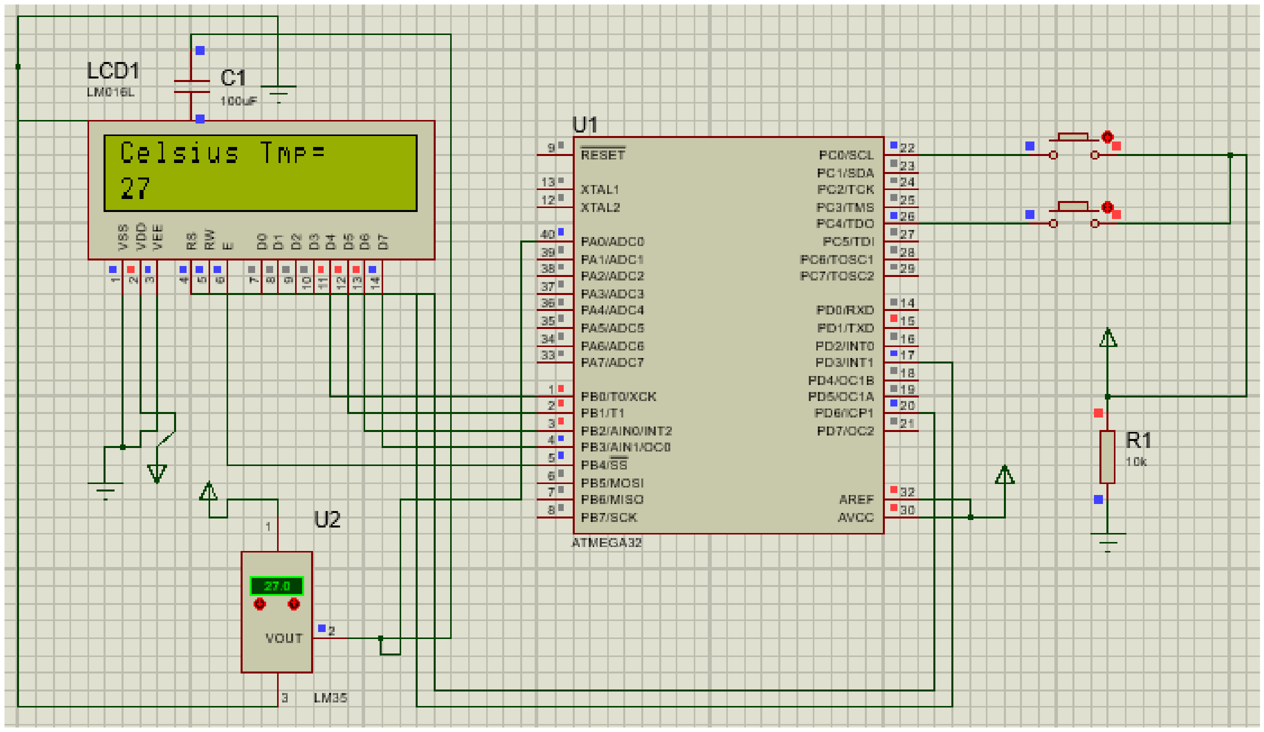
```
Done building target "CoreBuild" in project "lab3.cproj".
Target "PostBuildEvent" skipped, due to false condition; ('$(PostBuildEvent)' !=
Target "Build" in file "E:\Univer\Anul III\Embedded\AVR\7.0\Vs\Avr.common.targets
Done building target "Build" in project "lab3.cproj".
Done building project "lab3.cproj".

Build succeeded.
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

Schematics

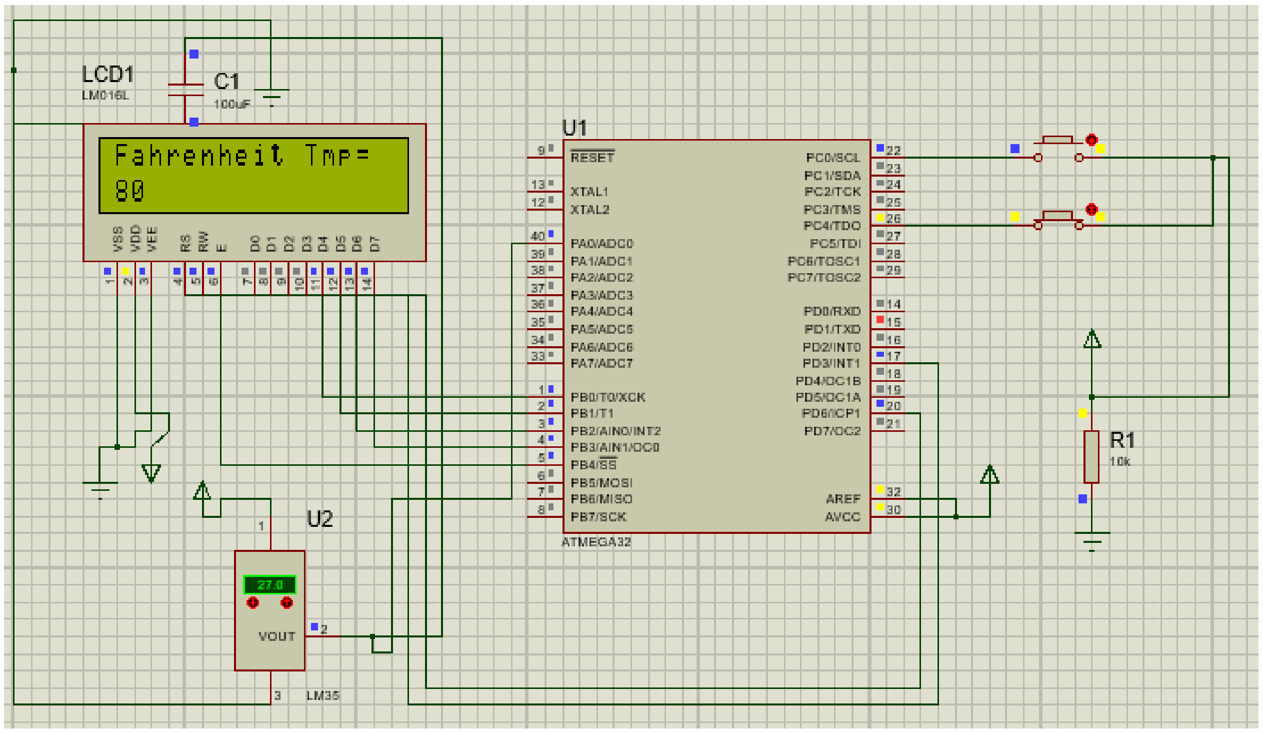


Simulation Result

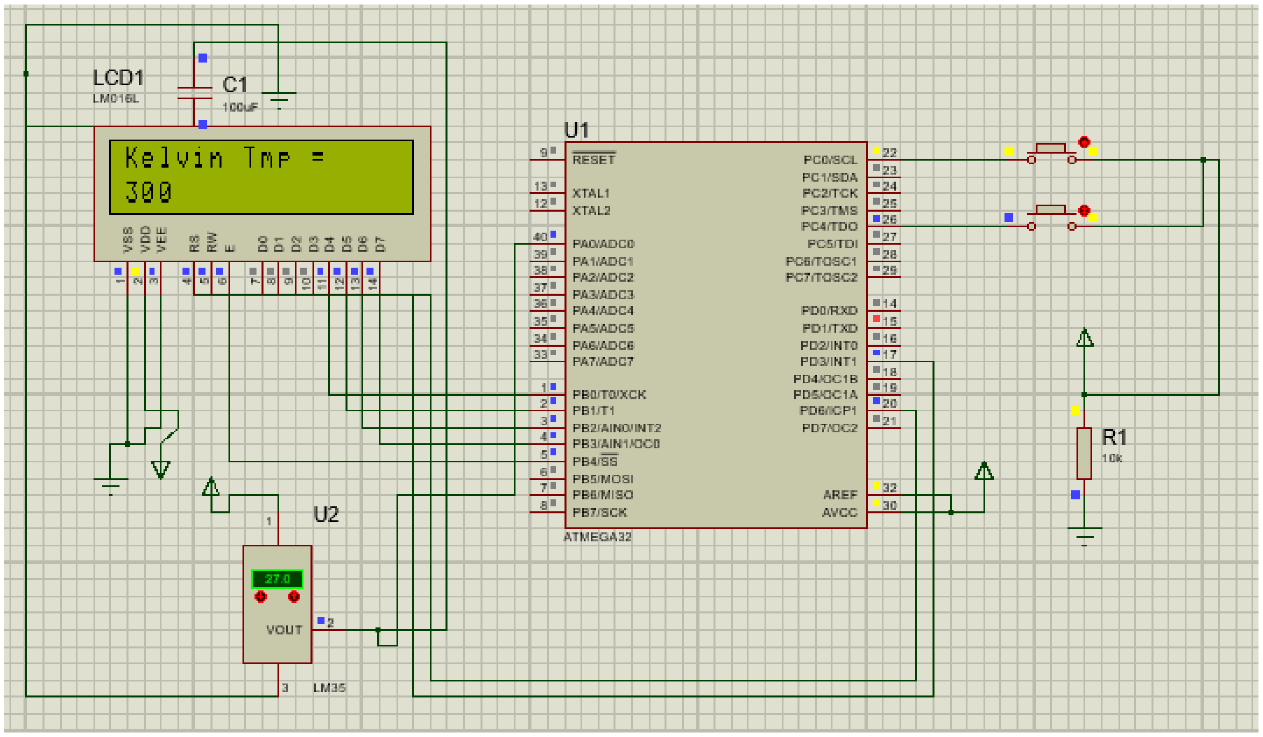
Default Celsius value displayed



Fahrenheit conversion button pressed



Kelvin conversion button pressed



Conclusion

This laboratory work gave us a basic concepts about ADC. We connected sensor to our MCU and have written Drivers for ADC and LM20 which actually prepares hardware and get’s Analog data from sensor then converts it to Digital Data. The hardest part of this laboratory work was ADC driver.Initialization was one of hardest thing. used linear conversion for getting Temperature from device, but it could be better to use formula from LM20 datasheet. This improvement will make our temperature more precise.

Appendix

main.c

```
#include <avr/delay.h>
#include "lcd/lcd_hd44780_avr.h"
#include "usart/usart.h"
#include "adc/adc.h"
#include "LM20/lm20.h"

struct Button *btnKelvin;
struct Button *btnFarenheit;

void initButtons();
void outputOnLCD(char *msg, char *buffer, int value);
void checkTConversion(char *buffer, int value);

int main(void) {
        unsigned int value;
        char buffer[3];
        initButtons();
        USARTInit();
        LM20_Init();
        LCDInit(LS_NONE);

        while(1) {
                value = LM20_GetCelsiusValue(value);
                itoa(value, buffer, 10);
                _delay_ms(50);
                checkTConversion(buffer, value);
        }
}
```

```c
void checkTConversion(char *buffer, int value) {

	if (isButtonPressed(&btnKelvin)) {
		value = convertToKelvin(value);
		outputOnLCD(" Kelvin Tmp = ", buffer, value);
	} else if (isButtonPressed(&btnFarenheit)) {
		value = convertToFarenheit(value);
		outputOnLCD(" Fahrenheit Tmp = ", buffer, value);
	} else {
		outputOnLCD(" Celsius Tmp = ", buffer, value);
	}
}


void initButtons() {
	initButton(&btnKelvin, PINC0, &DDRC, &PINC); // init button
	setButtonDDR(&btnKelvin);

	initButton(&btnFarenheit, PINC4, &DDRC, &PINC); // init button
	setButtonDDR(&btnFarenheit);

}


void outputOnLCD(char *msg, char *buffer, int value) {
	itoa(value, buffer, 10);
	LCDGotoXY(0,0);
	LCDWriteString(msg);
	LCDGotoXY(0,1);
	LCDWriteString("    ");
	LCDGotoXY(0,1);
	LCDWriteString( buffer);
}
```

lm20.h

```c
#ifndef LM20_H
#define LM20_H


void LM20_Init(void);


unsigned int LM20_GetCelsiusValue(int value);
int convertToKelvin(int value);
int convertToFarenheit(int value);


#endif
```

lm20.c

```c
#include "lm20.h"
void LM20_Init(void) {
	ADC_init();
}
```

```
unsigned int LM20_GetCelsiusValue(int value) {
value = ADC_read(0x00);
value = value * 500/1024;
return value;
}


int convertToFarenheit(int value) {
      return((int) value *9/5 + 32);
}


int convertToKelvin(int value) {
      return value + 273;
}
```

adc.h

```
#ifndef ADC_H
#define ADC_H


#define ADC_VREF_TYPE 0x40


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


void ADC_init(void);
unsigned int ADC_read(unsigned char adc_input);


#endif
```

adc.c

```
#include "adc.h"
void ADC_init(void) {
      ADMUX=(1<<REFS0);
      ADCSRA=(1<<ADEN)|(1<<ADPS2)|(1<<ADPS1)|(1<<ADPS0);
}


unsigned int ADC_read(unsigned char adc_input) {
      ADMUX=adc_input | (ADC_VREF_TYPE & 0xff);
      // Delay needed for the stabilization of the ADC input voltage
      _delay_us(10);
      // Start the AD conversion
      ADCSRA|=0b01000000;
      // Wait for the AD conversion to complete
      while ((ADCSRA & 0x10)==1);
      ADCSRA|=0b00000;
      return ADCW;
}
```

button.h

```
#ifndef BUTTON_H_
#define BUTTON_H_



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

struct Button {
uint8_t pinNr;
volatile uint8_t *ddr;
volatile uint8_t *ioReg;
};

void initButton(struct Button *obj,
uint8_t _pinNr,
volatile uint8_t *_ddr,
volatile uint8_t *_ioReg );

char isButtonPressed(struct Button *obj);
void setButtonDDR(struct Button *obj);

#endif
```

button.c

```
#include "button.h"


char  isButtonPressed(struct Button *obj) {
	if((*(obj->ioReg))&(1<<obj->pinNr))
		return 1;
	return 0;
}


void setButtonDDR(struct Button *obj) {
	*(obj->ddr) |= 1<<obj->pinNr;
}


void initButton(struct Button *obj,
	uint8_t _pinNr,
	volatile uint8_t *_ddr,
	volatile uint8_t *_ioReg ) {
		obj->pinNr = _pinNr;
		obj->ddr = _ddr;
		obj->ioReg = _ioReg;
	}
```

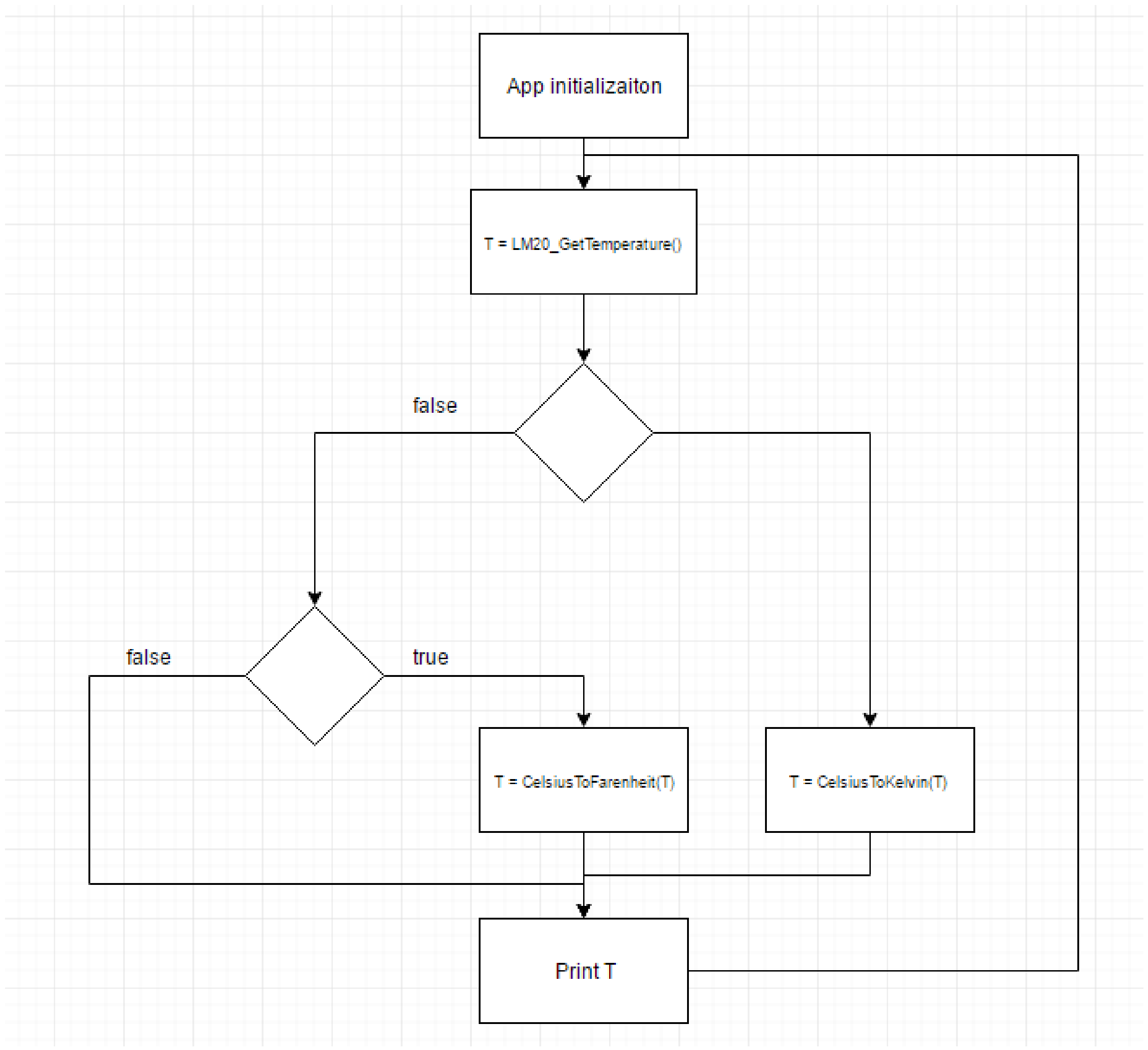
usart.h

```
#ifndef USART_H
#define USART_H


#include <avr/io.h>


void USARTInit();
char USARTReadChar();
void USARTWriteChar(char data);


#endif
```

usart.c

```
#include "usart.h"

void USARTInit() {
       UCSRA=0x00;
       UCSRB=0x18;
       UCSRC=0x86;
       UBRRH=0x00;
       UBRRL=0x33;
}


char USARTReadChar() {

       while(!(UCSRA & (1<<RXC))) { }
       return UDR;
}


void USARTWriteChar(char data) {


       while(!(UCSRA & (1<<UDRE))) { }
       UDR=data;
}
```

FlowChart


App initializaiton
T = LM20_GetTemperature()
false
false
true
T = CelsiusToFarenheit(T)
T = CelsiusToKelvin(T)
Print T