

AD converter

for Basics of Microprocessor technology



Nikolay Arsenov, Alexey Tukalo,
EFA12SF,
Information Technology,
Savonia University of Applied Sciences

December 19, 2014

Contents

1	AD - converter	2
1.1	List of Features	2
1.2	The main ATmega2560 features	2
2	Controlling the conversion frequency	4
3	Changing the input voltage	6
4	The scaling of the result	6
5	Programming the circuit	7
6	Programming tasks	9
6.1	The first task	9
6.2	The second task	10
6.3	The third task	13
6.4	The fourth task	15

1 AD - converter

An analog-to-digital converter (ADC, A/D, or A to D) is a device that converts a continuous physical quantity (usually voltage) to a digital number that represents the quantity's amplitude.

1.1 List of Features

- 10-bit Resolution
- 1 LSB Integral Non-linearity
- 2 LSB Absolute Accuracy
- $13\mu s - 260\mu s$ Conversion Time
- Up to 76.9kSPS (Up to 15kSPS at Maximum Resolution)
- 16 Multiplexed Single Ended Input Channels
- 14 Differential input channels
- 4 Differential Input Channels with Optional Gain of 10x and 200x
- Optional Left Adjustment for ADC Result Readout
- 0V V_{CC} ADC Input Voltage Range
- 2.7V V_{CC} Differential ADC Voltage Range
- Selectable 2.56V or 1.1V ADC Reference Voltage
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

1.2 The main ATmega2560 features

A 10-bit successive approximation ADC. The ADC converts an analog input voltage to a 10-bit digital value through successive approximation.

A successive-approximation ADC uses a comparator to successively narrow a range that contains the input voltage. At each successive step, the converter compares the input voltage to the output of an internal digital to analog converter which might represent the midpoint of a selected voltage range. At each step in this process, the approximation is stored in a successive approximation register (SAR). For example, consider an input voltage of 6.3 V and the initial range is 0 to 16 V. For the first step, the input 6.3 V is compared to 8 V (the midpoint of the 0-16 V range). The comparator reports that the input voltage is less than 8 V, so the SAR is updated to narrow the range to 0-8 V. For the second step, the input voltage is compared to 4 V (midpoint of 0-8). The comparator reports the input voltage is above 4 V, so the SAR is updated to reflect the input voltage is in the range 4-8 V. For the third step, the input voltage is compared with 6 V (halfway between 4 V and 8 V); the comparator reports the input voltage is greater than 6 volts, and search range becomes 6-8 V. The steps are continued until the desired resolution is reached.

16 Multiplexed Single Ended Input Channels. The ADC is connected to an 8/16-channel Analog Multiplexer which allows eight/sixteen single-ended voltage inputs. The analog input channel is selected by writing to the MUX bits in ADMUX and ADCSRB. Any of the ADC input pins, as well as GND and a fixed bandgap voltage reference, can be selected as single ended inputs to the ADC. A selection of ADC input pins can be selected as positive and negative inputs to the differential amplifier. If differential channels are selected, the voltage difference between the selected input channel pair then becomes the analog input to the ADC. If single ended channels are used, the amplifier is bypassed altogether.

Four Differential Input Channels with Optional Gain of 10x and 200x. The device also supports 16/32 differential voltage input combinations. Four of the differential inputs (ADC1 & ADC0, ADC3 & ADC2, ADC9 & ADC8 and ADC 11 & ADC 10) are equipped with a programmable gain stage, providing amplification steps of 0 dB (1x), 20 dB (10x) or 46 dB (200x) on the differential input voltage before the ADC conversion. The 16 channels are split in two sections of 8 channels where in each section seven differential analog input channels share a common negative terminal (ADC1/ADC9), while any other ADC input in that section can be selected as the positive input terminal. If 1x or 10x gain is used, 8 bit resolution can be expected. If 200x gain is used, 7 bit resolution can be expected.

Selectable 2.56V or 1.1V ADC Reference Voltage. Internal reference voltages of nominally 1.1V, 2.56V or A VCC are provided On-chip. The voltage reference may be externally decoupled at the AREF pin by a capacitor for better noise performance. The minimum value represents GND and the maximum value represents the voltage on the AREF pin. Optionally, AVCC or an internal reference voltage (1.1V or 2.56V) may be connected to the AREF pin by writing to the REFSn bits in the ADMUX Register that can help to improve noise immunity.

Interrupt on ADC Conversion Complete. The ADC has its own interrupt which can be triggered when a conversion completes.

The ADC contains a Sample and Hold circuit which ensures that the input voltage to the ADC is held at a constant level during conversion. A block diagram of the ADC is shown in Figure 1.

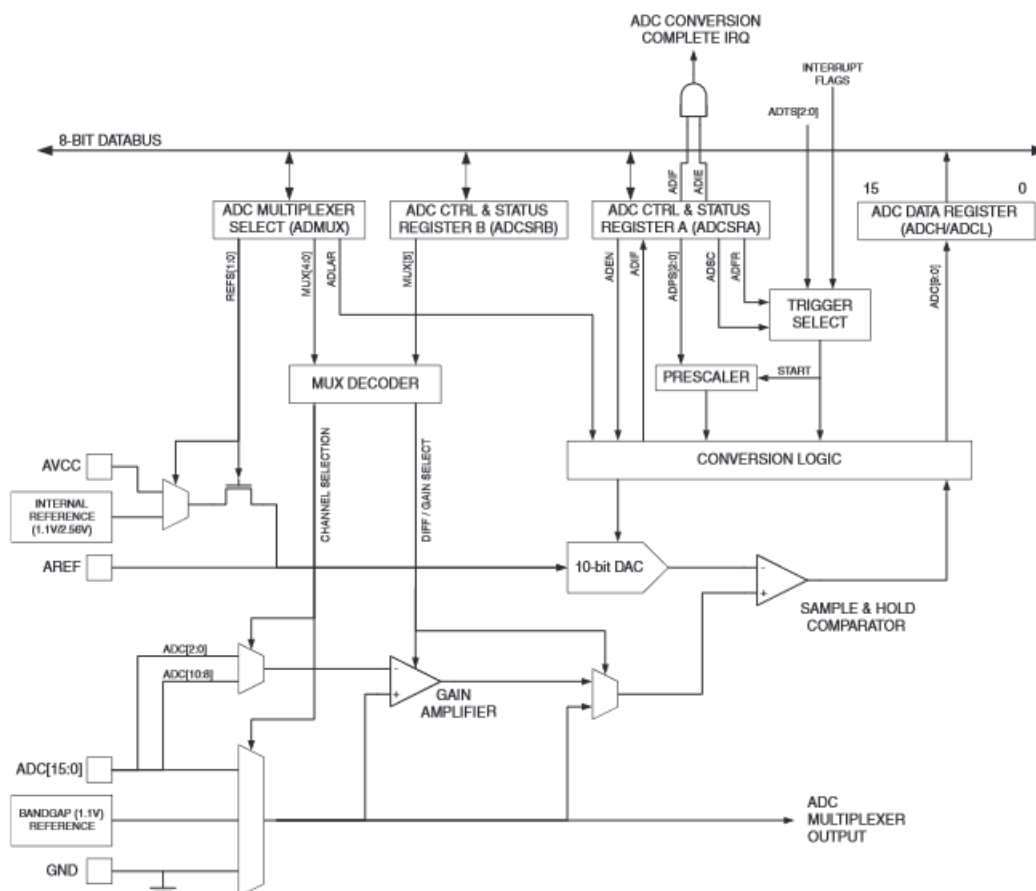


Figure 1: Analog to Digital Converter Schema

2 Controlling the conversion frequency

Before using the ADC it is necessary to select the reference voltage source and the ADC clock frequency. The voltage reference source options (Figure 2) can be determined by setting certain bits 6 and 7 in the ADCMUX register shown in Figure 3.

REFS1	REFS0	Voltage Reference Selection ⁽¹⁾
0	0	AREF, Internal V_{REF} turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Internal 1.1V Voltage Reference with external capacitor at AREF pin
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Figure 2: ADC reference voltage source

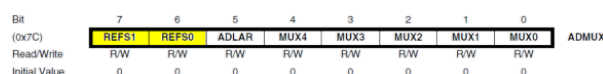


Figure 3: ADC multiplexer register, showing the reference source bits

The ADC can be operated in the following three modes:

- Single conversion: Started by writing a logical one to the ADC Start Conversion bit
- Triggered conversion: The conversion is initiated by the rising edge of the trigger input
- Free running: The next conversion takes place immediately after the previous conversion is finished

The ADC prescale controls the internal ADC clock that controls the conversion process. By default, the successive approximation circuitry requires an input clock frequency between 50kHz and 200kHz. If a lower resolution than 10 bits is needed, the input clock frequency to the ADC can be as high as 1000kHz to get a higher sample rate. The conversion process requires 13-14 ADC clock cycles and this sets an upper limit on the sampling frequency. The pre-scale options are shown in Figure 4.

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 4: ADC prescale values

The certain prescale value will give ADC clock frequency. For example, when we use a 16MHz oscillator then use a prescale value of 128 (taken from the Figure 4) will give ADC clock frequency of 125kHz (see a equation below) which is well within the 50kHz and 200kHz limits.

$$16\text{MHz}/128 = 125\text{kHz the ADC reference clock}$$

- $16 \text{ MHz} / 2 = 8 \text{ MHz}$
- $16 \text{ MHz} / 4 = 4 \text{ MHz}$
- $16 \text{ MHz} / 8 = 2 \text{ MHz}$
- $16 \text{ MHz} / 16 = 1 \text{ MHz}$
- $16 \text{ MHz} / 32 = 500 \text{ kHz}$
- $16 \text{ MHz} / 64 = 250 \text{ kHz}$
- $16 \text{ MHz} / 128 = 125 \text{ kHz}$

It is also necessary to carry out a single conversion by setting the ADSC bit in the ADCSRA register

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 5: ADC Control and Status Register, with clock prescale bits highlighted

Setting bits happens by using command below (as an example 128 prescale value): $ADCSRA = ((1 \ll ADPS2) | (1 \ll ADPS1) | (1 \ll ADPS0))$; A normal conversion in the ADC takes 13 ADC clock cycles. The first conversion takes 25 clock cycles (see Figure 6).

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

Figure 6: ADC Conversion Time

So our ADC clock speed needs to be factored down by 13 to figure out how many samples we will be able to achieve per second. The Arduino library is enabling the ADC with a 128 prescaler, giving a 125 kHz ADC clock speed when the core clock is 16 Mhz. A 125 kHz clock speed will result in $125 \text{ kHz} / 13 = 9600 \text{ Hz}$ sample speed. The hardware limit is 9,600 samples per second or 96 samples per millisecond. We cannot physically do better than that at the desired 10 bit resolution. As we can see, we got 9600Hz sample speed, it is the same value, which we will use in our programming tasks.

3 Changing the input voltage

Once the ADC is initialised single conversions on any channel are carried out by first selecting the desired channel, it is starting the conversion by writing a logical one to the ADC Start Conversion bit, ADSC. It named the conversion complete flag (ADSC bit in the ADCSRA register, shown in figure 5). Until the conversion is complete ADSC=1 and stays high as long as the conversation is in progress and will be set to ADSC=0 when the conversion is complete, will be cleared by hardware. Then finally it will read the converted value. If a different data channel is selected while a conversation is in progress, the ADC will finish the current conversation before performing the channel change.

Alternatively, a conversion can be triggered automatically by Auto Triggering, which can be enabled by setting the ADC Auto Trigger Enable bit, ADIF in ADCSRA. When a positive edge occurs on the selected trigger signal, the ADC prescaler is reset and a conversion is started. This provides a method of starting conversions at fixed intervals. If the trigger signal still is set when the conversion completes, a new conversion will not be started. If another positive edge occurs on the trigger signal during conversion, the edge will be ignored.

If we would like to measure input signals using differential channels, the certain type of conversion should be taken into consideration. We will use differential conversions that are synchronized to the internal clock. A conversion initiated that there are all single conversions, and the first is free running conversion. So, when the clock is low will take the same amount of time as a single ended conversion (13 ADC clock cycles), if we will have another conversion with another signal, the clock will be high and will take more time due to the synchronization mechanism (14 ADC clock cycles).

4 The scaling of the result

After the conversion is complete (ADIF is high), the conversion result can be found in the ADC

Result Registers (ADCL, ADCH).

For single ended conversion, the result is:

$$ADC = \frac{V_{IN} \cdot 1023}{V_{REF}}$$

Because of we have 10-bit resolution, we have $2^{10} = 1024$ possible values for each analog value in the range $(0 - V_{REF})$ Volts we have $(0 - 1023)$ decimals numbers.

In this equation V_{IN} means the voltage on the selected pin (Analog input pins) and V_{REF} is selected reference voltage (1.1V or 2.5V), depends on the programmed registers. For example: If we have input constant voltage taken from the power supply equals 1V and reference voltage chosen 2.5V in this case ADC output will be:

$$ADC = \frac{1V \cdot 1023}{2.5V} = \frac{1023}{2.5} = 409.2$$

ADC output presented in decimal number is 409.2. This is a digital number, which we can get from the AVR microcontroller on the desktop PC.

If differential channels are used, the result is:

$$ADC = \frac{(V_{POS} - V_{NEG}) \cdot Gain \cdot 1023}{V_{REF}}$$

In this equation V_{POS} means positive voltage, V_{NEG} is negative voltage are two differential input channels, Gain can be selected as 1x, 10x or 200x and V_{REF} is selected reference voltage (1.1V or 2.5V), depends on the programmed registers.

5 Programming the circuit

Usually, to make programming easier, programmers create a separate method for initializing necessary variables or to program necessary registers for microcontrollers bits.

```
// turn on the Internal 2.56V voltage reference
ADMUX = (1 << REFS1) | (1 << REFS0) | port;

// set frequency 1MHz / 8 = 125kHz
ADCSRA = (1<<ADPS1) | (1<<ADPS0);

ADCSRA |= 1 << ADEN; // ADC Enable
```

Figure 7: Example of a method called Init_ADC

Here we have a piece of code, which gives certain values to registers important for us. In ADMUX multiplexer register we can first of all choose the internal reference voltage for measurements, by setting values to pins (REFS1 and REFS2), or we can switch on/off MUXn (MUX0, MUX1, MUX2, MUX3, MUX4) to read bits from new channels see Figure 7 below.

$ADMUX = (1 \ll MUX0) | (1 \ll MUX3)$ means we set our mux into position 4, to work with ADC channel number three to read analog input from ADC3 and convert it to the digital signal.

ADCSRA give us an opportunity to program the conversion frequency by setting bits into ADPS1, ADPS2 and ADPS3 according Figure 5. Also we can enable and disable ADC in our microcontroller by setting value into ADEN register. Others programming principles will be shown in the programming exercises below.

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000	N/A	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	1x
11110	1.22 V (V_{BG})	N/A		
11111	0 V (GND)			

Figure 8: Example of a method called Init_ADC

6 Programming tasks

6.1 The first task

Listing 1: L8_1.c

```

#include "L4_123.h"
#include "L8_1.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define _CRT_SECURE_NO_WARNINGS
#define F_CPU 16000000

int ADC_Read(void)
{
    int result = 0;

    // reset the converter
    ADCSRA |= (1<<ADSC); //
    while(!(ADCSRA & (1<<ADIF))); // ADIF turns on after reset
    ADCSRA |= (1<<ADSC); // start conversion
    while(!(ADCSRA & (1<<ADIF))); // ADIF turns on after conversion
    result = ADCL; // read 1st lower part
    result += (ADCH << 8); // read upper part
    return result;
}

void ADC_Init(char port)
{
    if (port < 15)
    {
        ADMUX = (1 << REFS1) | (1 << REFS0) | port; // ref Volage 2.5V

        // set frequency 1MHz / 8 = 125kHz
        ADCSRA = (1<<ADPS1) | (1<<ADPS0); // prescale

        ADCSRA |= 1 << ADEN; // ADC Enable
    }
}

```

Listing 2: main.c

```

#include "L4_123.h"
#include "L8_1.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

```

```

int main (void)
{
    int result;
    // variable to read integer ADC value
    ADC_Init(1);
    // initialize the port what we want

    double voltage;
    // double type value to convert ADC value
    char string[30]="ADC starts.. ";
    // print the string in the beginning
    SetLineParameters( 0, 9600, EVEN_PARITY, BIT_FORMAT_8bit, STOP_BIT_2bit );
    // init board
    WriteString((unsigned) 0, string);
    // write the started string in the terminal

    while(1)
    {
        result=ADC_Read();
    // read integer
        voltage=(double)result;
    // convert to double
        voltage=(voltage*2.5)/1023;
    // convert to decimal
        sprintf(string, "\rResult = %.5f Volts", voltage);
    // set double value to string
        WriteString((unsigned) 0, string);
    // write the voltage on the screen
        _delay_ms(1000);
    // wait 1 second before clean the screen
        WriteString((unsigned) 0, "\033[2J");
    // clean a screen to measure again
    }
}

```

6.2 The second task

Listing 3: L8_2.c

```

#include "L4_123.h"
#include "L8_2.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define _CRT_SECURE_NO_WARNINGS
#define F_CPU 16000000

int ADC_DiffRead(void)
{
    ADCSRA|=(1<<ADSC); // start conversion
    while(!(ADCSRA&(1<<ADSC))); // wait conversion
}

```

```

        return ADCW;
    }

void ADC_DiffInit(char port1, char port2)
{
    if ((port1 == 0)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC0 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 1)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC1 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 2)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX1)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC2 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 3)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX1)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC3 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 4)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX2)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC4 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 5)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX2)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC5 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 6)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX1)|(1<<MUX2)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC6 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
    if ((port1 == 7)&&(port2 == 1))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX1)|(1<<MUX2)|(1<<MUX4);
        // enable Vref=5 volts and MUX for difference between ADC7 and ADC1 with gain x1
        ADCSRA |= (1 << ADEN); // ADC Enable
    }
}

```

```

        if ((port1 == 0)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC0 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
        if ((port1 == 1)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC1 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
        if ((port1 == 2)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX1)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC2 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
        if ((port1 == 3)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX1)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC3 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
        if ((port1 == 4)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX2)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC4 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
        if ((port1 == 5)&&(port2 == 2))
        {
            ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX2)|(1<<MUX3)|(1<<MUX4);
            // enable Vref=5 volts and MUX for difference between ADC4 and ADC2 with gain x1
            ADCSRA |= (1 << ADEN); // ADC Enable
        }
    }
}

```

Listing 4: main.c

```

#include "L4_123.h"
#include "L8_2.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

int main (void)
{
    int result;

    //task 2
    ADC_DiffInit(2,1);
    // define which ports we measure

```

```

        double voltage;
// double type value to convert ADC value
        char string[30]="ADC starts.. ";
// print the string in the beginning
        SetLineParameters( 0, 9600, EVEN_PARITY, BIT_FORMAT_8bit, STOP_BIT_2bit );
// init board
        WriteString((unsigned) 0, string);
// write the started string in the terminal

        //Task 2
        while(1)
        {
                result=ADC_DiffRead();
// read integer difference value
                voltage=(double)result;
// convert to double
                voltage=(voltage*5)/511;
// convert to decimal
                sprintf(string, "\rResult = %.5f Volts", voltage);
// set double value to string
                WriteString((unsigned) 0, string);
// write the voltage on the screen
                _delay_ms(1000);
// wait 1 second before clean the screen
                WriteString((unsigned) 0, "\033[2J");
// clean a screen to measure again

        }
}

```

6.3 The third task

Listing 5: L8_3.c

```

#include "L4_123.h"
#include "L8_3.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define _CRT_SECURE_NO_WARNINGS
#define F_CPU 16000000

void ADC_GainInit(char port1, char port2)
{
        // set frequency 1MHz / 8 = 125kHz
        ADCSRA = (1<<ADPS1) | (1<<ADPS0);
// prescale

        if ((port1 == 0)&&(port2 == 0))

```

```

    {
        ADMUX =(1 << REFS0)|(1<<MUX3);
// enable Vref=5 volts and MUX for difference between ADC0 and ADC1 with gain x10
    }
    if ((port1 == 1)&&(port2 == 0))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX3);
// enable Vref=5 volts and MUX for difference between ADC1 and ADC1 with gain x10
    }
    if ((port1 == 2)&&(port2 == 2))
    {
        ADMUX =(1 << REFS0)|(1<<MUX2)|(1<<MUX3);
// enable Vref=5 volts and MUX for difference between ADC2 and ADC1 with gain x10
    }
    if ((port1 == 3)&&(port2 == 2))
    {
        ADMUX =(1 << REFS0)|(1<<MUX0)|(1<<MUX2)|(1<<MUX3);
// enable Vref=5 volts and MUX for difference between ADC3 and ADC1 with gain x10
    }

    ADCSRA |= (1 << ADEN); // ADC Enable
}

int ADC_Read(void)
{
    ADCSRA|=(1<<ADSC); // start conversion
    while (!(ADCSRA&(1<<ADSC))); // wait conversion
    return ADCW;
}
//-----
int Voltage(unsigned gain)
{
    if (gain==10) // if gain 10, all MUX1 sets to zero for both ports
        ADMUX|=(0<<MUX1);
    else
        ADMUX|=(1<<MUX1); // if gain 200, all MUX1 sets to 1 for both ports
    return ADC_Read(); // and we go to reading ADC values
}

```

Listing 6: main.c

```

#include "L4_123.h"
#include "L8_3.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

int main (void)
{
    ADC_GainInit(1,0);
// initialize two ports to measure difference between them with gain
    double voltage;
// double type value to convert ADC value

```

```

        char string[30]="ADC starts.. ";
// print the string in the beginning
        SetLineParameters( 0, 9600, EVEN_PARITY, BIT_FORMAT_8bit, STOP_BIT_2bit );
// init board
        WriteString((unsigned) 0, string);
// write the started string in the terminal

        while(1)
        {
                voltage=(double)Voltage(10);
// read integer ADC with gain x10 or x200, convert to double
                voltage=(voltage*5)/1023;
// transform to decimal
                sprintf(string, "\rResult = %.5f Volts", voltage);
// set to string
                WriteString((unsigned) 0, string);
// print the voltage
                _delay_ms(1000);
// wait a second before cleaning a screen
                WriteString((unsigned)0, "\033[2J");
// clear the terminal to measure again

        }

}

```

6.4 The fourth task

Listing 7: L8_4.c

```

#include "L4_123.h"
#include "L8_4.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

#define _CRT_SECURE_NO_WARNINGS
#define F_CPU 16000000

int ADC_Read(void)
{
        int result = 0;

        // reset the converter
        ADCSRA |= (1<<ADSC); //
        while(!(ADCSRA & (1 << ADIF)));
// ADIF turns on after reset
        ADCSRA |= (1<<ADSC); // start conversion
        while(!(ADCSRA & (1 << ADIF)));
// ADIF turns on after conversion
        result = ADCL;
}

```



```

// read 1st lower part
    result += (ADCH << 8);
// read upper part
    //ADCSRA &= ~(1<<ADEN);
// AD-converter off
    return result;
}

void ADC_Init(char port)
{
    if (port < 15)
    {
        ADMUX = (1 << REFS1) | (1 << REFS0) | port;

        // set frequency 1MHz / 8 = 125kHz
        ADCSRA = (1<<ADPS1) | (1<<ADPS0); // prescale

        ADCSRA |= 1 << ADEN; // ADC Enable
    }
}

```

Listing 8: main.c

```

#include "L4_123.h"
#include "L8_4.h"
#include <asf.h>
#include <stdio.h>
#include <avr/delay.h>
#include <avr/io.h>
#include <avr/interrupt.h>

int main (void)
{
    int result;
    // variable to read integer value from ADC

    int i=0;
    // variable to initialize ADC ports in a loop

    double voltage;
    // double type value to convert ADC value
    char string[30]="ADC starts.. ";
    // print the string in the beginning
    SetLineParameters( 0, 9600, EVEN_PARITY, BIT_FORMAT_8bit, STOP_BIT_2bit );

    // init board
    WriteString((unsigned) 0, string);
    // write the started string in the terminal

    while (1) // infinite loop
    {
        for (i=0;i<8;i++)
        {
            ADC_Init(i);

```

```

// initialize the new port
                                result=ADC_Read();
// read a new port
                                voltage=(double)result;
// convert the integer value to double
                                voltage=(voltage*2.5)/1023;
// convert to decimal
                                sprintf(string, "\r\nResult ADC%i = %.5f Volts", i, voltage);
// set to the string
                                WriteString((unsigned) 0, string);
// show the string on a screen
                                }
                                _delay_ms(10000);
// wait 10 seconds
                                WriteString((unsigned) 0, "\033[2J");
// clear the terminal window for new values
                                }
}

```

ADC0, ADC1 are Grounded; ADC2, ADC3 have voltages from the power source and the rest ADCs pins are not grounded, that's why they have value equals to Vref.

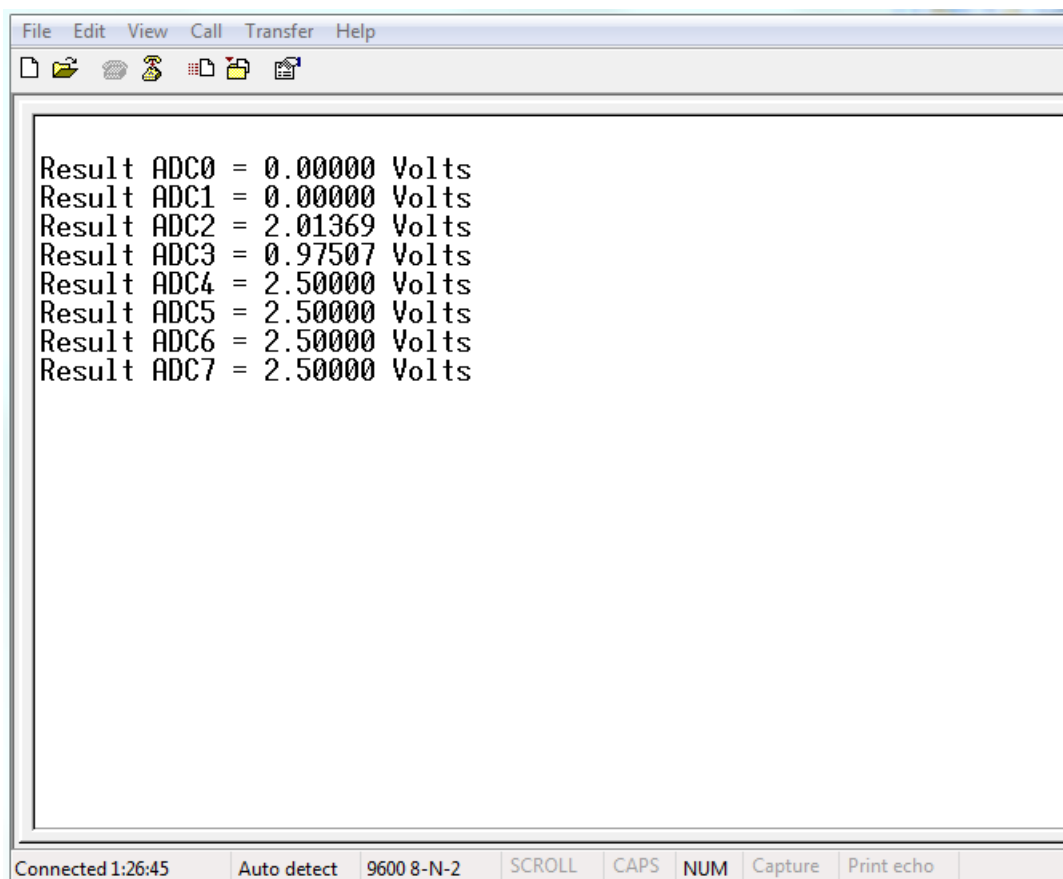
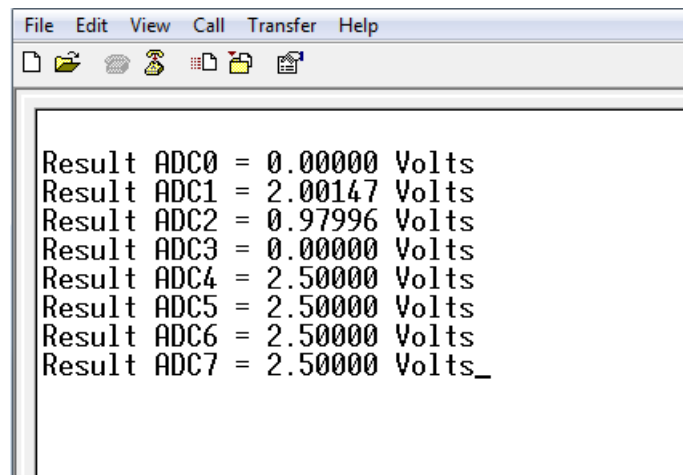


Figure 9: Results of the measurement

We can change inputs values during 10 seconds no changes will be shown, but after 10 seconds new values will be shown in the terminal. ADC0, ADC3 are Grounded; ADC1, ADC2 have voltages from the power source and the rest

ADCs pins are not grounded, that's why they have value equals to V_{ref} .



```
Result ADC0 = 0.00000 Volts
Result ADC1 = 2.00147 Volts
Result ADC2 = 0.97996 Volts
Result ADC3 = 0.00000 Volts
Result ADC4 = 2.50000 Volts
Result ADC5 = 2.50000 Volts
Result ADC6 = 2.50000 Volts
Result ADC7 = 2.50000 Volts_
```

Figure 10: Results of the measurement