Crystal Clear Electronics

The development of the Crystal Clear Electronics curriculum was supported by the European Commission in the framework of the Erasmus + programme in connection with the “Developing an innovative electronics curriculum for school education” project under “2018-1-HU01-KA201-047718” project number.

The project was implemented by an international partnership of the following 5 institutions:

- Xtalin Engineering Ltd. – Budapest
- ELTE Bolyai János Practice Primary and Secondary Grammar School – Szombathely
- Bolyai Farkas High School – Târgu Mureș
- Selye János High School – Komárno
- Pro Ratio Foundation working in cooperation with Madách Imre High School – Šamorín

Contact http://crystalclearelectronics.eu/en/
info@kristalytisztaelektronika.hu

22 - Processing Analog Signals, the ADC

Written by Erzsébet Szomorú-Ozsváth

English translation by Xtalin Engineering Ltd.

Revised by Gábor Proksa, Gergely Lágler

INTRODUCTION

We often face the problem in real-life where we want to measure some physical quantity, like what is the temperature, how noisy the street is, what color a petal is, or how tall someone is. These values usually vary in time. If we measure them continuously, these quantities can be represented by analog signals. An **analog signal** is a continuous physical quantity that is a continuous function of time. This means, that the analog signal takes a value at any given moment, and this value may even be different at any given moment.

We want the measured values to be displayed, stored in a user-friendly format, or to be transmitted to other devices such as a computer, microprocessor, FPGA, etc. for further processing. Since these devices operate with digital signals, therefore analog-to-digital conversion is essential for processing analog signals. Generally, the analog signal is a voltage to which a number is assigned, so the signal is converted to a series of number.

A **digital signal** is a signal that consists of a discrete (multiples of a unit) set of values, so it can only take a finite number of values.

In practice, we always have to make a trade-off when mapping analog signals to digital, because we need to choose the range and accuracy of the measurement. Of course, when measuring something we want to approach the real value as closely as possible, so we want to minimize the difference between the real value and the measured value, in other words the **measurement error**.

However, it is not the same if we want to measure the height of a house with an accuracy of meters or the height of a new-born with an accuracy of cm. When measuring the house, we do not start measuring with millimeter paper, while it is pointless to measure a new-born baby with a one-meter long stick. The range also determines the accuracy of the measurement, i.e. the **resolution**. We can see that it is important to know the parameters of the signal we want to measure in order to select the appropriate measuring device.

For example, if we want to measure the height of a person with a tape measure, the biggest measurement error we can make (if we are making accurate measurements) is a few millimeters because of the resolution of our measuring device (even though we know that people are growing continuously, and not in centimeter or millimeter increments). We can't read smaller values than a millimeter from the tape measure.

Another example could be that even though we see the temperature rise on the mercury display of a traditional outdoor thermometer continuously, due to the scale the temperature can be read only with

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

the accuracy of one degree Celsius or one tenth of a degree Celsius. We are trying to describe a continuous quantity by its discrete value, that is, **quantize** it.

As shown in the examples above, many signals are not available in the form of an electrical signal. In such cases it has to be converted into an electrical signal using appropriate transducers or sensors (see the chapter on sensors for more details). Then, this continuous electrical signal is connected to the input of an **Analog-Digital Converter** (ADC). Usually the signal range can only be positive, but there are also bipolar converters that operate in the negative range as well.

THE ANALOG-DIGITAL CONVERTER

The ADC is a converter which converts the continuous electrical signal connected to its input to a digital signal and the digital equivalent of the analog signal measured periodically (sampling time) appears on its output. The figure below shows a signal connected to an ADC input in green and the signal appearing on the output in red. Time is on the x-axis and the signal value is on the y-axis. After marking the maximum of the signal range, the range is divided into 8 equal parts. The measurement range is the range between the minimum and maximum value of the signal to be measured, in which we want to measure with a certain accuracy. The digital output shows which small range, interval, the measured signal falls into. For example, a value of 3 means that the output is in the third interval.

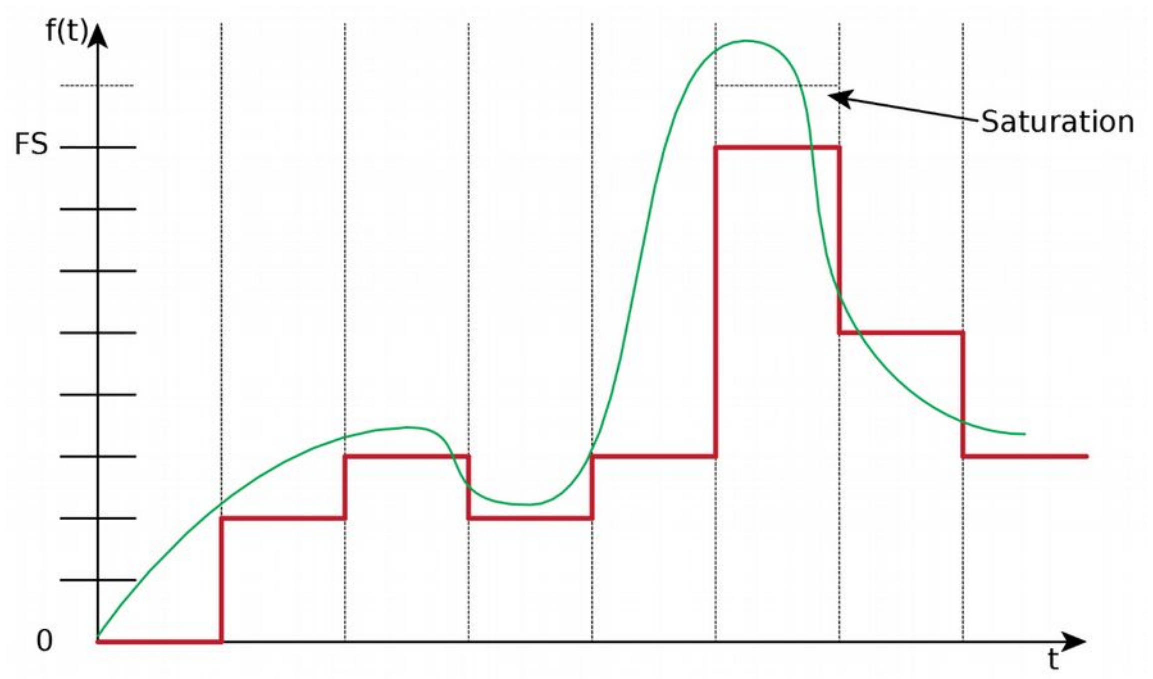


Figure 1 – Input (green) and output (red) signal of the ADC

This example highlights several problems we face during analog-digital conversion. First of all, the signal we received does not look like the input, the shape is different, there is a lot of **distortion**. The figure above shows that the peak amplitude of our input signal is greater than the maximum of the selected measurement range, so the output signal gone into “**saturation**”, which means it has reached the highest possible value. If this happens, we have no information about the signal above the saturation range, as

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

the ADC cannot show a higher value. This phenomenon indicates that the **Full Scale** of the selected ADC, that is, the maximum input analog voltage the converter can measure without saturation, is too small.

There are two types of ADCs: **unipolar** ADCs can measure between a reference voltage and ground, while **bipolar** ADCs can measure in a symmetric range around ground.

The Atmel microcontroller used in our measurements also has an ADC peripheral. On some of its pins we can measure and digitize the voltage appearing there. The microcontroller datasheet indirectly reveals that it is a unipolar ADC, because it has a pin, called "AREF", which stands for the ADC analog reference voltage input pin, and only a positive voltage can be connected to that pin.

We can decrease the distortion by measuring more frequently, increasing the **sampling frequency**, thus reducing the sampling period. The waveforms with a higher sampling frequency are shown in figure 2.

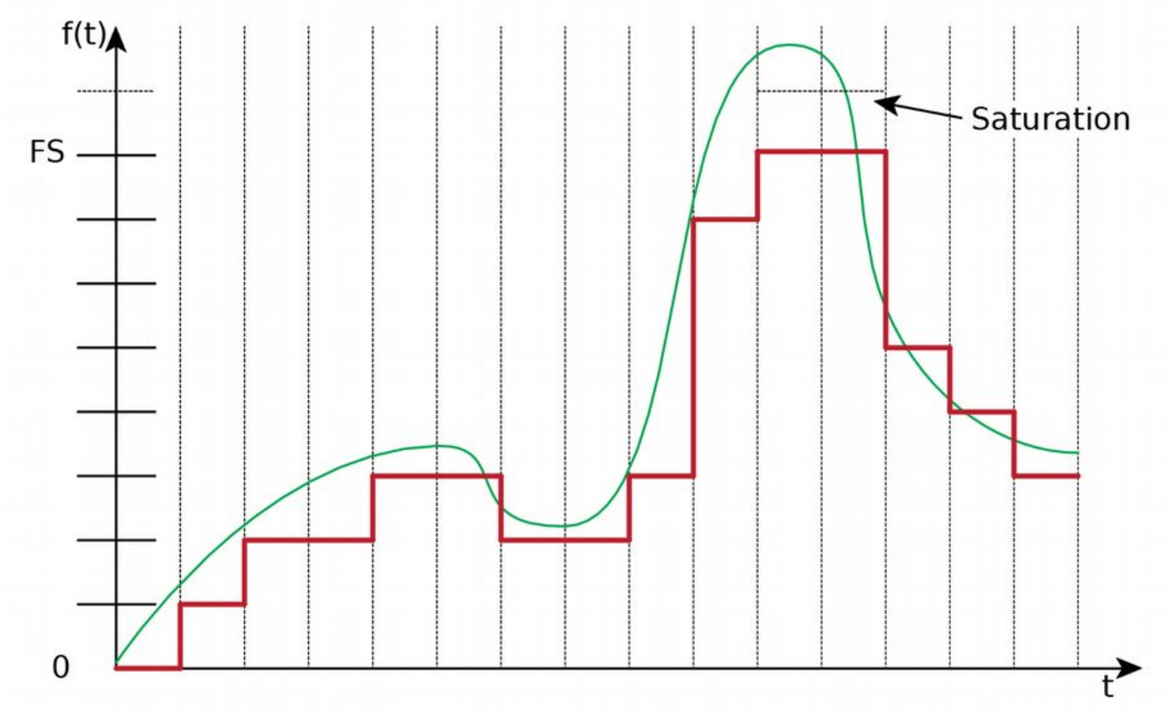


Figure 2 – Increasing the sampling frequency

You read about the operation of the ADC, including how to adjust the sampling frequency in the second half of this chapter.

Another way to decrease the distortion is to make the intervals on the y axis smaller, see Figure 3.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

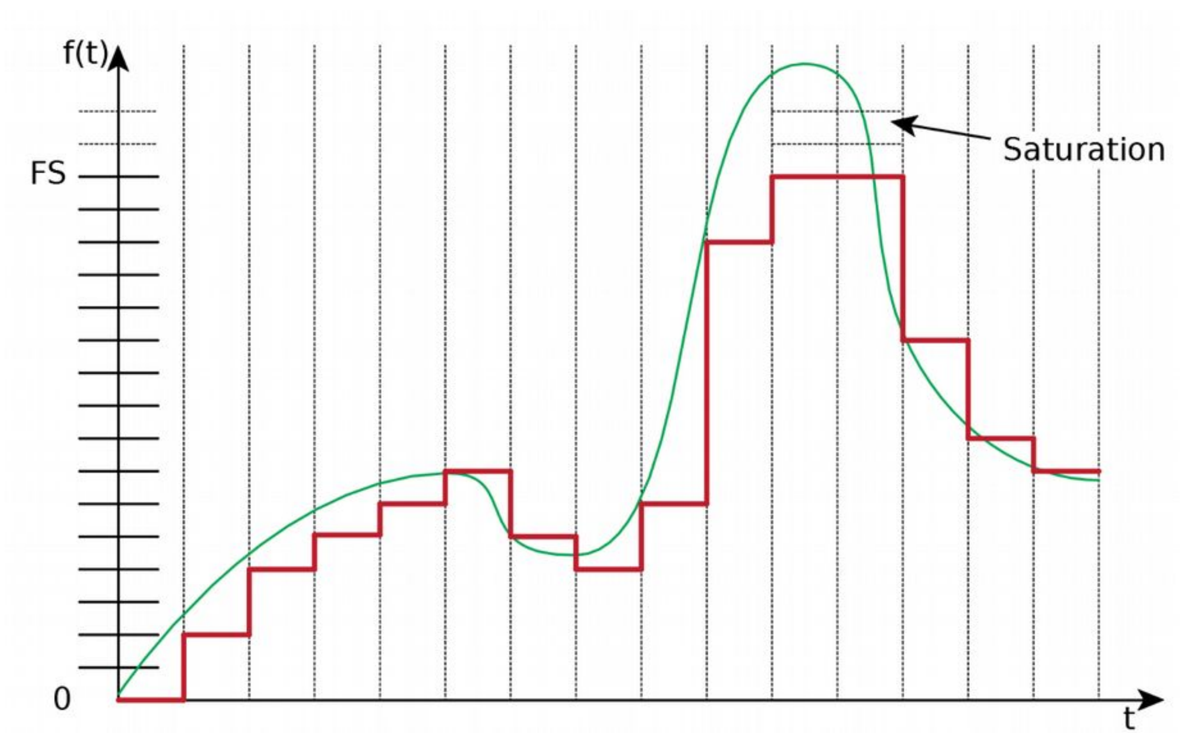


Figure 3 – Increasing the resolution

The number of intervals is called **resolution**, which shows that how many different values the analog input signal can be converted to by the device. This is usually given by **number of bits**. In our example, the 10-bit ADC has a resolution of 2^{10}, so it divides the measuring range into 1024 equal intervals. If the reference voltage is X, then we can measure with X/1024 V accuracy. This brings us to the next important concept, **LSB** (Least Significant Bit). The LSB shows the smallest difference in the input voltage needed to change the lowest bit of the output, i.e. it is the voltage of a digital step. The LSB can be calculated using the following formula:

$$LSB = \frac{KT}{(2^n - 1)},$$

where KT is the Full Scale (FS), and n is the number of bits.

Example

Calculate the LSB of a 10-bit ADC if the reference voltage is 5 V!

Solution:

The input range is 0-5 V, so the full scale (FS) is 5 V, and the number of bits is 10.

$$LSB = \frac{5\ [\mathrm{V}]}{2^{10} - 1} = 0.00489[\mathrm{V}] = 4.89\ [\mathrm{mV}]$$

What would the resolution be if we were only using 8 bits instead of 10?

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

From the above example, it is clearly visible that ADCs have to be selected carefully taking the application specific circumstances into consideration. In reality, an engineer must always pay attention to price as well. The more accurate, higher-resolution analog-to-digital converter we choose, the more expensive it will be. Just think about that the price increase of a component in a computer hardware, which is manufactured by the million will increase the total cost of production by a million times the price increase of the component. As a result, mass-produced products are usually not **overdesigned** (using better, more accurate, stronger than absolutely necessary). Unfortunately, the direct consequence of a price-sensitive market is that manufacturers often only want to meet the minimum requirements, therefore they do not design for safety margins, and this is partly the reason why electronic devices often fail after a few years, and unfortunately, garbage is produced in enormous amounts.

OPERATING MODES

Let's look at the different operating modes of ADCs, especially the ADC in the ATMega16A microcontroller (see for more details Chapter 22, "Analog to Digital Converter" of the datasheet).

Asymmetric mode

The **asymmetric** mode means that the voltage measured on the input channel (pin) is relative to the reference voltage. In this case, separate signals can be connected to each input pin. We will use this mode for the measurements used in this chapter of the curriculum.

Symmetric/differential mode

The other mode is the **differential** or **symmetric mode**, where the input signals run on two wires instead of a single wire. This is especially important for high frequency signals (above 1 - 10 MHz) to avoid unwanted noise. The consequence of this is that we cannot utilize each wire individually, but only in pairs. If more differential channels are required, then another type of microcontroller or an external ADC should be chosen.

How to avoid noise?

In wires, external magnetic and electrical fields can cause relatively large interference even at a short distance. With a certain amount of noise on the receiver side, it's hard to differentiate the original signal from the noise. To avoid this, there are many ways to eliminate noise.

Hardware noise reduction:

One of the most commonly used methods is using shielded wires. A metal sleeve is wrapped around the wire, acting as a Faraday cage. A good example is the commonly known coaxial cable.

Another important noise elimination method is the symmetric signal transmission mentioned above.

A twisted pair consists of two wires spirally twisted together. The point is that the signals of the pairs do not interact. It is used for telephone cables and internet cables. There are two types: Unshielded Twisted Pair (UTP) and Shielded Twisted Pair (STP). The shielded version protects against external noises as well.

In case of printed circuit boards, separate ground layers are usually placed between the high-frequency signal route layers so that the signals do not interact. The effect of the wires on each other can be further reduced if the wires are perpendicular to each other in the different layers.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Software noise reduction:

The most common software noise reduction mode is averaging. In this case, we make more measurements than necessary and average the measured values: a = $\Sigma\ a_n\ /n$

Using the ADC on the Microcontroller

Let's see the main features of the ADC built into the microcontroller:

- 8 channels in asymmetric mode
- 7 differential input channels in symmetrical mode
- successive approximation (we will talk about this later)
- 10 bits
- measurement range up to V_{ref} voltage
- interrupt when conversion is completed
- a maximum sampling rate of 15 ksps at maximum resolution

In this microcontroller the differential mode can be used by placing the negative signal on the ADC1 pin and the other on one of the other 7 channels. However, this mode is only available with the TQFP packaged microcontroller and not with the DIP package. Nevertheless, in our example measurement high noise immunity is not required, the asymmetric mode can be used as it is common in low frequency measurements. See chapter 22 of the datasheet for a more detailed explanation.

The ADC requires 50 to 200 kHz clock for accurate operation. If the 10-bit resolution is not required, the ADC clock may be over 200 kHz. We are satisfied with only 8-bit resolution as well. The conversion takes 13 ADC clock cycles (except for the first conversion because there it takes 25).

Calculation

Let’s calculate the maximum sampling frequency, that can be achieved in 10-bit mode.

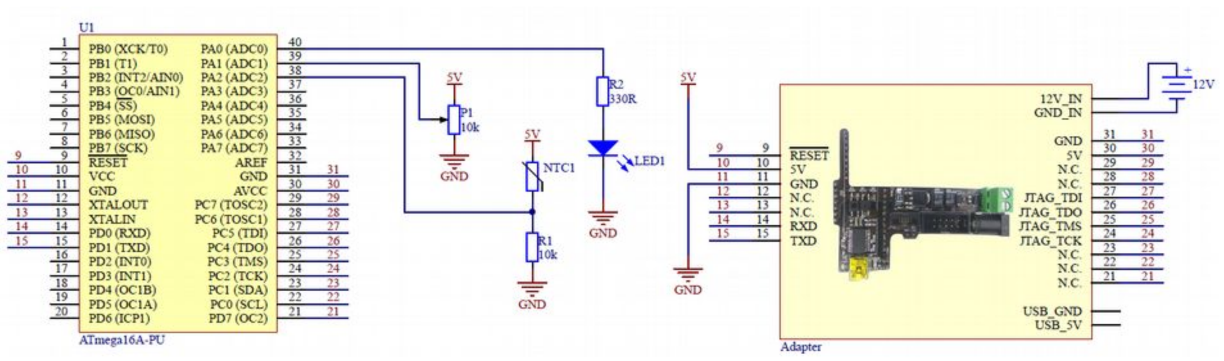
The highest speed can be achieved with the highest possible clock rate, so $f\ =\ 200\ [\mathrm{kHz}]$ is selected as the clock rate. One conversion takes place over 13 clock cycles, so a maximum sampling rate of $200\ \div\ 13\ =\ 15.4\ [\mathrm{ksps(kilosamples\ per\ second)}]$ can be achieved. This is the same as the value given in the datasheet.

In our examples, the built-in RC oscillator is used to generate the microcontroller clock. The operating frequency in this mode can be set with bits `CKSEL3:0` according to table 8.8 of the datasheet. We want to set 8 MHz clock, which is achieved by setting the `CKSEL3:0` bits to `0100`. To produce an 50-200 kHz clock, which is suitable for the ADC, the microcontroller's 8MHz clock is divided by 128 with the prescaler (see later the `ADCEnable()` function), which means a 62.5 kHz clock. From this, the sampling rate is approximately 4.8 ksps.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Measurement 1

Assemble the following circuit.



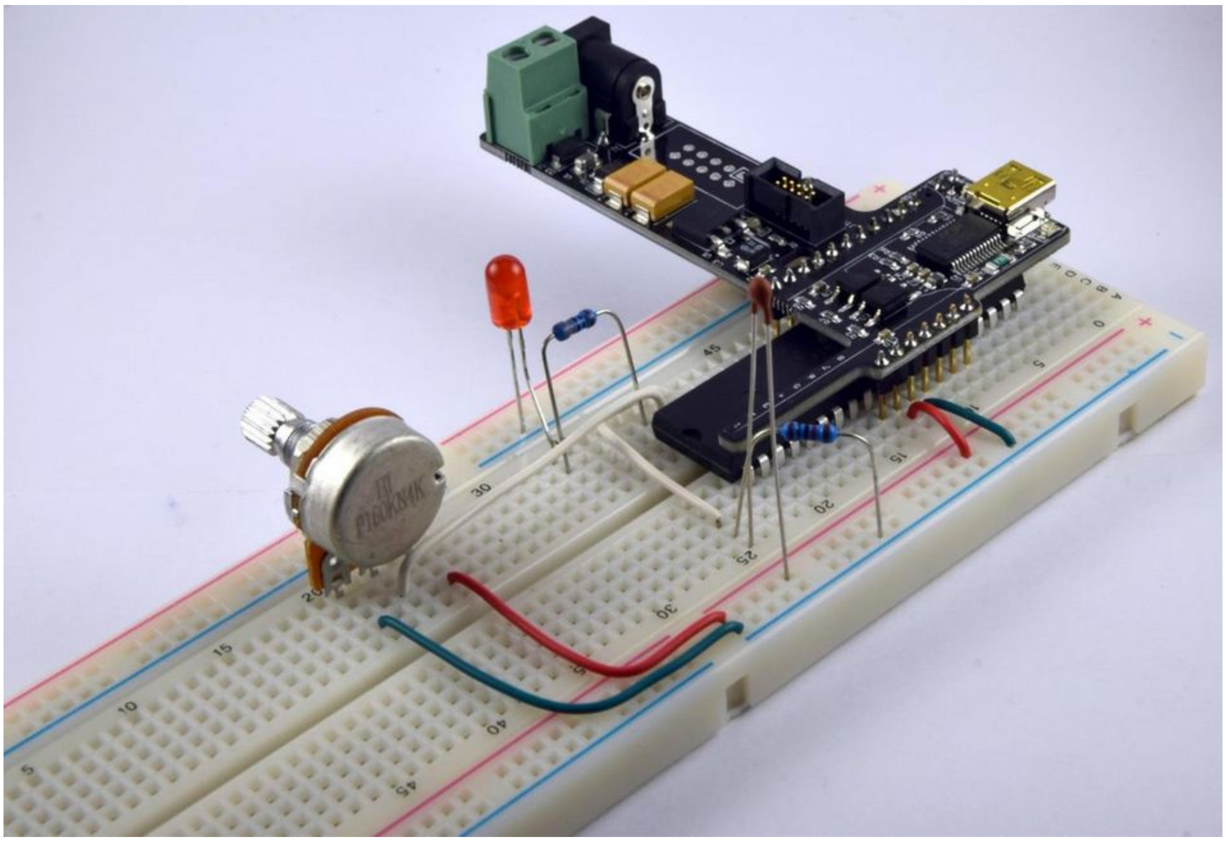


Figure 4 – Measurement layout

For better understanding, we recommend reading the datasheet simultaneously to the solution.

Task:

Compare the voltage of the potentiometer on PA1 (ADC1) with the voltage created by the voltage divider with the NTC. Use A/D conversion for the measurement. If the NTC voltage is greater than that measured voltage on the potentiometer, the LED on the PA0 pin should light up, otherwise it should not light up.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Implementation:

Initializing the IO ports

```
void IOInit()
{
    //Turning off PORTA pull-up resistors on the inputs
    PORTA = 0x00;
    DDRA = 0x00;
    sbi(DDRA, 0);   //The LED is on PA0, PORTA0 should be configured as output (0x01)

    //Turning off PORTB pull-up resistors, and all pins should be configured as inputs
    PORTB = 0x00;
    DDRB = 0x00;

    //Turning off PORTC pull-up resistors, and all pins should be configured as inputs
    PORTC = 0x00;
    DDRC = 0x00;

    //Turning off PORTD pull-up resistors, and all pins should be configured as inputs
    PORTD = 0x00;
    DDRD = 0x00;
}
```

As a first step, we define the operating modes of IO ports, whether they are configured as inputs or outputs, and we give them an initial value, using a technical term we initialize them, see function `IOInit()` in the “io.c” file.

In the `PORTA`, `PORTB`, `PORTC`, `PORTD` data registers, we turn off the pull-up resistors and define the pins as inputs, except for PA0 pin on `PORTA`, this is an output because it controls the status LED.

Initializing the clock

```
void TimerInit()
{
    //1x prescaler
    cbi(TCCR1B, CS12);
    cbi(TCCR1B, CS11);
    sbi(TCCR1B, CS10);

    //Timer1 interrupt on
    sbi(TIMSK, TOIE1);

    //Global interrupt on
    sei();
}
```

The clock is initialized in “timer.c” using the`TimerInit()` function. The `TCCR1B` (Timer Counter Control Register) contains the clock controller settings. The register bits are described in chapter 16.11.2 of the datasheet.

Table 16-6 shows that if no prescaler is needed then `CS12` = 0, `CS11` = 0, `CS10` = 1 must be set in this order. The other bits in the register are not set, they are not needed, so they remain at the default value of 0. As described in chapter 16.11.7, the interrupt of Timer1 is enabled by setting the `TOIE1` bit of the `TIMSK` (Timer / Counter Interrupt Mask Register) register, which is responsible for setting the clock interrupt. Finally, we enable interrupts globally.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Initializing the ADC

```
void ADCInit()
{
    //AVCC reference
    cbi(ADMUX, REFS1);
    sbi(ADMUX, REFS0);

    //ADC1 input
    sbi(ADMUX, MUX0);
    ADC_state = potmeter;

    //Result is shifted to the left -> upper 8 bit in ADCRH
    sbi(ADMUX, ADLAR);
}
```

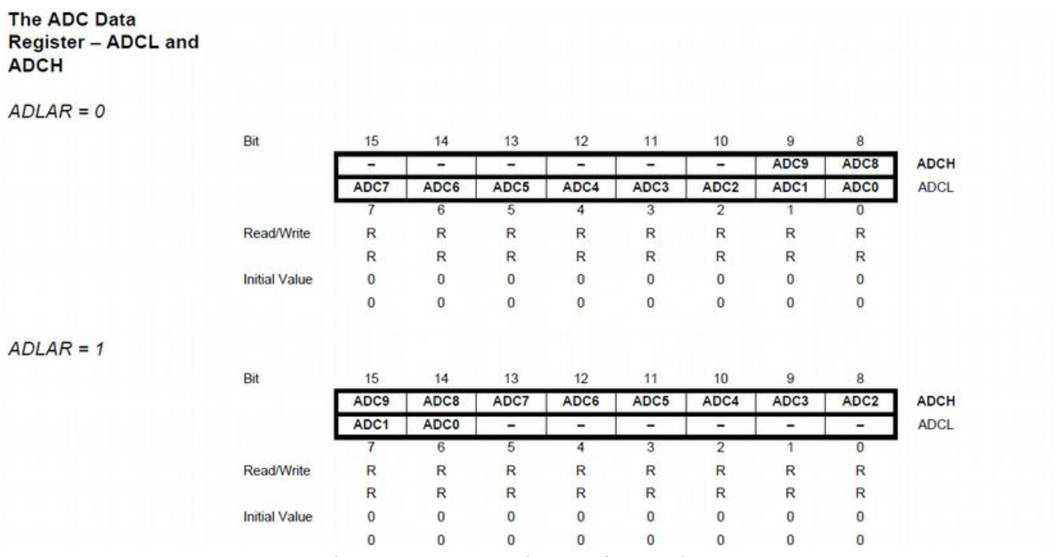
To initialize the ADC, you must first set the ADC reference voltage in the `ADCInit()` function using the ADMUX (ADC Selection Multiplexer Register) register according to chapter 22.9.1, which in our case equals with the external AVCC voltage (`REFS1` = 0, `REFS0` = 1).

Using Table 22.4, we select the register bits to set so that the first channel (ADC1) is the input. For this, `MUX0` must be set to 1, the remaining `MUX` bits can remain at the default value of 0, they do not need to be set.

We set the `ADC_state` flag to `potmeter`. This variable is important to be volatile because it is modified only in the interrupt handler function and since that only access after an interrupt, the compiler will otherwise optimize the variable.

The result of the ADC conversion is put into the `ADCH` and `ADCL` (ADC data register high and low) registers.

The figure below shows how the format of the conversion result is modified by the ADLAR bit of the ADMUX register. For the sake of simplicity, we are now satisfied with the only upper 8 bits.



The ADC Data Register – ADCL and ADCH

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	
	–	–	–	–	–	–	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	–	–	–	–	–	–	ADCL
	7	6	5	4	3	2	1	0	
Read/Write	R	R	R	R	R	R	R	R	
	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	
	0	0	0	0	0	0	0	0	

Figure 5 – ATMega16A ADC data registers
Source: ATMega16A datasheet chapter 22.9

For this reason, it is more practical to use ADLAR = 1 mode so that the result is left-aligned in the ADCH register.

Enabling the ADC

```
void ADCEnable()
{
    //prescaler 128
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS0);

    //Enabling interrupt
    sbi(ADCSRA, ADIE);

    //Enabling the AD converter
    sbi(ADCSRA, ADEN);
}
```

The ADC is enabled by the `ADCEnable()` function. In this function, we set the ADC control and status bits according to chapter 22.9.2 of the datasheet. The prescaler is set to 128 according to Table 22-5 by setting the ADPS0, ADPS1, ADPS2 bits to 1. We enable interrupt and ADC as well.

```
    while (1)
    {
        //If the voltage of the NTC divider is greater than the voltage of the potentiometer,
        // then the LED is on
        if (U_poti > U_NTC)
        {
            sbi(PORTA, 0);
        }

        //Otherwise the LED is off
        else
        {
            cbi(PORTA, 0);
        }
    }
```

In the main function, we start an infinite loop, which controls the LED. If the voltage of the potentiometer is higher than the voltage of the NTC divider, then the LED should light up, otherwise not.

```
ISR(TIMER1_OVF_vect)
{
    //The counter is increased, if it hasn’t reached the maximum yet
    if (cntr < CNTR_MAX)
    {
        cntr++;
    }

    //If it has reached
    else
    {
        cntr = 0;
        //AD conversion is started (sampling)
        ADCStart();
    }
}
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The interrupt is implemented in the ISR(TIMER1_OVF_vect) function. The value of the cntr counter is incremented until it reaches the maximum value (CNTR_MAX), which is defined as a constant value of 10. When the counter has reached this value, we set it to 0 and call the ADCStart() function, which does nothing else than sets the ADSC bit of the ADCSRA register, that is, the ADC conversion is started. The timer settings and the CNTR_MAX value could be different values as well, but then we would start the conversion less or more often.

```
ISR(ADC_vect)
{
    //Saves the measured voltage to the appropriate variable based on the ADC state (channel)
    switch (ADC_state)
    {
    case poti:
        U_poti = ADCH * ADC_CONST;
        break;

    case NTC:
        U_NTC = ADCH * ADC_CONST;
        break;

    default:
        break;
    }

    //Switching to the next ADC channel
    NextCH();
}
```

When the conversion is completed, another interrupt function is called, namely the ISR(ADC_vect) function. In this function, depending on the value of the flag ADC_state, the voltage of either the potentiometer or the NTC is saved in the appropriate variable. The voltage is obtained by multiplying the result of the ADC conversion by the resolution, which in our case is dividing the 5 V range into 256 parts (using only the top 8 bits of the ADC), see the value of ADC_CONST defined in “main.h”. This corresponds to a resolution of 20 mV.

After that we switch to the next channel using NextCH() function.

```
void NextCH()
{
    switch (ADC_state)
    {
    //On ADC1 pin is the potentiometer
    case poti:
        //The next pin is the ADC2, there is the NTC divider
        cbi(ADMUX, MUX0);
        sbi(ADMUX, MUX1);
        ADC_state = NTC;
        break;

    //On ADC2 pin is the NTC divider
    case NTC:
        //The next pin is the ADC1, there is the potentiometer
        cbi(ADMUX, MUX1);
        sbi(ADMUX, MUX0);
        ADC_state = poti;
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

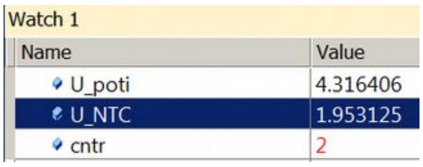
```
        break;

    default:
        break;
    }
}
```

In this case, the input channel and the value of the status flag must be changed.

If the value of the flag `ADC_state` is `potmeter`, so the voltage of the potentiometer has been measured, then it must be switched to the NTC, which means that according to Table 22-4 of the datasheet, the ADC2 (which has the NTC divider) channel must be chosen as input, and the state flag must be set to `NTC`.

In case of `ADC_State = NTC`, the reverse should be done, the input channel will be ADC1 (which has the potentiometer) and the state flag will be `potmeter`.

After getting through the technical details, let's enjoy a bit the results of our work. Put breakpoints to lines, which control the LED in the main function and add the`cntr` counter, `U_potmeter`, `U_NTC` variables to the variables to be monitored (right click on variable name in code and select "Add Watch"). Then our variables will be displayed in the watch window. When running our program in Debug mode, we can see the voltage measured on the potentiometer and the NTC divider in the watch window.

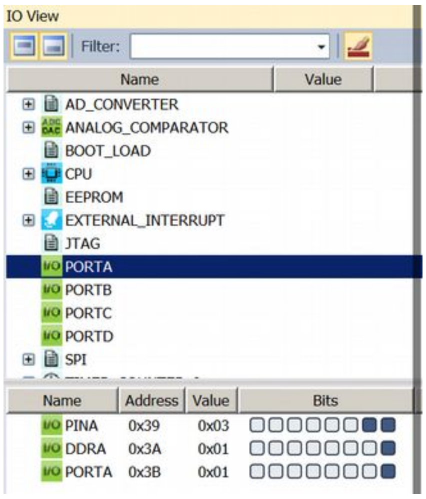


Watch 1

Name	Value
U_poti	4.316406
U_NTC	1.953125
cntr	2

Figure 6 – Watch variables during debug

When the counter has reached 10, one of the measured voltages is updated. Check the last bit of the `PORTA` register (PA0) in the IO View window based on the value of `U_potmeter` and `U_NTC`. We can see if everything has been done well, that if the voltage of the potentiometer (`U_potmeter`) is higher than the voltage measured on the NTC divider (`U_NTC`) then PA0 = 1, so the LED is on, otherwise 0 and the LED is off.



IO View

Filter:

Name	Value
AD_CONVERTER	
ANALOG_COMPARATOR	
BOOT_LOAD	
CPU	
EEPROM	
EXTERNAL_INTERRUPT	
JTAG	
PORTA	
PORTB	
PORTC	
PORTD	
SPI	

Name	Address	Value	Bits
PINA	0x39	0x03	□□□□□□■■
DDRA	0x3A	0x01	□□□□□□□■
PORTA	0x3B	0x01	□□□□□□□■

Figure 7 – PORTA register state in the IO View

As a practice, let's try setting the voltage on the potentiometer to 2 V.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Measurement 2
Pulse Width Modulation using analog-digital converter (ADC_PWM Project)

Let’s assemble the following circuit.

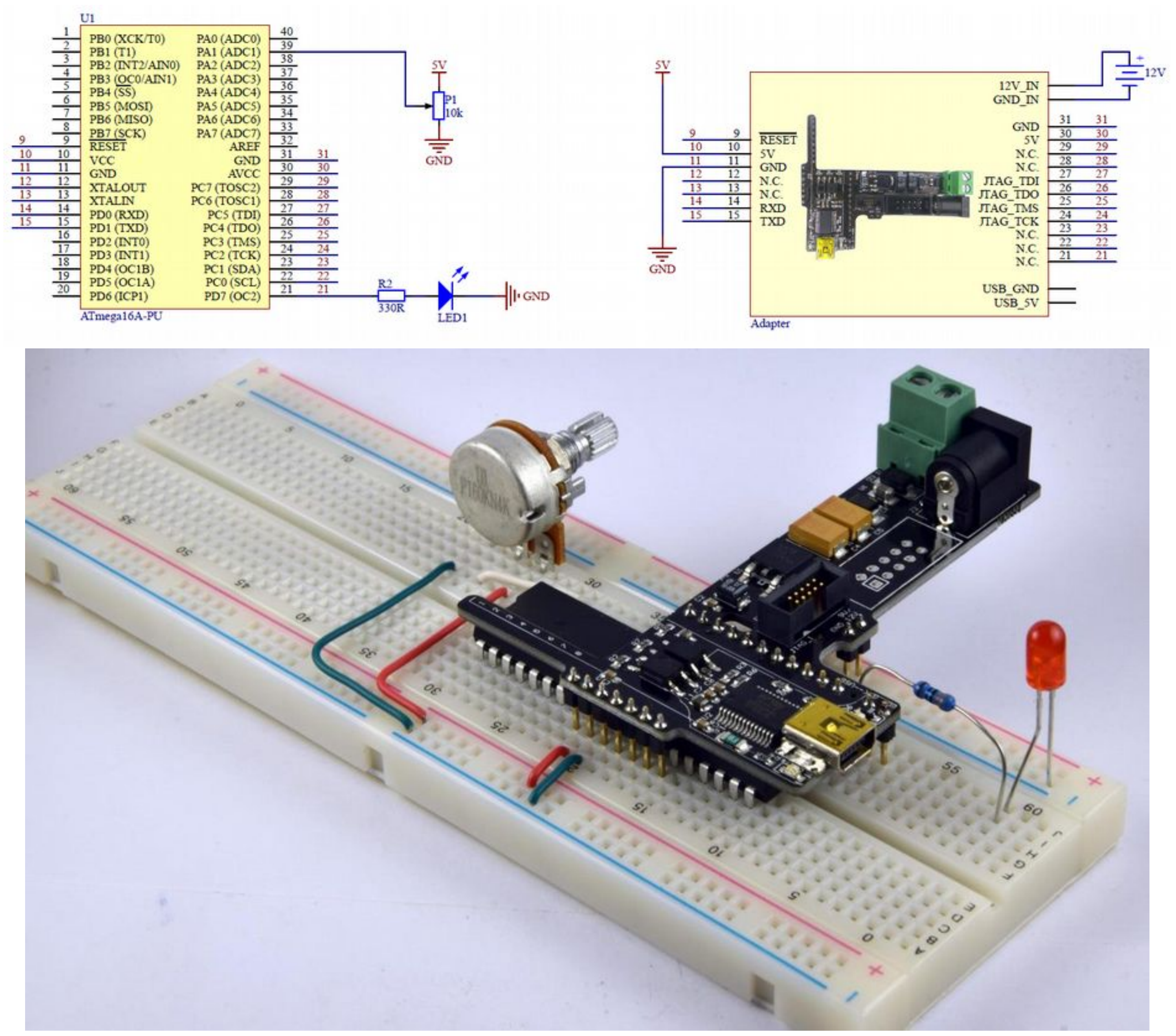


Figure 8 – Setup for measurement 2

Task:
The lower bit of the AD value of the potentiometer on the PA1 pin (ADC1) is written into the PWM register of TMR2, i.e. the potentiometer is used to adjust the duty cycle of the PWM signal appearing on PD7 pin.

Initializing the IO Ports

```
void IOInit()
{
    //Turning off PORTA pull-up resistors on the inputs
    PORTA = 0x00;
    DDRA = 0x00;
    //The LED is on PA0, so PORTA0 should be configured as output (0x01)
    sbi(DDRA, 0);
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```
    //Turning off PORTB pull-up resistors and all pins should be configured as inputs
    PORTB = 0x00;
    DDRB = 0x00;

    //Turning off PORTC pull-up resistors and all pins should be configured as inputs
    PORTC = 0x00;
    DDRC = 0x00;

    //Turning off PORTD pull-up resistors and all pins should be configured as inputs
    PORTD = 0x00;
    DDRD = 0x00;
    //The PD7 pin of PORTD should be configured as output, as Timer2 PWM output is on this pin
    sbi(DDRD, 7);
}
```

First we initialize the input and output pins, see function `IOInit()` in “io.c” file.

On `PORTA`, `PORTB`, `PORTC`, `PORTD`, we turn off the pull-up resistors and configure these pins as inputs, except the PA0 pin on `PORTA`, because this controls the status LED, and PD7 pin on `PORTD`, which has a special function, according to chapter 12.3.4 of the datasheet, the PWM output of Timer2 is on this pin.

Initializing the clock

```
void TimerInit()
{
    //1x prescaler
    cbi(TCCR1B, CS12);
    cbi(TCCR1B, CS11);
    sbi(TCCR1B, CS10);

    //Timer1 interrupt on
    sbi(TIMSK, TOIE1);
    //Global interrupt on
    sei();
}
```

The clock is initialized in “timer.c” using the `TimerInit()` function. The bits of the `TCCR1B` register are described in chapter 16.11.2 of the datasheet.

Table 16-6 shows that if no prescaler is needed, then bits `CS12`, `CS11`, `CS10` must be set 0,1,1 in this order. We do not set the value of the remaining bits, we do not need them, their value remains the default 0.

As described in chapter 16.11.7, enable the interrupt of Timer1 by setting the `TOIE1` bit and finally enable the interrupt (the global interrupt).

Setting up Pulse Width Modulation

```
void PWMInit()
{
    //TIMER2 (PWM)
    //1x prescaler (Page 126)
    cbi(TCCR2, CS22);
    cbi(TCCR2, CS21);
    sbi(TCCR2, CS20);
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

```
    //TIMER2 (PWM)
    //1024x prescaler (Page 126)
    //sbi(TCCR2, CS22);
    //sbi(TCCR2, CS21);
    //sbi(TCCR2, CS20);

    //Phase correct PWM mode (Page 125)
    cbi(TCCR2, WGM21);
    sbi(TCCR2, WGM20);

    //Non-inverting mode (Page 126)
    sbi(TCCR2, COM21);
    cbi(TCCR2, COM20);
}
```

Pulse width modulation is set by the `PwmInit()` function of the "pwm.c" file by setting the `TCCR2` register.

For Timer2 we use no clock division the same way we did for Timer 1 (prescaler 1x, see chapter 17.11.1 of the datasheet). According to the datasheet table 17.2 when selecting operating mode 1 the output waveform will be phase correct (`WGM21 = 1`,`WGM20 = 0`). The behaviour of`COM20` and `COM21` pins depends of the selected operating mode In mode 1 the table 17.5 is valid, which in non-inverting operation setting `COM21` and `COM20` pins to 1 and 0.

ADC Initialization

```
void ADCInit()
{
    //AVCC reference
    cbi(ADMUX, REFS1);
    sbi(ADMUX, REFS0);

    //ADC1 input
    sbi(ADMUX, MUX0);

    //Result is left-aligned -> upper 8 bit in ADCRH
    sbi(ADMUX, ADLAR);
}
```

To initialize the ADC, we must first set the ADC reference voltage in the `ADCInit()` function using the ADMUX register according to chapter 22.9.1, which in our case equals with the external AVCC voltage (`REFS1`, `REFS0` = 0, 1).

Using Table 22.4, we select the register bits to set so that the first channel (ADC1) is the input. For this, `MUX0` must be set to 1, the remainingMUX bits can remain at the default value of 0, and they do not need to be set.

If the ADLAR bit is 1, the result is left-aligned, in ADCRH the result is displayed on the top 8 bits. See measurement example 1 for an explanation.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

ADC enable

```
void ADCEnable()
{
    //128x prescaler
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS0);

    //Interrupt enable
    sbi(ADCSRA, ADIE);

    //AD converter enable
    sbi(ADCSRA, ADEN);
}
```

After initializing the ADC, you can enable the ADC with the `ADCEnable()` function.

In the ADCSRA ADC Controller register, we can set the clock rate of the ADC relative to the clock. We set this prescaler to 128 based on chapter 22.9.2 of the datasheet. Enable interrupt and the operation of the ADC.

Additional code snippets for the main function

```
    OCR2 = 0;

    //Infinite loop
    while (1)
    {
        //PWM setting happens by interrupt
    }
```

In the beginning, the duty cycle of the PWM is 0, that is, the LED connected to PD7 should not be lit. This is achieved by setting the Output Compare 8-bit register (`OCR2` register) to 0 (see chapter 17.7.4 of the datasheet).

We start an infinite loop because the logic is implemented in the interrupt handlers.

```
ISR(TIMER1_OVF_vect)
{
    //The counter is increased, if it hasn’t reached the maximum yet
    if (cntr < CNTR_MAX)
    {
        //The counter is increased
        cntr++;
    }

    //If it has reached it
    else
    {

        cntr = 0;
       //AD conversion is started (sampling)
        ADCStart();
    }
}
```

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

The interrupt function does nothing else but starts a counter that counts to 10, if this is done, we clear (set to 0) the counter and start the AD conversion with the `ADCStart()` function.

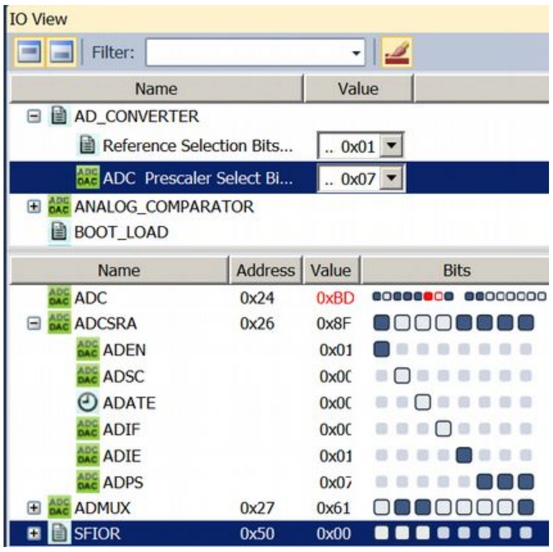
```
void ADCStart()
{
    sbi(ADCSRA, ADSC);
}
```

This function writes the ADSC bit of the ADCSRA control register to 1, which starts the conversion.

```
ISR(ADC_vect)
{
    //Load the upper 8 bits of the AD result into the PWM timer register
    OCR2 = ADCH;
}
```

When the conversion is completed, another interrupt function, namely `ISR(ADC_vect)`, is called, which copies the top 8 bits of the conversion result into the PWM Timer register, which already directly modifies the duty cycle of the output signal.

In IO View, observe the operation of theADCSRA and ADC registers within the AD_Converter/ADC Prescaler Select, with a breakpoint placed next to the commands of `ADCEnable()` and `ADCStart()` functions.


IO View

Filter:

Name	Value
AD_CONVERTER	
Reference Selection Bits...	.. 0x01
ADC Prescaler Select Bi...	.. 0x07
ANALOG_COMPARATOR	
BOOT_LOAD	

Name	Address	Value	Bits
ADC	0x24	0xBD	
ADCSRA	0x26	0x8F	
ADEN		0x01	
ADSC		0x0(	
ADATE		0x0(	
ADIF		0x0(	
ADIE		0x01	
ADPS		0x07	
ADMUX	0x27	0x61	
SFIOR	0x50	0x00	

Figure 9 – ADC registers in IO View

Put a breakpoint in front of the line, where`OCR2` is set in the main function and in the interrupt function. In IO View, observe the operation of the `OCR2` register within the `TIMER_COUNTER_2/Clock select` bits, and also look at our output LED. Turn the potentiometer and observe the change again.

When running the program we can observe with the naked eye, that with a higher duty cycle (the potentiometer is turned up), the LED is brighter.

Practice

Modify the task by changing the Timer2 prescaler to 1024. Notice how much the output changes. At this time, the LED will blink if the duty cycle is reduced.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

For deeper understanding this field, we briefly review the main types of analog-to-digital converters:

Flash ADC

In case of flash ADC, the input voltage is converted directly into a number proportional to the voltage in one step. It consists of parallel comparators, each comparing the input voltage with a reference voltage obtained from a voltage divider, which is built from different resistors connected in series. Then an encoder gives number of the highest comparator having a high output in binary. Although the advantage of the flash ADC is its speed, its huge need for the comparators makes the circuit complex, its power requirement is high, its resolution is low compared to its complexity and it is very expensive. Mostly used in low-resolution, high-frequency circuits.

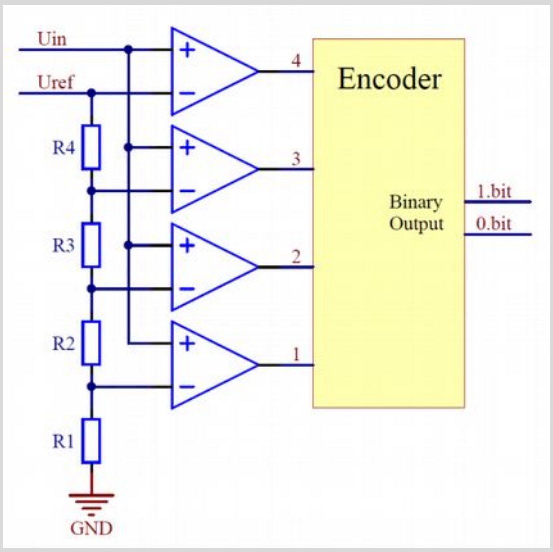


Figure 10 – Internal structure of the Flash ADC

Sigma-delta converter

Explaining how it works is beyond this curriculum, so we only sum up its main features. It achieves a high bit number by sampling much at a much higher rate than the clock signal. Due to oversampling, it is slow but in exchange, it has high resolution and does not require external precision components. Usually used in audio frequency applications.

Dual slope converter

The input signal charges a capacitor for a definite period of time. By integrating over time, noise is averaged. When the capacitor is discharged to a certain level, a counter counts the output bits. The higher the discharge time, the higher the counter value.

Compared to previous types, its noise immunity is higher, it is accurate, but slow, and external precision components (resistors, capacitors) are required for accuracy.

Nowadays, when integrated ADCs in microcontrollers are available, its role has diminished, but it is still worth considering when measuring noisy signals.

Erasmus+

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

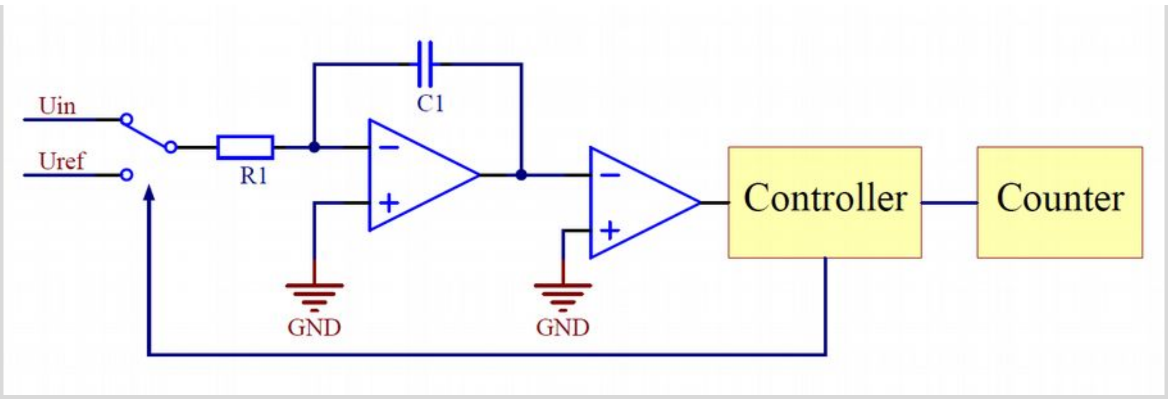


Figure 11 – Internal structure of the dual slope ADC

Successive approximation ADC

For us, this is the most interesting type because a successive approximation type ADC is built into our microcontroller. It's not a coincidence, because its structure is simple, and because of its accuracy and speed, it is one of the most common converters.

Here, the result is obtained in as many steps as the number of bits the ADC has (this requires that the input is not changed during the measurement, otherwise a false result will be obtained. To avoid this, we have to place a sample and hold circuit in front of the converter). We always specify only one place-value at a time, starting with the highest place-value. The comparator checks whether the input voltage is greater than the reference voltage for the highest place-value. This reference voltage is generated by a built-in DAC for this purpose (the operation of the DAC is reversed, compared to the ADC, so it assigns a voltage to a digital number). If the input voltage is less than the voltage at the output of the DAC, then the next bit of the SAR (Successive Approximation Register) is 0, otherwise 1. The rest is compared to the following place-value, and so on. We need as many reference voltages as the bit number.

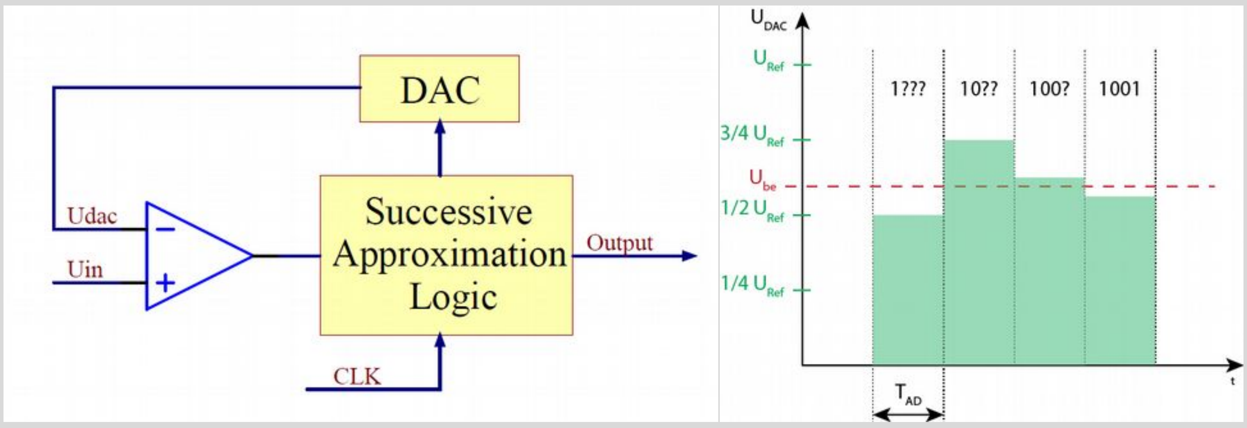


Figure 12 – Internal structure of the successive approximation ADC

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).

Independent measurement task

Software implementation of comparator with hysteresis

The comparator with hysteresis can be implemented by hardware, assembling physical components (see the Comparator chapter) and using ADC, from software code as well. We will implement the latte solution.

Add a potentiometer which changes its resistance according to position to our microcontroller circuit.

Modify the code so that an LED lights up if the ADC reading is above 600, and turns off below that, and set 3 V on the potentiometer.

Note that if we set the potentiometer very accurately close to where the ADC reading is 600, then the LED for the same input will sometimes light up and sometimes not. This phenomenon is possible because both the signal and power can be noisy, and comparisons (evaluating of the comparisons of the two input values) can produce different output even in case of very little difference.

This range, where the signal changes, can be extended using comparator with hysteresis, if we do not want that a small change in the input to appear immediately at the output, and this noise bouncing the output. Let's try out changing the comparison condition so that

```
if(ADC>600) output=1;
if(ADC<400) output=0;
```

then the output will change at about 3 V and 2 V, which will be exactly 1 V hysteresis.

SUMMARY

In this chapter of the curriculum, we have learned about how to convert analog signals into digital numbers and use them in our program. Because the world around us is analog, and has continuous signals, this has a big role in computing, where we can only operate on digital signals. In the later chapters we will build on the knowledge gained here.

This project was supported by the European Commission. The content of this publication does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the author(s).