Article citation info:
Baranski R, Galewski MS, Nitkiewicz S. The study of Arduino Uno feasibility for DAQ purposes. Diagnostyka. 2019;20(2):33-48
https://doi.org/10.29354/diag/109174

DIAGNOSTYKA, 2019, Vol. 20, No. 2

ISSN 1641-6414
e-ISSN 2449-5220
DOI: 10.29354/diag/109174

THE STUDY OF ARDUINO UNO FEASIBILITY FOR DAQ PURPOSES

Robert BARAŃSKI, Marek A. GALEWSKI, Szymon NITKIEWICZ

AGH University of Science and Technology, Faculty of Mechanical Engineering and Robotics,
al. Mickiewicza 30, 30-059 Kraków, Poland, e-mail: robertb@agh.edu.pl
Gdansk University of Technology, Faculty of Mechanical Engineering, str. G. Narutowicza 11/12,
80-233 Gdansk, Poland, e-mail: margalew@pg.edu.pl
Warmia and Mazury University, Faculty of Technical Sciences, 11 Oczapowskiego St., 10-710 Olsztyn, Poland,
School of Medicine, Collegium Medicum, 30 Warszawska St., 10-082 Olsztyn, Poland,
szymon.nitkiewicz@uwm.edu.pl

Abstract

Using microcontroller systems becomes a routine in various measurement and control tasks. Their wide availability together with a huge potential of extending their functionality by additional modules allows developing advanced measuring and monitoring systems by non-specialists. However, using popular example codes often leads the user to pass over or not to be aware of the limitations of the system and drawing too far-reaching conclusions on the basis of incorrectly performed measurements

This paper deals with the problem of choosing the right method for performing measurements using an acquisition system based on the budget Arduino UNO solution. The main assumption was to use the standard, widely available Arduino libraries. The work focuses on the scenario when data should be subject to time and frequency analysis in the later processing. The operating limits of the device were also determined depending on the data transmission method used.

Keywords: microcontroller, Arduino, ATmega, monitoring, data acquisition, signal analysis

STUDIUM PRZYDATNOŚCI PLATFORMY ARDUINO DLA CELÓW AKWIZYCJI DANYCH

Streszczenie

Pomiary wielkości fizycznych z wykorzystaniem układów opartych na mikrokontrolerach stają się standardem. Ich szeroka dostępność wraz z modułami rozszerzającymi ich funkcjonalność daje możliwość budowy zaawansowanych układów pomiarowych i monitorujących przez osoby nie będące specjalistami.

Szereg dostępnych przykładów umożliwia szybką budowę systemu pomiarowego. Niejednokrotnie jednak powoduje, iż użytkownik jak i konstruktor nie zdają sobie sprawy z ograniczeń układu i na podstawie pomiarów wyciągają zbyt daleko idące wnioski.

Niniejsza praca dotyka problematyki właściwej metody pozyskiwania danych pomiarowych. Na przykładach popularnie wykorzystywanych podejść do akwizycji danych, zobrazowano nie widoczne w pierwszym momencie skutki. W pracy skoncentrowano się na sytuacji, gdy w późniejszej obróbce dane mają podlegać analizom czasowym lub częstotliwościowym.

Całość poparto przykładami bazując na układzie Arduino UNO. Założeniem autorów było wykorzystanie standardowo dostępnych bibliotek.

Słowa kluczowe: mikrokontroler, Arduino, ATmega, monitoring, pozyskiwanie danych, analiza sygnału

1. INTRODUCTION

Measurement systems are becoming more and more available and widely used. Undoubtedly, the main merit is a significant drop of electronic elements prices, enabling the construction of inexpensive measuring systems [1, 2, 3]. Thanks to the available modules, it is possible to build monitoring systems or control devices of any type [4]. Recently, Arduino is one of the most popular platforms for achieving this type of goals. Its possibilities can be demonstrated by the use in a number of works, e.g. [1, 4, 5, 6]. Many modules dedicated to this platform are available along with examples of source codes enabling their implementation in various applications. However, people using them are not always aware of the limitations and consequences of its use.

In case of data acquisition systems (DAQ) one of the main elements of the system are measuring sensors and analog to digital converters (ADC). Almost all microcontroller families (μC) (AVR, ARM, MCS51 and others) are equipped with ADC converters. Current literature, both scientific and popular science, provides a number of examples of their use. However, they focus on the general draft of hardware capabilities along with code solutions that are easy to implement but omit analysis of the

effects of the application of individual solutions, assuming that they will not have a significant impact on the measurement result. Often some key issues related to e.g. signal processing theory are omitted, assuming that this subject is beyond the scope of the simple example.

In this study, the authors try to answer the question: what errors one can meet when applying specific code and how inaccuracies change with the change of the basic parameters related to signal acquisition? With this information the reader will be able to choose the appropriate method of data acquisition from the point of view of the desired application.

Arduino UNO (UNO) based on the ATmega 328P microcontroller was used as the test system [7]. Its technical parameters in comparison with other Arduino systems based on microcontrollers are included in [8].

It is important to notice that the intention of the authors was to use only the standard libraries available for Arduino. This assumption was connected with the utilitarian character of the presented results because it was about checking the possibilities of tested systems for a wide range of recipients without limiting it to specialists developing their own libraries.

The methods presented below were intended to illustrate the effects of certain settings that configure the device to work for the conscious use. This is important because inexperienced users often rely on solutions found on the Internet without being aware of the significant limitations or drawbacks that these solutions have and what are the consequences of using them. Due to the increasing use of measurement systems based on the Arduino platform in scientific works [1, 4, 5, 6], it is necessary to determine and verify the measurement limits of this system. This goal was also adopted by the authors of this article.

2. DATA ACQUISITION TITLE

Data acquisition is one of the essential elements in monitoring and measurement systems. In systems using microcontrollers only the data transformed into digital values become useful and are the basis for the central unit to make a decision. That is why the credibility of the acquired data is so important, which is influenced, among others, by correct operation of the ADC converter and its proper use. We usually have no influence on the ADC converter operating principles and most of its parameters, as it is an integral part of the microcontroller used. Experience shows, however, that it is not trivial to use it correctly and consciously.

Microcontrollers from the AVR family are single-thread devices that can only perform sequential operations. Therefore, any occupation of hardware resources for the purpose of, for example, data analysis, makes it impossible to perform another activity, such as measurement. That is why knowledge of the time of performing individual operations is critical. It is even more important as these systems do not have a DMA system (Direct Memory Access mechanism), which in more advanced processors (e.g. ARM based) greatly facilitates the automation of data transfer between the ADC and the operating memory [9]. Thus, while creating a system, it should be ensured that the measurement, transfer of results and their analysis do not block or delay each other. It should be emphasized that in most cases, in addition to measuring the physical quantity, time is also measured - directly (e.g. by explicitly recording information about the time instants when consecutive measurements was made) or indirectly (e.g. by measuring with a constant, known frequency). What's more, the correct measurement of time intervals between successive samples is crucial for proper, further analysis in time or frequency domain.

There is a need to select the appropriate sampling rate, which is strongly dependent on the phenomenon being studied and the needs related to further signal analysis, where it is also necessary to meet the Nyquist condition and other laws regarding signal processing [10].

From the point of view of a useful frequency band available using Arduino UNO, we can count on sampling of several teens kHz in case of cyclic use of data inside μC. In case of simultaneous communication with the PC using a serial port (which will be presented later in the paper), frequencies of several kHz can be obtained. An interesting feature is also the use of an external library that allows recording signals on an SD card with a sampling rate close to 40 kHz [11] (however, it is not the subject of this work).

The usefulness of the acquisition system or direct analysis of measurement data depends to a large extent on the measurement objective. Other sampling frequencies are necessary for the analysis of the GSM or technical diagnostic (MHz band) [12, 13] other for acoustics (tens of kHz) [14, 15, 16, 17] and yet another EMG signal and vibration signal analysis (several hundred Hz) [18, 19, 20].

A separate issue is the accuracy of reproducing the amplitude of the signal. It is a parameter that depends directly on the range and bit resolution of the transducer used and the amplitude of the measured signal in relation to the measuring range of the transducer. There are software methods that allow to increase the resolution of the converter, but this happens at the expense of the sampling frequency, and their correct operation depends on the fulfillment of a number of conditions. Therefore, they cannot be used in every situation [21].

If the microcontroller is used as a DAQ card, it is extremely important, apart from the above-mentioned elements, to access the data. Therefore,

one of the goals of the work was to check how does it change in the most common cases of use:

- continuous access to data by connecting the system to a PC,
- recording data on a medium (e.g. an SD card).

The time of a single measurement ($t_{one_measure}$) and, in effect, the signal sampling frequency will be affected by the three basic factors shown in Fig. 1. These are: measuring time (AD converter operation) (t_{ADC}), signal processing / computation time (t_{comp}), transmission time (sending to PC or memory card) (t_{send}).

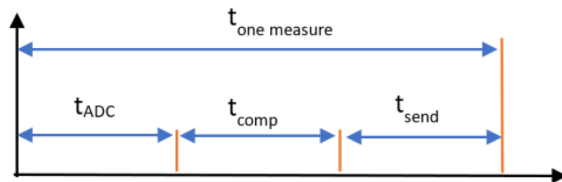

Fig. 1. Components of acquisition time

They influence the maximum DAQ system sampling rate. Transducer operating time necessary for data readout (t_{ADC}) is imposed by hardware and can only be modified by changing the AD converter clock. We can try to minimize t_{comp} time, assuming that raw data is sufficient for us, because further calculations will be made on the PC side. The remaining part is data sending time. It depends on the protocol configuration and the method of data transfer. At the same time, for each of these times the frequency of the microcontroller main clock is affected because it determines the overall speed of the processor.

From the point of view of signal processing, in addition to a single sample acquisition time, a very important parameter is the frequency of measurement of further samples. This frequency cannot be higher than it would result from the time of measuring one sample. In practice, the problem is not so much about achieving the maximum frequency but implementing of a desired, precisely defined sampling frequency. In order to ensure it, the most common approach is the control of measurement triggering with the use of timers peripheral subsystems and interrupt mechanism. The frequency of interrupt calls depends on the configuration of the timers, including the configuration of its clock signal. That's why sections 4 and 5 of this paper concentrate on a more detailed description of AD converter and timer configuration and data transmission systems as well as their impact on the sampling frequencies.

3. METHODOLOGY OF THE STUDIES

3.1. Time measurement

One of the basic parameter of the analysis was time measurement. The standard Arduino environment is equipped with two functions: *millis()* and *micros()* enabling the measurement of time during operation of μC. Both functions are based on the use of internal timers and Arduino interrupts [22].

In the case of *millis(),* the counted time changes every 1024 μs (hardware timer interrupt request call). So the indicated 1 ms (1000 μs) is in fact 1024 μs. The *millis()* function counts the number of timer overflows on the internal variable and is adding value of 3 to it with each overflow. When this variable exceeds 125, which happens every 41 to 42 ms, the counted time error is corrected by software by adding 1 ms to the counted time value and value of 125 is subtracted from the overflow counter. Due to this, an additional leap of the counted time value takes place. Time measurement error is therefore not constantly accumulated but it never exceeds 1 ms neither. However, one should be aware of its occurrence. In addition, the result of this error and its cyclical build-up and correction is that in order to achieve a stable and accurate measurement of time one should not use the millis() function but consider other methods.

In the case of the *micros()* function, much greater precision is obtained, however its real resolution is 4 μs [23]. For the prescaler (i.e. the frequency divider) implemented in the function equal to 64 (the clock value of μC equal 16 MHz) it gives the resolution value 64/16 MHz = 0.000004 s = 4 μs [21, 23, 24]. The micros() function counts the number of counter overflows multiplied by 1024 (the timer overflows every 256 cycles of its clock signal) and adds the current timer status. The final result of the time calculation is multiplied by 4 and is expressed in μs. It should also be added that both functions have limitations in the amount of time that they are able to count without a variable overflow. The 32-bit unsigned long variable is recommended for this purpose. For the *millis()* function, it can store up to 49.71 days, and for *micros()* up to 71.58 minutes.

3.2. Test system

The LabVIEW 2016 environment was used to carry out the tests. USART system with RS232 protocol was used to communicate UNO with the PC. Fig. 2. shows a block diagram of the data collection and analysis software utilized.

The system uses a structure based on two loops. VISA Read loop had the highest priority (with Timed Loop LabVIEW mechanism). Then, the received data was transferred by queue to the *acquiring* loop, converted there and then collected in a DBL type table (64-bit double precision float). After collecting a certain number of data (or the time elapsed), the data reception was interrupted and the data was analyzed (*offline analyze* block in Fig. 2.).

This solution give the opportunity to ensure the continuity of data collection and eliminating the impact of the analysis on the process of receiving data from the serial port.

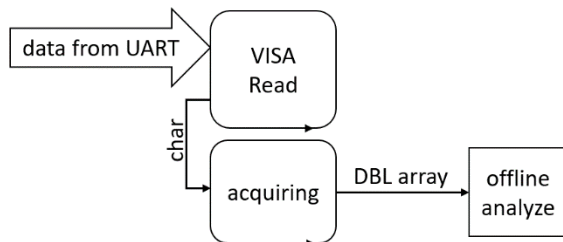

Fig. 2. A block diagram of data acquisition on the PC side (LabVIEW)

The comparative method was used to test the operation of the ADC transducer. The results of two measurements (UNO and NI USB-6212) were compared with each other. For this purpose, a higher-grade measurement card dedicated for cooperation with the LabVIEW environment was used as the reference point. The NI USB-6212 card has the following input / output parameters:

- Analog input (AI): 16 bit, 400 kHz, max range ±10V,
- Analog Output (AO): 16 bit, 250 kHz, range ±10V.

The results of time measurements presented in the further part of the article are based on the measurement of time performed on the μC using *micros()* function. The authors also performed time measurements on the PC side (counting the amount of data received in a time interval). The results of both methods were convergent, but there were some differences. They may result from transmission method, the time necessary to execute LabVIEW commands or the uncontrolled load of the PC processor by the Windows operating system.

For this reason, it was decided to use only more reliable timing by the μC limiting the impact of the above issues.

4. ADC TRANSDUCERS

This section contains information related to the correct ADC configuration. The results of measurements were presented to check the accuracy of the mapping of the measured signal and to measure the time necessary to carry out the measurement.

4.1. ADC prescaler

Arduino UNO is based on the ATMega 328P microcontroller, which has a clock frequency of 16 MHz. The ADC transducer is also clocked at this frequency. This is a very high frequency for a typical use of the μC converter that consumes significant amounts of energy. Therefore, it is possible to lower this frequency. In order to change the μC timing, set the appropriate ADPS2 ADPS1 ADPS0 bit (ADC Prescaler Select) values in the ADCSRA register (ADC Control and Status Register) [25, 26]. Table 1 presents the values of the prescaler, the set of bits of the register and the corresponding μC frequencies.

Table 1 Prescaler settings

Prescaler	ADPS2 ADPS1 ADPS0	Clock freq (MHz)
2	0 0 1	8
4	0 1 0	4
8	0 1 1	2
16	1 0 0	1
32	1 0 1	0.5
64	1 1 0	0.25
128	1 1 1	0.125

The timing frequency is calculated according to:

$$t_{freq} = \frac{t_{freq\mu C}}{prescaler} \quad (1)$$

For ATmega 328P with a clock frequency of 16 MHz, the choice of a prescaler value equal 16 will reduce the clock frequency to 1 MHz. The code appropriate for the above set-up is presented in Code 1.

Code 1

```
void setup() {
  // remove bits set by Arduino library
  ADCSRA &= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
  // set prescaler to 16
  ADCSRA |= (1 << ADPS2);
}
```

In Arduino Software IDE, the default value of the prescaler is 128. This means that if the programmer does not change the prescaler explicitly, the microcontroller ADC will be clocked at 125 kHz. In addition, as the clock speed increases, the accuracy of the ADC decreases. This issue will be discussed in paragraph 4.2.1.

4.2. Selection of timer interrupt trigger frequency

Interrupt is a hardware functionality that allows to perform a given action (program code in the form of an interrupt service function) at any time during execution of the main program code. Generally, interrupt is called by hardware - externally (by changing the state of the input line dedicated to reporting interruptions by an external device such as a sensor, button or other microcontroller) or internally (via a microcontroller peripheral sub subsystem such as a timer, ADC converter or USART system). Upon detection of the interrupt request the processor interrupts the currently executed program and proceeds to the interrupt handling function ISR (Interrupt Service Routine). This operation takes at least 4 processor cycles [26]. Return to the interrupted program after execution of an ISR also takes at least 4 cycles.

To obtain constant intervals between successive moments of signal sampling (constant sampling frequency of the signal) it is best to use one of the

microcontroller sub-systems dedicated to these type of tasks, such as timers. The ATmega328 microcontroller is equipped with 3 timers called *Timer0*, *Timer1*, *Timer2* [11]. *Timer0* and *Timer2* are 8-bit timers (they adopt values in the range of 0 ÷ 255). *Timer1* is a 16-bit timer (it adopts values in the range of 0 ÷ 65535).

In order to obtain the desired frequency of timer interrupt it is necessary to set several parameters, which values are defined in the configuration registers of the µC subsystems. For *Timer1* suitable bits in the registers are intended for this:

- TCCR1B – Timer/Counter Control Register,
  - Bits CS10, CS11, CS12 (Clock Source)– defining the timer clock signal source ,
- TCNT1 – Time Counter Register – register enabling the writing and reading of 16-bit values from the main timer,
- TIMSK1 – Timer Interrupt Mask Register – a register where the sources of interrupts reported by the timer are signaled,
- OCR1A, OCR1B – Output Compare Registers – registers containing reference values used by the timer comparator, equating the timer value with the value OCR1A or OCR1B value can trigger timer interrupt request.

Directly from the start, the timer counts impulses with the speed depending on the frequency of the µC clock. *Timer2* (16 bit) can count up to 65536 values. For µC with the clock speed set to 16 MHz, the timer will reach the maximum value in time ≅0.0041 s (16 MHz / 65536), i.e. the overflow will occur at the frequency of ~244.14 Hz

Since it is usually expected to sample the signal with a different, determined frequency, it is necessary to adjust the frequency of timer interrupt requests. For this purpose, the Timer / Counter Control Register (TCCRxB) can be used, where x is the timer number [27]. In this way the timer's prescaler can be set. Prescaler setting depends on the values of CS12, CS11, CS10 bits. However, accepted values can only be chosen in a discrete set presented below [26, 27].

Table 2 Clock Select Bit Description [26, 27]

CS12	CS11	CS10	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clkI/O/1 (No prescaling)
0	1	0	clkI/O/8 (From prescaler)
0	1	1	clkI/O/64 (From prescaler)
1	0	0	clkI/O/256 (From prescaler)
1	0	1	clkI/O/1024 (From prescaler)
1	1	0	External clock source on T1 pin. Clock on falling edge.
1	1	1	External clock source on T1 pin. Clock on rising edge.

For example, if prescaler setting 1024 is needed, the operation below must be performed:
TCCR1B |= (1 << CS12) | (0 << CS11) | (0 << CS10)

Thanks to this the timer overflows and interrupt requests occur after ≅ 4,194 s (16 MHz / 1024 / 65536), that is with frequency ~0.238 Hz.

From the above relationships, a formula for the selection of OCRxA values can be developed to obtain the desired interrupt triggering frequency.

$$\frac{\mu CCS}{TCCRxB * f_{IRQ}} - 1 = OCRxA \tag{2}$$

Where:
f_{IRQ} – desired interrupt request frequency (Hz),
$TCCRxB$ – the value of the prescaler resulting from the bit settings CS12,11,10 in Timer/Counter Control Register B,
μCCS – microcontroller clock speed (16 MHz for UNO).

This solution has a limitation. It is not possible to obtain any f_{IRQ} values, as the ORCxA value must be a positive integer from thr range 0 ÷ 65355 for *Timer2* and 0 ÷ 255 for *Timer1*. For example, to obtain the frequency of 500 Hz, the settings µCCS = 16 MHz, TCCR1B = 256, OCR1A = 124 should be used (see Code 2.)

Code 2. Microcontroller's timers setting

```
int data;
void setup(){
  cli();       // all interrupts stop
  TCCR1A = 0;  // set entire TCCR1A
register to 0
  TCCR1B = 0;  // set entire TCCR1B
register to 0
  TCNT1  = 0;  //initialize counter
value to 0
  OCR1A = 124; // computed value
  TCCR1B |= (1 << WGM12); // turn on
CTC mode

  // Set CS10 and CS11 and CS12 bits
for 1024 prescaler
  TCCR1B |= (1 << CS12) | (0 << CS11)
| (0 << CS10);
  TIMSK1 |= (1 << OCIE1A);// enable
timer compare interrupt
  sei();       // all interrupts start
}
void loop() {
}

ISR(TIMER1_COMPA_vect)  {
//interrupt handler function to run
when
//Counter1 Compare Match A
  data = analogRead(A1);
}
```

The presented code also contains an exemplary ISR function (TIMER1_COMPA_vect), which is

executed when an interrupt is called. The names of functions performed by ISR are strictly defined in the Arduino environment.

Table 3 List of Timers interrupts [28]

Name	Syntax
Timer/Counter2 Compare Match A	TIMER2_COMPA_vect
Timer/Counter2 Compare Match B	TIMER2_COMPB_vect
Timer/Counter2 Overflow	TIMER2_OVF_vect
Timer/Counter1 Capture Event	TIMER1_CAPT_vect
Timer/Counter1 Compare Match A	TIMER1_COMPA_vect
Timer/Counter1 Compare Match B	TIMER1_COMPB_vect
Timer/Counter1 Overflow	TIMER1_OVF_vect
Timer/Counter0 Compare Match A	TIMER0_COMPA_vect
Timer/Counter0 Compare Match B	TIMER0_COMPB_vect
Timer/Counter0 Overflow	TIMER0_OVF_vect

The presented calculations and an example concern the use of 8 bit *Timer1*.
When using timers, it is important to keep in mind that some standard functions and libraries also use them, so the programmer should be careful and assure that conflicts do not occur. For example, *Timer0* is utilized by functions like *millis()* and *micros()*, *Timer1* is used by the *Servo* library functions and *Timer2* – by the *tone()* function.

In case when there is more than one timer used in the code (including situation when, for example, the timers are used by functions from other libraries), *Timer2* interrupts have a higher priority than *Timer0* and *Timer1* interrupts [26, 29].

4.2.1. Influence of ADC clocking on measured values

In signal processing, the correctness of the measured value mapping is one of the key elements. Erroneous amplitude measurements lead to erroneous analize, which ultimately causes erroneous conclusions.

At first the linearity in the entire measurement range of the converter was checked. The method relies on a measurement of a preset, constant voltage values repeated 1000 times. Measured values are averaged then. Preset voltages were generated using 16-bit NI USB-6212 analog output channel.

What's important, the Arduino UNO measuring system uses 5V as the reference voltage (U_{Ref}) obtained from the voltage stabilizer. Therefore, it is assumed that the resolution of the 10-bit ADC is 5 V / 1023 = 4.887 mV. However, this approach may lead to misinterpretation of results, because the U_{Ref} (5V by default) may not be exactly as assumed. In our case, the measured U_{Ref} was 4.52V (measured with a voltmeter between GND and 5V). Hence the actual resolution of the AC converter was 4.418 mV. In the data sheet [26] and in the paragraph 1.6 of [30] a record regarding the so-called *gain error* is described. Its elimination is based on the measurement of the minimum and maximum values, and then the software correction with the corrective curve determined in this way is advised to be applied.

The above advised activities were performed. The results of the measurements after calibration are presented in Fig. 3. The identified error reaches a maximum of about 8 mV (which is ≈2 ADC quantization levels).

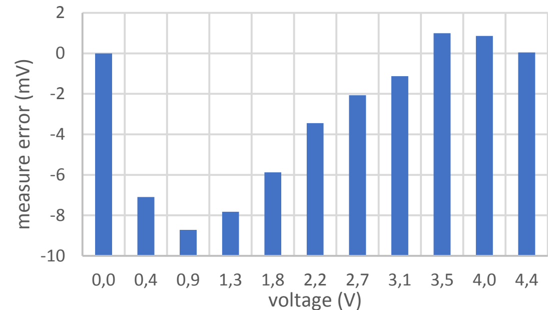

Fig. 3. Measure Error

Next, the measurement of the stability of the measured voltage were performed. Each measurement lasted about 10 seconds (with sampling = 1 kHz). The voltage was generated at the analog output of the NI USB-6212 card and then it was connected to the input of the ADC converter in the UNO and, at the same time, to the analog input of the NI USB-6212 card.

Table 4 Voltage stability

NI USB-6212 analog output value			Arduino UNO analog output value		
Umean (V)	std (μV)	max-min (mV)	Umean (V)	std (μV)	max-min (mV)
0.0522	0.2	3.29	0.0178	2.3	4.89
0.5174	0.2	2.96	0.5009	2.4	4.89
1.1673	0.2	1.81	1.1774	1.5	9.77
2.1876	0.2	1.32	2.2385	0.4	9.77
3.9382	0.2	1.32	4.0544	2.5	9.77

According to the data above, we can observe several elements. The increase in the measured value increases the difference between the NI and UNO card indications. Differences in maximum and minimum values read by UNO are 1-2 levels of quantization. Although the absolute values of these differences compared to the results from the NI USB-6212 card seem to be significant, the stability achieved can still be considered as good considering the 10-bit resolution of the AD converter in UNO.

The abovementioned measurements were performed for the default prescaler (set at 128), which results in ADC clock frequency = 125 kHz, which is the recommended value.

According to the documentation [23], the accuracy of the AD converter decreases with the increase of the μC clock frequency. According to paragraph 1.8 of [23], "*The ADC accuracy also depends on the ADC clock. The recommended maximum ADC clock frequency is limited by the*

internal DAC in the conversion circuitry. For optimum performance, the ADC clock should not exceed 200 kHz. However, frequencies up to 1 MHz do not reduce the ADC resolution significantly.". Thus the timing used is in accordance with the manufacturer's guidelines.

In order to check the effect of the prescaler described above on the reading value, measurements were taken for the setpoint voltage of 0.90523 V and 3.612 V, matched so that their values correspond to the selected integer-valued quantization level of the 10-bit Arduino UNO converter at the U_{Ref} voltage of 4.5217 V. The generated voltage was characterized by a stability of ±0.66 mV. Fig. 4. shows the mean values of 1000 measurements with maximum and minimum values. It can be seen that the average measured value, as expected, oscillates around the constant quantization value for all measurements. This can be interpreted as a good stability of the obtained results when using the averaging of the measured signal.

However, it should be noted that as the prescaler decreases (increasing the clock frequency of the ADC), its accuracy is noticeably reduced, resulting in an increase in the maximum and minimum values. This is especially evident for a prescaler values less than 8, which corresponds to an ADC clock frequency over 1 MHz. It should be noted that according to paragraph 1.8 of the document [30] „*Operating the ADC with frequencies greater than 1 MHz is not characterized*" so we can expect deterioration in ADC characteristics for such frequencies.

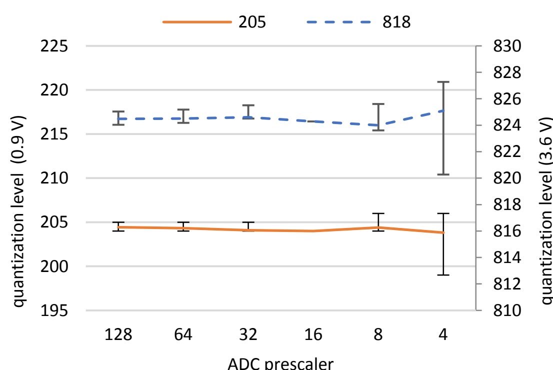

Fig. 4. Stability of quantization depending on the ADC prescaler

The situation was similar in for the measurements of other voltage values.

No results were presented for the prescaler 2, because regardless of the measured voltage, the ADC returned 1023 value or close to it so the correct voltage measurement was not obtained. This behavior of the transducer completely eliminates the use of prescaler 2 for measurements.

4.3. Measurement time and frequency

The subject of the research was to determine the time necessary to read data from the AD converter. Due to the fact that the microcontroller tested has the ability to change the prescaler responsible for clocking the ADC converter, its influence on the measurement time was checked.

In order to eliminate the influence of the time necessary to measure the implementation of the *micros()* function on the obtained results, the measurement of an empty loop performed 1000 times was performed. The average time of a single measurement was 0.47 (± 0.0038) µs. This value was included in further calculations.

4.3.1. ADC prescaler

In order to present the operation of the interrupt mechanism, Arduino UNO working at a default clock rate of 125 kHz, with a prescaler equal to 128 (Code 3) was used. All measurements were performed for voltages from the entire ADC measuring range. Regardless of the voltage value measured, the results were identical.

Code 3

```
unsigned long time1;
int data, i;
void setup(){
   Serial.begin(115200);

}
void loop(){
  time1=micros();
  for (i=0; i<1000;i++){
    data=analogRead(A0);
  }
  time1 = micros()-time1;
  Serial.println(time1);
}
```

According to the documentation, the ADC requires 13 clock cycles to read the value (the exception is the first reading occupying 25 cycles) - see paragraph 28.4 of the documents [26] and [27]. The ADC clock in Arduino UNO works with a maximum frequency of 16 MHz. By default, the prescaler has a value of 128, which reduces the clock speed to 125 kHz (16 MHz / 128). At this clock frequency, the theoretical maximum reading frequency from ADC is 125 kHz / 13 ≈ 9615 Hz.

The Table 5 shows the time to read 1000 of data from the ADC converter (for one channel). Informatively, the frequency fp_{theo} (theoretical frequency period) was calculated on its basis and was compared to the theoretical frequency resulting from the µC clocking. The value of fp_{mea} (frequency period measured) was computed based on *mean* time. There was also Standard Deviation (std), Maximum (max), and Minimum (min) time calculated.

It should be emphasized that from the point of view of usability of the AD converter there is no justification for using a prescaler with setting 2.

Table 5 ADC time – prescaler influence

ADC pre-scaler	*f. ADC (kHz)*	*mean (μs)*	*std (μs)*	*max (μs)*	*min (μs)*	*fp_{mea} (kHz)*	*fp_{theo} (kHz)*
4	4000	5.251	0.00340	5.260	5.240	190.4	307.7
8	2000	8.597	0.00516	8.612	8.584	116.3	153.8
16	1000	15.561	0.00454	15.576	15.556	64.3	76.9
32	500	29.643	0.00357	29.648	29.636	33.7	38.5
64	250	55.542	0.00346	55.544	55.536	18.0	19.2
128	125	111.544	0.00416	111.56	111.54	9.0	9.6

Despite the influence of the loop on time measurement the difference between the measured time and the theoretical one is noticeable. It results, among others, from the fact that theoretical *fp* only takes into account the processing time (13 cycles of the transducer), but not taking into account the operations performed by μC such as reading the measured value from ADC registers, returning this value by the *analogRead()* function to the variable 'data' or the interrupt handling time for *Timer0* interrupt used by the *micros()* function.

According to theoretical assumptions, the increase in the prescaler increases the time needed to perform the *analogRead()* function almost twice. This is observed down to the prescaler value of 8. Below this value the time measurement results are significantly influenced by commands and functions performed outside the read operation of the ADC value (note that we only interfere in the ADC converter prescaler, μC works at a fixed frequency). In addition, the clock rate of ADC with frequencies higher than 1 MHz is not recommended, as previously mentioned.

It is worth noting that the change in the ADC timing does not affect the speed of data transmission. For example, for BR = 115200 bps, the PC data receive frequency was ≈2350 Hz, for all μC prescaler values.

4.3.2. Sampling triggering using the timer and interrupts

The interrupts are a method of triggering the measurement of the signal from the ADC with the highest certainty regarding the uniformity of the measurement time. For this purpose, Code 4 was implemented in UNO. The time of triggering the interrupt by the algorithm implemented in μC was measured for 30 seconds for each case.

In Code 4 *Timer1* has been set to CTC (Clear Timer on Compare Match) mode. It is achieved by setting the value 1 of WGM12 (Waveform Generation mode) bit in the register TCCR1B (Timer / Counter Control Register) and 0 of WGM13 bit in the register TCCR1B as well as WGM11 and WGM10 bits in the register TCCR1A. The counter prescaler is set to the value of 8, which means the clock frequency is set at 2 MHz. Entering the value of 1999 in the OCR1A register will result in the frequency of timer interrupt request reporting equal to 1000 Hz.

Code 4 ADC – interrupts

```
unsigned long time1, time2;
void setup(){
  Serial.begin(115200);
  cli();          // all interrupts stop
  TCCR1A = 0;     // set entire TCCR1A
register to 0
  TCCR1B = 0;     // same for TCCR1B
  TCNT1  = 0;     //initialize counter
value to 0
  OCR1A = 1999; // computed value
  TCCR1B |= (1 << WGM12); // turn on
CTC mode
  TCCR1B |= (0 << CS12) | (1 << CS11)
| (0 << CS10);  // prescaler to 8
  TIMSK1 |= (1 << OCIE1A); // enable
timer compare interrupt
  sei();          // all interrupts
start
}
void loop(){
}
ISR(TIMER1_COMPA_vect)  {
 time_tmp=micros();
 time1=time_tmp-time2;
 time2=time_tmp;
  Serial.println(time1);
}
```

The results of measurements of the time after conversion to the sampling frequency fp_{mea} are presented in Table 6. In addition to the register settings for the *Timer1* counter and the determined values of fp, it also contains the desired frequency value (fp_{des}).

Table 6 Interrupts - time measurement

Timer1 mode	*OCR1A*	*time (μs)*	*std time (μs)*	*fp_{des} (Hz)*	*fp_{mea} (Hz)*
CTC	24999	12500.0	2.9	80	80.0
CTC	19999	10000.0	2.9	100	100.0
CTC	1999	1000.0	0.5	1000	1000.0
CTC	1249	625.0	1.8	1600	1600.0
CTC	999	500.0	2.7	2000	2000.0
CTC	639	320.0	0.0	3125	3125.0
CTC	399	200.0	0.5	5000	5000.0

It can be observed that the obtained results are equal to the theoretical values. It should be noted that the measurements were repeated several times and each time identical results were obtained, confirming the stability of internal µC time control methods like interrupts.

In the further part of the article, the principle of timer configuration so that the sampling frequency of the signal would be an integer value was applied. Such a solution enabled a trouble-free further analysis in any computing environment.
The stability of the interrupt triggering time with the use of an external device was also determined (for several selected interrupt settings). To this end, whenever the interrupt was called, the state of one of several digital output (DO) of the Arduino UNO was reversed and the time between successive changes was measured. Measurements of the DO signal were performed using the NI 6251 measurement card, with signal sampling set to 1.25 MHz for a period of 10 seconds. The Table 7 shows the results obtained for the selected frequencies.

Table 7 Stability of the trigger frequency of the interruption from the counter Timet1 overflow

fp (Hz)	*fp_{comp} (Hz)*	*mean time (ms)*	*std (µs)*
32	31.98	31.27	0.59
80	79.95	12.51	0.47
100	99.94	10.01	0.39
2000	1998.8	0.5003	0.47
3125	3123.2	0.3202	0.42
10000	9994.1	0.1001	0.49
20000	19988.2	0.0500	0.54

On the basis of the presented results it can be stated that the interrupt triggering time was stable. This is evidenced by the repeatability of the standard deviation for individual frequencies. At the same time, there are differences between the set frequency (fp_{comp}) and the calculated frequency (fp). These differences are proportional to the frequency and, for the tested specimen of the UNO kit, reach 0.059375% of fp_{comp}.

In the opinion of the authors, the above may indicate a discrepancy between the clock periods in Arduino UNO and NI 6251 cards. To confirm this observation these differences were examined for three different copies of Arduino sets. Assuming the clock of the National Instruments (NI) card as the reference point, the difference between the average clock frequency for these chips and the 16 MHz value was calculated from the measurements:

- UNO (original): 20 081 Hz below 16 MHz,
- UNO (clone): 9 824 Hz below 16 MHz,
- NANO (clone): 10 605 Hz above 16 MHz.

These tests confirm the difference between clocks for different systems, which may cause some discrepancies when comparing results obtained with the use of several Arduino systems. It should be noted, however, that this is a normal phenomenon resulting primarily from the fact that ceramic oscillators have been used in the above-mentioned Arduino sets. This type of oscillators usually have a frequency tolerance of ±0.5%, so for 16 MHz they can show up to 80 kHz difference from the nominal value. The differences shown in Table 7 will not have a significant impact on the reliability of time measurement or triggering ADC using interrupt, provided that the measurement will be short. However, in the case of measurements lasting longer in time, one can expect a noticeable error in time (or frequency) measurement. For example, after 24 hours in the UNO system under consideration the delay in time measurement would be over 50 s. Therefore, if the accuracy of time measurement is important, it is recommended to replace the µC resonator from ceramic to quartz. Quartz resonators usually have a frequency tolerance 100 ÷ 500 times better than ceramic resonators. It is worth noting that on the Arduino UNO board the resonator used by µC is placed close to the processor chip and has small dimensions, and it is often confused with a larger resonator located near the USB connector, which is used by the USB transmission system and which has a frequency of 16 MHz as well. It should be emphasized that the resonator exchange will positively affect not only the timers accuracy but also the interrupt triggering frequency, as well as the stability and accuracy of all timing and clock signals in the entire µC system, including the ADC timing.

4.4. Total Harmonics Distortion

The THD (Total Harmonics Distortion) parameter is one of the basic indicators used to parameterize the signal quality. It informs about the amount of harmonic components present in the signal in relation to the main signal component. It is used when the excitation signal is a mono-harmonic sinusoidal signal. The occurrence of harmonic components can then be interpreted as the frequency or amplitude disturbances occurrence in the signal (or in the measurement process).

As a reference, a 100 Hz sine signal generated by the NI USB 6212 card (16bit, ±10V, sampling rate 400 kHz) was used. This signal was recorded by the UNO and the NI 6212 card with the same sampling rate of 1600 Hz. Each measurement lasted 10 seconds. The recorded signal was then subjected to THD analysis in the LabView environment. The measurements were carried out for NI and UNO cards connected to a desktop computer and a laptop running on battery power (in order to separate it from an AC power grid). For comparison, identical measurements were made using the NI USB 6212 card. The results are presented in Table 8. Detected sine amplitude is denoted as *amplitude*. Description *no, TXT* means sending text data without using the interrupt mechanism. Similarly, in case of *no, BIN* – interrupts were not used and data was sent in

binary form (this will be further discussed in section 5 of the paper).

Table 8 THD measurements in various operating conditions

Mea system	THD (%)	amplitude (V)	power source	interrupts	fp (Hz)
NI6212	0.015	0.707	battery	yes	1600
NI6212	0.010	0.707	power grid	yes	1600
UNO	0.085	0.792	battery	yes	1600
UNO	0.488	0.833	power grid	yes	1600
UNO	0.704	0.826	power grid	no, TXT	2351.4
UNO	0.875	0.830	power grid	no, BIN	2939.56

THD

Analyzing the presented results, a strong dependence of the THD levels on the power source can be noticed. In the case of battery power supply they are an order of magnitude lower than in the case of power supply from the power grid. For comparison, for the NI6212 card, the THD value achieved similar values regardless of the power supply method.

In the case of control of the measurement time (interrupts) a significantly lower THD was obtained. It demonstrates, among others, the degree of sampling irregularity in the case of using software triggered measurements.

It should be mentioned, that simulation was carried out, where the generated signal (sine wave with 1 V amplitude) was subjected to virtual quantization by an ideal 10-bit converter. Obtained THD value was 0.085%. This value is consistent with the result obtained for measurements using UNO with battery powered laptop. This indicates proper operation of the converter, which does not introduce distortions greater than those resulting from its finite resolution.

Amplitude

Another element that was pointed out was the amplitude of the measured signal. While the difference in voltage indications between the NI6212 and UNO card may result from different levels of the reference voltage (the topic has been raised earlier), the essence is the observation that in the case of NI6212 card this value is constant (regardless of the cooperating device), while UNO changes the level depending on the power source. Although for a fixed power source, the obtained value can be taken as a constant, but it should be noted when comparing two values measured on different devices.

The solution to this problem may be an independent reference source (which many users recommend).

4.5. Summary

The presented research shows the impact of the AD converter's configuration and the chosen trigerring method on the stability of the sampling frequency and the measurement quality. As mentioned in chapter 1, other factors also influence the measurement time, i.e. the time of sample processing and transmission. The transmission time will be analyzed in section 5.

It is worth mentioning that in addition to triggering the ADC measurement in the interrupt service routine, the ATMega 328P microcontroller has Auto Trigger mode as well. In this mode the ADC can be automatically (without an explicit call of the interrupt service routine) triggered by one of several signals - including the counter. It is possible, for example, to configure the timer to countdown the preset time and once it is counted the next sample measurement performed by the ADC is started internally by the hardware. After each conversion the ADC may report an interrupt to indicate the end of one measurement. The advantage of this mode is the lack of a potential delay between the time count down and ADC triggering in the interrupt routine handling which may occur in the method described in chapter 4.3, especially when multiple interrupts are used by the system. In this mode measurement may be performed even more evenly over time. However, this mode can be configured only by directly setting the appropriate values of registers controlling the timers and ADC - it is not available from the standard Arduino libraries.

5. COMMUNICATION

The chosen communication method imposes the speed at which recorded signal values from the ADC can be transmitted. Among many forms of communication, the authors focused on the two most commonly used methods: serial communication using the RS232 protocol (USART system) and data recording on an SD card.

5.1. Serial port (Baud Rate)

In case of using Arduino UNO as a DAQ card cooperating with a PC the important parameter is the time of sending data to a computer. The baud rate (BR) is the main parameter to be set. The most common settings are in the range of 9600 ÷ 115200 bps (bits per second), but they can also take other values (in the present work a range of up to 2 million bps was tested). In Arduino platform the *Serial* library is used for this purpose. The most commonly used function is *Serial.print()*. It sends any string of characters in the ASCII code. Its extension is the *Serial.println()* command, which also adds the carriage return character '\r' and the new line character '\n' at the end of transmitted string. One of its benefits is easy separation of transmitted values. Data transmission in text form is also convenient because its access to data on a PC

side is possible without having to write your own data transmission and presentation program. All you need to do is to use common terminal emulation programs.

Unfortunately, the transmission in the text form has a significant disadvantage - the bandwidth of the serial link is not fully used. For example, the measurement result from a 10-bit converter can have a maximum of 4 decimal digits. After adding the end-of-line characters, we receive six 8-bit values. Sending the result directly in binary form will require the transmission of only two 8-bit numbers (10 bits of the result and 6 zero bits which complementary counts up to 16 bits), so it will last 3 times shorter. However, it will require their decoding on the PC side.

In order to limit the influence of the value of the transmitted variable (as it is in the text form), the same number of 1023 was sent repeatedly. This is the maximum value that can be returned by the AD converter. At the same time it contains the largest number of characters, therefore it will take the most time. Thanks to this, it was possible to control the correctness of sending and receiving data by checking whether the average value of the received data was 1023 and the standard deviation is equal to 0, which was interpreted as error free receiving data sent from the microcontroller.

In case of Arduino, the connection setting process is simplified to declaring BR values using the *Serial.begin()* command. Other parameter such as word length, parity bit, or stop bit can be set as well, however, the vast majority of applications meet the use of default settings (8 bit word, no parity, 1 stop bit).

Below are presented the pros and cons of using the two most popular methods of obtaining data from Arduino: data transfer in the form of text (ASCII code) and binary one.

Text data

The simplest method of using Arduino UNO for data transmission is the use of Code 5. It is a method of sending only one datum each time. The code has been supplemented with the necessary functions to measure the time of sending. In order to distinguish the measurement data received by the PC from the sent 1023 value, the time value has been increased by 2000.

The code measured the time of sending a single datum each time. The measurement series lasted 30 seconds each.

Code 5

```
unsigned long time1;
void setup(){
  Serial.begin(115200);
}
void loop(){
  time1=micros();
  Serial.println(1023);
  time1=micros()-time1;
  Serial.println(time1+2000);
}
```

The obtained results are presented in Table 9. To present the stability of the measured parameters, the percentage coefficient of variation (Vx) was used.

Table 9 Text send time

baud rate (bps)	*send time (µs)*	*std (µs)*	*Vx (%)*	*fp_{mea} (kHz)*
1 200	50 008.60	1.88	0.004	0.020
2 400	24 989.20	2.17	0.009	0.040
4 800	12 509.10	2.12	0.017	0.080
9 600	6 240.25	0.97	0.016	0.160
19 200	3 120.12	0.69	0.022	0.321
38 400	1 559.93	1.12	0.072	0.641
57 600	1 016.58	1.42	0.139	0.984
115 200	508.05	1.15	0.227	1.968
250 000	238.28	4.88	2.048	4.197
500 000	239.52	5.17	2.157	4.175
1 000 000	213.38	2.80	1.313	4.687
2 000 000	204.00	3.05	1.495	4.902

For clarification, frequencies fp_{mea} corresponding to the times with which µC sends data by USART are also presented. Measurements show that by using popular transmission speeds (up to 115200 bps), data reception frequencies up to 2 kHz can be achieved. However, importantly, even increasing this number to 2 Mbps results in a maximum value of 4.9 kHz only.

It should be noted that for the time measurement for values below 115200 bps, the obtained data sending time was characterized by very good stability, as evidenced by the very low value of V_x (below 0.227%).

Some limitation of the above approach is the dependence of data reception on the free resources of the PC. Therefore, the simultaneous execution of other activities on the computer brings a considerable risk of impact on the time of data reception and, ultimately, even their loss (data reception control is not implemented).

In addition, by analyzing Code 3, it can be seen that if this code was to replace the *Serial.println(1023)* line with *analogRead(A0)*, it would be possible to obtain a simple measurement of signal samples and their transmission. However, it should be emphasized that in the case of measurements of signals from the entire measuring range of the ADC, the number of characters in subsequent samples will be different, which will change the time necessary for data transmission. Triggering subsequent measurements will be implemented programmatically and additionally dependent on the time of sending the previous sample, so the signal sampling frequency will vary which practically disqualifies such an approach in serious measurement applications, where uniformity of sampling usually has key significance for signal analysis.

Binary data

Transferring data in a binary form is a much more economical method than sending them in text

form. A numerical value is sent immediately, without its previous conversion to ASCII characters. This means reducing the number of bytes necessary to send the same amount of information as well as saving the time necessary for both the transmission itself and the previous conversion of data into a text form. When sending data in binary form, the *write()* function should be used which sends individual bytes.

Due to the fact that data from the ADC of the ATmega328P microcontroller are 10 bit, they must occupy two 8-bit bytes. This, in turn, means that 16-bit number to transmit must be divided into two bytes. This can be obtained by the Code 6 presented below.

Code 6 To BIN conversion

```
A = data & 0xFF;
B = (data>>8) & 0xFF;
Serial.write(A);
Serial.write(B);
```

In this case, the variable 'data' is, for example, the result of the measurement with AI. For the assumption that datum value is 1023, it will look like this: from the number 00000011 11111111 (BA), clear the older byte and assign only the second (younger) to A; then the binary number 00000011 11111111 move right by 8 bits, the string will remain 000000000 00000011 and only the younger byte is assigned to B. Next, send byte A and then B. The received number on the PC side should be interpreted as A + B * 256.

For the data transferred in this way, the frequency of data reception by μC was measured again, obtaining the highest possible value of the frequency of sending data in a binary manner. The number 1023 was sent (among others to identify the correctness of the received data). The following code was used:

Code 7 BIN send time

```
  unsigned long time1;
  int data=1023;
  void setup() {
    Serial.begin(9600);
  }
  void loop() {
    time1=micros();        //trans. time
measurement starts
    Serial.write(data & 0xFF);
    Serial.write((data>>8) & 0xFF);
    time1=micros()-time1; // trans.
time measurement ends

    Serial.write(13);      //new line
    Serial.write(10);
    Serial.println(time1+2000); //send
measured trans.time
                                      }
```

Obtained results of time measurements are presented in Table 10. To comparison, the results of sending text data and calculated sampling frequency fp for different BR values obtained for sending text using *println()* are also presented there (selected data from Table 9).

Table 9 Comparition – bin vs ASCII

BR (bps)	μC			
	time (μs)		*fp* (kHz)	
	write()	println()	write()	println()
9600	2076	6240	0.482	0.160
57600	340	1017	2.942	0.984
115200	166	508	6.007	1.968
250000	71	238	14.147	4.197
500000	22	240	46.100	4.175
1000000	21	213	46.791	4.686
2000000	12	204	83.431	4.902

According to the theory, more than triple time reduction in relation to the *println*() function can be observed.

It should be emphasized that in the case of BR ≥ 250000 bps, the value of *fp* does not change almost linearly as it does for lower bps. The reasons for this can vary. Among other things, this may be the result of the implementation of text conversion and transmission support in Arduino standard libraries. At high BR, minor nuances affecting the performance of libraries begin to have noticeable effect because the time remaining between sending consecutive bytes of transmitted information is very short. Thus, the time it takes for the preparation of the next byte to be sent becomes critical. In addition, it is worth noting that the actual bps value that the USART system implements results from the division of processor clock frequencies by integer values and may differ from the set, expected bps [12]. For the same reason the actual bps may also differ from the bps set in the device (eg PC) on the other side of the communication channel. In addition, the CPU clock speed itself may differ slightly from the assumed one (see - considerations in section 4.3.2). In the event of actual discrepancies in the BR in transmitter and the receiver there may be errors during the transmission (e.g. lost or distorted data frames). However, during the tests of this type, errors occurred only for bps above 500000, and in addition they were sporadic (less than 0.002% of the number of bytes sent). Electrical characteristics of connected devices and the connecting wire itself as well as transmission line length may also have a noticeable impact (especially for high BRs) on the amount of transmission errors.

To summarize, although the transmission is possible even for the highest available bps, a sensible practical approach would be to use the speed of up to 250000 bps.

5.2. SD card

This section presents the results of research on determining the time necessary to record data on an SD card.

To support SD and SDHC memory cards, the SD library is used by default and the SPI library is

responsible for communication with the card module. Both libraries should be attached to the code. It is worth mentioning that if the SPI protocol is used, its capabilities may limit the maximum data transfer speed to the memory card.

The times of writing data to the SD card have been checked in two variants: measurement of the continuous recording time for 1 second and measurement of the time of single data recording. Using the function *println()* and *write()*, the file was written in in text and binary formats. For this purpose the code presented in Code 8 was used. Using it we can write to the file *file.txt* 1023 for a period of 1 second.

Exactly 60 files have been saved for each type of data (text and binary). In the case of textual data, the record took on average 412.3 (±49.7) µs (2456 (±258) values). In the case of binary data as expected, the recording time was nearly three times shorter 132.9 (±11.6) µs (7575 (±579) values). A standard deviation is given in brackets.

Code 8 SD card configuration

```
#include <SPI.h>
#include <SD.h>
unsigned long t_max;
File myFile;
void setup() {
  SD.begin(4);
  myFile = SD.open("file.txt",
FILE_WRITE); //open file
  unsigned long t_max = micros() +
1000000; //time to measure
  while (micros() <= t_max) {
    myFile.println(1023);
//print to file
  }
  myFile.close();
//close the file:
}
void loop(){
}
```

A time measurement was also performed using the modified while loop with Code 9. After modification, the code measured the time of a single datum write to the SD card and then sent the measured time to the PC. The variable date has been declared as follows:

int data = 1023;

It was the longest data to measure using ADC converter. The code enables writing data in the form of ASCII characters (the *myFile.println()*) or in a binary form (*myFile.write()*). It is presented in Code 9. In order to measure the data record in a proper way, the second method should be marked as a comment (using //).

Code 9 SD – save time

```
while (micros() <= t_max) {
  time_1=micros();
  myFile.println(data);
//save ASCII
  // myFile.write(data & 0xFF);
//save binary
  // myFile.write((data >> 8) & 0xFF);
//save binary
  time_1=micros()-time_1;
  Serial.println(time_1);
}
```

The measurement was repeated several times, each time saving several thousand values. The average time of writing text data was 239.9 µs. In the case of writing binary data, an average recording time of 77.7 µs was obtained.

The above test showed that the method of writing to an SD card does not guarantee a permanent recording time, as it was characterized by V_x volatility of 10%. This is due to the internal Flash memory structure (depending on the situation, for example, it may perform additional data copying operations).

6. FINAL TESTS

All previously presented results focused on the components affecting the measurement. The final results of the time measurement are presented below and referred to the components obtained earlier.

Fig. 1 presents a simplified diagram of the division of the µC work from the point of view of the time necessary to collect, and send the measured data to a desired place. As it was shown in the above points, the time necessary to implement each of the above depends on:

- The prescaler (affecting the operating time of the ADC converter),
- Data transfer locations - recording to SD card or PC (USART and baud rate parameter),
- The type of transferred data,
- µC time for performing other activities (such as data transform, checking the condition or stopping the measurement, etc.).

Taking into account the previously presented results, it was assumed that:

- BR = 115200 bps - due to the stability of data transmission (Serial port (Baud Rate)).
- Prescaler = 16 - due to the stability of the measurement results. The higher value increased the measurement error (section 4.2.1).

Table 11 contains the results for the above settings. The results from Table 11 can be considered as the maximum frequencies that can be obtained using standard libraries when measuring one program-triggered ADC channel and data transmission to a PC or SD card. This table shows also measured time and time

Table 11 Summaric times and fp (prescaler = 16, BR = 115200 bps)

		time (us)				fp (kHz)	
	Data type	ADC	send	sum	mea	compute	measure
PC	ASCII	15.56	508.05	**523.61**	**507.92**	1.910	1.969
	BIN		166.46	**182.02**	**164.22**	5.494	6.089
SD	ASCII		239.9	**255.46**	**250.78**	3.915	3.988
	BIN		77.7	**93.26**	**101.7**	10.723	9.833

The table shows both: the measured time (mea) and the total time of the previously calculated components (sum).

Obtained frequencies are quite high, because in practice (last column of Table 11) in the case of operations on binary data, they reach 6089 Hz for single channel measurement and sending data to PC and 9833 Hz for SD card data recording.

It should be emphasized that in the above case no mechanism was applied to ensure uniformity of sampling. Table 12 illustrates the results when using interrupts triggered by *Timer1* counter set to CTC mode for various OCR1A register settings. The timer's settings have been determined so as to obtain integer frequencies not greater than the sampling rates obtained with continuous uncontrolled time measurement. Table 12 presents the set values.

Table 12 Interrupt settings

		Timer 1 Mode	OCR1A	fp (Hz)
PC	ASCII	CTC	1249	1600
	BIN	CTC	3124	5120
SD	ASCII	CTC	499	4000
	BIN	CTC	249	8000

Summarizing, when Arduino UNO is used as a DAQ card measuring 1 channel without losing the measurement resolution, it is possible to obtain frequencies up to 5 kHz for on line data transmission to PC in binary form and up to 8 kHz for writing to an SD card.

It should be noted that these are the maximum theoretical values. The samples taken must still be processed. If the processing is included in the interrupt service function, the service time cannot be longer than the time spent until the next interruption because the processor will not be able to finish the interrupt service before the next request. If in the interrupt handling routine it will only read the measurement and its processing will be placed for example in the main loop, then either the samples will be read, but the algorithm will not be able to process them or the sample value can be replaced by another before the algorithm reads it.

7. DISCUSSION

The presented research results allow authors to conclude that using Arduino UNO makes it possible to build a data acquisition system that gives reliable results. It has some limitations resulting from the components used. For example. the technical paper [30], point 1.8, contains a line: „*The ADC accuracy also depends on the ADC clock. The recommended maximum ADC clock frequency is limited by the internal DAC in the conversion circuitry. For optimum performance, the ADC clock should not exceed 200kHz. However, frequencies up to 1 MHz do not reduce the ADC resolution significantly*.”.

On the one hand, the manufacturer recommends the use of ADC clock below 200 kHz, but at the same time does not give reason to believe that the use of 1 MHz clocking will noticeably affect the measurement results.

However, given the drop in the sampling rate resulting from the selection of interrupts, one can safely reduce the value of the ADC clock to get closer to the recommended 200 kHz. This will not affect the total reduction of the sampling frequency of the signal (based on Table 11, ADC time is only 20% of total time in case of send BIN data to SD card and 3% of total time in case of send ASCII data to PC).

When analyzing the results, one must realize that the measurement of values by ADC is only a part of the signal processing. If subsequent samples must be processed on a regular basis (e.g. in control systems), it is necessary to pay attention to the computational efficiency of the microcontroller. It must be sufficient to actually be able to implement the processing algorithm during one sampling period. Although the processing time strongly depends on the specific algorithm used, the dependencies presented in this article may be used to estimate the remaining components of the total processing time of one sample and, consequently, help the reader to make more conscious decisions regarding the use of Arduino UNO in measuring tasks of analog signals.

When analyzing the presented results (Table 8), it should also be emphasized that only triggering the measurement in the time instants indicated by the timer and using interrupts mechanism guarantees a constant frequency of signal measurement. Software triggering does not give such a confidence.

At the same time it is worth paying attention to the accuracy of the clock frequency of the entire microcontroller system. If it is particularly important, then the exchange of the ceramic oscillator to quartz one should be considered, which, however, involves interference in the UNO PCB board.

Resuming, Arduino UNO can be the basis for the construction of a simple DAQ card which can achieve sampling frequencies of several kHz. In

fact the limit lies not in ADC capabilities itself but in the ability to process subsequent samples (µC performance, data transfer to PC or writing them to the SD card).

SOURCE OF FUNDING

This work was supported by the AGH University of Science and Technology [project number 16.16.130.942/KMiW]; Gdansk University of Technology statutory research; University of Warmia and Mazury statutory research.

REFERENCES

1. Chen YC, Shen HY, Chen HY, Hsu CH. Low Cost Arduino DAQ Development and Implementation on an Android App for Power Frequency Measurement, 2016 International Symposium on Computer, Consumer and Control (IS3C), Xi'an, 2016:99-102. https://doi.org/10.1109/IS3C.2016.36
2. González A, Olazagoitia JL, Vinolas J. A Low-Cost Data Acquisition System for Automobile Dynamics Applications. Sensors, 2018;18(2): 366. https://doi.org/10.3390/s18020366
3. Jaskuła M, Łazoryszczak M, Peryt S. Fast MEMS application prototyping using Arduino/LabView pair. Meas. Autom. Monit., 2015; 61(12).
4. Carre A, Williamson T. Design and validation of a low cost indoor environment quality data logger. Energy Build., 2018;158:1751–1761 https://doi.org/10.1016/j.enbuild.2017.11.051
5. Zalabarria U, Irigoyen E, Martínez R, ArechaldeJ. Acquisition and Fuzzy Processing of Physiological Signals to Obtain Human Stress Level Using Low Cost Portable Hardware, Advances in Intelligent Systems and Computing, 2018; 649: 68–78. https://doi.org/10.1007/978-3-319-67180-2_7
6. Corbellini S, Vallan A. Arduino-based portable system for bioelectrical impedance measurement. IEEE MeMeA 2014 - IEEE International Symposium on Medical Measurements and Applications, Proceedings, 2014:1–5. https://doi.org/10.1109/MeMeA.2014.6860044
7. What is Arduino? [Online]. Available: https://www.arduino.cc/en/Guide/Introduction. [Accessed: 20-Apr-2017].
8. Arduino, Compare board specs. 2017. [Online]. Available: https://www.arduino.cc/en/Products/Compare. [Accessed: 14-Apr-2017].
9. Galewski MA. STM32: Applications and exercises in C language, in Polish. Wydawnictwo BTC, 2011.
10. Smith SW. Digital signal processing: a practical guide for engineers and scientists. Newnes, 2003.
11. Greiman A. Arduino FAT16/FAT32 Library. 2017-04-26, 2017. [Online]. Available: https://github.com/greiman/SdFat. [Accessed: 23-Aug-2017].
12. Miesowicz K, Staszewski WJ, Korbiel T. Analysis of Barkhausen noise using wavelet-based fractal signal processing for fatigue crack detection. International Journal of Fatigue, 2016; 83:109–116. https://doi.org/10.1016/j.ijfatigue.2015.10.002
13. Moreno JC, Sánchez AM, Baños A. Recent Advances in Circuits and Systems, no. Csc. WSEAS Press, 1998.
14. Barański R. Sound level meter as software application. Acta Phys. Pol. A, 2014;125(4A): 66–70, https://doi.org/10.12693/APhysPolA.125.A-66
15. Nowoświat A, Olechowska M. Fast estimation of speech transmission index using the reverberation time. Appl. Acoust., 2016; 102:55–61. https://doi.org/10.1016/j.apacoust.2015.09.001
16. Konior M, Klaczynski M, Wszolek W. Reduction of speech signal deformation in patients after nasal septum surgery (septolplasty). Acta Phys. Pol. A, 2011; 119(6A):1000–1004. https://doi.org/10.12693/APhysPolA.119.1000
17. Ozga A. Scientific ideas included in the concepts of bioacoustics, acoustic ecology, ecoacoustics, soundscape ecology, and vibroacoustics. Arch Acoust 2017; 42 :415–21. https://doi.org/10.1515/aoa-2017-0043.
18. Barański R, Grzeczka A. Simply and low coast electromyography signal amplifier. Diagnostyka. 2017;18(4):69-77.
19. Barański R, Kozupa A. Hand grip-EMG muscle response. Acta Phys Pol A 2014;125:A-7-A-10. https://doi.org/10.12693/APhysPolA.125.A-7.
20. Listewnik K, Grzeczka G, Klaczynski M, Cioch W. An on-line diagnostics application for evaluation of machine vibration based on standard ISO 10816-1. vol. 17. JVE International Ltd.; 2015.
21. Atmel Corporation, AVR121: Enhancing ADC resolution by oversampling. 2005.
22. Gammon N. "millis() overflow ... a bad thing?" 2013-08-26, 2013. [Online]. Available: http://www.gammon.com.au/millis [Accessed: 23-Aug-2017].
23. Arduino, Arduino: micros(), 2017. [Online]. Available: https://www.arduino.cc/en/Reference/Micros. [Accessed: 23-Aug-2017].
24. Eli JM. Examination of the Arduino micros() Function | µC eXperiment, 2012-03-17, 2012. [Online]. Available: https://ucexperiment.wordpress.com/2012/03/17/examination-of-the-arduino-micros-function/. [Accessed: 23-Aug-2017].
25. Gammon N. ADC conversion on the Arduino (analogRead), 2015-03-17, 2015. [Online]. Available: https://www.gammon.com.au/adc [Accessed: 09-Jun-2017].
26. Atmel, ATmega48A/PA/88A/PA/168A/PA/328/P DataSheet, AVR Microcontrollers, p. 660, 2015.
27. Atmel Corporation, ATmega328/P. 2016.
28. Gammon N. Interrupts, 2012-01-08, 2012. [Online]. Available: http://www.gammon.com.au/interrupts [Accessed: 23-Aug-2017].
29. Gammon N. Timers and counters. 2012-01-17, 2012. [Online]. Available: https://www.gammon.com.au/timers. [Accessed: 23-Oct-2017].
30. Atmel, AVR120 : Characterization and Calibration of the ADC on an AVR Microcontrollers Application Note. 2006:. 1–15.

Received 2019-02-12
Accepted 2019-05-06
Available online 2019-05-07

Eng, PhD **Robert BARAŃSKI** Postdoctoral researcher in the Department of Mechanics and Vibroacoustics, AGH University of Science and Technology. His scientific interests focus on vibrations and acoustics, biomechanics, EMG signals analysis, dedicated devices for rehabilitation, building flexible diagnostic and measurement systems.

DSc. PhD. **Marek GALEWSKI** works in a Department of Mechanics and Mechtronics at Gdansk University of Technology. His scientific interests concentrate on vibrations measurements, modal identification, applications of Artificial Intelligence, signal processing, measurement systems integration and programming.

DSc. PhD. **Szymon NITKIEWICZ,** Assistant Professor in the Department of Mechatronics and IT Education, Specialist in the School of Medicine, Collegium Medicum, University of Warmia and Mazury in Olsztyn. At his work, focuses on rehabilitation, biomechanics, diagnostics devices.