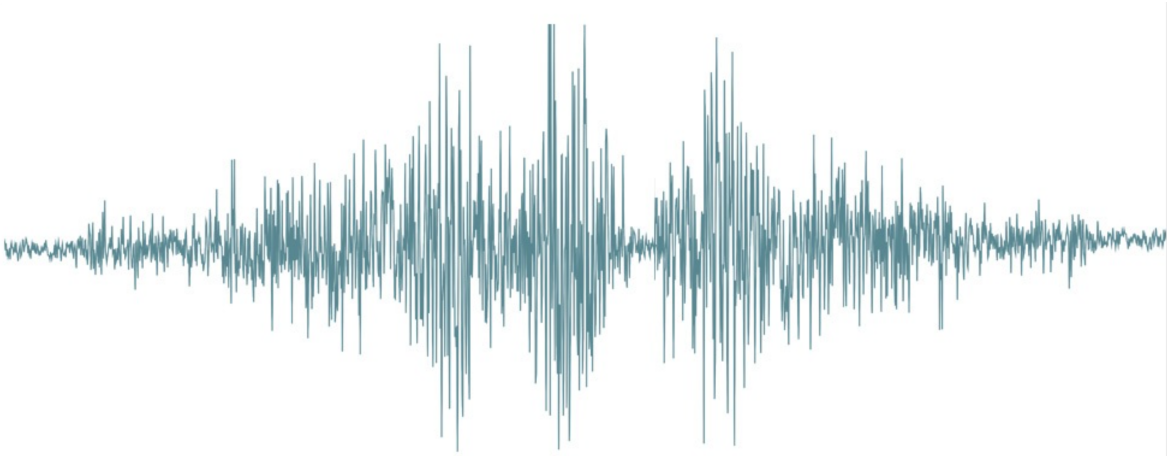
Building a Vibration Measuring System Final Report

February 2, 2023

Author:
Patrik Wall
pawa225@student.uu.se



Abstract

In this project an ATmega325 is combined with a ADXL345 to build an accelerometer. The data is forwarded to a computer via the UART protocol. The data acquisition speed is decided by the ADXL345 which lead to non-constant sampling speeds. The sampling speed is is around 3.2kHz. The system clock speed set to 8MHz and is driven by the internal RC-oscillator of the ATmega325. The frequency respons of the accelerometer is outlined with a frequency sweep by using a vibration speaker as a shaker. The system is showcased on a CPU-fan where changes in the frequency spectrum was observed when adding weights to one of the blades of the fan. An improvement to the system is to use an external crystal oscillator and implement an overflow interrupt to get a consistent sampling speed.

Contents

1 Background

The document at hand is the product of a Project-based learning course at Uppsala University, which is part of the Master Programme in Engineering Physics. The goal of the course is for the student to immerse in a embedded-system-like field which is outside of the ordinary available courses.

In a prior project-course, the author developed a system which aim was to characterize the features of a raw time-domain vibration signal with the use the Matching Pursuit algorithm. The goal and benefit is to foresee faults in a machine such that significant damage of the machine is avoided. This is called predictive maintenance.

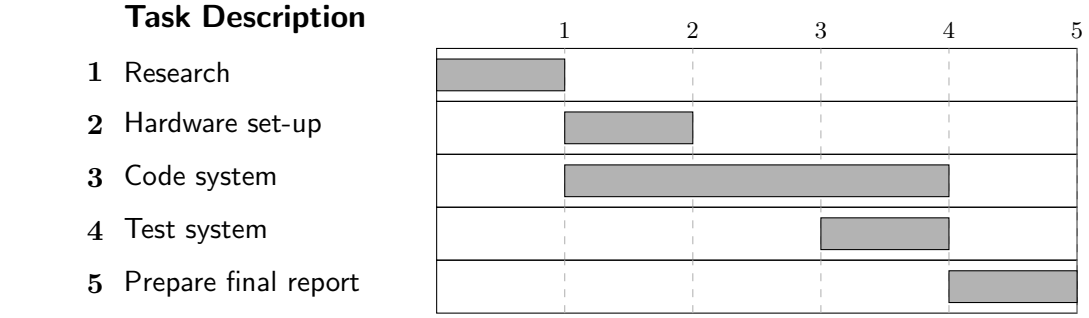
In this project the author widens the scope by building a system which acquires raw vibration data.

1.1 The Project Goal

- Build a vibration measurement system.
- With above-mentioned system, detect a variation in the vibration characteristics on a PC cooling fan.

1.2 Work Distribution and Planning

The project's time schedule which is measured in weeks.



The project team consists of one member (the author) which will carry out his project single-handedly with tutor support from Ping Wu.

2 Theory

2.1 The Basic Principle: Capacitance Accelerometer

A body which vibrates oscillates around a reference point. This phenomenon is measurable in typical kinematic units, displacement, velocity or acceleration. The later is measured in this project which is preformed with an analogue sensor, ADXL345 that uses a capacitive method. The basic operation principle of this accelerometer is outlined in this Section.

The ADXL345 chip is of the *Capacitance measuring* type and is manufactured with micro-machining technology (MEMS). With the use of MEMS, the accelerometers is made in the μm-range. Due to the small size it is commonly found in your smartphone. Furthermore, the MEMS method provides huge flexibility in design which is why the capacitive accelerometer are able to keep a low signal-to-noise ratio in a large variety of measuring ranges. Typically from $\pm\mu 1g$ to $\pm 250g$, where $1g \approx 9.8m/s^2$.

The capacitive accelerometers are built on the inverse-proportional relationship of capacitance (C) and distance (d) between the plates for a parallel-plate capacitor,

$$C \propto d^{-1}. \tag{2.1}$$

The basic principle is having one or more fixed conducting plates in parallel with one or more spring-suspended conducting plates. The spring-suspended plate have one degree of freedom that is orthogonal to the plate orientation. Because of Hooke's law,

$$F = k\Delta d,$$

the displacement is proportional to the immediate acceleration,

$$d \propto a. \tag{2.2}$$

With Eq (2.1) and Eq (2.2) yields the inverse relationship

$$C \propto a^{-1}, [1]. \tag{2.3}$$

An illustration for two different states, $a = 0$ and $a > 0$, of a differential capacitor can be seen in Figure 2.1. Note that Δd the is displacement from the idle state.

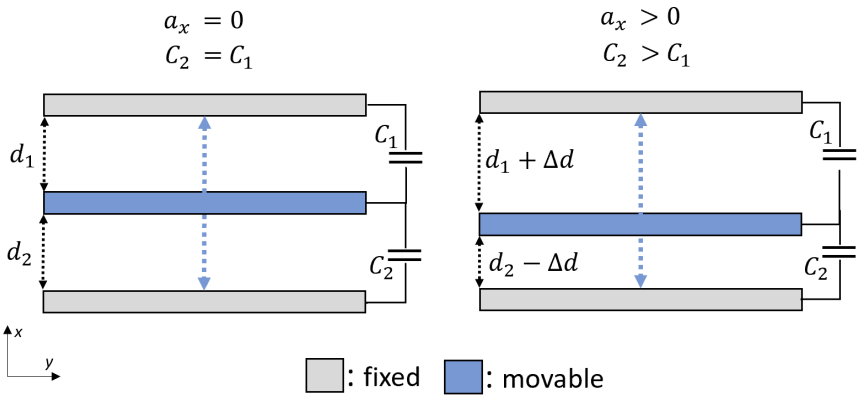


Figure 2.1: An illustration of two states for a differential capacitor. The left is subjected to $a_x = 0$ and the right, $a_x > 0$.

It remains to translate the capacitance to an voltage signal. This is part of the accelerometer package. The voltage level is converted to a digital signal with a ADC component which also is part of the circuitry.

2.2 Resonance

The aim for this section is to introduce the phenomenon of *resonance frequency*. This is showcased by preforming an analytical analysis of the spring-suspended palate mentioned in the section above. In this example the plate is modelled as a particle with mass m. The mass is constrained to one non-friction degree of freedom by an encapsulating case. Furthermore, the particle is connected with spring and a dampener to model the MEMS machined spring suspension. The spring constant is denoted with k and the viscous dampening constant is denoted with c. Let the case be fixated to a vibrating object which models a situation when the accelerometer is used. The vibrations is modeled with a harmonic oscillation, $bsin(\omega t)$. The system and a free-body-diagram of the system is seen in Figure 2.3 where x denotes the mass's relative position to the frame.

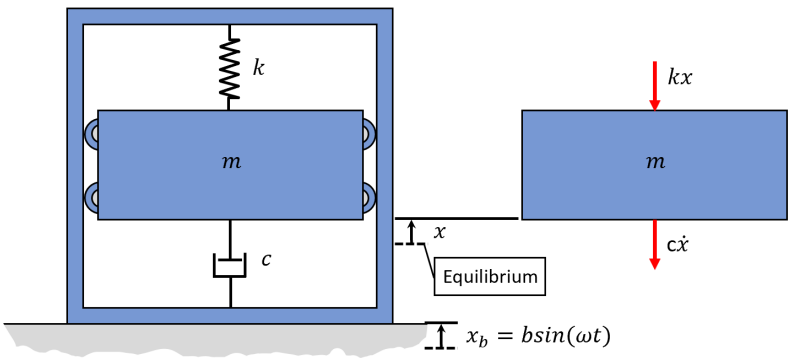


Figure 2.2: Model and free-body-diagram of the model.

The basic differential equation of motion is obtained by applying Newton's Second Law and can be written with a standard notation to

$$\ddot{x} + 2\zeta\omega_n\dot{x} + \omega_n^2 x = b\omega^2 sin(\omega t) \tag{2.4}$$

where $\omega_n = \sqrt{k/m}$ is the resonant frequency and $\zeta = c/(2m\omega_n)$ is called viscous dampening factor.

The solution to Eq (2.4) is the sum of the steady-state solution and the transient solution. In this context only the steady state solution is of interest since the dampener will always kill the transient.

The steady-state solution, x_p can be found with the ansatz

$$x_p = X sin(\omega t - \phi), \tag{2.5}$$

where ϕ represents the phase lag which the response lags the pushing frame. Solving for X yields

$$X = b(\omega/\omega_n)^2 M \tag{2.6}$$

where M is

$$M = \frac{1}{\sqrt{[1-(\omega/\omega_n)^2]^2 + [2\zeta\omega/\omega_n]^2}}, \tag{2.7}$$

which is called magnification ratio. Suppose that $\omega = \omega_n$ then Eq (2.6) with Eq (2.7) becomes

$$X(\omega = \omega_n) = \frac{b}{2\zeta}. \tag{2.8}$$

When the frequency of the pushing system approaches the natural frequency the steady-state solution becomes a strong function of the viscous dampening factor. In Eq (2.8) it can be seen that if $2\zeta > 1$, the system vibration amplitude X is larger than the pushing system's amplitude b. An example of this is pushing a person on a swing; a relatively high amplitude is maintained by small pushes with the right frequency. The amplitude ratio X/b can be derived from Eq (2.7) which is commonly used when studying this effect. A plot of X/b as a function of the frequency ratio with different values of ζ is shown in Figure 2.3. It can be seen that the magnification diverge as the ζ approaches zero when $\omega = \omega_n$ [5]. In practise, this is situation not wanted as the spring is running a high risk of breakage. The spring in a model could for example model restorative forces in a skyscraper.

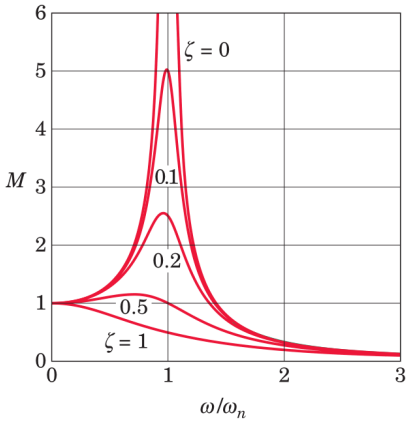


Figure 2.3: Model and free-body-diagram of the model.

Hence, is important to consider the resonant frequency ω_n and viscous dampening factor ζ when designing a system. For an accelerometer the dampening ratio should designed such that the magnification ratio is close to unity over the largest bandwidth possible which is between $\zeta = 0.5$ and $\zeta = 1$.

This analysis showcases the importance of vibration analysis in general. A poorly designed system could be subjected to material failure due to excessive forces. In worst case this could cost human lives. In practise, a system could inherit several resonance frequencies which needs to be considered and *tested*. A test could be preformed by inducing vibrations to an object with a *Shaker* and measure the vibrations with an accelerometer. By doing a frequency sweep with the Shaker the resonance frequencies could be found as peeks in the amplitude plot.

2.3 Communication Protocols

The way hardware communicate serially could be compared to how humans talk; the involved speak the same language, and one speaks at a time. In digital communication, protocols correspond to language. In this project the SPI protocol is used for communication between the MCU and the sensor. The UART protocol is used to transfer data from the MCU to a computer. These protocols are introduced in this section.

2.3.1 The SPI Protocol

The Serial Pheriphial Interface (SPI) protocol is a synchronous serial communication which means the data and timing is sent simultaneously. Motorola (today owned by Nokia) developed this protocol in 1980's which quickly became an industry standard.

The bus structure consists of one dedicated master and one or more slaves. What signifies the master is two-fold. Firstly, the master provides the clock signal to the slave(s) via a clock channel, commonly named SCLK. Secondly, only the master can initiate a transmission. A

transmission is initialized with a *chip select* (CS) channel such that when set to digital zero, the slave listens. When set to digital one; the slave stops listening.

The SPI protocol is categorized as a *full-duplex* system, this means that the data is simultaneously transferred in both directions. This is made possible through two *shift-registers* which is connected in series through ports and wires. Conceptually, the connected registers could be seen as one register. The connecting ports are commonly named *master-in-slave-out* MISO and *master-out-slave-in* MOSI, Figure 2.4 illustrates how registers are connected. During a transmission the register is shifted once for each clock cycle, hence a synchronous serial communication.

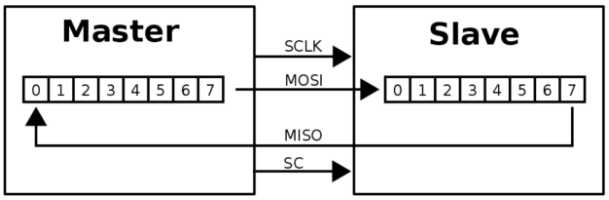


Figure 2.4: Illustration of how the shift-registers are intertwined for SPI communication, with MSB configuration and which other two channels, SCLK and CS is used.

Before a transmission, the master and slave needs to be agreed on three configurations. Two of them considers timings, namely *Clock Polarity* (CPOL) and *Clock Phase* (CPHA). In addition, CPOL and CPHA have two modes each, which results in 4 different SPI-modes. Firstly, CPOL modifies whether the register is shifted on rising or falling edge of the clock cycle. Secondly, CPHA modifies whether the register is shifted on the first edge or second edge. Thirdly, the master and slave needs to be agreed on whether the *most significant byte* (MSB) or *least significant byte* (LSB) comes first. An example of MSB first is illustrated in Figure 2.4.

It is convenient to connect more slave units to a master since all slave units can share the same SCLK, MISO, MOSI -ports. However, the cost is one additional chip select pin per added slave. The sharing of ports is made possible since a slave only "listens" when its chip-select is set to zero.

2.3.2 The USART Protocol

UART is an abbreviation for Universal Asynchronous Receiver Transmitter. This means that the data is transmitted in a continuous stream without a clock signal as opposed to the SPI protocol. Hence, the bus structure consists of two physical connections , one for transmission and one for acquisition. The ports are commonly named Tx, and Rx, which are connected as shown in Figure 2.5.

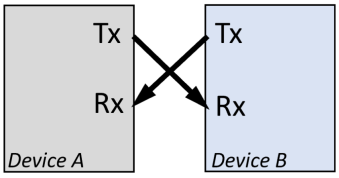


Figure 2.5: Illustration of how to connect two devices with UART communication.

In this protocol, the data package are framed with a *start bit* and a *stop bit* as illustrated in Figure 2.6. The *parity bit* is an optional feature and is not covered here since it is not used in the project.

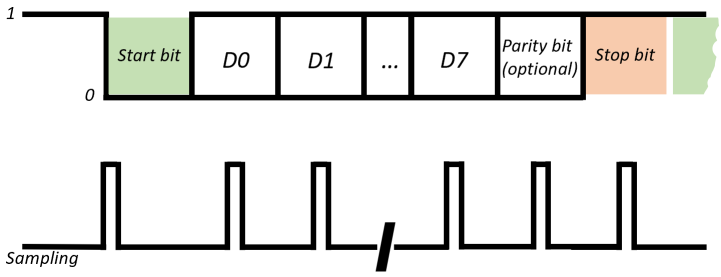


Figure 2.6: Serial frame of UART.

The start bit synchronizes the receiver such that the sampling is executed at half of a bit's life time. A prerequisite is that the transmitter and receiver have a prior agreement of the following configurations: data size, parity bit, one or two stop bits is used and the speed which with the data is sent. A common way to abbreviate the UART configuration is (speed)-(parity)-(data size)-(number of stop bits). An example could be 250000-N-8-1, where N means no parity bit. Furthermore, in this context the speed is called *baud rate* which translates to bits per second (bps).

The baud rate could be any speed within reason, hence the name "universal". However, the system clock frequency of both the receiver and transmitter needs to be considered when choosing baud rate. Namely, an acceptable common divisor between the systems' clocks is needed. In addition, the divisors are usually limited to a multiple of 8 or 16 due to the nature of a clock divider. An easy solution is to use a Standard Baud Rates List. These baud rates divides evenly with common crystal oscillator frequencies.

It is important for both clock sources of the receiver and transmitter to be accurate. The serial frame package could otherwise be misread if the accuracy is not within a tolerable level. For example, a data bit could be sampled twice or the stop bit could be accepted as a data bit which cause errors. When two clocks are not in sync is commonly called *clock skew*.

3 Hardware

This section outlines all hardware parts that is applied in this project.

3.1 Specifications of the ADXL345

This section discusses the sensor's sensitivity, range and measurement errors. How to configure the sensor and extract measurements is also covered.

Through internal read-and-write registers, the ADXL345 is configurable to different modes. For instance, the resolution have two different modes: ten-bit or *full-resolution*. In ten-bit mode the resolution is simply 10 bits long. And in full-resolution mode, the resolution increases with the configurable measurement limit ranges. Table 2.1 shows the different Measurement ranges and their corresponding resolution with full-resolution mode enabled. Secondly, this table also specify the configurable measurement limit ranges.

Table 3.1: Tabular of the corresponding resolutions each measurement range mode have in full-resolution mode for the ADXL345.

Measurement range [g]	**Full-resolution [bits]**
±2	10
±4	11
±8	12
±16	13

Two 8-bit registers is needed to be read in order to get one measurement. These values are *Two's Compliment* of a signed 16-bit integer. This is a common method of dealing with signed formats. One register is the *most significant byte* (MSBy) of the signed 16-bit integer and the other register is the *least significant byte* (LSBy). This is further exemplified in Figure 3.1.

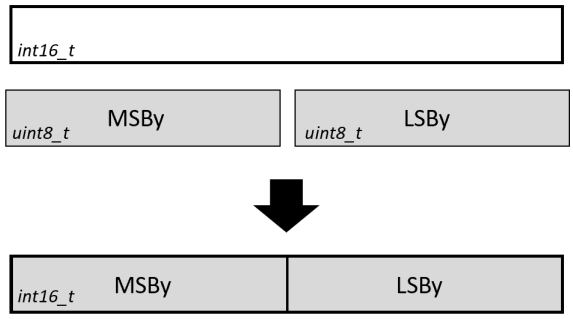


Figure 3.1: An illustration of how to process 8-bit two's compliment into a 16-bit value.

In this project it is desirable to have as fast sampling speed as possible. The reading rate is

configurable but limited to a top speed of 3200Hz.

The measurements is sensitive to temperate changes. This is specified to $1\%/C^\circ$ starting from $25C^\circ$ which is *factory calibrated*. The true acceleration value is not the highest priority, thus it is assumed this drift will not significantly affect the results.

The nonlinearity is specified to $\pm 0.5\%$ of full scale. The author assumes this refers to

$$nonlinearity\ (\%) = \frac{E_{max}}{Y_{max}} * 100 \tag{3.1}$$

where E_{max} is maximum input deviation from ideal linear curve and Y_{max} is the maximum full-scale output [2]. The Equation is is exaggeratedly exemplified in Figure 3.2.

Environmental factors typically affect the linearity. For example temperature and humidity. Moreover, design parameters could also affect the linearity. For example, Eq (2.1) assumes that the area of the plates are much larger that the distance between the plates. Also, the viscous dampening factor, as described in Section 2.2, could affect the linearity.
The degree of nonlinearity is neglected in this project since the measurements are sampled in standard room conditions. In addition, the specified deviation, $\pm 0.5\%$ is small.

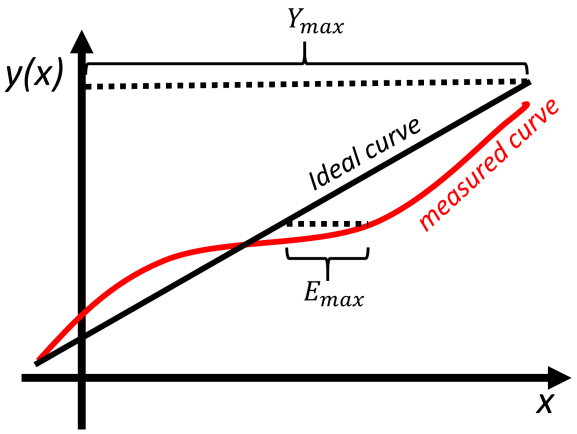


Figure 3.2: Illustration of a maximum measurment deviation from an ideal curve.

In some applications, the accelerometers needs to be re-calibrated to give trustworthy measurements. However, in this application it is not mandatory since the amplitudes in relation to time-domain is of interest.

3.1.1 The SPI Interface of ADXL345

This section summarizes findings regarding the SPI interface of the accelerometer used in this project, ADXL345.

The internal registers of the ADXL345 is eight bits long but the addresses are six bits long. The two most significant bits are used as additional instructions. The 7th bit is called *multiple-byte* (MB) bit. If this bit is set, the accelerometer will continuously transfer values of the registers in numerical order, given that the CS-pin is held to zero. That is, after eight clock cycles, the value of the next register is transferred during the next eight clock cycles, et cetera. The 8th bit is named *read-and-write* (RW) bit which instructs the accelerometer whether the master intends to read (value=1) or write (value=0) to called register. A timing diagram for how data is read from the accelerometer is seen in Figure 3.3. Note the not prior mentioned time definitions in

Figure 3.3 ("t" with different subscripts). These are time constraints in the nano-range and this project aim to read in $3.2k$Hz. The constraints are assumed to be negligible due to the large difference in magnitudes. However, worth-to-mention is that the maximum SCLK speed is specified to 5MHz.

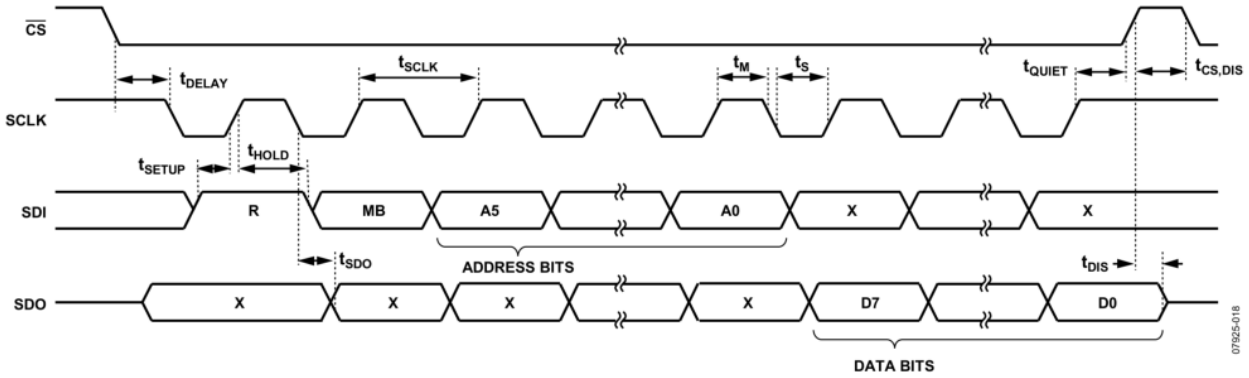


Figure 3.3: A timing diagram for how data is read from the accelerometer. The read-and-write- and the multiple-byte bit is set to 1. The chip-select pin is first set to zero where-after the first bit is shifted on the second edge of the clock signal. The answer-value of the slave is coming sequentially to the last bit of the address, this data should be shifted with a dummy-byte.

It is clear that the applied SPI interface here is not "truly" operating as a full-duplex system since the data-sent and data-received is sequentially sent. A consequence of this is that the master device is required to "re-write" its shift-register when expecting an answer. This is commonly known as sending a *dummy byte*.

Furthermore, it is not clear from the timing diagram but the SPI interface of the ADXL345 operates with both CPOL and CPHA set.

Some internal registers of the ADXL345 are read-only. Clearly, it is common practice to be extra cautious not to accidentally write to these registers; writing to these registers could in worst case cause damage the device.

3.2 The ATmega328

This RISC-architectured 8-bit MCU belongs to the AVR family of chips and is manufactured by *Microchip Technology*. The dimensions are around $37x7x3mm$. The flash type program memory is 32 kilo byte (kb). It has 1 kb EEPROM and 2 kb SRAM. It is estimated that these memory sizes are well within the scope of the project while maintaining flexibility for further development. At our disposal we have 23 general purpose I/O pins, external and internal interrupts, timers, a serial programmable UART and an SPI serial port.

3.2.1 Baud Rate Clock Accuracy

In this project the internal RC-oscillator of the ATmega328 is driving the baud rate clock generator. This is generally not common practise since a clock skews are likely to happen between the receiver and transmitter. More on this in Section 2.3.2. Ideally, the MCU should be driven with an external crystal oscillator which are more accurate. Such oscillator is not used in this since it was not considered in the planning. It is assumed that the clock skews is within tolerable limits. The start bit does synchronize the clock so in best case the RC-oscillator is sufficient. It is also possible to tune the RC-oscillator which could pose as a back-up plan.

3.3 USB Programmer

There are different ways of uploading instructions to AVR MCUs. In this project the C-code program is written and compiled with the Atmel Studo v7.0 software. The compiled code is transferred with the *AVRdudess* software to the USB programmer. The USB programmer is of the type USBasp and sends the program to the MCU's flash program memory through hardware. In addition, for a Windows machine, which is used in project, it is required to install USBasp drivers.

3.4 The FT232R

The capability of sending data in "real-time" to a computer is an important feature. There are several options to accomplish this. The plan in this project is to send the data to a PC through a USB Virtual serial port by using the serial UART protocol. However, a PC cannot directly interpret the UART signal from the MCU. By using a FT232R based breakout board the UART signal is converted to a USB protocol. A visualization of how the MCU transferrs data to a computer is seen in Figure 3.4.

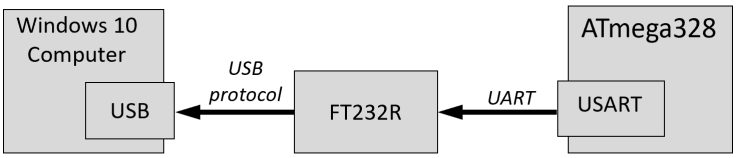


Figure 3.4: Overview of the workflow for transferring a program to the ATmega328. The overlapping bump signifies a physical port of the device.

This signal could potentially be connected to any real-time processing system for signal analysis and monitoring.

3.5 Shaker

A shaker is a transducer which convert electrical energy to vibrations. This is commonly used in vibration testing both for endurance tests or modal testing. In this project an Adin KBBT is used which is a 26 watt vibration speaker with an internal rechargeable battery. Further specifications is not found on the manufacture's homepage [3]. However, the original purpose of this device is to play music. Thus, it is assumed that the frequency response is adequately designed for the human hearing bandwidth, $\approx 20-20\,000$Hz. A further assumption is that we bandwidth which is measured in this project, $50-1\,600$Hz has constant gain. A picture if the shaker is seen in Figure 3.5

Figure 3.5: Picture of an Adin KBBT.

3.6 Mounting

The mounting of a vibration sensor have different impacts on the frequency response. The six most common mounting methods and their frequency response is compared in Figure 3.6.

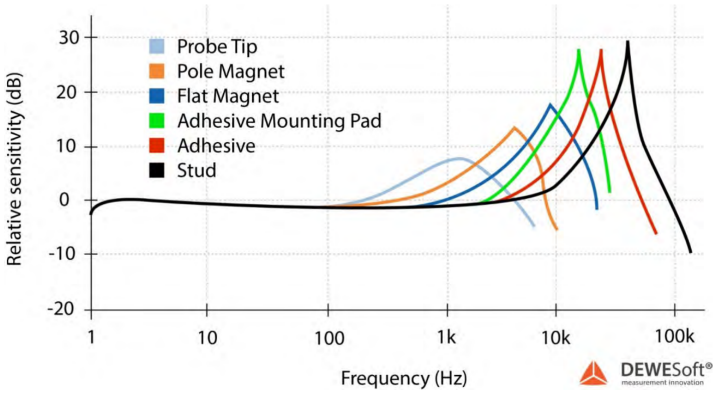


Figure 3.6: Overview of the workflow for transferring a program to the ATmega328.

3.7 CPU-Fan

The accelerometer is mounted to a CPU-fan to showcase the system. The diameter of the fan is 70mm and is DC powered. A photo of the CPU-fan is seen in the Appendix.

4 Software

In this section follows short introductions to the software applied in this project.

4.1 Atmel Studio 7.0 IDE

In this project the C-code program will be written and compiled with the Atmel Studio version 7.0 software [6].

4.2 AVRdudess

There are different ways of uploading instructions to AVR MCUs. In this project AVRdudess is used which will transferred a compiled hex-file to the USB programmer. Furthermore, AVRdudess offers a convinient user interface to set fuse bits. An overview of the workflow for transferring a program to the ATmega328 is seen in Figure 4.1.

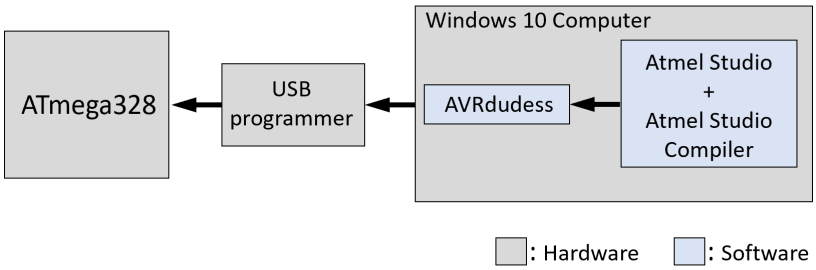


Figure 4.1: Overview of the workflow for transferring a program to the ATmega328.

4.3 PuTTY

For simplification, this project will log the signal with a terminal emulation program for post-processing. As terminal emulation program, puTTY is picked as the author already had the software installed and experience using it.

4.4 Matlab

For post-processing Matlab is used and this was picked for similar reasons. It should be mentioned that one could set up real-time data acquisition on Matlab. However, this will probably not be done in order to save time.

4.5 Tone Generator

The shaker is connected to the 3.5mm audio port of a computer which plays tones from an online tool. Link to the tool: https://onlinetonegenerator.com/frequency-sweep-generator.html. Through the tools GUI timed frequency sweeps is configurable as well as timed single tone playbacks.

5 Implementation

5.1 System Overview

A illustration of the complete system can be viewed in Figure 5.1.

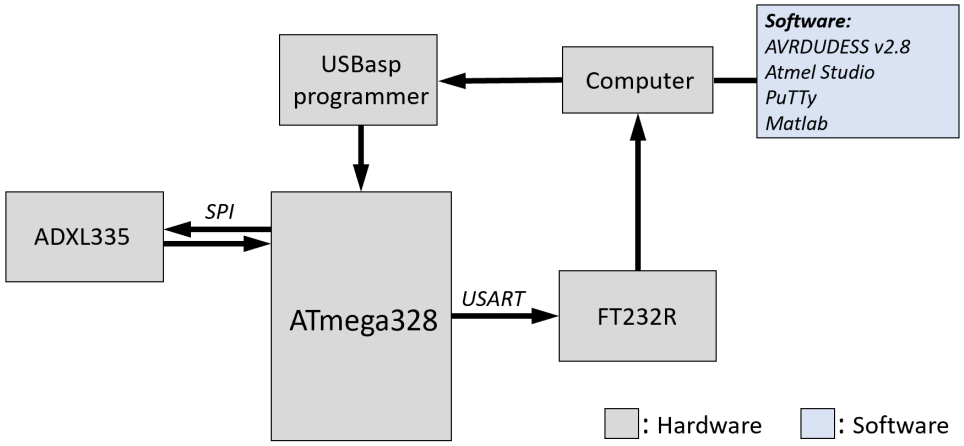


Figure 5.1: Illustration of the complete system.

5.2 ADXL345 Setup

The key registers used in this project is listed in Table 5.1.

Table 5.1: Tabular of the corresponding resolutions each measurement range mode have in full-resolution mode.

Register name	**Bit-names**	**Function if set**
0x2C — BW_RATE	D0, D1, D2, D3	Selects max output data rate
0x2E — INT_ENABLE	D7	Enable DATA_READY interrupt
0x2F — INT_MAP	D7	Interrupt on INT2-pin
0x31 — DATA_FORMAT	D0, D1 D2	±16g range
0x2D—POWER_CTL	D3	Enable measuring mode

The measuring limit range depend on the 3 least significant bits in the DATA_FORMAT register. Figure 5.2 shows what configurations corresponds to each range.

Setting		
D1	**D0**	***g* Range**
0	0	±2 *g*
0	1	±4 *g*
1	0	±8 *g*
1	1	±16 *g*

Figure 5.2: Bit configurations of D1 and D2 which sets the ADXL345 in different measuring ranges.

The tests in this project will be run with $\pm 16g$ range and with full-resolution mode. This is picked to get the best resolution. The amplitudes of the measurements is fitted by adjusting the volume slider in the computer for the shaker measurments. The volume are fitted such that the test's maximum amplitudes fits in the range.

5.2.1 Interrupt Configuration

Furthermore, the ADXL345 is configured such that the INT2 pin on the breakout board goes high when the a new measurement is ready. This configuration was not origianlly planned but implemented during the project. More about this in Section 5.6.2.

5.3 ATmega328 Setup

How the MCU communication interfaces are setup is discussed in this section. In addition, how the fuse bits are configured of the MCU.

5.3.1 The SPI interface of ATmega328

How the ATmega328 SPI-hardware is setup in this section. The method is reading through the SPI-chapter in the ATmega328 datasheet. The setup is made through modifying certain registers, similarly to how the ADXL345 was setup. Furthermore, the byte pointers and its bit-positions are denoted with names in the datasheet. In C-code, the names can be used as opposed to the numeric values since the compiler knows what pointers and bit-positions the names are associated with. This is convenient when programming because of two reasons. Firstly, the names hints the function which makes the code easier to read. Secondly, the coder saves time by only focusing on the associations instead of "counting bits".

In Figure 5.3, the bit-names associated with bit-position of the *SPCR* register is seen.

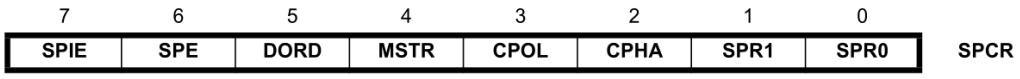


Figure 5.3: asd.

The 6th bit, associated with *SPE*, controls whether the SPI is enabled (SPE=1) or not (SPE=0). In C-code this bit can be set with the following code

```
SPCR = (1<<SPE);
```

Key registers used in this project, with regards to the SPI interface, is listed in Table 5.2.

Table 5.2: Tabular of the corresponding resolutions each measurement range mode have in full-resolution mode.

Register name	**Bit-names**	**Function**
SPCR	SPR0, SPR1	SPI Clock rate
	CPHA	Clock phase
	CPOL	Clock polarity
	MSTR	Master select
	DORD	Data order
	SPE	SPI enable
PRR	PRSPI	Enable SPI-moduel clock
SPSR	SPIF	SPI interrupt flag
	SPI2X	Double SPI speed
SPDR		SPI data register

The bits: SPR0, SPR1 and SPI2X configures SCLK speed with a divider of the system clock speed, f_{osc}, such as shown in Figure 5.4. The SPI interrupt flag on bit SPIF function such that it is set during a transmission. By polling this register, after a byte has been written to SPDR, possible overwrites are avoided. The polling could be done with the following C-code,

```
while(!(SPSR & (1 << SPIF)));
```

SPI2X	SPR1	SPR0	SCK Frequency
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

Figure 5.4: A Table displaying what bit setting of SPI2X, SPR1 and SPR0 results in what SCLK frequency.

5.3.2 The UART Interface of ATmega328

Key registers used in this project, with regards to the UART interface, is listed in Table **??**

Table 5.3: UART setup.

Register name	Bit-names	Function if set
UCSR0B	TXEN0	Enable transmissions
UCSR0C	UCSZ01, UCSZ00	Sets 8 bit transmission protocol
UBRR0H UBRR0L		Baud rate registers
UCSR0A0	UDRE0	USART Data Register Empty
UDR0		UART data register

The UDRE0 bit is equivalent to the SPI module's SPIF bit. Thus this bit is could be polled after writing to the UDR0 register as exemplified in the following C-code,

```
while (!( UCSR0A & (1<<UDRE0)));
```

Furthermore, the UART interface is by default set to a N-8-1 configuration.

5.3.3 FUSE bits in ATmega328

FUSE bits are non-volatile registers which controls certain configurations of the MCU. The program AVRDUDESS offers a graphical interface to change the fuse bits. The interface is seen in Figure .

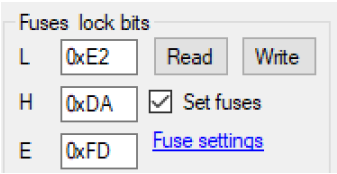


Figure 5.5: A Print screen of the AVRDUDESS Fuse lock bit configuration interface.

Clicking the Fuse settings link in Figure 5.5 opens a online tool which helps the user to choose values of the fuse bits for certain functions. The URL is,

https://www.engbedded.com/fusecalc?P=ATmega328&V_LOW=0xE2&V_HIGH=0xDA&V_EXTENDED=0xFD&O_HEX=Apply+values.

5.4 SPI clock speed calculation

Recall that the specified max speficified SPI clock speed of the ADXL345 is 5Mhz. Suppose that the ATmega328 is running at 8Mhz. Therefore, the best-pick is $8MHz/2 = 4MHz$. That is, setting SPI2X in accordance to Figure 5.4. Furthermore, one measurement requires three bytes of SPI transmissions; the address with multiple bit set and the read of MSBy and LSBy. It follows that the maximum measurement sampling rate with $SCLK = 4MHz$ is $0.25MHz$. To clarify, the answer could be calculated with

$$f_s = ([SCLK_{freq}]^{-1} * B * 8)^{-1}, \tag{5.1}$$

where B is the bytes needed for one measurment and $SCLK_{freq}$ is the SPI clockspeed.

It is obvious that for this ideal case, the data sampling rate is above the specified data rate, $0.25MHz > 3.2kHz$ and would not cause a bottle neck. In practise however, the data is also sent to the PC with UART before next sample which is considered in the implementation.

5.5 The UART Project Implementation

5.5.1 Baud Rate Clock Generator

A snippet of the ATmega328's UART clock generation logic presented in Figure 5.6.

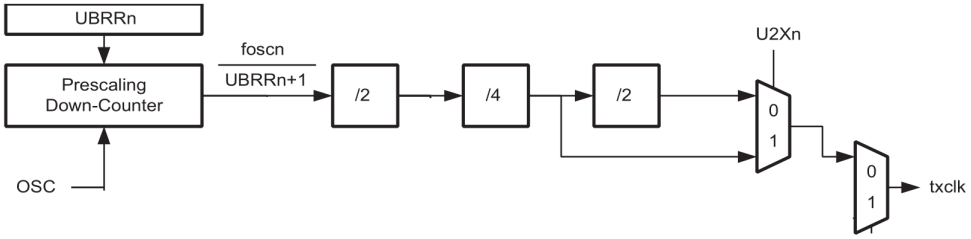


Figure 5.6: Serial frame of UART.

Particularly, it is seen that the UBRRn register sets a threshold for the *Prescaling Down counter*. Furthermore, the snippet shows three clock dividers in series which results in a divide-by-16, assumption that the U2Xn bit is not set. To clarify, the Prescaling Down-counter needs to underflow 16 times before the baud rate clock generator is switched. In conclusion, the required value of the UBRRn register could be calculated with

$$UBRRn = \frac{f_{osc}}{16[baud\ rate]} - 1 \tag{5.2}$$

where f_{osv} is the system clock speed and $[baud\ rate]$ is the target baud rate.

5.5.2 Calculating Minimal UBRn value

The minimal frequency which the UART needs to operate at to not cause a bottle neck is calculated with

$$f_{UART} = (f_{ADXL}^{-1} - f_s^{-1})^{-1} \tag{5.3}$$

where f_s is the frequency at which the SPI protocol transmits a measurement in accordance to Eq (5.1) and f_{ADXL} data rate of the ADXL. Suppose that the SPI interface is transferring a measurement at 0.25MHz and the max measuring rate of the ADXL356 of 3.2KHz. By Eq (5.3) the UART speed needs to be at least 3.25kHz. Recall that the measurement is 16 bits long and assume a N-8-1 UART protocol. Thus, 20 bits is needed to be transmitted at 3.25kHz which corresponds to the minimal baud rate of 65kbps. In this case, 250kbps is the best choice of Standard Baud Rate which with Eq 5.2 yields $UBRRn = 1$ with a system clock of 8Mhz.

5.6 Implementation

5.6.1 Problem with hardwired I2C

The first received accelerometer chip, ADXL345, is part of a multi-featured breakout board, named GY-85. This turned out to be problematic as this model is hardwired to set the accelerometer in a permanent I2C mode. The datasheet of the ADXL345B describes that I2C mode is enabled by pulling the *chip select* (CS) pin to the VDD_IO pin. With that said, it is seen in the schematic of the GY-85 breakout board that the CS pin is hardwired in such way, see Figure 5.7.

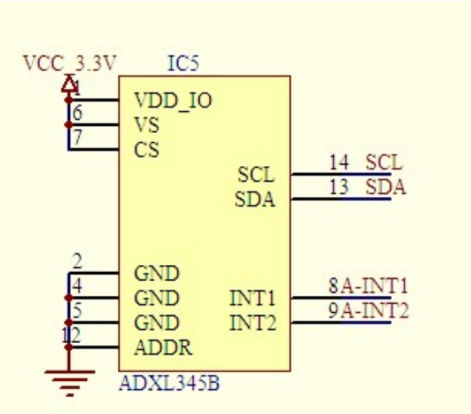


Figure 5.7: The schematics of how ADXL345 is wiered to the GY-85's pin-out. The red-brown text color denotes the GY-85's pin-out

The SPI protocol is judged to be preferable (as opposed to I2C) in this project. The biggest argument being that SPI protocol generally supports higher data transfer speeds. Secondly, the

project team never implemented this protocol prior to this project. Hence, the SPI protocol is also preferred for learning purposes.

Luckily the author had a similar breakout board laying around. This is a GY-291, which has the SPI at the users disposal, see Figure 5.8. Further details regarding the SPI protocol is postponed for the final report.

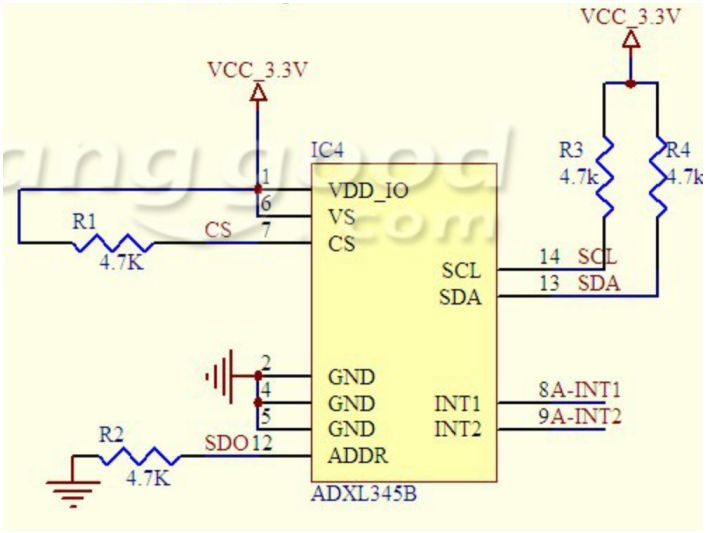


Figure 5.8: The schematics of how ADXL345B is wiered to the GY-291's pin-out. The red-brown text color denotes the GY-85's pin-out.

5.6.2 Slow Sampling Speed Problem

During the initial testing it was realized the sampling frequency was not as fast as expected. The cause was believed to be the frequency which the MCU requested measurments from the accelerometer. This frequency was "alot faster" than the specified 3.2kHz. An example of is seen in Figure 5.9 which is the result of sampling a three seconds sinusoidal 120Hz signal. Note, in this test the accelerometer is mounted with a quick-fix method on the shaker. In this program version the MCU calls measurments and sends it to PC, in 8Mhz with no delay.

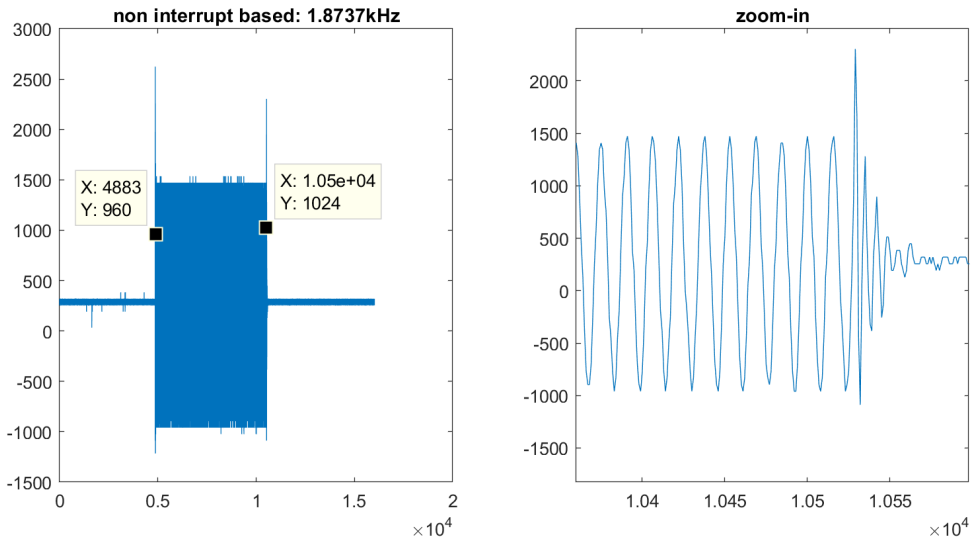


Figure 5.9: Results of measuring three seconds 120Hz sinusoid with the initial program version (right). A zoom in of the results at the end.

With ocular inspection the number of elements within the three second sample is counted and divided by three. This yields the sampling frequency of almost 1.9kHz.

It was believed that the above mentioned "over sampling" could be the cause of slower sampling frequency. Consequently the program was remade. The revised program is an interrupt based method where the ADXL345's DATA_READY is utilized. The DATA_READY function forwards a high signal on the breakoutboards INT2 pin as soon as a new measurment is available. This interrupt signal is connected to the MCU which is programmed with a interrupt service routine such that when the signal is high, the MCU requests the measurement with no delay. In short, the ADXL325 lets the ATmega328 know when a new measurement is ready. The result of the new program is show in Figure 5.12.

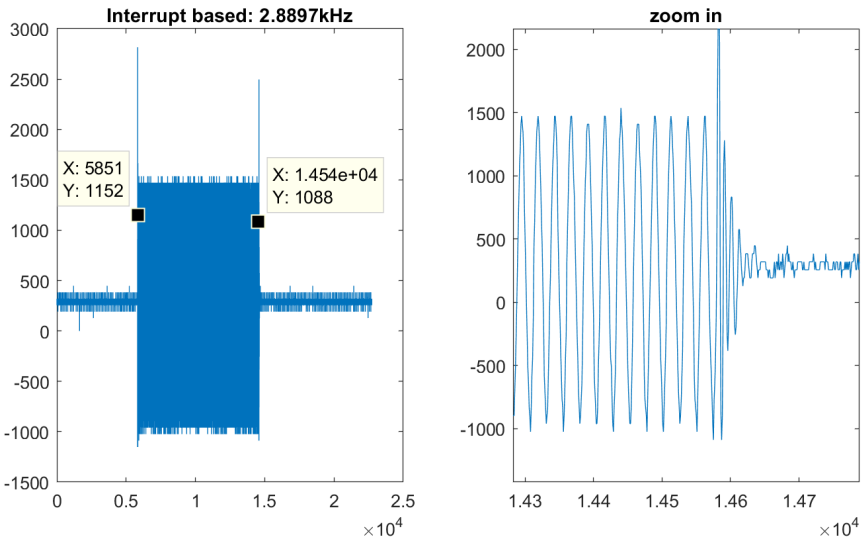


Figure 5.10: Results of measuring three seconds 120Hz sinusoid with the interrupt-based program (right). A zoom in of the results at the end.

In comparison to the old program, the frequency increased with 1kHz.

However, it is concerning that in the data sheet of the ADXL345 it is described that *"several reads may be needed to clear the DATA_READY bit"*. In addition, we have no knowledge whether the DATA_READY interrupt is consistent in terms of a timing. This could possibly be temperature dependent. A backup plan is to again let the MCU decide when to request measurements but count-down interrupt based. But for this approach it would be preferrable to run the MCU of a crystal oscillator to achieve consistent timings.

Moreover, the specified max speed (3.2KHz) is not obtained in this series and the cause of this is not known.

5.7 Shaker Mounting

A scrap part from a DVD-reader is used as a mounting device to mount the accelerometer to the shaker. The basic shape of the part is seen from two point of views in Figure 5.11.

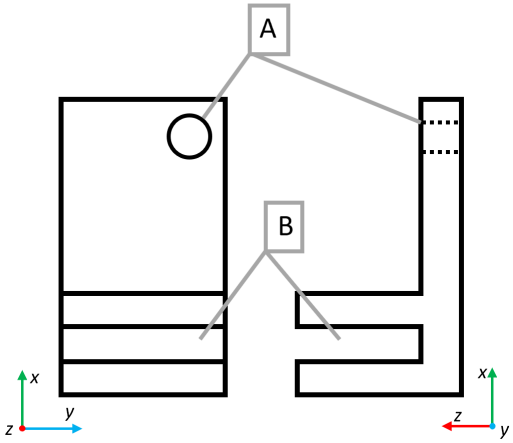


Figure 5.11: A Schematic of the mounting piece used to mount the accelerometer to the Shaker. Two point of views is seen. Label A is the hole and label B is the hook.

The shaker plate is pushed into the hook of the mounting piece which is kept in place by friction. The shaker plate has a plastic film which helped to skewer the mounting piece. A pincer was used to bend the hook to the right tolerance for a good friction fastening. It is assumed this method of mounting gives similar properties as a Stud mounting. The accelerometer is fastened with a nut to the mounting piece.

The mounting piece is not optimal, particularly due to the length in x-axis. In addition, the accelerometer's center off mass is offset to the pushing force of the shaker which causes displacement in z-axis. This results in a pendulum motion of the accelerometer and the mounting piece which probably affects the results.

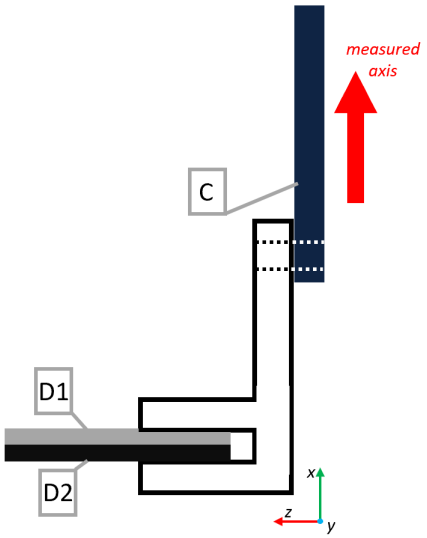


Figure 5.12: A Schematic of the mounting piece with mounted accelerometer and attatched to the shaker. Label C is the accelerometer. Label D1 is the plastic film on the shaker base. Label D2 is the metallic shakerbase.

Camera photos of the accelerometer mounted on the mounting device and shaker can be found in the Appendix.

5.8 CPU-fan Mounting

The accelerometer is mounted to the CPU-fan similarly to how the Mounting device was mounted to the shaker. A photo of the accelerometer mounting to the CPU-fan is put in Appendix.

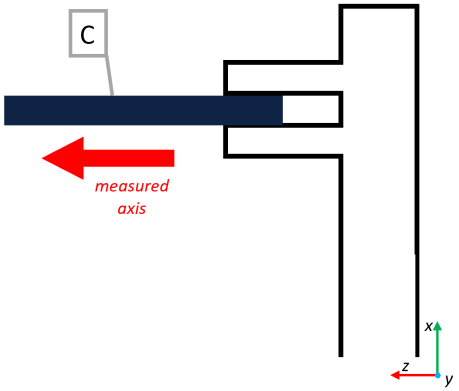


Figure 5.13: A Schematic of the accelerometer mounting on the CPU-fan. The fan spins around the z-axis. The accelerometer, labeled C, is measuring in z-axis

Two small neodymium magnets are used as wights to simulate a perturbation in the rotating system. The magnets are placed on opposite side of one of the fan blades such the the magnetic force fixates them in place. The extra weight causes an in-balance. A photo of the weight placement can be seen in Appendix

5.9 Implementation

The experiment arrangement is setup such that the measured axis is perpendicular to the gravitational force to the greatest possible extent. The shaker is put flat on a table and the position is finely adjusted with thin objects such that the accelerometer is measuring around zero in idle state. The reason for doing this is to get equal data space in the negative and positive direction. During a measurement series, the serial data is saved to a text file on the receiving computer which is done with a feature of the PuTTY software. The text file is read in Matlab with the *readtable()* function which creates a table variable. The data is transformed to a vector with the *table2array()* function. The sampling frequency is calculated each measuring sequence. This is done because the frequency differs each session. A reason for this could be the differing temperature in the room. The temperature could particularly affect the RC-oscillator in the ATmega328 and possibly the data-rate of the ADXL345. The sampling frequency is calculated with

$$f_s = \frac{Elements}{Samplingtime},$$

where *Elements* is the amount of data points, which could be read in Matlab and *Sampling time* is a priori known since the shaker output time is defined.

The one-sided frequency spectrum is produced with fast Fouier transform which is done with the Matlab standard function *fft()*. The Matlab code used is seen in the listing bellow.

```
sampling_time= length(B)/3200
%%
acc_data=B;
t = linspace(0,sampling_time,length(acc_data));         % Time (s)
L = length(t);
Ts = t(2)-t(1);                                         % Sampling Interval (sec)
Fs = 1/Ts;                                              % Sampling Frequency (Hz)
Fn = Fs/2;                                              % Nyquist Frequency (Hz)

FTa = fft(acc_data)/L;                                  % Fourier Transform (Scaled)
Fv = linspace(0, 1, fix(L/2)+1)*Fn;                     % Frequency Vector
Iv = 1:length(Fv);                                      % Index Vector
plot(Fv, abs(FTa(Iv))*2)
```

6 Results and Discussion

The C-code and wiring schematics can be found in the Appendix.

A 20 second frequency sweep from 100Hz to 1600Hz. The frequency increases linearly. The accelerometer is set in $\pm 16g$ mode in 13-bit resolution. Thus the amplitude corresponds to scaling factor of 3.9mG/LSB. The result is seen in Figure 6.1, where the number of elements captured is 65748. The sampling frquency is calculated to 3 280Hz. Two hills are seen around 220Hz and 1180Hz, which is assumed to be resonance frequencies of the mounting device. This assumption is based of noise which were present under these frequencies. The peak-value is around 70 which multiplied with the scaling factor yields a peak acceleration of almost 0.280g. The volume of the Shaker was turned down such that the peak around 220Hz did not overshoot the max value. Small volume adjustments greatly impacted the amplitude around 220Hz.

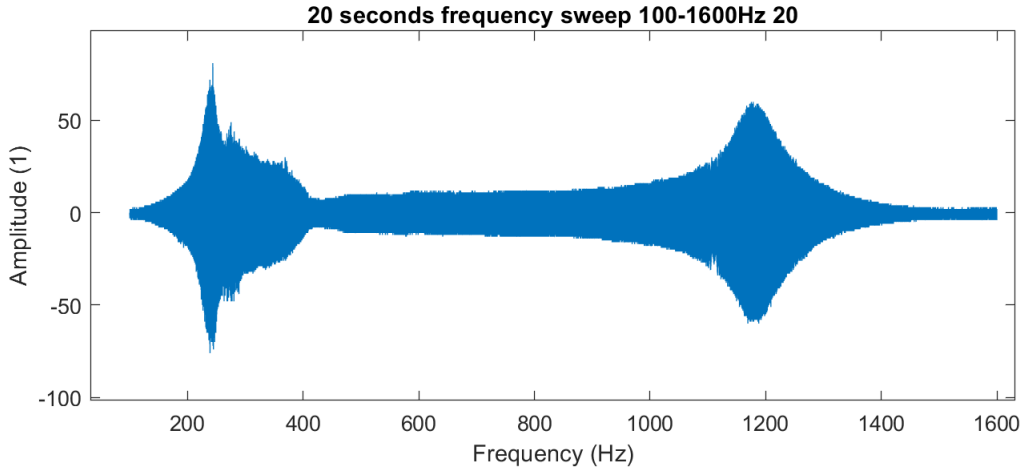


Figure 6.1: Frequency response of the accelerometer and the mounting piece using the Shaker.

In Figure 6.2 shows the frequency spectrum of two series of CPU-fan measurements. The top plot is the result of a 300 second long sampling period which amounts to almost one million data points. The bottom plot is the results of adding two magnets as wights to one of the fan blades of the CPU-fan.

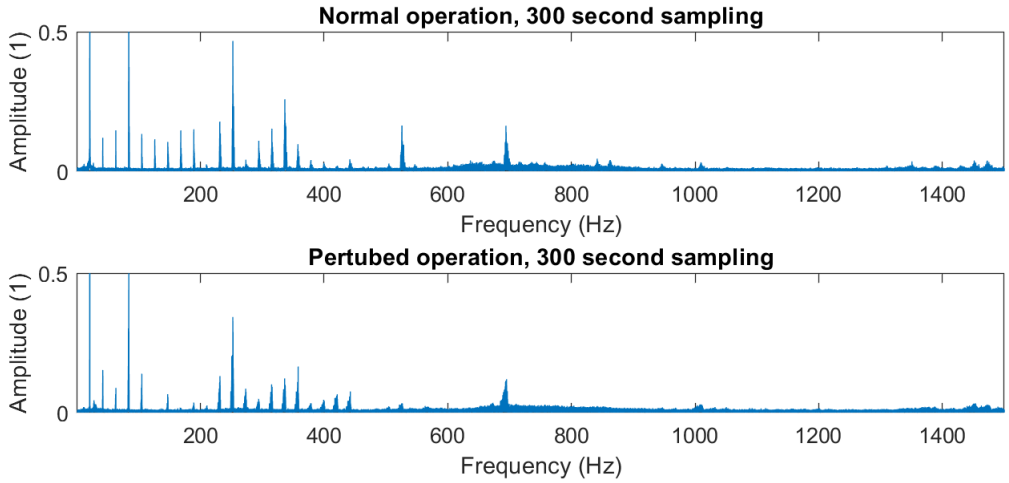


Figure 6.2: Two frquency spectrum tests of the CPU-fan. Top plot is normal fan operation. Bottom plot is with two weights on one fan balde of the CPU-fan.

It can bee seen that more frequencies is added and removed when adding the perturbation. The contribution at around 550Hz is dampened. New frequencies can be seen around 275Hz and 420Hz. The new frequencies can be observed in Figure 6.3 which is the same results as shown in Figure 6.2 but zoomed in and the two series are imposed on top of each other.

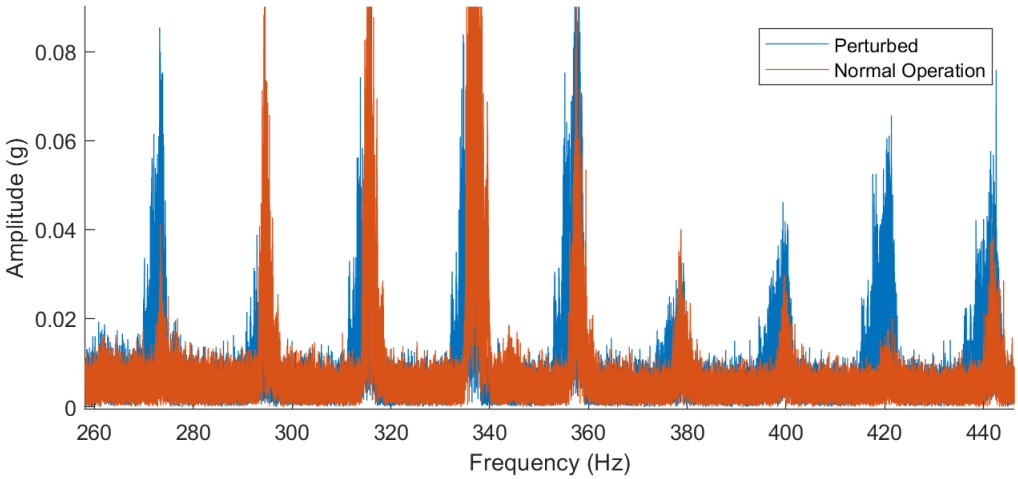


Figure 6.3: One-sided frequency spectrum of two different measurment series. The red plot is normal operations. The blue plot is perturbed operations.

7 Conclusions

In this project a vibration measuring device have been built with an ATmega328 and a ADXL345. The basic principle of the ADXL345 has been outlined for better understanding of its operation. Furthermore, the concept and importance of resonance has been investigated by solving the basic equation of motion for a discretized model of the accelerometer.

In order to communicate between hardware the SPI and UART protocol was outlined in detail. The time requirement and speed for data transfers was outlined in order to meet the specified max sampling speed of the ADXL345.

The CPU-fan tests shows that the system have the capability preforming in application. The goals of this project has been met. However, a better solution for mounting device is needed which presumably would mitigate the divergence behaviour around 220Hz.

A further improvement to the system is to use an external crystal oscillator and apply an overflow interrupt to get a consistent sampling speed.

References

[1] H. Jiangbo, W. Zhou, Y. Hujun and P. Peng, *"Structural Designing of a MEMS Capacitive Accelerometer for Low Temperature Coefficient and High Linearity"*, MDPI, Basel, Switzerland, February 2018.

[2] Sensor Terminology, NATIONAL INSTRUMENTS CORP, 2020 https://www.ni.com/sv-se/innovations/white-papers/13/sensor-terminology.html#section-958191954 URL-date: 03-12-20

[3] http://www.new-adin.com/en/Products/menu_10.html URL-date: 03-12-20

[4] S. Hanly *"Shock & Vibration Overview"*, enDAQ Midé Technology

[5] J.L Meriam, L.G. Kraige *"Engineering Mechanics Dynamis"*, 2nd ed., Vol. 2. Virginia Polytechnic Institute and State University,

[6] Atmel Studio 7.0, Informer Technologies, Inc. https://atmel-studio.software.informer.com/ URL-date: 09-12-20

8 Appendix

8.1 Wiring Diagrams

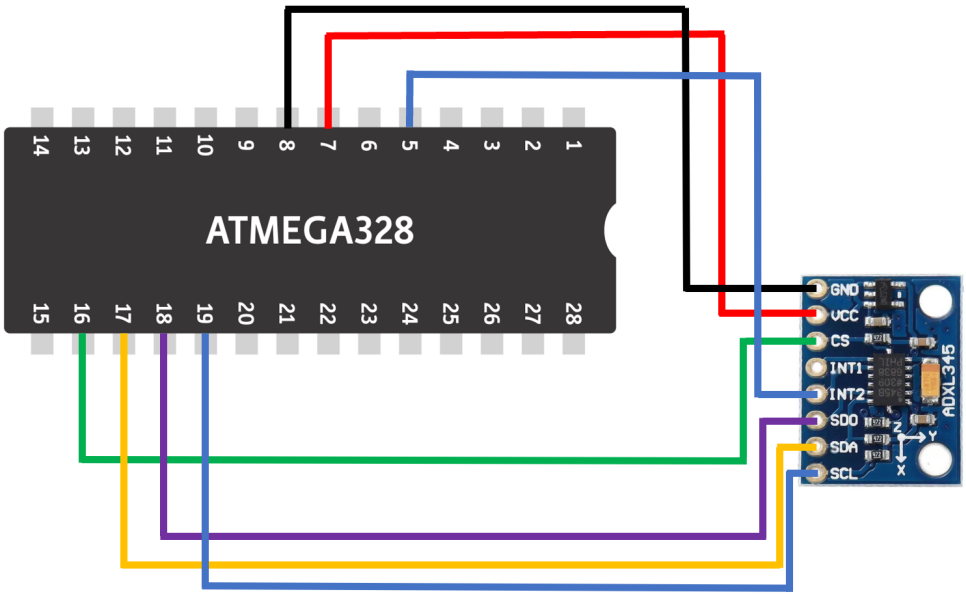


Figure 8.1: A wiring diagram of the ADXL345.

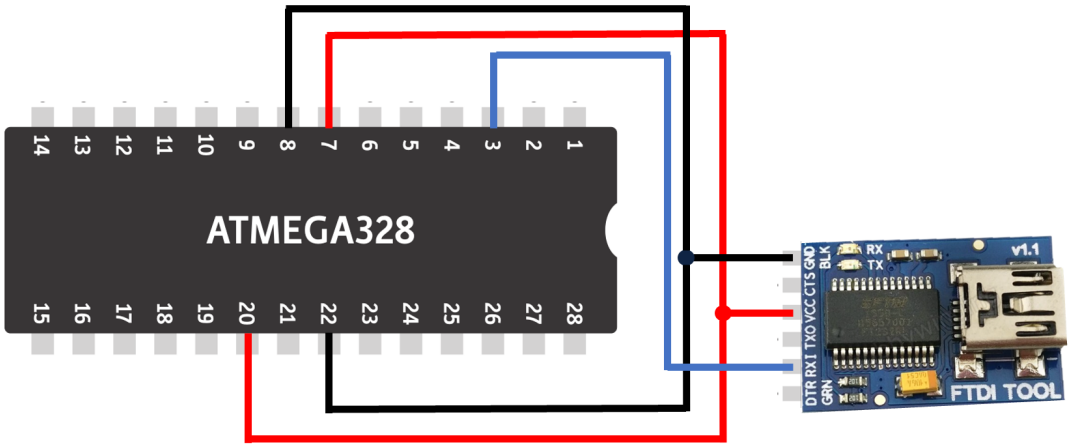


Figure 8.2: A wiring diagram of the FT232R.

8.2 Photos

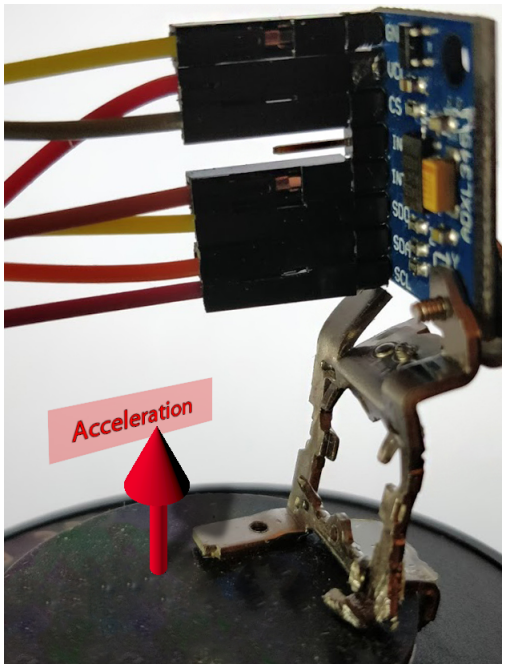


Figure 8.3: Serial frame of UART.

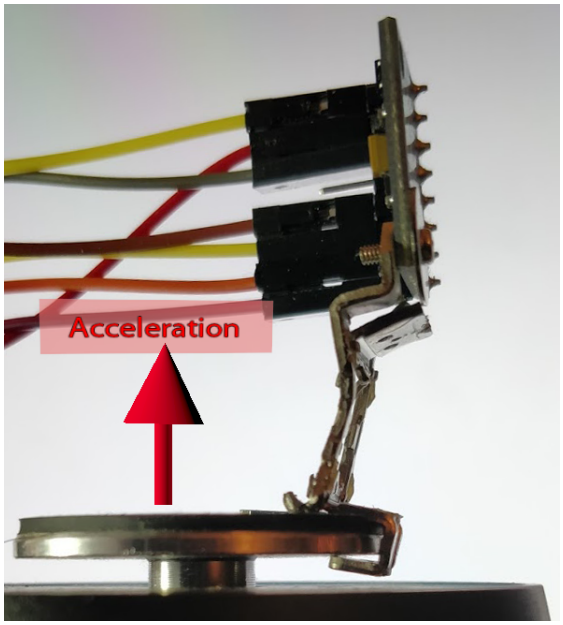


Figure 8.4: Serial frame of UART.

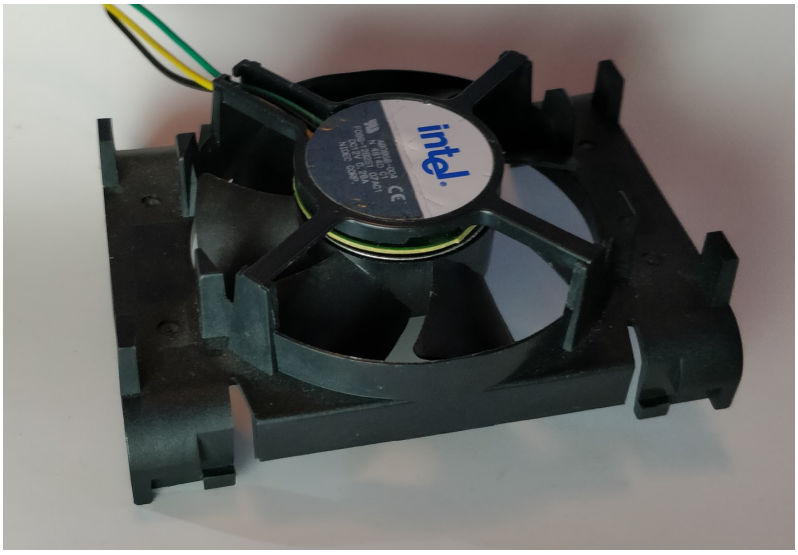


Figure 8.5: Photo of CPU-fan.

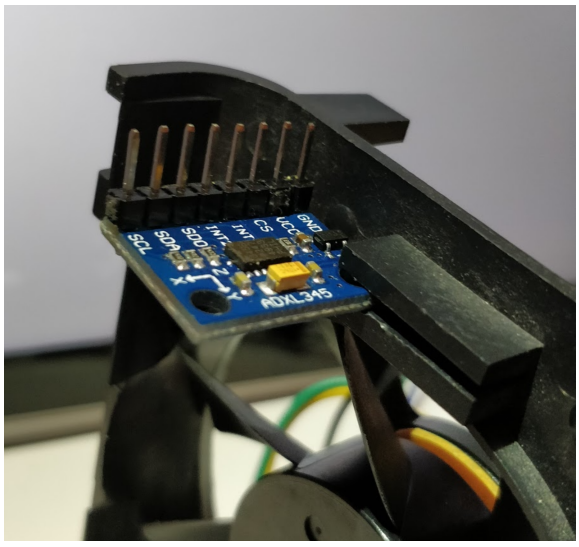


Figure 8.6: Photo of accelerometer mounted in the CPU-fan.

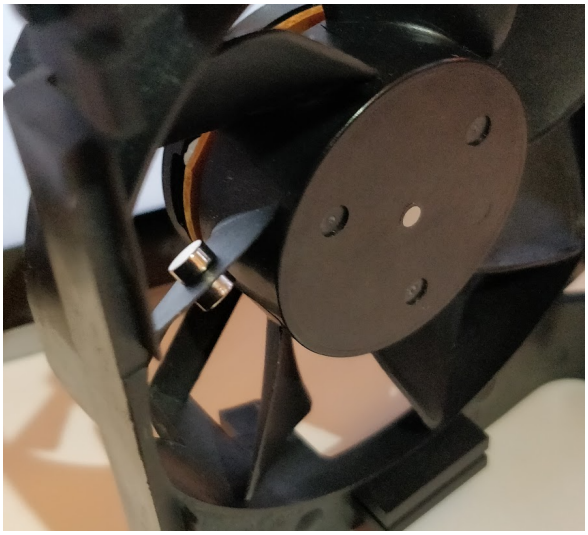


Figure 8.7: Photo of magnets on CPU-fan blade.

8.3 C-code

```
/*
 * accel.c
 *
 * Created: 2020-11-03 12:29:04
 * Author : patrik
 */
#define F_CPU 8000000UL
#include <avr/io.h>
#include <stdio.h>
#include <stdlib.h>
#include <avr/interrupt.h>
#include <util/delay.h>

#define SPI_DDR DDRB
#define CS      PINB2
#define MOSI    PINB3
#define MISO    PINB4
#define SCK     PINB5

// READ 11, WRITE 01
#define BR 0b00101100 //                          0x2c =0b101100
#define Data_Format 0b01110001 //                  0x31=0b110001
#define Power_Register 0b00101101 //               0x2d =0b101101
#define X_Axis_Register_DATAX0 0b11110010 //     0x32= 0b110010
#define X_Axis_Register_DATAX1 0b11110011 //     0x33=0b110011
#define X_Off 0b0111110 //                         0x1E =0b11110
#define INT_MAP 0b01101111 //                      0x2f =0b101111
#define INT_ENABLE 0b01101110 //                   0x2E =0b101110
```

```
#define INT_SOURCE 0b10110000 //                0x30 =0b110000 read only

#define BAUD_RATE_S 1 //

volatile uint8_t x0; //LSBy
volatile uint8_t x1; //MSBy
volatile uint8_t r_data;
volatile int16_t x_out;
volatile uint8_t i =0;
volatile uint8_t r_data;

uint8_t SPI_masterTxRx(uint8_t data)
{
    // transmit data
    SPDR = data;
    // Wait for reception complete
    while(!(SPSR & (1 << SPIF)));
    // return Data Register
    return SPDR;
}

ISR (INT1_vect){
    PORTB |= (1<<PB0); //LED ON
    PORTB &= ~(1 << CS);
    SPI_masterTxRx(X_Axis_Register_DATAX0); //multiple read
    x0 = SPI_masterTxRx(0xFF);
    x1 = SPI_masterTxRx(0xFF);
    PORTB |= (1 << CS);
    x_out = (x1 << 8) | x0;

    char data[8];
    //int length = snprintf( NULL, 0, "%d", x_out );
    //char* data  = malloc( length + 1 );
    snprintf( data, 8, "%d", x_out );

    i = 0;
    while(data[i] != 0){
        while (!( UCSR0A & (1<<UDRE0))); // Wait for empty transmit buffer*/
        UDR0 = data[i];
        i++;
    }
    //free(data);
    while (!( UCSR0A & (1<<UDRE0)));   /* Wait for empty transmit buffer       */
        UDR0 = '\n';

    while (!( UCSR0A & (1<<UDRE0)));   /* Wait for empty transmit buffer       */
        UDR0 = '\r';

    PORTB &= ~(1<<PB0); //LED OFF
}

int main(void)
{
```

```
//USART SETTINGS
UCSR0B |= (1<<TXEN0)|(1<<RXEN0); //enable,, transmission & reciever
UCSR0C |=(1<<UCSZ01) | (1<<UCSZ00); //set 8 bit transmissions
UBRR0H = (BAUD_RATE_S>>1);
UBRR0L = (BAUD_RATE_S);

DDRB |= (1 << PB0); //Set led pin as output
DDRD &= ~(1 << PD3); //D3 as input
PORTD &= (0 << PD3);
_delay_ms(500);

PRR &= ~(1 << PRSPI);   //enable clock to SPI module

// enable SPI, set as master, and clock to fosc/128
SPCR |= (1 << SPE) | (1 << MSTR) | (1 << CPOL) |(1<<CPHA);
SPSR |=(1<<SPI2X);
SPCR &= ~(1<<DORD) | ~(1<<SPR1) | ~(1 << SPR0);

// set CS, MOSI and SCK to output
SPI_DDR |= (1 << CS) | (1 << MOSI) | (1 << SCK);

//SPI_masterTransmit(Power_Register);
_delay_ms(10);

//Write Data_Format
PORTB &= ~(1 << CS);
SPI_masterTxRx(Data_Format);
SPI_masterTxRx(0b00001011);
PORTB |= (1 << CS);
_delay_ms(100);

//Set up int enable on accel
PORTB &= ~(1 << CS);
SPI_masterTxRx(INT_ENABLE);
SPI_masterTxRx(0b10000000);
PORTB |= (1 << CS);
_delay_ms(100);

PORTB &= ~(1 << CS);
SPI_masterTxRx(INT_MAP);
SPI_masterTxRx(0b10000000); //enable on int2 pin
PORTB |= (1 << CS);
_delay_ms(100);

PORTB &= ~(1 << CS);
SPI_masterTxRx(Power_Register);
SPI_masterTxRx(0x08);
PORTB |= (1 << CS);
_delay_ms(100);

PORTB &= ~(1 << CS);;
SPI_masterTxRx(BR);
SPI_masterTxRx(0b00001111);
```

```
    PORTB |= (1 << CS);
    _delay_ms(100);

    EICRA |= (1<<ISC11) | (1<<ISC10); //set rising edge int on int1
    EIMSK |= (1<<INT1); //enable interrupt on int1

    PORTB &= ~(1 << CS);
    SPI_masterTxRx(INT_SOURCE);
    SPI_masterTxRx(0xFF);
    PORTB |= (1 << CS);

    sei();
    PORTB &= ~(1 << CS);
    //SPI_DDR &= ~(1 << CS);
    SPI_masterTxRx(X_Axis_Register_DATAX0);
    SPI_masterTxRx(0xFF);
    SPI_masterTxRx(0xFF);
    PORTB |= (1 << CS);

    while(1){
    PORTB |= (1<<PB0); //LED ON
    _delay_ms(1000);
    PORTB &= ~(1<<PB0); //LED OFF
    _delay_ms(1000);
    }
}
```