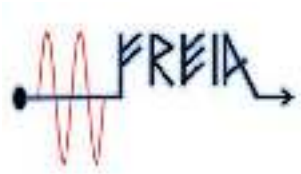
UPPSALA
UNIVERSITET



FREIA Report 2015/04
June 9, 2015

Department of Physics and Astronomy
Uppsala University

Measuring mechanical vibrations using an Arduino as a slave I/O to an EPICS control system

Adam Hjort & Måns Holmberg

Uppsala University, Uppsala, Sweden

Department of
Physics and Astronomy
Uppsala University
Box 516
SE-75120 Uppsala
Sweden

Measuring mechanical vibrations using an Arduino as a slave I/O to an EPICS control system

Adam Hjort & Måns Holmberg

Supervisor: Volker Ziemann & Konrad Gajewski

Department of Physics and Astronomy
Uppsala Universitet

Abstract

In this study we have assembled hardware and software to be used for measuring of mechanical vibrations in the FREIA-laboratory at Uppsala University. We have utilized an Arduino microcontroller as a slave I/O and equipped it with dual accelerometers to be used for vibration measurements and a serial adapter which was used to connect the hardware to an EPICS IOC for analysis. Data from the two accelerometers have then been cross correlated in order to find a transfer function. Our results where in good agreement with theory.

1 Introduction

It is of utmost importance when designing physical experiments that one takes into account the mechanical vibrations that may occur and affect the results. There are several ways to measure mechanical vibrations and in this study we look closer on how to measure them using an MEMS-based accelerometer [3]. By using two accelerometers we can see how vibrations transfers from one point to another and thereby gain some information into the characteristics of the medium the vibrations propagated through. To provide the accelerometers with power and collect the waveforms, an Arduino microcontroller is being used. The Arduino functions as a slave IO and can be connected to either MATLAB or an EPICS control system. During hardware testing a speaker was used to generate desired sine waves, seen in figure 6. Since this speaker propagates the sound directly into the material it is being placed on, it proved excellent as a frequency test device for the accelerometers. As an real world experiment, we measured the transfer function for a vacuum pump at the FREIA-laboratory.

2 Hardware

Figure 1: The Arduino Uno rev. 3

2.1 Arduino

Arduino is an open source microcontroller that has become very popular amongst students, hobbyists as well as with professionals. It has a very active community and the low cost of purchase makes it an excellent tool to quickly test and deploy ideas. We have chosen to work with the reference model Arduino Uno rev. 3 that can be seen in figure 1. It measures 68.6 x 53.4 mm and weights 25 g. It is based on the ATmega328 8-bit microcontroller. It has a

clock frequency of 16 MHz and a 32 KB flash memory. It operates at 5 V and can be powered over USB or an external power supply. On the board there are 6 analog pins and a total of 14 digital I/O Pins and 6 of these provide PWM output. There are also pins for power management [7].

The analog pins which are the ones we mostly work with in this project has a resolution of 10 bits meaning they can handle 1024 different values. This is usually done by having each value correspond to a voltage between ground and 5 V, however this can be changed by using the AREF pin (analog reference pin).

For communication with the Arduino UART TTL (5V) serial communication is used. The digital pins 0 (RX) and 1 (TX) can also be used to send and receive serial data. In the IDE there is a built-in serial monitor that can be used to send and receive information. When connected over USB to a computer the Arduino shows up as a virtual COM-port and any software capable of serial communication can be used.

2.2 Accelerometer

To measure frequencies we use an accelerometer similar to the ones you find in smartphones. The accelerometer measures as the name hints the acceleration that it being is subjected to. The model we have used is mounted on a breakout board from SparkFun and use the ADXL335 3-axis accelerometer from Analog Devices [1]. It measures $\pm 3\,g$ in three orthogonal axis labeled the X, Y and Z direction. It can read in the range of 0.5 Hz to 1600 Hz for the X and Y axis while the Z axis has a range of 0.5 Hz to 550 Hz. However the SparkFun model comes mounted with 0.1 μF capacitors that acts as a low-pass filter and limits the lower bandwidth of each axis to 50Hz.

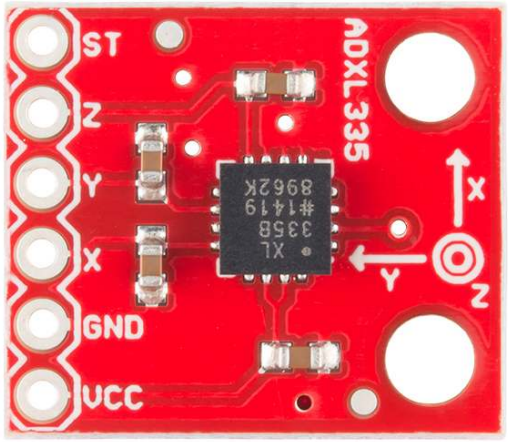


Figure 2: The accelerometer breakout board with the ADXL335

To operate the accelerometer it needs between 1.8 V to 3.6 V so we can't use the 5 V output on the Arduino and have to use the 3 V. This also means that simply plugging one of the axis into one of the Arduinos analog input pins will lead to complications since it expects a maximum value to be 5 V. To solve this we connect the supply voltage to the AREF pin on the Arduino as well as to the accelerometer and in the software tell the Arduino to use this voltage as a reference instead of the default 5 V.

The ADXL335 is a so called MEMS (Micro-Electro Mechanical System) accelerometer. The sensor in the ADXL335 is a polysilicon surface-micromachined sensor that is built on a silicon wafer. In the sensor there is a proof mass called a seismic mass that is tethered to deflectable plates. When subjected to acceleration the plates are deflected by the mass and this deflection is measured by a differential capacitor. The differential capacitor is made of independently fixed plates and the plates that are connected to the seismic mass. The fixed plates are driven by 180° out of phase square waves and when the plates are deflected the differential capacitor gets unbalanced and gives an output signal of a square wave whose amplitude is proportional to the acceleration.

By using demodulation techniques that are sensitive to the phase-magnitude and direction of the acceleration can be determined. The signal is then amplified and taken through a 32 kΩ resistor and now one signal for each axis is available. Each signal is then taken through a 0.1 μF capacitor that as was mentioned earlier acts as a low-pass filter. The ADXL335 uses one structure for the X, Y and Z axis which gives the axis high orthogonality that in turn leads to little cross-axis sensitivity [9] [2].

2.3 Serial adapter

To be able to integrate the Arduino into the EPICS environment used at FREIA it needs to be connected to a serial device server using a D-sub 9 connector. The serial device server in turn gives the device an IP adress and makes it accessible over the network. The Arduinos digital pin 0 and 1 are by default used as RX (receive) and TX (transmit) but this does not mean that one can simply attach D-sub 9 connector and get a working connection. The reason for this is that the Arduino communicates with the UART (Universal asynchronous receiver/transmitter) protocol that sends data with TTL (Transistor–transistor logic) voltage levels that are in the interval of 0 V to 5 V while the serial device server uses RS-232 that uses -15 V to -3 V for 0 and 3 V to 15 V for 1. communication. The data sent from UART can however be converted to work with RS232 devices by feeding the signal through an integrated circuit named MAX232. The MAX232 is a dual driver/receiver and works by changing the outgoing voltage to be in the RS232 compatible interval of approximatively $\pm$ 7.5 V and the incoming voltage is reduced to be between 0 V to 5 V [4].

For a complete view of all hardware used and how to connect it please see figure 11 in Appendix 7.1.

3 Software

3.1 Arduino

The Arduino microcontroller is programed using the Arduino language, which is based on C/C++, and comes with a user-friendly integrated development environment (IDE) [8]. The user only needs to define two functions, to make an executable program: a setup() and loop() function. The setup() function is only executed once, and is used to initialize variables, pin modes etc. The loop() function is essentially a infinite loop that is called repeatedly until the device is turned off, this is where your code is implemented. These types of programs are called cyclic executive programs.

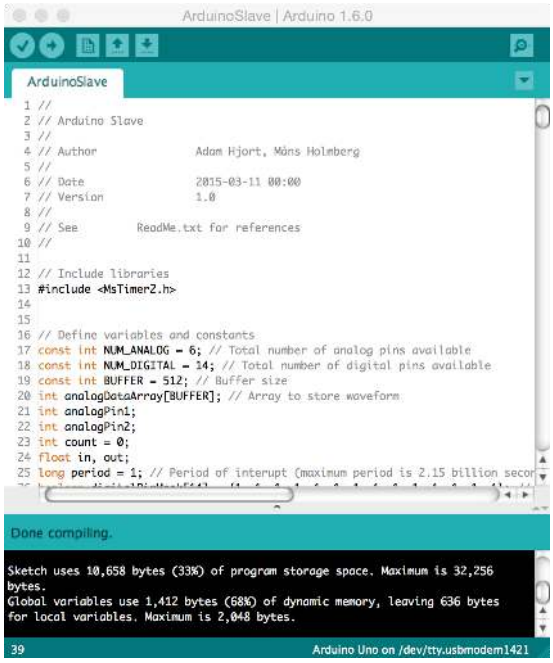


Figure 3: The Arduino IDE.

While one can create a wide variety of programs using only these two functions, the loop() function is not ideal for precision high speed applications. This is because it runs continuously, without the use of a timer [8]. Instead, we will be using interrupts, to allow for predictable timing, which is essential to high speed data collection. The interrupts are implemented using the library MsTimer2 [15]. The

library MsTimer2 combines both ease of use and good time resolution (1 ms). The events that triggers the interrupts are internal timer overflows. Each time a timer overflow, a chosen function is called and executed, in our case this will be a function that reads an analog pin (or two simultaneously). The maximum frequency of the interrupts is 1 kHz and is determined by the time resolution of MsTimer2, and hence limits the speed at which we can sample data.

Listing 1: Reading analog waveform

```
// Reads 512 values of analog pin 0 every
// 1 ms using MsTimer2
#include <MsTimer2.h>

const int buffer = 512; // Buffer size
int analogDataArray[buffer];
int count = 0;
int analogPin = 0;
int period = 1; // Period

void getWaveform() {
   analogDataArray[count] =
       analogRead(analogPin);
   count++;
   if (count >= buffer)
   {
    MsTimer2::stop();
    count = 0;
   }
}

void setup() {
   analogReference(EXTERNAL);
   MsTimer2::set(period, getWaveform);
   MsTimer2::start();
}

void loop() {
}
```

To read the voltage from an analog pin onboard the Arduino, we use the function analogRead(). The analog to digital converter (ADC) will turn the voltage into an digital signal, ranging from 0-1023, where the reference voltage (value 1023) is set by the function analogReference(). Because the ADXL335 accelerometer operates using 3.3 V, the function analogReference() will be set to EXTERNAL, which indicates that an reference voltage will be applied to the AREF pin. The time used to read an analog input using analogRead() is about 100 μs, therefore it does not limit the frequency of which we can sample [8].

In order to establish serial communication between the Arduino slave and EPICS, we need call the begin() method of the class Serial. The argument of begin() is the baud rate of the communication, which will be set to 115200 Bd. Data will be sent and received as human-readable ASCII text, with the methods print() and read(). Listing 2 illustrates the simple code needed to establish serial communication.

Listing 2: Exemple showing serial communication

```
void setup() {
   Serial.begin(115200);
   }
void loop() {
   Serial.println("Hello World");
}
```

To program a useful Arduino slave IO that will be able to respond and perform tasks upon different commands sent by the EPICS control system, we will use a switch statement. First we have to read the command sent by EPICS. This can by done by scanning the incoming characters until the terminator character is reached, which we have set to newline, shown in Listing 3.

Listing 3: Switch statement, reading serial input

```
String input;

void setup() {
   Serial.begin(115200);
   }

void loop() {
     while (Serial.available() > 0)
  {
    char lastRecived = Serial.read();
    input += lastRecived;
    if (lastRecived == '\n')
   {
```

```
      switch (input[0]) {
        case 'W':
          MsTimer2::set(period,
              getWaveform);
           MsTimer2::start();
          break;
          default:
          Serial.println("Error");
          break;
      }
      input = ""; // Clear recieved buffer.
    }
  }
}
```

We use a buffer of 512 elements (limited by the memory of the Arduino) to temporarily store the waveforms on the Arduino, before the data is sent to EPICS or MATLAB [7]. This method was not limited by the time delay introduced by constantly sending a command and receiving one data point at a time, which was our first approach.

Appart from acquiring analog values, we also implemented digital I/O. The represetive commands are detailed in the appendix 7.2.

3.2 MATLAB

MATLAB is a numerical computing environment that is built around an easy scripting language, which makes MATLAB perfect for quick testing and data analysis. First, we initialized the serial communication between MATLAB and the Arduino slave, with the native function `serial()`. Using this function, we created a serial object and set the parameters `DataBits = 8`, `StopBits = 1`, `BaudRate = 115200`.

```
serialObj = serial(comPort);
set(serialObj, 'DataBits', 8);
set(serialObj, 'StopBits', 1);
set(serialObj, 'BaudRate', 115200);
```

Then we implemented our protocol, see Appendix 7.2, into different functions that handled the serial communication. Below is the function `readWaveform()` that reads a waveform from an analog channel on the Arduino.

```
function waveform = ...
    readWaveform(serialObj, analogPin)

output = ['W', num2str(analogPin), '?'];
fprintf(serialObj, output);
input = strsplit(fscanf(serialObj,...
 '%c'), ' ');

if strcmp(output(1:end-1),...
    cell2mat(input(1)))
    waveform = str2double(input(2:end));
else
    error('Error');
end
end
```

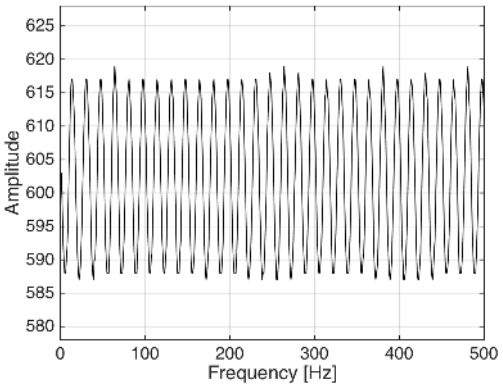


Figure 4: Waveform of a 60 Hz signal.

With only this simple code we are now able to perform tests and evaluate the performance of the ADXL335 accelerometer. Figure 4 shows the raw waveform obtained by the accelerometer placed near the Adin tone generator playing a 60 Hz sine wave. The sine wave was generated using onlinetonegenerator.com [14]. The collected waveform is then transformed from the time domain into the frequency domain, using fast fourier transform (FFT) [10], which is shown in figure 5. The 60 Hz signal is clearly distinguished from the background noise.

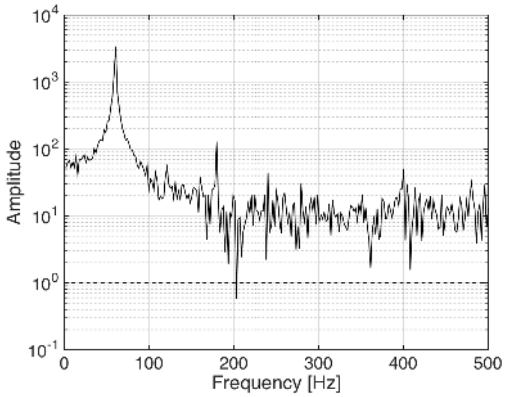


Figure 5: Frequency domain of the waveform in figure 4. Note the logarithmic scale.

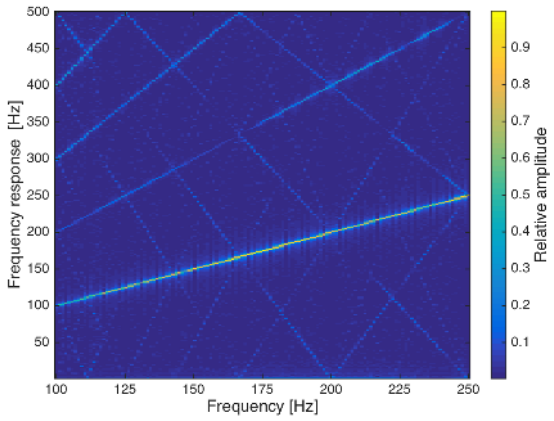


Figure 7: The frequency response of a 100 Hz to 250 Hz chirp waveform. The colormap represents the relative amplitude.

To test for the frequency response of the ADXL355, we generated a linear chirp from 100 Hz to 250 Hz with the Adin tone generator and collected the data over a period of 15 min [11]. The response is illustrated in figure 7. All lines with a positive slope is harmonics or the fundamental. The other lines with negative slope are Nyquist reflections.

Figure 6: The Adin KKBT speaker which has been used as a tone generator during the experiments.

3.3 EPICS

EPICS (Experimental Physics and Industrial Control System) is an open source software environment for development and management of control systems used globally in small and large scale projects [6]. EPICS is available for Windows and Linux and in this project we used Scientific Linux 6 as operating system. The EPICS version used is the standalone version CODAC Core System v4.1.0 that is distributed by the ITER Organization. EPICS utilizes Client/Server and Publish/Subscribe techniques to handle communications. In an EPICS environment a server is called Input/Output Controller and is abbreviated IOC. To an IOC multiple sensors and modules can be attached for measuring and controlling the system. Through the Channel Access (CA) network protocol other computers can interact with the IOCs and read data and send commands to them. EPICS is very scalable and a system can consist of a single IOC for small projects to thousands of IOCs for more massive projects [6].

On an IOC a protocol file is stored that tells how the communication with an attached device should be handle. This is done by defining commands in the file that tells EPICS what data to send and what to expect in return. There

is also a database file, where the records are defined. A record tells what commands will be available through the CA network. There are many different ways records and protocols can look like and we will show an example of how these can be structured to work with each other. The example shows how we measure a single analog reading from an Arduino that has been loaded with our serial protocol. When requesting a single analog read from pin 0 the user should send `A0?` to the Arduino. If the value on the pin at the moment is 496 the output will be `A0 496`. We will not cover how to install EPICS and create an application since there is plenty of guides already available online on that topic [5].

There are two files that needs to be configured before we can start making our record. In the file `userPreDriverConf.cmd` we configure how to connect to the Arduino which we have named to be `ARD0`:

```
drvAsynIPPortConfigure("ARD0",
    "192.168.10.9:4003")
```

We connect using an IP-adress but other methods are also possible. All the settings for the baud rate, stop bits and so on are handled by the serial switch so this is not something that needs to be set in this case but for other methods this can be configured here. In the file `dbToLoad.cmd` we specify what database to load when the IOC starts and what name that should be assigned to the variables `PREFIX` and `ARD_PORT`:

```
dbLoadRecords("strdev.db",
    "PREFIX=STRDEV,ARD_PORT=ARD0")
```

In the database file `strdev.db` we have a record that looks like this:

```
record(ai, "$(PREFIX):A0") {
    field(DTYP, "stream")
    field(INP, "@accel.proto get_analog(0)
        $(ARD_PORT)")
    field(SCAN, ".5 second")
}
```

In the first row we state that we want to create a record by writing `record`, in the parentheses that follows afterward we set what type of record we will be using. In our example it says `ai` meaning it's an analog input record. After that we see `"$(PREFIX):A0"` and this is the name of the record that will be used on the CA network. This is the most conventional way of naming records - `"NameOfDevice:Sensor"`. Next we see three rows of fields, a field is where the settings for the record is made. The first row says `DTYP` (which means device type field) and this sets what kind of device the record is going to be used with. In our example `"stream"` means that we will be using *StreamDevice* which is a device support module for EPICS that facilitates the use of devices that communicates using strings [16] [12]. The middle row says `INP` which stands for input link, as was said earlier this is an *analog input* record so this is where we specify where the input will come from. The line `"@accel.proto get_analog(0) $(ARD_PORT)")` specifies what protocol files to use, which in this case is `@accel.proto` and `get_analog` what commands to run in the protocol. The `(0)` is a variable that can be send along to the protocol. This means we can use the same the command in the protocol file for different inputs. The last field says `(SCAN, ".5 second")` and tells how and when a record processes which in this case is set to fetch a new value every 0.5 second. The command being called by the record in the protocol file is looks like this:

```
get_analog {
 out "A\$1?";
 in "A\$1 %d";
}
```

The first row says `get_analog` and is the name of the command that the records use to call it. Next we see `out "A\$1?"` and specifies what command will be sent to the Arduino. The `\$1` will be replaced with the variable that was sent along from the record which means that the command that is sent in this case will be `A0?`. The last row `in "A$1 %d"` tells EPICS what to

expect back from the Arduino. The `A$1` will of course once again translate into `A0` and the `%d` means that there will come a signed decimal afterwards representing the value of the analog pin.

Once the protocol and database are prepared there are several ways to interact with and read data from the IOC. The most common way is to create a GUI that displays the information but the easiest way to see that the application works is to use `caget` in the terminal. The following code shows how to start the application called `accel` and read data from the record that we created in the example above:

```
[User@localhost ~]$
    ./target/main/scripts/accel-ioc start
Starting IOC accel        [  OK  ]
[User@localhost ~]$ caget STRDEV:A0
STRDEV:A0                   496
```

4 Results

We have shown that our system, hardware and software, is working as predicted when subjected to tests with known outcomes. The next step was to test it on an unknown source, which we chose to be a running vacuum pump from Scrollvac. The pump was connected to the cryostat in the FREIA-laboratory and was bolted to the concrete floor. To measure the transmitted frequencies from the pump to the floor, we placed one accelerometer directly to the pump and the other on the nearby floor, illustrated in figure 8. Using MATLAB and the correlation command, see appendix 7.2, on the Arduino, we recorded data from both accelerometers simultaneously when the pump was running. The frequency domain of the waveforms are presented in figure 9. The transfer function is created by dividing the transformed response waveform by the transformed source waveform. Figure 10 shows the transfer function. All frequencies below a value of one are affected while the frequencies above one are amplified.

Figure 8: One accelerometer mounted with red tape on a vacuum pump and one with red tape on a metal cube on the floor

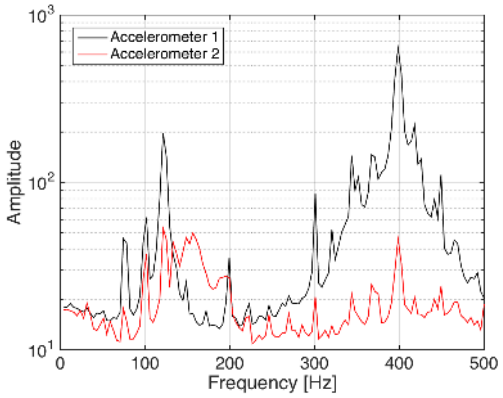


Figure 9: Data of vibrating vacuum pump from two accelerometers shown in the frequency domain. One is located at the base of the pump and the other is placed directly on the pump. Note the logarithmic scale.

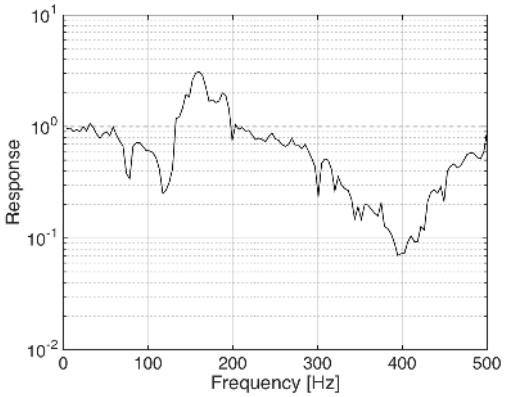


Figure 10: The transfer function of the data obtained in figure 9.

5 Conclusion

We can conclude from our results that it is possibly to perform reliable data acquisition and analysis using low-cost and easily available hardware and software.

6 Bibliography

[1] SparkFun Electronics. *SparkFun Triple Axis Accelerometer Breakout - ADXL335*. https://www.sparkfun.com/products/9269, 2015. [Accessed 22 May 2015].

[2] Analog Devices Inc. *ADXL335*. http://www.analog.com/en/products/mems/mems-accelerometers/adxl335.html, 2015. [Accessed 22 May 2015].

[3] Commtest Instruments. *How is Vibration Measured?* http://reliabilityweb.com/index.php/articles/how_is_vibration_measured/, 2006. [Accessed 22 May 2015].

[4] Texas Instruments. *MAX232*. http://www.ti.com/product/MAX232/description, 2015. [Accessed 22 May 2015].

[5] Pete Jemian. *Constant Lighting with EPICS*. http://prjemian.github.io/cmd_response/epics/index.html, 2014. [Accessed 22 May 2015].

[6] Argonne National Laboratory. *Experimental Physics and Industrial Control System*. http://se.mathworks.com/help/matlab/math/fast-fourier-transform-fft.html, 2015. [Accessed 22 May 2015].

[7] Arduino LLC. *Arduino Uno*. http://www.arduino.cc/en/Main/ArduinoBoardUno, 2015. [Accessed 22 May 2015].

[8] Arduino LLC. *Language Reference*. http://www.arduino.cc/en/Reference/HomePage, 2015. [Accessed 22 May 2015].

[9] Sergey Edward Lyshevski. *Nano- and Micro-Electromechanical Systems: Fundamentals of Nano- and Microengineering*. CRC Press, 2005.

[10] MathWorks. *Fast Fourier Transform (FFT)*. http://se.mathworks.com/help/matlab/math/fast-fourier-transform-fft.html, 2015. [Accessed 22 May 2015].

[11] MathWorks. *Spectrogram*. http://se.mathworks.com/help/signal/ref/spectrogram.html, 2015. [Accessed 22 May 2015].

[12] W. Eric Norum. *How to use StreamDevice and ASYN to create EPICS device support for a simple serial, GPIB, or network attached device*. http://www.aps.anl.gov/epics/modules/soft/asyn/R4-24/HowToDoSerial/HowToDoSerial_StreamDevice.html. [Accessed 22 May 2015].

[13] Friends of Fritzing foundation. *Fritzing*. http://fritzing.org/home/, 2015. [Accessed 22 May 2015].

[14] onlinetonegenerator.com. *Online Tone Generator*. http://onlinetonegenerator.com, 2015. [Accessed 22 May 2015].

[15] Javier Valencia. *MsTimer2 and FlexiTimer2 Libraries*. http://www.pjrc.com/teensy/td_libs_MsTimer2.html, 2015. [Accessed 22 May 2015].

[16] Dirk Zimoch. *EPICS StreamDevice*. http://epics.web.psi.ch/software/streamdevice/doc/, 2011. [Accessed 22 May 2015].

7 Appendix

7.1 Circuit diagram

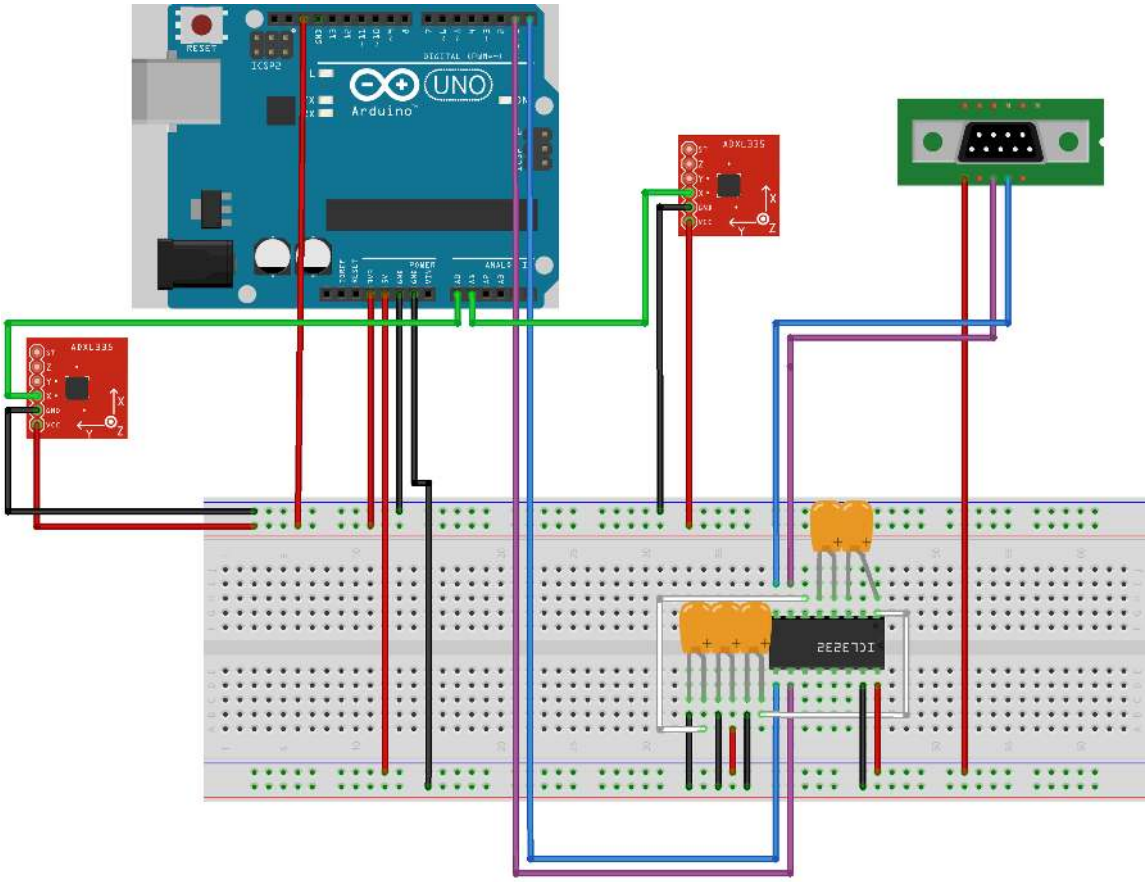


Figure 11: Circuit diagram showing all necessary connections for this project. Note that only the x-axis on the accelerometers are connected. This diagram was created using the software Fritzing [13].

7.2 Command List

Here is a full list of all the supported commands used in our serial protocol.

Command	Response	Description
AX?	AX *N*	Get one single value from an analog channel.
WX?	WX *N N N*...	Returns the waveform from an analog channel.
CXY?	CXY *N N N*...	Returns two waveforms from different analog channels (measured at the same time).
P?	P *N*	Returns the sampling period.
PN	PN	Sets the sampling period.
M?	M *NNNNNNNNNNNNNN*	Returns the IO mask
MNNNNNNNNNNNNNN	MNNNNNNNNNNNNNN	Sets the IO mask
MXN	MXN	Set the value of a specific digital pin.
DX?	DX *N*	Returns the value of a digital channel.
DXN	DXN	Sets the value of a digital channel.
Q?	Q *NNNNNNNNNNNNNN*	Returns all the digital channel values.
V?	V *S*	Returns the software version.
?	*S*	Returns some information about the device.

The terms X, Y denotes different pins onboard the Arduino (analog and digital), *N* is an integer corresponding to the data given by the command and *S* is a string. Note that every command that asks for a value *N* ends with a question mark.

7.3 MATLAB code

```
function serialObj = setupSerial(comPort, baudRate)
%% Initializes serial port communication between Arduino and MATLAB

serialObj = serial(comPort);
set(serialObj, 'DataBits',8);
set(serialObj, 'StopBits',1);
set(serialObj, 'BaudRate',baudRate);
set(serialObj, 'Parity','none');
set(serialObj, 'InputBufferSize', 4096);
end
```

```
function waveform = readWaveform(serialObj, analogPin)
%% Read single waveform from analog pin on the Arduiono

if mod(analogPin, 1) > 1e-6
    error('Analog pin must be an integer.');
    return
end

if analogPin < 0 || analogPin > 5
    error('Analog pin must have a value from 0-5.');
    return
end
```

```
output = ['W', num2str(analogPin), '?'];
fprintf(serialObj, output);
input = strsplit(fscanf(serialObj, '%c'), ' ');

if strcmp(output(1:end-1), cell2mat(input(1)))
    waveform = str2double(input(2:end));
else
    error('Error');
end
end
```

```
function [waveform1, waveform2] = readTwoWaveforms(serialObj, analogPin1, analogPin2)
%% Read waveforms from two analog pins on the Arduiono simultaneously

if mod(analogPin1, 1) > 1e-6 || mod(analogPin2, 1) > 1e-6
    error('Analog pins must be an integer.');
    return
end

if analogPin1 < 0 || analogPin1 > 5 || analogPin2 < 0 || analogPin2 > 5
    error('Analog pins must have a value from 0-5.');
    return
end

output = ['C', num2str(analogPin1), num2str(analogPin2), '?'];
fprintf(serialObj, output);
input = strsplit(fscanf(serialObj, '%c'), ' ');

if strcmp(output(1:end-1), cell2mat(input(1)))
    waveform1 = str2double(input(2:end/2+0.5));
    waveform2 = str2double(input(end/2+1.5:end));
else
    error('Error');
end
end
```

```
function [] = setIOMask(serialObj, mask)
%% Sets the IO mask on the Arduino

if length(mask) ~= 14
    error('IO mask must be of length 14');
    return
end

if any(mask > 1) || any(mask < 0) || any(mod(mask, 1) > 0)
    error('IO mask must only contain ones or zeros');
    return
end

output = regexprep(mat2str(mask), '[^\w]', '');
fprintf(serialObj, ['M', output]);
```

```
fscanf(serialObj, '%s')
end
```

```
function [] = setDigitalPin(serialObj, digitalPin, mode)
%% Sets the mode of a digital pin

if digitalPin < 0 || digitalPin > 13
    error('Analog pin must have a value from 0-5');
    return
end

if mode < 0 || mode > 1 || mod(mode, 1) > 0
    error('Mode must be ether 0 or 1');
    return
end

num = '';
if digitalPin < 10
    num = ['0', num2str(digitalPin)];
else
    num = num2str(digitalPin);
end

fprintf(serialObj, ['D', num, num2str(mode)]);
fscanf(serialObj, '%s');
end
```

7.4 Arduino code

```
//
// Arduino Slave
//
// Author     Adam Hjort, Måns Holmberg
//
// Date       2015-03-11 00:00
// Version    1.0
//
// See        ReadMe.txt for references
//

// Include libraries
#include <MsTimer2.h>


// Define variables and constants
const int NUM_ANALOG = 6; // Total number of analog pins available
const int NUM_DIGITAL = 14; // Total number of digital pins available
const int BUFFER = 512; // Buffer size
```

```
int analogDataArray[BUFFER]; // Array to store waveform
int analogPin1;
int analogPin2;
int count = 0;
float in, out;
long period = 1; // Period of interupt (maximum period is 2.15 billion seconds)
boolean digitalPinMask[14] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1}; // An array that
    holds the current IO modes for all digital pins
String input; // A string containing the input from the serial communication
String ver = "1.0"; // Software version

// getOneWaveform
//
// Brief Reads "BUFFER" number of value of one analog pin at accurate time periods.
//
void getOneWaveform() {
  analogDataArray[count] = analogRead(analogPin1);
  count++;
  if (count >= BUFFER)
  {
    MsTimer2::stop();
    count = 0;
  }
}

void SinSerial() {
  if ((input.charAt(2) == '?') && (input.charAt(3) == '\n'))
  {
  for (in = 0; in < 6.283; in = in + 0.001)
  {
    out = sin(in) * 127.5 + 127.5;
    Serial.println(out);
  }
    Serial.print('\n');
  }
  else
  {
    Serial.println("Error");
  }
}

// getTwoWaveform
//
// Brief Reads "BUFFER"/2 number of value of two analog pins simultaneously at accurate
    time periods.
//
void getTwoWaveform() {
  analogDataArray[count] = analogRead(analogPin1);
  count++;
  analogDataArray[count] = analogRead(analogPin2);
  count++;
  if (count >= BUFFER)
```

```
  {
    MsTimer2::stop();
    count = 0;
  }
}

// analogSerial
//
// Brief Returns a single value from chosen analog channel.
//
void analogSerial() {
  if ((input.charAt(1) > '/') && (input.charAt(1) < NUM_ANALOG + '0')
      && (input.charAt(2) == '?') && (input.charAt(3) == '\n'))
  {
    input.remove(2, 2);
    Serial.print(input);
    Serial.print(" ");
    Serial.print(analogRead(input.charAt(1) - '0'));
    Serial.print("\n");
  }
  else
  {
    Serial.println("Error");
  }
}

// waveformSerial
//
// Brief Retunrs a waveform from chosen analog channel.
//
void waveformSerial() {
  if ((input.charAt(1) > '/') && (input.charAt(1) < NUM_ANALOG + '0')
      && (input.charAt(2) == '?') && (input.charAt(3) == '\n'))
  {
    analogPin1 = input.charAt(1) - '0';
    MsTimer2::set(period, getOneWaveform);
    MsTimer2::start();
    delay(BUFFER * period);
    input.remove(2, 2);
    Serial.print(input);
    for (int i = 0; i < BUFFER; i++)
    {
      Serial.print(" ");
      Serial.print(analogDataArray[i]);
      analogDataArray[i] = 0;
    }
    Serial.print('\n');
  }
  else
  {
    Serial.println("Error");
  }
```

```
}

// correlationSerial
//
// Brief Retunrs two waveforms from chosen analog channels, that was capturet
    simultaneously.
//
void correlationSerial() {
  if ((input.charAt(1) > '/') && (input.charAt(1) < NUM_ANALOG + '0')
      && (input.charAt(2) > '/') && (input.charAt(2) < NUM_ANALOG + '0')
      && (input.charAt(1) != input.charAt(2))
      && (input.charAt(3) == '?') && (input.charAt(4) == '\n'))
  {
    analogPin1 = input.charAt(1) - '0';
    analogPin2 = input.charAt(2) - '0';
    MsTimer2::set(period, getTwoWaveform);
    MsTimer2::start();
    delay(BUFFER * period);
    input.remove(3, 2);
    Serial.print(input);
    for (int i = 0; i < BUFFER; i += 2)
    {
      Serial.print(" ");
      Serial.print(analogDataArray[i]);
      analogDataArray[i] = 0;
    }
    for (int i = 1; i < BUFFER; i += 2)
    {
      Serial.print(" ");
      Serial.print(analogDataArray[i]);
      analogDataArray[i] = 0;
    }
    Serial.print('\n');
  }
  else
  {
    Serial.println("Error");
  }
}

// periodSerial
//
// Brief Can be used to either get or set the sampling period.
//
void periodSerial() {
  // Returns the sampling period
  int inputLength = input.length();
  if ((input.charAt(1) == '?') && (input.charAt(2) == '\n'))
  {
    input.remove(1, 2);
    Serial.print(input);
    Serial.print(" ");
```

```
    Serial.println(period);
  }
  // Sets the sampling period
  else if (inputLength > 2 && (input.charAt(inputLength - 1) == '\n'))
  {
    input.remove(0, 1);
    boolean isCorrect = true;
    for (int i = 0; i < inputLength - 2; i++)
    {
      if (input.charAt(i) < '/' || input.charAt(i) > ':')
      {
        isCorrect = false;
        break;
      }
    }
    if (isCorrect)
    {
      long tempPeriod = input.toInt();
      if (tempPeriod > 0)
      {
        Serial.print("P");
        Serial.println(tempPeriod);
        period = tempPeriod;
      }
      else
      {
        Serial.println("Error");
      }
    }
    else
    {
      Serial.println("Error");
    }
  }
  else
  {
    Serial.println("Error");
  }
}

// maskSerial
//
// Brief Can be used to either get or set the IO mask of the digital pins (1 = input 0 =
    output).
//
void maskSerial() {
  // Returns the IO mask for the digital pins
  int inputLength = input.length();
  if ((input.charAt(1) == '?') && (input.charAt(2) == '\n'))
  {
    input.remove(1, 2);
    Serial.print(input);
```

```
  Serial.print(" ");
  for (int i = 1; i < NUM_DIGITAL + 1; i++)
  {
    if (digitalPinMask[i] == 0)
    {
      Serial.print("0");
    }
    else
    {
      Serial.print("1");
    }
  }
  Serial.print('\n');
}
// Sets the IO mask for the digital pins
else if (input.charAt(NUM_DIGITAL + 1) == '\n')
{
  boolean isCorrect = true;
  for (int i = 1; i < NUM_DIGITAL + 1; i++)
  {
    if ((input.charAt(i) != '0') && (input.charAt(i) != '1'))
    {
      isCorrect = false;
      break;
    }
  }
  if (isCorrect == true)
  {
    input.remove(NUM_DIGITAL + 1, 1);
    for (int i = 0; i < NUM_DIGITAL + 1; i++)
    {
      digitalPinMask[i] = input.charAt(i) - '0';
      pinMode(i, !digitalPinMask[i]);
    }
    Serial.println(input);
  }
  else
  {
    Serial.println("Error");
  }
}
// Set the IO status of a specific digital pin
else if ((inputLength > 3) && (inputLength < 6) && (input.charAt(inputLength - 1) ==
    '\n'))
{
  input.remove(0, 1);
  input.remove(inputLength - 1, 1);
  boolean isCorrect = true;
  for (int i = 0; i < inputLength - 2; i++)
  {
    if (input.charAt(i) < '/' || input.charAt(i) > ':')
    {
```

```
      isCorrect = false;
      break;
    }
  }
  int value = input.charAt(inputLength - 3) - '0';
  input.remove(inputLength - 3, 1);
  if (isCorrect)
  {
    int digitalPin = input.toInt();
    if ((digitalPin >= 0) && (digitalPin < NUM_DIGITAL) && (value == 0 || value == 1))
    {
      Serial.print("M");
      Serial.print(digitalPin);
      Serial.println(value);
      digitalPinMask[digitalPin + 1] = value;
      pinMode(digitalPin, !value);
    }
    else
    {
      Serial.println("Error");
    }
  }
  else
  {
    Serial.println("Error");
  }
 }
 else
 {
   Serial.println("Error");
 }
}

// digitalSerial
//
// Brief Can be used to either get or set the value of a digital pin (1 = low 0 = high).
//
void digitalSerial() {
  // Sets the value of a specific digital pin
  int inputLength = input.length();
  if ((inputLength > 3) && (inputLength < 6) && (input.charAt(inputLength - 1) == '\n')
      && (input.charAt(inputLength - 2) != '?'))
  {
    input.remove(0, 1);
    input.remove(inputLength - 1, 1);
    boolean isCorrect = true;
    for (int i = 0; i < inputLength - 2; i++)
    {
      if (input.charAt(i) < '/' || input.charAt(i) > ':')
      {
        isCorrect = false;
        break;
```

```
    }
  }
  int value = input.charAt(inputLength - 3) - '0';
  input.remove(inputLength - 3, 1);
  if (isCorrect)
  {
    int digitalPin = input.toInt();
    if ((digitalPin >= 0) && (digitalPin < NUM_DIGITAL) && (value == 0 || value == 1))
    {
      Serial.print("D");
      Serial.print(digitalPin);
      Serial.println(value);
      digitalWrite(digitalPin, value);
    }
    else
    {
      Serial.println("Error");
    }
  }
  else
  {
    Serial.println("Error");
  }
}
// Returns the value of a specific digital pin
else if ((inputLength > 3) && (inputLength < 6) &&
        (input.charAt(inputLength - 2) == '?') && (input.charAt(inputLength - 1) ==
            '\n'))
{
  input.remove(0, 1);
  input.remove(inputLength - 3, 2);
  boolean isCorrect = true;
  for (int i = 0; i < inputLength - 3; i++)
  {
    if (input.charAt(i) < '/' || input.charAt(i) > ':')
    {
      isCorrect = false;
      break;
    }
  }
  if (isCorrect)
  {
    int digitalPin = input.toInt();
    if ((digitalPin >= 0) && (digitalPin < NUM_DIGITAL))
    {
      Serial.print("D");
      Serial.print(digitalPin);
      Serial.print(" ");
      Serial.println(digitalRead(digitalPin));
    }
    else
    {
```

```
      Serial.println("Error");
     }
    }
    else
    {
     Serial.println("Error");
    }
  }
  else
  {
    Serial.println("Error");
  }
}

// getAllDigital
//
// Brief Returns the values of all digital pins (1 = input 0 = output).
//
void getAllDigital() {
  if ((input.charAt(1) == '?') && (input.charAt(2) == '\n'))
  {
    input.remove(1, 2);
    Serial.print(input);
    Serial.print(" ");
    for (int i = 0; i < NUM_DIGITAL; i++)
    {
     Serial.print(digitalRead(i));
    }
    Serial.print('\n');
  }
  else
  {
    Serial.println("Error");
  }
}

// versionSerial
//
// Brief Returns current software version
//
void versionSerial() {
  if ((input.charAt(1) == '?') && (input.charAt(2) == '\n'))
  {
    input.remove(1, 1);
    Serial.print("V ");
    Serial.println(ver);
  }
  else
  {
    Serial.println("Error");
  }
}
```

```
// Setup
//
// Brief Setup
//
void setup() {
  // Setup
  bitClear(ADCSRA, ADPS0); // Running with high speed clock (set prescale to 16)
  bitClear(ADCSRA, ADPS1);
  bitSet(ADCSRA, ADPS2);
  Serial.begin(115200); // Sets Serial baud rate
  analogReference(EXTERNAL); // Configures the reference voltage used for analog input
}

// Loop
//
// Brief Loop
//
void loop() {
  while (Serial.available() > 0)
    // Adds input char to the string "input" while there still is characters to read
  {
    char lastRecived = Serial.read();
    input += lastRecived; // Adds last recived char to "input"

    if (lastRecived == '\n')
      // Continue to do stuff if the last character recived was a new line
    {
      switch (input[0]) {
        case 'A':
          analogSerial();
          break;
        case 'W':
          waveformSerial();
          break;
        case 'C':
          correlationSerial();
          break;
        case 'P':
          periodSerial();
          break;
        case 'M':
          maskSerial();
          break;
        case 'D':
          digitalSerial();
          break;
        case 'Q':
          getAllDigital();
          break;
        case 'S':
          SinSerial();
```

```
        break;
      case 'V':
        versionSerial();
        break;
      case '?':
        // Returns some information about this device
        Serial.println("Arduino slave");
        Serial.print("Version ");
        Serial.println(ver);
        Serial.println("Uppsala University, Sweden");
        Serial.println("Software written by Adam Hjort and Mans Holmberg");
        break;
      default:
        Serial.println("Error");
        break;
    }
    input = ""; // Clear recieved buffer.
  }
 }
}
```