

Electrical and Computer Engineering

ELEC 291/292 Winter 2020

Instructor: Dr. Jesus Calvino-Fraga

Section 201

**Electrical Engineering Design Studio I**

**Project 2 - Frequency-Meter Metal Detector**

|  |  |  |
| --- | --- | --- |
| **Student Number** | **Name** | **%Points** |
| 76149921 | Mihir Bhatia | 150 |
| 81087413 | Prashant Wason | 150 |

Date of Submission: April 14th, 2021

**2. Table of Contents**

1. Title Page
2. Table of Contents
3. Introduction
4. Investigation
   1. Idea Generation
   2. Investigation Design
   3. Data Collection
   4. Data Synthesis
   5. Analysis of Results
5. Design
   1. Use of Process
   2. Need and Constraint Identification, Problem Specification
   3. Solution Generation
   4. Solution Evaluation
   5. Detailed Design
   6. Solution Assessment
6. Life-Long Learning
7. Conclusion
8. References
9. Appendices

**3. Introduction**

For this Project, we designed, built, programmed and tested a microcontroller based frequency meter metal detector. The purpose of the metal detector was primarily to have a portable device that can detect small and large ferrous and non ferrous objects while treasure hunting, security screening, etc. For this project, we built an inductor coil out of an enameled copper magnet wire, the inductance of which would vary as it neared a metallic object, depending upon the size and type of metal of the object. We connected this inductor to a Colpitts Oscillator, the frequency of which would change with inductance in the coil, and measured frequency using the ATMEGA328P microcontroller. By measuring the frequency of the oscillator when different types of objects were near it, we could determine the minimum or maximum frequencies that would be reached for the different types of objects. The microcontroller was connected to a speaker/ headphones circuit with SPI flash memory, and programmed it to play a sound corresponding to the identification of the type of object (small or large, ferrous or non ferrous) when the frequency measured indicated that the object was nearby. We programmed the microcontroller using C, storing an external sound(.WAV) file in SPI memory.

**4. Investigation**

1. **Idea Generation**

Our primary focus as we planned the project was building and integrating the coil with the oscillator circuit, and using frequency readings to accurately identify the kind of object near the coil. Given that frequency was inversely proportional to inductance of the circuit, and that the inductance would decrease when a non ferrous object was nearby, and would increase when a when a ferrous object was nearby, we hypothesised that the frequency would decrease by a greater magnitude the larger a ferrous object was close to it, and would increase by a greater magnitude the larger a non-ferrous object was close to it.

1. **Investigation Design**

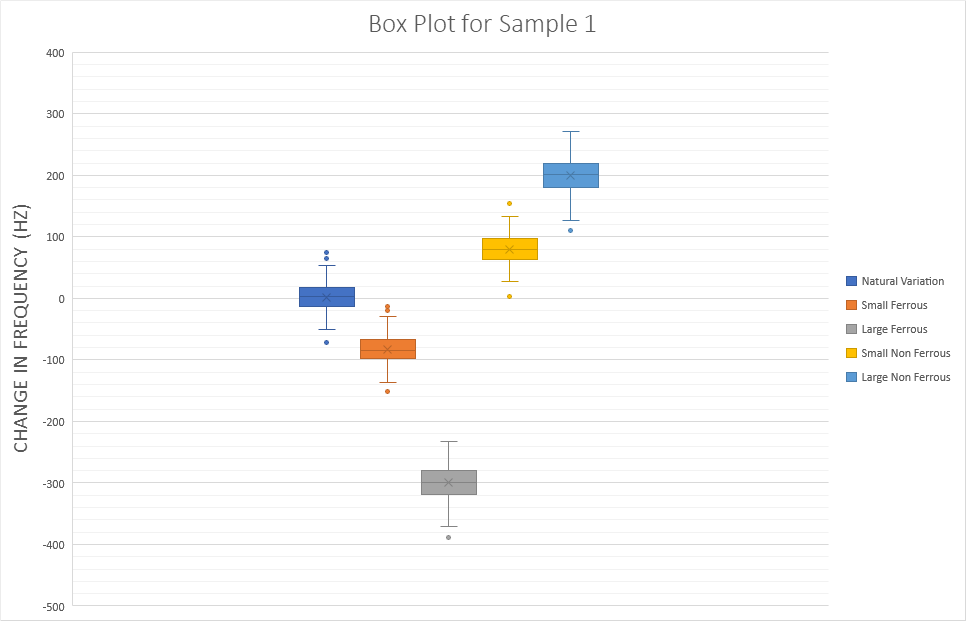
To investigate and develop an accurate modelling for frequency of the oscillator circuit and the type of object (small or large, ferrous or non-ferrous), we built coils using 28 AWG enameled copper wire, having a perimeter of 40 cm and 120 turns, tied together with zip ties, and connected it to out Colpitts Oscillator. We programmed our microcontroller using its built in Timer 1, to calculate and display the frequency of the Oscillator on a computer screen. The change in frequency was measured as different types of objects were placed inside, on or just outside the coil. We repeated the experiment numerous times, and took several measurements of change in frequency vs type of object, to account for extremities and natural variation of the frequency of the oscillator.

1. **Data Collection**

We used small screw bits as our small ferrous object, crumpled aluminum foil as our small non ferrous object, a set of iron wire strippers for our large ferrous object, and an aluminum empty water bottle for our large non-ferrous object. We used excel to plot the change of frequency vs type of object. There was constant variation in frequency readings with or without the objects, and so all the measurements were recorded over a duration of 30 seconds, after the object had been placed. We used excel to plot the change in frequency vs the type of object. We also plotted the natural variation in frequency of the circuit by measuring frequency for 30 seconds at the start of each experimental trial. The mean of this set of values was used as a reference to compute change of frequency.

1. **Data Synthesis**

Using excel, we plotted the natural variation in frequency of the circuit as well as the change in frequency for different types of objects, in the form of box plots. There were found to be no significant differences in frequency change for different placements of the object, though greater changes were observed in general when the object was inside the coil, and thus modelling was only done for trials with the objects placed in the coil. The modelling of one of the samples is shown below. The rest are given in Appendix A.



***Fig 1.*** *Box Plot of Change in Frequency vs Type of Object*

1. **Analysis of Results**

The results seem to indicate that frequency decreases due to increased inductance from the coil when a ferrous metal is placed in it, and the opposite is observed when a non ferrous metal is placed in it. This is shown by the median for each experimental group, which differs significantly between the groups and follows the pattern described. It is unlikely that this could be the result of spurious correlations, given the large number of readings for each sample (approximately 150 for each condition), and similar results being achieved across samples.

The experiments also highlight that there is a significant amount of natural variation in the frequency of the oscillator circuit, with an average standard deviation of 22 Hz across all the samples for the natural variation condition. This could introduce error in measurement for small ferrous and non ferrous objects, since the difference in means between these conditions and the natural variation averaged across the different samples was only 80 Hz lower (small ferrous) or 82 Hz higher (small non ferrous). This would imply that the natural variation in the frequency of the oscillator circuit could cause the microcontroller to falsely identify non metallic objects, or nothing at all as ferrous or non ferrous objects (false positives). This risk would have to be minimized while designing algorithms and writing code.

Given the limitation of the natural variance in the coil, the possibility of this error was not significant for larger metallic objects, and so this limitation was overall deemed acceptable.

**5. Design**

1. **Use of Process**

The engineering design process was used to find relevant stakeholders, understand stakeholder needs, form technical requirements and evaluation criteria, generate ideas given the constraints and evaluate solutions.

The first step was to understand and clarify the design problem, by identifying key stakeholders and their needs. We studied the different uses of the device, such as treasure hunting, security scans, landmine screening, etc, which helped us identify potential stakeholders (casual and serious users, manufacturers). Through that, we were able to identify different, and sometimes competing needs and form design constraints and requirements. Finally, we developed a few possible solutions, all involving the making and usage of an inductor coil in an oscillator circuit. We evaluated these solutions and chose the most promising one. During implementation, we made small and constant iterations to improve upon the design.

1. **Need and Constraint Identification and Problem specification**

By finding out different uses for the device, we identified the key stakeholders. These were casual users (who’d use the device for treasure hunting, or exploring, or as a plaything for children) and more serious users (who’d use the device for security screenings, explosives detection, etc.) and manufacturers. We identified stakeholder needs and wants by considering what aspects of such a device will be most important to them.

More casual users of such a product would place an emphasis on the device being easy to operate. Key tenets of that would be that the device should be portable and have a low learning curve. A key design constraint/ requirement that this leads to is that the device should be battery powered, so as to not be connected to an external power source like a computer, since that would limit portability. Additionally, a performance criteria would be that the coil be as small (in terms of area) as possible, since it would be easier to carry. The ease of learning to use the device could be operationalised by the simplicity of the circuit and the function, such as not having many buttons for different modes of operation, just an on/ off switch. This leads to a performance evaluation criteria: simplicity of operation, with the easier the operation, the better the performance.

Serious users of the device on the other hand, besides the needs of the casual users, would also want the device to be as accurate as possible. Given that they could be using the device to probe potentially dangerous situations, the safety of operation is another key need. The need for safety while operating leads to the constraint that the device can be operated from a little distance- achieved by having the wires connecting the coil to the circuit be longer. The accuracy of the device could be operationalised and evaluated by having a low rate of false positive detections, and of false negative detections. Given the background natural variation/ noise in the frequency readings described earlier, this could have multiple solutions (described and evaluated for performance in later sections). The accuracy constraint also places requirements on the build of the coil, since a coil with greater area or more turns (higher base inductance) would respond more to objects nearby, hence increasing accuracy.

The manufacturers of the device on the other hand, are mainly concerned with the simplicity of and cost of construction of the device, and that it is easy to replicate. The last of these needs impose the requirement of clear circuit schematic(s) for different parts of the circuit. The simplicity and cost of construction could be evaluated by the simplicity of the circuit and by having common components and parts that are easily obtainable, and a coil that is easily constructed.

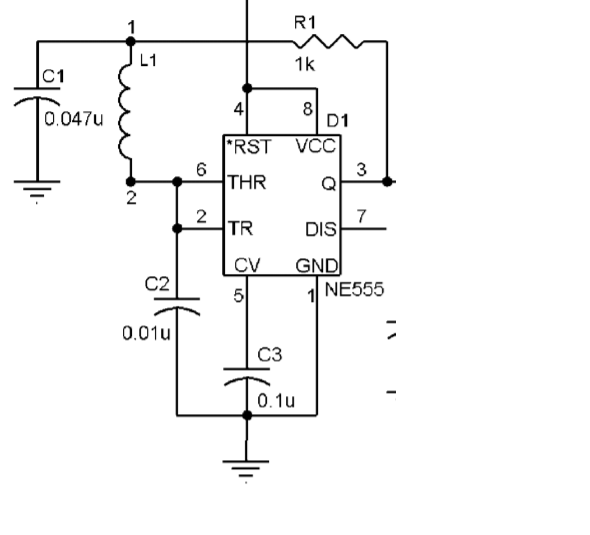
Finally, we were also constrained by the electronics available for the device. All our design solutions were limited to using a simple programmable microcontroller (ATMEGA-328P), a Colpitts Oscillator circuit and SPI flash memory.

1. **Solution Generation**

The project had multiple different parts, all of which affected different performance criteria. The project can be broken into 5 main areas. These were the Colpitts Oscillator, Speaker Circuit with SPI flash Memory, Inductor Coil, C Program Design and the Voltage Regulator. These are described below, and how they relate to performance criteria (where relevant) is explained.

*Colpitts Oscillator:*

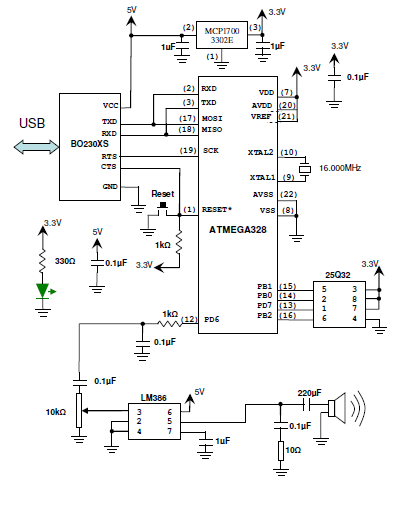
A Colpitts Oscillator uses capacitors and an inductor (the coil in our case) and NE555 timer. This oscillator has a frequency that is inversely proportional to the square root of inductance. Thus as objects are placed on or near the inductor (coil), the frequency will change. The output of the 555 timer goes to one of the pins of the ATMEGA32P microcontroller, where its inbuilt timer is used to compute the frequency.



***Fig 2.*** *Circuit schematic of a Colpitts oscillator using a 555 timer*

*Speaker Circuit with SPI memory:*

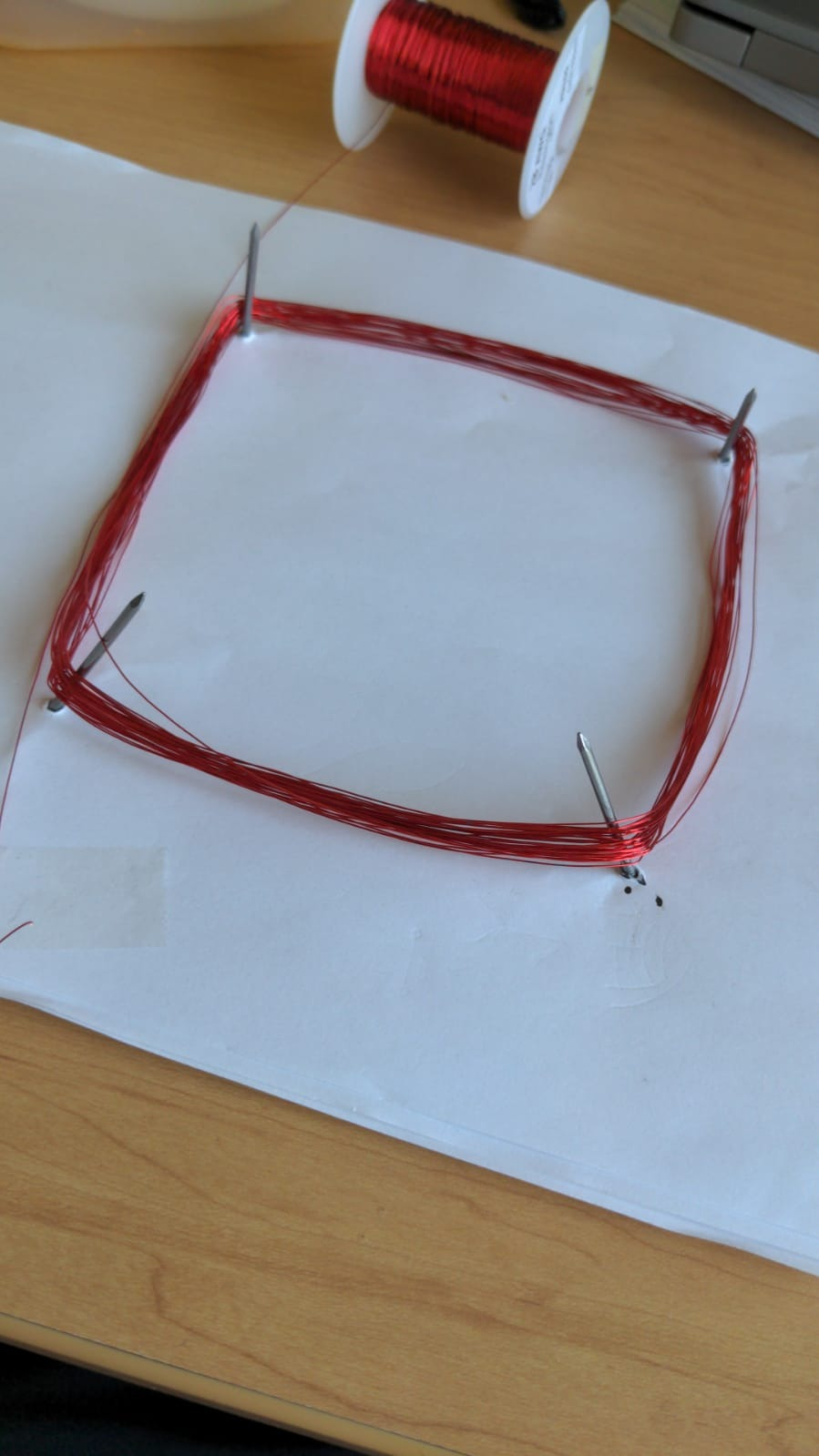
The purpose of the circuit was to store a .WAV audio file in the 25Q32 SPI flash memory microchip. It enables playback of specific parts of the audio file after a certain trigger (to be programmed on the ATMEGA32P microcontroller). A detailed schematic of the circuit used is shown below.



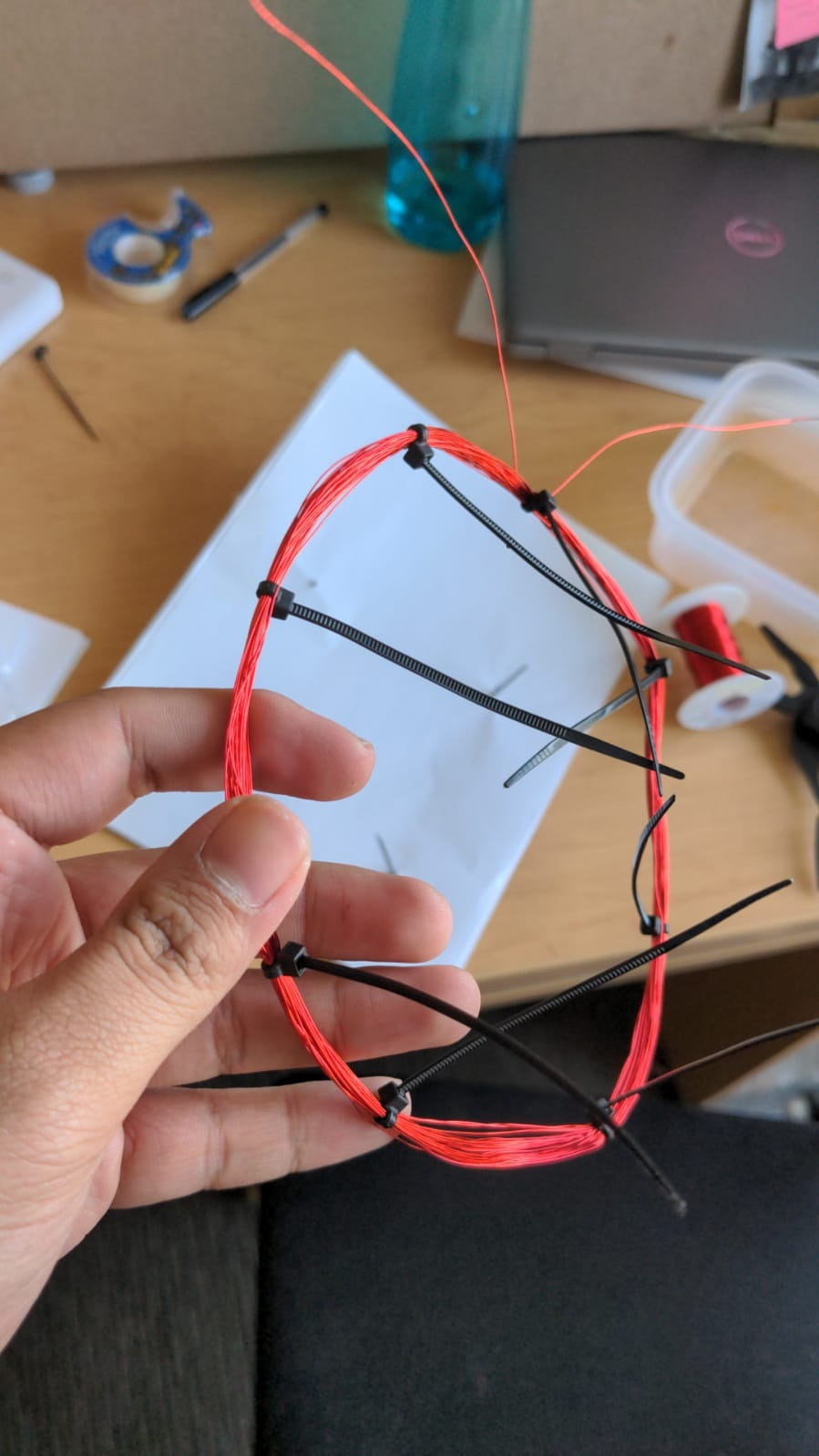
***Fig 3.*** *Speaker and SPI Circuit Schematic*

*Inductor Coil:*

To maximize the area of the coil (since it would increase accuracy), the coils were all shaped as a circle, since a circle produces the largest area for a given perimeter (length of coil). Given this, there were 2 main designs for the inductor coil. One of the designs was to maximize the area of the coil by having a larger circumference, but have a fewer number of turns, since this will produce a higher net inductance. The other approach was to maximise the number of turns, and reduce the coil area. This would make the coil more portable. Both the designs of the coils used zip ties to hold the coil in place once it had been winded. The larger coil had a perimeter of approximately 80 cm, but only 60 turns. The smaller coil would have a perimeter of approximately 40 cm, but 120 turns.The winding process and one of the sample coils is presented below. Both these designs are evaluated in the next section.



***Fig 4.*** *Winding process for one of the coils that would be tested*



***Fig 5.*** *The finished smaller coil with 120 turns*

*C Program Design:*

The primary consideration when designing the C program was for a method to minimise the background noise/ natural variation in the frequency readings of the Oscillator circuit, since it could lead to false positives (i.e. the speaker would play sound corresponding to the detection of a small ferrous or non ferrous object, despite no object being there). There were 2 design approaches that were considered. The first one was sampling how the frequency varied when the program initiated, and using the sample to compute a base mean frequency and standard deviation, and then only triggering sound output when the signal was 2.5 standard deviations away (this would provide a max false positive rate of 1.2%). The other approach was to independently sample the variation in frequency, and determine, through experimentation, a deviation value/ trigger, that would minimise the rate of error while ensuring a maximum number of true positives would be detected. Each of these methods are evaluated in the next section.

*Voltage Regulator:*

Due to portability requirements, the device had to be battery powered. This meant using a 9V battery to power the different circuits. Thus a voltage regulator LM7805 was used in conjunction with 2 capacitors to convert the 9V battery voltage to 5 volts, for use in powering the LM786 amplifier and BO230XS.

1. **Solution Evaluation**

Given the many design constraints imposed by the limited availability of electronics and the project itself, we defaulted to using ATMEGA32P as a microcontroller, a Colpitts Oscillator constructed from a NE555 Timer, and 25Q32 SPI flash memory. The designs of the coils and the software approaches to improve performance by minimising the effect of background noise were evaluated separately.

The designs of the coils were constructed and tested on their ability to detect objects. It was found that the smaller coil was better overall at detecting objects, likely since the greater number of turns meant that a smaller object would cause a greater change in inductance (less false negatives). They were also scored on performance criteria developed and outlined in section 5b), adapted for the coil in particular. The relative performance in detecting objects, and the scoring on different criteria was then fed into a Weighted Decision Matrix (WDM) to facilitate a decision. The weights for the criteria were developed keeping in mind the relative importance of the stakeholders, as well as the overlap of their needs. This WDM is presented below and can also be found in Appendix B.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | **Weight** | **Idea** | | | |
| **Larger Coil, Fewer Turns** | | **Smaller Coil, Greater Turns** | |
| **Score** | **Weighted Score** | **Score** | **Weighted Score** |
| Portability | 30.0% | 3 | 0.9 | 8 | 2.4 |
| Accuracy (Smaller Rate of False Positives) | 22.5% | 7 | 1.575 | 4 | 0.9 |
| Accuracy (Smaller Rate of False Negatives) | 22.5% | 2 | 0.45 | 8 | 1.8 |
| Cost | 10.0% | 5 | 0.5 | 5 | 0.5 |
| Ease of Construction | 15.0% | 10 | 1.5 | 5 | 0.75 |
| Total | 100.0% |  | **4.925** |  | **6.35** |

***Table 1.*** *WDM showing the scores of the different coils*

The results of the WDM show that the larger coil scored 4.925 out of 10, and that the smaller coil scored 6.35 out of 10, indicating that the smaller coil was better suited for this application. The difference in scores comes mainly from the difference in performance in portability, since the smaller area of the coil makes it easier to carry around, and the higher accuracy resulting from a much smaller rate of false negatives in detecting objects. The larger coil scored significantly higher on ease of construction, since it would have to be turned less number of times. The smaller coil was hence used in the design.

Similarly, the two approaches of designing the program were evaluated in terms of simplicity of code, memory requirements, processing requirements, and accuracy of detection. These criteria were weighted based on the needs of the stakeholders, and using a Weighted Decision Matrix, a selection was made. The WDM is shown below:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | **Weight** | **Idea** | | | |
| **Program Samples Base Frequency** | | **Independent Sampling of Base Frequency** | |
| **Score** | **Weighted Score** | **Score** | **Weighted Score** |
| Simplicity of Code | 10.0% | 2 | 0.2 | 9 | 0.9 |
| Memory Requirements | 20.0% | 5 | 1 | 9 | 1.8 |
| Processing Requirements | 20.0% | 8 | 1.6 | 10 | 2 |
| Accuracy (Smaller Rate of False Positives) | 25.0% | 10 | 2.5 | 6 | 1.5 |
| Accuracy (Smaller Rate of False Negatives) | 25.0% | 6 | 1.5 | 8 | 2 |
| Total | 100.0% |  | **6.8** |  | **8.2** |

***Table 2.*** *WDM showing the scores of the different programming approaches*

The results of the WDM show that the approach of using the program to sample the variation in frequency scored 6.8/10, and that the approach of independently sampling the variation in base frequency scored 8.2/10, indicating that the later was a better choice. The second approach scored significantly better on every index, besides the accuracy, since it had a comparatively significantly higher rate of detecting false positives. The second approach was hence adopted for the final design. This approach is described in detail in the next section.

1. **Detailed Design**

The entirety of the project can be divided into 4 sections: Inductor Coil, Colpitts Oscillator, Speaker with SPI Flash Memory, and General Software and Algorithm Design. In this section of the report, we will go into details of each of these sections, explaining how the hardware and software comes together to make the metal detector work.

*Inductor Coil:*

The finalised inductor coil was made of 28 AWR enameled copper wire, had 120 turns with a circumference of 40 cm, held together by zip ties. The inductor coil is essential for the project and acts as the probe for the metal detector. An inductor coil has inductance given by , where N is the number of turns, A is the surface area of the coil, I is current through the coil and 𝜇 is the permeability of the space enclosed in the coil. Since current is constant, and the permeability of air is constant, the inductance of the coil is roughly constant. However, when any metal enters the space enclosed by the coil, it changes the permeability, hence causing inductance to change. Ferrous metals which can be magnetised cause the inductance to increase, and non ferrous metals which apply magnetic forces in the opposite direction cause the inductance to decrease. This change in inductance, and the magnitude of change enables us to identify the type of object and its size.

*Colpitts Oscillator:*

The inductor coil described above connects to a Colpitts Oscillator which is used to detect changes in the inductance. Using the hardware block given in figure 2, the Colpitts Oscillator was constructed, with the frequency of the output given by where L1 is the inductance of the coil, and C is the series combination of C1 and C2. The output of the oscillator is connected to pin PD5 of the ATMEGA328p board. We used and modified the given ‘period.c’ (Appendix D) program to measure and display the frequency of the oscillator on the Putty window of a connected PC. This program configures PD5 as an input and uses the inbuilt 16 bit Timer 1 of the ATMEGA328p board to measure the period and consequently calculate the frequency. Configuring PD5 as an input makes the sinusoidal signal a square wave, enabling the program to compute the period using no prescalars on it Timer 1 by measuring the time between two highs (1s) on the pin.

*Speaker and SPI Flash Memory Circuit:*

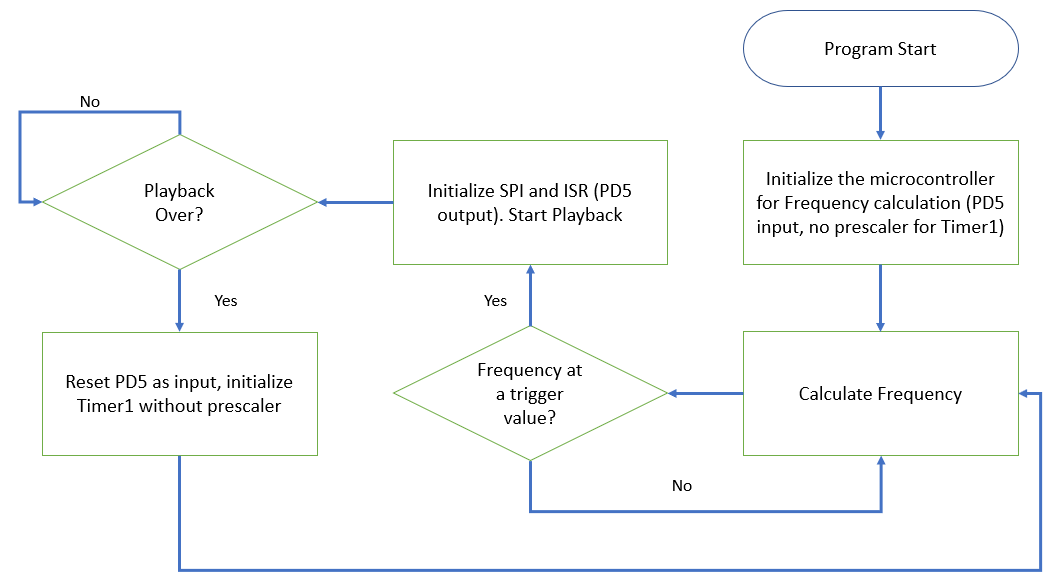
We used the SPI flash memory of the 25Q32 microchip to store audio in the form of a .WAV file, and play certain parts of it when triggered. Using the hardware block shown in figure 3, the circuit was constructed. Once audio had been stored in the 25Q32 microchip, upon being triggered via Pin PB1 of the microcontroller, it would send the requested information (part of the audio file) to the microcontroller via Digital Pin PB0. The microcontroller would then send this information to LM386 amplifier via Pin PD6. An important thing to note here is that the amplifier uses Analog signals, but there is no inbuilt DAC in the ATMEGA328p microcontroller. Thus the ‘AVR\_receiver.c’ (Appendix c) program provided was used to perform pulse width modulation (PMW) on the output pin PD6. This enables the smooth operation of the circuit, with no damage to the amplifier and clear sound output. This program also enabled the storage and playback of audio. An important thing to note was that the part of the audio to be played was determined by using a pointer in the audio file, and a certain number of indexes forward. The number of indexes forward was matched using interrupts. An interrupt flag was triggered when the Start\_Playback function was called.

This interrupt sends audio output to PD6 at a preset frequency using Timer1 inside the PD5 pin. This interrupt only works by comparing the frequency of Timer1 to the value of inbuilt register A, which contains the frequency of Timer1 when pin PD5 is set as an output.

*General Software and Algorithm Design:*

The final and vital task for the metal detector to work was bringing the different programs regarding frequency measurement and playing sound together, so that when the frequency changes corresponding to a certain type of object, a particular part of the audio file is played, enabling the identification of the object. The circuit for the same was built by combining the 2 previously shown schematics together, since they used different components and different pins on the microcontroller. This final program can be found in Appendix E. There were several challenges for this program design due to the limitations imposed by the microcontroller and the natural variation in base frequency readings of the oscillator. The approaches used to solve these are described below.

The biggest limitation of the microcontroller was that both the frequency calculation and the SPI memory ISR used the 16 bit Timer1 inside pin PD5. In the first case, pin PD5 had to be configured as an input, and in the second, as an output. Since the microcontroller had only 1 16 bit timer, both of these functions of the program were carried out by the same PD5 pin. This was done by having the program initialise and start calculating frequencies, and then whenever an object was detected, it would initialise and set up the SPI playback, including setting PD5 as an output. The program would then pause until the playback had been completed, then reset PD5 and Timer1 as inputs before resuming to calculate the frequency. This shown through the flowchart below:



***Fig 6.*** *Flowchart showing how Timer1 and PD5 toggles between input and output*

The diagram above demonstrates how when frequency is at a value that would cause the trigger of the ISR, how SPI would be initialized and Timer1 and PD5 would be configured as output, before being configured as input again after the playback completes. This highlights the second limitation: picking frequencies for the trigger points for the playback of the sound to minimize errors (false positives and negatives).

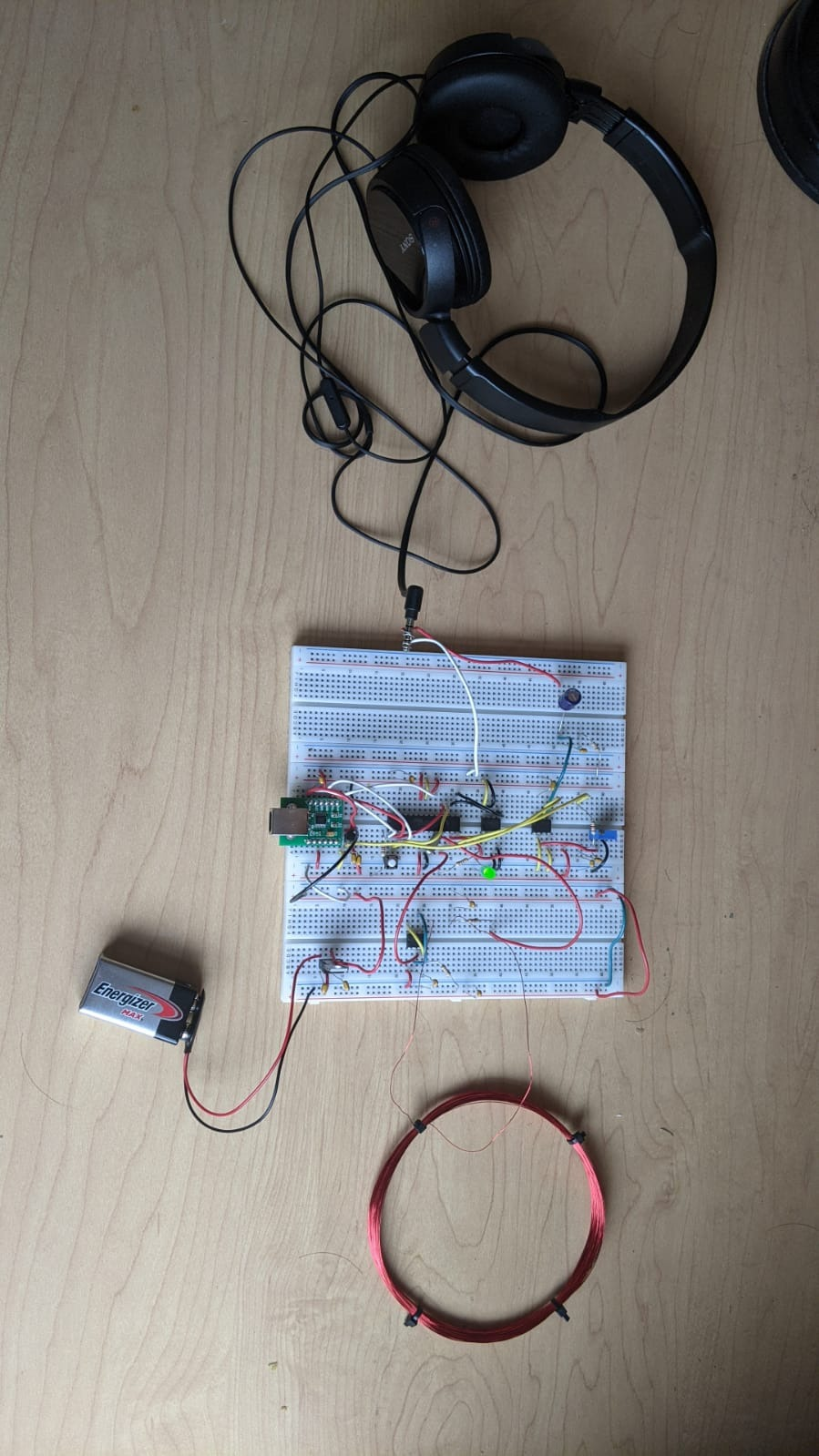
To minimise errors, the program was set up such that it would not have any pre-stored value for base frequency, just values for triggering different ISR routines. The preset frequency was instead set to be the frequency calculated once at the beginning of the program. Then the frequency was constantly compared to this base frequency. If the frequency was in a certain range above or below the base frequency, different ISRs would be triggered.

These range of frequencies were determined by way of repeated measurements of frequency with different objects placed inside (as shown in figure 1). From these it was determined that for small non-ferrous objects, the average frequency was approx 80 Hz higher than base, and 80 Hz lower than base for small ferrous objects. Likewise, it was 200 Hz higher for larger non-ferrous objects, and 300 Hz lower for larger ferrous objects. All of these, along with the natural variation has a standard deviation of approximately 20 Hz. Thus, an ISR corresponding to small non-ferrous objects was triggered when frequency calculated was 40Hz higher than base. Similarly, the value was 150Hz higher than base for larger non ferrous objects, 40Hz lower than base for small ferrous objects, and 250Hz lower for larger ferrous objects. (i.e. within 2 standard deviations of their mean values). These points were chosen because they minimised the probability of a false positive or negative (frequency would have to vary by 2 standard deviations when no object was there for false positive, or by the same amount when a small object was there which happens only about 5% of the time).

It is worth noting that the different ISRs described above were all of sound playback. They only differed in the number of indexes to be played, and their starting points, depending upon the audio file generated/ used for the purpose. These values were determined through tests by choosing different starting indexes to see at which indexes the different identifications began.

1. **Solution Assessment**

The finalised and constructed metal detector is shown below.



***Fig 7.*** *Finished metal detector. Instead of speaker, headphones were used to output sound*

Once it had been built, the finished product underwent a series of tests. The first test was simply to switch it on and leave it running for one minute, to see the number of times a false positive error was made. The error rate was lower than expected, with playback occurring briefly only 4 times. This meant a high amount of accuracy, which was one of our performance criteria.

Next, systematically, different objects were placed inside and near the coils to see how quickly a detection would be made. The objects were correctly identified immediately (under 2 seconds) most of the time, with the only errors in identification happening when larger objects were placed more than 5 cm away from the coil, causing the frequency change to not be as high, and a subsequent misidentification as a small object. This was corrected however, the moment the distance between the coil and the object was shortened.

All things considered, this is a strong design. It has a high rate of accuracy, is sensitive to and can detect various objects from a decent distance away, and has easily replicable circuit schematics for construction, with common and cheap components.

One of the limitations of the design are wear and tear to the coil with use which would complicate the change in frequency with different objects. Another limitation is that although unlikely, it is possible for the base frequency measurement to be an outlier, rendering the detector to be unable to identify small objects of 1 of the types until it is switched off and on again. A final limitation is the lack of visual indicators when a detection is made, making the device not usable for people with hearing aids.

**6. Life-Long Learning**

Through this project, we learnt how to construct a Colpitts oscillator using a 555 timer. We learnt how to program SPI flash memory using C. We learned a lot about the ATMEGA architecture, and how to operate its internal timers to run several programs at once. We learned to program ISRs via C and using vectors on the ATMEGA architecture. This project also developed our communication, collaboration and time management skills.

**7. Conclusion**

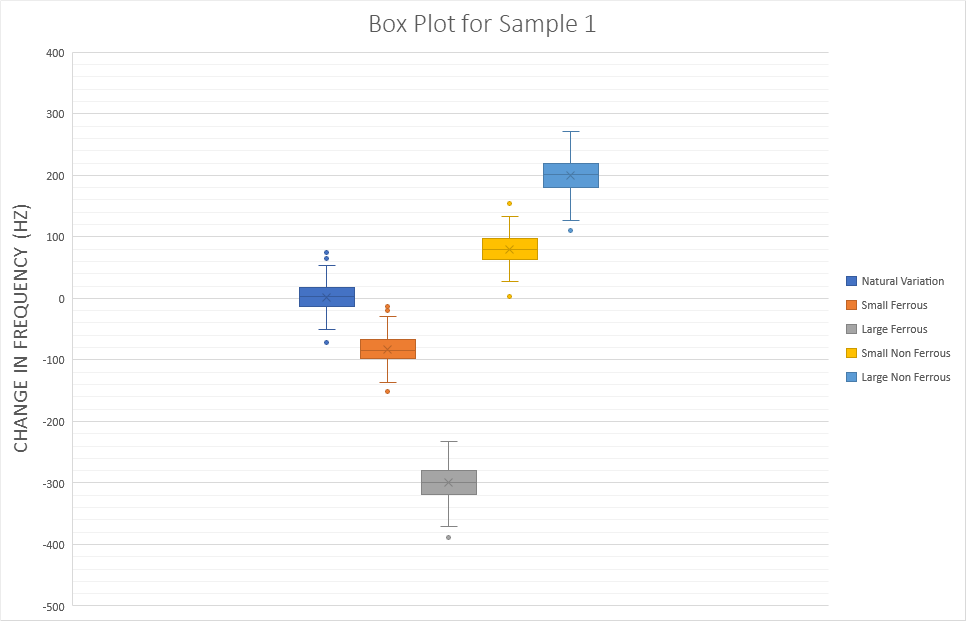
Overall, for this project, we designed, built, programmed and tested a microcontroller based frequency meter metal detector. We used a Colpitts oscillator, an inductor coil as a probe, SPI flash memory to store .WAV files, an ATMEGA328p microcontroller, programmed with C and a battery connected to a voltage regulator to design a portable metal detector that could identify and play sounds identifying small or large, ferrous or non ferrous objects. This was done by measuring the frequency of the oscillator, which would vary as metal would get closer to the coil, and comparing it to a base frequency to calculate whether the change in frequency was great enough for a particular type of object. When this was the case, and ISR would be triggered that would play a part of the audio file stored in the SPI flash memory, corresponding to the identification of the object. During this project, we encountered several design challenges and options, such as both the frequency calculations and the ISR using the same built in timer, and methods to account for the natural variability in frequency, and the ideal design for the coil. Overall, the project took 30 hours to complete.

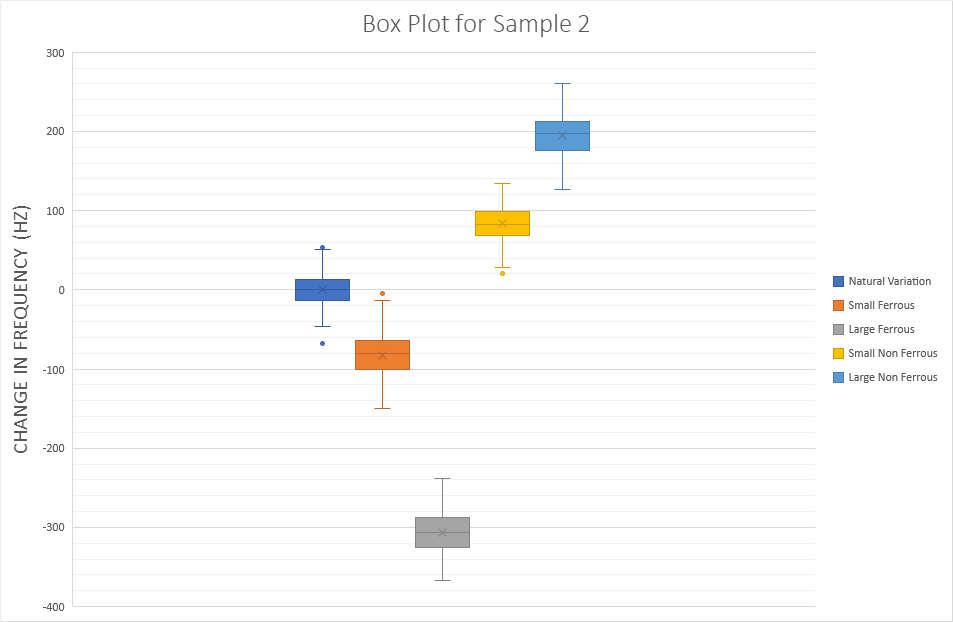
**8. References**

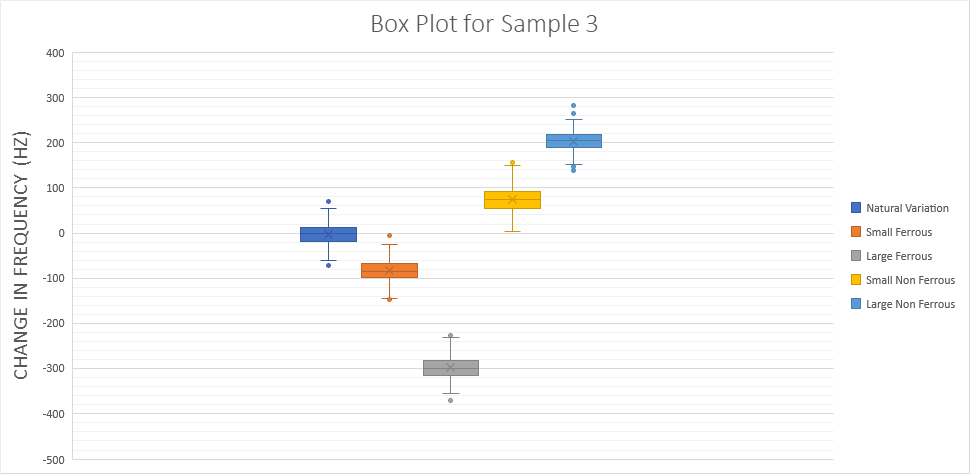
1. Calvino-Fraga, J (2021), AVR\_Receiver [C]. <https://canvas.ubc.ca/courses/59445>
2. Calvino-Fraga, J ELEC 291. Project Document, Topic: “ELE291\_Project\_2\_2021.pdf”. Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, April 14,2021.
3. Calvino-Fraga, J ELEC 291. Project Document, Topic: “Circuit Cellar article - Frequency Meter Metal Detector.pdf”. Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, April 14,2021.
4. Calvino-Fraga, J (2021), Period [C]. <https://canvas.ubc.ca/courses/59445>

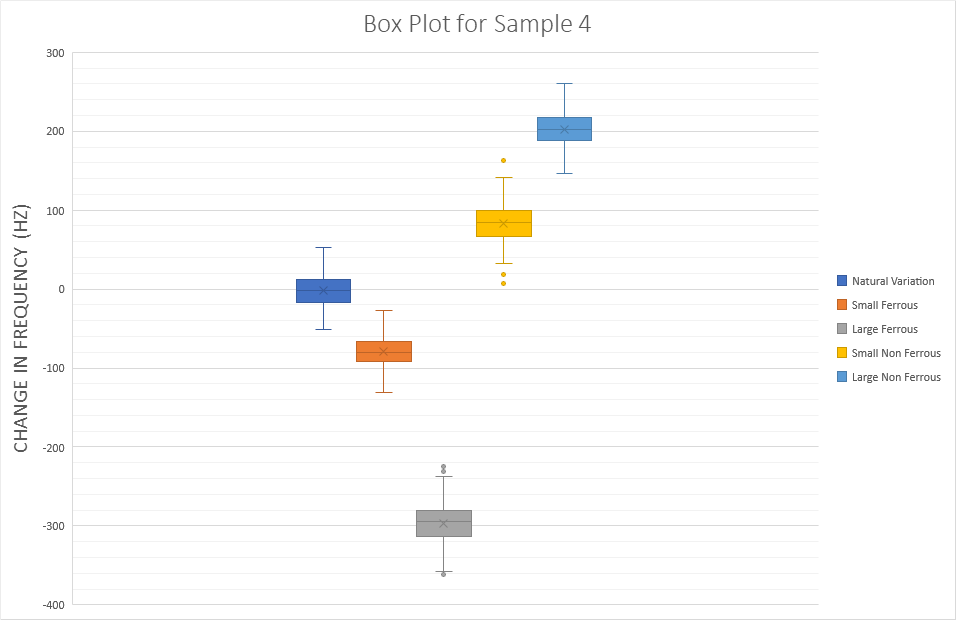
**9. Appendices**

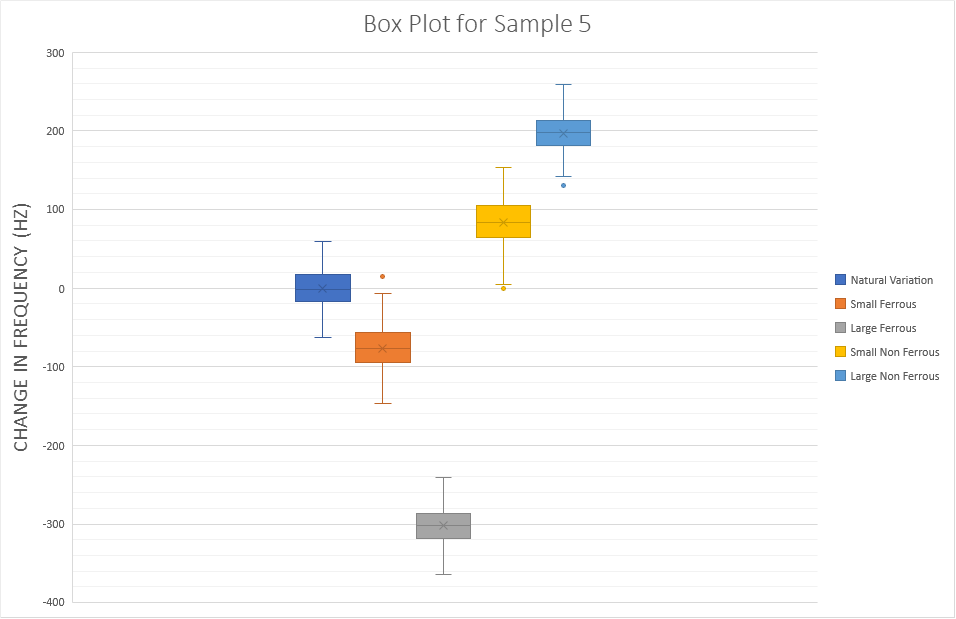
1. **Box plot of change in frequency for different objects**

****









1. **Weighted Decision Matrices**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | **Weight** | **Idea** | | | |
| **Larger Coil, Fewer Turns** | | **Smaller Coil, Greater Turns** | |
| **Score** | **Weighted Score** | **Score** | **Weighted Score** |
| Portability | 30.0% | 3 | 0.9 | 8 | 2.4 |
| Accuracy (Smaller Rate of False Positives) | 22.5% | 7 | 1.575 | 4 | 0.9 |
| Accuracy (Smaller Rate of False Negatives) | 22.5% | 2 | 0.45 | 8 | 1.8 |
| Cost | 10.0% | 5 | 0.5 | 5 | 0.5 |
| Ease of Construction | 15.0% | 10 | 1.5 | 5 | 0.75 |
| Total | 100.0% |  | **4.925** |  | **6.35** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | **Weight** | **Idea** | | | |
| **Program Samples Base Frequency** | | **Independent Sampling of Base Frequency** | |
| **Score** | **Weighted Score** | **Score** | **Weighted Score** |
| Simplicity of Code | 10.0% | 2 | 0.2 | 9 | 0.9 |
| Memory Requirements | 20.0% | 5 | 1 | 9 | 1.8 |
| Processing Requirements | 20.0% | 8 | 1.6 | 10 | 2 |
| Accuracy (Smaller Rate of False Positives) | 25.0% | 10 | 2.5 | 6 | 1.5 |
| Accuracy (Smaller Rate of False Negatives) | 25.0% | 6 | 1.5 | 8 | 2 |
| Total | 100.0% |  | **6.8** |  | **8.2** |

1. **Source code of the modified ‘period.c’ file measuring frequency of the Colpitts Oscillator and displaying it on the Putty Window.**

// This program shows how to measure the period of a signal using timer 1 free running counter.

#define F\_CPU 16000000UL

#include <stdio.h>

#include <stdlib.h>

#include <avr/io.h>

#include <avr/interrupt.h>

#include "usart.h"

unsigned int cnt = 0;

void wait\_1ms(void)

{

unsigned int saved\_TCNT1;

saved\_TCNT1=TCNT1;

while((TCNT1-saved\_TCNT1)<(F\_CPU/1000L)); // Wait for 1 ms to pass

}

void waitms(int ms)

{

while(ms--) wait\_1ms();

}

#define PIN\_PERIOD (PIND & 0b00100000)

// GetPeriod() seems to work fine for frequencies between 30Hz and 300kHz.

long int GetPeriod (int n)

{

int i, overflow;

unsigned int saved\_TCNT1a, saved\_TCNT1b;

overflow=0;

TIFR1=1; // TOV1 can be cleared by writing a logic one to its bit location. Check ATmega328P datasheet page 113.

while (PIN\_PERIOD!=0) // Wait for square wave to be 0

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}

}

overflow=0;

TIFR1=1;

while (PIN\_PERIOD==0) // Wait for square wave to be 1

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}

}

overflow=0;

TIFR1=1;

saved\_TCNT1a=TCNT1;

for(i=0; i<n; i++) // Measure the time of 'n' periods

{

while (PIN\_PERIOD!=0) // Wait for square wave to be 0

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return 0;}

}

while (PIN\_PERIOD==0) // Wait for square wave to be 1

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return 0;}

}

}

saved\_TCNT1b=TCNT1;

if(saved\_TCNT1b<saved\_TCNT1a) overflow--; // Added an extra overflow. Get rid of it.

return overflow\*0x10000L+(saved\_TCNT1b-saved\_TCNT1a);

}

int main(void)

{

long int count;

float T, f;

usart\_init(); // Configure the usart and baudrate

DDRD &= 0b11011111; // Configure PD5 as input

PORTD |= 0b00100000; // Activate pull-up in PD5

// Turn on timer with no prescaler on the clock. We use it for delays and to measure period.

TCCR1B |= \_BV(CS10); // Check page 110 of ATmega328P datasheet

waitms(500); // Wait for putty to start

while (1)

{

count=GetPeriod(100);

if(count>0)

{

T=count/(F\_CPU\*100.0);

f=1/T;

printf("f=%fHz (count=%lu) \r", f, count);

}

else

{

printf("NO SIGNAL \r");

}

waitms(200);

}

}

1. **Source Code of ‘AVR\_reciver.c’ which enables reading and writing audio files to the 25Q32 Flash Memory**

/ AVR\_Receiver.c: This program implements a simple serial port

// communication protocol to program, verify, and read SPI flash memories. Since

// the program was developed to store wav audio files, it also allows

// for the playback of said audio. It is assumed that the wav sampling rate is

// 22050Hz, 8-bit, mono.

#include <avr/io.h>

#include <avr/interrupt.h>

#include <stdio.h>

#include "uart.h"

#include <avr/io.h>

#include <util/delay.h>

#define DEF\_FREQ 22050L

#define OCR1\_RELOAD ((F\_CPU/DEF\_FREQ)+1)

/\* Pinout for DIP28 ATMega328P:

(PCINT14/RESET) PC6 (1 28) PC5 (ADC5/SCL/PCINT13)

(PCINT16/RXD) PD0 (2 27) PC4 (ADC4/SDA/PCINT12)

(PCINT17/TXD) PD1 (3 26) PC3 (ADC3/PCINT11)

(PCINT18/INT0) PD2 (4 25) PC2 (ADC2/PCINT10)

(PCINT19/OC2B/INT1) PD3 (5 24) PC1 (ADC1/PCINT9)

(PCINT20/XCK/T0) PD4 (6 23) PC0 (ADC0/PCINT8)

VCC (7 22) GND

GND (8 21) AREF

(PCINT6/XTAL1/TOSC1) PB6 (9 20) AVCC

(PCINT7/XTAL2/TOSC2) PB7 (10 19) PB5 (SCK/PCINT5)

(PCINT21/OC0B/T1) PD5 (11 18) PB4 (MISO/PCINT4)

(PCINT22/OC0A/AIN0) PD6 (12 17) PB3 (MOSI/OC2A/PCINT3)

(PCINT23/AIN1) PD7 (13 16) PB2 (SS/OC1B/PCINT2)

(PCINT0/CLKO/ICP1) PB0 (14 15) PB1 (OC1A/PCINT1)

\*/

// Flash memory commands

#define WRITE\_ENABLE 0x06 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define WRITE\_DISABLE 0x04 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define READ\_STATUS 0x05 // Address:0 Dummy:0 Num:1 to infinite fMax: 32MHz

#define READ\_BYTES 0x03 // Address:3 Dummy:0 Num:1 to infinite fMax: 20MHz

#define READ\_SILICON\_ID 0xab // Address:0 Dummy:3 Num:1 to infinite fMax: 32MHz

#define FAST\_READ 0x0b // Address:3 Dummy:1 Num:1 to infinite fMax: 40MHz

#define WRITE\_STATUS 0x01 // Address:0 Dummy:0 Num:1 fMax: 25MHz

#define WRITE\_BYTES 0x02 // Address:3 Dummy:0 Num:1 to 256 fMax: 25MHz

#define ERASE\_ALL 0xc7 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define ERASE\_BLOCK 0xd8 // Address:3 Dummy:0 Num:0 fMax: 25MHz

#define READ\_DEVICE\_ID 0x9f // Address:0 Dummy:2 Num:1 to infinite fMax: 25MHz

// SPI Flash Memory connections:

// PB0 (pin 14) (MISO) -> Pin 2 of 25Q32

// PB1 (pin 15) (MOSI) -> Pin 5 of 25Q32

// PB2 (pin 16) (SCLK) -> Pin 6 of 25Q32

// PD7 (pin 13) (CSn) -> Pin 1 of 25Q32

// 3.3V: connected to pins 3, 7, and 8

// GND: connected to pin 4

volatile unsigned long int playcnt=0;

volatile unsigned char play\_flag=0;

#define SET\_CS PORTD |= 0b10000000

#define CLR\_CS PORTD &= 0b01111111

#define SET\_MOSI PORTB |= 0b00000010

#define CLR\_MOSI PORTB &= 0b11111101

#define SET\_SCLK PORTB |= 0b00000100

#define CLR\_SCLK PORTB &= 0b11111011

#define MISO\_SET ((PINB & 0b00000001)==0b00000001)

void Setup\_BB\_SPI (void)

{

DDRB|=0b00000110; // PB1 and PB2 are outputs. PB0 is input.

DDRD|=0b10000000; // PD7 is output.

PORTB |= 0b00000001; // Activate pull-up in PB0

SET\_CS;

CLR\_MOSI;

CLR\_SCLK;

}

// Bitbang SPI. Surprisingly fast! SCLK is about 1.25MHz.

unsigned char SPIWrite(unsigned char tx)

{

unsigned char i, rx, mask;

mask=0x80;

rx=0;

for(i=0; i<8; i++)

{

if(tx & mask)

SET\_MOSI;

else

CLR\_MOSI;

SET\_SCLK;

if(MISO\_SET) rx |= mask;

mask>>=1;

CLR\_SCLK;

}

return rx;

}

// 'Timer 1 output compare A' Interrupt Service Routine

ISR(TIMER1\_COMPA\_vect)

{

OCR1A = OCR1A + OCR1\_RELOAD;

PORTD ^= 0b00100000; // Toggle PD5 (pin 11) to check that we have the right frequency

if(play\_flag!=0)

{

if(playcnt==0)

{

SET\_CS; // Done playing: Disable 25Q32 SPI flash memory

play\_flag=0;

}

else

{

OCR0A=SPIWrite(0x00); // Output value to PWM (used as DAC)

playcnt--;

}

}

}

void Init\_pwm (void)

{

DDRD |= (1 << DDD6); // PD6 is now an output (pin 12 of DIP28)

OCR0A = 128; // set PWM for 50% duty cycle

TCCR0A |= (1 << COM0A1); // set none-inverting mode

TCCR0A |= (1 << WGM01) | (1 << WGM00); // set fast PWM Mode

TCCR0B |= (1 << CS00); // set prescaler to none and starts PWM

}

void Init\_Timer1 (void)

{

DDRD|=0b00100000; // PD5 (pin 11) is an output now

TCCR1B |= \_BV(CS10); // set prescaler to Clock/1

TIMSK1 |= \_BV(OCIE1A); // output compare match interrupt for register A

sei(); // enable global interupt

}

void Start\_Playback (unsigned long int address, unsigned long int numb)

{

CLR\_CS; // Select/enable 25Q32 SPI flash memory.

SPIWrite(READ\_BYTES);

SPIWrite((unsigned char)((address>>16)&0xff));

SPIWrite((unsigned char)((address>>8)&0xff));

SPIWrite((unsigned char)(address&0xff));

playcnt=numb;

play\_flag=1;

}

void Enable\_Write (void)

{

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(WRITE\_ENABLE);

SET\_CS; // Disable 25Q32 SPI flash memory

}

void Check\_WIP (void)

{

unsigned char c;

do

{

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(READ\_STATUS);

c=SPIWrite(0x55);

SET\_CS; // Disable 25Q32 SPI flash memory

} while (c&0x01);

}

static const unsigned short crc16\_ccitt\_table[256] = {

0x0000U, 0x1021U, 0x2042U, 0x3063U, 0x4084U, 0x50A5U, 0x60C6U, 0x70E7U,

0x8108U, 0x9129U, 0xA14AU, 0xB16BU, 0xC18CU, 0xD1ADU, 0xE1CEU, 0xF1EFU,

0x1231U, 0x0210U, 0x3273U, 0x2252U, 0x52B5U, 0x4294U, 0x72F7U, 0x62D6U,

0x9339U, 0x8318U, 0xB37BU, 0xA35AU, 0xD3BDU, 0xC39CU, 0xF3FFU, 0xE3DEU,

0x2462U, 0x3443U, 0x0420U, 0x1401U, 0x64E6U, 0x74C7U, 0x44A4U, 0x5485U,

0xA56AU, 0xB54BU, 0x8528U, 0x9509U, 0xE5EEU, 0xF5CFU, 0xC5ACU, 0xD58DU,

0x3653U, 0x2672U, 0x1611U, 0x0630U, 0x76D7U, 0x66F6U, 0x5695U, 0x46B4U,

0xB75BU, 0xA77AU, 0x9719U, 0x8738U, 0xF7DFU, 0xE7FEU, 0xD79DU, 0xC7BCU,

0x48C4U, 0x58E5U, 0x6886U, 0x78A7U, 0x0840U, 0x1861U, 0x2802U, 0x3823U,

0xC9CCU, 0xD9EDU, 0xE98EU, 0xF9AFU, 0x8948U, 0x9969U, 0xA90AU, 0xB92BU,

0x5AF5U, 0x4AD4U, 0x7AB7U, 0x6A96U, 0x1A71U, 0x0A50U, 0x3A33U, 0x2A12U,

0xDBFDU, 0xCBDCU, 0xFBBFU, 0xEB9EU, 0x9B79U, 0x8B58U, 0xBB3BU, 0xAB1AU,

0x6CA6U, 0x7C87U, 0x4CE4U, 0x5CC5U, 0x2C22U, 0x3C03U, 0x0C60U, 0x1C41U,

0xEDAEU, 0xFD8FU, 0xCDECU, 0xDDCDU, 0xAD2AU, 0xBD0BU, 0x8D68U, 0x9D49U,

0x7E97U, 0x6EB6U, 0x5ED5U, 0x4EF4U, 0x3E13U, 0x2E32U, 0x1E51U, 0x0E70U,

0xFF9FU, 0xEFBEU, 0xDFDDU, 0xCFFCU, 0xBF1BU, 0xAF3AU, 0x9F59U, 0x8F78U,

0x9188U, 0x81A9U, 0xB1CAU, 0xA1EBU, 0xD10CU, 0xC12DU, 0xF14EU, 0xE16FU,

0x1080U, 0x00A1U, 0x30C2U, 0x20E3U, 0x5004U, 0x4025U, 0x7046U, 0x6067U,

0x83B9U, 0x9398U, 0xA3FBU, 0xB3DAU, 0xC33DU, 0xD31CU, 0xE37FU, 0xF35EU,

0x02B1U, 0x1290U, 0x22F3U, 0x32D2U, 0x4235U, 0x5214U, 0x6277U, 0x7256U,

0xB5EAU, 0xA5CBU, 0x95A8U, 0x8589U, 0xF56EU, 0xE54FU, 0xD52CU, 0xC50DU,

0x34E2U, 0x24C3U, 0x14A0U, 0x0481U, 0x7466U, 0x6447U, 0x5424U, 0x4405U,

0xA7DBU, 0xB7FAU, 0x8799U, 0x97B8U, 0xE75FU, 0xF77EU, 0xC71DU, 0xD73CU,

0x26D3U, 0x36F2U, 0x0691U, 0x16B0U, 0x6657U, 0x7676U, 0x4615U, 0x5634U,

0xD94CU, 0xC96DU, 0xF90EU, 0xE92FU, 0x99C8U, 0x89E9U, 0xB98AU, 0xA9ABU,

0x5844U, 0x4865U, 0x7806U, 0x6827U, 0x18C0U, 0x08E1U, 0x3882U, 0x28A3U,

0xCB7DU, 0xDB5CU, 0xEB3FU, 0xFB1EU, 0x8BF9U, 0x9BD8U, 0xABBBU, 0xBB9AU,

0x4A75U, 0x5A54U, 0x6A37U, 0x7A16U, 0x0AF1U, 0x1AD0U, 0x2AB3U, 0x3A92U,

0xFD2EU, 0xED0FU, 0xDD6CU, 0xCD4DU, 0xBDAAU, 0xAD8BU, 0x9DE8U, 0x8DC9U,

0x7C26U, 0x6C07U, 0x5C64U, 0x4C45U, 0x3CA2U, 0x2C83U, 0x1CE0U, 0x0CC1U,

0xEF1FU, 0xFF3EU, 0xCF5DU, 0xDF7CU, 0xAF9BU, 0xBFBAU, 0x8FD9U, 0x9FF8U,

0x6E17U, 0x7E36U, 0x4E55U, 0x5E74U, 0x2E93U, 0x3EB2U, 0x0ED1U, 0x1EF0U

};

unsigned short crc16\_ccitt(unsigned char val, unsigned short crc)

{

unsigned short tmp;

tmp = (crc >> 8) ^ val;

crc = ((unsigned short)(crc << 8U)) ^ crc16\_ccitt\_table[tmp];

return crc;

}

// Get a 24-bit number from the serial port and store it into a unsigned long

void get\_ulong(unsigned long \* lptr)

{

unsigned char \* bytes;

bytes=(unsigned char \*) lptr;

bytes[3]=0;

bytes[2]=uart\_getc();

bytes[1]=uart\_getc();

bytes[0]=uart\_getc();

}

int main( void )

{

unsigned char c;

unsigned int j, n;

unsigned long start, nbytes;

unsigned short crc;

uart\_init(); // configure the usart and baudrate

Init\_pwm(); // Initialize the PWM output used as DAC

Init\_Timer1(); // Timer 1 is used as playback ISR

Setup\_BB\_SPI(); // The hardware SPI is not available, so use bitbang SPI instead

playcnt=0;

play\_flag=0;

SET\_CS; // Disable 25Q32 SPI flash memory

while(1)

{

c=uart\_getc();

if(c=='#')

{

playcnt=0;

play\_flag=0;

SET\_CS; // Disable 25Q32 SPI flash memory

c=uart\_getc();

switch(c)

{

case '0': // Identify command

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(READ\_DEVICE\_ID);

c=SPIWrite((unsigned char)(0x00));

uart\_putc(c);

c=SPIWrite((unsigned char)(0x00));

uart\_putc(c);

c=SPIWrite((unsigned char)(0x00));

uart\_putc(c);

SET\_CS; // Disable 25Q32 SPI flash memory

break;

case '1': // Erase whole flash (takes a long time)

Enable\_Write();

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(ERASE\_ALL);

SET\_CS; // Disable 25Q32 SPI flash memory

Check\_WIP();

uart\_putc(0x01);

break;

case '2': // Load flash page (256 bytes or less)

Enable\_Write();

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(WRITE\_BYTES);

c=uart\_getc(); // Address bits 16 to 23

SPIWrite(c);

c=uart\_getc(); // Address bits 8 to 15

SPIWrite(c);

c=uart\_getc(); // Address bits 0 to 7

SPIWrite(c);

n=uart\_getc(); // Number of bytes to write

if(n==0) n=256;

for(j=0; j<n; j++)

{

c=uart\_getc();

SPIWrite(c);

}

SET\_CS; // Disable 25Q32 SPI flash memory

Check\_WIP();

uart\_putc(0x01);

break;

case '3': // Read flash bytes (256 bytes or less)

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(READ\_BYTES);

c=uart\_getc(); // Address bits 16 to 23

SPIWrite(c);

c=uart\_getc(); // Address bits 8 to 15

SPIWrite(c);

c=uart\_getc(); // Address bits 0 to 7

SPIWrite(c);

n=uart\_getc(); // Number of bytes to write

if(n==0) n=256;

for(j=0; j<n; j++)

{

c=SPIWrite(0x55);

uart\_putc(c);

}

SET\_CS; // Disable 25Q32 SPI flash memory

break;

case '4': // Playback a portion of the stored wav file

get\_ulong(&start); // Get the start position

get\_ulong(&nbytes); // Get the number of bytes to playback

Start\_Playback(start, nbytes);

break;

// WARNING: CRC calculation here is way slower than in the EFM8 or SoC-8052.

// Modify computer\_sender.c so it doesn't time out.

// Change line:

// maxwait=length/300000.0;

// to:

// maxwait=length/30000.0;

case '5': ; // Calculate and send CRC-16 of ISP flash memory from zero to the 24-bit passed value.

get\_ulong(&nbytes); // Get the total number of bytes used to calculate the crc

crc=0;

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(READ\_BYTES);

SPIWrite(0x00); // Address bits 16 to 23

SPIWrite(0x00); // Address bits 8 to 1

SPIWrite(0x00); // Address bits 0 to 7

for(start=0; start<nbytes; start++)

{

c=SPIWrite(0x00);

crc=crc16\_ccitt(c, crc); // Calculate CRC here

}

SET\_CS; // Disable 25Q32 SPI flash memory

uart\_putc(crc/0x100); // Send high byte of CRC

uart\_putc(crc%0x100); // Send low byte of CRC

break;

case '6': // Fill flash page (256 bytes or less).

Enable\_Write();

CLR\_CS; // Enable 25Q32 SPI flash memory.

SPIWrite(WRITE\_BYTES);

c=uart\_getc(); // Address bits 16 to 23

SPIWrite(c);

c=uart\_getc(); // Address bits 8 to 15

SPIWrite(c);

c=uart\_getc(); // Address bits 0 to 7

SPIWrite(c);

c=uart\_getc(); // byte to copy to page

for(j=0; j<256; j++)

{

SPIWrite(c);

}

SET\_CS; // Disable 25Q32 SPI flash memory

Check\_WIP();

uart\_putc(0x01);

break;

}

}

}

}

1. **Final program using Colpitts Oscillator, speaker and SPI memory flash**

#define F\_CPU 16000000UL

#include <stdio.h>

#include <stdlib.h>

#include <avr/io.h>

#include <avr/interrupt.h>

#include "usart.h"

#include <util/delay.h>

// Flash memory commands

#define WRITE\_ENABLE 0x06 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define WRITE\_DISABLE 0x04 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define READ\_STATUS 0x05 // Address:0 Dummy:0 Num:1 to infinite fMax: 32MHz

#define READ\_BYTES 0x03 // Address:3 Dummy:0 Num:1 to infinite fMax: 20MHz

#define READ\_SILICON\_ID 0xab // Address:0 Dummy:3 Num:1 to infinite fMax: 32MHz

#define FAST\_READ 0x0b // Address:3 Dummy:1 Num:1 to infinite fMax: 40MHz

#define WRITE\_STATUS 0x01 // Address:0 Dummy:0 Num:1 fMax: 25MHz

#define WRITE\_BYTES 0x02 // Address:3 Dummy:0 Num:1 to 256 fMax: 25MHz

#define ERASE\_ALL 0xc7 // Address:0 Dummy:0 Num:0 fMax: 25MHz

#define ERASE\_BLOCK 0xd8 // Address:3 Dummy:0 Num:0 fMax: 25MHz

#define READ\_DEVICE\_ID 0x9f // Address:0 Dummy:2 Num:1 to infinite fMax: 25MHz

#define SET\_CS PORTD |= 0b10000000

#define CLR\_CS PORTD &= 0b01111111

#define SET\_MOSI PORTB |= 0b00000010

#define CLR\_MOSI PORTB &= 0b11111101

#define SET\_SCLK PORTB |= 0b00000100

#define CLR\_SCLK PORTB &= 0b11111011

#define MISO\_SET ((PINB & 0b00000001)==0b00000001)

#define DEF\_FREQ 22050L

#define OCR1\_RELOAD ((F\_CPU/DEF\_FREQ)+1)

unsigned int cnt = 0;

volatile unsigned long int playcnt=0;

volatile unsigned char play\_flag=0;

// Bitbang SPI. Surprisingly fast! SCLK is about 1.25MHz.

unsigned char SPIWrite(unsigned char tx)

{

unsigned char i, rx, mask;

mask=0x80;

rx=0;

for(i=0; i<8; i++)

{

if(tx & mask)

SET\_MOSI;

else

CLR\_MOSI;

SET\_SCLK;

if(MISO\_SET) rx |= mask;

mask>>=1;

CLR\_SCLK;

}

return rx;

}

// 'Timer 1 output compare A' Interrupt Service Routine

ISR(TIMER1\_COMPA\_vect)

{

OCR1A = OCR1A + OCR1\_RELOAD;

PORTD ^= 0b00100000; // Toggle PD5 (pin 11) to check that we have the right frequency

if(play\_flag!=0)

{

if(playcnt==0)

{

SET\_CS; // Done playing: Disable 25Q32 SPI flash memory

play\_flag=0;

}

else

{

OCR0A=SPIWrite(0x00); // Output value to PWM (used as DAC)

playcnt--;

}

}

}

void wait\_1ms(void)

{

unsigned int saved\_TCNT1;

saved\_TCNT1=TCNT1;

while((TCNT1-saved\_TCNT1)<(F\_CPU/1000L)); // Wait for 1 ms to pass

}

void waitms(int ms)

{

while(ms--) wait\_1ms();

}

#define PIN\_PERIOD (PIND & 0b00100000)

// GetPeriod() seems to work fine for frequencies between 30Hz and 300kHz.

long int GetPeriod (int n)

{

int i, overflow;

unsigned int saved\_TCNT1a, saved\_TCNT1b;

overflow=0;

TIFR1=1; // TOV1 can be cleared by writing a logic one to its bit location. Check ATmega328P datasheet page 113.

while (PIN\_PERIOD!=0) // Wait for square wave to be 0

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}

}

overflow=0;

TIFR1=1;

while (PIN\_PERIOD==0) // Wait for square wave to be 1

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>5) return 0;}

}

overflow=0;

TIFR1=1;

saved\_TCNT1a=TCNT1;

for(i=0; i<n; i++) // Measure the time of 'n' periods

{

while (PIN\_PERIOD!=0) // Wait for square wave to be 0

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return 0;}

}

while (PIN\_PERIOD==0) // Wait for square wave to be 1

{

if(TIFR1&1) { TIFR1=1; overflow++; if(overflow>1024) return 0;}

}

}

saved\_TCNT1b=TCNT1;

if(saved\_TCNT1b<saved\_TCNT1a) overflow--; // Added an extra overflow. Get rid of it.

return overflow\*0x10000L+(saved\_TCNT1b-saved\_TCNT1a);

}

void Init\_pwm (void)

{

DDRD |= (1 << DDD6); // PD6 is now an output (pin 12 of DIP28)

OCR0A = 128; // set PWM for 50% duty cycle

TCCR0A |= (1 << COM0A1); // set none-inverting mode

TCCR0A |= (1 << WGM01) | (1 << WGM00); // set fast PWM Mode

TCCR0B |= (1 << CS00); // set prescaler to none and starts PWM

}

void Init\_Timer1 (void)

{

DDRD|=0b00100000; // PD5 (pin 11) is an output now

TCCR1B |= \_BV(CS10); // set prescaler to Clock/1

TIMSK1 |= \_BV(OCIE1A); // output compare match interrupt for register A

sei(); // enable global interupt

}

void Setup\_BB\_SPI (void)

{

DDRB|=0b00000110; // PB1 and PB2 are outputs. PB0 is input.

DDRD|=0b10000000; // PD7 is output.

PORTB |= 0b00000001; // Activate pull-up in PB0

SET\_CS;

CLR\_MOSI;

CLR\_SCLK;

}

void Start\_Playback (unsigned long int address, unsigned long int numb)

{

CLR\_CS; // Select/enable 25Q32 SPI flash memory.

SPIWrite(READ\_BYTES);

SPIWrite((unsigned char)((address>>16)&0xff));

SPIWrite((unsigned char)((address>>8)&0xff));

SPIWrite((unsigned char)(address&0xff));

playcnt=numb;

play\_flag=1;

}

int main(void)

{

long int count,countbase;

float T, f,basef,baseT;

usart\_init(); // Configure the usart and baudrate

DDRD &= 0b11011111; // Configure PD5 as input

PORTD |= 0b00100000; // Activate pull-up in PD5

// Turn on timer with no prescaler on the clock. We use it for delays and to measure period.

TCCR1B |= \_BV(CS10); // Check page 110 of ATmega328P datasheet

waitms(500); // Wait for putty to start

countbase=GetPeriod(100);

if(countbase>0)

{

baseT=countbase/(F\_CPU\*100.0);

basef=1/baseT;

printf("Base f=%f",basef);

}

while (1)

{ DDRD &= 0b11011111; // Configure PD5 as input

PORTD |= 0b00100000; // Activate pull-up in PD5

count=GetPeriod(100);

if(count>0)

{

T=count/(F\_CPU\*100.0);

f=1/T;

printf("f=%fHz (count=%lu) \n\r", f, count);

if(f>basef+40 && f<basef+150) //small non ferrous

{

Init\_pwm(); // Initialize the PWM output used as DAC

Init\_Timer1(); // Timer 1 is used as playback ISR

Setup\_BB\_SPI(); // The hardware SPI is not available, so use bitbang SPI instead

playcnt=0;

play\_flag=0;

Start\_Playback(0x06500,0x008000);

while(play\_flag);

}

else if(f>basef+150) //large non ferrous

{

Init\_pwm(); // Initialize the PWM output used as DAC

Init\_Timer1(); // Timer 1 is used as playback ISR

Setup\_BB\_SPI(); // The hardware SPI is not available, so use bitbang SPI instead

playcnt=0;

play\_flag=0;

Start\_Playback(0x014000,0x007500);

while(play\_flag);

}

else if(f<basef-250) //large ferrous

{

Init\_pwm(); // Initialize the PWM output used as DAC

Init\_Timer1(); // Timer 1 is used as playback ISR

Setup\_BB\_SPI(); // The hardware SPI is not available, so use bitbang SPI instead

playcnt=0;

play\_flag=0;

Start\_Playback(0x0010000,0x005000);

while(play\_flag);

}

else if(f<basef-40 && f>basef-230) //small ferrous

{

Init\_pwm(); // Initialize the PWM output used as DAC

Init\_Timer1(); // Timer 1 is used as playback ISR

Setup\_BB\_SPI(); // The hardware SPI is not available, so use bitbang SPI instead

playcnt=0;

play\_flag=0;

Start\_Playback(0x000000,0x006800);

while(play\_flag);

}

}

else

{

printf("NO SIGNAL \r");

}

waitms(200);

}

}