MIE438 Final Report

Audio Encryption System

Date Submitted

April 15, 2016

Team Members

Zain Nasrullah (998892685)

Nauman Sohani (999296701)

Abdullah Siddiqui (998971830)

1. Project Summary

1.1. Original Proposal: Haptic Feedback Collision Detection

The original design concept involved simulating a vehicle collision system using proximity sensors and wirelessly transmitting the sensory data to the user. However, this idea placed heavy emphasis on designing the vehicle which is a rather standard project. Furthermore, the sensing aspect of the project seemed to be a minor addition rather than the focus of the design.

1.2. Revised Project: Audio-Based Security System

With the intention of maintaining wireless communication between devices, the second iteration of the project involved creating an encrypted security system that can be operated remotely. To create a complex key for our device and distinguish it from similar products on the market, we opted for an audio-based solution that will take advantage of tones. Using a Fast Fourier Transform (FFT)[1], it is possible to deconstruct an audio signal into its constituent frequencies and use a pre-determined pattern for authentication. The goal of this project was to ultimately use audio as a means of remote authentication.

1.3. Further Iteration: Enhanced Audio-Based Security System

With the completion of initial project goals, we decided to add complexity in the form of a more secure key. The original design sampled audio, performed the FFT/FHT and authenticated on the transmitter side sending only the open command to the receiver. This posed significant security concerns because it is theoretically possible to send the open command to a receiver through any transmitter. To address this issue, we decided to authenticate on the receiver side using a pattern of predetermined frequencies, which would constitute a personal identification number (PIN). If the data sent from the transmitter matched a user-defined PIN, then the receiver would proceed with authentication and unlock the door, so to speak. This ultimately led to two additional considerations:

1. Embedding high-pitched frequencies into an audio signal to obfuscate the actual key.
2. Allowing the user to set the key on the receiver side using a number pad.

Our implementation used time-delays between a predetermined set of three high-pitched frequencies, nearly undetectable to the adult human ear while a song is playing, as the PIN. More complex PINs would require (n+1) tone detections, where n is the number of digits within the PIN. Figure 1.3.1 illustrates a ‘black-box diagram’ of the input-output transformations the embedded system would perform.

[1] A Fast Hartley Transform (FHT) was used in the final implementation and is discussed in the Detailed Program Design.

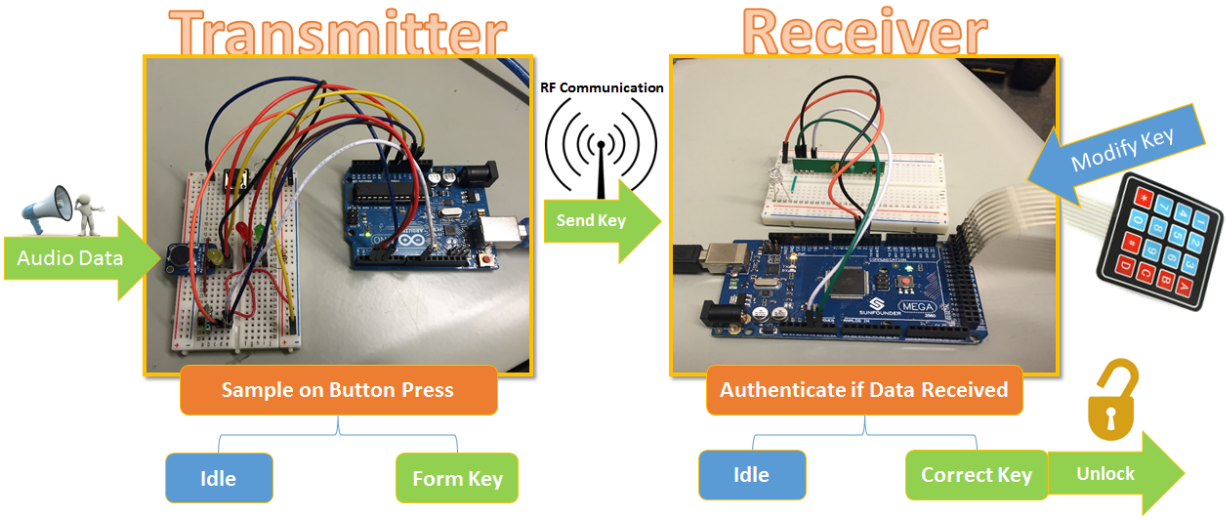

Figure 1.3.1: Black-box Diagram of Input-Output Transformations

2. Hardware

This section comprehensively discusses the hardware used in our project and the engineering rationale that supports each selection.

2.1. Final Microcontroller Specifications

Two devices were required to create a remote audio encryption system as given below.

Table 2.1.1: Summary of Microcontrollers Selected

Device & µC Roles	Relevant Arduino Specifications	Unique Specifications
Arduino Uno R3 ATmega328P [1] FHT & Transmitter	• 16 MHz Clock Speed • 8-Bit Memory/Register Width • ADC (10-bit Resolution) • 5V Operating Voltage [5V & 3.3V Pins Available] • AnalogReference (AREF) Pin	• 32KB Flash Memory w/ 1KB EEPROM • 6 Analog Inputs • 14 Digital I/O Pins (6 PWM)
Arduino MEGA 2560 ATmega2560 [2] Receiver & Authentication		• 256KB Flash Memory w/ 4KB EEPROM • 16 Analog Inputs • 54 Digital I/O Pins (15 PWM)

The two boards differ in terms of the number of pins available for input/output and memory sizes available for storage. However, in terms of performance, both devices possess the same critical specifications (clock speed, bit width & ADC resolution) [3].

2.2. Rationale Behind μC Selection

The Arduino, as an entry-level device, offered many features relevant to audio sampling at a low cost:

1. To obtain a maximum frequency near 20 kHz (upper-end of human perception) in our FFT, it is necessary to sample at twice the bandwidth (44 kHz) as per the Nyquist-Shannon sampling criterion [4]. Since the Arduino's ADC clock is set to 16 MHz, it is possible to set the sampling rate by modifying the prescaler factor (default 128) in the *ADCSRA* register by setting bits 0, 1 & 2 (11100**101**) [5][6]. 101 enables a clock prescale factor of 32. Since a single conversion requires 13 clock cycles, this yields:

$$\frac{16\ MHz}{32\ Prescale\ Factor} = 500\ Hz\ Clock\ Speed$$

$$\frac{500\ Hz}{13\ Instructions\ per\ Cycle} = 38.4\ kHz\ Sampling\ Rate$$

   The result corresponds to an audio bandwidth of 19.2 kHz which is near project requirements.

   Notes

   - Using a 38.4 kHz sampling rate with 256 bins corresponds to a frequency resolution of:

$$\frac{38.4\ kHz}{256\ Bins} = 150\ Hz\ per\ Bin$$

   - It is advisable to use a bit depth of 16 for characterizing the sound because this corresponds to 2^16 or 65,546 levels of dynamic range (amplitude) in the audio as opposed to a mere 256 levels with a bit depth of 8. An Arduino FFT library takes 7 ms to sample 256 bins of data at 16-bit depth while the FHT library requires only 3 ms due to code optimizations [7].

2. Arduino's built-in ADC possess two additional relevant features: free-running mode and analog referencing. The former enables automatic sample conversation once the ADC is initiated which is useful for monitoring continuous sensory data [8]. For our project, where real-time sampling of auditory data is required, this feature is essential. Free-running mode can be enabled by setting bit 5 in the *ADCSRA* register (11**1**00101) [8]. The second relevant feature, analog referencing, improves the accuracy of the conversion. Since the ADC operates at a 10-bit resolution, it maps the analog signal to 2^10 or 1024 unique values ranging between 0 and 1023 [9]. This is an important consideration because using a reference of 5V (default) will yield a resolution of 5V/1024 = 4.9mV. Since Arduino boards possess a 3.3V output pin and the selected microphone (discussed below) optimally performs at this voltage as well, it should be used as a reference instead of the board's default operating voltage. At 3.3V, one would expect a 3.3V/1024 = 3.2 mV resolution which corresponds to approximately 35% improved accuracy in converted data.

Notes

- The maximum ADC clock frequency in the ATMEGA328 is limited to 200 kHz by the internal Digital to Analog Convertor (DAC) in the conversion circuitry [10]. Operating beyond this speed, as we are in this case, may diminish the 10-bit ADC resolution (see Appendix A) [11]. However, manufacturer notes and experimental data suggest that the diminished accuracy is insignificant prior to 1 MHz for most applications [10][12].

3. The program required only 2 bytes of data for storage of key on the receiver. We used the EEPROM (Electronically Erasable Programmable Read-Only Memory) memory on board the Arduino Mega microcontroller. All Arduino and Genuino boards offer at least 512 bytes of EEPROM memory which allows the storage of values even when the board is turned off [13]. For our receiver program, the memory needs to hold the key on the receiver board unless changed by the user and retrieve it when verifying the transmission. To access the EEPROM memory of the microcontroller, the library *EEPROM.h* was included in the sketch.

4. Arduino also allows for modular integration of numerous hardware components with pre-existing libraries for support. Consequently, code complexity was minimized. We implemented readily available components such as a microphone, keypad, on/off switch, transmitter and receiver modules with ease into our program. They were integrated seamlessly by including the appropriate libraries in the sketch. Please refer to section 2.3 for hardware considerations.

5. Many hobbyists select Arduino boards for their DIY (do it yourself) projects as they are easy to use, implement and find. We had Arduino boards available from previous projects and thus we were familiar with the Arduino family of microcontrollers.

These features made a compelling case for selecting an Arduino as the basis of our design.

2.3. Board Selection

Upon selecting an Arduino-based solution, the next decision was to select particular models from the product line. We initially opted for two Leonardos for this project, since the clock-speed and ADC capabilities do not vary significantly between boards [14], and these boards were readily available to us. However, we ran into performance and compatibility issues with both devices (this product line has been discontinued [15]) prompting us to switch to more recent Uno & MEGA models.

Holistically, this project does not have many hardware requirements; consequently, we also did not require many input and output pins on either board given that only a microphone, transmitter, receiver, touchpad & few LEDs were used. It is thus possible to use two Uno devices which is more cost-effective than the implementation used in our project. However, we were able to obtain those boards freely and thus chose to use what was most readily available at the time.

There are two potential benefits for using a MEGA board as a receiver. Firstly, it became possible to connect the touchpad - which requires 8 sequential pins due to the wired connections - directly to the board without infringing on the TX and RX pins. Of course, it is

possible to circumvent issue by connecting non-sequential pins (on an Uno) to a breadboard with additional cabling. Secondly, the MEGA possess larger memory which could be used to store more data on the receiver side in future iterations of the design. This change can add to customizability and security by allowing the user to set more complex passcodes that utilize a combination of expected frequency, tone length, delay between tones, and number of tones as authentication. Moreover, additional memory could even be used to develop user profiles and store different passwords for unique users or for different situations; e.g. leaving the house for a walk may merit a shorter password, whereas leaving for a two-week vacation may merit a longer or more complex password. With that said, the Uno has 1 kB of EEPROM memory which is already sufficient for this application.

2.4. Other Hardware Considerations

The four main components used for this project were:

Microphone [16]

The microphone used was the Electret Microphone module with adjustable gain (25x to 125x) to capture audio ranging between 20 Hz and 20 kHz. This was a low cost microphone designed for audio-reactive projects (such as FFT) supporting VCC voltages between 2.4-5V with ideal performance at 3.3V using an Arduino board. The microphone features a MAX4466 op-amp with built-in power supply noise rejection; consequently, it can be implemented directly into the microcontroller's ADC pin without additional filtering or amplification. The library used to integrate this mic was *FHT.h*.

RF Transmitter/Receiver [17][18]

Radio Frequency was used to transmit and receive data from one board to another. The minimum requirements for our project was to wirelessly transmit data one-way to show proof of concept of the FHT-based passcode. We used 434 MHz band transmitter and receiver modules which have 500 ft range, data rate of 4800 bps and 5 V supply voltage. RF Transmitter/Receiver modules are very cost efficient and fulfilled the minimum requirements. Wireless communication between Arduino boards gets costly once we get into Bluetooth or WiFi, requiring the use of appropriate shields. The library used to achieve wireless communication was *RCSwitch.h*. The weakness of RCSwitch is that it can only send one piece of data at a time. An alternative library that we could have used is the *VirtualWire.h* which can send an array of data continuously. Ultimately, the number of instructions used to transmit and receive is what determined the selection of library as RCSwitch uses less lines of code than VirtualWire, making the code easy to read.

4x4 Keypad [19]

The keypad supports 16 buttons arranged in a 4x4 matrix. Consequently, it requires 8 digital pins corresponding to row and column values. Each entry in this matrix corresponds to one of 16 characters stored within the code (digits 0-9, *, #, A, B, C, D). The keypad was used to put the Receiver Arduino in either receiver mode or password change mode. In receiver mode, the Receiver board displays the received timestamps of the tones on the serial monitor. The password change mode allows the user to set the password using the numbers on the keypad. To integrate the keypad into the program, we included the *Keypad.h*.

On/Off Switch

The On/Off switch was used to control when to sample for the FFT/FHT. The program samples only when the switch is pressed.

LEDs

To give indication whether the FHT passcode is correct or not. A red LED is used on the transmitter to indicate when sampling is being done and when the passcode is wrong. A green LED is needed to indicate a correct passcode. On the receiver side, there is a blue LED to indicate the passcode has been received.

3. Detailed Design

3.1. Final Design Overview and User Operation

As shown in Figure 1.3.1, a user initiates the system operation by holding the sample button on the transmitter to activate the system sampling mode. While holding this button down, the user plays an audio file embedded with the high-frequency tone password. The sampling button was added as an additional optimization measure: the user would presumably know which segment of an audio file contained the high-frequency tones and thus the system would only sample during this portion of the song. Extraneous and unneeded sampling, which could adversely affect the system's reliability, are avoided.

The audio input is captured via a microphone. The microphone amplifies the incoming signal to an adequate level and consequently no additional hardware was required or implemented to sense the analog audio signal. The analog audio signal is then processed through the 10-bit ADC native to the Arduino Uno and sampled and deconstructed for its frequency content using the FHT. In each iteration of the FHT, 256 samples are collected and processed - the size of the FHT was selected to minimize lag caused by the sampling sequence while maintaining a reasonable sample size to extract meaningful and reliable frequency information of the input signal. Results collected during testing (Section 4) corroborate an adequate FHT. The sampling operation is discussed in more detail in Section 3.2. Once the system has detected three high-frequency tones in the audio signal, the time delays between each high-frequency tone in the audio signal are transmitted to the receiver for authentication.

The receiver itself has two modes of operation, which should be configured prior to operating on the transmitter. The recommended default mode is accessed by selecting 'A' on the keypad upon receiver startup. In this 'receive mode,' the receiver constantly polls for data from the transmitter. Once the receiver obtains the time offsets sent by the transmitter, verification is performed against the PIN stored in the EEPROM. Upon verification, the receiver LED is activated, representing an unlocked door. By selecting 'B' on the keypad, the user accesses the 'PIN mode.' In this mode, the user has the option to save a new PIN to the EEPROM. Implementing PIN authentication and indeed PIN modification on the receiver is safe protocol since the receiver would presumably be housed in a secure location.

An overview of the hardware layout is provided in Appendix B. User operation and general program structure are captured schematically in Appendix C.

3.2. FHT Implementation

The Fast Hartley Transform is a core feature of this audio-based security system. The FHT falls under the family of Fourier-related transforms and seeks to produce the same type of data, i.e. frequency-domain representation of a signal. However, it is specifically constructed for operating on real inputs and thus can be performed with fewer computations [7]. For the purposes of this system, the FHT library was adapted from [7] and modified as necessary. In particular, [7] called for disabling 'timer0' to reduce jitter; however, this conflicted with the use of the millis() timekeeping function, which was integral to the functionality of the system. Consequently, timer0 was left enabled - testing proved that the output quality did not suffer appreciably. The FHT library makes extensive use of the ADCSRA register, which is responsible for enabling analog-to-digital conversion and interrupt control (Figure 3.2.1) [6].

Bit	7	6	5	4	3	2	1	0
ADCSRA	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
Read / Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initial Value	0	0	0	0	0	0	0	0

Figure 3.2.1: ADCSRA register [6]

Prior to sampling, this register is originally given the hexadecimal value of 0xE5, which along with controlling the pre-scalar factor (Section 2.2.1), sets the ADC enable, ADC start conversion, and ADC free-running mode. The interrupt flag (ADIF) is originally set to 0 - this bit is set to 1 by hardware once the ADC is complete and the relevant data registers are updated. The ADIF bit is actually manually cleared by setting it to 1 instead of 0 [6].

During the sampling sequence, the ADIF flag of the register is compared with 0x10 in an empty while loop to essentially wait for the ADC to be ready. Once this condition is met, the ADSRA is reset to the hexadecimal value of 0xF5 to preserve the pre-scale factor and appropriate bit-setting. The converted data is fetched and constructed into a 16-bit signed integer after bit-shifting and byte concatenation. This process is repeated an additional 255 times. Note that bit 3 in ADCSRA, which corresponds to the interrupt enable, remains 0 throughout the program execution: if bit 3 and the I-bit in SREG are set to 1, then the ADC conversion complete interrupt is activated [6]. Raising this interrupt unpredictably would impede program execution and thus ADC data is only retrieved when the sampling sequence is activated.

Once 256 samples are collected, the fht_window(); fht_reorder(); fht_run(); and fht_mag_log(); functions are called sequentially from the FHT.h library. These functions perform the optimized FHT to ascertain the frequency content of the sampled signal. The output of the transform, fht_log_out[256], can be written to the serial to display using an external spectrum analyzer. This was a particularly useful feature to visually understand where the spectral power in the input was concentrated. The program was written in a high-level C-like language to leverage the existing libraries. As the FHT was already highly optimized, there was not much to be gained from writing in assembly. Moreover, high-level C made it easier for us to evaluate where changes needed to be made due to familiarity with the language structure and syntax (Appendices D and E for code).

4. Results

4.1. Summary of Data

The user input and password-saving features operated as expected. Upon setting a new passcode, the data is stored and recovered from memory irrespective of resetting the board (due to use of EEPROM) (Figure 4.1.2a). The results of the FHT were monitored in an external spectrum analyzer (Figure 4.1.1).

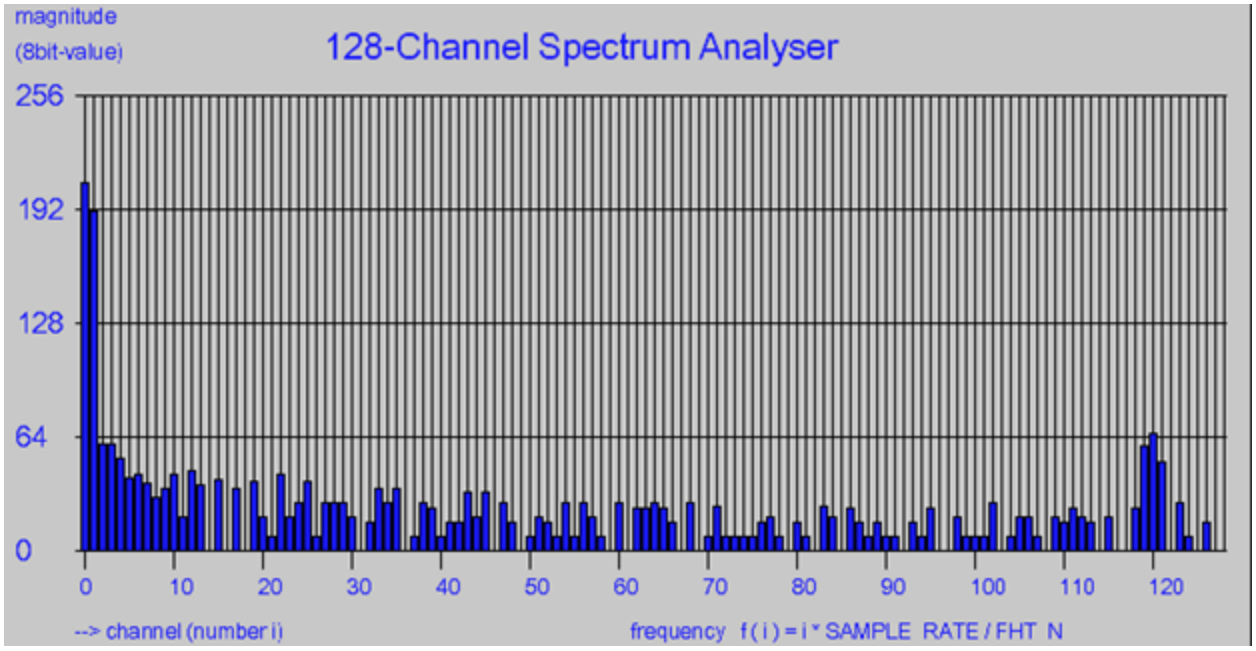

Figure 4.1.1: Sample output of FHT

In the figure above, ambient noise can be observed at all frequency levels but a high pitched frequency in the 18 kHz range (bin 120) can be clearly distinguished. This testifies to the feasibility of using high-pitched frequencies embedded into comparatively low-pitched songs—see Appendix F for the FHT graph of 18.5 and 19 kHz. Transmitting all 3 tones with the correct delays (do nothing on incorrect key) to the receiver yields Figure 4.1.2b & 4.1.2c.

The received times, 6008 & 7990 ms, correspond to the expected password of 6s & 8s (success shown via a blue LED rather than a serial print out) within the set tolerance of 500ms. A ±10 ms accuracy is observed using the millis function due to minimizing the number of actions prior to saving these values to an array. This is much higher than expected suggesting that there is still room for improvement in terms of setting the tolerances.

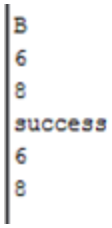

Figure 4.1.2a

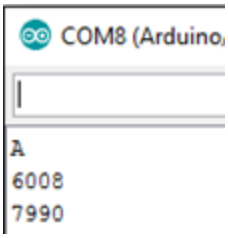

Figure 4.1.2b

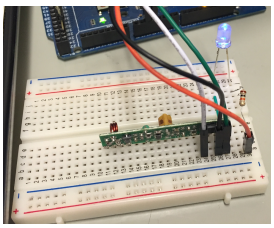

Figure 4.1.2c

4.2. Analysis of Microcontroller Selection

In order to assess the merits of the microcontroller and boards selected for this system, we need to consider to what extent the microcontroller allowed us to satisfy design objectives.

At the core of this project, the embedded system is required to analyze the frequency content of an audio signal consistently and accurately. Doing so requires sampling at ~40 kHz to avoid aliasing effects. Consequently, we needed to determine whether the Arduino board could perform an FHT without maximizing its resources, and how many samples were needed to reliably represent the frequency content of our signal. As evidenced by the experimental data, high frequencies in the range of 18-19 kHz were detected at the predicted bins between 120 and 126. Moreover, this FHT required 256 samples and 3 ms to produce the output as shown in Figure 4.1.1. As the current set of passwords operated on delays on the order of seconds, an execution time of 3 ms was an acceptable level of granularity. In fact, for testing and validation purposes, the high frequency tones were intentionally held for 2 s to ensure that we could detect them and verify that the system was performing as expected. The system capability is evidently much greater and we could conceivably make the tones much shorter, and thus less perceptible, to leverage the high processing capability of the Arduino. Moreover, the Arduino FHT operated on the principle of 16-bit depth for characterizing the sound as discussed in Section 2.2.1. While 32-bit depth would add even more resolution into the result of Figure 4.1.1, it would require much more processing power without any significant benefit particularly because 16-bit depth was already proven to be effective in detecting the high-frequency tones. 8-bit depth, however, would prevent this level of detection and would adversely affect the system reliability. The results and testing corroborate that the Arduino frequency content analysis was completely adequate for the current implementation of the design. In fact, it can be expanded to implement even more complex passwords in future iterations of the system.

One particular challenge with using the libraries already developed for Arduino was competition surrounding timers. In particular, the FHT implementation called for disabling timer0 to reduce jitter associated with the transform. However, disabling this timer disrupted the millis() function, which was used to track the offsets between the high-frequency tones. Tone time-tracking was a more critical consideration than smooth FHT results and therefore, the timer was left active and the FHT results were tested to verify reliability. This could possibly be circumvented by using an external timer or different microcontroller, but we suspect that other microcontrollers would encounter similar challenges; therefore, this was not considered an egregious penalty against the Arduino board. Challenges associated with transmitting the data through RF were considered more of an issue with the library rather than a problem isolated with the

microcontroller. In fact, subsequent iterations of the design, which can make use of more robust and secure transmission methods such as Wi-Fi or Bluetooth pair well with Arduino because of readily available hardware and software libraries. From this perspective, the Arduino IDE is a major benefit of using the microcontrollers selected for this design.

On the receiver portion of the design, software support for the keypad was also readily available, allowing us to easily integrate the customizable PIN feature. While this integration ability is not necessarily limited to the Arduino platform - for example, the Dragon12-P microcontroller evaluation boards running on the Freescale 68HC12 also incorporate a keypad - we were confident that the Arduino platform could be further developed with other modules, such as Bluetooth or Wi-Fi for the communication capability without much obstruction. Moreover, the physical size of the development board was a consideration as well, as the Dragon12-P was generally excessive for the demands of this project. Arguably, the Arduino boards were also excessive in some respects; for example, the simple set of sensors used in this system did not require the abundant analog and digital I/O pins. Nevertheless, the selected microcontroller proved adequate to satisfy the objectives of this design. Indeed, the Arduino platform, i.e. the microcontroller and IDE, offer numerous tools to expand the capabilities of this design, as discussed in Section 5. We would use this microcontroller again, particularly because it is inexpensive, adequately powerful for what we seek to accomplish, and is backed by tremendous development support.

5. Recommendations

As mentioned earlier in the report, the objective of this project was to show the concept of decoding a pass code consisting of embedded high frequency tones in a sound file by using FHT. Moving forward, there are some areas of improvement and the potential to add key features that will make the design more secure and simpler for the end user.

The passcode can be encrypted, thereby adding an extra layer of security. This would make it harder for any person to breach. One way of implementing this is by using a cryptographic algorithm whereby the passcode goes through a hash function to map it to some hash value [20]. Unless the real key is known, one cannot invert the hash value to the original key [20].

Moreover, a more secure wireless transmission protocol can be used instead of RF which is not the most secure communication as one can easily tune to the correct frequency band. Bluetooth technology can also be used to securely pair transmitting and receiving devices [21].

Additionally, it was observed while testing sessions that high frequency tones which are in the audible range were very unpleasant to the ear and noticeable. An ultra-high frequency device can be used to generate short length higher frequency tones to make them inaudible in combination to the audio file. Consequently, the prescale factor would need to be reduced to get a smaller sampling frequency. Additionally, this design would require a new microphone, as the current microphone is rated for up to 20 kHz. We will need to create an elegant solution get the right information without having too much delay. A possibility is to conduct an initial sampling operation to set the thresholds based on the ambient noise. Lastly, we did not exploit the full capability of the microcontroller, which we can utilize to generate more complicated passwords and allow higher customizability by the user.

6. References

[1] "Arduino - ArduinoBoardUno", *Arduino.cc*, 2016. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardUno

[2] "Arduino - ArduinoBoardMega2560", *Arduino.cc*, 2016. [Online]. Available: https://www.arduino.cc/en/Main/arduinoBoardMega2560

[3] "Arduino vs Arduino Mega – Which To Use? | Utopia Mechanicus", *Utopiamechanicus.com*, 2012. [Online]. Available: http://www.utopiamechanicus.com/article/arduino-versus-arduino-mega-which-to-use/

[4] "Sample Rates - Audacity Wiki", *Wiki.audacityteam.org*, 2015. [Online]. Available: http://wiki.audacityteam.org/wiki/Sample_Rates

[5] "FHTExample - Open Music Labs Wiki", *Wiki.openmusiclabs.com*, 2012. [Online]. Available: http://wiki.openmusiclabs.com/wiki/FHTExample

[6] "Robot Platform | Knowledge | ADC Registers 1", *Robotplatform.com*. [Online]. Available: http://www.robotplatform.com/knowledge/ADC/adc_tutorial_2.html

[7] "ArduinoFHT - Open Music Labs Wiki", *Wiki.openmusiclabs.com*, 2014. [Online]. Available: http://wiki.openmusiclabs.com/wiki/ArduinoFHT

[8] "Design Note #021", *AVRFreaks.net*, 2002. [Online]. Available: https://cours.etsmtl.ca/ele542/labo/ref-stk500-atmega32/AppNote%2021%20--%20Using%20the%20ADC.pdf

[9] "Welcome To AVRbeginners.net!", *Avrbeginners.net*. [Online]. Available: http://www.avrbeginners.net/architecture/adc/adc.html

[10] "AVR120: Characterization and Calibration of the ADC on an AVR", *Atmel*, 2006. [Online]. Available: http://www.atmel.com/Images/doc2559.pdf

[11] F. Alferink, "Arduino analog measurements :: Electronic Measurements", *Meettechniek.info*, 2013. [Online]. Available: http://meettechniek.info/embedded/arduino-analog.html

[12] G. Berg, "Advanced Arduino ADC – Faster analogRead()", *Microsmart.co.za*, 2014. [Online]. Available: http://www.microsmart.co.za/technical/2014/03/01/advanced-arduino-adc/

[13] "Arduino - EEPROMWrite", *Arduino.cc*, 2015. [Online]. Available: https://www.arduino.cc/en/Tutorial/EEPROMWrite

[14] "Arduino - ArduinoBoardLeonardo", *Arduino.cc*, 2016. [Online]. Available: https://www.arduino.cc/en/Main/ArduinoBoardLeonardo

[15] J. Payne, "Arduino reduce their product range!", *Paynetech.co.uk*, 2015. [Online]. Available: https://www.paynetech.co.uk/arduino-discontinued

[16]"Electret Microphone Module - Adjustable Gain", *Creatron Inc*. [Online]. Available: https://www.creatroninc.com/product/electret-microphone-module-adjustable-gain/?search_query=microphone+arduino&results=5

[17] "434MHz RF Link Transmitter", *Creatron Inc*. [Online]. Available: https://www.creatroninc.com/product/434mhz-rf-link-transmitter/?search_query=transmitter&results=19

[18] "434MHz RF Link Receiver", *Creatron Inc*. [Online]. Available: https://www.creatroninc.com/product/434mhz-rf-link-receiver/?search_query=receiver&results=49

[19] "4x4 Keypad Membrane", *Creatron Inc*, 2016. [Online]. Available: https://www.creatroninc.com/product/4x4-keypad-membrane

[20] "An Introduction to Cryptographic Hash Functions | Dev-HQ: Other Tutorial", Dev-hq.net, 2012. [Online]. Available: http://www.dev-hq.net/posts/4--intro-to-cryptographic-hash-functions

[21] "Comparison of RF and Bluetooth | DigiKey", *Digikey.ca*, 2010. [Online]. Available: http://www.digikey.ca/en/articles/techzone/2010/dec/comparison-of-rf-and-bluetooth

7. Appendices

Appendix A: Effect of Prescale Factor on Resolution

The following graph shows results from Analog Read at clock speeds modified by varying prescale factors[11]:

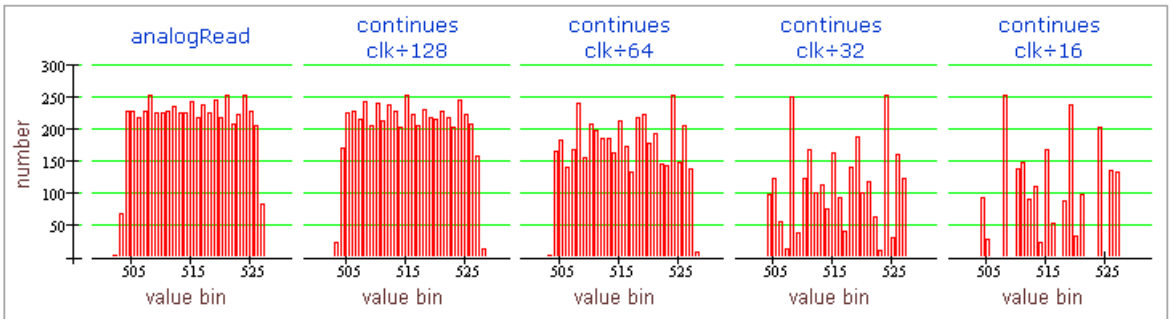

Figure A1: Digital Reading for Varying Clock Speeds

As mentioned, running a clock speed greater than 200 kHz (or using a prescale factor smaller than 32) shows diminished resolution. However, other data suggests that sampling up to 1 MHz (prescale factor of 16) is possible yielding samples every 20 microseconds [12]:

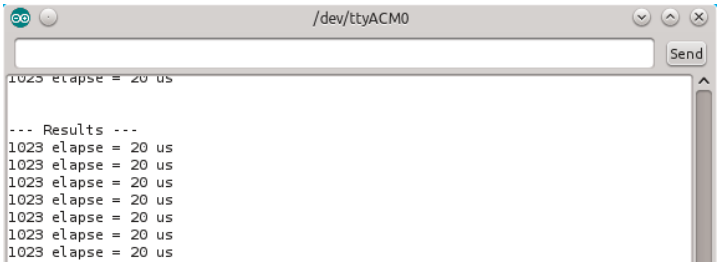

Figure A2: Sampling at Prescale Factor of 16

The author of this study suggests the implementor should attempt the implementation and determine themselves whether the accuracy is acceptable given the application.

Appendix B: Breadboard Layouts

Transmitter

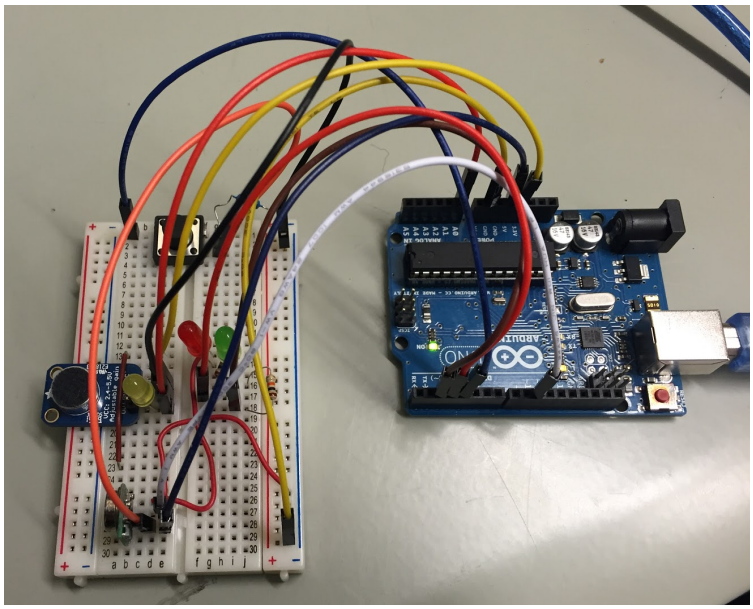

Figure A3: Transmitter board layout

Receiver

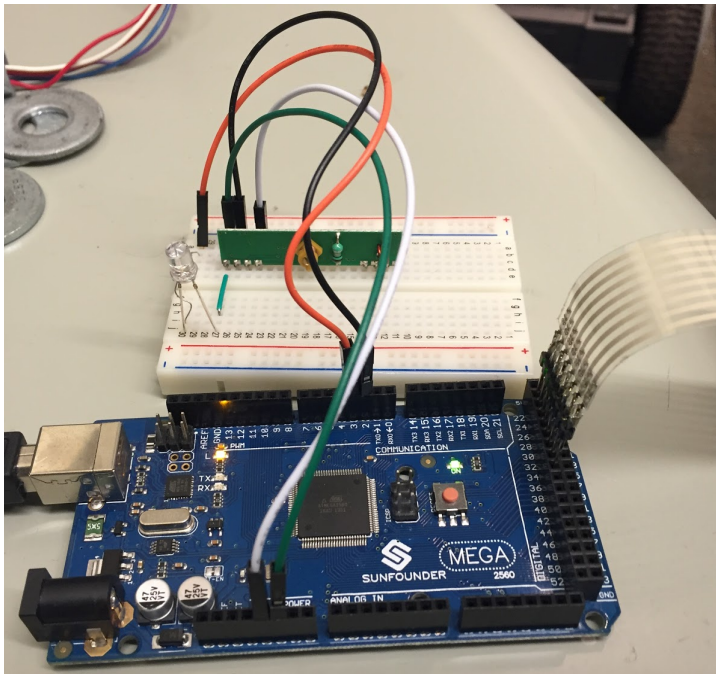

Figure A4: Receiver board layout

Appendix C: Program Flow

Note that user inputs to the system are given in blue boxes and system outputs are given orange boxes:

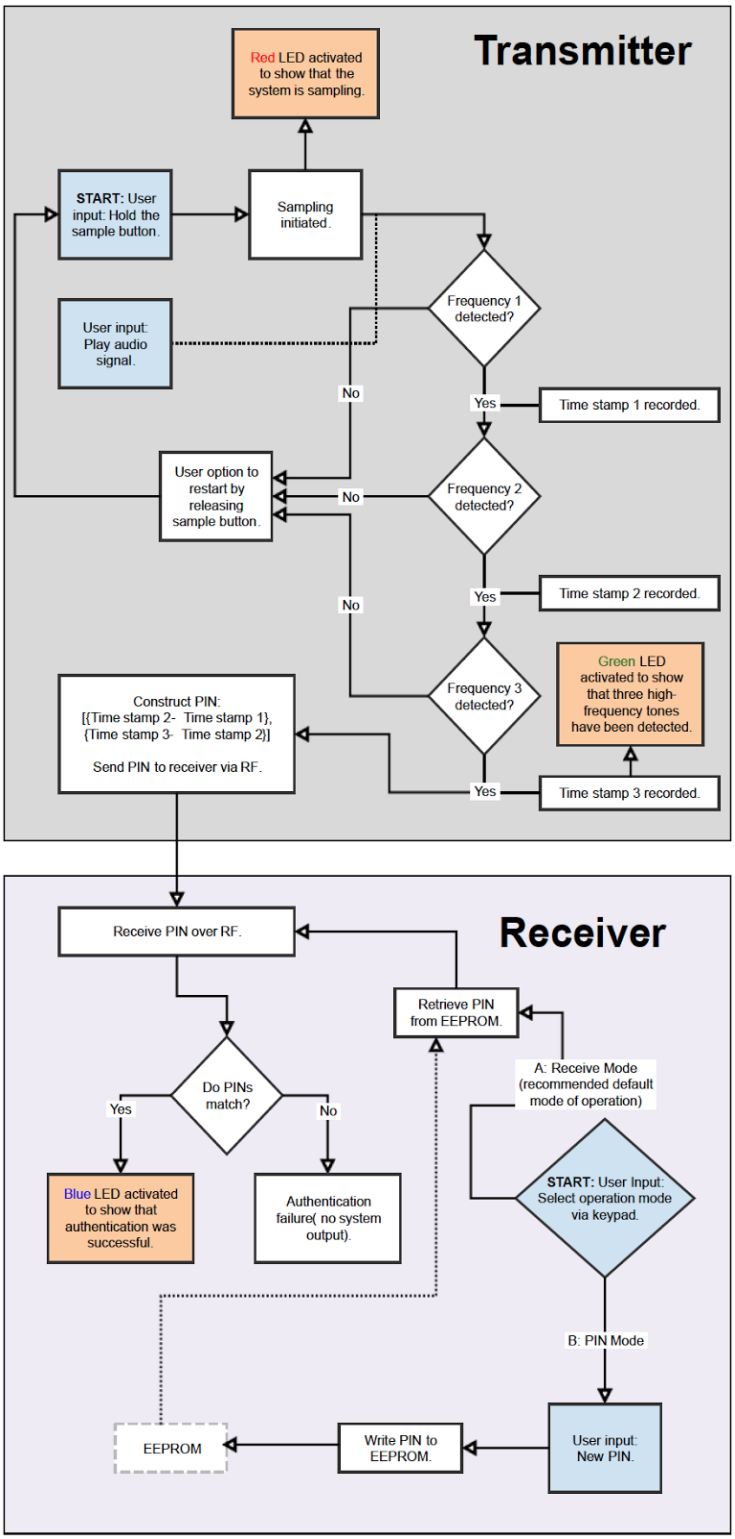

Figure A5: Schematic of System Operation and Program Logic

Appendix D: Uno (Transmitter) Code

```
/* Things that this program does:
  - if the sample button is held down, then the program will sample until the button is
released or the time_offset array is filled;
  - once the time_offset array has been populated, a second loop is used to calculate the
deltas between each time offset;
    these offsets are transmitted to the receiver.
*/

#define LOG_OUT 1      // use the log output function
#define FHT_N 256      // use a 256 point FHT (--> max frequency bin will be 128)
#define THRESHOLD 50   // threshold for bin value
#define NO_TONES 3     // number of tones

// array containing bin numbers of each tone
int tone_array[NO_TONES] = {120, 123, 127};

// array containing time stamps when tones occur; initialize to zero for safety
unsigned long time_offset[NO_TONES] = {0, 0, 0};

// array containing data to be sent to receiver
int time_delta[NO_TONES - 1] = {0, 0};

// constructed pin sent over transmitter
unsigned long pin;


#include <FHT.h>
#include <RCSwitch.h>

// switch controlling the Radio Control
RCSwitch mySwitch = RCSwitch();

void setup()
{
  Serial.begin(115200); // use the serial port
  ADCSRA = 0xe5; // set the adc to free running mode
  ADMUX = 0x40; // use adc0
  DIDR0 = 0x01; // turn off the digital input for adc0

  mySwitch.enableTransmit(10);
  //mySwitch.setRepeatTransmit(4);
  pinMode(4, INPUT); //sample button
  pinMode(3, OUTPUT); //fail pw
  pinMode(2, OUTPUT); //successful pw
  //pinMode(5, OUTPUT); //RESET switch
}
```

```
/* function to perform the FHT. The FHT is performed using a sample of 256 values that are
fetched from the analog-to-digital conversion native to Arduino. This sampled value, after
bit-shifting and byte concatenation, is formed into a 16-bit signed integer which is stored in
the fht_input[] array. This array is passed into the relevant fht functions. Data can be
optionally written to the serial for visualization using an external spectrum analyzer. */
void sample()
{
  // will take one instance of 256 samples
  for (int i = 0 ; i < FHT_N ; i++) { // save 256 samples
    while (!(ADCSRA & 0x10)); // wait for adc to be ready
    ADCSRA = 0xf5; // restart adc
    byte m = ADCL; // fetch adc data
    byte j = ADCH;
    int k = (j << 8) | m; // form into an int
    k -= 0x0200; // form into a signed int
    k <<= 6; // form into a 16b signed int
    fht_input[i] = k; // put real data into bins
  }
  fht_window(); // window the data for better frequency response
  fht_reorder(); // reorder the data before doing the fht
  fht_run(); // process the data in the fht
  fht_mag_log(); // take the output of the fht
  Serial.write(255); // send a start byte
  Serial.write(fht_log_out, FHT_N / 2); // send out the data
}

/* main loop function. Sampling is initiated upon holding the sampling button. Upon
successfully detecting the high-frequency tones, time offsets between each detection are
calculated and transmitted to the receiver. Multiple attempts are made to transmit the values
to ensure that the values successfully reach the receiver. Repeated values are filtered out at
the receiver. */
void loop()
{
  // variable to control entries in time_offset
  int i = 0;

  // loop while the sample button is HIGH and we haven't fully populated time_offset
  while (digitalRead(4) == HIGH && i < NO_TONES)
  {
    digitalWrite(3, HIGH);
    fht_log_out[tone_array[i]] = 0;
    while (digitalRead(4) == HIGH && fht_log_out[tone_array[i]] < THRESHOLD)
    {
      sample();
    }
    time_offset[i] = millis();
    i++;
  }
  digitalWrite(3, LOW);
  //Serial.println(i);
```

```
    if (i == NO_TONES)
    {
      // construct pin to send
      for (int j = 0; j < (NO_TONES - 1); j++)
      {
        // ensure individual pin entries are slightly different for processing on receiver
        if (j > 0 && time_delta[j] == time_delta[j - 1])
        {
          time_delta[j]++;
        }
        time_delta[j] = time_offset[j + 1] - time_offset[j]; // Calculate Time Delays (Key)
        Serial.println(time_delta[j]);
        // send each entry multiple times in case sending fails on any single event
        for(int k = 0; k<10; k++)
        {
          mySwitch.send(time_delta[j], 24);  //RCSwitch Function for Transmitting an Integer
        }
      }
    if (i == NO_TONES)
    {
      Serial.println (time_offset[0]);
      Serial.println (time_offset[1]);
      Serial.println (time_offset[2]);
      Serial.println (time_delta[0]);
      Serial.println (time_delta[1]);
      digitalWrite(2, HIGH); //Blink Green LED for 5 seconds
      delay(5000);
      digitalWrite(2, LOW);
      }
  }
}
```

Appendix E: MEGA (Receiver) Code

```
/*receiver*/

#define NO_TONES 3

//array containing data received from transmitter
int delta_received[NO_TONES - 1] = {
  0, 0};

// array user-defined PIN; values adjusted based on retrieval from EEPROM
int key[NO_TONES - 1] = {
  7000,8000}; //include EEPROM values here
int i = 0;

// relevant libraries required for receiver operation
#include <RCSwitch.h>
#include <Keypad.h>
#include <EEPROM.h>

// define switch for receiving
RCSwitch mySwitch = RCSwitch();

// addresses and values of each 'digit' of the PIN
int interval_1_add = 0;
int interval_2_add = 2;
int interval_1;
int interval_2;

// configure keypad
const byte ROWS = 4; //four rows
const byte COLS = 4; //four columns
//define the cymbols on the buttons of the keypads
char hexaKeys[ROWS][COLS] = {
  {'1','2','3','A'},
  {'4','5','6','B'},
  {'7','8','9','C'},
  {'*','0','#','D'}
};
byte colPins[COLS] = {28,26,24,22}; //connect to the row pinouts of the keypad
byte rowPins[ROWS] = {36,34,32,30}; //connect to the column pinouts of the keypad

//initialize an instance of class NewKeypad
Keypad customKeypad = Keypad( makeKeymap(hexaKeys), rowPins, colPins, ROWS, COLS);

void setup() {
  Serial.begin(9600);
  pinMode(3, OUTPUT); //successful pw
  mySwitch.enableReceive(0);  // Receiver on interrupt 0 => that is pin #2
}

/* ReceiveMode is one of the main modes of operation of the receiver. It can be accessed by
pressing 'A' on the keypad upon receiver initialization. In this mode, the PIN digits are read
from EEPROM and stored to key[]. Received data is filtered to eliminate extraneous and
repeating values (repeating values may occur because of multiple attempts to send on the
transmitter to ensure that the value was transmitted).

The received values are compared with data in key[] and upon matching to a tolerance of +/-500
ms, the LED on pin 3 was set to HIGH signalling successful authentication. */
void ReceiveMode()
```

```
{
  key[0] = EEPROM.read(interval_1_add);
  key[0] = key[0]*1000;
  key[1] = EEPROM.read(interval_2_add);
  key[1] = key[1]*1000;
  int flag = 0;
  for(i = 0; i < (NO_TONES - 1); i++)
  {
        if (mySwitch.available())
        {
        delta_received[i] = mySwitch.getReceivedValue();
        }
        if (delta_received[i] == 0)
        i--;
        if (i!=0 && delta_received[i] == delta_received[i-1])
        i--;
        mySwitch.resetAvailable();
  }
  //Serial.println(flag);

  for(i = 0; i< (NO_TONES - 1); i++)
  {
   //Serial.println(delta_received[i]);
        if(delta_received[i] < key[i]-500 || delta_received[i] > key[i] + 500)
        flag = 1;
  }

  if (flag == 0) {
        Serial.println(delta_received[0]);
        Serial.println(delta_received[1]);
        digitalWrite(3, HIGH);
        delay(5000);
        digitalWrite(3, LOW);
        delay (5000);
        //i = 0;
  }
}

/* keypadReset, also known as PIN mode, allows the user to change the PIN on the receiver.
This mode can be accessed by selecting ‘B’ on the keypad upon receiver startup. Upon entering
PIN mode, the current PIN is written to the serial. The keypad enters into a polling loop to
sequentially obtain the PIN digits. PIN digits are entered by selecting one value and then
inputting to the system using ‘A.’

The current PIN digits are restricted to values between 3 and 9 (inclusive). This is because
current delays in the tones are ~2s in length for testing purposes. This function also checks
to ensure that each PIN digit meets these requirements; upon successfully meeting the
requirements for each pin digit, these PIN digits are stored in the address locations for
interval_1 and interval_2 on EEPROM. If the requirements are not met, then the PIN digits are
set to 5 and 5 as this was one of our default passcodes - this can be essentially considered a
hard reset although for a final product, a sensible solution would be to revert the PIN digits
to their previous values.*/
void keypadReset(){

  int temp;
  interval_1 = EEPROM.read(interval_1_add);
  interval_2 = EEPROM.read(interval_2_add);
   Serial.println(interval_1);
   Serial.println(interval_2);
//Delay 1
  char keyEntry1 = customKeypad.getKey();
```

```
  char keyEntry2 = customKeypad.getKey();
do {
  while(!keyEntry1)
  {
        keyEntry1 = customKeypad.getKey();  //Enter Interval 1 Value
  }
  keyEntry2 = customKeypad.getKey();
  while(!keyEntry2)
  {
        keyEntry2 = customKeypad.getKey();  //Press Confirmation Key for Interval 2 (Character
A)
  }
} while (keyEntry2 != 'A');

  if (keyEntry2 == 'A'){
        interval_1 = keyEntry1 - '0';
        //temp = temp*1000;
   // Serial.println(interval_1);
  }
//End of Interval 1

// Interval 2
  keyEntry1 = customKeypad.getKey();
 do {
  while(!keyEntry1)
  {
        keyEntry1 = customKeypad.getKey(); // Enter Interval 2 Value
  }
  keyEntry2 = customKeypad.getKey();
  while(!keyEntry2)
  {
        keyEntry2 = customKeypad.getKey(); //Enter Confirmation Key for Interval 2 (Character
B)
  }
 } while (keyEntry2 != 'B');
  if (keyEntry2 == 'B'){
        interval_2 = keyEntry1 - '0';
        //temp = temp*1000;
        //Serial.println(interval_2);
  }

if (interval_1 > 2 && interval_1 < 10 && interval_2 > 2 && interval_2 < 10)
  {
        Serial.println("success");
  }
else if ((interval_1 <= 2 || interval_1 >9) && (interval_2 > 2 && interval_2 < 10))
{
  Serial.println("fail");
  interval_1 = 5;
}
else if ((interval_2 <= 2 || interval_2 >9) && (interval_1 > 2 && interval_1 < 10))
{
   Serial.println("fail");
  interval_2 = 5;
}
else
{
   Serial.println("fail");
  interval_1 = 5;
  interval_2 = 5;
}
```

```
interval_1 = interval_1;
interval_2 = interval_2;
 Serial.println(interval_1);
 Serial.println(interval_2);
 EEPROM.write(interval_1_add, interval_1);
 EEPROM.write(interval_2_add, interval_2);
}

/* main loop for the receiver. The receiver initiates in a loop to detect for a character
selection on the keypad.

Upon selecting ‘A,’ the receiver enters into receiveMode in which it continuously polls for
data from the transmitter. This is also the function that performs authentication.

Upon selecting ‘B’, the receiver enters into PIN or keypad mode in which the user can change
the PIN digits. These digits are stored in EEPROM at the designated address locations.*/
void loop()
{
  char keyEntry = customKeypad.getKey();
  //do {
  while(!keyEntry)
  {
        keyEntry = customKeypad.getKey();  //Press Confirmation Key for Interval 2 (Character
A)
  }
//} while (keyEntry != 'A' || keyEntry != 'B');
Serial.println(keyEntry);
if (keyEntry == 'A')
  ReceiveMode();
else if (keyEntry == 'B')
 keypadReset();
  }
```

Appendix F: Complete Frequency Results

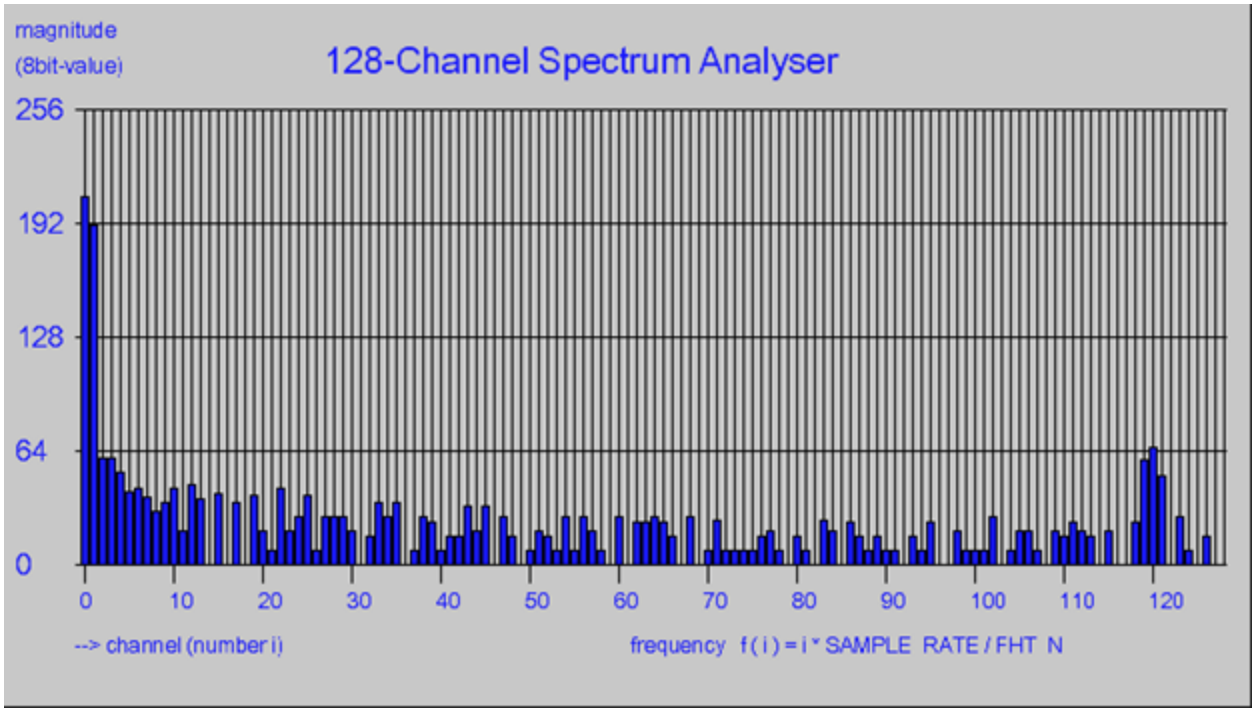

Figure A6: First Peak of 18 kHz @ Bin 120

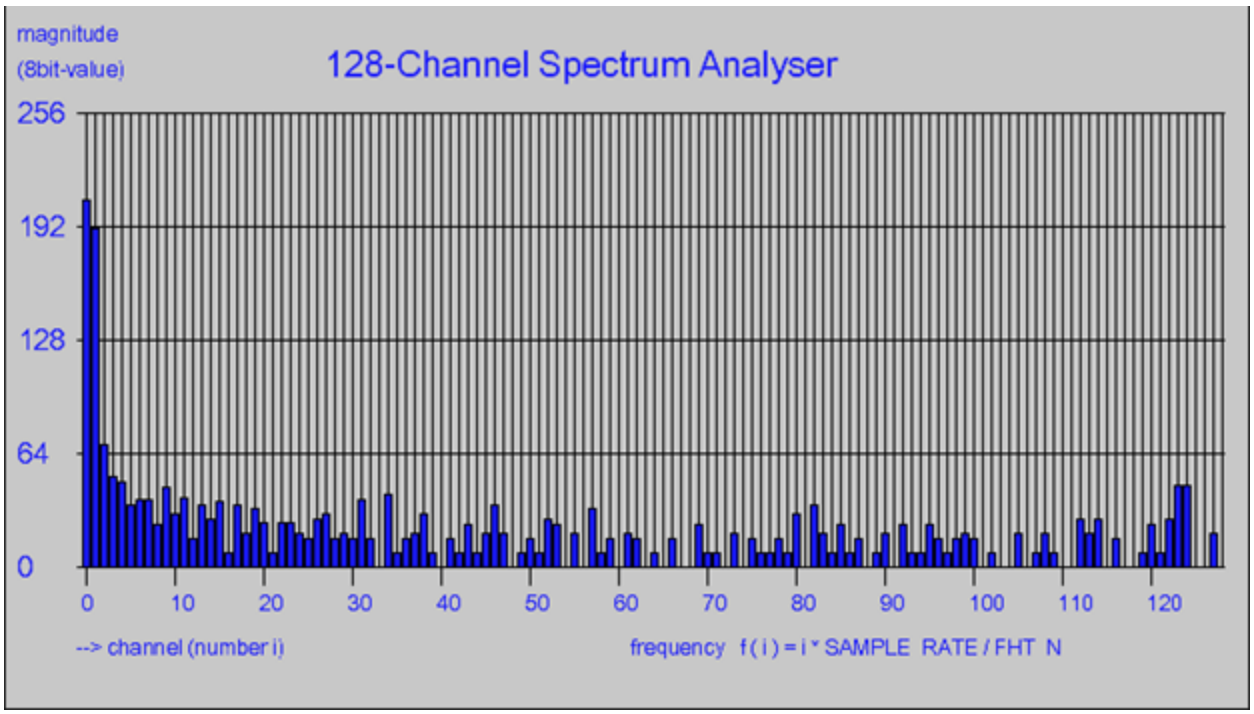

Figure A7: Second Peak of 18.5 kHz @ Bin 123

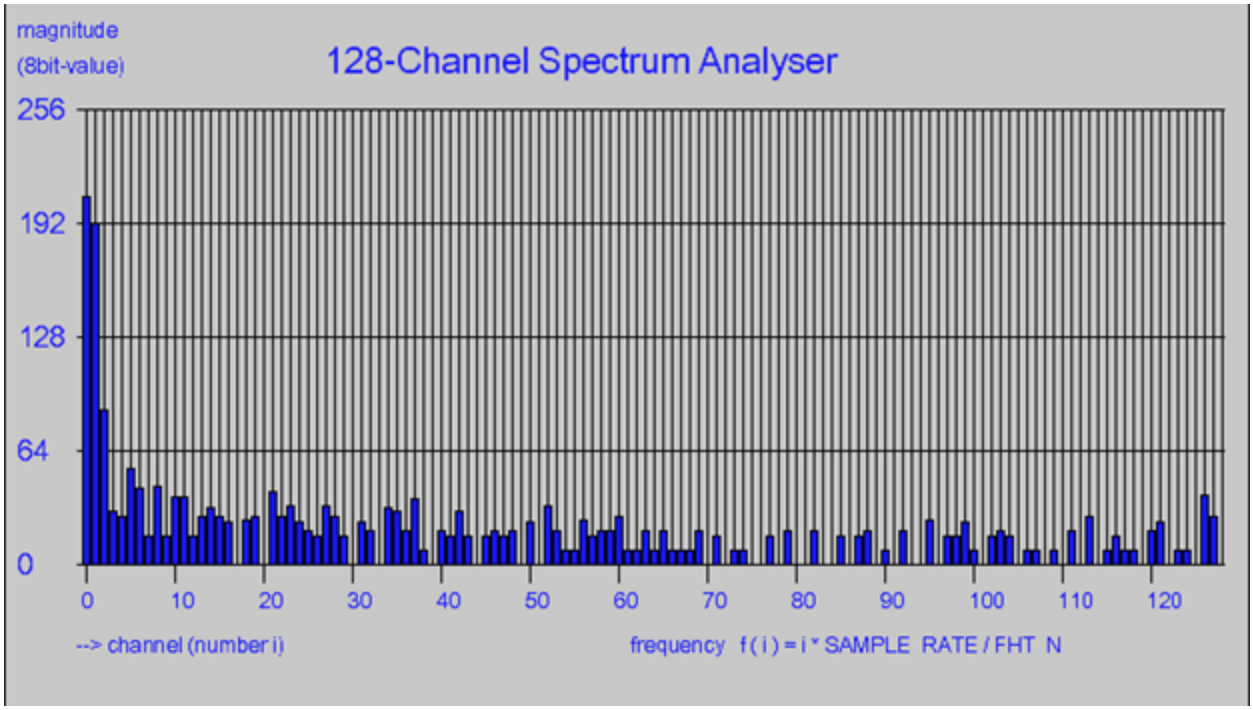

Figure A8: Third Peak of 19 kHz @ Bin 127