SILENT COMMUNICATION DEVICE

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Electrical Engineering

by

Christopher W. Schutter

March 2017

COMMITTEE MEMBERSHIP

TITLE:	Silent Communication Device
AUTHOR:	Christopher W. Schutter
DATE SUBMITTED:	March 2017
COMMITTEE CHAIR:	Jane Zhang, Ph.D. Professor of Electrical Engineering, Associate Dept. Chair, Graduate Coordinator
COMMITTEE MEMBER:	Tina Smilkstein, Ph.D. Associate Professor of Electrical Engineering
COMMITTEE MEMBER:	Jeffery Gerfen, M.S. Lecturer of Electrical Engineering

ABSTRACT

Silent Communication Device

Christopher W. Schutter

Oral communication has constituted as a necessary aspect of how people interact with one another, but there are always situations where this form of communication can create distractions, irritation, or even danger. Take for example, a student in a laboratory who needs to communicate effectively with a lab partner without creating a distraction to those trying to work around said student or a soldier on a battlefield who needs to relay information effectively to his or her comrades without revealing his or her position to the enemy. It becomes apparent that people need a more exclusive form of communication in order to ensure not only the safety of soldiers, but efficiency in the workplace as well.

This project focuses on solving these problems by developing a small, concealable, and non-invasive, electronic device capable of transmitting communication silently by linking to a phone, computer, or radio channel. This device ensures completely silent communication between only those who use communicating devices and only requires that the user apply nodes to his or her throat when thinking of what he or she wishes to communicate with another for proper operation. Unlike other devices which rely on EEG and thus involve cumbersome headwear, this device performs as easily removable, concealable, hands free, conveniently pocket-sized, compatible with other devices used for communication, and able to have a user input versus device output accuracy of at least 70%.

Using wavelet analysis and a MSP432 microcontroller, subvocal signals originating from the throat can be classified to an overall accuracy of at least 70% within a project budget of $50.

ACKNOWLEDGMENTS

I would like to thank the committee for giving me the opportunity to present this thesis. I would like to thank Jane Zhang for the support and advice she gave me throughout this project, Tina Smilkstein for the advice and materials for this project, and Jeffrey Gerfen for the support as well. I would also like to thank Travis Harper for the plate capacitors.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1. INTRODUCTION

This chapter covers the problem that the thesis is trying to address, the motivation behind its creation, and the initial design of thesis' project.

1.1 Statement of problem

Distractions in the workplace act as a rampant problem that affect productivity and the focus of workers. On average, office workers get interrupted every three minutes, many of these distractions coming from "loud colleagues" in which getting back on task after said interruptions takes 23 minutes [14]. With the increase in use of mobile phones in the workplace, these distractions only get worse and increase in number.

In order to solve these problems and increase work productivity, a type of vocal communication needs to exist that does not make any audible noise. Fortunately, aspects of the human body exist that allow people to communication without having to speak or make any significantly audible noise. One of such aspects involves a phenomenon of the human body known as subvocalization or subvocal speech. Subvocal speech is a phenomenon in which, when a person thinks of words or phrases, his or her human vocal cords create small vibrations that emit these as sound back into the brain for the person to hear as his or her own thoughts [6].

Organizations such as NASA (NASA release 04-093) [13] and various colleges such as the Fed. Univ. of Maranhao [3] have made attempts at making vocal communication silent by making use of subvocal speech by using EEG medical electrodes and small vibration detectors to detect and read such signals. Unfortunately, most of the solutions are large, expensive, and uncomfortable for the user. To fully make use of silent communication to efficiently reduce the significant amount of loss of production in the workplace due to distractions, a device that provides such communication needs to be made that is convenient and affordable enough to be mass produced.

This is where the Silent Communication Device Project comes in play. By making use of subvocal speech, this device provides the ability to communicate silently through attaching electrode nodes to the user’s throat in which the user simply has to think of what he or she wishes to say to communicate silently without the need for surgery in order to attach nodes inside the user’s body [3]. The user’s voice is then sent to an electronic device of the user such as a cellphone, computer, or radio transmitter, which can then be sent to an electronic device of the intended recipient. The device accomplishes this function while being affordable, small enough to be carried in the user’s pocket, and comfortable.

The use of this device not only reduces distractions in the workplace and thus increases productivity, but also provide a means for soldiers in the military to have a method of communication that nearby enemies cannot overhear. In addition, it allows people to communicate confidential information without having to waste time finding a place in which others cannot hear them, thus saving production efficiency, time, and even lives.

Similar devices have been made as shown in projects such as [23] and [33] in which said projects employ wavelet analysis (explained in much greater detail later in chapter 3) in order to effectively process subvocal speech to allow said subvocal speech to be more easily classified as its intended speech for the final output. The wavelet analysis allows these devices to classify subvocal speech up to accuracies between 70% and 80%, but only through the use of expensive and not pocket-sized equipment. While these devices use Matlab programming on laptops to perform their wavelet analysis and classification, recent technology such as the MPS432P401R microcontroller show the potential to perform said wavelet analysis and classification through much cheaper and dimensionally smaller means. Thus, in order to prove that it can compete with these devices with similar performance under a much more constrained budget and smaller dimensions, the Silent Communication Device should, in theory, be able to classify subvocal speech with similar accuracy through use of the MSP432P401R microcontroller and wavelet analysis (it should be noted that some projects such as [3] use more advanced processing methods such as independent component analysis to gain classification accuracy up to 95%, but are too complex in terms of processing power for a microcontroller to handle). Therefore, through the creation of the Silent Communication Device, this thesis aims to not only create a device that addresses the aforementioned problems of the workplace, but also prove that subvocal speech originating from the throat can be classified to an overall accuracy of at least 70% within a project budget of $50.

1.2 Related Work

As previously mentioned, the motivation before this project was inspired papers such as ‘‘Electro-myographic patterns of sub-vocal Speech: Records and classification,” by L. E. Mendoza, J. P. Rodríguez, and J. L. R. Valencia [23] and “Sub-vocal Phoneme-Based EMG Pattern Recognition and its application in Diagnosis,” by M. Jahan and M.

Khan [33] which feature the extraction of EMG subvocal signals from the throat, amplification and filtering of said signals to allow them to be large enough in amplitude to be accurately sampled without noise, wavelet analysis to extract features from the EMG signals, and classification using a neural network in order to classify the signals to accuracies between 70% to 80%, showing a relatively simple way of translating subvocal speech into a range of words, vowels, or phonemes with decent performance. In addition, the paper "MSP430 Implementation of Wavelet Transform for Purposes of Physiological Signals Processing," by R. Stojanović and S. Knežević [31] demonstrates an example of a microcontroller being used to employ wavelet analysis on signals extracted from the human body, showing that wavelet analysis through use of a microcontroller is possible, while the paper "Subvocal Speech Recognition Based on EMG Signal Using Independent Component Analysis and Neural Network MLP," by J.A.G. Mendes [3] demonstrates a way of amplifying and filtering subvocal speech taken from the throat that could easily be reproduced as a cheap prototype breadboard circuit. Although other means of extracting features from subvocal signals exist such as using Independent Component Analysis instead of wavelet analysis [3] that give much more accurate classification results, these methods are much more complex than that of wavelet analysis and thus would require much more complex programs to implement them, which microcontrollers are unlikely to be able to process without at least very slow or even poor performance, while [31] has proven that wavelet analysis works with microcontrollers effectively. Combining the wavelet analysis through use of a microcontroller [31] (using the more advanced microcontroller MSP432P401R instead of a MSP430 microcontroller), the amplification and filtering circuit of [3], and the subvocal speech translation methods of [23] and [33], the Silent Communication Device could be brought to fruition.

1.3 Initial Design and Block Diagrams

The initial design of the Silent Communication Device first needs to have its inputs and outputs defined. The device takes its inputs from the user's vocal cords as a subvocal signal, power from a battery, and user control to allow the user to change settings for the device such as the volume of the output, the type of output given (e.g. an output with or without wavelet analysis), and the ability to control when the device takes its input. The device's output is the user's voice, translated from the subvocal signals coming from the user's throat, which is tramsitted into an attached electronic device (e.g. cellphone). Figure 1.1 below shows the level 0 block diagram.

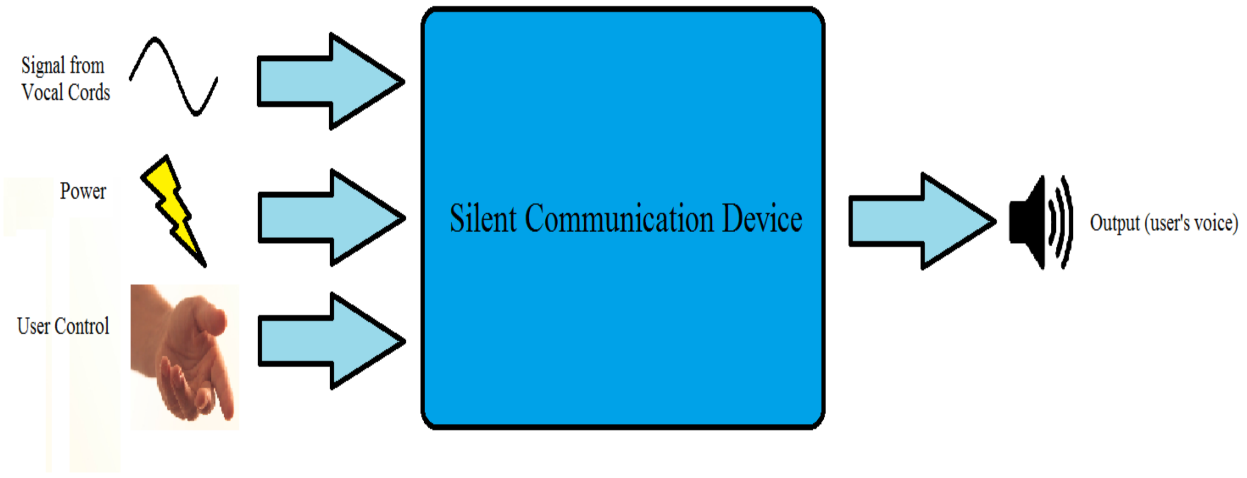


Figure 1.1: Level 0 Block Diagram

Table 1.1: Level 0 Block Diagram Description Table

Table 1.1 below describes the inputs, outputs, and functionality of the level 0 block diagram.

Type	Name	Description
Input	Signal from Vocal Cords	Input received from the vibration of the user's vocal cords [6].
Input	Power	Power received from battery.
Input	User control	Input from user for desired settings for the device.
Output	User's Voice	User's voice transmitted electronically to cell phone, computer, etc.
Overall functionality	The silent communication device uses signals produced from the vibration of the user's vocal cords to output the user's voice electronically to an attached device. The battery powers the device in which the device operates according to the user's determined settings.	

After having its basic inputs and outputs, a more detailed block diagram of the Silent Communciation Device can be created. The Silent Communication Device uses surface EMG electrodes to extract the input subvocal waveform, an amplifier/filter to obtain, amplify, and filter the input EMG waveform (10 Hz to 450 Hz), a wavelet transform function to deal with sources of noise (discussed in greater detail in chapter 3), a neutral network classification system to properly identify and classify each input from the wavelet transform function to its proper output (example: classifies each signal by the letter or vowel it is supposed to represent), and finally a data USB interface to translate the resulting output data to a proper USB format in order to interact with a laptop or smart phone. The design is made to take into account the many different types of noise that an EMG signal experiences as well as the level of classification needed to properly correlate and match each and every English letter uttered by the human vocal cords to its appropriate output. Figure 1.2. below shows the level 1 block diagram.

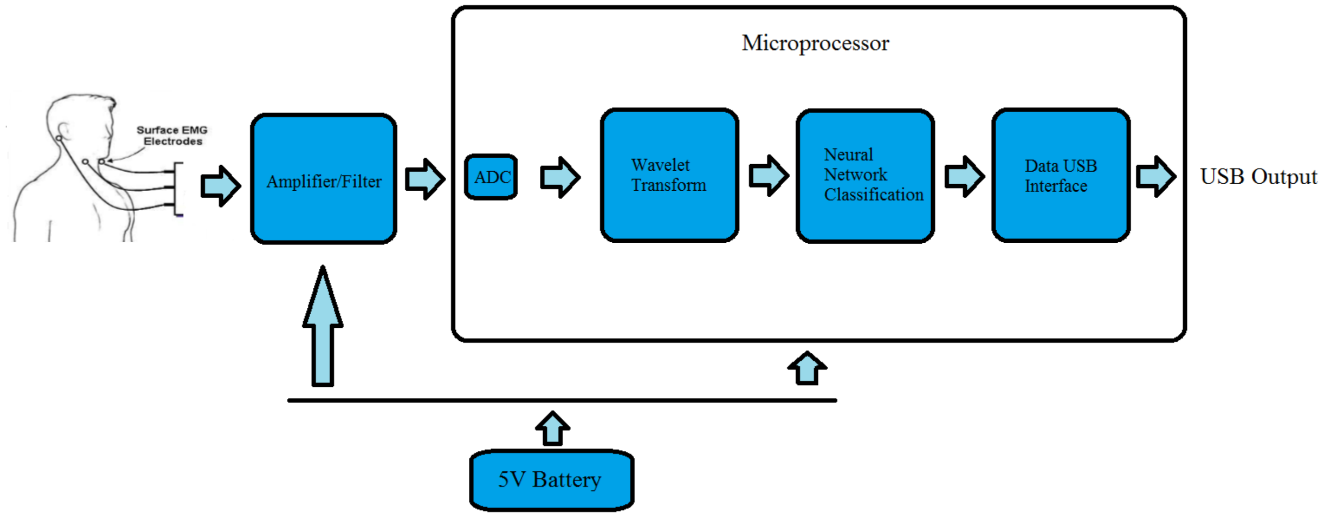


Figure 1.2: Level 1 Block Diagram

Table 1.2: Level 1 Block Diagram Description Table

Table 1.2 below describes the inputs, outputs, and functionality of the modules of the level 1 block diagram.

Module	Amplfier/ filter	ADC	Wavelet Transform	Neural Network Classification	5V Battery	Data USB interface
Inputs	-Subvocal EMG input signal [6] -5V DC voltage	-Filtered and amplified waveform: 5 Hz to 450 Hz [22] -5V DC voltage	-Digital audio signal: (1 V to 2.5 V peak) -5V DC voltage	-Data compressed signal (noise removed) (1 V to 2.5 V peak). -5 V DC voltage.	-5 V from charger (computer, wall charger, etc.)	-Classified digital signal - 5 V DC voltage
Outputs	Filtered audio signal: 5 Hz to 450 Hz (micro volts in amplitude) [33].	Digital Audio signal: 1 V to 2.5 V peak, 5 Hz to 450 Hz [26].	Data compressed signal (noise removed) (1 V to 2.5 V peak).	Classified digital signal.	5 V DC voltage.	-Audio signal (user's voice)
Functionality	Filter input so that only signals with frequencies of 5 Hz to 450 Hz are given a large voltage gain to compensate for the small amplitude voltage. Input resistance > 1 MΩ and output resistance < 100 Ω.	ADC built into microcontroller. Converts the analog and filtered input signal into a digital signal in order to be usable to the microcontroller.	Deconstruction and reconstruct of the digital signal using a discrete wavelet transform in order to compress the data into a more general form. This allows for unwanted details created by noise to be removed, thus making the signal easier to classify.	Compare input signals to previously used signals matching English vowels and constants (neural network "training" signals) and appropriately match and classify according to these "training" signals.	Provide power to the rest of the device and its modules.	Converts the classified signals into a proper audio signal that can be inputed into a computer or cell phone.

2. CUSTOMER NEEDS, REQUIREMENTS, AND SPECIFICATIONS

This chapter covers the customer needs that the Silent Communcation Device addresses and the requirements and specifications of the said device.

2.1 Customer Needs Assessment

Customer needs were determined by interviewing students about noise distractions in the workplace and considering what design for the device would be the most convenient for the user. Many of the replies consisted of grievances about too much talking in the workplace. When asked about how a device that allowed people to communicate silently could be convenient for them, many of the replies involved wanting compatibility with a cellphone and for it to not be too expensive. Thus, it became evident that customer needs involved not just allowing the device to make speech silent, but also compatibility (both in terms of outputting to a cellphone and staying powered as long as a cellphone stays powered) and a cost able to discourage potential customers from buying existing alternatives. As a result, the marketing requirements were created from these customer needs. These include being pocket-sized, having long-lasting power, being affordable, having the right size of audio jack to connect with other electronic devices, having control over the volume of their transmitted voice, and, of course, allowing silent speech.

2.2 Requirements and Specifications

The requirements and specifications for this device were determined by considering what was most efficient for creating a device that allows for silent speech as well as what was most convenient for the user. For example, the first requirement takes into consideration how much the public might view the worth of this product, how expensive it should be in order for it to be mass produced for soldiers in the military, and what kind of parts for it to operate properly. Anything beyond $50 would be considered too expensive for what would be considered a peripheral device in which consumers might still consider keyboard and pads a better alternative. The requirements for the 3” × 4” × 1” dimensions were determined by the fact that, since this device needs to be used in conjunction with another electronic device which would most likely be a cellphone, dimensions that allow the user to carry the device with a cellphone, while still not being too small to hold enough circuitry to operate properly are necessary. The dimensions must also be large enough to include a microcontroller (3” x 4”) and be expected to have integrated circuitry stacked on top of each other to prevent the device from being too long to fit in a pocket (1”). The requirements for the 5V battery to power the device and the need for an audio jack with industry standard dimensions [24] were determined from considering device’s compatibility with the device it outputs the user’s voice to. For instance, 3.5 mm audio jack dimensions are the standard audio jack dimensions for most cellphones and other electronic devices [24], allowing the device to connect to these devices easily. The battery ensures that the device uses enough power to last at least as long as the cellphone would last, while providing a common enough voltage of 5V to power its components properly. Other requirements include requiring the audio amplifier portion of the device to provide enough gain to amplify and use the very small subvocal input signal, not having any exposed wiring and conductive devices exposed to the user to keep the user safe and comfortable, and allowing volume control for the output of the device up to 85 dB (loudest possible before it becomes hazardous) [15]. Finally, the third requirement allows the device to achieve its main purpose, allowing completely silent speech. Making any noise heard by unintended recipients that is louder than 25 dB (a quite conversation) [15] would make whispering a better alternative to using this device. The engineering specifications and marketing requirements are shown in table 2.1 below.

Table 2.1: Silent Communication Device Project Requirements and Specifications

Marketing Requirements	**Engineering Specifications**	**Justification**
1	The device must cost the customer at most $50.	Based on the price range of most pocket-sized devices that are used by the public, this price makes the device affordable as well as provide a good profit for the producer.
2	The device's dimensions must not exceed 3" × 4" × 1" (inches) (previously 2" x 4" x 0.4", but made larger to fit the microcontroller)	The rectangular shape and thinness makes the device easily able to fit in a person's pocket, while still providing enough room for plenty of circuitry for efficient operation.
5	The device must prevent unintended recipients at least 10 feet away from the user from hearing at most 25 dB of the user's subvocal speech [6].	In order for the device to achieve its primary purpose, making speech completely silent, others who are not the indented recipients of the communication must not be able to hear the user when he or she is using the device. 25 dB is the sound of a quiet conversation in which anything higher would be too loud and whispering would be a better alternative [15].
4	The device's circuitry must be powered by a battery able to output 5V for 24 hours.	A value of 5V is a small enough voltage to ensure that the device stays powered for a fairly long period of time before recharging is required. 5V is also a common voltage needed for powering common components in low-power circuitry.
3	The device must be able to output to any audio jack with industry standard dimensions [24].	Since the device needs to be able to connect to common electronic devices, an audio jack with industry standard dimensions such as having a (3.5mm) diameter outer conductor are needed [24].
6	The device is non-invasive (no part of the device is required to enter the human body for proper operation).	The device would have a lot less appeal and be a lot less comfortable to customers if a surgery was required to use it.

Marketing Requirements	Engineering Specifications	Justification
5	The device must be able to provide a gain of at least 10 [3] to a maximum of 18700 to amplify subvocal input signals [23].	In order to successfully allow completely silent speech, the device needs to provide a sufficient amount of gain to amplify the subvocal input signal as the signal with has very small amplitude or else said input is unable to provide a proper output (user's voice) [23].
8	The acoustic sensor must not cause electrical shocks or physical injury to the user.	Customers are not comfortable or safe if the device risks electrically shocking them through exposed conducting material.
7	The user control for the device must have a range of volume from 0 dB to 85 dB for the output (user's voice).	Any volume above 85 dB is considered hazardous to health and too loud [15].

Marketing Requirements

1. The device should be relatively low cost and affordable for the general public.
2. The device should be pocket-sized and easily carried.
3. The device should be compatible with other devices used for communication such as cellphones, computers, and audio jacks.
4. The device should have long-lasting power.
5. The device should allow completely silent speech.
6. The device should be comfortable for the user.
7. The device should allow the user to control its output's volume.
8. The device should be safe for the user.

3. DESIGN METHODS INCORPORATED, DESIGNS CONSIDERED, AND REDESIGN.

This chapter covers the intial designs of the project, the redesigns to the project itself, and the research behind them.

3.1 Subvocal EMG Signals

The properties of the subvocal signal and the types of noise that it experiences must first be considered to design a project focused on obtaining subvocal signals, filtering them of noise, and properly classifying them.

As previously stated in the introduction, subvocalization is a phenomenon in which, when a person thinks of words or phrases, their human vocal cords create small vibrations that emit these as sound back into the brain for the person to hear as their own thoughts [6]. In order to capture and classify these small vibrations as usable data, the small, collective electric signals controlled by the nervous system that are produced during muscle contraction (EMG (Electromyograpy) signals) can be captured as they pass through the muscles that vibrate the vocal cords during subvocalization [25] via EMG electrodes attached to these muscles. The signal bandwidth in which significant EMG activity occurs that is the most useful in terms of classification and usable data is from 5 Hz to 450 Hz [26], while amplitudes range on average from about 0 to 600, occasionally spiking up to about 1000 µV with Ag-AgCl surface EMG electrodes [33]. However, amplitudes have been shown to become slightly higher when using needle electrodes such as the concentric needle electrode, which can pick up amplitudes higher than 1000 µV [45].

These subvocal EMG signals experience seven different types of intrinsic noise that must be considered in the design of the project so that they can be properly removed.

1. The first is the inherent noise in the EMG electrode which ranges from 0 Hz to several thousand Hz. When using non-invasive electrodes, one of the most common strategies for removal of this noise is by using electrodes made of silver/silver chloride with the electrode dimensions of 10 x 1 mm because the electrode size and materials give just the right amount of impedance for an adequate signal-to-noise ratio while also being electrically very steady, although dimensions may vary if the statistical power of the EMG signal is very high or very small.
2. The second type of noise to consider is from the movement between the electrode attached to the skin and the skin itself, called the movement artifact, which causes a significant amount of noise between 0 Hz and 10 Hz. Several common methods of removing this noise include placing a conductive gel layer between the skin and the electrode, scratching the skin, and using an adaptive filter.
3. The third type is the what is known as electromagnetic noise, which arises usually around 50 Hz or 60 Hz and in its harmonics (Ex: 100 Hz, 200 Hz, 300 Hz, and 400 Hz for 50 Hz noise). Common methods of removing this noise include using notch filters to filter out a specific frequency or using adaptive notch filters for removing not just one frequency but its harmonics as well.
4. The fourth is from picking up signals from other EMG signals that are unwanted, also known as crosstalk, which have a wide range of signal frequencies and tend to increase in magnitude with increasing subcutaneous fat thickness. Since

filtering the noise is difficult, crosstalk noise is most commonly removed by choosing electrodes with the right amount of conductive area, interspacing, and axis direction relative to muscle fiber direction and choosing a location of said electrodes that avoids neighboring muscle fibers as much as possible.

5. The fifth is the internal noise produced by physical capacitive effects due to the muscle tissue and fat. Since the strength of this type of noise depends on the depth and location of the muscle fibers that are needed to obtain the subvocal EMG signals, this type of noise is considered negligible since there is minimal tissue blocking the way between the skin and muscle fibers being used. However, if excess body fat is involved, removing the noise is commonly done through surgical fat layer reduction or partially removing the noise by using high pass spatial filters, the latter obviously being the more marketable option, but, for the sake of simplicity, it is assumed that the user does not have excess body fat.
6. The sixth is the inherent stability of the EMG signal itself which a result of the signal being quasi-random in nature and is the strongest between 0 Hz and 20 Hz. Since the noise is a quality of the signal itself, it is impossible to remove practically, but its behavior can be changed based on the mechanical nature of the muscle fibers.
7. Finally, the seventh type of noise is a result of electrocardiographic ECG artifacts which are caused by the electrical activity of the heart and usually occurs at frequency above 100 Hz. A common method to removal this noise is through high-pass filtering.

In addition, there are also non-intrinsic sources of noise that have to be considered such as EMG artifacts from swallowing, muscle fatigue tremors, or coughs in which using the electrodes when these EMG artifacts are not occurring is generally ideal.

3.2 Amplifier/filter Design

Now that the type of input signal that is being dealt with has been discussed, the design of the amplifier/filter module can be understood.

In order to first extract the signal from the user non-invasively, two EMG electrodes are applied to the user's throat on the muscles near the user's larynx to obtain the signal, while another one is attached below the user's right ear to act as a signal to ground for a total of three electrodes (demonstrated in Figure 4.55). To help minimize the effect of the inherent noise in the electrode and cross talk, silver/silver chloride (10x1mm) electrodes [27] are used because of their high signal-to-noise ratio and electrically stability [26] which negate a significant amount of the inherent noise in the electrode, while the two electrodes that obtain the EMG signal (excluding the ground electrode) are placed 5 cm from the left and right from the user's larynx in order to obtain a proper signal while minimizing cross talk by keeping them at least the radius of the each electrode in distance away from each other [26].

After extracting the signals through EMG electrodes, these inputs need to be amplified and filtered (each of the two EMG electrodes obtaining the signal are output to two separate but same amplifier/filters, while sharing the same ground electrode). Since it is a differential input, the amplifier/filter needs to have an input component with a high common-mode rejection ratio to deal with common noise from both differential inputs.

Taking this into consideration, the inputs are fed into an instrumentation amplifier (INA128P is used for its high CMRR,120 dB at gain > 100, and its specialization at dealing with EMG, ECG, and EEG signals [28]) and which provides a gain. After the instrumentation amplifier, the signal passes through a bandpass filter designed to filter the signal between 10 Hz and 450 Hz to obtain, as previously stated, the most usable data across the EMG subvocal signal spectrum in which it is being low-passed at 10 Hz as opposed to 5 Hz to remove noise created from the movement artifact. After being filtered, the signal is then sent into the microcontroller to be further processed. It is important to note that, since there are three electrodes with one of said electrodes acting as ground for the other two electrodes, two identical amplifiers/filters are needed for each of the two electrodes obtaining the signal, but the project shall be treated as needing only one amplifier/filter for now for the sake of brevity until later in chapter 4.

The first design of the amplifier/filter module involved using a first-order high pass filter at 10 Hz and a unity gain sallen-key low-pass filter at 450 Hz to provide the bandpass response, while the instrumentation amplifier provided the gain. Unfortunately, this design did not work well as the sallen-key's second-order low-pass response and high Q factor resulted in a low-pass response that was too strong and ended up turning the band-pass response into a low-pass response with a corner frequency at 3 Hz.

The second design of the amplifier/filter focuses on spreading out the gain and giving a weaker low-pass response. Modeling after the amplification and filtering circuit shown in "Subvocal Speech Recognition Based on EMG Signal Using Independent Component Analysis and Neural Network MLP" [3], the second design forms its band-pass filter with a first-order high-pass filter at 10 Hz and first-order AC integrator Op-

Amp with gain at 450 Hz, allowing not only the instrumentation amplifier to provide gain, but the integrator as well. The Op-Amp chosen is the mc33078p because it provides a high open-loop AC gain (800 at 20 KHz) and can operate at 5V [29]. The circuit simulation in LTSpice IV is shown below in Figure 3.1 in which the LT1167 is used to replace the INA128P because there is not a model available to use (the circuits are very similar, have the same gain equation, and pinout, but with the INA128P being specialized to deal with EMG signals) and the LT1457 is used to replace the mc33078P because there also is not a model currently available to use (both op-amps still being very similar). 104 µF and 47 pF capacitors are placed on the reference voltage of the instrumentation amplifier to AC couple any noise waveforms both in the high and low frequency ranges. The Op-Amp's positive terminal is supplied with a voltage of 2.5 V via voltage divider to raise the output signal to have a DC voltage of 2.5V. This allows the output signal to not lose half of its waveform by having half of the waveform in the negative voltage range (the microcontroller's ADC does not accept negative voltages). Gain is kept around 0 dB at band-pass until is determined whether or not it needs a higher gain.

Calculations:

High pass: f_o (corner frequency) = $1/(2\pi RC) = 1/(2\pi(0.1E\text{-}6)(160000)) = 9.94718$ Hz.

Low-pass: f_o (corner frequency) = $1/(2\pi RC) = 1/(2\pi(120E\text{-}12)(2700000)) = 491.2189$ Hz.

DC gain: $- R_2/R_1 = 2700000/27000 = 100$.

Instrumentation amplifier gain: $1+ 50K/R_G = 1 + 50K/50K = 2$.

Voltage divider: $R_1/(R_1 + R_2) = 200K/(200K + 200K) = 0.5$.

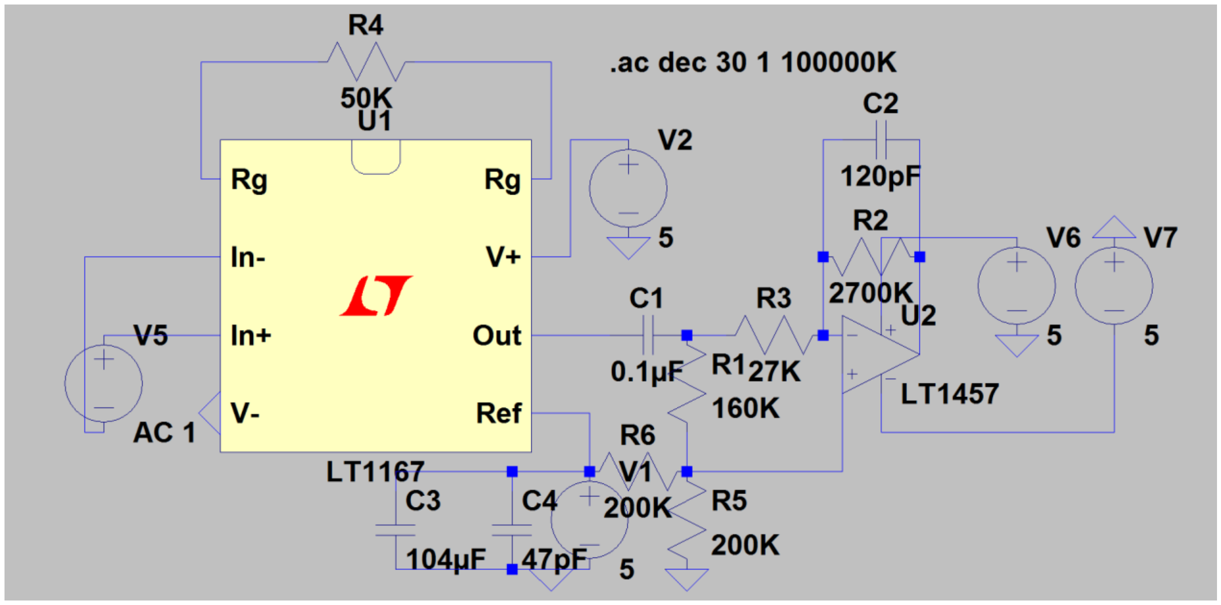


Figure 3.1: Amplifier/filter Design LTSpice IV Simulation Circuit

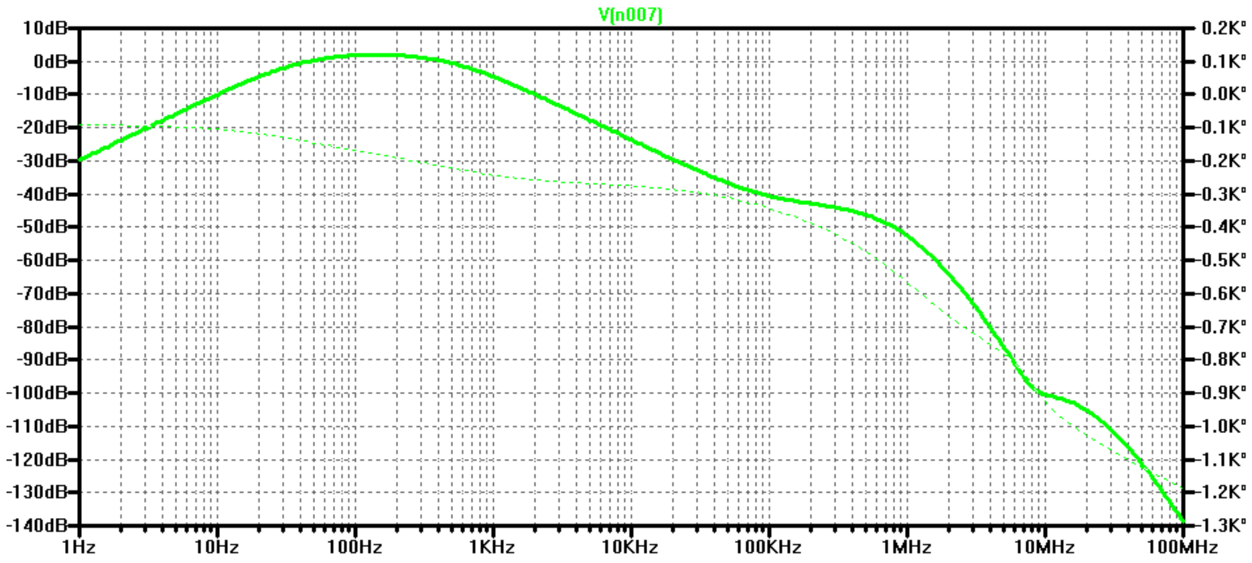


Figure 3.2: Amplifier/filter Design LTSpice IV Simulation at Output

Figure 3.2 above shows that this design provides a proper band-pass response between 10 Hz and 450 Hz.

3.3 Discrete Wavelet Transform

Once the subvocal EMG signal is filtered, amplified, and converted into a digital waveform by the microcontrollers ADC, the signal passes through a signal processing method known as the discrete wavelet transform. The discrete wavelet transform is an orthogonal function used for a finite group of data (in this case our subovcal EMG signal samples) in which the data is decomposed into separate frequency bands, much like the discrete fourier transform except it is also able to display these bands in respect to time (locations).

This "decomposition" works by sending and resending the data through two filters, one low-pass and the other high-pass, as mathematically represented by the two equations below:

Low-pass: $a_i = \frac{1}{2}\sum_{j=1}^{N} c_{2i-j+1} f_j, \ \ i = 1, \ldots., N/2$

High-pass: $b_i = \frac{1}{2}\sum_{j=1}^{N} (-1)^{j+1}\, c_{j+2-2i}\, f_j, \ i = 1, \ldots., N/2$

Where "c" refers to the coefficients that retain the qualities the each individual frequency component, a and b are the outputs, N is the number of inputs (or input block size) determined by N = 2^D (D is the number of dilations), and f is the input function. Each input that passes through the low-pass filter is generally called an "odd" input, while the inputs that pass through the high-pass filter are called the "even" inputs. Figure 3.3 below, as referenced from [30], shows a representation of the decomposition process as a group of data inputs are decomposed four times (sent through four "dilations").

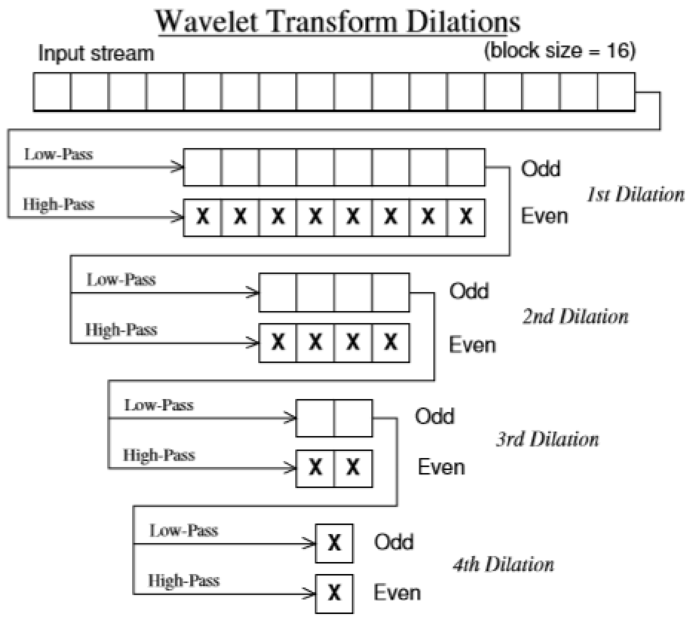


Figure 3.3: Wavelet transform dilations diagram: an input stream being sent through four dilations via discrete wavelet transform.

As Figure 3.3 also shows, the even outputs and final odd output (represented by “x”s) are the outputs of the wavelet transform after each dilation, while the odd inputs continue to be sent through each dilation until the final output is reached. By constantly sending each of these inputs through low-pass and high-pass filters, the wavelet transform is able to separate each of the signal’s data samples into individual frequency components on a time scale, thus resulting in a set of data useful for signal classification. After being decomposed, the data can also be reconstructed by sending the data through the reverse (coefficients on the lattice filter reversed) of the original wavelet function, resulting in the same input data waveform with noise from lower frequencies removed, which, in terms of the project, is useful for removing the other types of the noise the amplifier/filter could not deal with.

In terms of the project at hand, the number of dilations and type of coefficients must be chosen wisely when designing the discrete wavelet transform. Since each dilation results in the chance for more and more input data samples to low-passed, it is necessary to have at least a moderate amount of dilations to make sure low frequency components of the signal get separated in which, since the amplifier/filter is sending in signals in the bandwidth of 10 Hz to 450 Hz, having a moderate amount of dilations is necessary. However, there is a problem considering that the discrete wavelet transform occurs on a microcontroller as opposed to a computer, the usual environment in which a discrete wavelet transform would usually perform, in which the amount of memory and processing power to handle the discrete wavelet transform is limited. This means that there must be a compromise between having enough dilations to obtain low frequency components within the 10 Hz to 450 Hz bandwidth and not having so many dilations that the microcontroller cannot handle it. Fortunately, using data from a similar project involving using a microcontroller to implement wavelet transforms on biomedical signals [31], it can be seen that 4 dilations is sufficient, resulting in an input block size of $N = 2^4 = 16$. The type of coefficients used are the Haar coefficients ($c_0 = 1.0$, $c_1 = 1.0$, $c_{2,3,4,5} = 0$) for their simplicity, resulting in faster calculations and less memory space being taken up.

Using the MSP432P401R LaunchPad microcontroller for its processing power, memory, and variety of capable mathematical functions (The MSP430 LaunchPad microcontroller served as the first microcontroller to test the discrete wavelet transform, but was discovered to not have the processing power or a multiplier function to properly process the code), the discrete wavelet transform is coded in C as shown in appendix A as Figure A.1.

For the initial design of the wavelet transform, the code was referenced from the wavelet transform code in [30] and modified to take inputs from the port inputs of the microcontroller as opposed to manually typed inputs and input files. The basic structure of the code is a lattice filter structure modified for use as a wavelet transform filter as shown below in Figure 3.4.

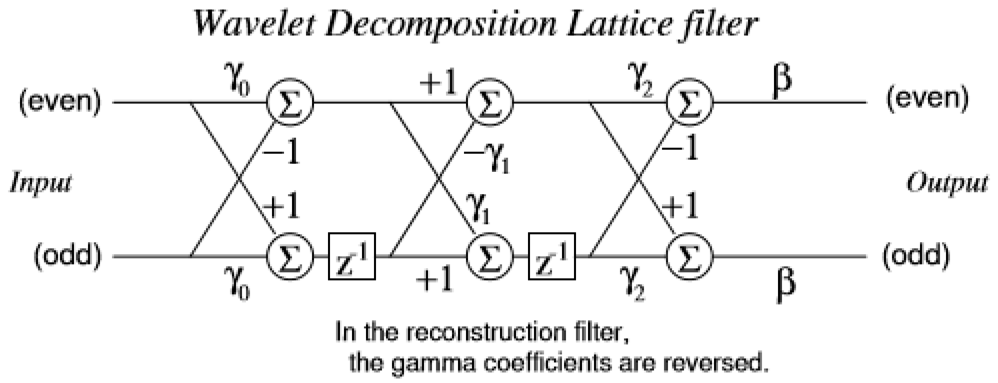


Figure 3.4: Wavelet decomposition lattice filter basic structure (6th order) as referenced from [30].

In this case, the 6th order lattice filter is used in which each of its three "rungs" is associated with its γ (gamma) coefficients, the 6th order representing the six coefficients used (the Haar coefficients: $c_0 = 1.0$, $c_1 = 1.0$, $c_{2,3,4,5} = 0$, $\beta = 1/\sqrt{2}$) in which the γ_0 coefficients are the first two coefficients (1.0 and 1.0), γ_1 are the next two (0 and 0), and γ_2 are the final two (0 and 0). While segmenting a continuous stream of inputs into 16 sample blocks (4 dilations for each 16 sample block) for every wavelet transform, each even numbered input goes into the "even" input and each odd numbered input goes into the "odd" input, resulting in the lattice filter to filter each even and odd sample to be either an "even" (high-passed) or "odd" (low-passed) output as shown in Figure 3.3, thus sending said samples through a wavelet transform. As each input shifts from rung to rung, the delay cycles (z^{-1}) assure that the input is held for one “shift cycle” for every z^{-1} delay it passes through, allowing the second and third rungs to do arithmetic with the results of previous inputs.

Next, a system needs to be put into place that handles the inputs, outputs, and delay cycles of the lattice filter to make sure that all samples pass through the filter in the appropriate order (including separating samples into even and odd orders, filtering them, delaying them, and then organizing them in the order shown in 3.3 for the output). This is done by creating what are called lookup tables which consist of patterns of numbers in arrays used to organize samples to enter and exit parts of the lattice filters at the appropriate times in order to ensure that they are inputted, filtered, and outputted in the correct order, while also allowing the inputs to be stored for each delay cycle and then used again after said delay cycle has ended while passing through the lattice filter. Discussing how the number array is organized for the lookup tables in full detail would be cumbersome and not entirely necessary since this method was not implemented in the final version of the project and thus is not discussed. The level 1 block design for handling the inputs, outputs, and delay cycles is shown below in Figure 3.5.

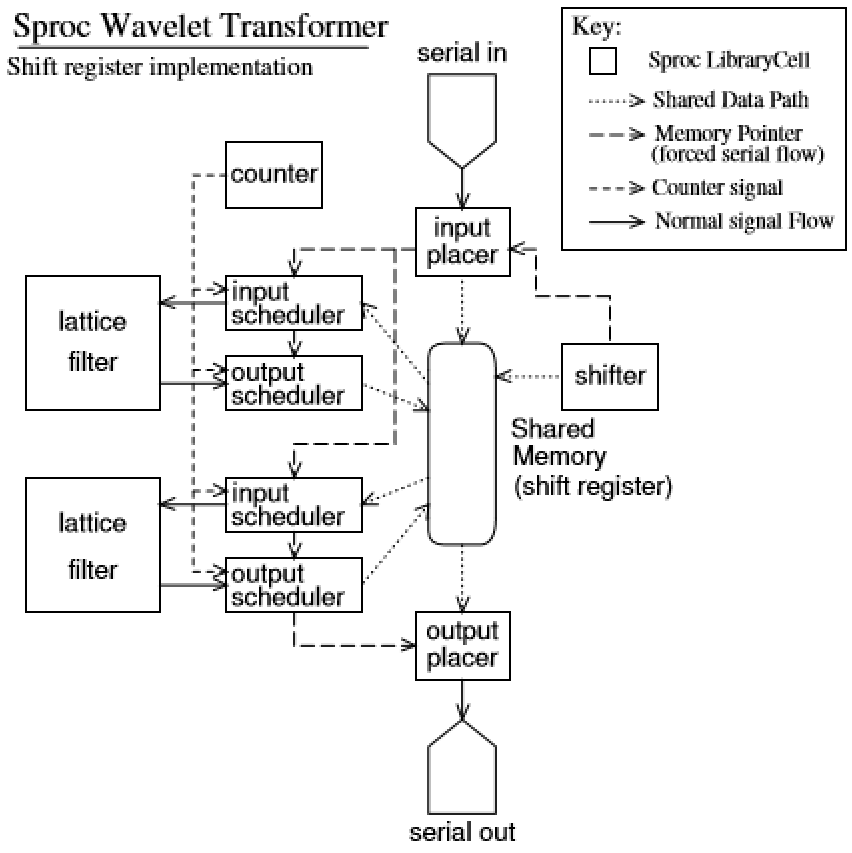


Figure 3.5: Input and output handler for the wavelet transform code as referenced from [30].

In addition to the wavelet transform or decomposition filter, a reconstruction filter is also created to, if chosen by the user, reverse the effects of the wavelet transform or reconstruct the signal to receive a reconstructed version of the raw signal with filtered out noise as opposed to just outputting the wavelet transform version of the input signal. This is done by creating a copy of the wavelet transform filter (decomposition filter) code and modifying it so that its lattice filter has its coefficients reversed (γ_2 coefficient switched with γ_0 coefficient), while its lookup tables are altered to reverse the order at which the sample inputs from the decomposition filter are placed in the lattice filter of the reconstruction filter, thus reconstructing the signal outputted from the decomposition filter. The constant low-passing and high-passing of the wavelet transform results in the removal of noise from the reconstructed signal.

3.4 Project Scope Redesign

The neural network and USB interface may be designed now that the decomposition and reconstruction filters have been designed. Unfortunately, there were a lot of problems with including these into the full design, mostly concerning the neural network classification system. For every neural network, data is needed to "train" the network for it to be able to recognize and classify signals according to the signals that are used to train it. Therefore, to get accurate results, the network must be trained with hundreds of samples for each source (or in this case, letters and words) to get an accurate response. The amount of data, research, and time needed to collect the amount of data needed for not only every letter of the English language as well as the amount needed to further classify each string of letters into words, phrases, and sentences is beyond the scope of this work (this would take years of research and testing). In addition, a microprocessor needed to handle this level of data and processing power, in addition to the wavelet transform and ADC functions, would be far too costly to meet the $50 max budget design specification.

With these problems in mind, it becomes clear that the project's scope and purpose needs to be redefined. With the neutral classification system and data USB interface (no letters, words, or phrases are available to be sent to a smart phone or computer) removed, Figure 3.6 shows the current level 1 block diagram of the project in which the project now focuses on taking EMG subvocal signals from the user and translating them into usable data for analysis and classification, no longer having much purpose for everyday communication, but certainly useful for medical and research applications such as subvocal signal analysis. In addition, the subvocal signal is now able to be reconstructed as well as wavelet transformed in order to provide a raw, denoised output. An external DAC is now used as opposed to using the internal DAC of the microcontroller because an external DAC can accommodate the micrcontroller's 14-bit resolution, while it's internal DAC's can only provide outputs in 12-bit resolution [32].

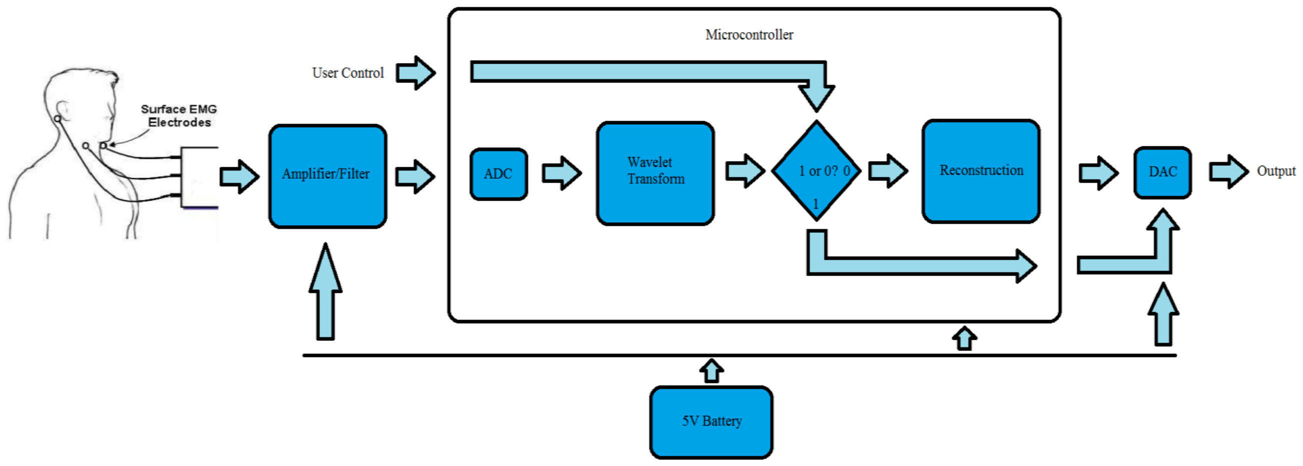


Figure 3.6: Final Design Choice.

In addition to removing the neural network and USB interface from the microcontroller code, a new neural network designed in Matlab was decided to be created that would receive the output of the microcontroller and classify the data in order to prove the project's effectiveness for subvocal signal analysis. To keep the scope of the classification simple, the classification results with be limited to only five vowels ("a", “e”, “i”, “o”, and “u”). Since the neural network was designed and implemented much later, details on the design of the neural network are discussed later in this chapter.

3.5 Initial Microcontroller Code Input and Output Handler

Finally, the input and output formatting for the MSP432P401R microcontroller must be designed as well to ensure that the microcontroller can receive samples from the amplifier/filter and output them to the neural network. When the microcontroller is first started, a number of initialization statements are processed to setup the settings for the multiple assets being used to allow the microcontroller to input and output signals. These include the TIMERA timer, the DCOCLK (clock used to run other clocks such as the SMCLK which is used to run the UCBxCLK), ADC14, SPI, and the buttons used for the user interface of the microcontroller. After these initialization statements are processed, the microcontroller waits in an infinite loop until a button is pressed. Next, the input is taken by calling an interrupt routine that is triggered by pressing a button wired to the microcontroller. This ensures that the microcontroller only takes samples when the user wants it to (as long as the button is being pressed). Once the button is pressed, samples are taken via the ADC (analog-to-digital converter) that is built into the microcontroller (ADC14). The samples are taken in 16 sample blocks before being sent to the decomposition function (wavelet transform).

In regards to determining the timing of the sample clock for each input sample, the number of clocks cycles not only have to take into account the maximum frequency possible to sample, but also the time it takes for the ADC of the microcontroller to sync, sample, and convert its input. Since the desired frequency range of the input signal is 10

Hz to 500 Hz, the maximum frequency is 500 Hz which takes a total sampling time of 1/(20 × 500 Hz) = 100E-6 seconds. The ADC sync, sample, and conversion is determined by referencing the MSP432P4xx family user's guide [32] in which the sync time takes at least one ADC14 clock cycle, sample time takes multiples of 4 × ADC14 clock cycles (chosen 16 cycles by the ADC14SHT1x register), and the conversion times takes 16 × ADC14 clock cycles as shown in Figure 3.7. With an ADC14 clock source of MODCLK (24 MHz, typical), the resulting time is no less than (1/24 MHz) × (16 + 16 + 1) = 1.375E-6 seconds. Combining this with the maximum frequency sampling time, the result is 100E-6 + 1.375E-6 = 101.375E-6 seconds to get a sample.

Finally, since the timer that determines when an input sample is taken is a TIMERA timer (timer built into the microcontroller that can trigger an interrupt when a certain number of clock cycles are reached or CCR0 cycles), converting this into CCR0 clock cycles results in (1/12 MHz) = 83.33E-9 seconds (time of each SMCLK cycle), 101.375E-6/83.33E-9 = 1216 cycles (add 84 cycles to compensate for error for a total 1300 cycles).

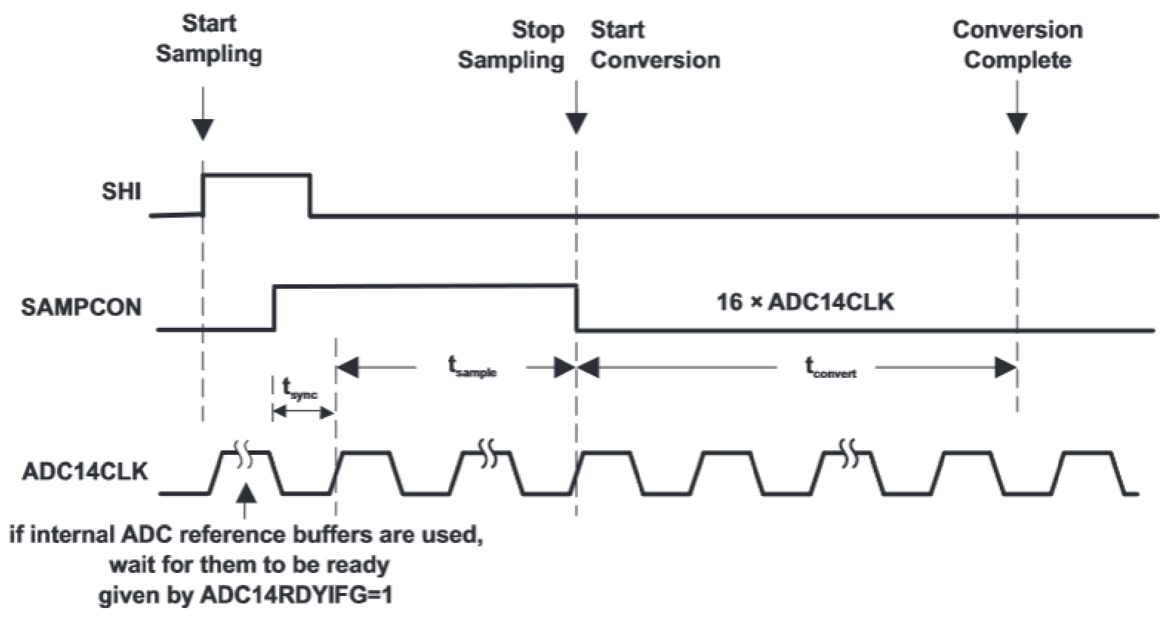


Figure 3.7: ADC14 sync, sample, and conversion time, referenced from [32].

However, it must be taken into account that this is only the minimum time it takes to get each sample as the wavelet transform must be run after each sample is taken. To have each sample equally spaced on a time scale, the number of CCR0 clock cycles must take into account the time it takes to run the wavelet transform as well. Further experimentation is needed to determine the total number of CCR0 clock cycles so, for now, a sample is taken every 1300 CCR0 clock samples.

Concerning how the microcontroller gives its output, a second button is used, when pressed, to trigger an interrupt routine that determines whether or not the output will be from the decomposition filter or the reconstruction filter (default is decomposition filter and changes when button is pressed). Whether from the decomposition filter or the reconstruction filter, the output signal is sent to the function “do_some_conversions” which converts the value type of the output from float to int type. This is necessary because the ADC14 takes its samples as float values automatically, but, in order to output values out of the UCB0TXBUF register (register used to output register values out of the microcontroller), the samples must be int type values. After being processed by the "do_some_conversions" function, the output values are sent to the "Drive_DAC" function which handles the output values of the UCB0TXBUF register and drives an external DAC (digital-to-analog converter) that converts the digital output to an analog signal, allowing the now analog output to be processed by the neural network. The "Drive_DAC" function drives the external DAC by providing three signals from the microcontroller: the SPI signal which contains the output data, the /CS signal which tells the external DAC when it should be expecting an SPI input, and the UCBxCLK which is the clock that runs the external DAC, thus allowing the microcontroller to effectively output the wavelet transformed or reconstructed signal to the external DAC. Full initial code shown below in Figure A.1, coded in Code Composer Studio 6.1.1.

3.6 Neural Network

Before designing the neural network, a literature review was conducted to obtain ideas and sources of influence for the best possible design of the neural network for classifying subvocal signals.

The first and most significant approach in terms of influence on the design of the neural network was the paper "Sub-vocal Phoneme-Based EMG Pattern Recognition and its application in Diagnosis" by Mosarrat Jahan and Munna Khan [33]. In this paper, EMG signals were obtained using EMG electrodes on the throat sampled at 500 samples/ second. The signal was then sent into a laptop to be processed. Due to the non-stationary nature of EMG signals (contain different components of frequency at different instants of time), the EMG signals were processed with Discrete Wavelet Analysis (three levels or coefficients using mother wavelet Daubechies ‘db4’), which decomposes a signal using “wavelet functions” and allows each component to be analyzed in different frequency domains. After the wavelet analysis processing, the individual frequency components had their Mean absolute deviations (MAD) and Standard deviations (SD) calculated, which were used as inputs into a “linear classifier” (assumed to be a perceptron). The perceptron then used these inputs to classify each subvocal EMG signal as a particular Hindi phoneme. The database was taken from four student subjects, each giving 10 example trials of each of the four hindi phonemes being classified and resulting in a database size of forty total raw samples (10 for each phoneme) for each student (160 total samples for all four students). The resulting classification accuracy reached from about 70 to 80 percent. A very similar method was also used in the paper “Subvocal Speech Recognition System based on EMG Signals” by Yukti Bandi, Riddhi Sangani, Aayush Shah, Amit Pandey, and ArunVaria [34], which used almost the same extraction and processing methods to acquire the signal (insignificant differences), but used a multilayer perceptron neural network instead with also only a few inputs to classify the whether or not the subvocal EMG signal was either the word “forward” or “reverse”. The resulting classification accuracy was 74.4% for “forward” and 72.5% for “backward”. Both of these methods have the advantage of using discrete wavelet analysis to process the EMG signals, which allows the differences in frequency components between each EMG signal to be analyzed much more easily than without the analysis. However, they also possess some limitations. It should be noted that both methods use very simple classification methods, the first using only a linear classifier, which is unable to properly classify any non-linearly separable input patterns with their outputs and the second using just a multilayer perceptron with back propagation for training (using "patternnet" would most likely be better for the job considering it specializes in handling classification and pattern recognition). In addition, both methods use very little inputs (features such as the root mean square of signal) to distinguish input signals from each other. Using more inputs and more qualified neural networks would improve classification accuracy at the expense of processing time and memory.

Another useful paper to mention is "The use of Artificial Neural Network in the Classification of EMG Signals" by Md. R. Ahsan, Muhammad I. Ibrahimy, and Othman O. Khalifa [35]. Although this paper does not use subvocal EMG signals in its experiments, but EMG signals taken from human hands to classify hand movement from said EMG signals instead, the method taken to classify these signals demonstrates the advantage of using many features of the EMG signal as inputs to the classification neural network in terms of improving accuracy. For example, the paper's experiments on classifying EMG signals to hand motions involve the use of 7 (not two) features as inputs to a feed-forward neural network. The results show accuracies with an average success rate of 88.4% (much better than 70%). However, this method is also limited for the use of subvocal EMG signals as it does not use a wavelet transform and therefore would not be able to classify subvocal signals as well due to the fact that subvocal signals are much harder to distinguish from each other than that of non-subvocal signals as non-subvocal signal have much larger magnitudes than that of subvocal signals and thus are easier to distinguish from noise.

The papers “Subvocalization – Toward Hearing the Inner Thoughts of Developers” by Chris Parnin [6]; “Random Forests Based Sub-Vocal Electromyogram Signal Acquisition and Classification for Rehabilitative Applications” by Biswajeet Champaty, Bibhu K. biswal, Kunal Pal, and D. N. Tibarewala [36]; and “The Application of AR Coefficients and Burg Method in Sub-vocal EMG Pattern Recognition” by Muuna Khan and Mosarrat Jahan [37] also contribute the design of the project’s neural network such as providing various methods of extracting and processing subvocal EMG signals. For instance, the first paper by Chris Parnin mentions the idea of bypassing the subvocal signal completely and simply measuring the EMG signals from facial muscles of person mouthing words (but not talking) instead, thus eliminating the need for complex signal processing and allowing classification accuracy to be much easier to achieve by using the much larger in magnitude non-subvocal signals from facial muscles while achieving silent speech. This may sound like a feasible method and although it has the advantage not needing complex signal processing, it has limitations such as the added inconvenience of having mouth out words to achieve silent speech and is much more likely to give unwanted noise signals due to stray movements in facial muscles, which vocal muscles are much less likely to give. While one may argue that mouthing out words provides an additional method of silent communication by allowing one to read another’s lips, it is important to note that the silent communication device is designed to allow silent communication through users’ electronic devices, which transmit communication across long distances. Therefore, mouthing out words provides no advantage unless users are directly in front of each other, making lip reading only useful under very limited circumstances (e.g. two soldiers communicating across a battlefield should not have to rely on lip reading to communicate silently). While a computer could perform lip reading, it would have to have some way of viewing the lips, which would require an awkward angle of viewing that would most likely require hand movement or an exposed camera that could easily get in the user's way. For the most convenient and reliable EMG signals, the subvocal EMG signals from the throat perform better as a result. The second paper (Random Forests Based Sub-Vocal Electromyogram Signal Acquisition) is significant for offering an alternative method to using neural networks for classification by instead using an ensemble of decision trees called Random Forests (RF) to carry out the classification, which showed a classification accuracy of 90% while only using two features as inputs, a significant improvement over the other mentioned methods. However, its limitations are shown by the fact that the analysis is much more complicated, requiring two programs (LabVIEW 2010 and Microsoft Excel) as opposed to one (Matlab) to implement in addition to requiring more hardware to extract and process the signal as well, thus resulting in a more expensive method requiring more processing power and memory. Finally, the third paper by Muuna Khan and Mosarrat Jahan demonstrates another alternative to using a neural network for classification by using an algorithm known as the Burg Method to classify sub-vocal EMG signals. Like the Random Forests method, however, this method of classification is also much more complicated and requires use of expensive equipment such as a MP30 acquisition unit and BIOPAC system.

After taking into consideration the advantages and limitations of each of the methods discussed in the literature review section, the design approaches the problem of accurately classifying sub-vocal EMG signals by improving upon the design presented in paper “Sub-vocal Phoneme-Based EMG Pattern Recognition and its application in

Diagnosis" by Mosarrat Jahan and Munna Khan. For example, the design follows the same methods of extraction and processing of the subvocal EMG signal as that of the design in the paper, but improves upon it by adding more features (inputs) from the subvocal EMG signal as inputs for the classification neural network. In theory, this allows the neural network to more easily distinguish between different subvocal EMG signals by taking into account more properties about the subvocal EMG signals when comparing them. In addition, another improvement added is to change the "linear classifier" (assumed perceptron) to a neural network that can handle non-linearly separable relationships between the inputs and outputs, thus allowing for even more classification accuracy. Of course, despite these advantages, this approach has some limitations as well, as the increase in the number of inputs into the neural network and the increase in complexity of the neural network results in a much larger use of memory and processing power, but since we are focusing on improving accuracy and are not restricted to a computer to process and classify the subvocal EMG signals that is restricted in terms of memory and processing power, these limitations can be ignored.

After passing the input EMG signals from the throat through the amplifiers/filters, Matlab is then used to process and classify the subvocal EMG signals. Matlab provides both the neural network and the wavelet transform, but this will be changed later when the microcontroller implements the wavelet transform. For now, Matlab provides the wavelet transform to test the performance of the project without the microcontroller. This is done by using the function "audiorecorder" to sample the signal entering the USB port. The user (person whom the EMG electrodes are attached to) is then sent messages to begin emitting the subvocal "vowels" by thinking of the vowel asked for (for example, "recording "AAAA" in three seconds"). The labeled data consists of five different vowels ("A", "E", "I", "O", an "U"). Twenty samples of each vowel are then recorded (100 samples total, one second recording for each vowel sample) in which the data points taken for each sample are organized into respective matrices for each sample. These signals are sampled with a sampling frequency of 1000 Hz in 16 bits as a high frequency sampling rate is not needed as the signal is already being filtered from 10 Hz to 450 Hz and 16 bits is the minimum amount of bits needed to properly represent the signal after it is converted from an analog signal to a digital signal without using up too much memory. The data points of these signals are then processed using maximal overlap discrete wavelet analysis (maximal overlap is used because Matlab only allows single levels for the discrete wavelet transform function) down to three levels or coefficients using mother wavelet Daubechies 'db4' as the wavelet function (this is removed later when the microcontroller is implemented). This splits each of the sample data points into four different rows (L+1-by-N matrix) of wavelet coefficients for scale 2^{L-1} (wavelet coefficients d1, d2, and d3), where L is the level and N is the input signal length, with the fourth row containing the scaling coefficients for 2^{Lmax} (wavelet coefficient A3). With the discrete wavelet analysis having separated the sample signals into individual frequency components, the non-stationary nature of subvocal EMG signals can be more easily analyzed, allowing for increased classification accuracy as a result. Next, since the neural network must receive inputs to characterize these sample signals, six features of each wavelet coefficient row are calculated, resulting in a total of 24 inputs to the neural network (six features for every four rows of wavelet coefficients). These features are the mean absolute deviation of the signal, the root mean square of the signal, the variance of the signal, and the standard deviation of the signal, the number of zero-crossings of the signal (number of times the signal reaches zero), and finally the number of slope changes of the signal. These features are referenced from “The use of Artificial Neural Network in the Classification of EMG Signals” by Md. R. Ahsan, Muhammad I. Ibrahimy, and Othman O. Khalifa [35] (with the exception of the mean absolute deviation of the signal) due to their demonstrated ability to represent statistical time and time-frequency based features of the sample signals (for example, the number of zero-crossings gives an indication of the number of high or low frequencies in the signal due to there being more zero-crossings for high frequency signals). Finally, once the features have been calculated, the neural network is declared (“declared” meaning “defined” in Matlab coding terms). The neural network is a “patternnet” type neural network due to the fact that these types of neural networks are specially designed by Matlab to handle classification and pattern recognition and consist of three layers: the input layer (24 neurons for 24 inputs), the hidden layer (20 neurons to be close in number to the number of inputs), and the output layer (5 neurons each representing a vowel output, for instance 10000 for “A”, 01000 for “E”, etc.). The activation function of the neurons in the hidden is the hyperbolic tangent function (‘tansig’) for the purpose of handling the fact that the relationship between the inputs and the outputs is not linearly separable, while the activation for the output layer is a linear function (‘purelin’). The training algorithm used is the Levenberg-Marquardt algorithm as it is one of the fastest methods for training a moderately sized neural network and is set to train for 100 epochs, an epoch being a single pass of the neural network through the all data used for training, with the desired MSE being 1E-12. All the biases and weights between the input layer, hidden layer, and output layer are initialized randomly. Finally, the neural network is initialized using the "configure" function and then trained using the "train" function with the INPUT matrix holding all the input features and the OUT matrix holding all the expected outputs of the input features. The 70% of the data is used for training the neural network, 15% is used to test neural network, and the final 15% is used to validate the results out of all 100 samples. Code is shown in Figure A.5 (wavelet transform code removed due to the large size of the code and because it was not included in the final design of the project) in appendix A.

4. IMPLEMENTATION, TESTING, AND RESULTS.

This chapter covers the implementation of the designs created in chapter 3, the testing of said designs and changes made to have them working properly, and the final results.

4.1 Amplifier/Filter Prototype and Testing

In order to be able to obtain an EMG signal to test the wavelet filter with, it was decided to build and test a prototype of the amplifier/filter circuit. The circuit was constructed as shown in Figure 3.1 (shown in chapter 3) on a circuit breadboard using components fixed in the node holes on the breadboard. A schematic was also drawn on paper to help label connections. The circuit was initially tested by using an Agilent 33120A function generator to simulate the circuit's response to an EMG input signal by using small signals (about 20 mV in amplitude) at a range of frequencies as inputs to the circuit. The output response of the circuit would be monitored using an Agilent DSOX2014A oscilloscope in order to see if the circuit properly filters out signals with frequencies outside the range of 10 Hz to 450 Hz and provides enough gain to amplify a signal within the 10 Hz to 450 Hz frequency range to a 0 to 5 V amplitude range (2.5 V being the reference voltage). The ±5 V voltage rails were given using an Agilent E3631A power supply.

Unfortunately, the initial testing gave no results as the circuit would not produce an output. Upon using an Agilent 34410A multimeter to probe nodes of the circuit in order to test for any short or open circuit faults, it was discovered that all node voltages were stuck at the 2.5 V reference rail. Suspecting that the reference rail might be the problem, the 2.5 V reference rail was then removed with all of its previous connections set to ground. Further testing proved that it was not the issue (all node voltages simply went to zero), but it was decided to keep it removed until the rest of the circuit could prove that it could at least operate at a ground reference voltage.

After failing to find any short or open faults, each part of the circuit was separated into individual circuits (both INA128p instrumentation amplifier circuits, both mc33078pc low-pass high gain filters, and both high-pass circuits: six circuits total) in order to test which part of the entire circuit was not operating properly. At this point in the testing process, due to a change in location, difficult testing equipment had to be used for the testing procedure. The Agilent DSOX2014A oscilloscope was replaced by a SainSmart DSO note II DS202 oscilloscope, the Agilent 22120A function generator replaced by a Lab-Volt AA 777 function generator, the power supply replaced by a TP-30003D-3 0-30V-0-3A DC power supply, and the Agilent 34410A multimeter replaced by a Cen-tech 7-Function digital multimeter. By using the function generator to input a signal below 450 Hz into the low-pass high gain filter, it was promptly discovered that the filter was producing no output via a faulty mc33078p chip. Upon replacing the chip with a new one, both low-pass high gain filters were able to function properly based on their input signals as shown in Figure 4.3. Using the function generator to input a test signal into the INA128p instrumentation amplifiers (differentially into Vin^- and Vin^+ ports), it was also discovered that the instrumentation amplifier was not producing an output. Upon examining the schematic of the INA128p instrumentation amplifier [28], it was discovered that the initial assumption that the reference voltage port of the instrumentation amplifier should be put at the high rail (+5 V) would not allow for proper operation. To solve this, the reference voltages on both amplifiers were connected to ground instead to prevent the output signal from being clipped due to the reference voltage being at the top of the rail. In addition, a load resistance of 1 KΩ was put across the output of the amplifier to ground to allow the reference voltage to have the same ground as that of the output voltage and to make sure that the output voltage is properly referenced to ground. It was also realized that a mistake was made by setting the negative rail port of the amplifier to ground. This was fixed by connecting the negative rail port to the negative rail (-5 V), thus allowing the signal to not get clipped when having a negative amplitude.

After adding these corrections and bringing the circuit back together (minus the 2.5 V rail), the circuit was tested once again using the function generator and was able to produce an output. However, as shown in Figure 4.4, the output signal, even without an input signal, was filled with many layers of noise, the strongest being a large sine-wave like signal at 60 Hz (about 200 mV). This is due to the large gain of the circuit amplifying the 60 Hz electromagnetic noise from surrounding electronic equipment. Upon measuring every node of the circuit using the oscilloscope, it became clear that the 60 Hz noise was prevalent throughout the circuit in which, although it would cause a loss of information in the output, it was decided that a notch filter at 60 Hz should be added at the output to cancel out the noise as the noise was larger than the expected input signal and drowned out any useful information as a result of its large size.

The design of the notch filter was designed using three resistors and three capacitors to create a passive notch filter (an active design was avoided in order to prevent more noise from being produced) with a mc33078p voltage follower amplifier on the output on the notch to act as a buffer to keep the output impedance low as shown in Figure 4.1. Calculations and design for the notch filter are shown as below and in Figure 4.1:

$$fc = \frac{1}{2\pi RC} = 60\ Hz = \frac{1}{2\pi R(10000\ \mu F)}\ , R = 265.25824\ \mathrm{K\Omega}$$

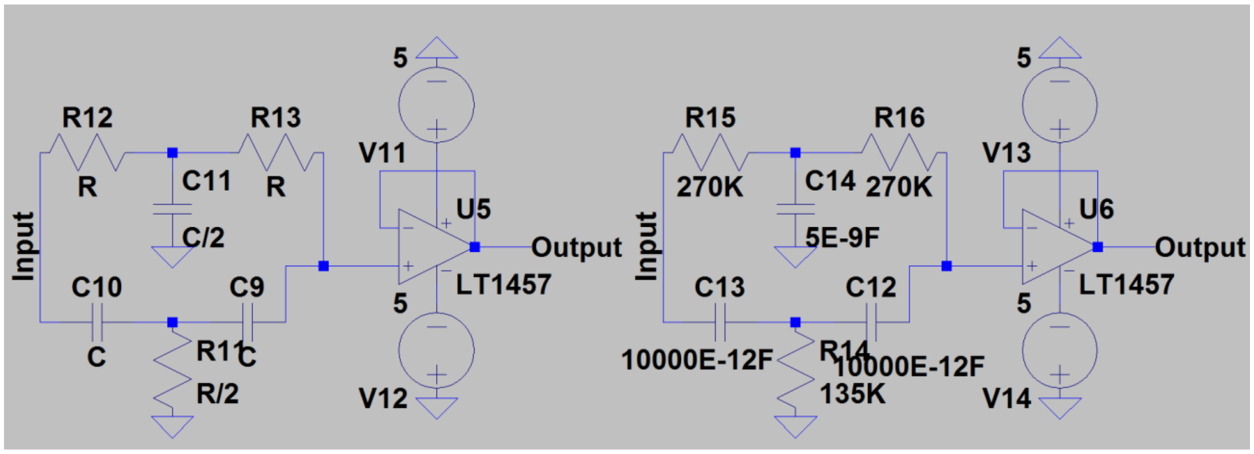


Figure 4.1: 60 Hz notch filter schematic and design.

After implementing the 60 Hz notch filter on the output, further testing on the output of the filter showed that the 60 Hz noise had been reduced to zero. However, other layers of noise still remained, the next largest being a sawtooth waveform of about 200 mV in amplitude and 200 Hz in frequency on the output. This was due to inductance in the power strips connecting the power supply and function generator to their power sources (this was tested with multiple power strips and wall sockets to confirm that it was not just one specific power strip or wall socket causing the problem). To remove this noise, the power strip was removed and the power supply and function generator were both plugged directly into a wall socket. After removing said noise, only white noise of about 80 mV in amplitude remained, which was removed by putting more capacitors (47 uF) across both voltage supply nodes to ground. The circuit was tested once again using the function generator to see if a clean signal could be acquired. The result of these tests are shown in Figure 4.5, showing a successfully clean signal.

Now that the layers of noise that occurred without an input were removed, it was time to try adding the EMG sensor electrodes to the circuit in order to obtain some data. This was done by using a 3 pad sensor cable, audio jack (3.5mm), audio jack breakout circuit, and various biomedical sensor pads. The audio jack breakout circuit was soldered to the audio jack, while wires acting as inputs to the circuit itself were soldered into the connections on the audio jack breakout circuit to assure connection to the circuit. The sensor cable would then be plugged into the audio jack, allowing the EMG sensor electrode pads to input signals into the circuit.

With the EMG sensor electrodes added to the circuit, the circuit was then tested to see if it could successfully obtain, filter, and amplify actual EMG signals obtained using the sensor electrodes. This was done by first attaching EMG electrode sensor pads attached to the ends of the three sensor cables (one acting as ground for the other two positive lead sensor pads) to a person’s throat. Then said person would speak or think in order to see if the circuit could output a signal in response. This was tried in two configurations only using two of the three electrodes, the first being that two EMG electrodes would be put to the right and left of the larynx (5 cm from the larynx) on the throat, making sure that the two positive lead electrodes are on throat muscles, while the ground electrode would be put behind an ear in order to capture EMG signals going down both sides of the throat. The second configuration was done by placing a single electrode on one side of the larynx (5 cm from it), while the ground electrode would be placed on the other side (5 cm from it) in order to capture an EMG signal going across the vocal cords. Unfortunately, more layers of noise were obtained when trying to obtain a signal, the first being a very large waveform (about 8 V in amplitude on the output) that was due to the circuit amplifying the current signal produced by the magnetic field produced by the wires of the three sensor cables moving relative to each other when picking up a signal. This was solved by taping down the wires when testing so that they would not be able to move. The second layer of noise was a layer of white noise of about 40 mV which was a result of the voltage difference in referenced grounds for the circuit and the EMG electrodes. This was solved by adding plate capacitors (10 uF) (not electrolyte capacitors, they only added noise) in series between the EMG electrodes and the circuit itself in order to DC couple out any dc voltage difference between the ground of the EMG electrodes and the ground of the circuit. The resulting circuit is shown below in Figure 4.2.

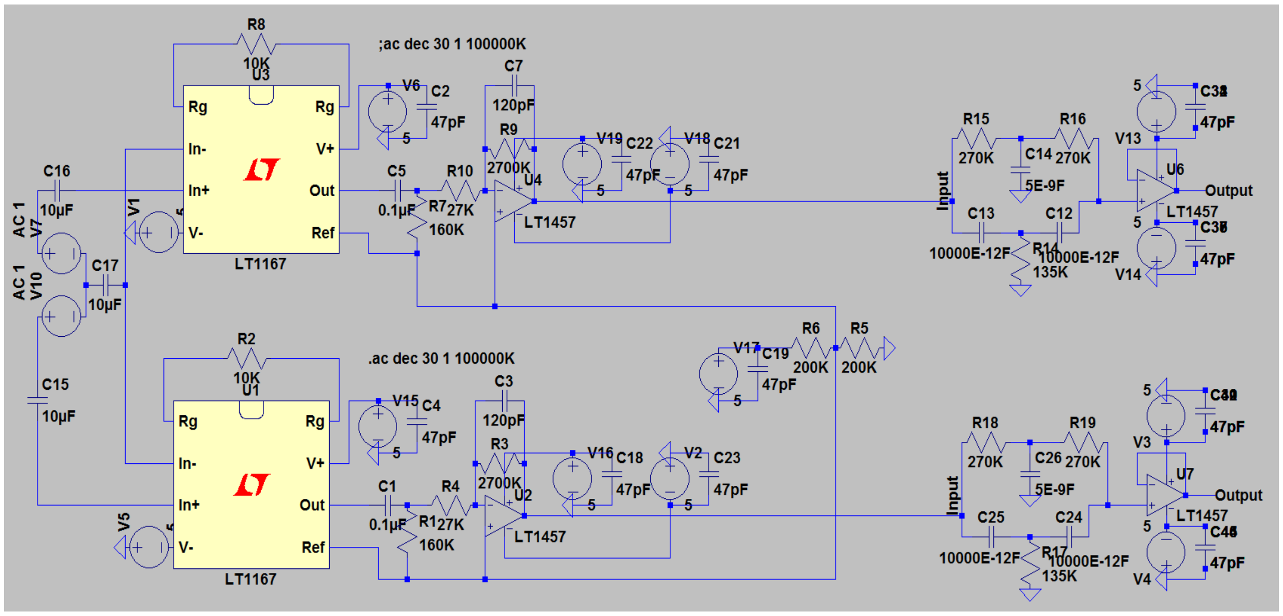


Figure 4.2: First amplifier/filter schematic and circuit.

Finally, after removing these layers of noise, a signal could be obtained (about 80 mV in amplitude), but only while the user was talking, coughing, or clearing their throat and not while using subvocalization (the second configuration of the EMG electrodes proving to be the most successful for obtaining this signal). The circuit will have to have its gain increased and needs further testing in order to obtain the subvocal signal.

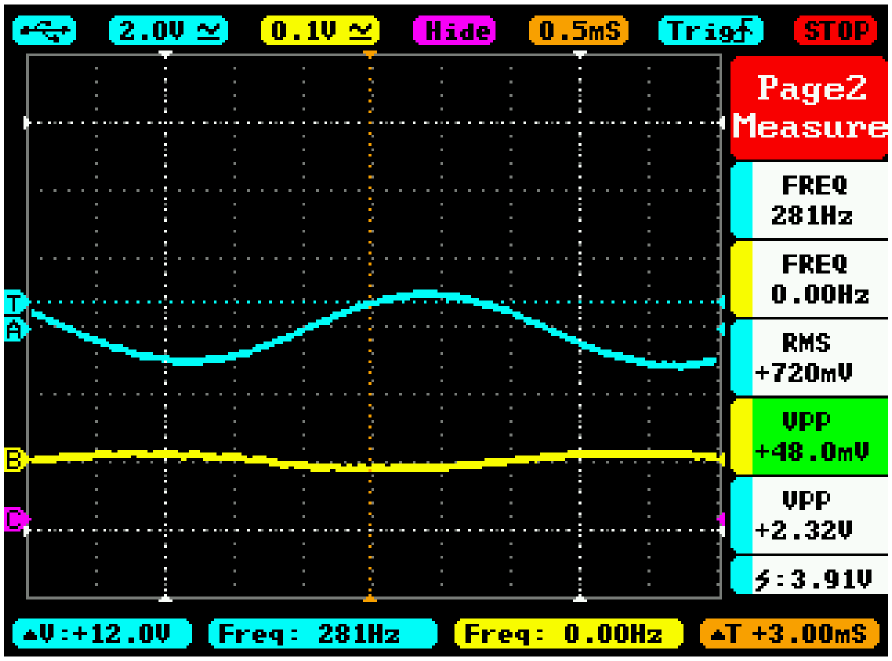


Figure 4.3: Output signal (blue) of both low-pass high gain filters in response to a 50 mV, 300 Hz input signal (yellow).

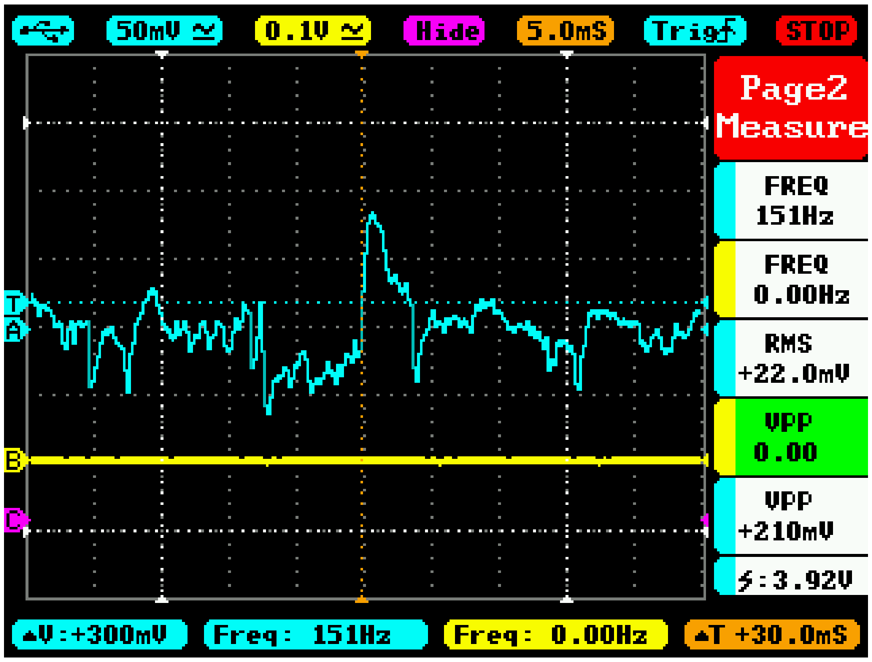


Figure 4.4: Demonstration of noise layers (blue) on output of amplifier/filter circuit.

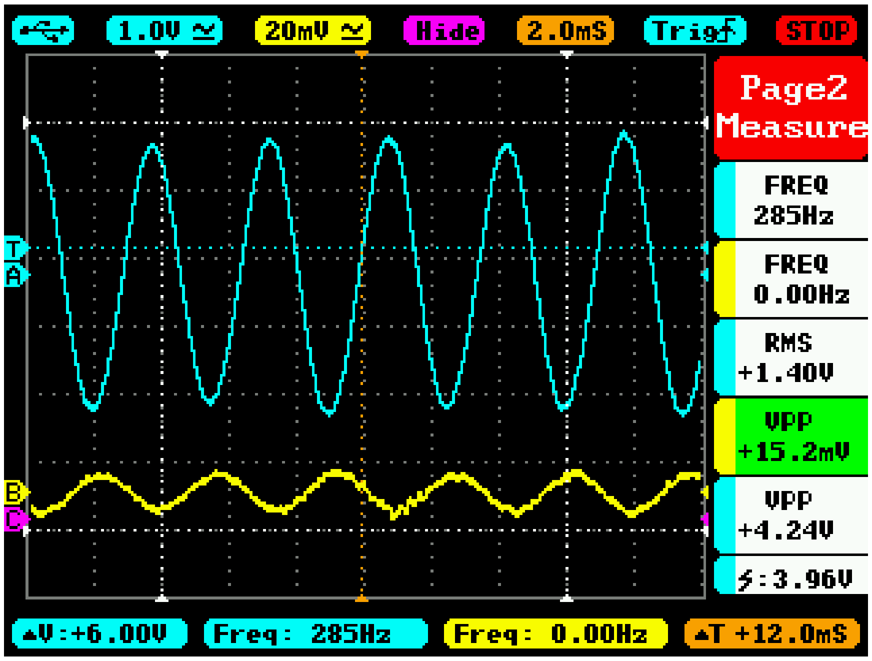


Figure 4.5: Demonstration of clean output (blue) in response to a 15 mV, 300 Hz input signal (yellow) of both low-pass high gain filters after modifications on amplifier/filter.

MSP432 code:

Changes that still need to be made to code:

1) Fix loops in decomposition filter and reconstruction filter so they loop forever.
2) Set up button to allow user to choose the output from the decomposition filter or the reconstruction filter.
3) Make sure functions are declare/called correctly.
4) Disable interrupts until outputs are made.
5) See how not performing in real time affects its performance.
6) Find out how long the code takes to carry itself out in seconds.

In order to obtain a subvocal signal, it was decided that the gain for the amplifier would need to be increased as signals with much larger expected magnitude were being picked up (coughing, throat clearing, etc.). For this reason, a potentiometer was added in parallel to the 27 KΩ resistor connected to the negative terminal of the op-amp of the amplifier (Figure 4.2) in order to raise the gain of the second amplifier by making the resistance on its negative terminal smaller. The amplifier/filter was tested using a voltage divider on the input of the amplifier/filter to bring the input waveform amplitude down to magnitudes that would more closely simulate the magnitude of a subvocal EMG signal, while the Lab-Volt AA 777 function generator supplied a sinewave input waveform and the TP-30003D-3 0-30V-0-3A DC power supply provided the rails to the amplifier/filter. The voltage divider consisted of two resistors (R1 = 20000 Ω and R2 = 100 Ω) in the configuration as shown in Figure 4.6, resulting in the voltage gain of G = R2/(R1 + R2) =

100/(20000 + 100) = 4.975124378E-3 for the input waveform signal from the function generator.

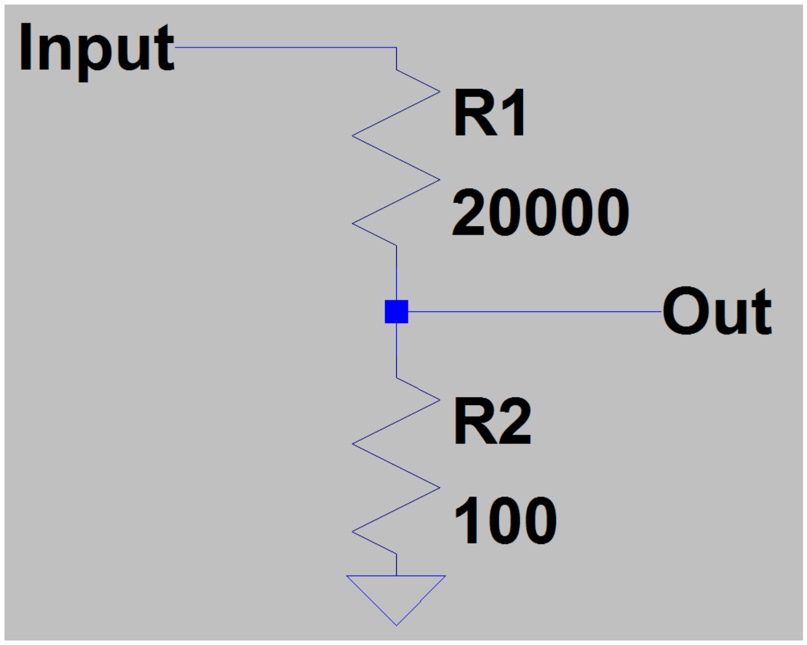


Figure 4.6: Circuit schematic of voltage divider on input of filter/amplifier.

The potentiometer was adjusted to give a variety of larger gains to the amplifier of the circuit, but, unfortunately despite amplifying the signal, the output signals resulted in having the harmonics of the 60 Hz noise signal amplified, resulting in large noise signals at frequencies of 120 Hz, 180 Hz, etc. These noise signals made it difficult to analysis any possible subvocal signals when testing using EMG electrodes and filtering them out in addition to the already filtered out 60 Hz noise would result in a much larger loss in possible information (useful signals at these filtered frequencies in the subvocal signals would be lost) so it was decided to remove the potentiometer and find another way to find the subvocal signals. The results using the voltage divider without the potentiometer are shown below in Figure 4.7 and 4.8, proving that the amplifier/filter can amplify signals around the magnitude of subvocal EMG signals up to 1.28 Volts in amplitude, while the results with the potentiometer are shown in Figure 4.9 and 5.0 (the harmonics signals were shown at a much lower amplitude signal the signal generator). The results were taken using the SainSmart DSO note II DS202 oscilloscope.

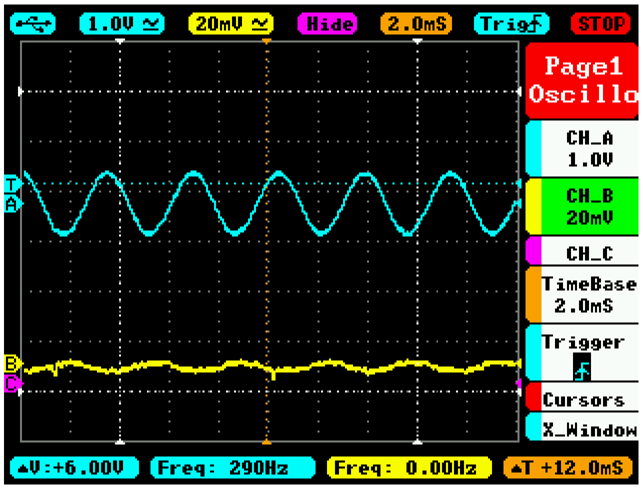


Figure 4.7: Output of filter (blue) with voltage divider to simulate EMG signal, input (yellow) after voltage divider. Channel properties are shown on the right.

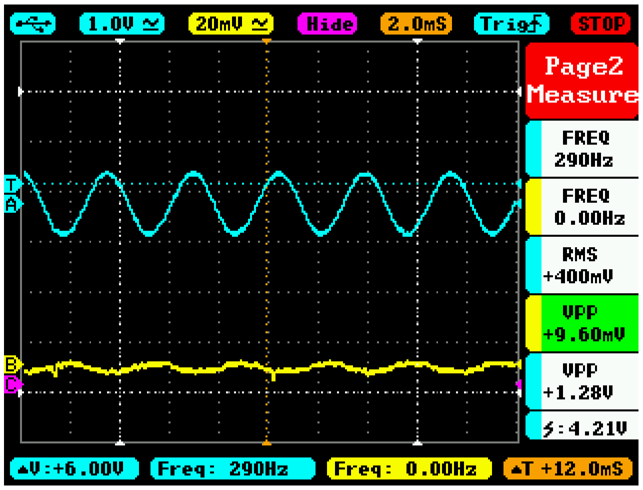
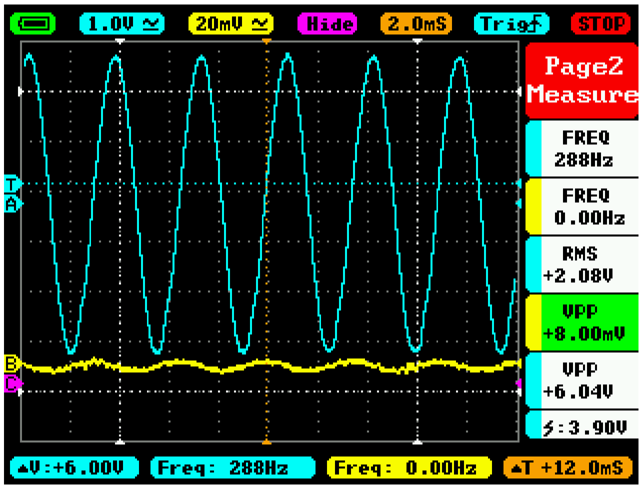


Figure 4.8: Output of filter (blue) with voltage divider to simulate EMG signal, input (yellow) after voltage divider. Signal measurements are shown on the right.


1.0V ≃ 20mV ≃ Hide 2.0mS Trig STOP

Page2 Measure

FREQ 288Hz

FREQ 0.00Hz

RMS +2.08V

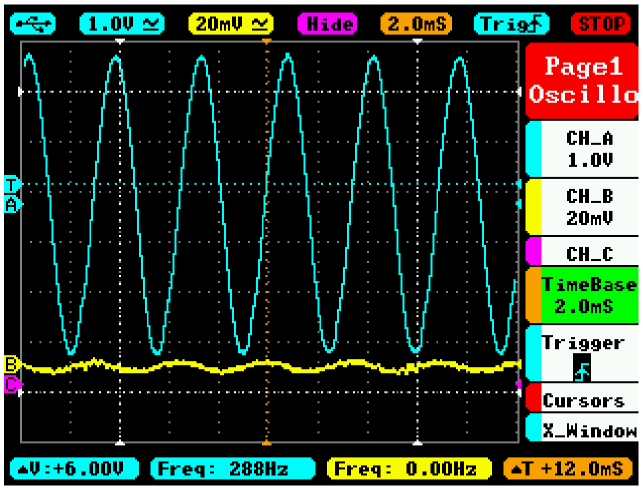
VPP +8.00mV

VPP +6.04V

⚡:3.90V

▲V:+6.00V Freq: 288Hz Freq: 0.00Hz ▲T +12.0mS

Figure 4.9: Output of filter (blue) with voltage divider to simulate EMG signal and potentiometer added to low-pass filter (in parallel with 27K resistor, resulting in total resistance of 910 ohms), input (yellow) after voltage divider. Gain increased massively (function generator attenuator turned to max). Signal measurements are shown on the right.


1.0V ≃ 20mV ≃ Hide 2.0mS Trig STOP

Page1 Oscillo

CH_A 1.0V

CH_B 20mV

CH_C

TimeBase 2.0mS

Trigger

Cursors

X_Window

▲V:+6.00V Freq: 288Hz Freq: 0.00Hz ▲T +12.0mS

Figure 4.10: Output of filter (blue) with voltage divider to simulate EMG signal and potentiometer added to low-pass filter (in parallel with 27K resistor results in 910 ohms), input (yellow) after voltage divider. Gain increased massively (function generator attenuator turned to max). Channel properties shown on the right.

Now that the amplifier/filter had proven that it could successfully amplify signals at the magnitude expected of subvocal EMG signals to a useable magnitude, the amplifier/filter was tested under the same conditions only using actual subvocal signals recorded using EMG electrodes on the throat as input instead of signals from a function generator (the voltage divider was removed of course when testing for actual subvocal EMG signals). Many different configurations of positions of the EMG electrodes on the throat were tried, including the two configurations mentioned previously, but these resulted in either getting no signal at all or signals that should not resemble subvocal EMG signals (for example, DC voltage). Finally, the best results were obtained by placing the ground EMG electrode behind the left ear, while the red sensor was placed just underneath the chin on the left side of the face, making sure that the EMG electrode was over the location of the left anterior belly of digastric muscle (shown in Figure 4.11).

In addition, the person used as the test subject was grounded by using a cable strap strapped on the wrist that was clipped to a large piece of metal (metal box). Although the anterior belly of digastric muscle is not the only muscle one can receive these subvocal EMG signals from, it produced the clearest results as shown below in Figures 4.12, 4.13, 4.14, 4.15, 4.16, and 4.17, which demonstrate signals taken from vocally speaking the vowel “a”, clearing the throat, doing nothing, subvocalizing “a”, subvocalizing “e”, and subvocalizing “o”.

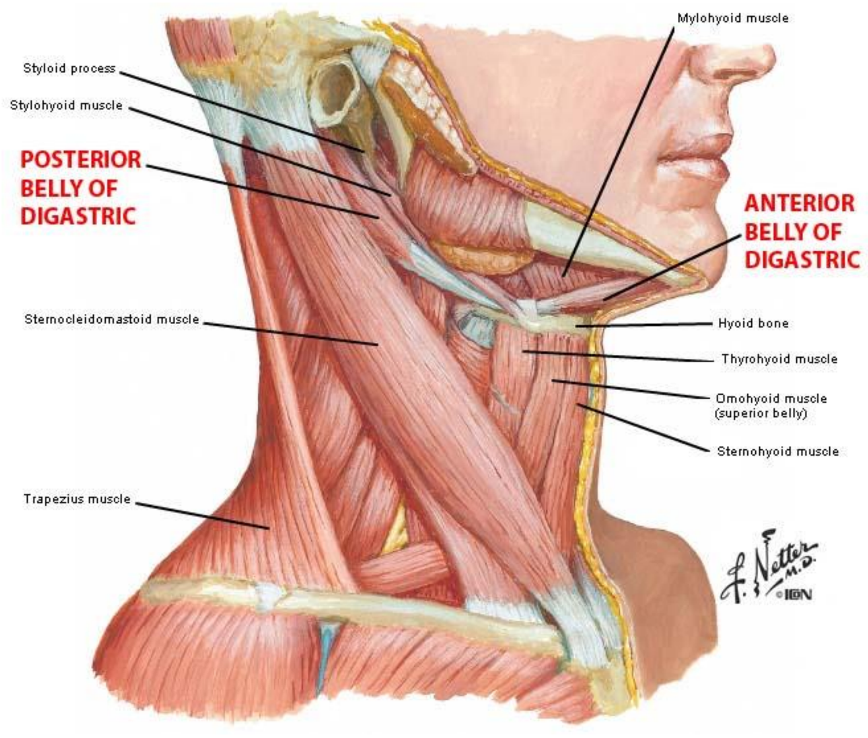

Mylohyoid muscle
Styloid process
Stylohyoid muscle
POSTERIOR BELLY OF DIGASTRIC
ANTERIOR BELLY OF DIGASTRIC
Sternocleidomastoid muscle
Hyoid bone
Thyrohyoid muscle
Omohyoid muscle (superior belly)
Sternohyoid muscle
Trapezius muscle

Figure 4.11: Muscles of the human throat, showing the location of the anterior belly of digastric.

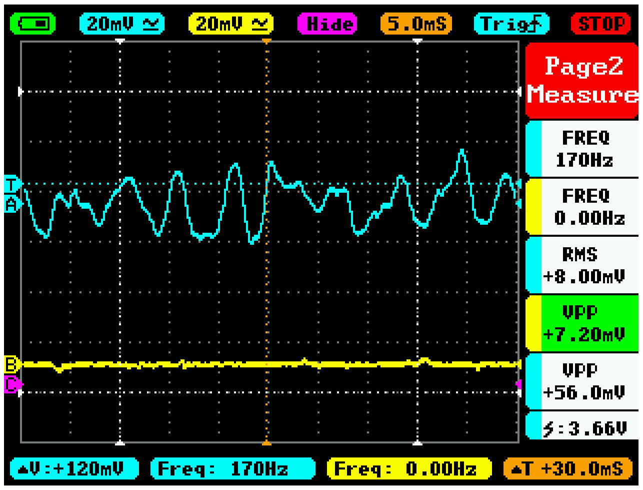


Figure 4.12: First experiment: signal taken while vocally talking (“aaaaa”).

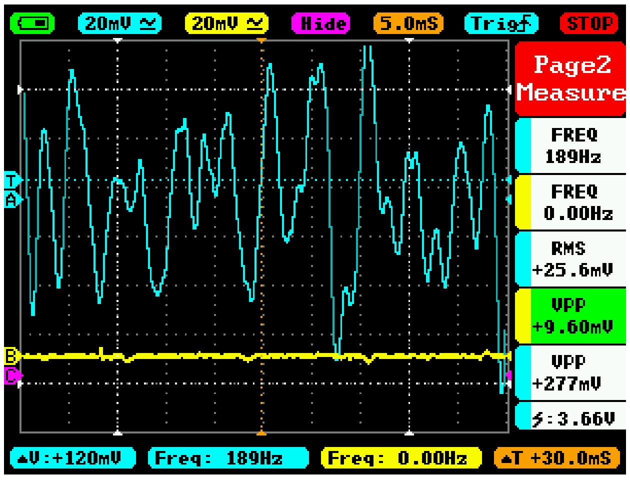


Figure 4.13: First experiment: signal taken while clearing throat.

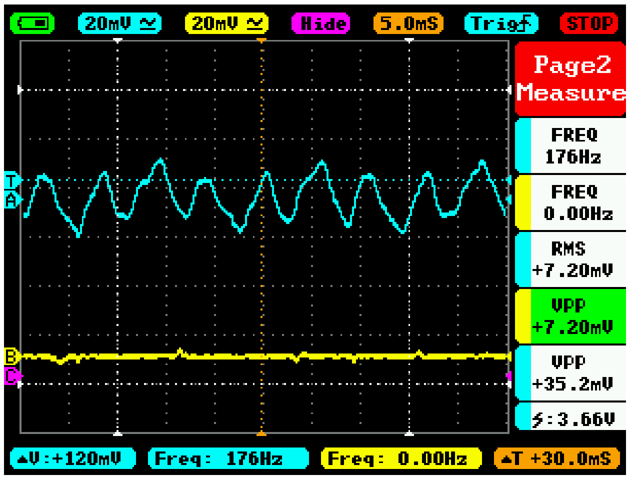


Figure 4.14: First experiment: signal taken while doing nothing (EMG of throat).

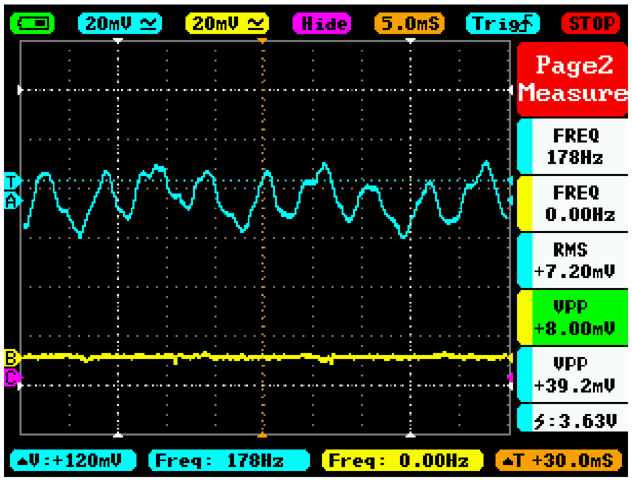


Figure 4.15: First experiment: signal taken while subvocalizing “eeeeeee”.

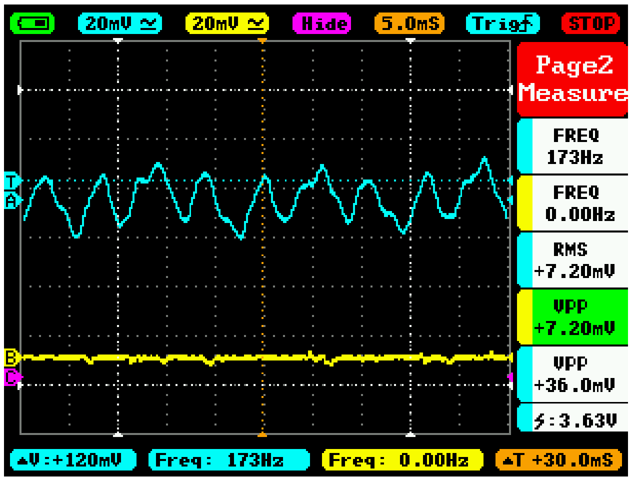


Figure 4.16: First experiment: signal taken while subvocalizing “oooooo”.

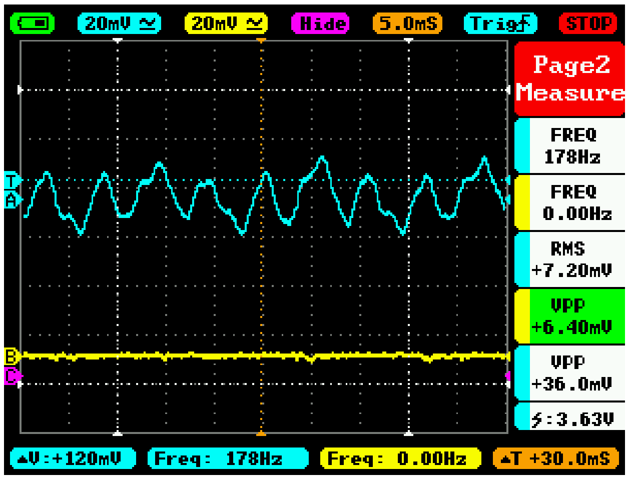


Figure 4.17: First experiment: signal taken while subvocalizing “aaaaaa”.

Upon examining the results, it is clear to see that the these signals needed to be further processed before any discrepancies between these signals could be found as no significant differences can really be seen between any of the subvocal signals and the signal retrieved from doing nothing so it is still unclear even if a subvocal signal exists within these signals. However, vocally speaking and clearing the throat showed clear signs of EMG signals produced from these actions as both of the signals captured after vocally speaking and clearing the throat showed clear increases in magnitude. This proved that EMG signals were being captured, but finding the discrepancies between the subvocal signals required further processing as the subvocal signals clearly did not cause significant changes in magnitude. Further testing was tried by removing the low-pass capacitor to see if removing some filtering would allow for more of the subvocal signal to be shown, but, as expected, no changes were observed between the EMG subvocal signals taken. Thus, since it was clear that the amplitude/filter was then operational, testing the wavelet transform code was the next step. The wavelet transform would sort out the individual frequency components of each of the signals, allowing for further analysis between the differences between the signals to be possible.

4.2 Microcontroller Code Implementation V1.0 (User Interface V1.0)

For the testing of the wavelet transform code, the operation of the buttons that controlled what kind of output would be received and when data was being taken was done first. A debouncing circuit was designed and created on a breadboard to handle any unwanted multiple button presses when manually pressing the buttons (buttons generally trigger more than once when pressed, thus resulting in error) by filtering out the excess signals created after pressing the button. Design of the debouncing circuit is referenced from [38] in which Figure 4.18 shows the design, where the switch represents the button, the output is between the R2 resistor and the C2 capacitor, and ground is below R1, C2, and the voltage source (negative terminal). The button sends its signal (when it is pressed) when the switch is closed, causing the voltage source to send voltage across the switch and the resistors, resulting in the output signal. The RC circuit (R2 and C2 in series) then acts as a one-pole RC low-pass filter, filtering out any of the excess signals caused by the switch accidentally switching open and closed from one button press. When the switch is finally stays open again for a long period of time (button is no longer pressed), the pull-down resistor (R1) pulls the voltage signal to zero. Equations are as shown:

Time constant = RC = (10000 Ω)(390000E-12 F) = 3.9E-3 seconds.

Corner frequency of low-pass circuit: fc = 1/(2πRC) = 1/(2π(10000 Ω)(390000E-12 F)) = 0.80896 Hz (rejects high frequency signals from the switch).

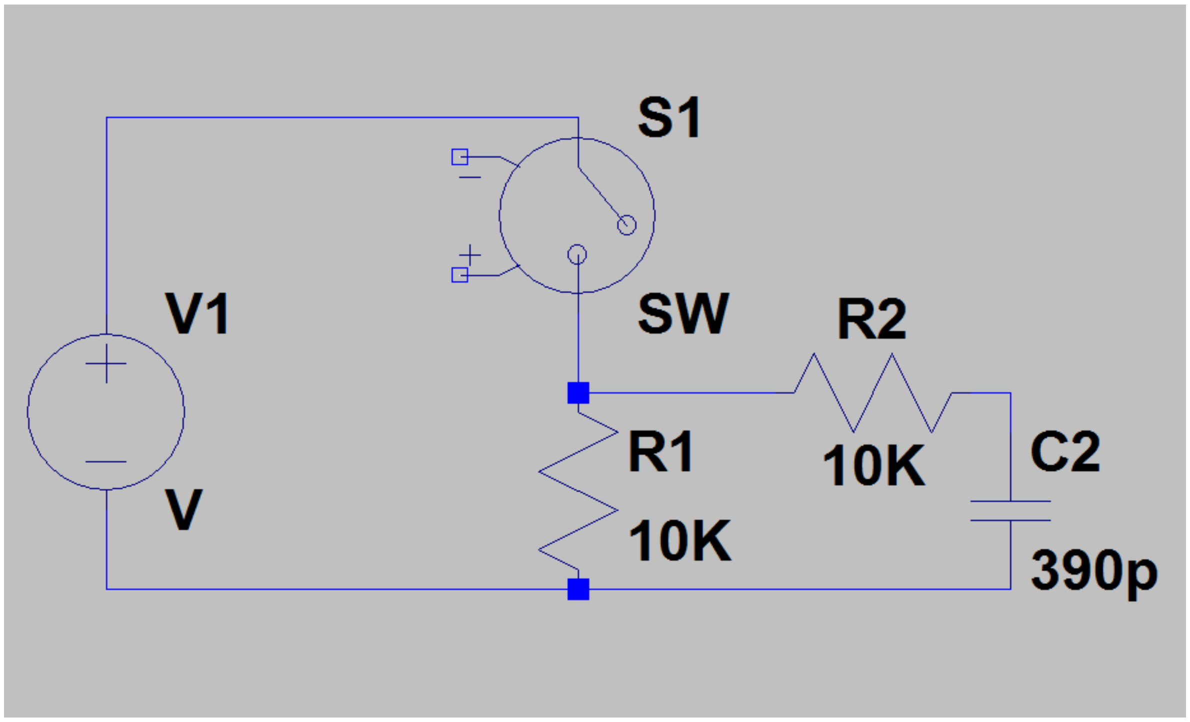


Figure 4.18: Debouncing circuit schematic.

Since the MSP432P401R microcontroller takes a positive-going input threshold voltage of 1.35 V to 2.25 V on it ports, the voltage source was set to be 2V for the debouncing circuit (this voltage was increased later). The buttons were tested by using the SainSmart DSO note II DS202 oscilloscope to capture the output of the debouncing circuit, which showed the debouncing circuit operated properly as it produced a smooth increase in voltage as the button was pressed (no excess waveforms). The waveform transform code was then used to test if the outputs from the buttons would successfully trigger interrupts and LED outputs in the program by connecting the outputs of the debouncing circuits to input ports on the microcontroller (Ports P2.4 and P3.0). Unfortunately, upon testing the waveform transform code and despite having no debug errors, the program would reset constantly before any lines of code could be processed and was thus unable to test the buttons.

As expected, despite having no debug errors, it was clear that the waveform transform code would need some testing to assure that it worked properly. To solve the problem of constantly resetting, it was decided to first test to see if the microcontroller itself was working properly. This was done by testing to see if it could handle a simple program which would cause a LED output to flash constantly when active and was referenced from the MSP432P401R microcontroller example program files, shown in appendix A as Figure A.2.

The microcontroller was able to handle this code effectively, thus showing that it can handle a program and that it was a problem associated with wavelet transform code in particular.

This was solved by adding the function below, which prevented the watchdog timer (default system clock that resets the microcontroller constantly for every one of its clock cycles) from resetting the system because it could not load the program (due its large size) into the microcontroller in time before resetting the whole microcontroller. The original line of code meant to do this, WDTCTL = WDTPW | WDTHOLD;, was not freezing the watchdog timer because the program needed to load before it could implement this line of code, but the code ("system_pre_init") below freezes the watchdog timer before the rest of the program is loaded into the microcontroller, thus preventing the microcontroller from constantly resetting. Code is shown in appendix A as Figure A.3.

Since the program was very large, further testing was done by individually separating different parts of the code into a separate program, which would be added to by other parts of the full program once the first part proved it worked properly (this was done because it was suspected that more memory problems might occur because the program was so large). The first part tested was the "main" program, which handled the initialization of functions and interrupts and contained the infinite loop state that the program would stay in when no interrupt was being called. With only one part of the code being tested separate from the other code, the microcontroller no longer reset constantly, showing that it was a problem involving the size of the program. The interrupt sections were added (functions called when an interrupt is called) and tested to see if the buttons would trigger the interrupts. Outputs to LEDs were added to the interrupt sections to test if they were being called (LEDs would blink constantly if an interrupt was called), but the interrupts did not work. Upon examining the user manual of the MSP432P401R, it was clear that a mistake was made regarding how to initialize interrupts as it assumed that the interrupt initialization process was the same of that of the MSP430 microcontroller, but, in fact, an extra step was added to account for the fact that the

MSP432P401R can handle multiple interrupts at once in one program. The lines NVIC_ISER1 = 1 << ((INT_PORT3 - 16) & 31); and NVIC_ISER1 = 1 << ((INT_PORT2 - 16) & 31); needed to be added to the main program to initialize interrupts for the set of input ports "three" (P3.1, P3.2,…) and for the set of input ports "two" (P2.1, P2.2, ….) as input ports in both of these sets of input ports were being used with the buttons as interrupts, while the startup file for the program needed to be edited as well to account for the initialization of these interrupts. As a result of this fix, the interrupts finally became operational, but came with other problems such as interrupts being called as soon as they were initialized and then being unable to be called again afterward until the program was reloaded, even without the buttons being pressed.

Testing was done to discover why the interrupts were being called immediately after being initialized and it was suspected, upon probing the ports used as inputs with an oscilloscope, that the initialization of the pull-up resistors for each input port was causing an increase in voltage on the input ports (initialization of the pull-up resistor essentially adds a resistor to the input circuit, causing a change in voltage as the resistor is added) that would trigger the input port to call an interrupt as soon as the interrupts were initialized as an increase in voltage on interrupt ports results in an interrupt being called. By commenting in and out the code that initialized the pull-up resistors, it was confirmed that the pull-up resistor was causing this increase in voltage. The increase in voltage on port P2.4 is shown below in Figure 4.19:

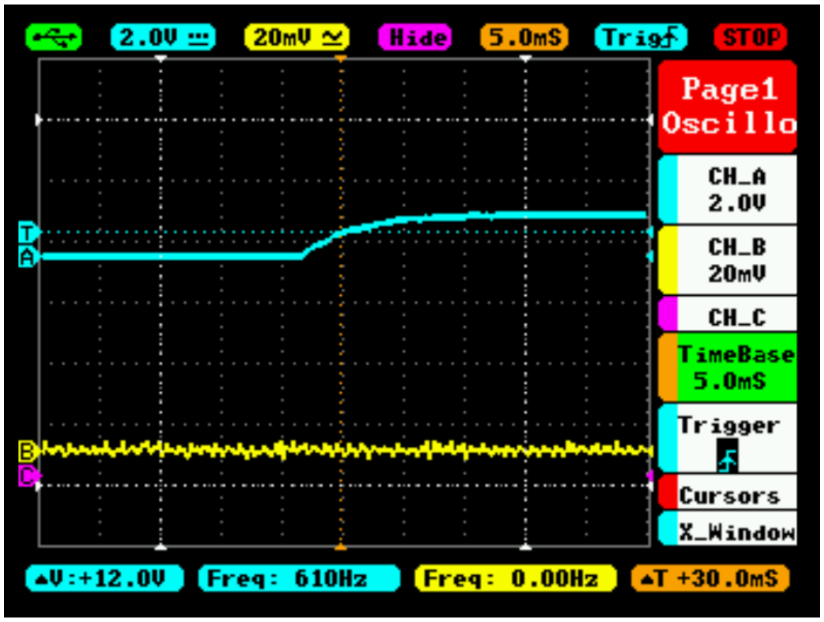


Figure 4.19: Screen capture of increase in voltage on port input P2.4 when running microcontroller. About 10 ms = 10E-3 s in time to increase voltage back to DC levels.

To solve this issue, the time it took for the voltage to increase to a DC value when the pull-up resistor was initialized in the code was recorded using the screen capture in Figure 4.19. This time was used to find the number of clock cycles it would take for this increase in voltage to become a DC value. The results were calculated as:

One cycle = 1/SMCLK = 1/24 MHz = 41.66666667E-9 s

Number of cycles to delay: 10E-3 s/41.66666667E-9 s = 240E3

These 240000 cycles were then used in the function `__delay_cycles(240000);` between the line where the pull-up resistor was initialized and the line were the interrupts were initialized, which would delay the program by the amount of time it took for the increase in voltage to become a DC value, thus preventing the increase in voltage on the interrupt input ports from occurring while the interrupts were being initialized.

Unfortunately, implementing the delay cycles did not solve the problem as the interrupts continued to be called as soon as they were enabled even after increasing the delay cycle to 300E8 cycles. Stopping the pull-up resistor from being initialized despite it being needed to receive the signal to initiate the interrupt did not solve the problem either. Thus, it was apparent that the problem had nothing to do with the pull-up resistors.

After many different tests to try and discover the source of this problem, the interrupt for port 2 stopped enabling immediately after initialization after commenting out some parts of the program (parts that had nothing to do with the interrupt) and then reprogramming the microcontroller (the interrupt for port 3 was still being enabled immediately unfortunately). It was then decided that a memory issue was causing the interrupts to be enabled immediately.

Upon examining the way in which testing was done, it was found that the practice of resetting and reloading the microcontroller to reach the first line of code each time the program reached an infinite loop implemented for testing was the cause of the memory problems as, after turning off the microcontroller and then implementing more conservative use of reloading the microcontroller, the port 2 interrupt more consistently stopped being enabled as soon as it was initialized. As for the port 3 interrupt, it was decided to use a different port for the interrupt in which P4.0 (port 4) was used. The port 4 interrupt was operational, but did not work until the voltage from button enabling it was increased to 2.3 V (now giving enough threshold voltage), which then, like the port 2 interrupt, only would be able to be enabled once and then could not be enabled again until the microcontroller was reloaded.

With the interrupts being able to operate properly at least once, the other parts of the code were added (deconstruction loop, reconstruction loop, conversion program, and drive_DAC output function) to the testing program, which were individually tested by placing infinite loops at the ends of each program to see not only if each line of code could be processed without error, but also to see if the second interrupt function could guide the code to reach the reconstruction loop output or deconstruction loop output (second interrupt determines whether or not the microcontroller will output a reconstructed output signal or discrete wavelet transform output signal). All tested parts of the code were processed without error, but the effect of the second interrupt (changing the type of output of the microcontroller) was not occurring despite the fact that the interrupt was being called. The interrupts also had a tendency to behave fickly while testing, sometimes tending to go into interrupt faults or not working at all. With the interrupts behaving fickly, only enabling once, not carrying out their functions, and the code responsible for initializing the interrupts working properly, testing was done by examining the register map in the code composer for the microcontroller to examine the individual bits in each register to see if the bits responsible for enabling and maintaining the interrupts were actually being toggled and not changing or being edited unintentionally. Results showed that, as the program was carried out, all bits were set as expected (interrupt flags for ports 2 and 4 stay at zero after being set to zero, etc.).

Despite the fact that the interrupts were behaving strangely, they were able to at least consistently enable once per reset, allowing for testing of the amplifier/filter with the microcontroller to be possible. In order to give an output that is in the input range of the microcontroller, the 2.5 V rails need to be added to the amplifier/filter to give its output signal a DC voltage of 2.5 V to make sure that the signal does not go negative in amplitude. A voltage divider was then added to the amplifier/filter as shown in Figure 2 (R5 and R6 resistors). However, because the voltage divider did not give proper results the last time it was tested with the amplifier/filter, an improvement was added as shown in Figure 4.20. A voltage follower was added between the voltage divider and the output to prevent too much current from being drawn from the voltage source. The 1KΩ loads going to ground at the outputs of the instrumentation amplifiers were also removed as these were deemed unnecessary. Upon testing the amplifier/filter with the new voltage follower, the results showed that the output of the amplifier/filter contained a DC voltage bias of 0.750 V (the notch filter does not have this bias yet). Further testing is required to get this voltage to 2.5 DC volts, but, since a bias is given, it was finally possible to test it with the microcontroller and the neural network. State of wavelet transform code after button interrupt routine adjustment is shown in appendix A as Figure A.4.

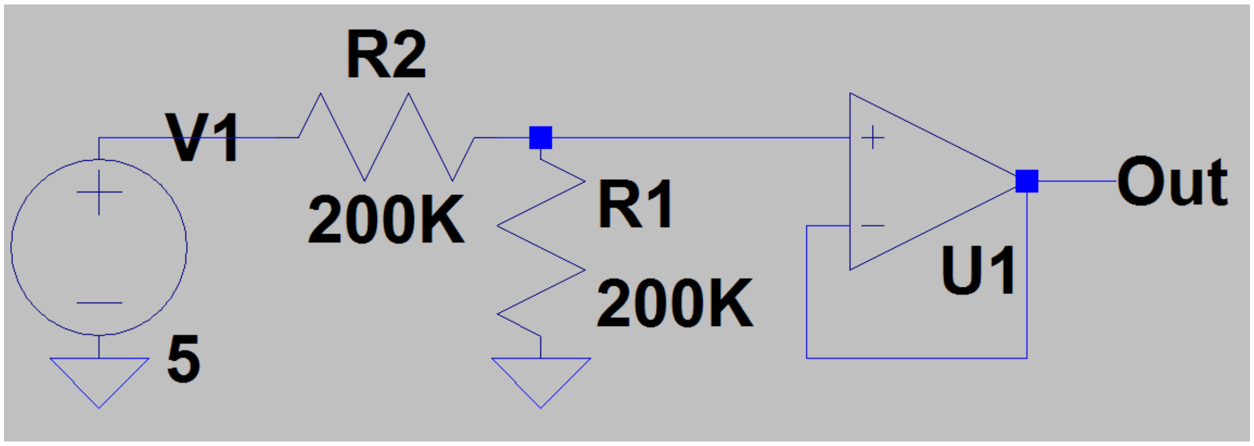


Figure 4.20: 2.5 V rail for amplifier/filter with voltage follower.

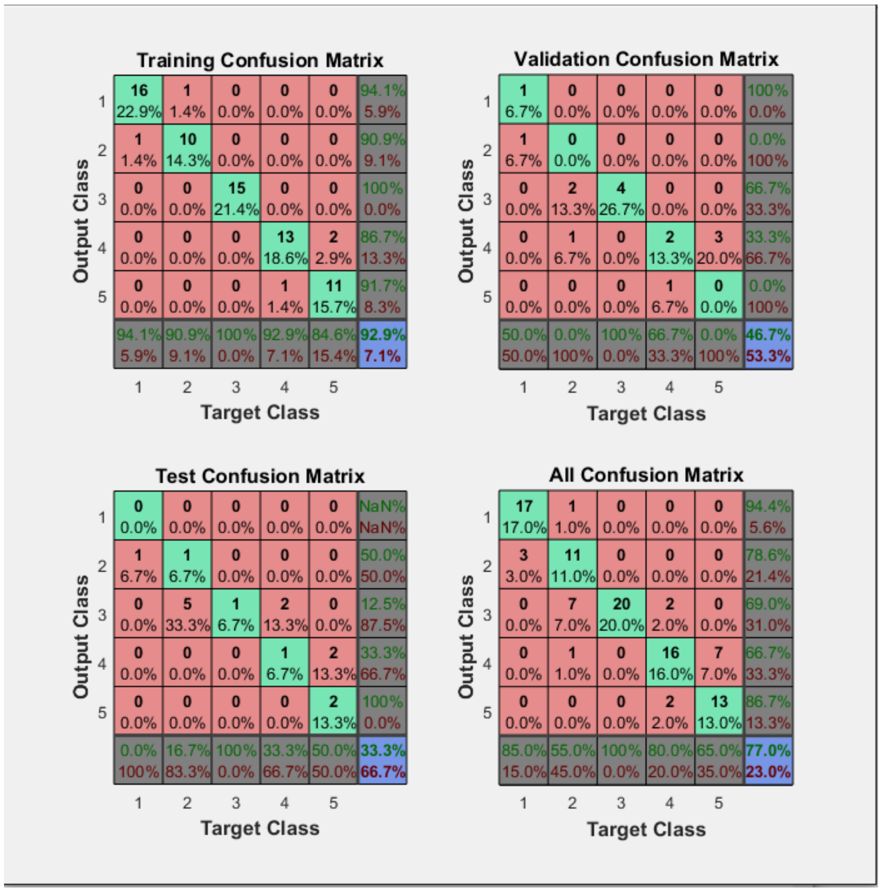
4.3 Neural Network Tests with Amplifier/Filter

Before testing with the microcontroller, the amplifier/filter was first tested with the neural network to see if the amplifier/filter could now pick up subvocal signals with enough distinction from one another that they could be classified separately since the last test using the EMG electrodes gave unclear results. With the neural network matlab code having its own wavelet transform, testing it with the amplifier/filter can confirm if the amplifier/filter can acquire EMG subvocal signals that can be classified accurately. As previously mentioned in section 3.6, the neural network works by using features extracted from subvocal signal samples to train itself to classify each of the subvocal signal samples as either one of five vowels ("a", "e", "i", "o", and "u"). The neural network is a "patternnet" type neural network due to the fact that these types of neural networks are specially designed by Matlab to handle classification and pattern recognition and consist of three layers: the input layer (24 neurons for 24 inputs), the hidden layer (20 neurons to be close in number to the number of inputs), and the output layer (5 neurons each representing a vowel output, for instance 10000 for "A", 01000 for "E", etc.). The activation function of the neurons in the hidden layer is the hyperbolic tangent function ('tansig') for the purpose of handling the fact that the relationship between the inputs and the outputs is not linearly separable, while the activation for the output layer is a linear function ('purelin'). The training algorithm used is the Levenberg-Marquardt algorithm as it is one of the fastest methods for training a moderately sized neural network and is set to train for 100 epochs, an epoch being a single pass of the neural network through the all data used for training, with the desired MSE being 1E-12. All the biases and weights between the input layer, hidden layer, and output layer are initialized randomly.

The neural network was implemented through Matlab using a laptop, involving not just the neural network but an input vector for organizing the input samples going into the neural network for each test, a wavelet transform, and feature extraction. The neural network would receive these input samples through use of a stereo cable and “iMic Griffen” audio plug in which the output of the amplifier/filter would act as the input to the stereo cable. The stereo cable would then provide the means to send the output of the amplifier/filter into the “iMic Griffen” audio plug in which said plug would act as a USB converter to send the output to the laptop for the Matlab program to receive and subsequently sample into the input vector. The samples in the input vector would be wavelet transformed and then have their features extracted to be finally used as inputs for the neural network.

For the input vector, two different methods were employed to determine how it would organize its input samples. The first being the recording of one long sample (100 seconds of recording) in which the user would be cued at certain points in the recording process when to subvocalize which vowel. Said long sample would then be divided into multiple samples and fed into the neural network according to which vowel they corresponded to. Unfortunately, confusion matrix results showed 100% accuracy every time, even when no input was being given and no signals were being recorded by the Matlab program, showing that the sampling process was creating some sort of pattern in the samples that the neural network was mistakening for disparities between each sample type which resulted in dishonest results. The second and far more successful method of sampling was simply taking twenty samples for each vowel (100 samples total), each one second in recording time for 100 seconds total.

For testing and training the neural network, the results of three different scenarios were compared: a “patternnet” type neural network using all 6 features (24 neural network inputs) as designed in chapter 3, the original method as shown in “Sub-vocal Phoneme-Based EMG Pattern Recognition and its application in Diagnosis” by Mosarrat Jahan and Munna Khan in which only two features, mean absolute deviation and standard deviation, are used (8 neural network inputs) and the neural network is a linear perceptron as opposed to a “patternnet” type, and a “patternnet” type neural network using only said two features as neural network inputs. The first scenario was tested to test the performance of the neural network intended for use for the project, while the second scenario was tested to see how the improvements to Mosarrat’s and Munna’s design improved accuracy of classification, since the design created in chapter 3 was based on their design. Finally, the third scenario was tested to see how adding more than two features improved the accuracy of classification. Accuracy of classification would be shown through a confusion matrix for each test (100 samples) in which 70% of the data is used for training the neural network, 15% is used to test neural network, and the final 15% is used to validate the results out of all 100 samples, while the inputs for the amplifier/filter were taken by placing EMG electrodes on the throat in which a single electrode was placed on one side of the larynx of the throat (5 cm from it), while a ground electrode was placed on the other side (5 cm from it) in order to capture an EMG signal going across the vocal cords while the person with said throat was speaking. The results for the first scenario are shown below in Figures 4.21 and 4.22, the results for the third scenario in Figures 4.23 and 4.24, and the results for the second scenario in Figures 4.25 and 4.26. For the confusion matrices of Figures 4.21, 4.23, and 4.25, the green and red numbers of the bottom row represent the percentage of targets (samples) that were classified correctly (green) and incorrectly (red) to a class (vowel) out of all the targets of said class (20 samples for each vowel), each cell of the row representing said percentages for each class. The green and red numbers of the right-most column represent the percentage of assignments to a class that were correct (green) and incorrect (red) out of all the assignments to a class, likewise having each cell of the column represent said percentages for each class. For the cells between said row and column, the green cells represent the number and percentage out of all 100 samples of correctly classified samples for each vowel, while the red cells represent that of the incorrectly classified samples. Finally, the overall accuracy is determined from the total percentages of each green cell, while the total inaccuracy is determined from the total percentages of each red cell.



Training Confusion Matrix

Output Class	1	2	3	4	5	
1	16 22.9%	1 1.4%	0 0.0%	0 0.0%	0 0.0%	94.1% 5.9%
2	1 1.4%	10 14.3%	0 0.0%	0 0.0%	0 0.0%	90.9% 9.1%
3	0 0.0%	0 0.0%	15 21.4%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	13 18.6%	2 2.9%	86.7% 13.3%
5	0 0.0%	0 0.0%	0 0.0%	1 1.4%	11 15.7%	91.7% 8.3%
	94.1% 5.9%	90.9% 9.1%	100% 0.0%	92.9% 7.1%	84.6% 15.4%	92.9% 7.1%

Target Class

Validation Confusion Matrix

Output Class	1	2	3	4	5	
1	1 6.7%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
2	1 6.7%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0.0% 100%
3	0 0.0%	2 13.3%	4 26.7%	0 0.0%	0 0.0%	66.7% 33.3%
4	0 0.0%	1 6.7%	0 0.0%	2 13.3%	3 20.0%	33.3% 66.7%
5	0 0.0%	0 0.0%	0 0.0%	1 6.7%	0 0.0%	0.0% 100%
	50.0% 50.0%	0.0% 100%	100% 0.0%	66.7% 33.3%	0.0% 100%	46.7% 53.3%

Target Class

Test Confusion Matrix

Output Class	1	2	3	4	5	
1	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	NaN% NaN%
2	1 6.7%	1 6.7%	0 0.0%	0 0.0%	0 0.0%	50.0% 50.0%
3	0 0.0%	5 33.3%	1 6.7%	2 13.3%	0 0.0%	12.5% 87.5%
4	0 0.0%	0 0.0%	0 0.0%	1 6.7%	2 13.3%	33.3% 66.7%
5	0 0.0%	0 0.0%	0 0.0%	0 0.0%	2 13.3%	100% 0.0%
	0.0% 100%	16.7% 83.3%	100% 0.0%	33.3% 66.7%	50.0% 50.0%	33.3% 66.7%

Target Class

All Confusion Matrix

Output Class	1	2	3	4	5	
1	17 17.0%	1 1.0%	0 0.0%	0 0.0%	0 0.0%	94.4% 5.6%
2	3 3.0%	11 11.0%	0 0.0%	0 0.0%	0 0.0%	78.6% 21.4%
3	0 0.0%	7 7.0%	20 20.0%	2 2.0%	0 0.0%	69.0% 31.0%
4	0 0.0%	1 1.0%	0 0.0%	16 16.0%	7 7.0%	66.7% 33.3%
5	0 0.0%	0 0.0%	0 0.0%	2 2.0%	13 13.0%	86.7% 13.3%
	85.0% 15.0%	55.0% 45.0%	100% 0.0%	80.0% 20.0%	65.0% 35.0%	77.0% 23.0%

Target Class

Figure 4.21: Confusion matrix when using all 24 inputs (6 features) with “patternnet” neural network for project design.

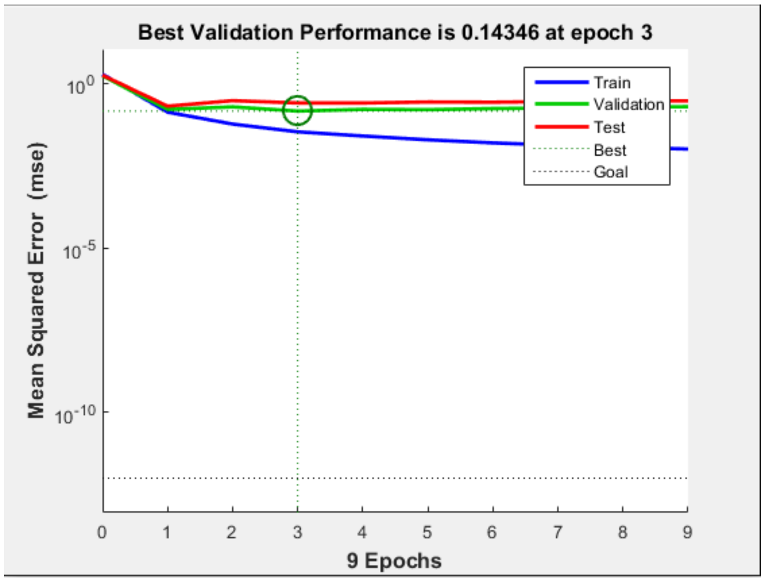


Figure 4.22: Best validation performance when all 24 inputs (6 features) with “patternnet” neural network for project design.

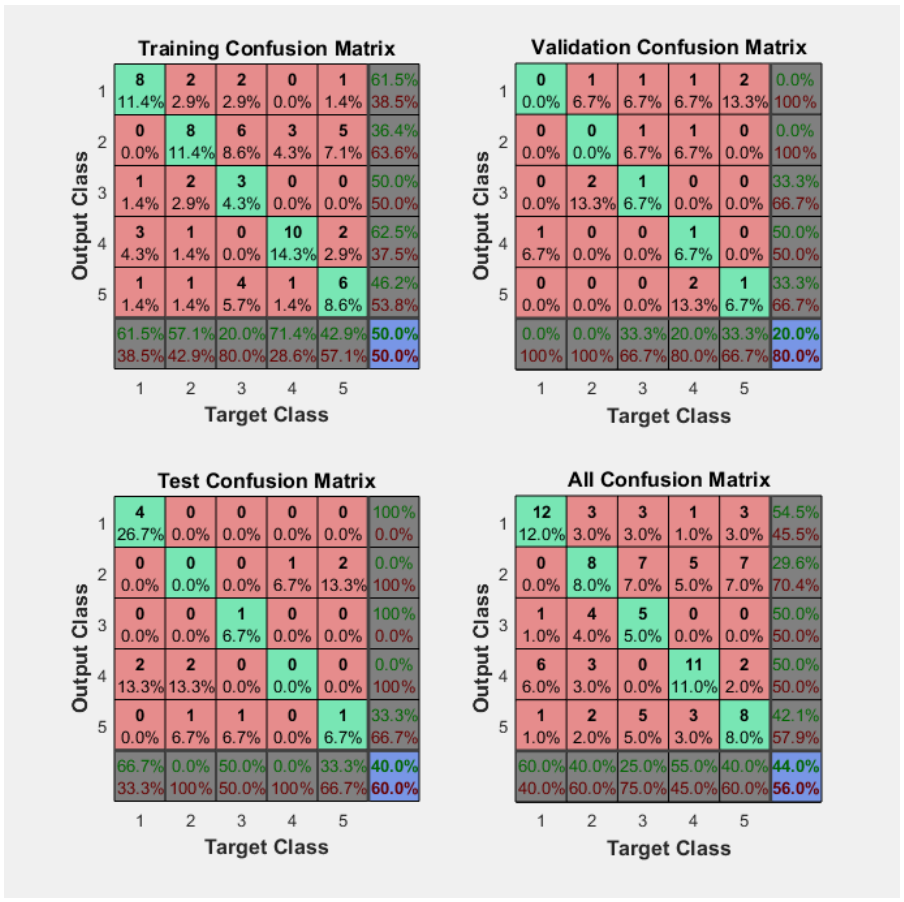


Figure 4.23: Confusion matrix when using 8 inputs (2 features) with “patternnet” neural network.

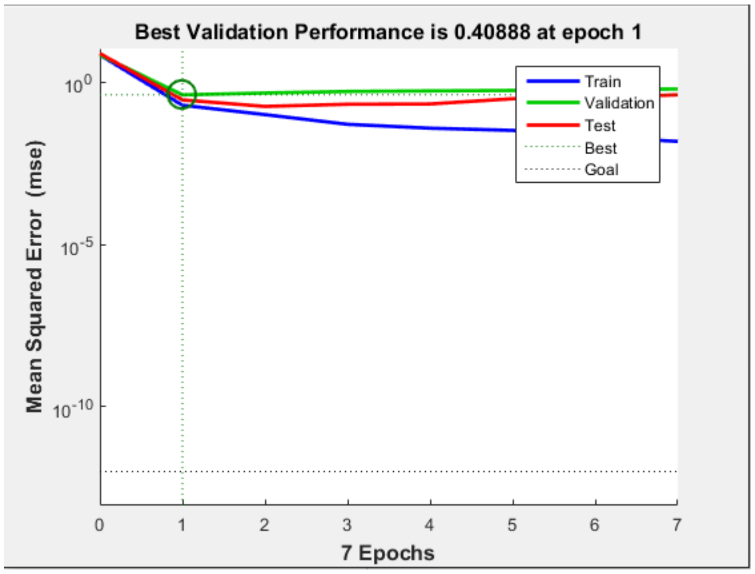


Figure 4.24: Best validation performance when using 8 inputs (2 features) with “patternnet” neural network.

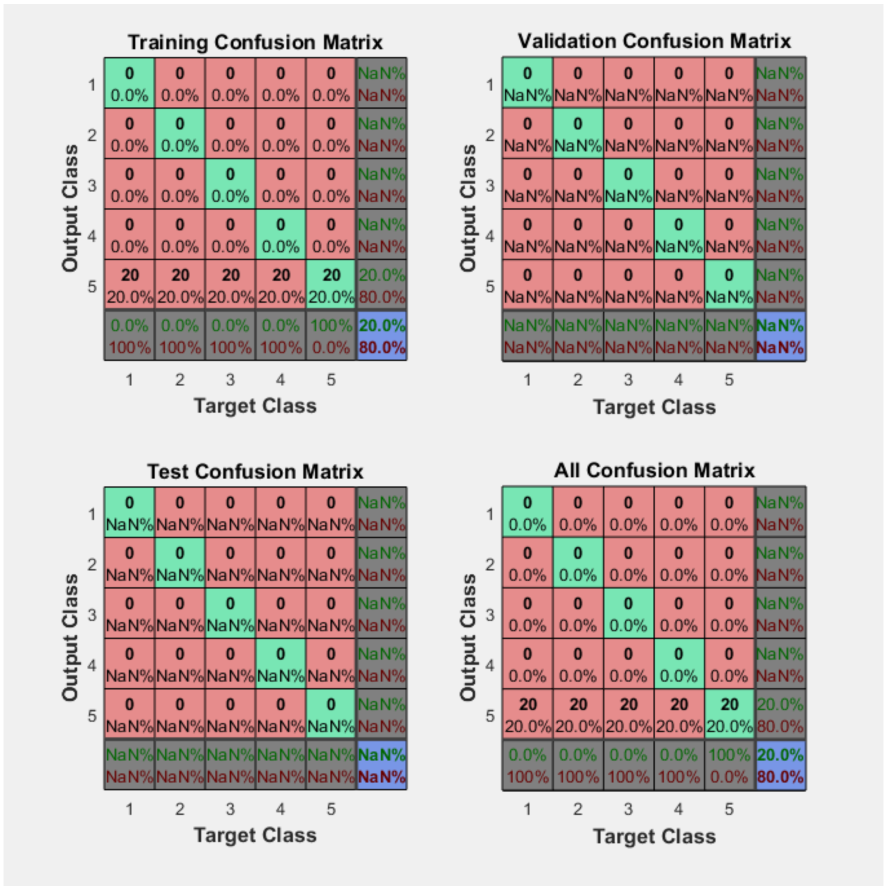


Figure 4.25: Confusion matrix with 8 inputs (2 features) and a linear perceptron network to duplicate the results of the original design for comparison.

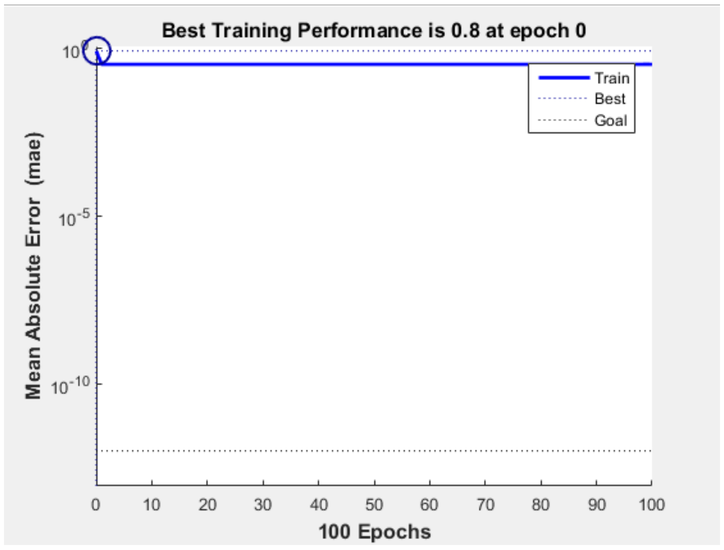


Figure 4.26: Best validation performance with 8 inputs (2 features) and a linear perceptron network to duplicate the results of the original design for comparison.

Upon analyzing the results, it was clear to see that the neural network's adjustments to the original experiment presented in [33] certainly resulted in an improvement in accuracy as tripling the number of features for inputs increased the overall accuracy from 44% to 77% as shown in the confusion matrices shown in Figures 4.21 and 4.23. The tripling of features also caused the best validation performance to change from 0.40888 to 0.14346 (mean squared error, MSE) as shown by comparing Figures 4.22 and 4.24. This demonstrates that the increase in the number of features allowed the neural network to have more clues to classify one signal from another, thus increasing the accuracy dramatically. It is also important to note that the attempted duplication of the results of the original experiment by using the linear classifier (perceptron) resulted in a poor overall accuracy of only 20% and validation performance of 0.8 as shown in Figures 4.25 and 4.26 as a result of the linear classifier being unable to classify inputs that have non-linearly separable relationships with their respective outputs. This leaves the question of why wasn't the result around 70 to 80% accuracy like in the original experiment and why was the modified version (although a significant improvement from 20% accuracy) only at 77% overall accuracy? The answer is most likely due to the fact that the equipment and hardware used to record and extract the subvocal EMG signals for this project is much lower in quality compared to the equipment and hardware used in the original experiment. Although a similar design, the filter used to extract and filter the subvocal EMG signals for the project is only a breadboard prototype design and, due to a lack of funds, does not contain the hardware to create more advanced filters used such as the Infinite Impulse Response band-pass filter to filter out more types of noise and thus is likely to have much more noisy data than that of the original experiment, thus resulting in data that is much harder to classify and therefore had much more error. It also showed that the neural network was able to find differences in each of the signals of differing vowels despite no sign of difference between signals outputted from the amplifier/filter (monitored using oscilloscope) with an accuracy of 77%, showing that a subvocal signal is possibly present but not appearing on oscilloscope captures by either being too small in amplitude or drowned in noise. In conclusion, this showed that the amplifier/filter needed improvements in terms of dealing with noise and possibly needed more gain, but with the amplifier/filter now proven that it can provide some sort of possible subvocal signal with a 0.750 V DC voltage, it was time to test it with the microcontroller. Neural network code is shown in appendix A as Figure A.5 (wavelet transform code removed due to the large size of the code and because it was not included in the final design of the project).

4.4 Microcontroller Code Implementation V2.0 (Input Sampling)

After proving that the amplifier/filter operated with a bias voltage on the output, the microcontroller could process the code for the wavelet transform, and that the interrupts for the buttons could be activated, it was time to test the performance of the wavelet transform on output signals from the amplifier/filter. Since initial testing with the full program showed no output when receiving an input, testing whether or not the microcontroller can receive an input and output the result was done by isolating the part of the program that receives the input (button triggered interrupt routine) and the part that gives the output (DRIVE_DAC function) and then placing them in their own program in order to see if these parts perform their functions on their own. The microcontroller received the usual 0.3 V sine wave input at 300 Hz, while the output pin (P1.6, SIMO, slave in master out, output pin that outputs the SPI data) was received by the oscilloscope to see if the microcontroller could receive and output the sinewave input. Results showed that the microcontroller was not outputting anything out of the SPI output. After checking over the MSP432 family user's guide manual [32], it was apparent that the output pins being used to send the SPI data and the clock to the external DAC (not added yet) were not set to their "primary functions" (SIMO and SPI clock) and instead were being treated as port inputs and outputs. This was solved by adding the instruction P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK which sets the pin P1.6 to its SIMO function and pin P1.5 to its SPI clock function and removing the instruction P1DIR |= BIT6; //set P1.6 output which was setting the P1.6 pin to be a just a regular port output instead of a SPI data output. In addition, the output pin was changed from P1.4 to P6.4 as P1.4 could not be found on the microcontroller board as a breadboard compatible pin. Next, the external DAC was added to the microcontroller. The DAC chosen was the mcp4921 DAC as it was the cheapest DAC with an SPI compatible input and 12-bit resolution available at the time (a 14-bit external DAC would be added later for larger resolution). Unfortunately even after applying the external DAC, no output was measured from the SPI data output or the output of the DAC.

[Note: TIMERA based interrupt not used in final design, skip to next paragraph for ADC14 based interrupt] Upon further examining the value of the "decominput" array after ADC14MEM15 received its input (ADC14MEM15 takes a sample from the ADC14 and puts it in the "decominput" array), it was noticed that the “decominput” array was not receiving any inputs at all, showing that the way the microcontroller handled its input needed to be changed. First, it was noticed that a mistake was made regarding how the TIMERA timer was implemented to delay the time between each input sample being taken. It was assumed that by adding 1300 CCR0 cycles to the TIMERA timer between each time a sample was taken, it would act as 1300 delay cycles that would delay the time between each sample by 1300 cycles, but this was misconception concerning how the TIMERA timer worked. Adding CCR0 cycles to the TIMERA timer only increases the times it takes to call another TIMERA interrupt and since the TIMERA interrupt was not even being used, increasing the CCR0 cycles was pointless. To fix this issue, the instruction to increase the number of CCR0 cycles (TA0CCR0 += 1300;) was replaced with a “delay cycle” function for 1300 cycles. Even after experimenting with the number of cycles, no input was being received. Considering these results and how unreliable the button interrupts had previously shown themselves to be, it became clear that the method of using the button interrupts to call the microcontroller to take inputs needed to be changed. Instead of using a button interrupt routine (INT_PORT4_Handler) to take input samples, a TIMERA (TA0_N_ISR_HANDLER) interrupt routine was used instead. With the new TIMERA interrupt routine, an input sample would be taken every time 1300 CCR0 clock cycles passed as opposed to taking 16 samples every time a button was pressed. In order to allow the user to still be able to control when microcontroller takes data, it was planned that a new button interrupt would be added later that would simply stop the microcontroller when pressed. After testing the new program and even experimenting with the number of CCR0 cycles, the new interrupt routine was not being called. This was solved by toggling the "TAIE" bit in the TA0CTL register which enabled the TIMERA interrupt (only the CCR0 interrupt bit needed to be toggled when using the MSP430 for the TIMERA interrupt to trigger, but the MSP432 apparently requires both to be toggled). The TIMERA interrupt now was working, but no values were appearing in the "decominput" register despite the change in the interrupt routine (no inputs taken by the ADC14). In addition to no input being received, other problems included the microcontroller not sending the UCB0CLK to the external DAC. Since both the UCB0CLK and the TIMERA timer were using the SMCLK (sub-main clock), it was assumed that these problems were a result of the SMCLK not operating properly (perhaps the number of CCR0 cycles were not being counted properly?). To fix this, the CSSTAT register values SMCLK_ON and DCO_ON were toggled as an initialization instruction (Most MSP432 code examples observed involving the use the SMCLK did not toggle these bits and still managed to work, and thus, the bits were originally not toggled as they were thought to not be necessary), which toggle the SMCLK and DCO clocks to be in the "active" state (the SMCLK uses the DCO clock so the DCO clock needed to be toggled as well). In addition, other improvements were added such as the instruction TA0R = 0; which sets the TIMERA timer CCR0 cycle counter to zero when first initialized (this is generally assumed to be default, but was added just to make sure). The initialization instruction for the ADC14, ADC14CTL1 = ADC14RES_3, was also added to set the conversion resolution of the ADC14 to 14-bit as opposed to the default 8-bit to give a larger range of input values in terms of precision, the UCB0 register names were updated from MSP430 register names to MSP432 register names (UCB0CTLW0, this made no difference in the end, but was added just to make sure it was not causing a problem), toggling the bits DCOEN in the CSCTL0 register (enables the DCO clock) and SELS_3 in the CSCTL1 register (selects the DCO clock as the source for the SMCLK to ensure that the SMCLK has a clock to source from), and the instruction TA0CCTL0 &= ~CAP; was added to ensure that the TIMERA timer stayed in its default "compare" mode. It was also noticed that a mistake was made regarding the purpose of the "V_{REFA}" pin on the mcp4921 external DAC as it was assumed that it was for the reference voltage of the output voltage and thus was tied to ground. Upon looking at the mcp4921 datasheet [39], it was noticed that this assumption was a mistake as this pin was actually the reference voltage input for the DAC (why there were two separate pins for powering both the entire chip and the only DAC on the chip was probably for manufacturing convenience) and thus was tied to the 5 Volt rail instead. Unfortunately, despite these improvements and experimenting more with the number of CCR0 cycles and Drive_DAC delay cycles (delay cycles that ensure that the microcontroller has enough time to send out enough UCB0CLK cycles to give an output), no input was being received from the ADC14 and no output was coming out of the external DAC. After several more experiments, progress was finally achieved by simply changing the input pin from A15 (P6.0) to A1 (P5.4) (Changing ADC14MEM15 to ADC14MEM0, a mistake discovered later that should have been ADC14MEM1 for A1, and ADC14SHT1_2 to ADCSHT0_2, appropriately), which resulted in an input to finally be obtained. Unfortunately, despite giving the microcontroller the 0.3 V, 300 Hz sinusoidal test signal as an input, each input value obtained from the ADC14 remained a constant value (1938) even after experimenting with the number of CCR0 cycles. The program at this point is shown in appendix A as Figure A.6.

Due to the issues involving the input, it was decided to change the interrupt routine for obtaining the inputs once again, this time taking influence from the msp432 ADC code example given in the files that come with Code Composer Studio 6.1.1. Like the ADC code example, it was decided to have the interrupt routine that determined how the input was taken would be triggered by the ADC14 interrupt instead of the TIMERA interrupt. This would allow the input to be taken whenever the ADC14 would be available to take an input instead of relying on a timer, but would also make it harder to control the exact sample rate at which the inputs would be taken. Like the example code, the sample conversion timing was put into “pulse sample mode” (ADC14SHP = 1, bit toggled) as opposed to “extended sample mode”, allowing the ADC14SHT0_2 setting to control the length of the sample period as opposed to the SHI signal (this was a mistake to not include this before as it would have made the number of CCR0 cycles calculation more accurate) and a number of delay cycles were put between each time the ADC14 would be enabled to take an input to help control the input sample rate. After implementing these changes, the ADC14 started to accurately take inputs (no longer just a constant value) based on comparing the inputs taken over time to the 0.3 V, 300 Hz sinusoidal signal used as the input (Despite the mistake in choosing ADC14MEM0 instead of ADC14MEM1, inputs taken were accurate until later for some reason). When using both the test input signal and the amplifier/filter to input a waveform, the external DAC seemed to be outputting the input waveform as well, but at a much smaller amplitude and included a noise waveform at 60 Hz (changing the gain value in the “Drive_DAC” function did not increase the value of the output). Program progress at this point shown in appendix A as Figure A.7.

To remove the 60 Hz noise, the 2.5 V rail was finally added to the notch filter and added to the rest of the amplifier/filter, resulting in the output of the amplifier/filter to have a DC voltage of 2.0 V (an improvement from the 0.750 V DC voltage achieved earlier). It was also noticed that microcontroller was taking inputs very slowly, which was fixed by commenting out the "__sleep( );" and "__no_operation;" instructions and adjusting the number of delay cycles in the delay loop for enabling the ADC14 to take samples. In addition, the initialization instructions P5SEL1 |= BIT4; and P5SEL0 |= BIT4; were added, which configure the P5.4 input pin for the ADC14 to be set as the A1 input port for the ADC14 as opposed to just a generic input port. It was a mistake to not include this before, but, strangely, it did not make any difference as the ADC14 seemed to be able to operate without it.

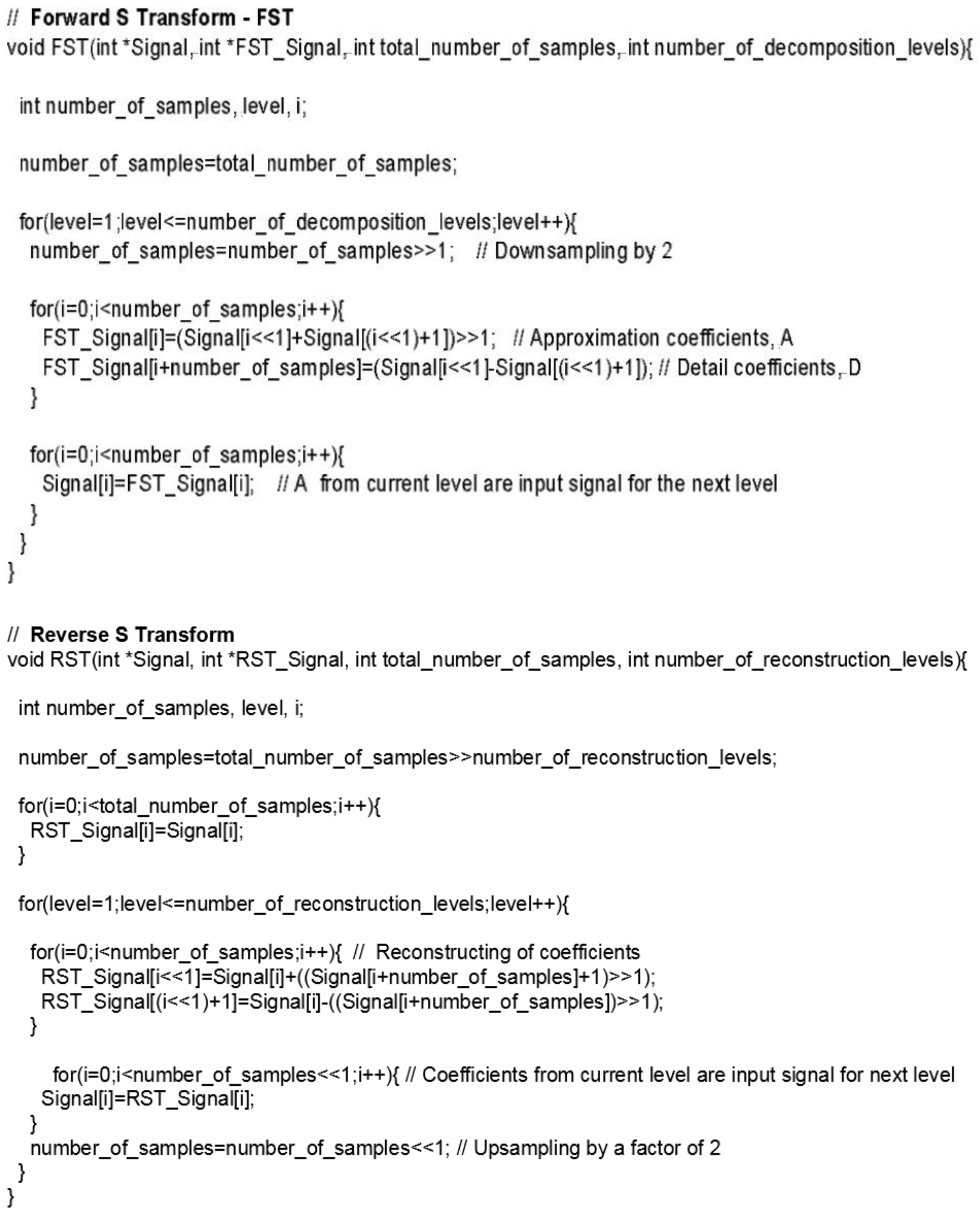
4.5 Microcontroller Wavelet Transform and Reconstruction Code

Since the microcontroller was now successfully taking inputs and seemed (at the time, this was later discovered not to be the case) to be reproducing the input as the output on the external DAC and thus proving that the microcontroller could take inputs and deliver outputs, it was decided that the wavelet transform (decomposition function) and inverse wavelet transform (reconstruction function) needed to be added to the program. Even though the microcontroller was able to process the decomposition function and reconstruction function before, more problems occurred as the input to both functions was no longer zero. When attempting to process the wavelet transform code, the microcontroller would enter an unexpected interrupt (fault interrupt), meaning that the microcontroller would fail to process the code. In addition, upon checking the values of the lookup tables (example: array “evenintable”) using “CCS debug” mode in code composer studio after they had been processed, the values in each register were not matching what they were declared to be, some even being incomplete (zeros where values should be) or having incorrect values. When checking the amount of memory used by the microcontroller for this program, only 3% of the main memory and 4% of the “SRAM” memory was used. Due to technical problems, the CPU load caused by the program could not be recorded, but measuring the runtime of the program showed that Code Composer Studio was failing to show runtimes for the wavelet transform and reconstruction functions despite the fact that the program was clearly processing both functions (pointer was passing through and reaching the “Drive_DAC function), showing that these functions were most likely overwhelming the CPU load since memory was shown to be not the issue. After testing different parts of the wavelet transform code and still having these errors, it was concluded that the decomposition and reconstruction functions involved too much processing power for the microcontroller and needed to be simplified in order to work with the microcontroller.

Simplifying these functions involved considering what lattice coefficients were being used in comparison to the rest of the filter and using said knowledge to remove any unnecessary components of the wavelet transform code. For example, it was noted that the wavelet transform being used was designed to be a generic lattice filter that could use any kind of 6th order lattice coefficients, but, since the simplest of 6th order coefficients were being used (Haar coefficients, $c_0 = 1.0$, $c_1 = 1.0$, $c_{2,3,4,5} = 0$, $\beta = 1/\sqrt{2}$ for their simplicity and speed), it was noticed by examining the lattice filter structure shown in Figure 3.4 that the four coefficients γ_1 and γ_2 (two γ_1’s and two γ_2’s) were zero ($c_{2,3,4,5}$ =

0). With these four coefficients being zero, the second and third rungs of the lattice filter no longer contribute any filtering to the inputs as they pass through the lattice filter (the second and third rungs now only multiply the output of the first rung by one), thus also rendering the delay cycles (z^{-1}) pointless (no point in delaying the output of the first rung if no further significant arithmetic is done). Thus, since the delay cycles were then no longer needed, the lookup tables were no longer needed as well as only a simple shift register was then needed to pass the inputs through the lattice filter. Finally, since only one rung of the lattice filter was being used, it was much simpler to express the high-pass and low-pass filters as simple equations, high-pass as $(1/\sqrt{2})[x(2n) + x(2n + 1)]$ and low-pass as $(1/\sqrt{2})[x(2n) – x(2n + 1)]$, where "n" is the sample number and "x" is a input. Coincidentally, the wavelet transform code referenced from the paper "MSP430 IMPLEMENTATION OF WAVELET TRANSFORM FOR PURPOSES OF PHYSIOLOGICAL SIGNALS PROCESSING" [31] (the system being improved), used this exact method with a few exceptions: the low-pass and high-pass equations were switched, which is the result obtained if one considers the even numbered input odd and vice versa, and, while the "beta" coefficient was decided to be spread equally among the low-pass and high-pass results for the original method, the MSP430 implementation method gave the low-pass filter a "beta" coefficient equal to one-half and the high-pass filter a "beta" coefficient equal to one, most likely because one cannot use the multiplication function with the MSP430 so the shift operator was used instead, which divides or multiplies a binary number by two. With these equations, the lookup table was no longer needed and much less memory would be used, but with the limitation of no longer being able to choose different lattice coefficients other than the Haar coefficients.

However, since using other coefficients involve including lookup tables which have shown to cause memory issues with the microcontroller, it was clear that the Haar coefficients were the best choice for the microcontroller so this limitation was not going to matter. It was decided to use the equations from the MSP430 implementation document (low-pass as (1/2)[x(2n) + x(2n + 1)] and high-pass as (1/2)[x(2n) – x(2n + 1)], code can be referenced in [31] and is shown below in Figure 4.27) as opposed to the calculated equations because the document had been proven that they give accurate results, while the calculated equations were used later for testing results.



```
//  Forward S Transform - FST
void FST(int *Signal, int *FST_Signal, int total_number_of_samples, int number_of_decomposition_levels){

  int number_of_samples, level, i;

  number_of_samples=total_number_of_samples;

  for(level=1;level<=number_of_decomposition_levels;level++){
   number_of_samples=number_of_samples>>1;    // Downsampling by 2

   for(i=0;i<number_of_samples;i++){
     FST_Signal[i]=(Signal[i<<1]+Signal[(i<<1)+1])>>1;   // Approximation coefficients, A
     FST_Signal[i+number_of_samples]=(Signal[i<<1]-Signal[(i<<1)+1]); // Detail coefficients, D
   }

   for(i=0;i<number_of_samples;i++){
     Signal[i]=FST_Signal[i];    // A  from current level are input signal for the next level
   }
  }
}

//  Reverse S Transform
void RST(int *Signal, int *RST_Signal, int total_number_of_samples, int number_of_reconstruction_levels){

  int number_of_samples, level, i;

  number_of_samples=total_number_of_samples>>number_of_reconstruction_levels;

  for(i=0;i<total_number_of_samples;i++){
    RST_Signal[i]=Signal[i];
  }

  for(level=1;level<=number_of_reconstruction_levels;level++){

   for(i=0;i<number_of_samples;i++){  //  Reconstructing of coefficients
     RST_Signal[i<<1]=Signal[i]+((Signal[i+number_of_samples]+1)>>1);
     RST_Signal[(i<<1)+1]=Signal[i]-((Signal[i+number_of_samples])>>1);
   }

       for(i=0;i<number_of_samples<<1;i++){ // Coefficients from current level are input signal for next level
     Signal[i]=RST_Signal[i];
   }
   number_of_samples=number_of_samples<<1; // Upsampling by a factor of 2
  }
}
```

Figure 4.27: Decomposition (forward S transform) and reconstruction function (reverse S transform) code referenced from [31].

Now that the decomposition and reconstruction functions had been simplified, it was time to implement them into the program. First, the decomposition function (forward S transform function) was added to the program and operated easily without errors. However, it was important to note that both the decomposition and reconstruction functions used “int” (integer, no decimal places) type values for their variables when the inputs and outputs are intended to be “float” type (able to have 6 decimal places). Using the ability of the MSP432 to have multiplication arithmetic (MSP430 does not have), the shift operators (<< and >>) which shift bits left or right (essentially dividing or multiplying the value by two for a binary number system) used halve the output values of filters for both functions were replaced by “*(1.00/2.00)”. By using the multiplication arithmetic and adding decimal points to the numbers multiplying the filter outputs, this allowed the calculation that creates the filter outputs to no longer have to round the filter outputs to the nearest integer by giving the output decimal points, thus allowing the variables of the wavelet transform functions to be float values (the shift operators were kept for the counter variables because they have no need to be float values). This change from integer to float values allowed the wavelet transform functions to not only receive values of greater precision, but also output values of greater precision as well. Upon testing the decomposition function with the rest of the program with the usual 0.3 V, 300 Hz sinusoidal input waveform, the output registers of the decomposition function showed that the function was correctly processing the input signal, showing that the decomposition function was operating correctly. The reconstruction function was then added with the decomposition function sending its output to the reconstruction function’s input and tested using the same input waveform in which the results (reading output registers of the reconstruction function and comparing them to the input waveform) showed that the reconstruction function was successfully reconstructing the output of the decomposition function with only slight differences. These differences were expected as the process of wavelet transforming and then reconstructing the signal would remove high frequency noise, thus causing some differences in the output compared to the input. Finally, an “if” function was added that would allow the user to select whether or not the microcontroller would output from the decomposition function or the reconstruction function (after passing through the decomposition function, of course) depending on the value of the “change” variable.

It should also be noted that the program, now with the simplified decomposition and reconstruction functions restricted to only one type of wavelet coefficient, only used 2% of main memory and 2% of “SRAM” memory, while the new functions only providing about 9.5% each of the total runtime of the program, showing that plenty of memory and processing power remained for possible improvements. These possible improvements included adding the neural network to the microcontroller code and using a more complex wavelet coefficient such as Daubechies-4, which involves only one zero as opposed to two like the Haar wavelet coefficient. However, based on the performance of the lattice filter structure wavelet transform, the microcontroller still clearly would not be able to handle adding the full neural network for each word and phrase of the English language (would probably involve a hidden layer of at least 10,000 neurons which would easily overwhelm the CPU load!). Adding the previously tested neural network would certainly be possible now that there was plenty of memory space and processing power to spare, but this would not be of much use and be very limited with only 5 possible outputs for a device designed to provide data useful for analysis from subvocal signals so it was not included in the microcontroller code. As for the Daubechies-4 wavelet coefficient and for the sake of simplicity and time, it would not be included unless the Haar wavelet coefficient would limit the project from reaching its minimal 70% overall accuracy goal.

4.6 Microcontroller Code Implementation V3.0 (external DAC operation)

Unfortunately, despite the success of the wavelet transform filter functions, the output of the external DAC still seemed to match the input signal when the microcontroller outputted the results of the decomposition function instead of producing a wavelet transformed output. The declaration of the variable "DAC_Word" was moved out of the "Drive_DAC" function in order to be able to check its value more easily when the microcontroller was running in which reading its values showed that the "Drive_DAC" function was successfully sending the microcontroller's SPI data output to the external DAC, thus showing that the problem was occurring with the external DAC. After using the oscilloscope to measure the voltage differences between the rails provided by the power supply and one of the rails of the microcontroller and noticing that the external DAC would still output a constant voltage even when the power supply for the amplifier/filter and DAC was off, it became obvious that the problem was a result of the power supply of the microcontroller (provided via USB from a laptop) and the power supply of the DAC and amplifier/filter conflicting with each other and causing a noisy version of the output of the amplifier/filter (in this case, the test waveform) to appear on the output of external DAC due to the microcontroller sending inputs to a chip (DAC) that was supplied by a difference source voltage. In other words, the voltage difference between both power supplies was appearing on the output of the DAC as a noisy version of the input test signal. The first attempt to solve this problem was done by removing the power supply entirely and having the entire circuit (external DAC and amplifier/filter) to be powered by the microcontroller using its +5 V and ground pins. This failed to work because the microcontroller was not able supply enough current to have the amplifier/filter function properly (it was no longer amplifying or filtering the input) so, instead, the microcontroller was used to only power the external DAC, while the power supply would power only the amplifier/filter. Upon testing the circuit again, the external DAC was no longer constantly outputting the input signal to the microcontroller and noise was removed from the inputs to the external DAC. However, the external DAC now failed to give an output at all. In addition, this change caused the microcontroller to stop taking inputs as well which was fixed by changing the input pin form P5.4 (A1) to P5.5 (A0), thus finally matching the input ADC14 register ADC14MEM0 with its corresponding input pin (A0) (this mistake did not make a difference until now for an unknown reason).

[Note: progress in paragraph barely included in final design] With the power supply and the microcontroller no longer coming into conflict with each other, it was time to solve the issue as to why the external DAC was not outputting anything. It was noticed that the “Drive_DAC” function was outputting 16 blocks of output values from the wavelet transform filter functions without any delay between each output, resulting in the output signals from the external DAC to not accurately correspond to their input frequencies. In order to space out the time between each output, a TIMERA interrupt routine was added that would be called for every number of CCR0 cycles equivalent to the the input sample time which would then send an output to the “Drive_DAC” function. Each of the 16 blocks of outputs would then have the same number of delay cycles between each individual output, thus allowing the microcontroller to output all output samples to the external DAC at the same frequency that the ADC14 takes inputs. Unfortunately, since the TIMERA interrupt routine called its interrupts faster than the decomposition function could finish filtering its inputs, this resulted in the decomposition and reconstructions functions to no longer be able produce outputs. Increasing the number of CCR0 cycles (increasing the time between each interrupt trigger), did not fix this problem so, instead, the TIMERA interrupt routine was removed and replaced with delay cycles between each individual output in the “Drive_DAC” function (the same number of cycles between each input sample taken to make sure the input sample frequency and the output frequency match). After this change, the SPI data output was much more visible and consistent in frequency upon measuring the SPI data output pin with an oscilloscope, but the external DAC was still not giving an output. Other changes in an attempt to fix the problem included changing the V_{REFA} voltage on the external DAC from 5 V to 3.3 V using the + 3.3 V pin on the microcontroller (experimenting with reference voltage), changing the resolution of the ADC14 input from 14-bit to 12-bit (the mcp4921 DAC uses 12-bit resolution), changing the variables of the “Drive_DAC” function from “int” type to “unsigned int” in order to prevent the microcontroller from outputting negative values (the microcontroller has no way of outputting negative values, but this change simply changed all negative values to zero, which was fixed by adding 2000 or 2.0 V to every value to offset negative values to positive values), and changing the “do_some_conversions” function to output “unsigned int” type values to the

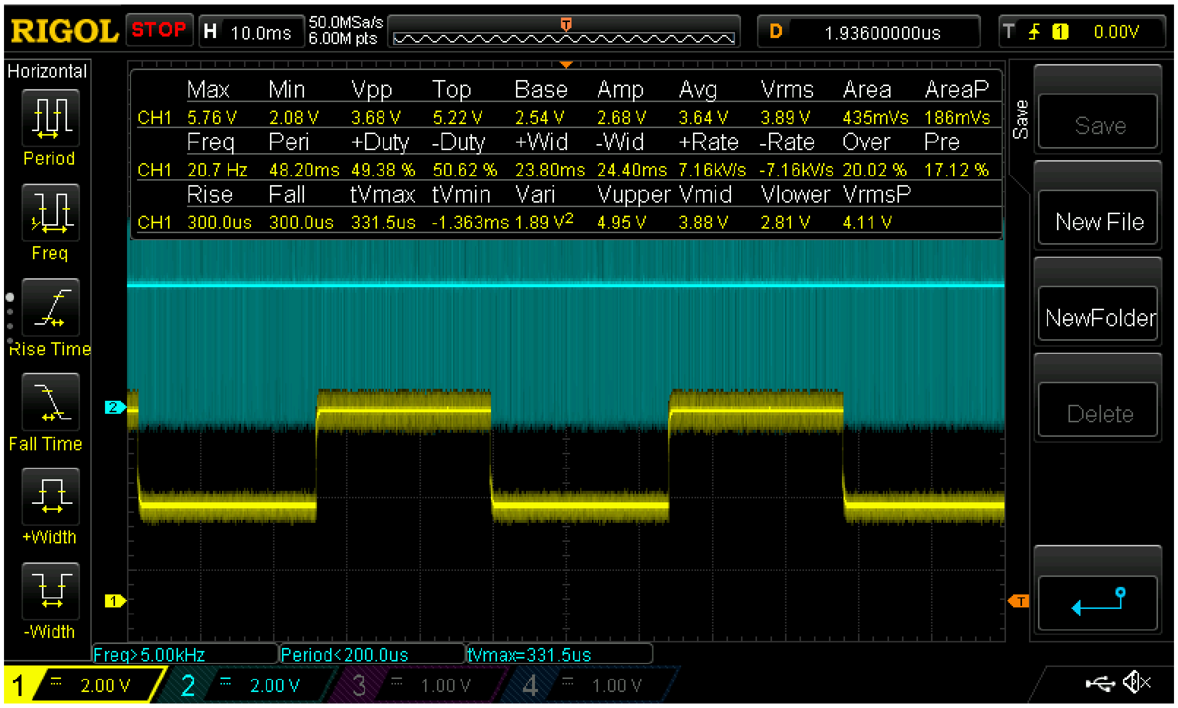
“Drive_DAC” function. Upon checking the UCB0CLK output pin with the oscilloscope, it was evident that no UCB0CLK clock was being sent to the external DAC. Using the register read tool in Code Composer 6.1.1, checking the CSCTL registers (the registers that contain the bits that control the microcontroller clock settings) while the microcontroller was running the program showed that, despite giving initialization instructions for these registers, the bits that were being commanded to be toggled were not being toggled, which would cause the SMCLK clock for the for UCB0CLK to not be enabled. The MSP432 family user’s guide [32] was observed to mention a “security” feature added since the MSP430 which required that the value 0x695A should be written to the CSKEY register in order to access the CSCTL registers in which this instruction was written into the program, resulting in the CSCTL register bits to finally start toggling. Unfortunately, the UCB0CLK was still not appearing. Since the mcp4921’s ideal clock frequency was 20 MHz according to its datasheet [39], it was theorized that by changing the SMCLK to be exactly 20 MHz as opposed to the set 24 MHz frequency, the external DAC might receive the UCB0CLK (SMCLK frequency). Unlike the MSP430, the MSP432 microcontroller was able to use any frequencies between a set of frequencies (1.5 MHz, 3 MHz, 6 MHz, 12 MHz, 24 MHz, and 48 MHz) set by the DCORSEL bits in the CSCTL0 register, but would require calculation of a value according to an equation given in the user’s guide which would be placed in the DCOTUNE bits of the CSCTL0 register with the DCORSEL bit (for example, getting 20 MHz would require selecting the DCORSEL bits that gave the set frequency of 24 MHz and then calculating the DCOTUNE value that would be equivalent to -4 MHz, resulting in 24 MHz – 4 MHz = 20 MHz). The equation, referenced from the application report

"Multi-Frequency Range and Tunable DCO on MSP432xx Microcontrollers” [40], was $N_{DCOTUNE}$ = ($F_{DCO,nom}$ – $F_{RSELx\_CTR,nom}$)x(1 + $K_{DCOCONST}$ x (768 – $FCAL_{CSDCOxRCAL}$))/($F_{DCO,nom}$ x $K_{DCOCONST}$), where $F_{DCO,nom}$ = target nominal frequency, $F_{RSELx\_CTR,nom}$ = calibrated nominal center frequency for DCO frequency range x, $K_{DCOCONST}$ = DCO Constant (floating-point value), $N_{DCOTUNE}$ = DCO Tune value in decimal, and $FCAL_{CSDCOxRCAL}$ = DCO Frequency Calibration value for range x for internal or external resistor modes. In order to find the values for these constants, the “h” files for the MSP432 had to be accessed through Code Composer Studio 6.1.1 while the microcontroller was running (some of these constants were unique to the microcontroller) in which the following values were found: $FCAL_{CSDCOxRCAL}$ = 0x00000184 = 388, “max positive time for DCORSEL_0 to _4” = 0x00000600 = 1536, “max negative time for DCORSEL_0 to _4” = 0x00001600 = 5632, and $K_{DCOCONST}$ = 0x3BA20147 = 0.004944. Putting these into the equation results in [(20 – 24) x (1 + 0.004944 x (768 – 388) x 8]/(20 x 0.004944) = -92.19/98.88E-3 = -931.6245955 ≈ -932 = 0xFFFFFFFFFFFFFC5C. Since the max negative value is -5632 and -932 is larger, this value is within the allowed range. However, when this value was added as the DCOTUNE value, reading the registers showed that it registered as -7260 (various forums on the topic of the accuracy of the DCOTUNE values state that this error is common apparently) and the UCB0CLK still did not appear. Upon checking for additional sources of error, it was noticed that the UCSLA10 and UCMM bits had been toggled in the UCB0CTLW0 register even though they were never intended to be toggled, which causes the “slave” (in this case, the external DAC) to be addressed with a 10-bit address (supposed to be a 16-bit SPI data address) and configures the microcontroller to deal with multiple “masters” (circuits that control other circuits) when there was only one “master” (the microcontroller). At the time, it was assumed that these bits should not be toggled and were causing error so they were set to zero by the instruction UCB0CTLW0 &= ~UCSLA10 + ~UCMM; (it was not released until later that this was a huge mistake). With the UCB0CLK still not appearing, it was decided to change the output pins going to the external DAC with new pins since changing the input ADC14 pin had solved problems before. The pins P1.5 (UCB0CLK), P1.6 (SPI data), and P6.4 (/CS) were changed to the P3.5 (UCB0CLK), P3.6 (SPI data), and P5.0 (/CS), resulting in the same result, SPI data and /CS signal were appearing, but not the UCB0CLK. Reading the CSSTAT register showed that the “SMCLK_ready” bit had been toggled when running the microcontroller, showing that the SMCLK (being using for the UCB0CLK) was ready to use and thus must be working (the TIMERA interrupt routine also used SMCLK and worked fine). Keeping this evidence in mind, it was thought that maybe the oscilloscope just was not picking up the UCB0CLK. By adjusting the vertical voltage scale on the oscilloscope, it could be seen that there was a clock coming out of the UCB0CLK pin where one could not be seen before, but it was very small in voltage amplitude, had a much smaller frequency then expected (only about 29 KHz as opposed to the intended 20 MHz), and appeared as small “spikes” in voltage (not a square waveform of clock cycles). It appeared that the UCB0CLK had been there all along, but was still clearly not operating correctly. Also, the clock source for the ADC14 was changed from the default MODCLK to SMCLK to make sure that the sample and output frequencies of the ADC14 and “Drive_DAC” function were the same and delay cycles were experimented with in the “Drive_DAC” function, still not fixing the UCB0CLK. Current form of code shown in appendix A as Figure A.8.

[Note: progress in paragraph barely included in final design] Considering difficulty solving the problem with the UCB0CLK, it was decided to create another program with the only function of outputting an artificial square wave created in the code. This was created to separate the rest of the code from the code that functions to output to the external DAC in order to isolate any possible mistake in the code that might cause the UCB0CLK to not work properly. With only the “Drive_DAC” function and the “main” function as the program, the UCB0CLK “spiked” much more frequently, but still did not give an output for the external DAC. Experiments were tried such as shorting delay cycles and using DCOTUNE to change the SMCLK frequency from 20 MHz to 16 MHz (example programs had shown to drive external mcp4921 DACs at this frequency), but still no output appeared out of the external DAC. At this point, it was theorized that the perhaps the external DAC was not working properly and a new one was ordered, but this did not change the results. Assuming there might have been a problem with the microcontroller itself or the program running the microcontroller (Code Composer Studio 6.1.1), the microcontroller was factory reset and had its programs reloaded, while Code Composer Studio 6.1.1 was also updated to its most recent version (CMSIS update). After updating the code to fit the CMSIS update standards, the results were still the same. The “Drive_DAC” function code was replaced with a new “Drive_DAC” function code taken from an online example [41] that had been proven to successfully drive an external DAC, but this created no difference in the results either. At this point, also considering that the microcontroller had been giving the error “can't run target cpu: (error -2134 @

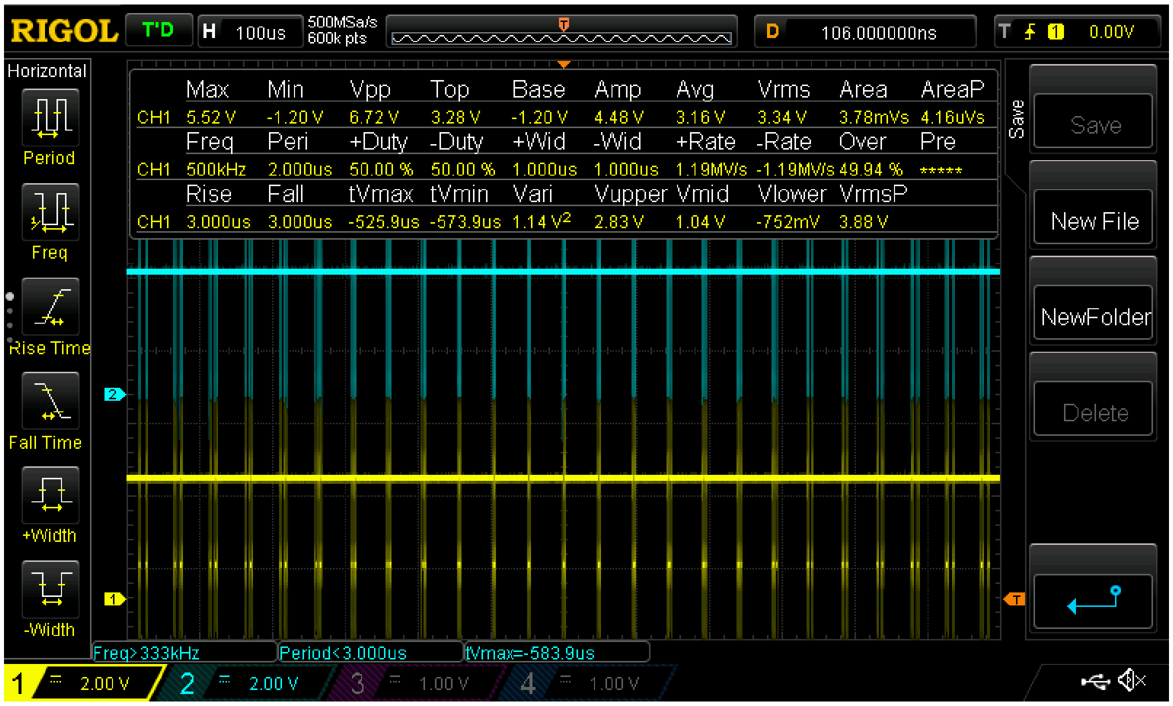
0x0) unable to control device execution state." in its console, it was decided that a new microcontroller should be ordered to solve this problem due to possible damage to the microcontroller (touching the microcontroller while it ran also showed that parts of board were very hot!). Thus, a new MSP432 microcontroller was ordered in which testing showed that the error in the console had been removed (microcontroller was also no longer burning hot to touch), but the external DAC still did not give an output. Next, the LDAC (needs to be set low for the external DAC to give an output) pin, originally left open, on the external DAC was tied to ground to make sure it was not in the high state, but did not change the result. After observing code examples involving use of the SPI included with Code Composer Studio 6.1.1, it was noticed that all the examples (as well as most online as well) were written in a higher level language known as "MSP432Driverlib" for the MSP432 microcontroller. Although it did not seem to make sense as to why a higher level language would be required to operate the MSP432's SPI, rewriting the SPI code with this language was worth a try considering the success rate of the current code so far. After rewriting the code with the "MSP432Driverlib" language, the microcontroller produced no outputs to the external DAC at all as a result of some of the instructions not toggling bits as they were instructed (for example, the instruction to set P3.5 as the UCB2CLK output was not registering). Due to instructions not performing their functions, the higher level language was deemed to be not very reliable and the previous program was once again tested. Attempted square waveform code in "MSP432DriverLib" shown in appendix A as Figure A.9.

At this point, every system involved (microcontroller, microcontroller code, the Code Composer 6.1.1 program running the code, the external DAC, the power source, and the wiring) had been thoroughly checked for error with the exception of the oscilloscope measuring the signals. It was originally assumed that the UCB0CLK (now UCB2CLK because the pins were changed) was not operating properly because each "spike" of the clock appeared too infrequently in order to drive the external DAC and should not have been spikes but square waves instead, but, upon checking over the oscilloscope specifications (SainSmart DSO Note II, DSO202 [42]), the maximum analog frequency bandwidth of the oscilloscope was 1 MHz, while the UCB2CLK was expected to be about 16 MHz, meaning that the oscilloscope would be unable to show the UCB2CLK properly. To solve this problem, the RIGOL DS1054z oscilloscope was used instead for its frequency bandwidth of 50 MHz. Upon analyzing the inputs into the external DAC using the new oscilloscope, the increased bandwidth allowed the oscilloscope to "zoom in" and see the inputs with more detail, showing that the "spikes" that were assumed to be the clock cycles from the UCB2CLK were actually made up of multiple square waveform cycles (each of the two "spikes" for every /CS "spike" turned out to be eight clock cycles, resulting in 16 clock cycles for every /CS cycle, just as intended). This closer inspection also revealed that the /CS input was out of sync with the UCB2CLK (the /CS input was going high in the middle of the clock cycles instead of going high after they passed) and was going high at times when it was supposed to be low. The problem of going high or low at the wrong times was solved by setting the output of the /CS pin to be zero by default when first initialized as the microcontroller was setting the output to be high by default when it was expected to be low. The out of sync problem was solved by inspecting the registers using Code Composer 6.1.1 while the microcontroller was running, which showed that a mistake was made when previously setting the UCSLA10 and UCMM bits to zero in the UCB0CTLW0 register. It was assumed that the bits in the UCB0CTLW0 register that controlled IC2 operation (contained the UCSLA10 and UCMM bits) and SPI operation were separate bits in the register, but were in fact the same bits that simply changed definition depending on what mode of operation was set (in this case, SPI mode) instead. This meant that setting the UCSLA10 and UCMM bits to zero was setting bits needed for SPI operation to zero as well, which is why, on closer inspection, the UCMSB bit was being set to zero even though no statement was given to set it to zero. Thus, after removing the initialization statement UCB0CTLW0 &= ~UCSLA10 + ~UCMM; and using the square waveform code, the DAC finally started giving an output as shown below in Figure 4.28. In addition the /CS input, UCB2CLK, and SPI data are shown below in Figures 4.29, 4.30, and 4.31. The microcontroller code used for creating a square waveform from an external DAC is shown in appendix A as Figure A.10.



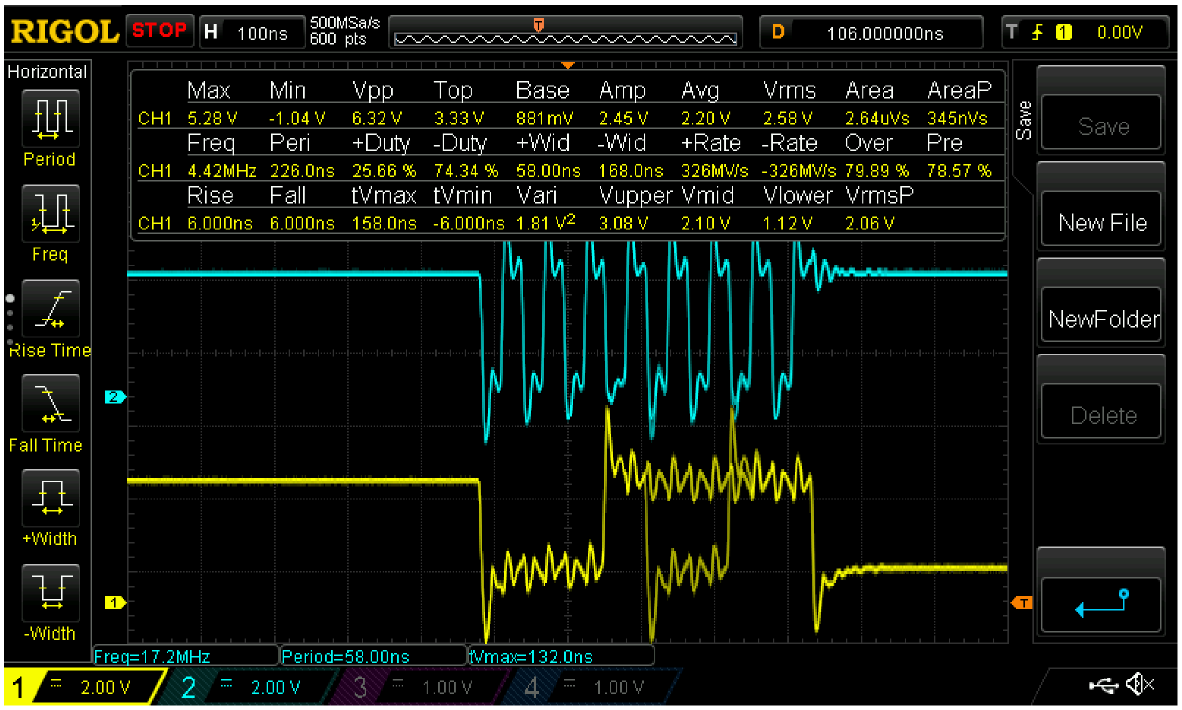
```
RIGOL STOP  H 10.0ms  50.0MSa/s 6.00M pts   D 1.93600000us   T 0.00V

     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms     Area     AreaP
CH1  5.76 V   2.08 V   3.68 V   5.22 V   2.54 V   2.68 V   3.64 V   3.89 V   435mVs   186mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate    Over     Pre
CH1  20.7 Hz  48.20ms  49.38 %  50.62 %  23.80ms  24.40ms  7.16kV/s -7.16kV/s 20.02 % 17.12 %
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower   VrmsP
CH1  300.0us  300.0us  331.5us  -1.363ms 1.89 V²  4.95 V   3.88 V   2.81 V   4.11 V

Freq>5.00kHz   Period<200.0us   tVmax=331.5us
1  2.00 V   2  2.00 V   3  1.00 V   4  1.00 V
```

Figure 4.28: Square wave output from external DAC (yellow) with UCB2CLK (blue) using square wave code.



```
RIGOL T'D  H 100us  500MSa/s 600k pts   D 106.000000ns   T 0.00V

     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms     Area     AreaP
CH1  5.52 V   -1.20 V  6.72 V   3.28 V   -1.20 V  4.48 V   3.16 V   3.34 V   3.78mVs  4.16uVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate    Over     Pre
CH1  500kHz   2.000us  50.00 %  50.00 %  1.000us  1.000us  1.19MV/s -1.19MV/s 49.94 % *****
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower   VrmsP
CH1  3.000us  3.000us  -525.9us -573.9us 1.14 V²  2.83 V   1.04 V   -752mV   3.88 V

Freq>333kHz   Period<3.000us   tVmax=-583.9us
1  2.00 V   2  2.00 V   3  1.00 V   4  1.00 V
```

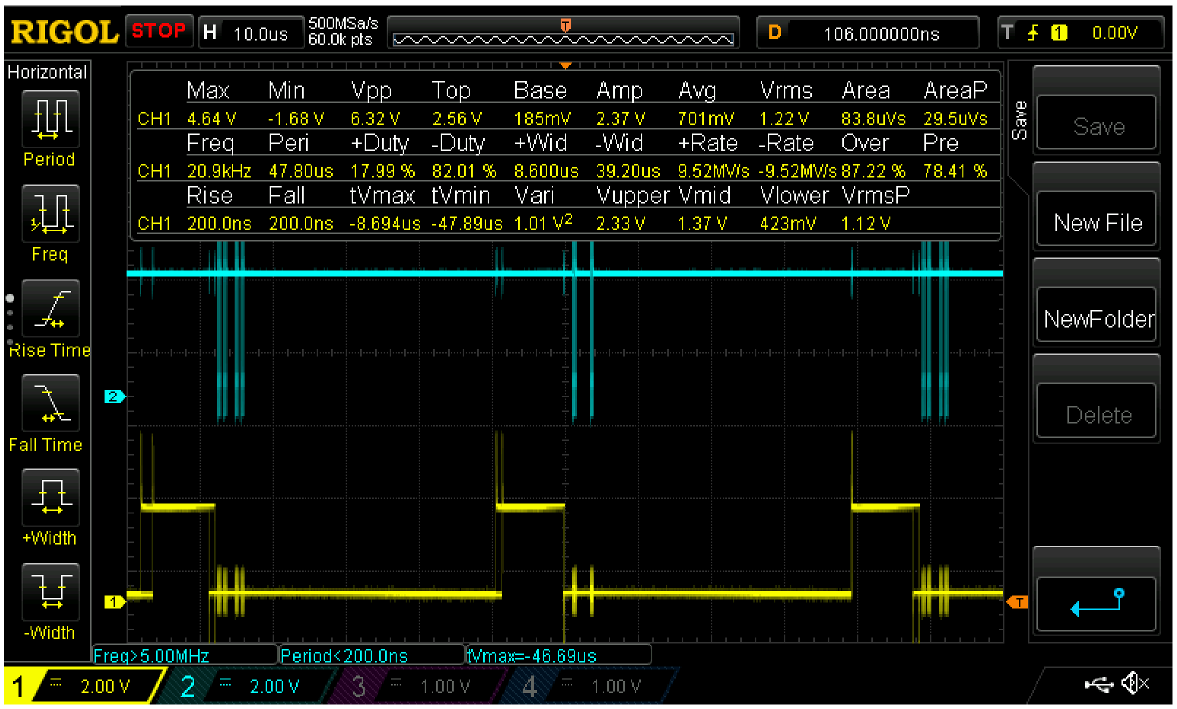
Figure 4.29: UCB2CLK (blue) and SPI data (yellow) shown as “spikes” using square wave code.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	5.28 V	-1.04 V	6.32 V	3.33 V	881mV	2.45 V	2.20 V	2.58 V	2.64uVs	345nVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	4.42MHz	226.0ns	25.66 %	74.34 %	58.00ns	168.0ns	326MV/s	-326MV/s	79.89 %	78.57 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	6.000ns	6.000ns	158.0ns	-6.000ns	1.81 V²	3.08 V	2.10 V	1.12 V	2.06 V	

```
Freq=17.2MHz    Period=58.00ns    tVmax=132.0ns
```

Figure 4.30: UCB2CLK (blue) and SPI data (yellow) “zoomed in” using square wave code.

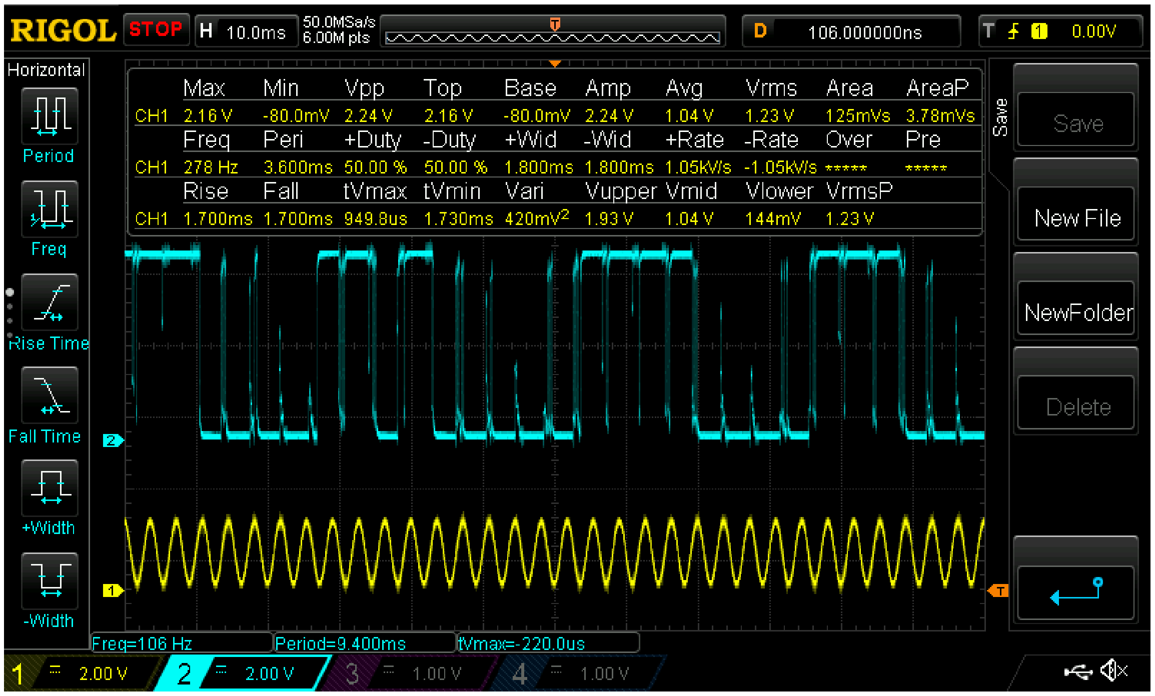


	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.64 V	-1.68 V	6.32 V	2.56 V	185mV	2.37 V	701mV	1.22 V	83.8uVs	29.5uVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	20.9kHz	47.80us	17.99 %	82.01 %	8.600us	39.20us	9.52MV/s	-9.52MV/s	87.22 %	78.41 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	200.0ns	200.0ns	-8.694us	-47.89us	1.01 V²	2.33 V	1.37 V	423mV	1.12 V	

```
Freq>5.00MHz    Period<200.0ns    tVmax=-46.69us
```

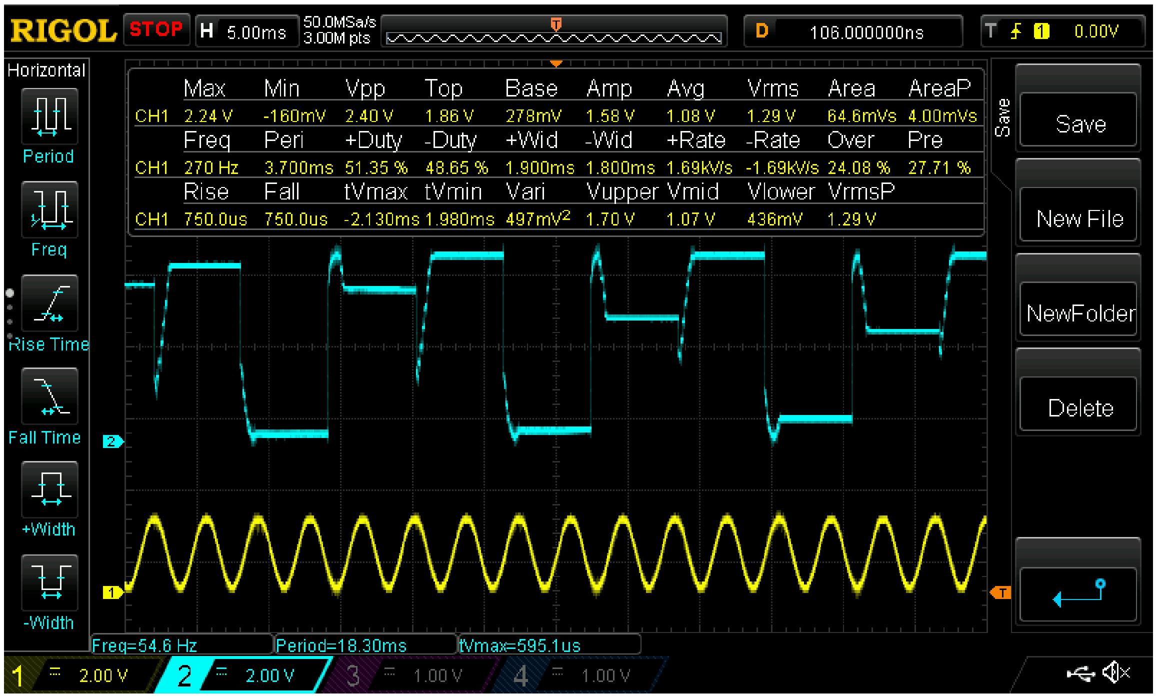
Figure 4.31: UCB2CLK (blue) and /CS input to external DAC (yellow) using square wave code.

Since the microcontroller could now successfully produce an output from the external DAC, the microcontroller was prepped for providing the wavelet transform and reconstruction outputs. After updating the code providing the wavelet transform to the CMSIS update standards (the new CMSIS startup file was not used as it caused the interrupts to fault so the old startup file was used instead) and with the improvements from the square waveform code, the microcontroller was able to provide both a wavelet transformed output and a reconstructed output as shown in Figures 4.32 and 4.33 below. Upon observing the outputs, there were delays between each block of 16 output samples as expected, showing the time it takes to process and output the output signals. The output signals were also larger in amplitude than that of the inputs signals and lowering gain in the “Drive_DAC” function did not seem to lower it (the fact that 2.0 V was added to every value was a contributing factor) which was later discovered to be because the mcp4921 DAC swings its output from ground to its reference voltage as shown in its datasheet [39] so the output would always swing from 0 V to 5 V. The frequency also had doubled from observing the reconstructed output. Increasing the time between samples or increasing the number of input samples taken would, in theory, improve the accuracy of the wavelet transform output, but, since the microcontroller could finally do its intended function, it was time to once again focus on the user interface of the microcontroller instead. The microcontroller code for outputting wavelet transform or reconstructed signal with working external DAC is shown in appendix A as Figure A.11.



```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area     AreaP
CH1  2.16 V   -80.0mV  2.24 V   2.16 V   -80.0mV  2.24 V   1.04 V   1.23 V    125mVs   3.78mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over     Pre
CH1  278 Hz   3.600ms  50.00 %  50.00 %  1.800ms  1.800ms  1.05kV/s -1.05kV/s *****    *****
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  1.700ms  1.700ms  949.8us  1.730ms  420mV²   1.93 V   1.04 V   144mV     1.23 V

Freq=106 Hz   Period=9.400ms   tVmax=-220.0us
```

Figure 4.32: Wavelet transform output (blue) from external DAC and input to microcontroller (yellow).



```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area     AreaP
CH1  2.24 V   -160mV   2.40 V   1.86 V   278mV    1.58 V   1.08 V   1.29 V    64.6mVs  4.00mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over     Pre
CH1  270 Hz   3.700ms  51.35 %  48.65 %  1.900ms  1.800ms  1.69kV/s -1.69kV/s 24.08 %  27.71 %
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  750.0us  750.0us  -2.130ms 1.980ms  497mV²   1.70 V   1.07 V   436mV     1.29 V

Freq=54.6 Hz   Period=18.30ms   tVmax=595.1us
```

Figure 4.33: Reconstruction output (blue) from external DAC and input to microcontroller (yellow).

4.7 User Interface V2.0 and Second Amplifier/Filter Implementation and Performance

The user interface for the microcontroller, which consisted of a button for stopping microcontroller operation and a button for controlling which output the microcontroller gives, was implemented once again by adding the button interrupt routines that previously were having trouble triggering more than once. After updating the code to CMSIS update standards, the button interrupts were triggering more than once (this is most likely due to the fact that the microcontroller is new and no longer damaged), but were still behaving strangely as the P4.0 button interrupt was triggering constantly and had a 2.2 V DC voltage on its pin, while the P2.4 button interrupt had no such voltage, but was still operating fickly (only occasionally triggering sometimes). After switching to a new pin (P4.0 pin to P4.1 pin), changing the power source of the buttons from the power supply to the microcontroller, and other tests with no change in the result, it was decided that the button interrupts routines were too unreliable and, instead, the infinite loop in the "main" function would contain code that would constantly check for inputs from the button pins in order to implement the user interface. Unfortunately, the constant checking for button inputs slowed down the performance of the microcontroller considerably and it was decided that the button interrupt routines once again had to be used. Upon testing the voltage of the pins while modifying the button pin settings in the code, the button P2.4 button interrupt stopped behaving fickly after settings were added to initialize the button pin (they had not been added from the previous button interrupt code, accidentally) and the constant voltage on the P4.1 pin (and now P2.4 pin as well) was removed by removing the initialization of the pull-up resistors for each button input, thus stopping the button interrupt routine from being triggered constantly. However, the button interrupt routines, despite now finally triggering as intended, were still not performing their functions as nothing would happen upon triggering the interrupts. This was solved for the first button that controls when the microcontroller stops operating by changing the infinite loop in the routine to break (move out of infinite loop) only when the button input was zero (interrupt was triggered by rise in voltage on pin in which the processor remains stuck in an infinite loop until the voltage on the pin drops to zero), which caused the routine to, for some reason, give an even better result as the microcontroller would not only stop operating once the button was pressed, but would continue to stay stopped until the button was pressed a second time instead of having to hold the button to stop the microcontroller. For the second button (controls type of output), it was discovered upon using a counter that the button interrupt routine was being called twice consistently every time the button was being pressed, despite the fact that the debouncing circuit was supposed to prevent this, which caused the output type to switch twice and thus cause nothing to happen. To fix this, the counter was then used to only switch the output type when it was an even number to switch the output type only once per button press, allowing both button interrupt routines to finally perform as intended.

Now that the entire system was capable of taking, amplifying, filtering, and wavelet transforming or reconstructing an input, it was time to complete the second circuit that would allow EMG signal inputs to be taken from both sides of the throat as opposed to just one side. Since the second amplifier/filter section was already completed and operated successfully, the second notch filter needed to be added. An exact copy of the previous notch filter (with insignificant differences) was then built and implemented on the breadboard with the rest of the amplifier/filter circuits with a second 2.5 V rail using a voltage follower to provide 2.5 V DC on the output for the second amplifier/filter as well. Despite being an exact copy of the first notch filter, the second notch filter did not work at first as the output only produced a constant 4.0 V DC voltage, but, after many tests, it was determined that something was wrong with the wiring of the notch filter in which the notch filter was moved to a second breadboard to make the wiring more accessible to modify (space was cramped before), allowing the notch filter to work properly. Upon adding it to the second amplifier/filter, the notch filter's output became the same as that of the first amplifier/filter's notch filter when using a 0.3 V, 300 Hz, sinusoidal signal as input, thus showing that it worked properly, but both outputs still contained a lot of noise and were fickle in their operation, sometimes not working until particular wires on the circuit board was moved or pushed back and forth. These problems would be fixed in the future.

[Note: 32 single sample block system referenced below not used in final design] Next, the microcontroller needed to be able to handle both inputs from both amplifier/filters. Originally, it was decided that the microcontroller would take inputs from each amplifier/filter and output them individually through separate external DACs so both signals could be observed individually, but, as shown in Figures 4.32 and 4.33, since there was already large delays between outputs due to the processing time between taking inputs samples, transforming said samples, and then outputting them, it was decided to instead combine inputs from each amplifier/filter into a 32 sample block (two 16 sample blocks from both amplifiers/filters) and use that as the input into the decomposition function, resulting in only one more level of decomposition/reconstruction to be added to the decomposition and reconstruction functions (32 divided by two five times is one, as opposed to 16 divided by two four times being one). Instead of having to run the decomposition and reconstruction functions twice for two 16 blocks of samples, this method of combining two sample blocks into one sample block resulted in less delay between each block of outputs compared to processing both blocks of inputs separately. To implement this, an input pin (P4.7) was initialized for use as a second ADC14 input and the bit ADC14_CTL0_CONSEQ_1 was toggled in the ADC14CTL0 register to set the ADC14 for "sequence of channels" mode, which allows the ADC14 to take inputs from more than one channel (input). After modifying the decomposition and reconstruction functions, the resulting outputs acquired are shown below in Figures 4.34, 4.35, 4.36, and 4.37. Using the function generator, a 2.0 V, 200 Hz, sinusoidal input was applied to each microcontroller input one at a time to test if there would be differences in the output of the microcontroller depending on which input was used. Comparing these to the performance of the single input microcontroller code, the delay between each block of outputs has increased, as expected, but the performance has stayed the same for outputs resulting from inputs into pin P4.7 (double the frequency compared to input) as shown in Figure 4.37. Surprisingly, despite the fact that the input samples from P4.7 included noise while those of input P5.5 did not, the reconstructed output resulting from inputs into P5.5 seemed to be less accurate in terms of capturing the input signal as the output (Figure 4.34) only shows half of the signal. Improvements to the quality of these outputs would be added later.

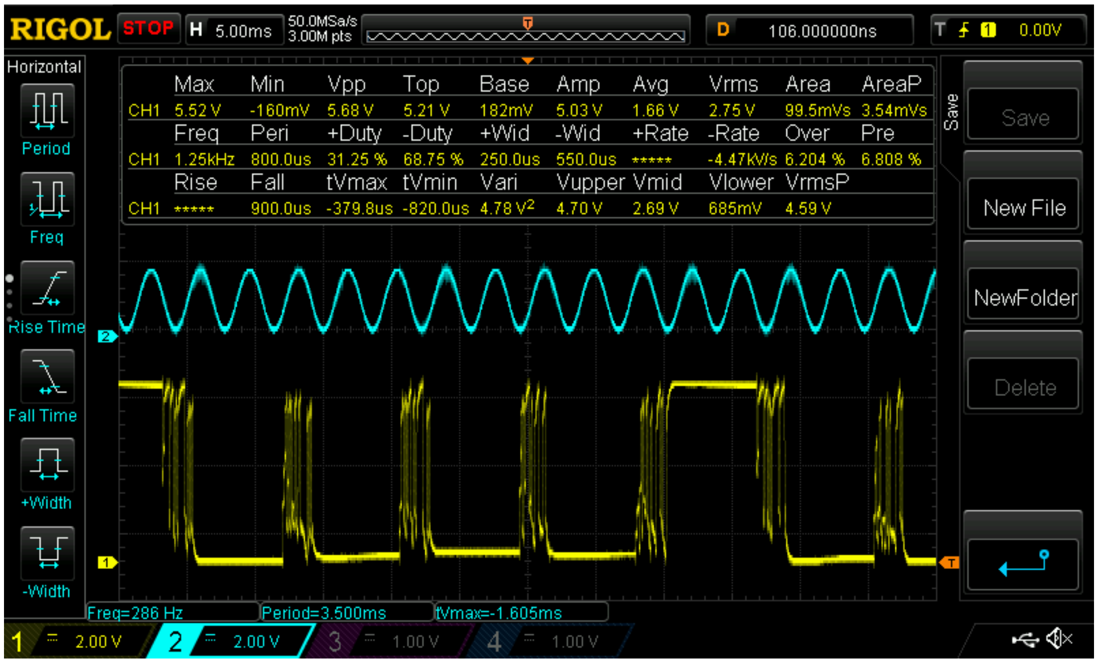

```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area      AreaP
CH1  5.52 V   -160mV   5.68 V   5.21 V   182mV    5.03 V   1.66 V   2.75 V    99.5mVs   3.54mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over      Pre
CH1  1.25kHz  800.0us  31.25 %  68.75 %  250.0us  550.0us  *****    -4.47kV/s 6.204 %   6.808 %
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  *****    900.0us  -379.8us -820.0us 4.78 V²  4.70 V   2.69 V   685mV     4.59 V

Freq=286 Hz    Period=3.500ms    tVmax=-1.605ms
```

Figure 4.34: Wavelet transformed output (yellow) and input (blue) from multiple input microcontroller code with input on pin P5.5.

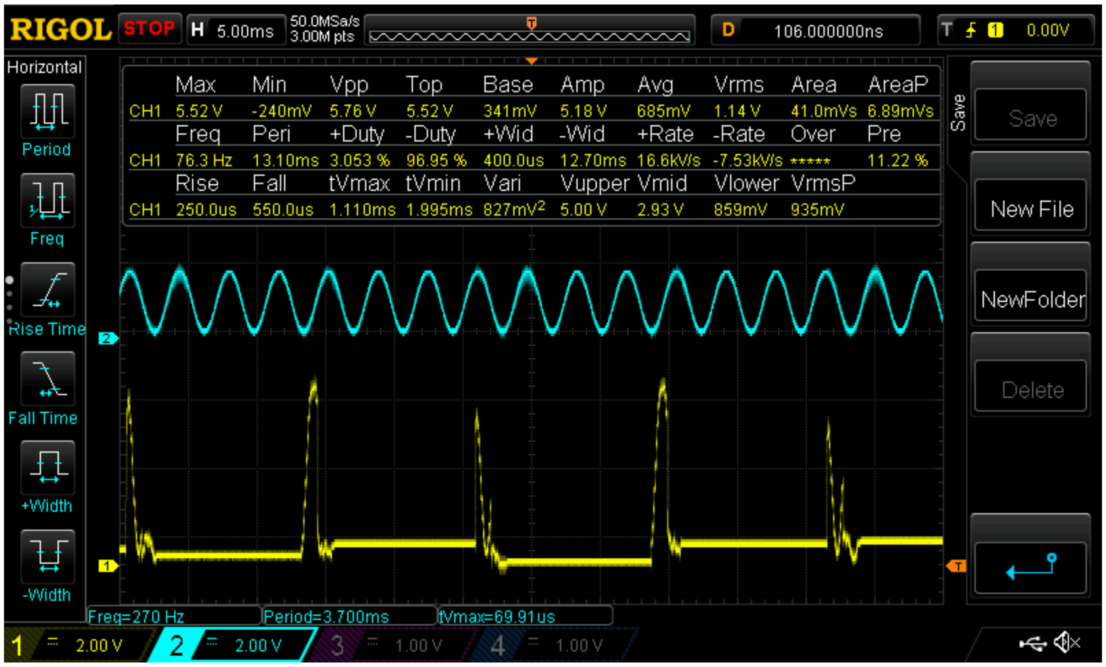
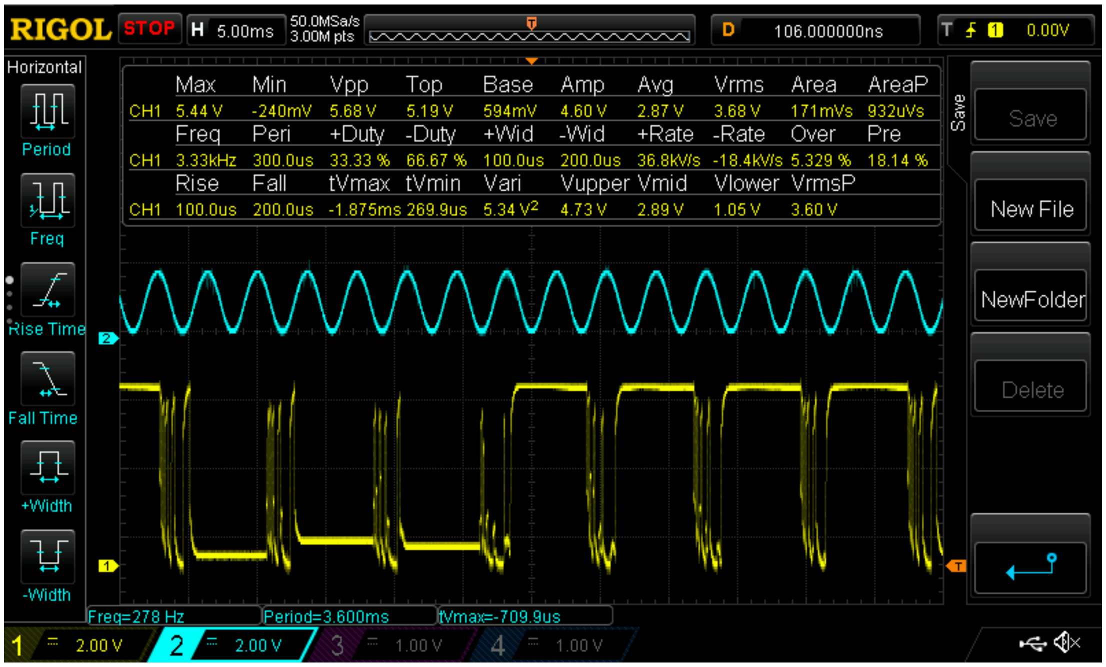
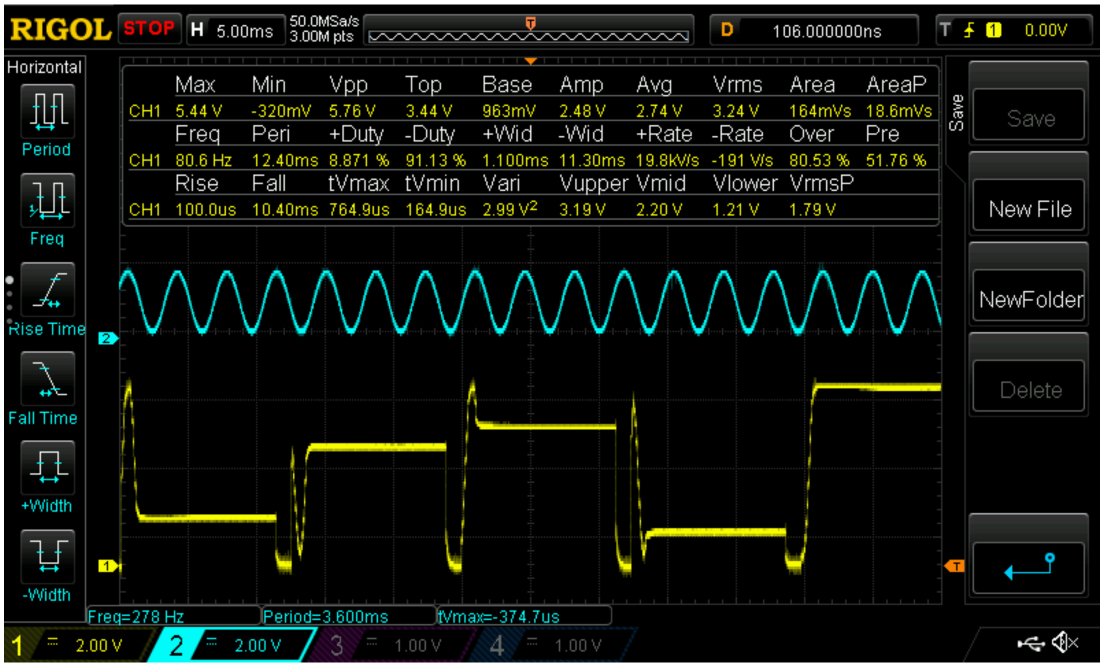

```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area      AreaP
CH1  5.52 V   -240mV   5.76 V   5.52 V   341mV    5.18 V   685mV    1.14 V    41.0mVs   6.89mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over      Pre
CH1  76.3 Hz  13.10ms  3.053 %  96.95 %  400.0us  12.70ms  16.6kV/s -7.53kV/s *****     11.22 %
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  250.0us  550.0us  1.110ms  1.995ms  827mV²   5.00 V   2.93 V   859mV     935mV

Freq=270 Hz    Period=3.700ms    tVmax=69.91us
```

Figure 4.35: Reconstructed output (yellow) and input (blue) from multiple input microcontroller code with input on pin P5.5.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	5.44 V	-240mV	5.68 V	5.19 V	594mV	4.60 V	2.87 V	3.68 V	171mVs	932uVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	3.33kHz	300.0us	33.33 %	66.67 %	100.0us	200.0us	36.8kV/s	-18.4kV/s	5.329 %	18.14 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	100.0us	200.0us	-1.875ms	269.9us	5.34 V^2	4.73 V	2.89 V	1.05 V	3.60 V	

Figure 4.36: Wavelet transformed output (yellow) and input (blue) from multiple input microcontroller code with input on pin P4.7.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	5.44 V	-320mV	5.76 V	3.44 V	963mV	2.48 V	2.74 V	3.24 V	164mVs	18.6mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	80.6 Hz	12.40ms	8.871 %	91.13 %	1.100ms	11.30ms	19.8kV/s	-191 V/s	80.53 %	51.76 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	100.0us	10.40ms	764.9us	164.9us	2.99 V^2	3.19 V	2.20 V	1.21 V	1.79 V	

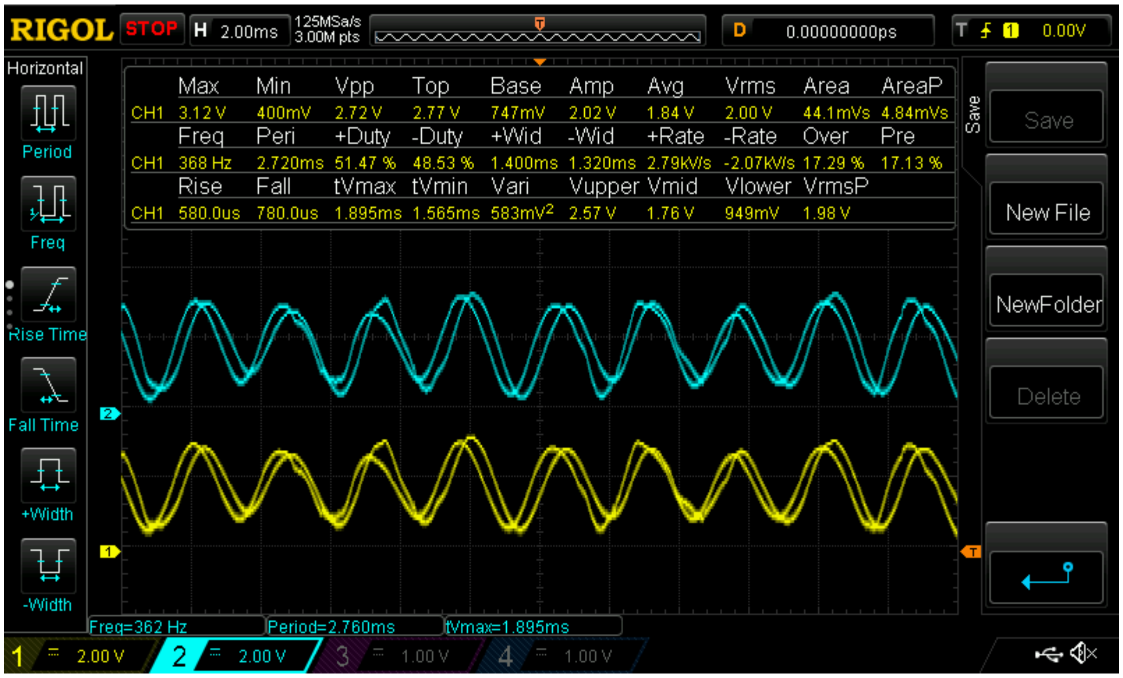
Figure 4.37: Reconstructed output (yellow) and input (blue) from multiple input microcontroller code with input on pin P4.7.

[Note: 14-bit DAC referenced below not used in final design] Now that the microcontroller could take both inputs and output a single output, it was time to take advantage of the MSP432's 14-bit ADC14 resolution (MSP430 only has 12-bit resolution) and upgrade the external DAC from a 12-bit DAC to a 14-bit DAC. The external DAC will then be able to take SPI data in 14-bit resolution and output 14-bit wavelet transformed and reconstructed outputs, while the microcontroller takes inputs in 14-bit resolution, resulting in more accurate outputs. The LTC1658 chip was chosen to be the external 14-bit DAC as it was the only 14-bit SPI compatible DAC that was compatible with a breadboard and could be found within a reasonable price range. Unfortunately, upon replacing the MCP4921 chip with the LTC1658 chip and rewiring it appropriately, the total swing voltage between the lowest voltage and the highest voltage on the outputs of the LTC1658 was only 0.3 V as opposed to the 5.0 V output swing on the MCP4921, which caused signal-to-noise ratio to become much smaller. Changing the gain of the SPI data and the 2000 (2.0 V DC) DC output value to 8000 (14-bit version of 2.0 V DC) did not change the swing voltage (changing the gain did change the voltage values, but not the swing). Upon referencing the LTC1658 datasheet [43], the output voltage of LTC1658 swings according to the equation (GND + V_{OS}) to (V_{CC} – V_{OS}), where V_{CC} is tied to a 5.0 V pin on the microcontroller, GND is ground, and V_{OS} is the voltage offset error, which was the lowest code that guarantees the output will be greater than zero, showing a large voltage offset error was most likely responsible for the small swing. Upon testing with code output values that were very small, but larger than zero (example: one, ten, etc.), the output swing still did not become larger than 0.3 V, while increasing the signal input to the microcontroller did not change this swing voltage either.

Adding a 47 µF capacitor between the 5.0 V pin and the ground pin going to the LTC1658 to AC couple any noise between the supply rail and ground that might be causing voltage offset error and replacing the LTC1658 chip with a new LTC1658 chip in case the original LTC1658 chip was damaged did not change the swing voltage either. At this point, there seemed to be no way to fix the small voltage swing and performance of the LTC1658 would have to be compared to the performance of the MCP4921 later to see how much the small voltage swing inhibits said performance, but, for now, efforts were focused on improving the reliability of the amplifiers/filters and removing their noise.

Removing the noise from the amplifiers/filters proved to be a much more difficult task than anticipated, especially since the gain had been changed from 500 to 5000 after changing the 10 KΩ gain resistors on the INA128 chips to 1KΩ resistors to give a better chance at picking up subvocal signals after the results of previous tests. Upon adding a DC coupling capacitor to the input of the second amplifier/filter while the notch filter was attached, it was noticed that the output of amplifier/filter would gradually increase in noise and voltage amplitude over time even when no input was being sent into it. Upon experimenting with the position of wires and capturing signals from different parts of the circuit using the oscilloscope, it was clear to see that the sudden increases in noise were due to feedback of noise between crossing wires in the circuit as a result of the magnetic fields projecting from wires moving current in wires crossing with them in which the large gain would cause this noise to increase exponentially over time as long as the feedback continued. To fix this, wire lengths were shortened, wires were repositioned to have wires that were close to each stretch across the breadboard parallel to each other as opposed to crossing in order to follow the right-hand rule of magnetism, and the

10000000 pF DC coupling plate capacitors were replaced with 0.05 MF plate capacitors to make sure both circuits were matching (there were only two 10000000 pF plate capacitors available before, but three 0.05 MF plate capacitor were available, allowing for symmetry). Unfortunately, these changes did not solve the problem so it was decided to separate both amplifiers/filters from each other to help assure that the least of amount of wires crossed each other as possible and to help prevent the amplifiers/filters from influencing each other’s outputs with magnetic noise. First, the second notch for the second amplifier/filter was moved to another breadboard to prevent the first breadboard from becoming too crowded with crossing wires close to each other, while the DC coupling capacitors were repositioned, rewired to be facing parallel to one another, and spaced father out from each other. While this did remove some noise, the noise feedback effect was still occurring so it was decided that a possible cause might be the fact that both amplifiers/filters shared the same mc33078p op-amp chip, the close proximity of each op-amp to one another and large gain causing them to infect each other with their own signals or because the op-amp was damaged and needed to be replaced. Thus, more mc33078p op-amps were ordered in which the old op-amp was replaced by a new one, which did not solve the noise feedback problem and showed that the op-amp was not damaged, but it was discovered that the leads going into the breadboard were bouncing out of the breadboard over time, slightly popping out after extended periods of time, resulting in noise and gradual disconnections. To solve this, the op-amp chip was moved to the breadboard with the second notch filter as said breadboard had larger pin holes, thus allowing the mc33078p chip to hold itself more securely into the breadboard. This changed removed a lot of noise on the outputs of the amplifiers/filter and sudden disconnections became much rarer, but the noise feedback effect still occurred. After capturing signals from parts of the circuit using the oscilloscope and conducting more tests, removing the DC coupling capacitors and the voltage dividers used for simulating subvocal amplitude signals removed the noise feedback effect (DC coupling capacitors and voltage dividers would be added back later), showing that the capacitors and voltage dividers were creating an unstable system. Although some noise remained after this change, the output signals of the amplifiers/filters (even without inputs) were much less chaotic in nature as the remaining noise could be discerned as high frequency noise. Since the amplifiers/filters were designed to bandpass between 10 Hz and 450 Hz, the presence of this noise showed that the amplifiers/filters were not appropriately filtering the signals in which the 120 pf capacitors of the mc33078p op-amps were replaced by 470 pf capacitors in order to change the low-pass frequencies from 491.2 Hz to 125.42 Hz (the large gain keeping the bandpass relatively between 10 Hz and 450 Hz) in order to further low-pass high frequency signals. This resulted in the removal of high frequency noise, but 120 Hz noise still remained due to harmonics of 60 Hz noise. To filter out this noise, both amplifiers/filters were given their own mc33078p chips as op-amps as opposed to sharing one and were entirely placed on the breadboard with the second notch filter and the op-amps (circuit was no longer split between two breadboards) with both amplifiers/filter built away from each other in a way that assured that no wires of one amplifier/filter were crossing with that of the other, resulting in a decrease in amplitude of the 120 Hz noise but not completely eliminating it yet. For this new configuration, the first amplifier/filter is now referred to as “right” and the second amplifier/filter now referred to as “left”. After adding the DC coupling capacitors once again, the amplifiers/filters would become more unpredictable and failed to give outputs occasionally, but shortening and repositioning more wires to assure less crossing and shorter paths from chip to chip significantly decreased the number of times the amplifiers/filters suddenly failed to give outputs. Oscilloscope captures of the inputs and outputs of the amplifiers/filters are shown below in the Figures below (note that the function generator was providing a lot of 120 Hz noise in its outputs as well), where Figures 4.39, 4.40, 4.41, and 4.42 refer to the left amplifier/filter; Figures 4.43, 4.44, and 4.45 refer to the right amplifier/filter; and Figures 4.38, 4.46, and 4.47 occurred when an input was being applied to both at the same time and show filtering take place as increasing the frequency decreases the signals voltage amplitude, revealing the remaining 120 Hz noise. Bode plots of the magnitude gain of the amplifiers/filters are shown in Figures 4.48 and 4.49, showing that the amplifiers/filters peak in magnitude between 100 Hz and 500 Hz (3 dB bandwidth between 100 Hz and 500 Hz). Input signals were applied with a voltage divider on the input, effectively multiplying each input signal by 0.010 before entering the circuit to help better simulate an actual subvocal signal in amplitude. Data was unable to be taken showing the input signal while DC coupling capacitors were attached as the noise feedback effect would occur if an oscilloscope probe was attached to the input of the amplifiers/filters while the DC coupling capacitors were attached, showing that some improvements would have to be added in the future in order to prevent this occurrence.



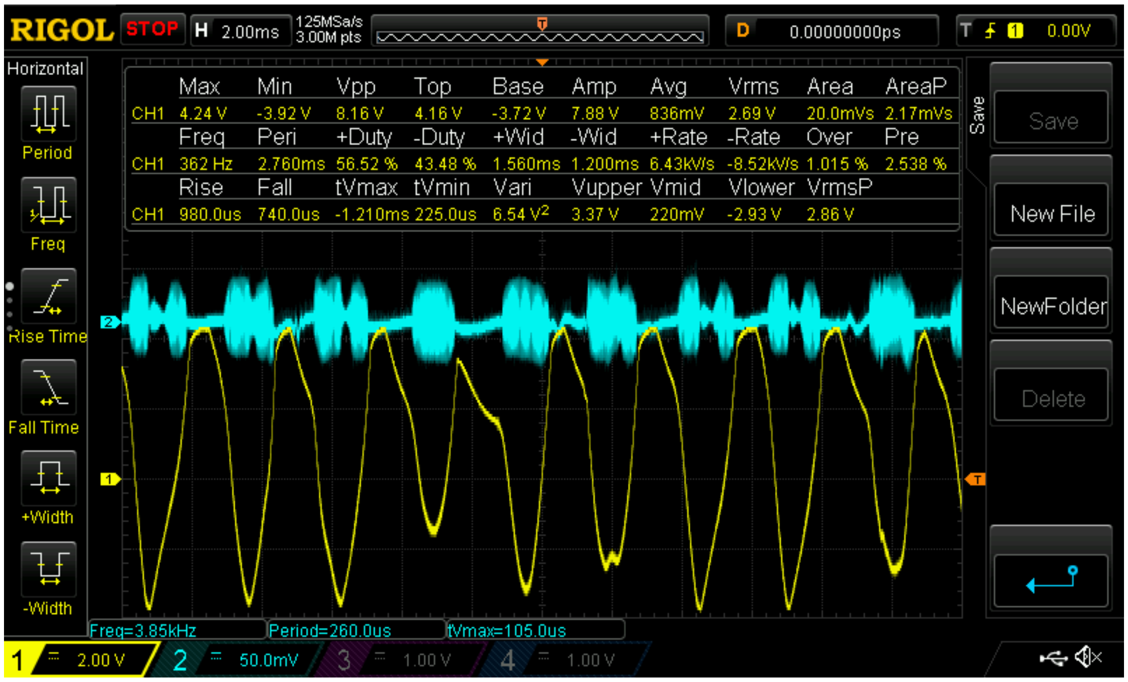
RIGOL STOP H 2.00ms 125MSa/s 3.00M pts D 0.00000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	3.12 V	400mV	2.72 V	2.77 V	747mV	2.02 V	1.84 V	2.00 V	44.1mVs	4.84mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	368 Hz	2.720ms	51.47 %	48.53 %	1.400ms	1.320ms	2.79kV/s	-2.07kV/s	17.29 %	17.13 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	580.0us	780.0us	1.895ms	1.565ms	583mV^2	2.57 V	1.76 V	949mV	1.98 V	

Freq=362 Hz Period=2.760ms tVmax=1.895ms

1 2.00 V 2 2.00 V 3 1.00 V 4 1.00 V

Figure 4.38: Outputs of left (yellow) and right (blue) amplifiers/filters with 600E-6 V (before going through 0.010 voltage divider), 300 Hz, sinusoidal input signal (there was a second signal coming from the function generator for some reason).



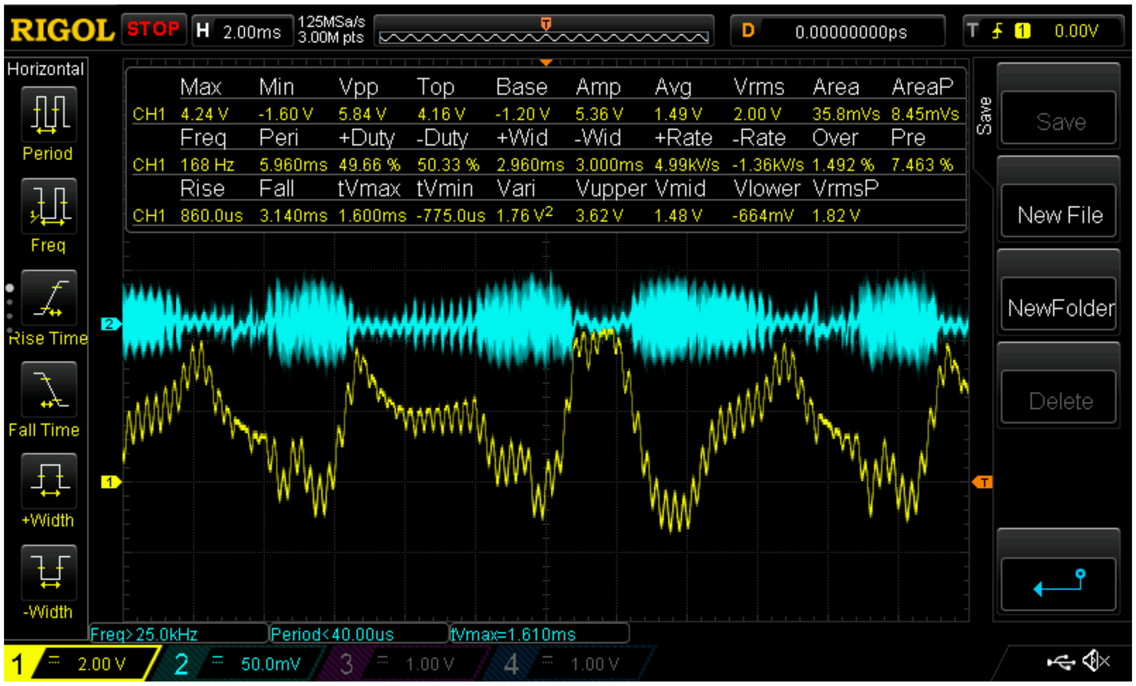
RIGOL STOP H 2.00ms 125MSa/s 3.00M pts D 0.00000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.24 V	-3.92 V	8.16 V	4.16 V	-3.72 V	7.88 V	836mV	2.69 V	20.0mVs	2.17mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	362 Hz	2.760ms	56.52 %	43.48 %	1.560ms	1.200ms	6.43kV/s	-8.52kV/s	1.015 %	2.538 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	980.0us	740.0us	-1.210ms	225.0us	6.54 V^2	3.37 V	220mV	-2.93 V	2.86 V	

Freq=3.85kHz Period=260.0us tVmax=105.0us

1 2.00 V 2 50.0mV 3 1.00 V 4 1.00 V

Figure 4.39: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 300 Hz, sinusoidal input signal.



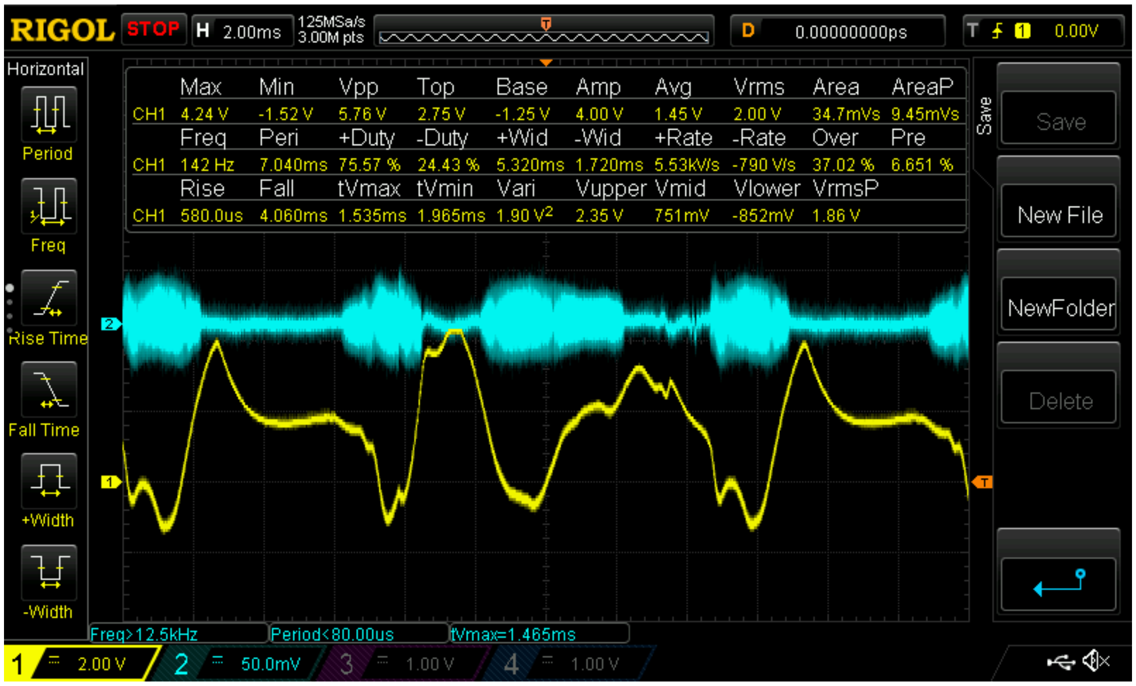
H 2.00ms 125MSa/s 3.00M pts D 0.000000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.24 V	-1.60 V	5.84 V	4.16 V	-1.20 V	5.36 V	1.49 V	2.00 V	35.8mVs	8.45mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	168 Hz	5.960ms	49.66 %	50.33 %	2.960ms	3.000ms	4.99kV/s	-1.36kV/s	1.492 %	7.463 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	860.0us	3.140ms	1.600ms	-775.0us	1.76 V²	3.62 V	1.48 V	-664mV	1.82 V	

Freq>25.0kHz Period<40.00us tVmax=1.610ms

1 2.00 V 2 50.0mV 3 1.00 V 4 1.00 V

Figure 4.40: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 3000 Hz, sinusoidal input signal.



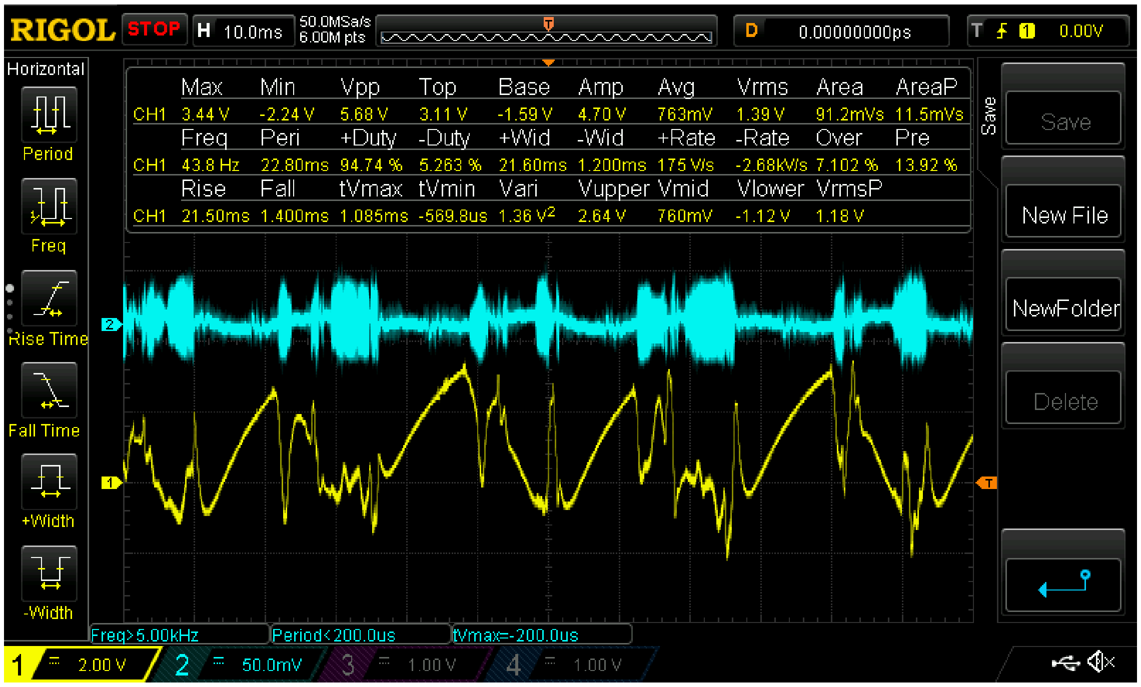
H 2.00ms 125MSa/s 3.00M pts D 0.000000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.24 V	-1.52 V	5.76 V	2.75 V	-1.25 V	4.00 V	1.45 V	2.00 V	34.7mVs	9.45mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	142 Hz	7.040ms	75.57 %	24.43 %	5.320ms	1.720ms	5.53kV/s	-790 V/s	37.02 %	6.651 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	580.0us	4.060ms	1.535ms	1.965ms	1.90 V²	2.35 V	751mV	-852mV	1.86 V	

Freq>12.5kHz Period<80.00us tVmax=1.465ms

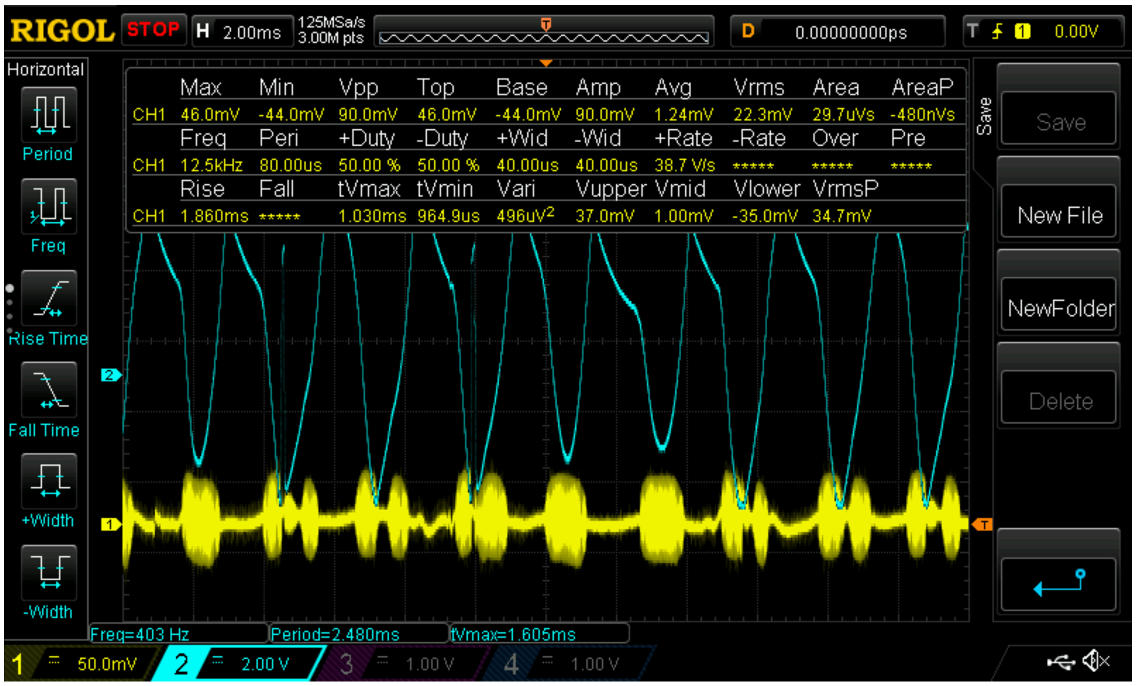
1 2.00 V 2 50.0mV 3 1.00 V 4 1.00 V

Figure 4.41: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 30 kHz, sinusoidal input signal (completely filtered out from output).



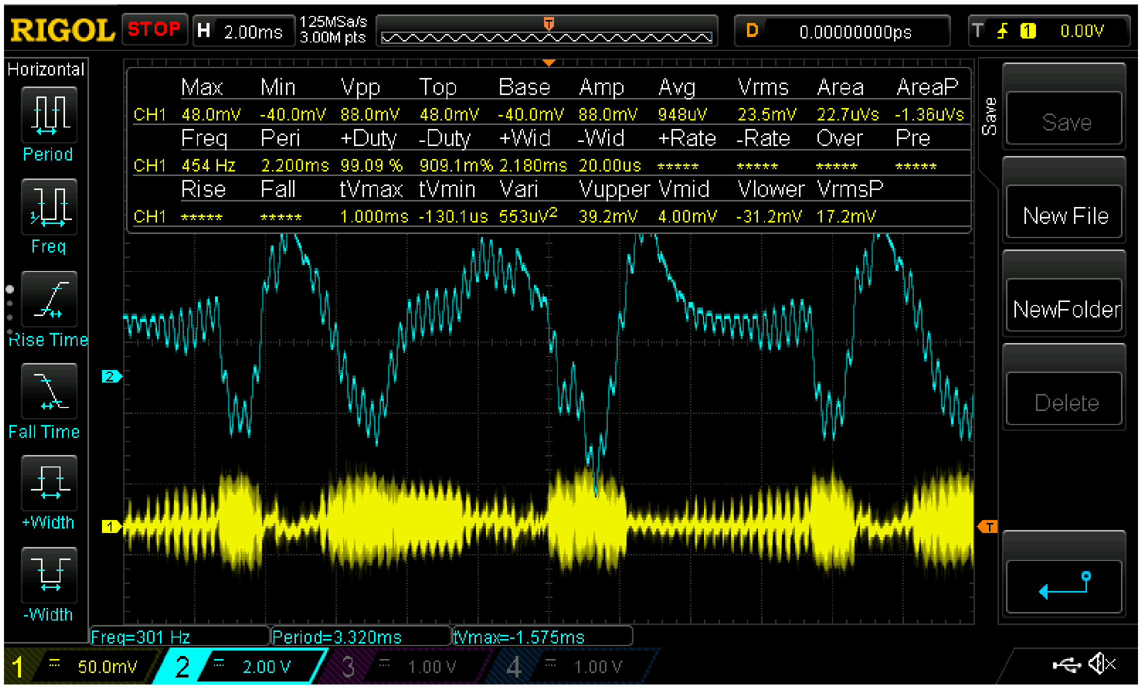
```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area     AreaP
CH1  3.44 V   -2.24 V  5.68 V   3.11 V   -1.59 V  4.70 V   763mV    1.39 V    91.2mVs  11.5mVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over     Pre
CH1  43.8 Hz  22.80ms  94.74 %  5.263 %  21.60ms  1.200ms  175 V/s  -2.68kV/s 7.102 %  13.92 %
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  21.50ms  1.400ms  1.085ms  -569.8us 1.36 V^2 2.64 V   760mV    -1.12 V   1.18 V

Freq>5.00kHz   Period<200.0us   tVmax=-200.0us
```

Figure 4.42: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 30 Hz, sinusoidal input signal.



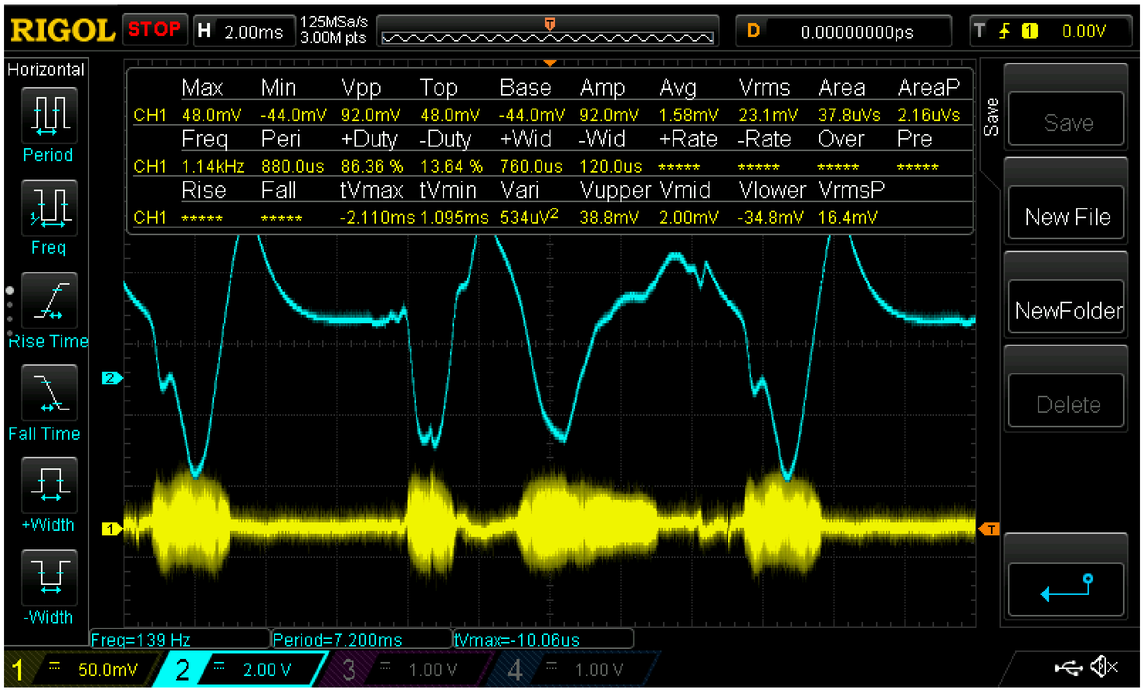
```
     Max      Min      Vpp      Top      Base     Amp      Avg      Vrms      Area     AreaP
CH1  46.0mV   -44.0mV  90.0mV   46.0mV   -44.0mV  90.0mV   1.24mV   22.3mV    29.7uVs  -480nVs
     Freq     Peri     +Duty    -Duty    +Wid     -Wid     +Rate    -Rate     Over     Pre
CH1  12.5kHz  80.00us  50.00 %  50.00 %  40.00us  40.00us  38.7 V/s *****     *****    *****
     Rise     Fall     tVmax    tVmin    Vari     Vupper   Vmid     Vlower    VrmsP
CH1  1.860ms  *****    1.030ms  964.9us  496uV^2  37.0mV   1.00mV   -35.0mV   34.7mV

Freq=403 Hz   Period=2.480ms   tVmax=1.605ms
```

Figure 4.43: Output of right amplifier/filter (yellow) and input (blue) from function generator as a 300 Hz, sinusoidal input signal.



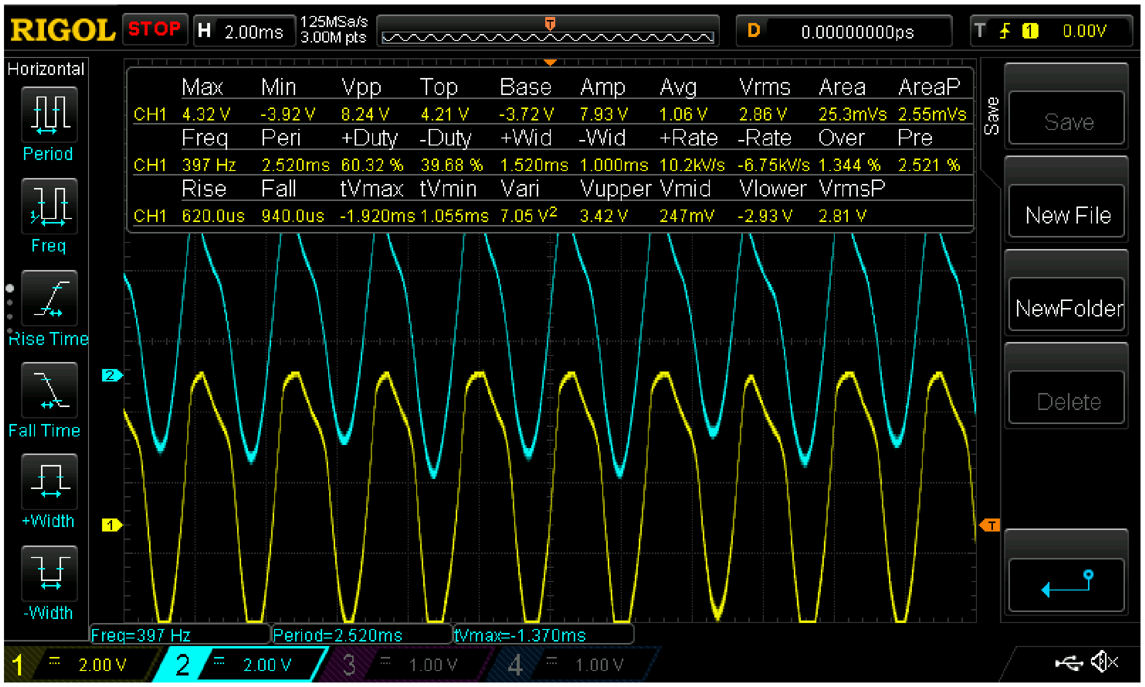
```
RIGOL STOP  H 2.00ms  125MSa/s 3.00M pts   D 0.00000000ps   T 0.00V
     Max     Min      Vpp     Top      Base     Amp     Avg     Vrms    Area     AreaP
CH1  48.0mV  -40.0mV  88.0mV  48.0mV   -40.0mV  88.0mV  948uV   23.5mV  22.7uVs  -1.36uVs
     Freq    Peri     +Duty   -Duty    +Wid     -Wid    +Rate   -Rate   Over     Pre
CH1  454 Hz  2.200ms  99.09 % 909.1m%  2.180ms  20.00us *****   *****   *****    *****
     Rise    Fall     tVmax   tVmin    Vari     Vupper  Vmid    Vlower  VrmsP
CH1  *****   *****    1.000ms -130.1us 553uV²   39.2mV  4.00mV  -31.2mV 17.2mV
Freq=301 Hz   Period=3.320ms   tVmax=-1.575ms
1  50.0mV   2  2.00 V   3  1.00 V   4  1.00 V
```

Figure 4.44: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 3000 Hz, sinusoidal input signal.



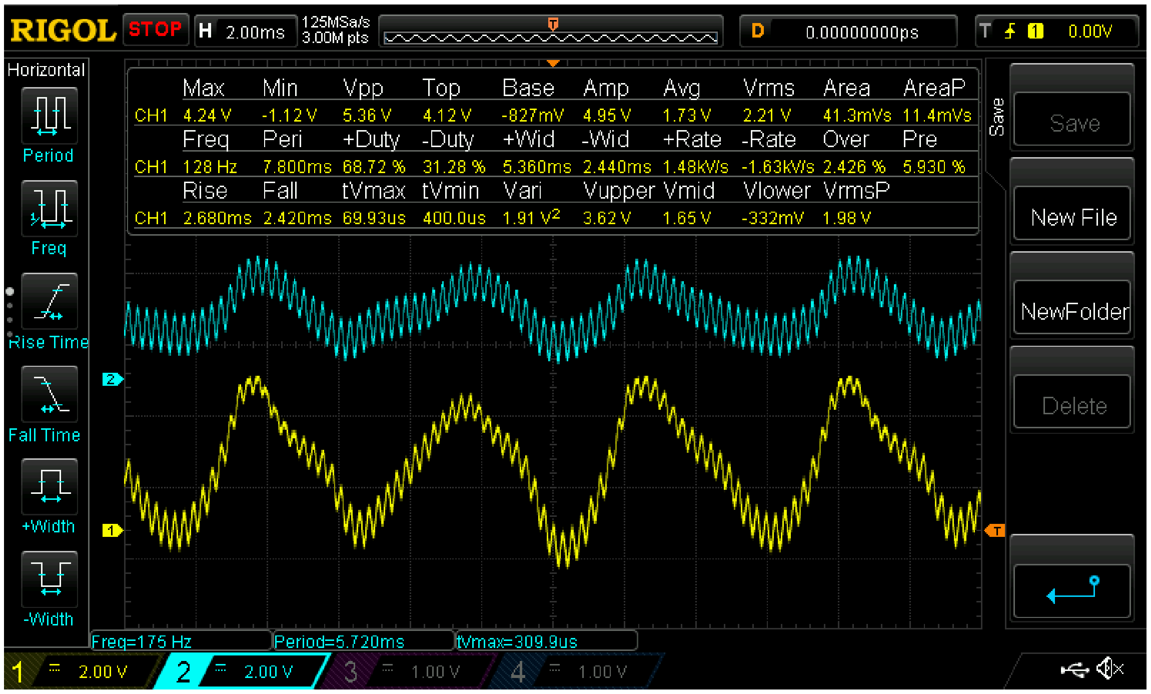
```
RIGOL STOP  H 2.00ms  125MSa/s 3.00M pts   D 0.00000000ps   T 0.00V
     Max     Min      Vpp     Top      Base     Amp     Avg     Vrms    Area     AreaP
CH1  48.0mV  -44.0mV  92.0mV  48.0mV   -44.0mV  92.0mV  1.58mV  23.1mV  37.8uVs  2.16uVs
     Freq    Peri     +Duty   -Duty    +Wid     -Wid    +Rate   -Rate   Over     Pre
CH1  1.14kHz 880.0us  86.36 % 13.64 %  760.0us  120.0us *****   *****   *****    *****
     Rise    Fall     tVmax    tVmin    Vari    Vupper  Vmid    Vlower  VrmsP
CH1  *****   *****    -2.110ms 1.095ms  534uV²  38.8mV  2.00mV  -34.8mV 16.4mV
Freq=139 Hz   Period=7.200ms   tVmax=-10.06us
1  50.0mV   2  2.00 V   3  1.00 V   4  1.00 V
```

Figure 4.45: Output of left amplifier/filter (yellow) and input (blue) from function generator as a 30 kHz, sinusoidal input signal.



```
     Max       Min       Vpp       Top       Base      Amp       Avg       Vrms      Area      AreaP
CH1  4.32 V    -3.92 V   8.24 V    4.21 V    -3.72 V   7.93 V    1.06 V    2.86 V    25.3mVs   2.55mVs
     Freq      Peri      +Duty     -Duty     +Wid      -Wid      +Rate     -Rate     Over      Pre
CH1  397 Hz    2.520ms   60.32 %   39.68 %   1.520ms   1.000ms   10.2kV/s  -6.75kV/s 1.344 %   2.521 %
     Rise      Fall      tVmax     tVmin     Vari      Vupper    Vmid      Vlower    VrmsP
CH1  620.0us   940.0us   -1.920ms  1.055ms   7.05 V²   3.42 V    247mV     -2.93 V   2.81 V

Freq=397 Hz    Period=2.520ms    tVmax=-1.370ms
```

Figure 4.46: Outputs of left (yellow) and right (blue) amplifiers/filters with a 300 Hz, sinusoidal input signal and with coupling capacitors added on inputs (could take input capture when capacitors applied).



```
     Max       Min       Vpp       Top       Base      Amp       Avg       Vrms      Area      AreaP
CH1  4.24 V    -1.12 V   5.36 V    4.12 V    -827mV    4.95 V    1.73 V    2.21 V    41.3mVs   11.4mVs
     Freq      Peri      +Duty     -Duty     +Wid      -Wid      +Rate     -Rate     Over      Pre
CH1  128 Hz    7.800ms   68.72 %   31.28 %   5.360ms   2.440ms   1.48kV/s  -1.63kV/s 2.426 %   5.930 %
     Rise      Fall      tVmax     tVmin     Vari      Vupper    Vmid      Vlower    VrmsP
CH1  2.680ms   2.420ms   69.93us   400.0us   1.91 V²   3.62 V    1.65 V    -332mV    1.98 V

Freq=175 Hz    Period=5.720ms    tVmax=309.9us
```

Figure 4.47: Outputs of left (yellow) and right (blue) amplifiers/filters with a 3000 Hz, sinusoidal input signal and with coupling capacitors added on inputs (could take input capture when capacitors applied).

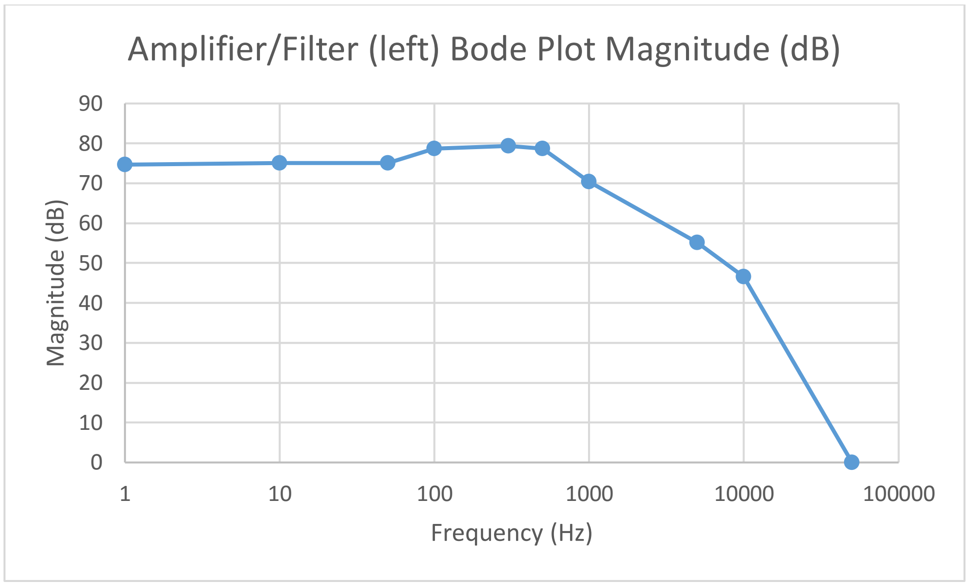


Figure 4.48: Bode plot of magnitude gain of left amplifier/filter (no coupling capacitors).

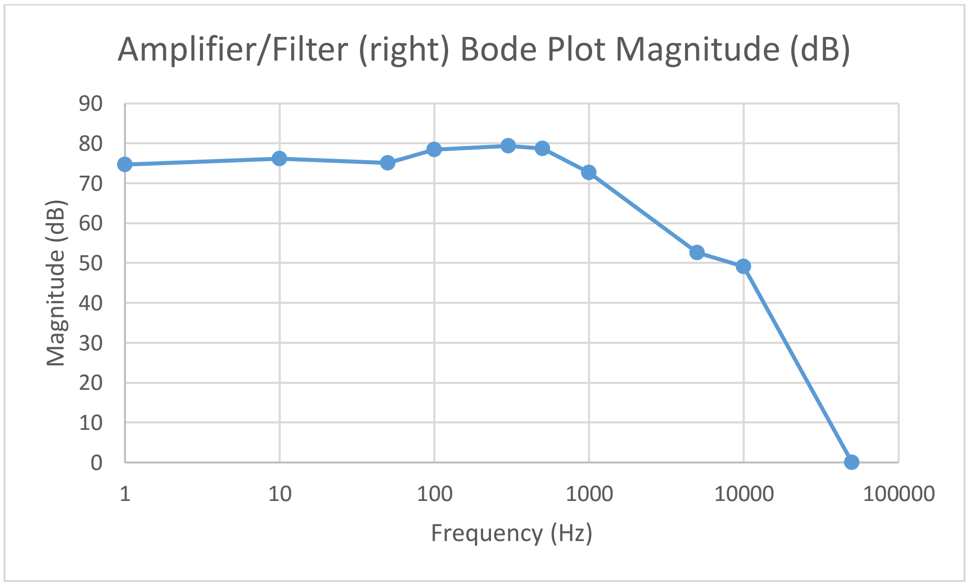


Figure 4.49: Bode plot of magnitude gain of right amplifier/filter (no coupling capacitors).

4.8 Device Operation with Portable Battery Power

[Note: the content in this section was not included in the final design] Since the microcontroller could now successfully take two inputs at once and output their reconstructed and wavelet transformed outputs as one signal while both amplifiers/filters were able to output waveforms with significantly less noise than before, it was time to focus on adding the battery system that would allow the whole circuit and microcontroller to operate without being attached to a large power supply in order to fit the engineering specification of being a handheld, pocket sized device able to be powered by a 5 V battery for up to 24 hours. For the whole system to be able to operate on a portable battery, the microcontroller required a USB compatible port that supplied 5.0 V (the 3.3 V and 5.0 V pins on the microcontroller were already being used to power the external DAC and the button switches, so the only way to power the microcontroller was through the USB input), while the amplifiers/filters required a positive 5.0 V rail and a negative 5.0 V rail with attached grounds. To fulfill these requirements, a battery was needed that provided a USB port and two 5.0 V rails and was also small enough to fit inside a pocket. Thus, to fit these requirements, the Model: TM10, High-Power Automobile Mobile Power Supply, was chosen as the battery for the system as it provides a USB 5.0 V output and two separate outputs supplying 12 V, 16 V, or 19 V DC outputs, which, by using voltage regulators to lower both outputs (both set to 12 V) down to 5 V, could provide negative and positive 5.0 V rails. Unfortunately, the TM10 model battery, being 6.5’’ x 3.0’’ x 1.35’’ (inches), would be a tight fit for a pocket sized battery, but, since no power supply or battery that fulfilled the other requirements could be found in a smaller size, it would have to do for now as the best option. Upon testing the battery with the circuit, the battery could supply the microcontroller with power, but would only last about 6 seconds before suddenly shutting off, while the other outputs of the battery had trouble even outputting any sort of power when trying to supply the rails to the amplifiers/filters, showing that the circuit did not have a large enough load for the battery to supply enough constant current to before triggering the battery to shut off. Since the TM10 model battery was designed to be powering electronics of that of a car, it became clear that a battery was needed that was designed to power a much smaller load and with more compatible output ports for a breadboard circuit (TM10 model had plug outputs that were difficult to attach wires to). Thus, the M-008(YN-010) model power bank was used instead as it was the smallest battery designed to power smaller devices such as phones and laptops with three outputs (three USB ports) that could be found within the $50 budget constraint, it still being about a big as the TM10 model battery. While one USB port would power the microcontroller, the two other USB ports would provide the 5.0 V positive and negative rails for the amplifiers/filters. The rails would be provided by taking two USB cords, cutting off one of their ends while keeping the USB plug, using the 5.0 V (red) and ground (black) wires to supply the rails as shown in Figure 4.50 below, and sodering breadboard compatible wires to the ends of the red and black wires. Black and red cords would be switched for negative rail.

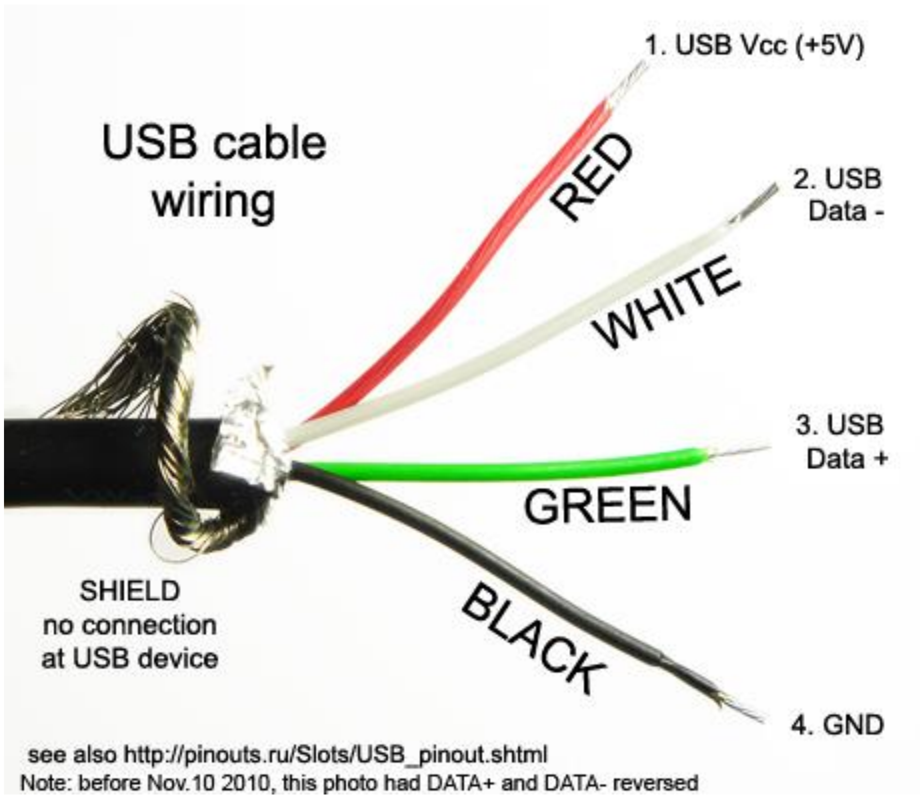
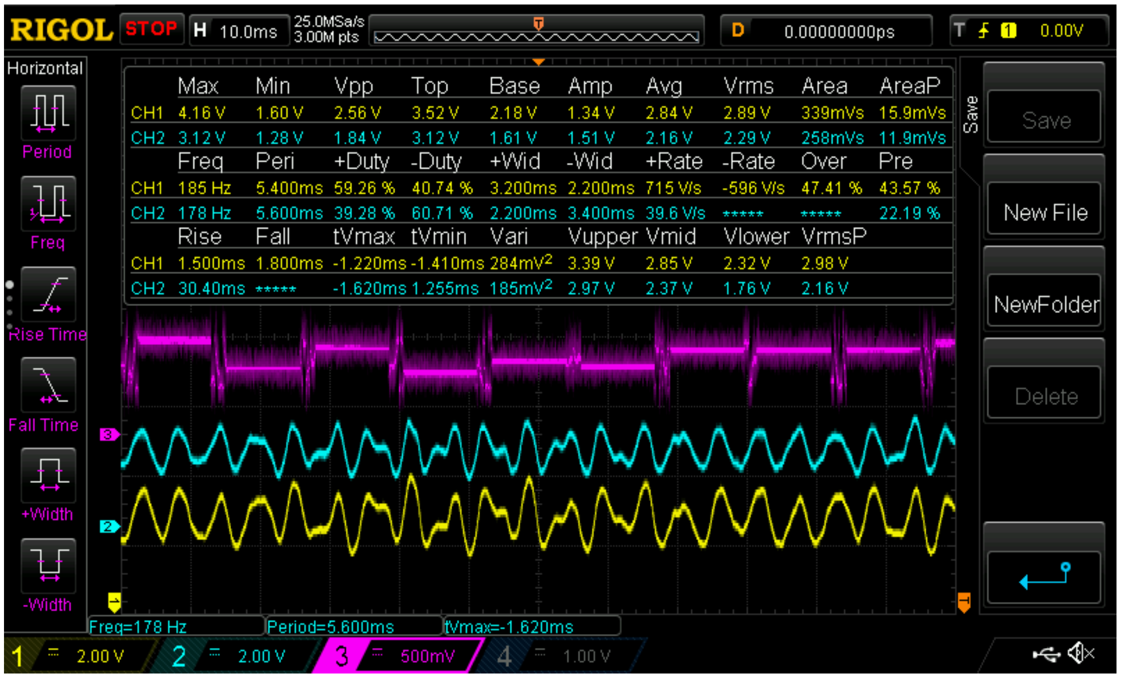

USB cable wiring
1. USB Vcc (+5V)
RED
2. USB Data -
WHITE
3. USB Data +
GREEN
BLACK
4. GND
SHIELD no connection at USB device
see also http://pinouts.ru/Slots/USB_pinout.shtml
Note: before Nov.10 2010, this photo had DATA+ and DATA- reversed

Figure 4.50: Schematic of USB cord.

Unfortunately, after attempting to power the entire circuit, the M-008(YN-010) power bank would not even turn on when trying to supply both rails, but would work if it just supplied a single rail and powered the microcontroller and achieved this without suddenly shutting off. It became clear that the battery was not designed to share grounds with multiple USB ports or else it would not turn on, which is why it could not supply both rails. Thus, in order to power the whole circuit, it was clear that a second battery was needed in which the TM10 battery was used to provide the second rail for the amplifiers/filters. In order to give the TM10 battery a large enough load to not shut off suddenly while supplying power, a resistor was placed across one of the outputs of the TM10 battery in order to draw enough current to act as a proper load for the battery. The resistor value was calculated based on the current draw input of a radio phone that was charged by the battery and the voltage supplied by the battery (when charging the phone, the TM10 battery did not stop supplying power until disconnected so this was assumed to be a large enough load), resulting in 60 Ω (12 V/ 0.2 A). Unfortunately, the resistor did not prevent the battery from shutting off when supplying a single rail, but simply keeping the radio phone attached did. When using both batteries to power the whole circuit (M-008(YN-010) for microcontroller and positive rail and TM10 for negative rail with load), the M-008(YN-010) battery refused to turn on, but operated when not supplying power to the microcontroller and only supplying the positive rail. Both batteries could power the amplifiers/filters, but, unfortunately, only for about 10 seconds before both shutting down. Both batteries provided about 30 mA of current consistently until they shut off, which was measured by placing 0.5 Ω resistors in series with the USB wires providing the 5 V rails and the breadboard, measuring the voltage across them, and calculating the current. They were expected to provide currents of at least 9.4 mA based on the typical currents required to run the four mc33078p chips (4 x 2.05 mA [29]) and the two INA128 chips (2 x 700 µA [28]), not counting the current draw of the resistors and capacitors involved so the measured 30 mA of current was to be expected. The current capacity of the M-008(YN-010) battery was 22400 mAh, while the current capacity of the TM10 battery was 12000 mAh (both read from the back of each battery).

At this point, due to the load problems and the refusal to power more than one device with another battery powering another part of the circuit, it became evident that, in order to properly power the entire circuit, three batteries (one for the microcontroller, positive rail, and negative rail) would be needed that were custom designed to handle the load provided by the circuit, which was unfortunately beyond the budget constraint and resources in possession for this project. However, it was proven that the circuit could operate appropriately while powered by these portable batteries as shown below in Figures 4.51 and 4.52 (amplifiers/filters powered by both batteries, while microcontroller powered by laptop, microcontroller had already been proven to be able to operate on one battery) for at least 10 seconds before having to manually turn on the batteries again. In addition, upon comparing the outputs of circuit when powered by batteries and when powered by the power supply, the amplifiers/filters produced even less noise on their outputs when powered by batteries as opposed to being powered by the power supply as shown by comparing Figures 4.51 and 4.52 to 4.53 and 4.54. Thus, the system could operate on portable batteries and thus be practically portable for at least 10 seconds and, in theory, for much longer if given the resources to create custom batteries for this system.



H 10.0ms 25.0MSa/s 3.00M pts D 0.00000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.16 V	1.60 V	2.56 V	3.52 V	2.18 V	1.34 V	2.84 V	2.89 V	339mVs	15.9mVs
CH2	3.12 V	1.28 V	1.84 V	3.12 V	1.61 V	1.51 V	2.16 V	2.29 V	258mVs	11.9mVs

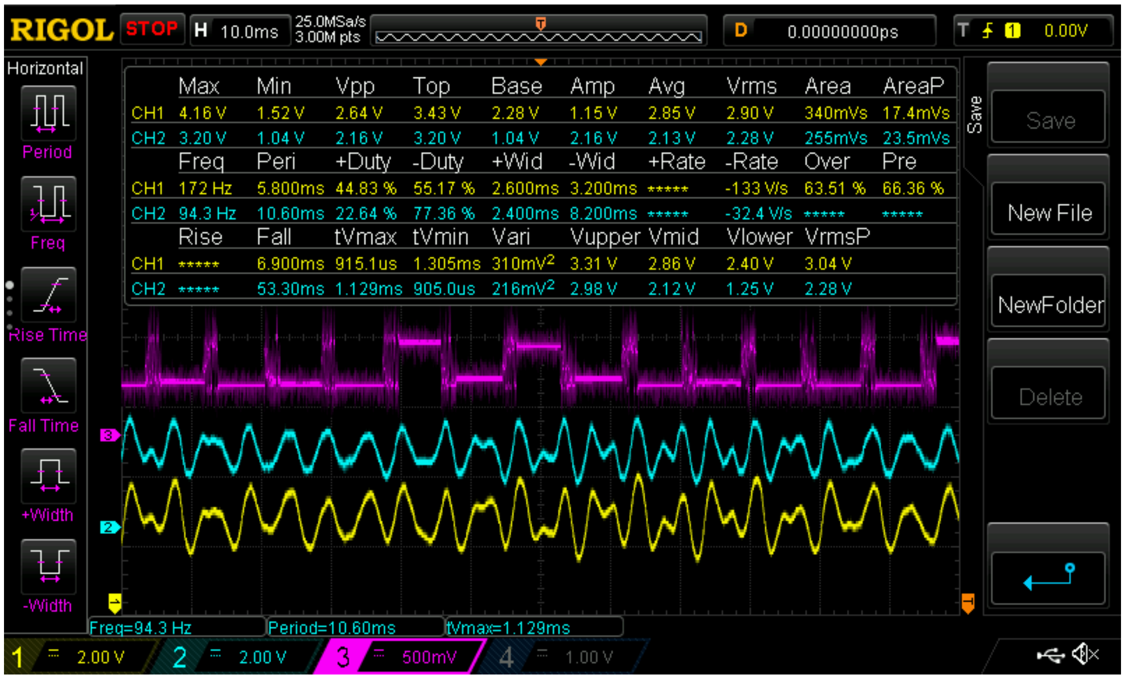
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	185 Hz	5.400ms	59.26 %	40.74 %	3.200ms	2.200ms	715 V/s	-596 V/s	47.41 %	43.57 %
CH2	178 Hz	5.600ms	39.28 %	60.71 %	2.200ms	3.400ms	39.6 V/s	*****	*****	22.19 %

	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP
CH1	1.500ms	1.800ms	-1.220ms	-1.410ms	284mV^2	3.39 V	2.85 V	2.32 V	2.98 V
CH2	30.40ms	*****	-1.620ms	1.255ms	185mV^2	2.97 V	2.37 V	1.76 V	2.16 V

Freq=178 Hz Period=5.600ms tVmax=-1.620ms

1 2.00 V 2 2.00 V 3 500mV 4 1.00 V

Figure 4.51: Outputs of left (yellow) amplifier/filter, right (blue) amplifier/filter, and wavelet transform output of microcontroller (purple), while batteries powered amplifiers/filters.



H 10.0ms 25.0MSa/s 3.00M pts D 0.00000000ps T 0.00V

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.16 V	1.52 V	2.64 V	3.43 V	2.28 V	1.15 V	2.85 V	2.90 V	340mVs	17.4mVs
CH2	3.20 V	1.04 V	2.16 V	3.20 V	1.04 V	2.16 V	2.13 V	2.28 V	255mVs	23.5mVs

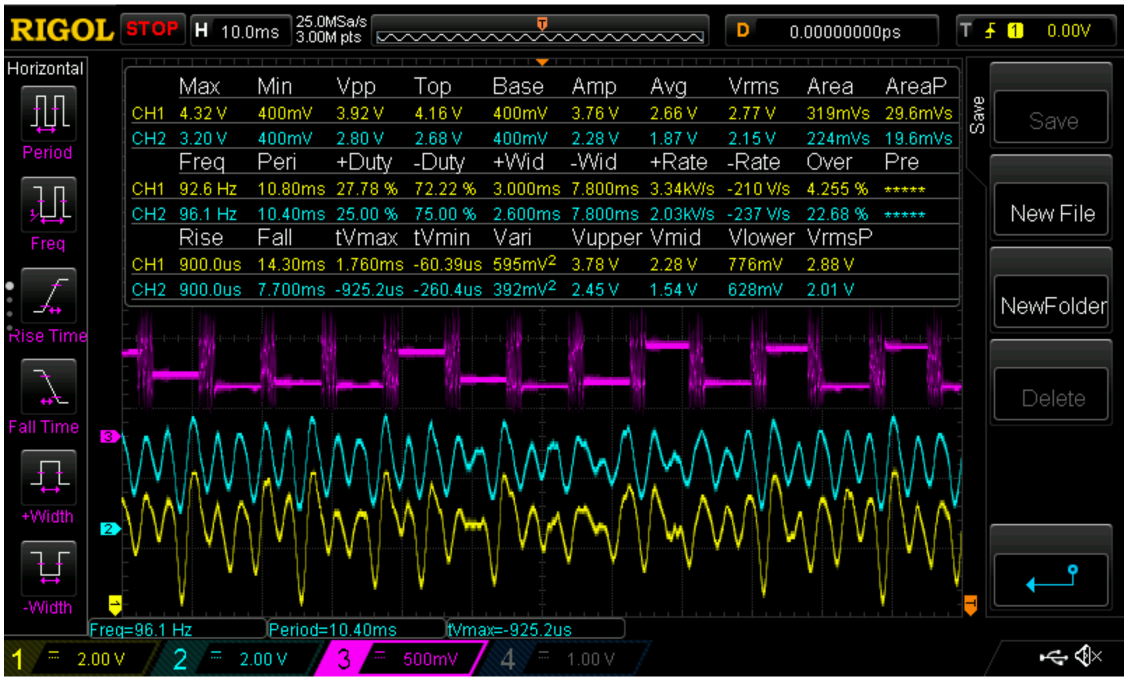
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	172 Hz	5.800ms	44.83 %	55.17 %	2.600ms	3.200ms	*****	-133 V/s	63.51 %	66.36 %
CH2	94.3 Hz	10.60ms	22.64 %	77.36 %	2.400ms	8.200ms	*****	-32.4 V/s	*****	*****

	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP
CH1	*****	6.900ms	915.1us	1.305ms	310mV^2	3.31 V	2.86 V	2.40 V	3.04 V
CH2	*****	53.30ms	1.129ms	905.0us	216mV^2	2.98 V	2.12 V	1.25 V	2.28 V

Freq=94.3 Hz Period=10.60ms tVmax=1.129ms

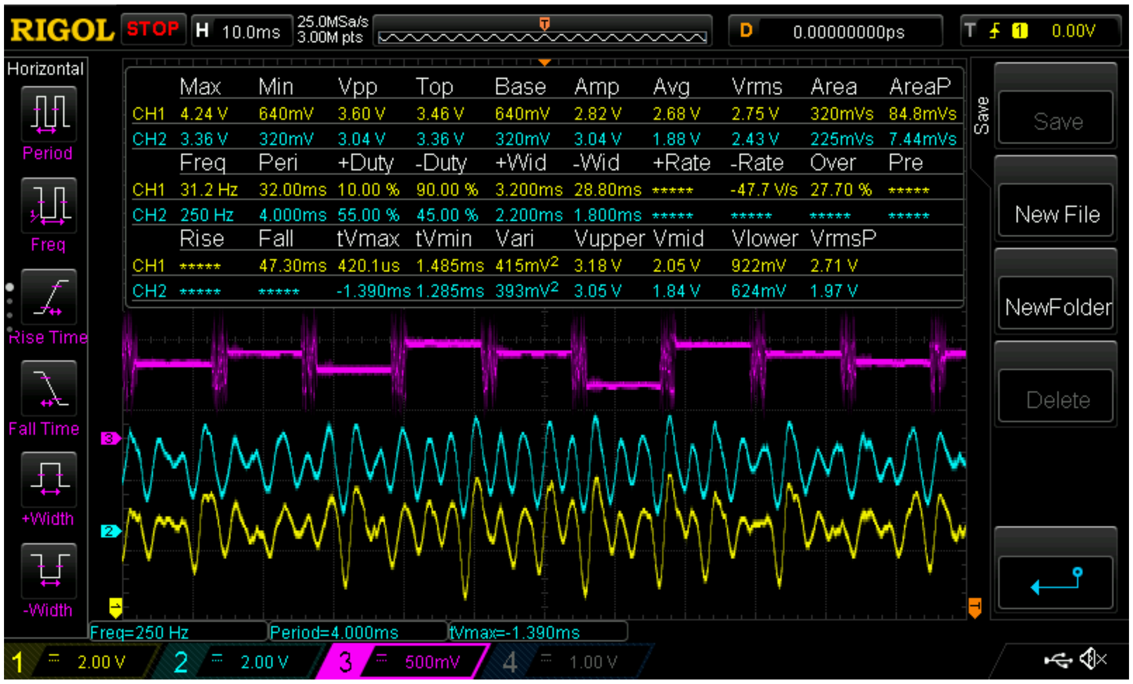
1 2.00 V 2 2.00 V 3 500mV 4 1.00 V

Figure 4.52: Outputs of left (yellow) amplifier/filter, right (blue) amplifier/filter, and reconstructed output of microcontroller (purple), while batteries powered amplifiers/filters.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.32 V	400mV	3.92 V	4.16 V	400mV	3.76 V	2.66 V	2.77 V	319mVs	29.6mVs
CH2	3.20 V	400mV	2.80 V	2.68 V	400mV	2.28 V	1.87 V	2.15 V	224mVs	19.6mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	92.6 Hz	10.80ms	27.78 %	72.22 %	3.000ms	7.800ms	3.34kV/s	-210 V/s	4.255 %	*****
CH2	96.1 Hz	10.40ms	25.00 %	75.00 %	2.600ms	7.800ms	2.03kV/s	-237 V/s	22.68 %	*****
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	900.0us	14.30ms	1.760ms	-60.39us	595mV^2	3.78 V	2.28 V	776mV	2.88 V	
CH2	900.0us	7.700ms	-925.2us	-260.4us	392mV^2	2.45 V	1.54 V	628mV	2.01 V	

Figure 4.53: Outputs of left (yellow) amplifier/filter, right (blue) amplifier/filter, and wavelet transform output of microcontroller (purple), while power supply powered amplifiers/filters (not powered by batteries).



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.24 V	640mV	3.60 V	3.46 V	640mV	2.82 V	2.68 V	2.75 V	320mVs	84.8mVs
CH2	3.36 V	320mV	3.04 V	3.36 V	320mV	3.04 V	1.88 V	2.43 V	225mVs	7.44mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	31.2 Hz	32.00ms	10.00 %	90.00 %	3.200ms	28.80ms	*****	-47.7 V/s	27.70 %	*****
CH2	250 Hz	4.000ms	55.00 %	45.00 %	2.200ms	1.800ms	*****	*****	*****	*****
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	*****	47.30ms	420.1us	1.485ms	415mV^2	3.18 V	2.05 V	922mV	2.71 V	
CH2	*****	*****	-1.390ms	1.285ms	393mV^2	3.05 V	1.84 V	624mV	1.97 V	

Figure 4.54: Outputs of left (yellow) amplifier/filter, right (blue) amplifier/filter, and reconstructed output of microcontroller (purple), while power supply powered amplifiers/filters (not powered by batteries).

4.9 Neural Network and Microcontroller Code Optimization

Next, now that the microcontroller could provide both a wavelet transformed output and a reconstructed output, while two amplifiers/filters could operate with only a little noise, it was time to test the microcontroller with the neural network (wavelet transform implemented in Matlab removed because it was no longer needed) to see if the microcontroller could provide wavelet transformed and reconstructed outputs that could be classified accurately by the neural network and to see what were the best possible settings for the microcontroller to give the most accurately classified results. First, the wavelet transform section of the neural network Matlab code was removed, resulting in 6 inputs to the neural network as opposed to 24 inputs as the wavelet transform split each of the 6 input features into 4 inputs (6 x 4 = 24) so that only the neural network itself remained in the Matlab code (the microcontroller would now provide the wavelet transform). Next, the microcontroller was fed sinusoidal signals from the function generator into both of its inputs, each signal distinguished with a frequency for each vowel: 100 Hz for “A”, 200 Hz for “E”, 300 Hz for “I”, 400 Hz for “O”, and 500 Hz for “U”. These signals would be either wavelet transformed or reconstructed by the microcontroller and then fed into a laptop containing the neural network Matlab code which would attempt to classify these signals in which its accuracy would be observed using a confusion matrix.

Since it was not clear whether or not the 14-bit or 12-bit DAC would give better results due to the 14-bit DAC’s small voltage swing issue but larger resolution, some experimentation was required. Using the 14-bit DAC and the 32 input block (two 16 input blocks each for both inputs), the results proved to be not very consistent, changing drastically in accuracy from test to test. When changing from the 14-bit DAC to the 12-bit DAC, the results became much more consistent with overall accuracies of about 60% to 70% for wavelet transformed outputs and slight improvements (65% to 75%) for reconstructed outputs, showing that the small swing output voltage and small signal-to-noise ratio of the output of the 14-bit DAC was unfortunately making the 14-bit DAC not very reliable. Interestingly, the wavelet transformed output managed to produce higher accuracy than the reconstructed output when the function generator malfunctioned and started producing multiple signals (same frequency as each other, but slightly out of phase with each other like in Figure 4.38) into the input to the microcontroller, showing that the wavelet transform was better suited for dealing with multiple signals than reconstruction, but the results, unfortunately, could not be reproduced due to the function generator no longer malfunctioning.

In addition to external DAC resolution, the size of the input block was also experimented with to see if increasing or decreasing its size would improve results. Since the wavelet transform had to take its inputs in powers of two, the input block size was increased to 64 (two 32 blocks for each input, number of decomposition/reconstruction levels = 6). Unfortunately, this did not improve results as overall accuracies stayed consistently around 60% to 70% for wavelet and 65% to 75% for reconstruct, but, for one test, an important discovery was found when results improved (60% to 90% for wavelet and around 80% for reconstruction) when one of the wires sending an input signal into one of the inputs fell off of the microcontroller, showing that accuracy improved when only one input was being sent into the microcontroller as opposed to two or because the current method of taking two signals via passing both blocks through one wavelet transform was removing vital parts of the signals necessary for classification. After recoding the microcontroller code back to taking only one input (32 block for single input), as opposed to just ignoring the second input by taking zeros (nothing), the performance of the wavelet transform improved from between 60% to 70% overall accuracy to between 80% to 90% accuracy, while that of reconstruction improved from around 80% overall accuracy to between 85% to 90% overall accuracy, showing that the current method of sampling two inputs was causing the results to be less accurate by removing the two input sampling method while still taking only one input. In addition, increasing the 32 single input block to a 64 input block caused performance accuracy to drop by about 10% for both the wavelet transform and reconstruction results, while decreasing the block size to a single 16 block input block drastically improved results (consistently 90% to 100% overall accuracy for both wavelet transform and reconstruction results), showing that the 16 block size was the best in terms of producing accurate results.

Next, to hopefully retain this same accuracy while allowing the microcontroller to take two inputs as opposed to one, the two input sampling method was changed to take both input blocks separately and wavelet transform/reconstruct them separately as well, outputting one after the other. With both inputs having separate wavelet transform/reconstruction functions as opposed to using just one wavelet transform/reconstruct for both inputs, the processing time between taking inputs and outputting the results increased significantly, but, using a 16 block size for both inputs (8 block size was much too small to give enough detail, although was not tested), the resulting overall accuracies stayed consistently at 90% to 100% for both wavelet transform and reconstruction results which showed that this sampling method was much more successful.

After proving which block input size number (16), external DAC (MCP4921), and input sampling method produced the most accurate results, it was time to test which beta coefficients would produce the more accurate results for the wavelet transform and reconstruction functions. When previously calculating the new equation for the simplified lattice filter, the same equation was calculated as that of the equation used in [31] with the exception of having beta coefficients of 1/1.414 and 1/1.414 for both the low-pass and high-pass filters (even and odd inputs were switched as well, but this gives the same result even if they weren’t switched), while the equation from [31] had beta coefficients of 1 and 1/2. The equation in [31] used these coefficients as opposed to the calculated ones presumably because the MSP430 microcontroller lacked the multiplication arithmetic to carry out multiplying the filter outputs by 1/1.414 and used 1 and ½ (shift command) instead. Fortunately, with the MSP432 microcontroller, the multiplication arithmetic can now be used to implement these 1/1.414 beta coefficients to hopefully improve accuracy. After multiple tests, the results showed no relative differences between using the 1/1.414 beta coefficients and the 1 and ½ beta coefficients in terms of accuracy of classification, but the 1/1.414 beta coefficients were kept for final testing anyway since there was no difference in results. Code (final version) is shown in appendix A as Figure A.12 and a flowchart for the final code is shown in section 4.11 in Figure 4.74.

4.10 Amplifier/Filter Optimization

Now that the microcontroller code wavelet transform and reconstruction functions had been optimized for producing the most accurate classification results, it was finally time to test the whole system and see if it was ready for final results. A number of different electrode configurations on the throat were used to experiment and find the positions that gave the most accurate results. These positions included the electrodes being attached across both sides of the larynx (part of throat) with the distance between both positive electrodes being 5 cm across the larynx with the negative electrode placed behind the right ear for grounding as shown in Figure 4.55 below (this configuration was to try and capture the subvocal signals going through the muscles around the "vagus nerves" as shown in Figure 4.56 and referenced from [3]), the electrodes being attached in the same configuration but with the positive electrodes lower down on the throat (referenced from [6]), both positive electrodes placed on both the left and right anterior belly of the digastric muscles with the negative electrode behind the right ear as ground as done in previous tests (Figure 4.11), the positive electrodes on both the left and right anterior belly of the digastric muscles but with the negative electrode attached on the left side of larynx (over vagus nerve muscles), and the positive electrodes on both the left and right mylohyoid muscles (see Figure 4.11 for location of mylohyoid muscles) with the negative electrodes attached on the left side of the larynx (over vagus nerve muscles) as shown in Figure 4.57 below with the exception of not having four electrodes (only one lower electrode, referenced from [13]). In addition to the usual electrodes used [27], two more kinds of electrodes (Meditrace and 3M red dot electrodes) were used to experiment to see if one type gave better results than the others.

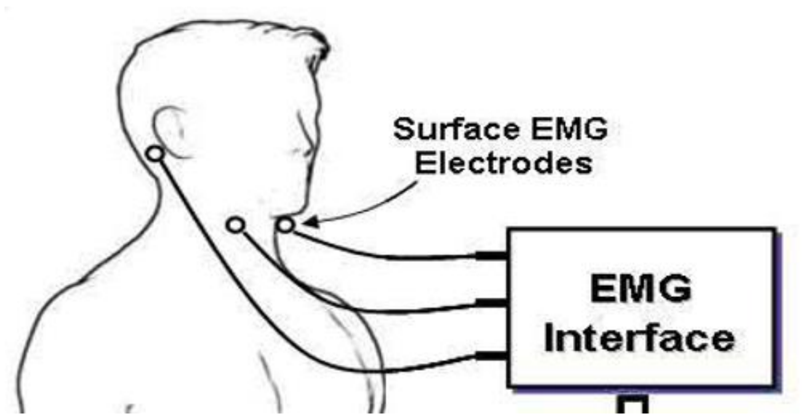


Figure 4.55: Electrodes attached across both sides of the larynx with the distance between both positive electrodes being 5 cm and the negative electrode placed behind the right ear for grounding [3].

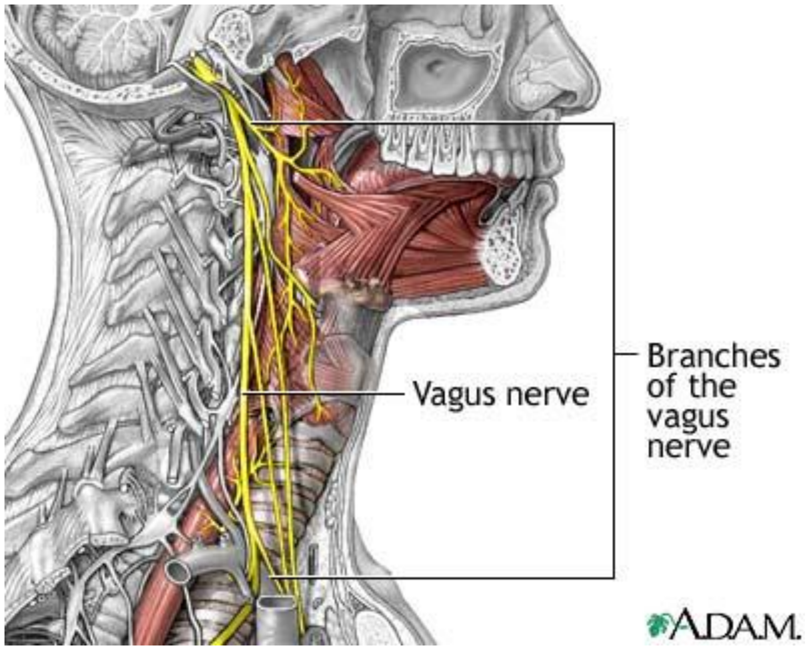


Figure 4.56: Image showing location of the "vagus nerve" and how they control muscles involved in vocal cord and throat movement.

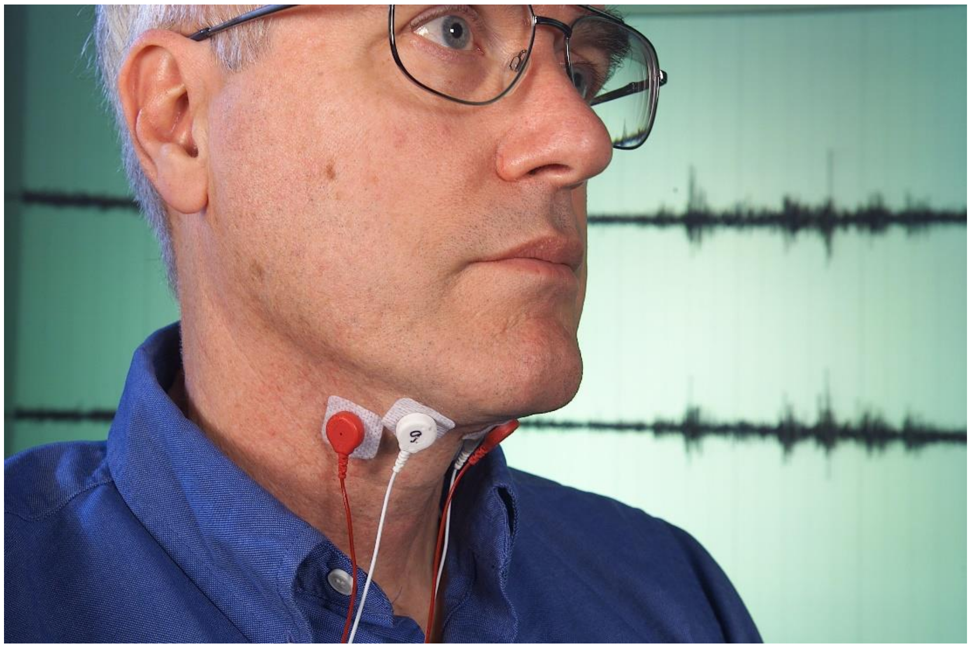
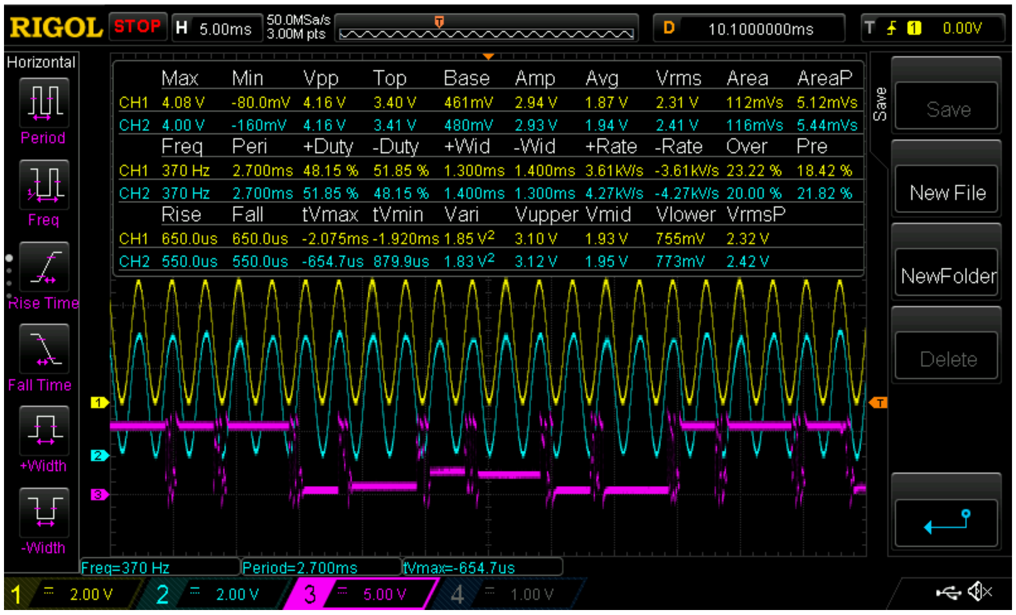


Figure 4.57: Demonstration of final electrode placement: positive electrodes on both the left and right mylohyoid muscles with the negative electrode attached on the left side of the larynx (only one negative electrode over vagus nerve muscle instead of two) [13].

In addition to using multiple different electrode configurations, as a result of many papers on the subject of subvocalization being vague in terms of what exactly creates subvocalization (some claiming triggering it involves just thinking, while others mouth out speech or move throat muscles with the mouth closed as if reading something to themselves in their thoughts), multiples methods of triggering subvocalization were also employed. These involved just thinking of the five vowels used for testing, mouthing out the vowels themselves but without actually speaking them, and reading them off of a paper while keeping mouth closed, making sure that throat muscles move in the process.

Finally, in addition to testing different electrode configurations, different electrode types, and different methods of subvocalization, different levels of gain were also tested with all these conditions to find a balance between amplification of the subvocal signal and noise levels. By switching the “R_g” resistor of the INA128 instrumentation amplifiers with different resistors (gain calculated using equation, 1 + (50 kΩ/x), where x is the resistance of the "R_g" resistor [28]), the gain of the instrumentation amplifier could be modified and multiplied by the gain of the op-amp (100) to produce the total gain. Since the INA128 instrumentation amplifier can provide a maximum gain of 1000 [28], while the mc33078p op-amp can provide a high open-loop AC gain of 800 at 20 kHz with a high-gain bandwidth product of 16 MHz [29], the amplifier/filters could, in theory, be more than capable of providing gain within the specified 10 to 18700 range from the combined gains of the INA128 and mc33078p. The range of gains experimented with were 100 (1 MΩ "R_g" resistor), ≈200 (56 kΩ "R_g" resistor), 600 (10 kΩ "R_g" resistor), 1100 (5 kΩ "R_g" resistor), 5100 (1 kΩ "R_g" resistor), ≈15000 (337 Ω "R_g" resistor), 18600 (270 Ω "R_g" resistor), and 50000 (100 Ω "R_g" resistor). Every time there was a change in gain between tests using electrodes, the amplifiers/filters would be tested using the function generator by comparing the input signal amplitude to the output signal amplitude to make sure it provided the intended gain before testing said gain with subvocal signals.

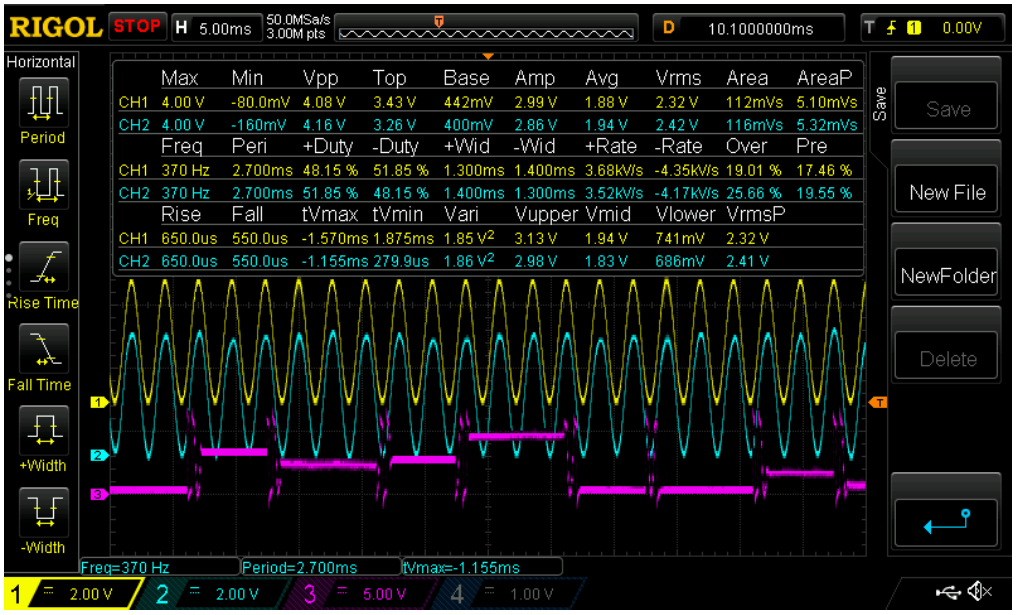
Upon testing the circuit with higher ranges of gain (larger than 6000), the previously mentioned 120 Hz noise became much larger in amplitude in which it became apparent that the amplifiers/filters needed even more noise reduction in order to prevent the subvocal signal from being drowned in noise. At high levels of gain such as 15000, the amplifiers/filters became so sensitive to noise that even raising a hand over them (not even touching the circuit) caused the 120 Hz noise to increase in amplitude significantly. In addition, the amplifiers/filters themselves became much more unstable and unpredictable in behavior, sometimes not providing the gain expected, suddenly refusing to even output a signal, or suddenly being filled with noise at random times. After shortening even more wires, shortening paths from chip to chip, reorganizing directions of wires to be as parallel to other wires close to them as possible and still not completely blocking out the 120 Hz noise, it became clear that some sort of the shielding was needed for the circuit to prevent magnetic fields from surrounding electronic equipment to affect the amplifiers/filters. This was done by taking a small box that could enclose the circuit, wrapping it in aluminum foil, grounding it (earth ground and grounding to the ground of the circuit were both tried), and enclosing the amplifiers/filters inside this box. Despite being such a cheap method for shielding a circuit, the box managed to significantly reduce the 120 Hz noise down to being barely noticeable. Figures 4.58, 4.59, 4.60, and 4.61 below show the outputs of the amplifiers/filters, wavelet transformed and reconstructed outputs of the microcontroller, and input applied to both amplifiers/filters (20 mV, 370 Hz, sinusoidal signal, which was passed through a 1/75 voltage divider, $0.020/75 \times 15000 = 4.0$ V) after applying these changes and the box, showing that the noise had been significantly reduced even at 15000 gain. By applying no input signal, Figure 4.63 shows that some noise still remains but it is much smaller than before.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.08 V	-80.0mV	4.16 V	3.40 V	461mV	2.94 V	1.87 V	2.31 V	112mVs	5.12mVs
CH2	4.00 V	-160mV	4.16 V	3.41 V	480mV	2.93 V	1.94 V	2.41 V	116mVs	5.44mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	370 Hz	2.700ms	48.15 %	51.85 %	1.300ms	1.400ms	3.61kV/s	-3.61kV/s	23.22 %	18.42 %
CH2	370 Hz	2.700ms	51.85 %	48.15 %	1.400ms	1.300ms	4.27kV/s	-4.27kV/s	20.00 %	21.82 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	650.0us	650.0us	-2.075ms	-1.920ms	1.85 V²	3.10 V	1.93 V	755mV	2.32 V	
CH2	550.0us	550.0us	-654.7us	879.9us	1.83 V²	3.12 V	1.95 V	773mV	2.42 V	

Freq=370 Hz Period=2.700ms tVmax=-654.7us

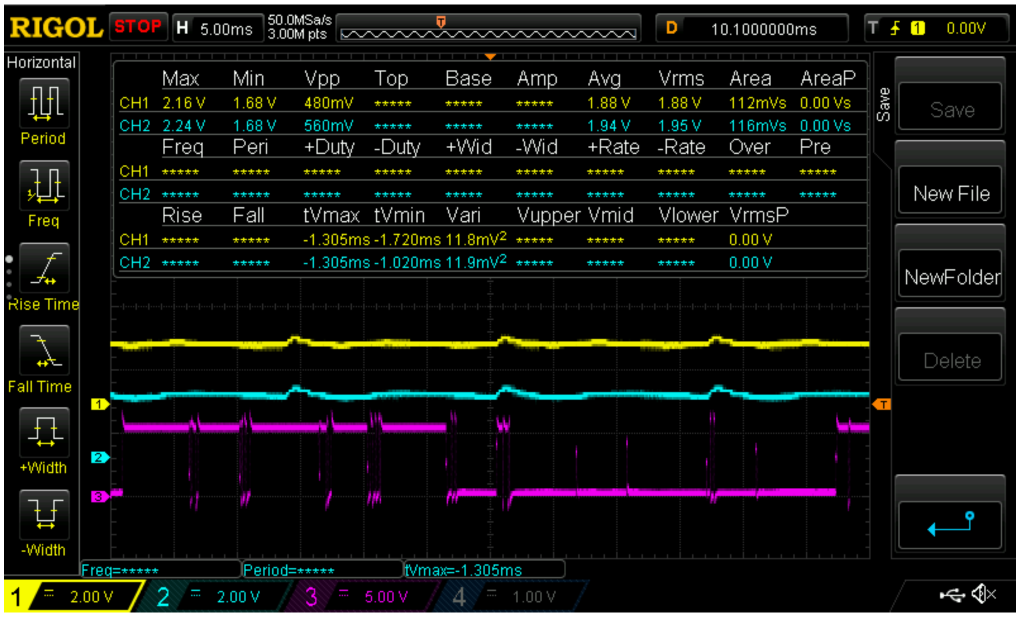
Figure 4.58: Outputs of left (yellow) and right (blue) amplifiers/filters with 15000 gain and a 20 mV, 370 Hz, sinusoidal signal input which was passed through a 1/75 voltage divider after applying foil box and noise reduction rewiring. Wavelet transformed output from microcontroller is shown as the purple signal.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	4.00 V	-80.0mV	4.08 V	3.43 V	442mV	2.99 V	1.88 V	2.32 V	112mVs	5.10mVs
CH2	4.00 V	-160mV	4.16 V	3.26 V	400mV	2.86 V	1.94 V	2.42 V	116mVs	5.32mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	370 Hz	2.700ms	48.15 %	51.85 %	1.300ms	1.400ms	3.68kV/s	-4.35kV/s	19.01 %	17.46 %
CH2	370 Hz	2.700ms	51.85 %	48.15 %	1.400ms	1.300ms	3.52kV/s	-4.17kV/s	25.66 %	19.55 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	650.0us	550.0us	-1.570ms	1.875ms	1.85 V²	3.13 V	1.94 V	741mV	2.32 V	
CH2	650.0us	550.0us	-1.155ms	279.9us	1.86 V²	2.98 V	1.83 V	686mV	2.41 V	

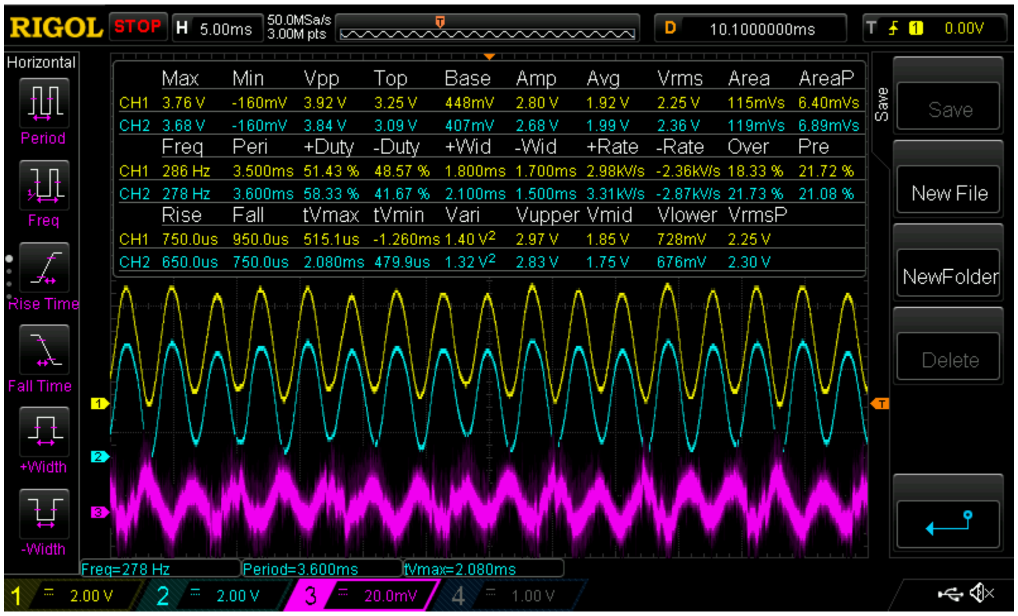
Freq=370 Hz Period=2.700ms tVmax=-1.155ms

Figure 4.59: Outputs of left (yellow) and right (blue) amplifiers/filters with 15000 gain and a 20 mV, 370 Hz, sinusoidal signal input which was passed through a 1/75 voltage divider after applying foil box and noise reduction rewiring. Reconstructed output from microcontroller is shown as the purple signal.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	2.16 V	1.68 V	480mV	*****	*****	*****	1.88 V	1.88 V	112mVs	0.00 Vs
CH2	2.24 V	1.68 V	560mV	*****	*****	*****	1.94 V	1.95 V	116mVs	0.00 Vs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
CH2	*****	*****	*****	*****	*****	*****	*****	*****	*****	*****
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	*****	*****	-1.305ms	-1.720ms	11.8mV^2	*****	*****	*****	0.00 V	
CH2	*****	*****	-1.305ms	-1.020ms	11.9mV^2	*****	*****	*****	0.00 V	

Figure 4.60: Outputs of left (yellow) and right (blue) amplifiers/filters with 15000 gain and no input after applying foil box and noise reduction rewiring. Wavelet transformed output from microcontroller is shown as the purple signal.

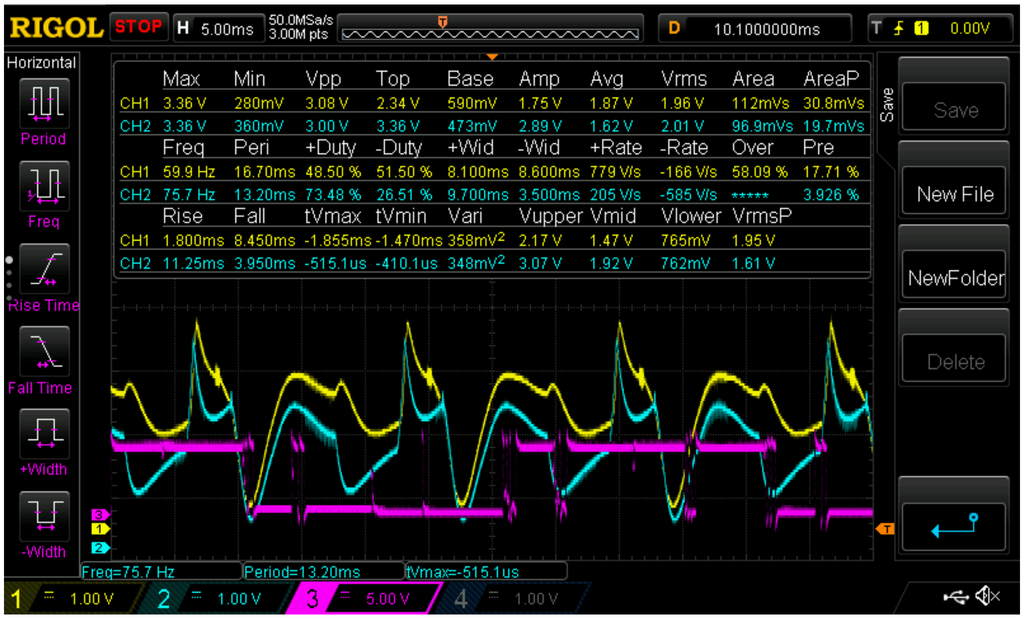


	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	3.76 V	-160mV	3.92 V	3.25 V	448mV	2.80 V	1.92 V	2.25 V	115mVs	6.40mVs
CH2	3.68 V	-160mV	3.84 V	3.09 V	407mV	2.68 V	1.99 V	2.36 V	119mVs	6.89mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	286 Hz	3.500ms	51.43 %	48.57 %	1.800ms	1.700ms	2.98kV/s	-2.36kV/s	18.33 %	21.72 %
CH2	278 Hz	3.600ms	58.33 %	41.67 %	2.100ms	1.500ms	3.31kV/s	-2.87kV/s	21.73 %	21.08 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	750.0us	950.0us	515.1us	-1.260ms	1.40 V^2	2.97 V	1.85 V	728mV	2.25 V	
CH2	650.0us	750.0us	2.080ms	479.9us	1.32 V^2	2.83 V	1.75 V	676mV	2.30 V	

Figure 4.61: Outputs of left (yellow) and right (blue) amplifiers/filters with 15000 gain and a 20 mV, 370 Hz, sinusoidal signal input (purple) which was passed through a 1/75 voltage divider after applying foil box and noise reduction rewiring. Note: trying to capture the input signal after being passed through the voltage divider would inject noise into the input, causing the output to be very noisy so the input could only be captured before going through the voltage divider.

When taking inputs from the surface electrodes, the aluminum foil box proved less effective, as noise still managed to corrupt the outputs of the amplifiers/filters, showing that the noise was coming from the electrodes and not an outside source. This noise was reduced by having the user ground themselves to the ground of the circuit by wrapping an exposed (no isolation) part of a wire around the user's finger that was attached to the ground of the amplifiers/filters. Figures 4.62 and 4.63 show the outputs of the amplifiers/filters when corrupted by noise and after applying the user to ground with no subvocal input, showing that this change significantly reduced the noise.

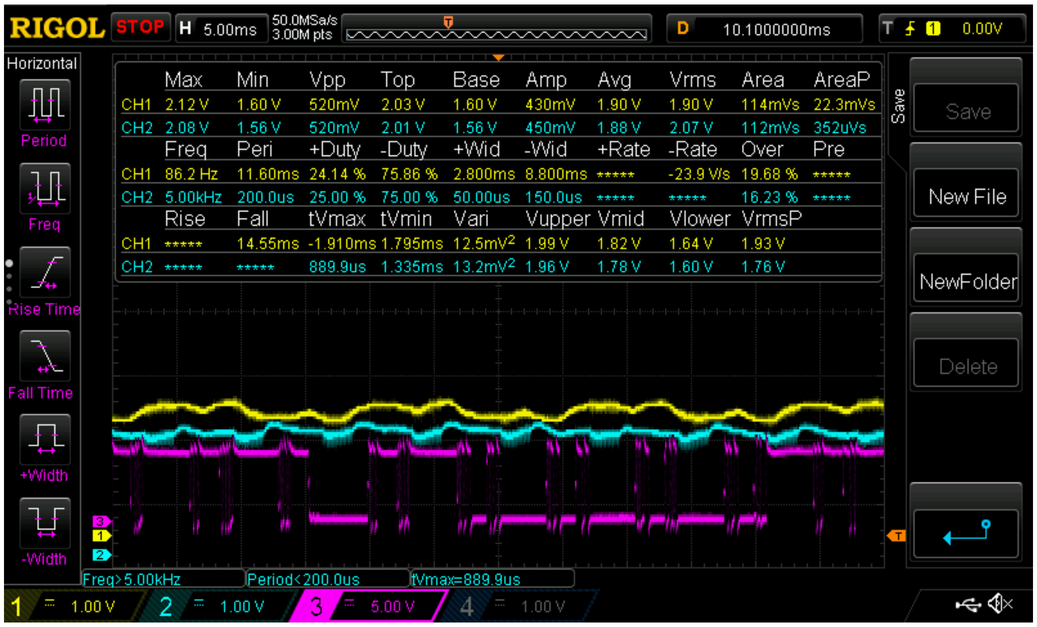
However, this did not remove all problems, as the amplifiers/filters still demonstrated unpredictable behavior at high levels of gain in which, it became less of question as to whether or not the amplifiers/filters could provide high levels of gain, but how long they could sustain it before problems would arise. When the amplifiers/filters would start showing this behavior, getting the amplifiers/filters to behave as expected once again would be achieved by wobbling wires of the circuit with a non-conductive object or temporarily removing the user's ground only to apply it again after a few seconds, showing that the problem was most likely a cause of more isolation problems, but at this point, after reconstructing the amplifier/filter circuit so many times to reduce the impact of magnetic noise and still not being able to completely prevent all isolation issues, it became apparent that the level of isolation needed to completely reduce these problems would require the circuit to no longer be on a breadboard and be rebuilt as an integrated circuit with a high level of isolation. Unfortunately, reconstructing the entire amplifier/filter circuit into an integrated circuit with high levels of isolation was beyond the budget constraint ($50) and resources provided for the project so these problems would have to be endured.



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	3.36 V	280mV	3.08 V	2.34 V	590mV	1.75 V	1.87 V	1.96 V	112mVs	30.8mVs
CH2	3.36 V	360mV	3.00 V	3.36 V	473mV	2.89 V	1.62 V	2.01 V	96.9mVs	19.7mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	59.9 Hz	16.70ms	48.50 %	51.50 %	8.100ms	8.600ms	779 V/s	-166 V/s	58.09 %	17.71 %
CH2	75.7 Hz	13.20ms	73.48 %	26.51 %	9.700ms	3.500ms	205 V/s	-585 V/s	*****	3.926 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	1.800ms	8.450ms	-1.855ms	-1.470ms	358mV²	2.17 V	1.47 V	765mV	1.95 V	
CH2	11.25ms	3.950ms	-515.1us	-410.1us	348mV²	3.07 V	1.92 V	762mV	1.61 V	

Freq=75.7 Hz Period=13.20ms tVmax=-515.1us

Figure 4.62: Outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes without using the user as ground. Wavelet transformed output of microcontroller also shown (purple).



	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	2.12 V	1.60 V	520mV	2.03 V	1.60 V	430mV	1.90 V	1.90 V	114mVs	22.3mVs
CH2	2.08 V	1.56 V	520mV	2.01 V	1.56 V	450mV	1.88 V	2.07 V	112mVs	352uVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	86.2 Hz	11.60ms	24.14 %	75.86 %	2.800ms	8.800ms	*****	-23.9 V/s	19.68 %	*****
CH2	5.00kHz	200.0us	25.00 %	75.00 %	50.00us	150.0us	*****	*****	16.23 %	*****
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	*****	14.55ms	-1.910ms	1.795ms	12.5mV^2	1.99 V	1.82 V	1.64 V	1.93 V	
CH2	*****	*****	889.9us	1.335ms	13.2mV^2	1.96 V	1.78 V	1.60 V	1.76 V	

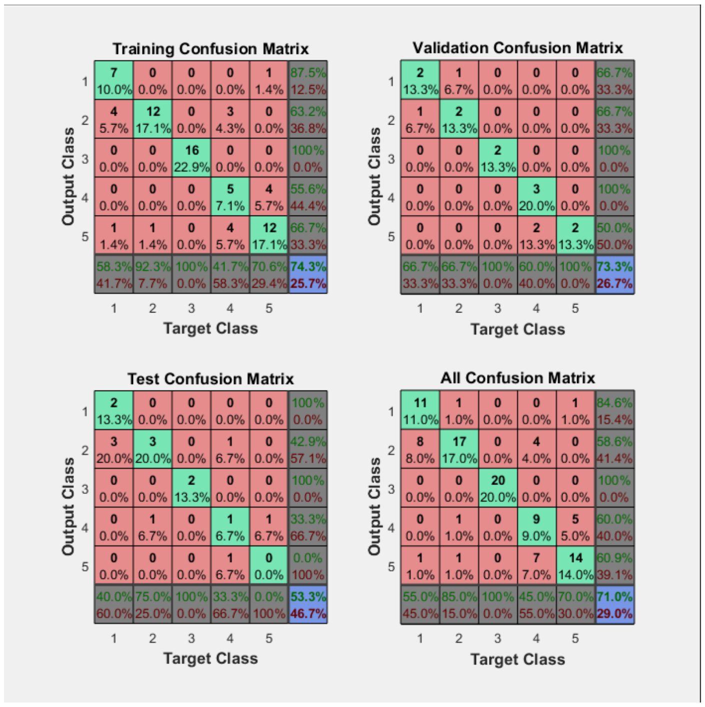
Figure 4.63: Outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes when using the user as ground with no subvocal signal input (user doing nothing). Wavelet transformed output of microcontroller also shown (purple).

4.11 Final Results

Finally, after managing to remove most of the 120 Hz noise and adding all the adjustments and improvements to the project so far within the bounds of the constraints placed on the project, the circuit was finally declared ready to produce the final results. Testing with all the different electrode configurations, electrode types, subvocalization methods, and gains mentioned previously, the combination that produced the most accurate classification results was the electrode configuration in which the positive electrodes were placed on both the left and right mylohyoid muscles with the negative electrode attached on the left side of the larynx (over vagus nerve muscle) as in figure 4.57 except with only three electrodes instead of four, the smallest diameter surface electrode type [27], and subvocalizing while reading vowels off of a paper at 18600 gain, producing a consistent (consistent at least for the first 3 to 5 tests, the decrease in accuracy is explained later) overall accuracy between 69% to 71% for reconstructed outputs and around 45% to 55% for wavelet transformed outputs (70% average accuracy for 5 tests). The inputs for each test were given in the usual training method for the neural network, as vowels were subvocalized as “AAAA”, “EEEE”, “IIII”, “OOOO”, and “UUUU” for twenty one second samples (twenty samples, each being one second long in recording) for each vowel for a total of 100 samples for the neural network for each individual test. For the confusion matrix created from the results of each of these 100 samples (each test), 70% of the data was used for training the neural network, 15% was used to test the neural network, and the final 15% was used to validate the results out of all 100 samples. Examples of confusion matrix results are shown below in Figures 4.64 and 4.65. All other electrode configurations provided signals that only produced overall accuracies of 35% to 50% for both wavelet transformed and reconstructed outputs. Simply thinking of vowels did not produce any subvocal signals, even when attaching electrodes across the vagus nerve muscles, while mouthing out the vowels without actually speaking actually managed to produce signals, but created audible noise via smacking of lips and facial movements in which the method of keeping one’s mouth shut while “reading” vowels proved to be the best method as it produced subvocal signals while keeping audible noise from the user to a minimum. However, it is important to note that subvocal signals were acquired via concentrating **very hard** on moving throat and tongue muscles without opening the mouth (tongue muscles moved involuntarily as subtle movements when subvocalizing using this method). Basically, the user had to pretend like they were reading something and repeating the vowels back inside their head but to an extreme volume (practically "screaming" said words). If the user subvocalized too "quietly" inside their head, the signals would be too small for the microcontroller to accurately sample. As a result, actually obtaining subvocal signals became exhausting for the user, sometimes resulting in headaches after too many tests and increasing the gain beyond 18600 to prevent the user from having to strain themselves so hard only caused the unstable nature of the amplifier/filter to become so prevalent that the circuit became practically unusable. The fact that the throat and tongue muscles were so heavily involved in producing the necessary EMG subvocal signals, however, shows why the configuration that placed the positive electrodes on the mylohyoid muscles and the negative electrode on the vagus nerve controlled muscles was able to produce the best accuracy as a result of the fact that mylohyoid muscles help elevate the tongue and hyoid [44] in which they were able to produce EMG signals that signified the subtle tongue movements the user employed while subvocalizing, while the negative electrode provided a ground where muscles controlled by the vagus nerves emitted EMG signals that vibrate vocal cords inside the larynx, creating a input signal capturing features from both the tongue movements and vocal cord vibration by taking their difference while the other configurations focused on taking signals from muscles that had less of a role in subvocalization. Levels of gain that were lower than 18600 did not amplify the subvocal signals to be large enough for the microcontroller to sample, while, as stated previously, attempting to increase the gain to be even larger than 18600 (say, 50000) caused the unstable nature of the amplifier/filter circuit to become a lot more prevalent to the point where the circuit was practically unusable. The other types of electrodes other than the [27] type of electrode performed worse in terms of accuracy of classification because the

[27] electrode type had the smallest diameter, allowing it to pick up signals on muscles with more precision with less chance of accidentally overlapping with other muscles producing different signals.

Upon analyzing the subvocal signals themselves, the signals demonstrated specific behaviors proving that they are, in fact, subvocal signals (note: subvocal signals were originally defined as the EMG signals causing the vocal cords to vibrate when thinking of words and noises, but since the subtle, involuntary movements of the tongue while subvocalizing also clearly played a role in classification accuracy, the EMG signals going to the tongue through the mylohyoid muscles are also going to be defined as subvocal signals). For example, Figure 4.63 above shows the outputs of both amplifiers/filters and microcontroller (wavelet transformed output) without any subvocalization, while Figures 4.66, 4.67, 4.68, 4.69, 4.70, and 4.71 below show the outputs of both amplifiers/filters (blue was the electrode on the left mylohyoid muscle and yellow was the electrode on the right mylohyoid muscle, while negative electrode (ground) was vagus nerve muscle left of the larynx) and microcontroller (wavelet transformed output) with subvocalization for each vowel emitted, in which, unlike in the previous tests of Figures 4.14, 4.15, 4.16, and 4.17, there are clear distinctions between the instances where no subvocalization is happening and where subvocalization is happening as well as distinctions between each "vowel" of subvocalization as well, showing that any subvocal signals that might have existed in the tests of Figures 4.14, 4.15, 4.16, and 4.17 were clearly buried in noise and had too little amplification to be seen. In addition, when subvocalizing each vowel, the oscilloscope would show a jump in amplitude at the beginning of each "vowel" held ("aaaaaaa" held consistently) as shown in Figure 4.66 below and then lower into a constant amplitude with a somewhat repeating sequence of frequencies, the jump in amplitude showing the mylohyoid muscles stretching and contracting when the tongue first moves and then staying constant as the mylohyoid muscles hold the position of the tongue until release, while the somewhat repeating sequence of frequencies shows some sort of an organized signal as opposed to random noise and thus showing that the signals being captured were most likely EMG signals. It is also important to note that the gain and classification accuracy of the subvocal signals would decrease gradually over time, usually after about 3 to 5 tests (decreasing from the range of 69% to 71% accuracy gradually down to approximately 65% by the sixth test and then continue to decrease for each successive test), but would return to 69% to 71% accuracy when the electrodes were replaced with new ones. Since this decrease in classification accuracy over time did not occur when testing with the function generator (function generator providing the input) and would restart upon replacing the electrodes with new ones (electrodes were only one-time use), this phenomenon was most likely the cause of the sweat building under the electrode over the course of testing as a result of the large amount of effort the user would have to exert to get the subvocal signals to be a large enough amplitude to be sampled by the microcontroller. In addition, the amount of effort needed to sustain a steady subvocal signal also made it more and more difficult over time to continue tests with the same quality of subvocal signal unless a rest was taken for the user. This was also the reason why only one user was used for these tests as the strain on the user to obtain a quality subvocal signal was too much to force on someone else without due payment and because the EMG electrodes became saturated with sweat too quickly in which it would be too expensive to buy more for multiple users for multiple tests. It should also be noted that the reason the subvocal signals from the left amplifier/filter were, in general, smaller than the subvocal signals from the right amplifier/filter as shown in the Figures below was most likely because of the fact that the ground (negative) electrode was placed on the vagus nerve muscle to the left of the larynx, making an asymmetric electrode placement because no fourth electrode was available in which both positive electrodes would be receiving different signals of varying amplitude due to the fact that there was a larger difference in distance across the throat between the negative electrode (ground) and the left positive electrode compared to the distance between the same negative electrode (ground) and the right positive electrode.


Training Confusion Matrix
Validation Confusion Matrix
Test Confusion Matrix
All Confusion Matrix
Output Class
Target Class

Training Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	
1	7 10.0%	0 0.0%	0 0.0%	0 0.0%	1 1.4%	87.5% 12.5%
2	4 5.7%	12 17.1%	0 0.0%	3 4.3%	0 0.0%	63.2% 36.8%
3	0 0.0%	0 0.0%	16 22.9%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	5 7.1%	4 5.7%	55.6% 44.4%
5	1 1.4%	1 1.4%	0 0.0%	4 5.7%	12 17.1%	66.7% 33.3%
	58.3% 41.7%	92.3% 7.7%	100% 0.0%	41.7% 58.3%	70.6% 29.4%	74.3% 25.7%

Validation Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	
1	2 13.3%	1 6.7%	0 0.0%	0 0.0%	0 0.0%	66.7% 33.3%
2	1 6.7%	2 13.3%	0 0.0%	0 0.0%	0 0.0%	66.7% 33.3%
3	0 0.0%	0 0.0%	2 13.3%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	0 0.0%	0 0.0%	3 20.0%	0 0.0%	100% 0.0%
5	0 0.0%	0 0.0%	0 0.0%	2 13.3%	2 13.3%	50.0% 50.0%
	66.7% 33.3%	66.7% 33.3%	100% 0.0%	60.0% 40.0%	100% 0.0%	73.3% 26.7%

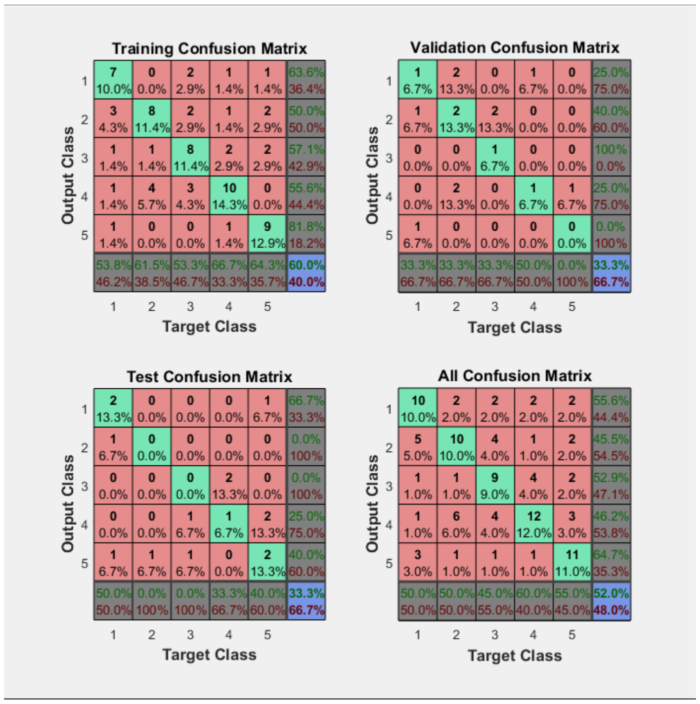
Test Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	
1	2 13.3%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	100% 0.0%
2	3 20.0%	3 20.0%	0 0.0%	1 6.7%	0 0.0%	42.9% 57.1%
3	0 0.0%	0 0.0%	2 13.3%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	1 6.7%	0 0.0%	1 6.7%	1 6.7%	33.3% 66.7%
5	0 0.0%	0 0.0%	0 0.0%	1 6.7%	0 0.0%	0.0% 100%
	40.0% 60.0%	75.0% 25.0%	100% 0.0%	33.3% 66.7%	0.0% 100%	53.3% 46.7%

All Confusion Matrix

Output Class \ Target Class	1	2	3	4	5	
1	11 11.0%	1 1.0%	0 0.0%	0 0.0%	1 1.0%	84.6% 15.4%
2	8 8.0%	17 17.0%	0 0.0%	4 4.0%	0 0.0%	58.6% 41.4%
3	0 0.0%	0 0.0%	20 20.0%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	1 1.0%	0 0.0%	9 9.0%	5 5.0%	60.0% 40.0%
5	1 1.0%	1 1.0%	0 0.0%	7 7.0%	14 14.0%	60.9% 39.1%
	55.0% 45.0%	85.0% 15.0%	100% 0.0%	45.0% 55.0%	70.0% 30.0%	71.0% 29.0%

Figure 4.64: Final example of confusion matrix result for reconstructed subvocal signals at 18600 gain within the first 3 to 5 tests of newly applied surface electrodes.



Training Confusion Matrix

Output Class	1	2	3	4	5	
1	7 10.0%	0 0.0%	2 2.9%	1 1.4%	1 1.4%	63.6% 36.4%
2	3 4.3%	8 11.4%	2 2.9%	1 1.4%	2 2.9%	50.0% 50.0%
3	1 1.4%	1 1.4%	8 11.4%	2 2.9%	2 2.9%	57.1% 42.9%
4	1 1.4%	4 5.7%	3 4.3%	10 14.3%	0 0.0%	55.6% 44.4%
5	1 1.4%	0 0.0%	0 0.0%	1 1.4%	9 12.9%	81.8% 18.2%
	53.8% 46.2%	61.5% 38.5%	53.3% 46.7%	66.7% 33.3%	64.3% 35.7%	60.0% 40.0%

Target Class

Validation Confusion Matrix

Output Class	1	2	3	4	5	
1	1 6.7%	2 13.3%	0 0.0%	1 6.7%	0 0.0%	25.0% 75.0%
2	1 6.7%	2 13.3%	2 13.3%	0 0.0%	0 0.0%	40.0% 60.0%
3	0 0.0%	0 0.0%	1 6.7%	0 0.0%	0 0.0%	100% 0.0%
4	0 0.0%	2 13.3%	0 0.0%	1 6.7%	1 6.7%	25.0% 75.0%
5	1 6.7%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0.0% 100%
	33.3% 66.7%	33.3% 66.7%	33.3% 66.7%	50.0% 50.0%	0.0% 100%	33.3% 66.7%

Target Class

Test Confusion Matrix

Output Class	1	2	3	4	5	
1	2 13.3%	0 0.0%	0 0.0%	0 0.0%	1 6.7%	66.7% 33.3%
2	1 6.7%	0 0.0%	0 0.0%	0 0.0%	0 0.0%	0.0% 100%
3	0 0.0%	0 0.0%	0 0.0%	2 13.3%	0 0.0%	0.0% 100%
4	0 0.0%	0 0.0%	1 6.7%	1 6.7%	2 13.3%	25.0% 75.0%
5	1 6.7%	1 6.7%	1 6.7%	0 0.0%	2 13.3%	40.0% 60.0%
	50.0% 50.0%	0.0% 100%	0.0% 100%	33.3% 66.7%	40.0% 60.0%	33.3% 66.7%

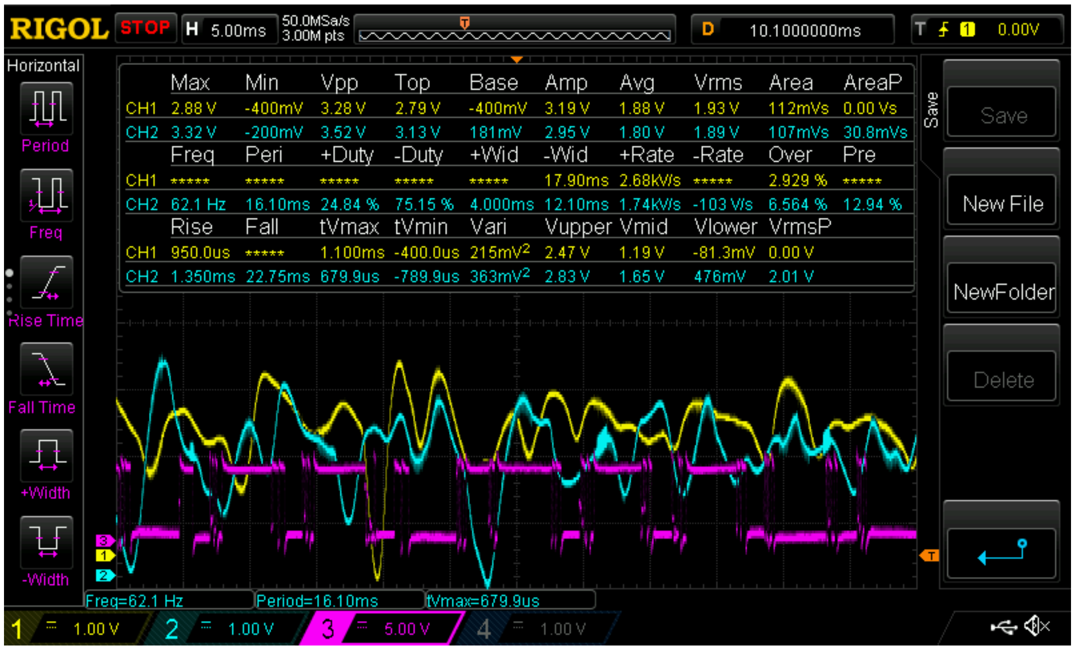
Target Class

All Confusion Matrix

Output Class	1	2	3	4	5	
1	10 10.0%	2 2.0%	2 2.0%	2 2.0%	2 2.0%	55.6% 44.4%
2	5 5.0%	10 10.0%	4 4.0%	1 1.0%	2 2.0%	45.5% 54.5%
3	1 1.0%	1 1.0%	9 9.0%	4 4.0%	2 2.0%	52.9% 47.1%
4	1 1.0%	6 6.0%	4 4.0%	12 12.0%	3 3.0%	46.2% 53.8%
5	3 3.0%	1 1.0%	1 1.0%	1 1.0%	11 11.0%	64.7% 35.3%
	50.0% 50.0%	50.0% 50.0%	45.0% 55.0%	60.0% 40.0%	55.0% 45.0%	52.0% 48.0%

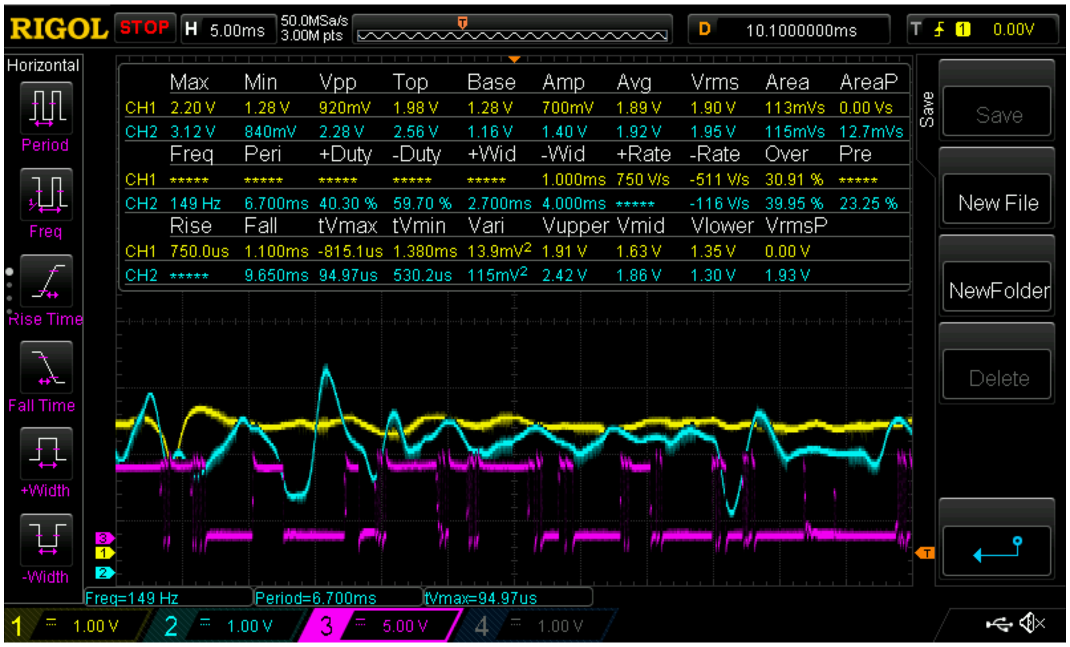
Target Class

Figure 4.65: Final example of confusion matrix result for wavelet transformed subvocal signals at 18600 gain within the first 3 to 5 tests of newly applied surface electrodes.



```
     Max     Min     Vpp     Top      Base     Amp     Avg      Vrms     Area    AreaP
CH1  2.88 V  -400mV  3.28 V  2.79 V   -400mV   3.19 V  1.88 V   1.93 V   112mVs  0.00 Vs
CH2  3.32 V  -200mV  3.52 V  3.13 V   181mV    2.95 V  1.80 V   1.89 V   107mVs  30.8mVs
     Freq    Peri    +Duty   -Duty    +Wid     -Wid    +Rate    -Rate    Over    Pre
CH1  *****   *****   *****   *****    *****    17.90ms 2.68kV/s *****    2.929 % *****
CH2  62.1 Hz 16.10ms 24.84 % 75.15 %  4.000ms  12.10ms 1.74kV/s -103 V/s 6.564 % 12.94 %
     Rise    Fall    tVmax   tVmin    Vari     Vupper  Vmid     Vlower   VrmsP
CH1  950.0us *****   1.100ms -400.0us 215mV²   2.47 V  1.19 V   -81.3mV  0.00 V
CH2  1.350ms 22.75ms 679.9us -789.9us 363mV²   2.83 V  1.65 V   476mV    2.01 V
```

Figure 4.66: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with the “AAAA” subvocal signal input “spike” at the beginning of muscle contraction. Wavelet transformed output of microcontroller also shown (purple).

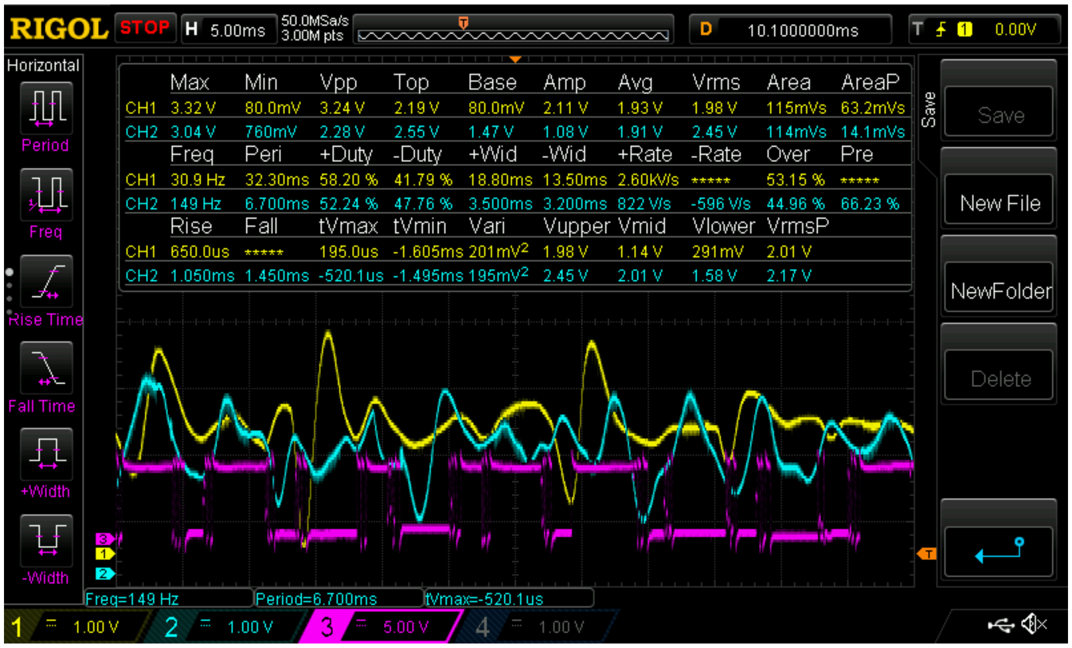


```
H 5.00ms   50.0MSa/s 3.00M pts   D 10.1000000ms   T 0.00V
```

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	2.20 V	1.28 V	920mV	1.98 V	1.28 V	700mV	1.89 V	1.90 V	113mVs	0.00 Vs
CH2	3.12 V	840mV	2.28 V	2.56 V	1.16 V	1.40 V	1.92 V	1.95 V	115mVs	12.7mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	*****	*****	*****	*****	*****	1.000ms	750 V/s	-511 V/s	30.91 %	*****
CH2	149 Hz	6.700ms	40.30 %	59.70 %	2.700ms	4.000ms	*****	-116 V/s	39.95 %	23.25 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	750.0us	1.100ms	-815.1us	1.380ms	13.9mV^2	1.91 V	1.63 V	1.35 V	0.00 V	
CH2	*****	9.650ms	94.97us	530.2us	115mV^2	2.42 V	1.86 V	1.30 V	1.93 V	

```
Freq=149 Hz   Period=6.700ms   tVmax=94.97us
1  1.00 V   2  1.00 V   3  5.00 V   4  1.00 V
```

Figure 4.67: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with consistent “AAAA” subvocal signal input after initial “spike”. Wavelet transformed output of microcontroller also shown (purple).

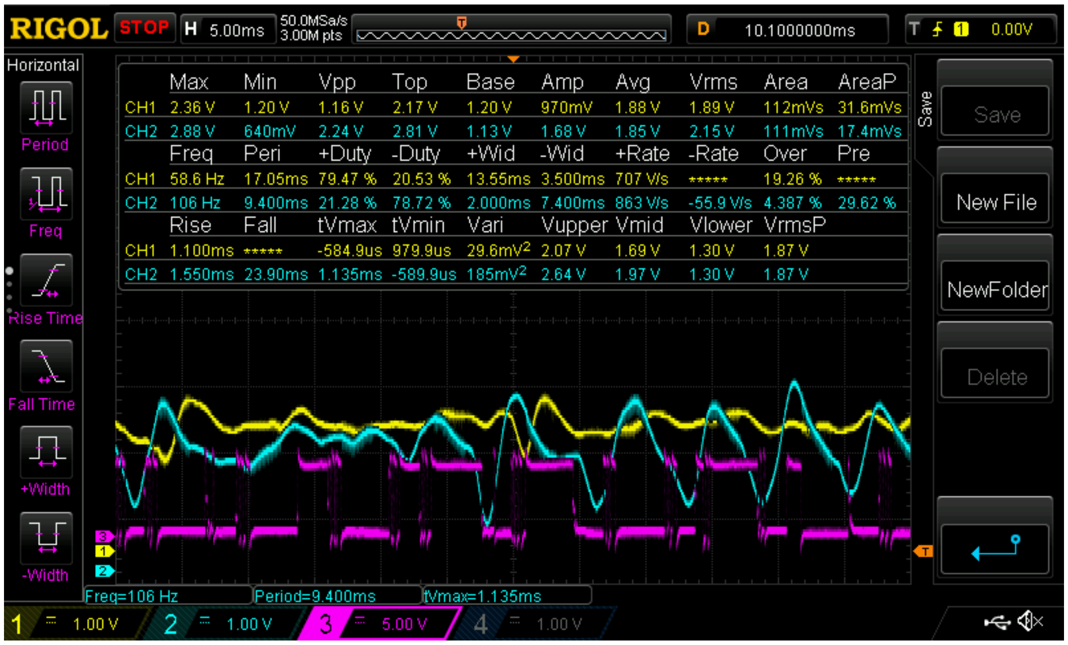


```
H 5.00ms   50.0MSa/s 3.00M pts   D 10.1000000ms   T 0.00V
```

	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	3.32 V	80.0mV	3.24 V	2.19 V	80.0mV	2.11 V	1.93 V	1.98 V	115mVs	63.2mVs
CH2	3.04 V	760mV	2.28 V	2.55 V	1.47 V	1.08 V	1.91 V	2.45 V	114mVs	14.1mVs
	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	30.9 Hz	32.30ms	58.20 %	41.79 %	18.80ms	13.50ms	2.60kV/s	*****	53.15 %	*****
CH2	149 Hz	6.700ms	52.24 %	47.76 %	3.500ms	3.200ms	822 V/s	-596 V/s	44.96 %	66.23 %
	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP	
CH1	650.0us	*****	195.0us	-1.605ms	201mV^2	1.98 V	1.14 V	291mV	2.01 V	
CH2	1.050ms	1.450ms	-520.1us	-1.495ms	195mV^2	2.45 V	2.01 V	1.58 V	2.17 V	

```
Freq=149 Hz   Period=6.700ms   tVmax=-520.1us
1  1.00 V   2  1.00 V   3  5.00 V   4  1.00 V
```

Figure 4.68: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with consistent “EEEE” subvocal signal input after initial “spike”. Wavelet transformed output of microcontroller also shown (purple).

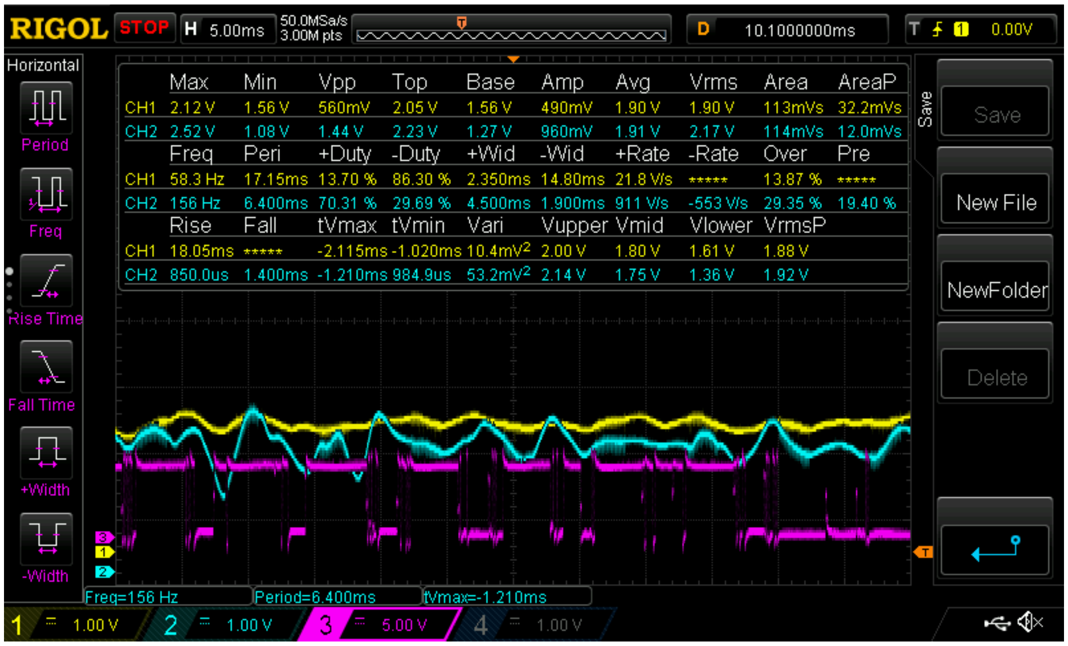


	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	2.36 V	1.20 V	1.16 V	2.17 V	1.20 V	970mV	1.88 V	1.89 V	112mVs	31.6mVs
CH2	2.88 V	640mV	2.24 V	2.81 V	1.13 V	1.68 V	1.85 V	2.15 V	111mVs	17.4mVs

	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	58.6 Hz	17.05ms	79.47 %	20.53 %	13.55ms	3.500ms	707 V/s	*****	19.26 %	*****
CH2	106 Hz	9.400ms	21.28 %	78.72 %	2.000ms	7.400ms	863 V/s	-55.9 V/s	4.387 %	29.62 %

	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP
CH1	1.100ms	*****	-584.9us	979.9us	29.6mV^2	2.07 V	1.69 V	1.30 V	1.87 V
CH2	1.550ms	23.90ms	1.135ms	-589.9us	185mV^2	2.64 V	1.97 V	1.30 V	1.87 V

Figure 4.69: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with consistent “IIIII” subvocal signal input after initial “spike”. Wavelet transformed output of microcontroller also shown (purple).

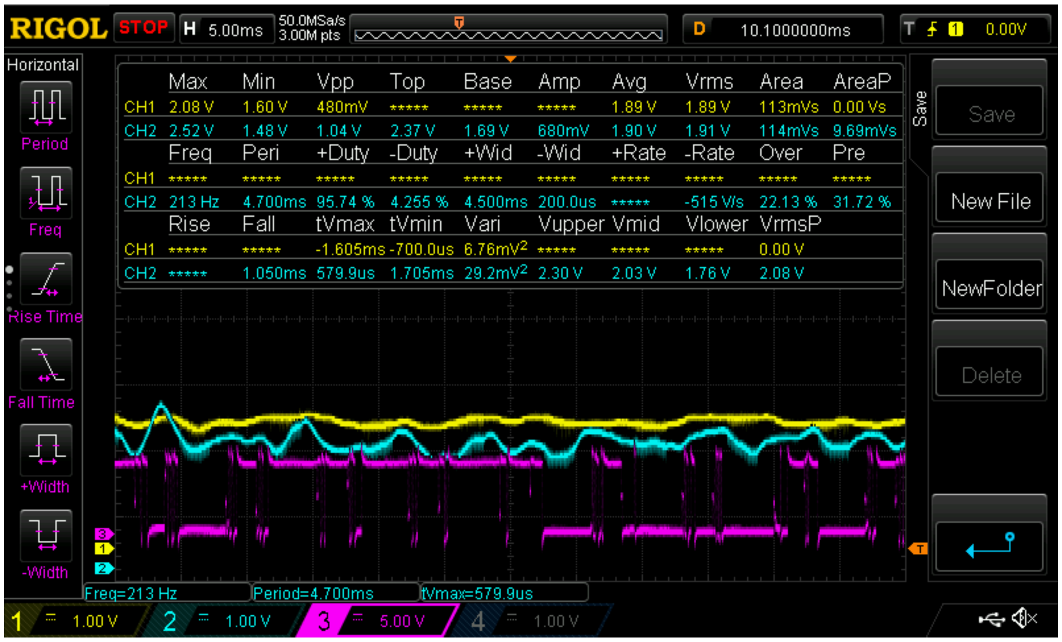


	Max	Min	Vpp	Top	Base	Amp	Avg	Vrms	Area	AreaP
CH1	2.12 V	1.56 V	560mV	2.05 V	1.56 V	490mV	1.90 V	1.90 V	113mVs	32.2mVs
CH2	2.52 V	1.08 V	1.44 V	2.23 V	1.27 V	960mV	1.91 V	2.17 V	114mVs	12.0mVs

	Freq	Peri	+Duty	-Duty	+Wid	-Wid	+Rate	-Rate	Over	Pre
CH1	58.3 Hz	17.15ms	13.70 %	86.30 %	2.350ms	14.80ms	21.8 V/s	*****	13.87 %	*****
CH2	156 Hz	6.400ms	70.31 %	29.69 %	4.500ms	1.900ms	911 V/s	-553 V/s	29.35 %	19.40 %

	Rise	Fall	tVmax	tVmin	Vari	Vupper	Vmid	Vlower	VrmsP
CH1	18.05ms	*****	-2.115ms	-1.020ms	10.4mV^2	2.00 V	1.80 V	1.61 V	1.88 V
CH2	850.0us	1.400ms	-1.210ms	984.9us	53.2mV^2	2.14 V	1.75 V	1.36 V	1.92 V

Figure 4.70: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with consistent “OOOO” subvocal signal input after initial “spike”. Wavelet transformed output of microcontroller also shown (purple).



```
     Max     Min      Vpp      Top      Base     Amp      Avg     Vrms     Area    AreaP
CH1  2.08 V  1.60 V   480mV    *****    *****    *****    1.89 V  1.89 V   113mVs  0.00 Vs
CH2  2.52 V  1.48 V   1.04 V   2.37 V   1.69 V   680mV    1.90 V  1.91 V   114mVs  9.69mVs
     Freq    Peri     +Duty    -Duty    +Wid     -Wid     +Rate   -Rate    Over    Pre
CH1  *****   *****    *****    *****    *****    *****    *****   *****    *****   *****
CH2  213 Hz  4.700ms  95.74 %  4.255 %  4.500ms  200.0us  *****   -515 V/s 22.13 % 31.72 %
     Rise    Fall     tVmax    tVmin    Vari     Vupper   Vmid    Vlower   VrmsP
CH1  *****   *****    -1.605ms -700.0us 6.76mV²  *****    *****   *****    0.00 V
CH2  *****   1.050ms  579.9us  1.705ms  29.2mV²  2.30 V   2.03 V  1.76 V   2.08 V

Freq=213 Hz    Period=4.700ms    tVmax=579.9us
```

Figure 4.71: Final outputs of left (yellow) and right (blue) amplifiers/filters when receiving inputs from surface electrodes with consistent “UUUU” subvocal signal input after initial “spike”. Wavelet transformed output of microcontroller also shown (purple).

Despite the success of finally finding subvocal signals that can at least somewhat be classified, these results still leave some questions when compared to expected results and other projects similar to this project. For example, since subvocal EMG signals tend to be within the voltage range from 0 to 600 µV in amplitude [33], it was expected that the gain would have to be within the range of about 1000 to 5000 to obtain a signal within the 0 V to 5 V rail range, but, instead, it required 18600 gain to obtain them, much larger than expected. Why did the amplifier/filter circuit need a gain of 18600 to obtain said signals instead of 1000 to 5000? This was most likely due to the possibility that the electrodes were different from those used in [33] and were not quite as sensitive as those in [33] despite both being Ag-AgCl surface EMG electrodes (the electrodes used [27] were deliberately cheap in price to not go over the 50$ budget constraint and thus were probably not very sensitive compared to more expensive ones), thus causing the surface electrodes to acquire subvocal signals at much smaller amplitudes than those acquired in [33]. One may also argue that the gain was fluctuating due to the unpredictable behavior mentioned previously at high levels of gain (the gain would be tested before applying electrodes with function generator to make sure it was accurate, but sometimes it still changed during testing with electrodes as testing again after using the surface electrodes would sometimes show a different gain) in which, while the gain was advertised as 18600, it might have lowered during testing to be between 1000 to 5000 and then raised back up to 18600 when applying a test with the function generator afterward, although the possibility of the circuit lowering its gain to only 1000 to 5000 consistently only during testing with surface electrodes seems unlikely in which the unpredictable behavior was most likely not a reason for the large amount gain needed.

Another question to ask would be why the reconstructed output performed better in terms of classification accuracy than that of the wavelet transformed output when it was expected that the wavelet transformed output would perform better than the reconstructed output due to its ability to split the signals into separate frequency components for analysis. This was most likely caused by the fact that the microcontroller had delays between each block of samples taken where it would take the time to process the data and output it, but, as a result, would not be able to sample the input signal when processing and outputting data in which details of the subvocal signal would be missed during these times. It was assumed that, since subvocal signals had tendencies to repeat themselves when subvocalizing a single vowel, word, or noise, occasionally missing data when processing in real time would not be an issue because the repeated segments of the signal could always be sampled again later if missed initially, but the complexity of the subvocal signals showed that missing details while processing data did not give the wavelet transform function enough detail to fully decipher the subvocal signal into individual frequency components, while simply reconstructing the signal into its original form (with some noise sampled out, of course, due to reconstruction) clearly provided more detail than a wavelet transform could, thus leading to a more accurate classification when reconstructing the signal. This also explains why the wavelet transform used with the neural network in the original Matlab program (before removing the wavelet transform coded in Matlab) provided a much more accurate result of an overall accuracy of 77% because the Matlab program preemptively took all samples at a constant sample frequency without delays before training the neural network instead of having delays between each block of samples. An attempt was made to reduce the delay between sampling blocks of samples by simplifying the “do_some_conversions” function by removing all of the functions that made the number rounding involved in the conversion from a floating number type to an integer number type as precise and accurate as possible. While this change did reduce the delay slightly, the outputs of the “do_some_conversions” function were rounded with far less precision, resulting in the outputs of the microcontroller to lose a lot of detail necessary for classification accuracy in which simplifying the “do_some_conversions” function did not make a difference in the classification accuracy (still remained around 45% to 55% for wavelet transformed output and 69% to 71% for reconstructed output) despite the reduction in delay. One could argue that removing the real-time processing ability of the microcontroller program and making the microcontroller record all of its samples before processing them would fix this issue, but not only would this inconvenience the user by making them have to record what they have to subvocalize before being able to see the result but would also require the microcontroller to be able to store a large amount of data which would considerably slow down processing time and likely cause faults and crashes if too much memory would be needed as demonstrated when experimenting with the first attempted wavelet transform program (crashed the microcontroller because of too much memory usage, thus showing that the ability to process in real-time was necessary in which these delays between sampling blocks of samples are inevitable. Schematic of final circuit and pictures of circuit shown below in Figures 4.72 and 4.73. Flowchart for final version of code is shown below in Figure 4.74.

A final question to consider would be as to why the results were not able to achieve 100% classification accuracy. This was most likely the result of many factors such as the microcontroller missing parts of the subvocal signals while processing data, details of the subvocal signals being lost through analog to digital and digital to analog conversion, the inability of the amplifiers/filters to amplify the subvocal signals to be even larger in amplitude to allow the microcontroller to more easily sample the output signals, and the various different kinds of noise clustering the subvocal signals that the filters and the wavelet reconstruction processes could not filter out such as the remaining 120 Hz noise, electromagnetic noise from surrounding electronic equipment that the aluminum foil box could not completely block, electromagnetic noise acquired from EMG electrode leads, and various other noise sources from the human body that the filter could not completely filter out.

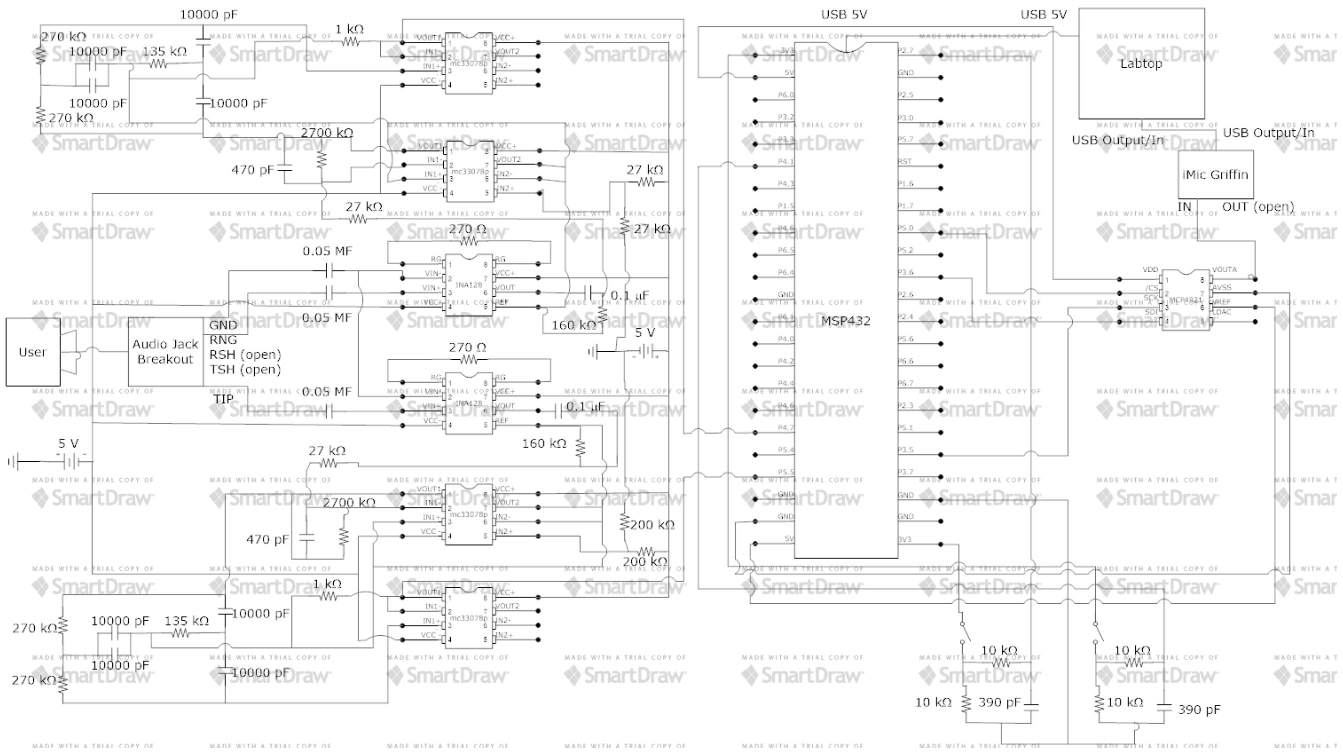


Figure 4.72: Final schematic of project.

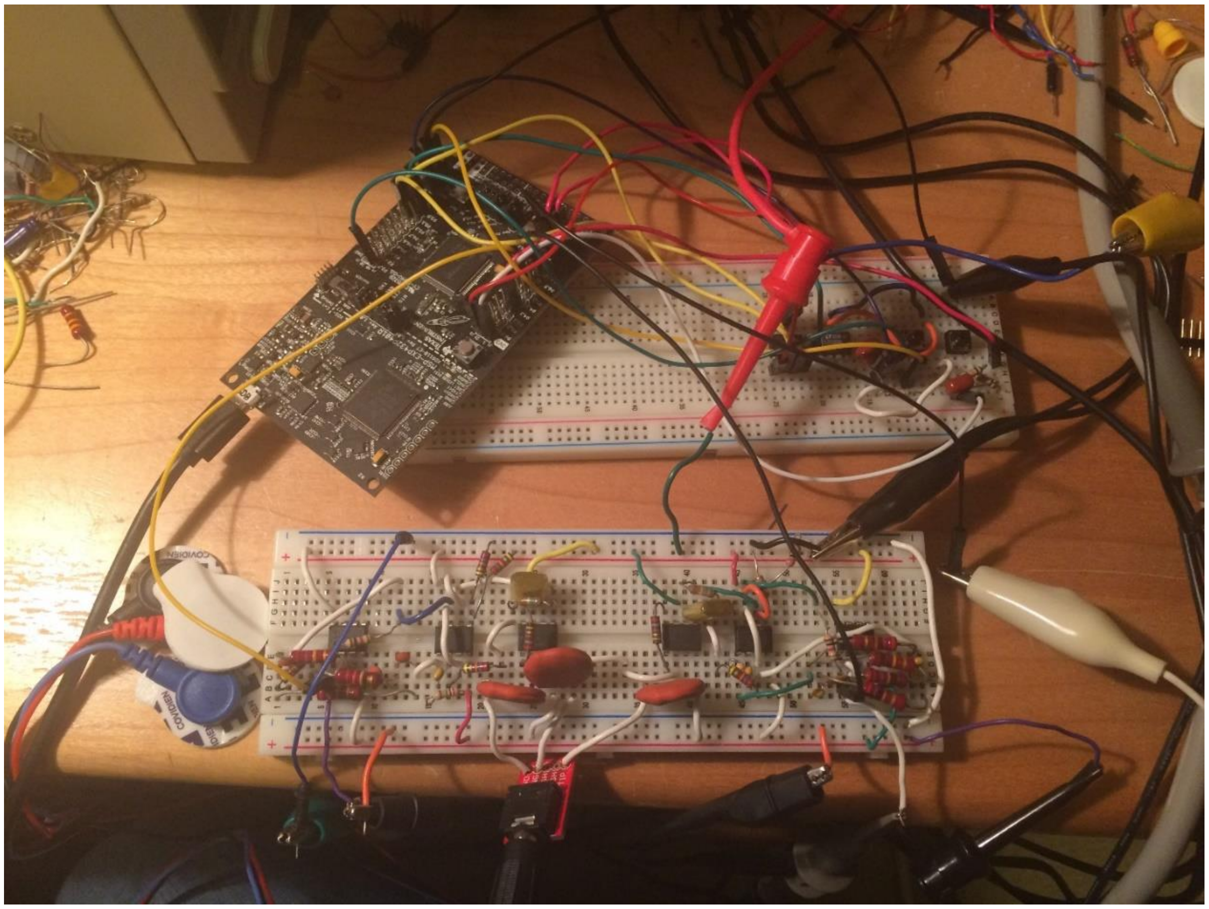


Figure 4.73: Picture of final product.

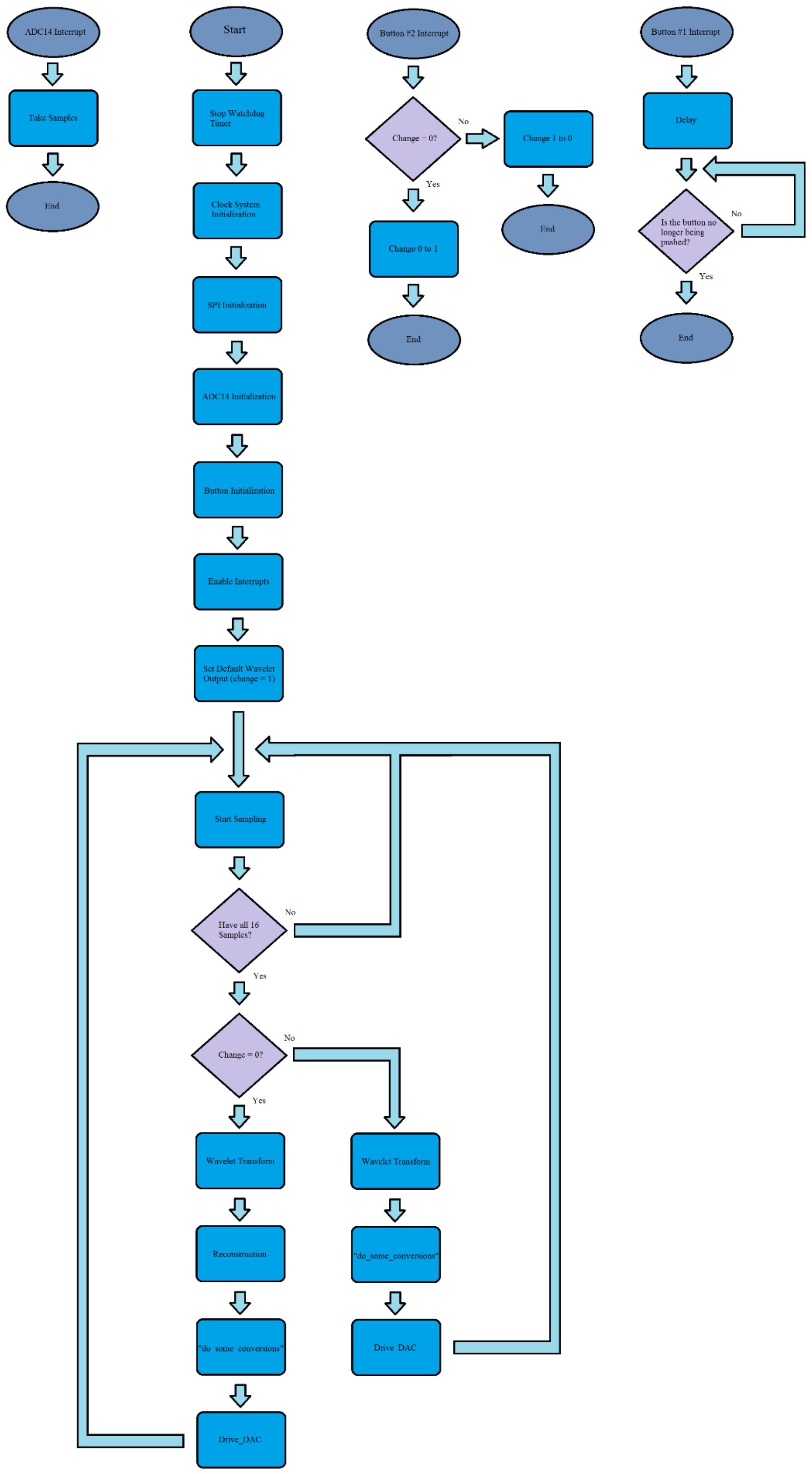


Figure 4.74: Flowchart for final version of code.

5. CONCLUSIONS AND IDEAS FOR FUTURE WORK

This chapter covers a summary of the work accomplished, how well the final product of the project fits the engineering specifications established in chapter 2, how well the project answered the thesis question, and possible future improvements to the project.

5.1 Summary

After being redesigned from a device made to translate subvocal speech into actual speech to a device made to translate subvocal speech into usable data valuable for the analysis of subvocal speech due to scope constraints, a prototype of the Silent Communication Device was built, consisting of the following components:

1. EMG electrodes that acquire subvocal signals from the user's throat, two amplifiers/filters that use INA128 instrumentation amplifiers and mc33078p op-amps to amplify subvocal signals with 18600 gain and filter them with a 3 dB bandwidth of 100 Hz to 500 Hz (low-pass at 125.42 Hz and high-pass at 10 Hz) and a 60 Hz notch filter.
2. A MSP433P401R microcontroller that wavelet transforms the amplified and filtered signals into either individual frequency coefficients (wavelet coefficients) or wavelet transforms and reconstructs said signals into denoised subvocal signals (noise removed due to the constant low-passing and high-passing of the wavelet transform).
3. A MCP4921 DAC that converts the output from the microcontroller from digital to analog in 12-bit resolution (14-bit resolution would have been preferred, but

the LTC1658 DAC provided too small of a swing voltage for the output in which the signal-to-noise ratio was too small).

In order to test the quality of its output, a “patternnet” type neural network was created in Matlab in which the output from the Silent Communication Device has six features extracted, consisting of the mean absolute deviation of the signal, the root mean square of the signal, the variance of the signal, the standard deviation of the signal, the number of zero-crossings of the signal (number of times the signal reaches zero), and finally the number of slope changes of the signal, which are used as inputs for the neural network. The neural network itself consists of an input layer (six neurons for six inputs), a hidden layer (20 neurons), and the outer layer (5 neurons for each “vowel” output) with a hyperbolic tangent function for the hidden neurons’ activation functions, a linear function for the output layer neurons’ activation functions, and the Levenberg-Marquardt algorithm training algorithm for the training of the neural network. 70% of the data received by the neural network is used for training the neural network, 15% is used to test the neural network, and the final 15% is used to validate the results, resulting in a confusion matrix overall accuracy percentage used to measure the classification accuracy resulting from the output of the Silent Communication Device. The user interface of the device consists of two buttons, one which allows the user to choose how the microcontroller processes the input (reconstruction or wavelet transform) and one which allows the user to control when the device takes its input and when it does not. Although the device has proven that it can operate while being powered by portable batteries, it can only operate for about ten seconds at most before the batteries shut off from a lack of consistent current draw in which more resources and a higher budget are required for the creation of custom batteries for the device. As a result, the Silent Communcation Device is powered by a large power supply providing two 5 V rails, while the microcontroller is powered by a laptop.

The final output (most accurate result) of the Silent Communication Device was taken from the average overall accuracies of confusion matrices resulting from 5 tests, each test being the user subvocalizing a vowel for twenty one second samples (twenty samples with each sample being one second long), resulting in 100 samples total (twenty samples for each of the five vowels). Only 3 to 5 tests were taken for one user for each test configuration due to a decrease in accuracy after 3 to 5 tests as a result of sweat accumulating under the EMG electrodes due to the strain of subvocalization, resulting in EMG electrodes having to be replaced after 3 to 5 tests. Since EMG electrodes are expensive and are one-time use, only 3 to 5 tests for one user could be spared for each test configuration. The test configuration that produced the final output (most accurate result) was a result of using the electrode configuration in which the positive electrodes were placed on both the left and right mylohyoid muscles with the negative electrode attached on the left side of the larynx (over vagus nerve muscle), using the smallest diameter surface electrode type [27], and subvocalizing while reading vowels off of a paper at 18600 gain. The user also had to ground themselves to the amplifiers/filters' ground as well to keep noise under control. This produced a consistent overall accuracy between 69% to 71% (average 70%) for reconstructed outputs and around 45% to 55% for wavelet transformed outputs for 3 to 5 tests until the EMG electrodes needed to be replaced, just barely reaching the minimal 70% overall accuracy prediction.

However, despite the success of being able to obtain and process subvocal speech for analysis, the Silent Communication Device does not come without problems. For example, as previously mentioned, after about 3 to 5 tests, the EMG electrodes have to be replaced due to sweat building under the electrodes, resulting in the expense of the EMG electrodes having to be constantly replaced. The user also has to exert a large amount of effort to successfully subvocalize a large enough subvocal signal for the microcontroller to sample, resulting in the possibility of headaches for the user. The amplifiers/filters are extremely suspectible to external noise and behave erratically at the required 18600 gain, often changing its gain almost randomly and increasing in noise over time, which is reset only by moving wires of the amplifier/filters with a non-conductive object or removing the user from ground for a moment only to reattach the user to ground again. Also, as previously mentioned, the device requires custom batteries to operate without a large power supply as more portable batteries have a tendency to shut off after about ten seconds of applying power due to not having a large enough current draw from the device. These problems cannot currently be solved with the current budget and resource constraints, but can hopefully be solved in future works.

5.2 Requirements and Specifications Fulfillment

How well did the final product fit engineering specifications? The first engineering specification states that the final product must cost the customer at most $50. Appendix B shows the total expenses as $221.36, but this is not an accurate representation of how much the final product costs a customer as many of the expenses were not at mass production levels (> 1 ku prices) or were for components that were only needed for testing and experimentation and were not included in the final product. When calculating the total price when assuming mass production levels (assuming > 1 ku prices); counting only the components appearing in the final schematic (Figure 4.72) with the exception of the iMic griffin (only used for interfacing with the laptop), the laptop, and the power supply; assuming non-IEEE office prices for capacitors (one can get these much cheaper by ordering them, assuming 0.16 each based on 10000 pF capacitor prices); not counting the prices of both breadboards (if the final product is mass manufactured, they would not be built on a breadboard, but on a cheap integrated circuit board instead); and assuming only three electrodes provided (biomedical sensor pads), the total price comes out as $48.39, just under $50! Of course, this is an underestimation as it does not included the price of cheap integrated circuitry and the price of the custom made batteries designed for the circuit's load with USB outputs so, assuming the cost of said components is larger than $1.61, the final product unfortunately goes over the $50 specification, but the total amount at which it goes over cannot be determined with the current resources for the project, while the known amount still remains under $50. Decreasing the price of the current components could be possible by buying cheaper alternative instrumentation amplifiers and op-amps, but this would certainly affect the performance as well, most likely dropping the classification accuracy below 70% in which it becomes apparent that the current form of the project is the cheapest form available for 70% classification accuracy. Thus, within the prices that are possible to be known with given resources, the project fulfills this requirement.

The second engineering specification states that the device's dimensions must not exceed 3" x 4" x 0.5" for the purpose of making it pocket sized. While the MSP432 microcontroller, external DAC, and button debouncing circuit certainly fit within this dimension (assuming stacking them on top of each other), the amplifier/filter circuit unfortunately does not fit within this dimension as a result of the fact that the circuit components had to be spaced out from each other to prevent magnetic noise from causing noise feedback issues. With better isolation and manufactured integrated circuitry, the amplifier/filter circuit could certainly fit within the dimension requirements as the circuit was originally built within this dimension, but had to have its components spaced after noise feedback issues occurred. Thus, within the dimensions that still allow the project to operate properly that are possible with given resources, the project cannot fulfill this requirement as it is too large as a prototype breadboard circuit and making it smaller prevents the circuit from performing properly.

The third engineering specification states the device must prevent unintended recipients at least 10 feet away from the user from hearing at most 25 dB of the user’s subvocal speech for the purpose of assuring that the device is silent while operating. 25 dB is the sound volume of a quiet conversation (e.g. whispering). The final product was able to obtain, amplify, filter, and wavelet transform or reconstruct a subvocal signal without the user having to make any noticeable, audible noise. The user’s mouth can remain completely shut in which the user only has to move throat and tongue muscles to provide a subvocal signal that the device can pick up, making practically inaudible noise. The loudest action that the user has to take when using the device is the subtle movements of the tongue inside the user’s mouth when subvocalizing, which should not require audio measurements to prove that these movements are quieter than whispering. Thus, the project fulfills this requirement.

The fourth engineering specification states that the device's circuitry must be powered by a battery able to output 5 V for 24 hours in order to ensure that the device can operate for extended periods of time. While the device has certainly proven that it can operate on 5 V (positive and negative 5 V rails), the previously mentioned (section 4.8) testing with batteries showed that the device required three batteries to provide each rail and power the microcontroller as opposed to one battery because attaching grounds between rails on a single battery with multiple outputs caused the said battery to refuse to provide any power. In addition, batteries used had a tendency to stop providing power after about 10 seconds of operation unless restarted (repress "on" button) due to the fact that the device could not provide a large enough load to get a consistent current draw from each battery despite adding loads to the batteries that were proven to give steady current draw, showing that custom batteries needed to be designed for the device with loads that allowed for the batteries to not shut off after 10 seconds. Unfortunately, designing custom batteries was outside of the resources provided and budget constraint for making the device, but performance while being powered by batteries showed not only proper operation, but removal of noise as well. Therefore, the project showed that it could be powered by batteries outputting 5 V, but could not show whether or not it could sustain this for 24 hours due to problems involving the need for a specific design of battery that was beyond the resources and budget constraints of the project. Thus, the project could only partially fulfill this requirement.

The fifth engineering specification states that the device must be able to output to any audio jack with industry standard dimensions. This engineering specification was to make sure that the device can output to commonly used devices such cell phones or laptops through audio jacks. Using a simple stereo cord with a 2.5 mm diameter audio jack output with two alligator clips attached to the one of the stereo plugs for both the input (taken from clipping alligator clip to wire attached to the external DAC's output) and ground (GND of microcontroller), this specification was easy to fulfill as the stereo audio jack could easily plug into any laptop or cell phone, while the iMic Griffin also allowed the device to output into a USB port as well. Thus, the project fulfills this requirement.

The sixth engineering specification states the device is non-invasive (no part of the device is required to enter the human body for proper operation) for the purpose of increasing the appeal and convenience of the device by not having to puncture or open the body of the user (e.g. sticking a needle into the body) just to get a proper input for the device. This specification was fulfilled by using surface EMG electrodes instead of using more invasive means such as EMG needles to obtain subvocal signals, which obtain signals by simply being attached to the skin over a muscle that the user wishes to record signals from as opposed to penetrating through the skin directly into the muscle to record signals. Thus, the project fulfills this requirement.

The seventh engineering specification states that the device must be able to provide a gain of at least 10 to a maximum of 18700 to amplify subvocal input signals in order to ensure that the device has enough gain to amplify subvocal signals to amplitudes that the microcontroller can accurately sample. The range of gain was set by [3] and [23] as [3] had managed (somehow) to obtain subvocal signals with only about 10 gain (12.127 on calculation) which was the lowest recorded gain seen out of all the papers referenced that managed to amplify a subvocal signal to be large enough to get classification accuracy above 70%, while [23] acquired subvocal signals at 18700 gain in a 0 V to 100 V voltage range which was the largest recorded gain seen out of all the papers referenced that managed to also get classification accuracy above 70%. The source, [33], demonstrated subvocal signals to be about 0 V to 600 μV in amplitude, leading to a theory that the gain of the amplifier/filter circuit would be somewhere in the range of 1000 to 5000 gain, but turned out to be 18600 gain that finally provided ample amplification to have subvocal signals become large enough in amplitude for the microcontroller to successfully sample. The amplifier/filter circuit was also proven to be able to provide gain between the 10 to 18700 range, but had difficulty maintaining gains at 15000 or larger due to isolation problems that could not be fixed within the resources and budget constraints of the project. Thus, despite the difficulty of maintaining higher levels of gain, the device is able to provide enough gain within the 10 to 18700 range to amplify subvocal signals to amplitudes that the microcontroller can successfully sample and therefore fulfills the requirement.

The eighth engineering specification states the "acoustic sensor" (general term used for whatever would be presumed to be making contact with the user during the time of creating the engineering specifications, later discovered to be the surface EMG electrodes) does not cause electrical shocks or physical injury to the user. When originally designing the device in concept, it was not clear what kind of equipment would be sampling the subvocal signal from the user. The surface EMG electrodes were used for their non-invasive means of sampling EMG signals from muscles underneath the skin. Since surface electrodes do not require power to operate and do not expose any electrically conducted metal to the user, these electrodes cannot cause electrical shock to the user. However, since the device requires a lot of effort to subvocalize hard enough to get adequate subvocal signals for the microcontroller to sample, the danger of giving the user a headache or even a migraine is possible if the user strains themselves too hard, but it is important to note that acquiring data for the particular neural network used required the user to sustain a single vowel for up to twenty seconds in which tests for analyzing subvocal speech could easily be designed to be less harsh on the user in the future (e.g. recording single words or phrases instead of sustaining long signals such as “aaaaaaa”). Thus, the project fulfills this requirement.

The ninth and final engineering specification states the user control for the device must have a range of volume from 0 dB to 85 dB for the output (user’s voice). Since the device was originally designed to be able to not only acquire, amplify, filter, and wavelet transform or reconstruct a subvocal signal for analysis, but also classify any subvocal signal to its intended speech and output said signal as an actual audio signal of human speech until the scope of the project was redefined to just analyzing subvocal signals as detailed in section 3.4, this engineering specification was designed to ensure that the device could give an audible range of sound for its output when communicating through, say, a cell phone or a laptop, but now that the device has had its scope and purpose redefined, this engineering specification no longer applies to the device. Thus, the project does not fulfill this requirement and has no reason to.

In conclusion, based upon the constraints placed upon the project and within the resources available for the creation and design of the project, the project was able to successfully produce a small, inexpensive prototype device that can obtain, filter, amplify, and wavelet transform or reconstruct subvocal signals for the purpose of analyzing subvocal signals.

5.3 Thesis Statement Fulfillment

After finally gaining the ability to acquire and classify subvocal signals as well as proving to fit the engineering specifications to the best of its ability, the project must be able to provide an answer to the question proposed in section 1.1. Using wavelet analysis and a MSP432 microcontroller, can subvocal signals originating from the throat be classified to an overall accuracy of at least 70% with a project budget of $50? The 70% overall accuracy requirement was based on the results of similar experiments using wavelet analysis and classified using neural networks such as [23] and [33] which both obtained classification accuracies between 70% to 80% for their results, but obtained these results using much more expensive and professional equipment than that of the project. Taking advantage of more recent technologies such as the MSP432 microcontroller which provides high processing power and 14-bit resolution (could only use 12-bit in the end) for only about $13, it was theorized that this same level of performance could be achieved with much less expensive equipment through using a prototype breadboard circuit and a MSP342 microcontroller with results verified by a neural network coded in Matlab. Fortunately, the project was barely able to achieve this goal, reaching a consistent classification accuracy of 69% to 71% (average of 70%, until surface electrodes need to be replaced) under a known budget of $48.39 (without batteries, that of which the price could not be determined and assuming mass production prices) by using a prototype breadboard circuit to filter subvocal signals representing five vowels within a 10 Hz to 450 Hz frequency range (peaked between 100 Hz and 500 Hz), amplify them using 18600 gain, wavelet transform and then reconstruct them using the MSP432 microcontroller to remove even more noise for wavelet analysis, and then classify them using a neural network coded in Matlab as a proof of concept. Within the context of the project being a prototype circuit (not for mass production), the budget could be reduced even further as the mc33078p and MCP4921 chips can be obtained for free if ordered as samples, lowering the budget to $41.3, which leaves more room for the price of cheap, custom made batteries within the constraint of $50. Therefore, yes, when discounting the price of custom batteries and assuming mass production prices or prototype production prices, using wavelet analysis and a MSP432 microcontroller, subvocal signals originating from the throat can be classified to an overall accuracy of at least 70% within a project budget of $50.

5.4 Future Works

Future improvements for the project, when no longer restricted by time and budget constraints, include buying and implementing a 14-bit DAC that does not have problems involving voltage swing in order to allow the MSP432 microcontroller the ability to sample and output subvocal signal data in 14-bit resolution for better detail and accuracy, improving the isolation to make the circuit behavior more predictably and remove more noise, and increasing the range of classification from just vowels to words and every letter of the English language as well in order to give the device practical uses such as replacing typing with a keyboard with subvocal speech. In addition, adding a second processor or more to the microcontroller, although no such variant for the

MSP432 currently exists, might allow the microcontroller to be able to have the processing power to handle the original lattice filter structure wavelet transform, allowing any type of wavelet coefficient to be used and thus possibly increase the overall accuracy of the output. The increased processing power would allow the addition of the neural network to the microcontroller code to increase functionality as well. Thus, with enough time and money, the project may be improved upon enough to reach its original design of being able not only acquire, amplify, filter, and wavelet transform or reconstruct a subvocal signal for analysis, but also classify any subvocal signal to its intended speech and output said signal as an actual audio signal of human speech or, in other words, create an actual silent communication device.

REFERENCES:

[1] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, McGraw-Hill, 2007, p. 37

[2] *IEEE Std 1233, 1998 Edition*, p. 4 (10/36), DOI: 10.1109/IEEESTD.1998.88826

[3] J.A.G. Mendes, *"*Subvocal Speech Recognition Based on EMG Signal Using Independent Component Analysis and Neural Network MLP," *IEEE Xplore Digital Library*, 2008. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4566152. [Accessed: October 13, 2014] doi: [10.1109/CISP.2008.741]

[4] Analog Devices, "Precision, Low Power, Micropower Dual Operational Amplifier," OP290 datasheet, 2009. Available: http://www.analog.com/static/imported-files/data_sheets/OP290.pdf [Accessed: October 18, 2014]

[5] C. H. Lin, "Signal processing and fabrication of a biomimetic tactile sensor array with thermal, force and microvibration modalities," *IEEE Xplore Digital Library*, 2009. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5420611&queryText%3Dmicrovibration. [Accessed: October 19, 2014] doi: [10.1109/ROBIO.2009.5420611]

[6] C. Parnin, "Subvocalization - Toward Hearing the Inner Thoughts of Developers," *IEEE Xplore Digital Library*, 2011. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=5970156&queryText%3Dsubvocalization. [Accessed: October 18, 2014] doi: [10.1109/ICPC.2011.49]

[7] D. Self, "Opamps and their properties," in *Small Signal Audio Design 2e*. 2nd Ed. CRC Press, 2014, pp. 119-167.

[8] T. Furukawa, "Imaging Cells through a MultiPhoton Process," in *Biological Imaging and Sensing*. Ed. Springer Science & Business Media, 2004, pp.15-66.

[9] S. J. Scalise, A. S. Rainone, D. W. Davis, "Auscultation augmentation device," U.S. Patent 5 812 678, Sep 22, 1998.

[10] H. Sheikhzadeh, "Waveform-based speech recognition using hidden filter models: parameter selection and sensitivity to power normalization," *IEEE Xplore Digital Library*, 1994. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=260337&queryText%3Dspeech. [Accessed: October 19, 2014] doi: [10.1109/89.260337]

[11] N. Komatsu, M. Aota, S. Komatsuda, T. Toshihiro, Y. Matsumoto, “Speech detection circuit,” U.S. Patent 5 371 800, Dec 6, 1994.

[12] R. F. Lyon, “Cost, power, and parallelism in speech signal processing,” *IEEE Xplore Digital Library*, 1993. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?tp=&arnumber=590718&queryText%3Dspeech+Signal+Processing. [Accessed: October 20, 2014] doi: [10.1109/CICC.1993.590718]

[13] J. Bluck, M. Braukus, “NASA Develops System To Computerize Silent, "Subvocal Speech”,” *NASA News*, 2004. [Online]. http://www.nasa.gov/home/hqnews/2004/mar/HQ_04093_subvocal_speech.html. [Accessed: November 9, 2014]

[14] J. B. Smelcer, “Interruption, Distractions, and Technology,” *Fairfieldprofessionals.com*, 2012. [Online]. Available: http://fairfieldprofessionals.com/insights/145-interruptions-distractions-and-technology. [Access: November 9, 2014]

[15] CERN – European Organization for Nuclear Research, “The Physical Work Environment,” *safetyguide.web*, 2004. [Online]. Available: http://safetyguide.web.cern.ch/SafetyGuide/Part3/25.2.html. [Accessed: November 13, 2014]

[16] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, McGraw-Hill, 2007, p. 205

[17] T. T. Ahonen, A. Moore, “The Annual Mobile Industry Numbers and Stats Blog - Yep, this year we will hit the Mobile Moment,” *Communities Dominate Brands*, 2013. [Online]. Available:

http://communities-dominate.blogs.com/brands/2013/03/the-annual-mobile-industry-numbers-and-stats-blog-yep-this-year-we-will-hit-the-mobile-moment.html. [Accessed: November 13, 2014]

[18] H. Lewis, “Why Recycle My Mobile Device?” *Earthworksaction.org*, para. 6, 2012. [Online]. Available: http://rmcp.earthworksaction.org/why_recycle#.VGaM2MnNE4t. [Accessed: November 13, 2014]

[19] R. Ford and C. Coulston, *Design for Electrical and Computer Engineers*, McGraw-Hill, 2007, p. 216

[20] T. C. Mazor, “Information Technology,” *Santa Clara University*, 2014. [Online]. Available: http://www.scu.edu/ethics/publications/iie/v6n1/homepage.html. [Accessed: November 13, 2014]

[21] T. Schick Jr. & L. Vaughn, *Doing Philosophy,* McGraw Hill, 2003, p. 354-35

[22] I.R. Titze, *Principles of Voice Production*, Prentice Hall (currently published by NCVS.org), 1994, p. 188

[23] L. E. Mendoza, J. P. Rodríguez, J. L. R. Valencia, “Electro-myographic patterns of sub-vocal
Speech: Records and classification,” *uelbosque.edu*, November 29, 2013. [Online]. Available:
http://www.uelbosque.edu.co/sites/default/files/publicaciones/revistas/revista_tecnologia/volumen12_numero2/3Articulo_Rev-Tec-Num-2.pdf. [Accessed: November 13, 2014]

[24] “What is the Standard Headphone Jack Size?” *Reference*, 2017. [Online]. Available: https://www.reference.com/technology/standard-headphone-jack-size-f9f4b241138f2f50 [Accessed: March 25, 2017]

[25] R. H. Chowdhury, M. B. I. Reaz, M. A. B. M. Ali, A. A. A. Bakar, K. Chellappan, and T. G. Chang, “Surface Electromyography Signal Processing and Classification Techniques,” *Sensors*, 2013. [Online]. Available: www.mdpi.com/journal/sensors [Accessed: January 15, 2015].

[26] Van Boxtel, A. (2001). Optimal signal bandwidth for the recording of surface EMG activity of facial, jaw, oral, and neck muscles. Psychophysiology 38, 22–34.

[27] Covidien, “Arbo™ H124SG Ref. Code: 31.1245.21,” Kendall™ ECG Electrodes Product Data Sheet, 2008. Available:
https://cdn.sparkfun.com/datasheets/Sensors/Biometric/H124SG.pdf [Accessed: January 15. 2015]

[28] Texas Instruments, “INA128, INA129,” INA12x Precision, Low Power Instrumentation Amplifiers, 2015. Available:
http://www.ti.com/lit/ds/symlink/ina128.pdf [Accessed: January 15, 2015]

[29] Texas Instruments, “MC33078,” Dual High-Speed Low-Noise Operational Amplifier, 2006. Available: http://www.ti.com/lit/ds/symlink/mc33078.pdf [Accessed: January 15, 2015]

[30] T. Edwards, “Discrete Wavelet Transforms: Theory and Implementation,” *Stanford University*, 1992. [No longer available online].

[31] R. Stojanović, S. Knežević, “MSP430 Implementation of Wavelet Transform for Purposes of Physiological Signals Processing,” *University of Montenegro, Faculty of Electrical Engineering, Montenegro*, 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6532238/?reload=true&arnumber=6532238 [Accessed: January 15, 2015]

[32] Texas Instruments, “MSP432P4xx Family,” Technical Reference Manual, 2016. Available: http://www.ti.com/lit/ug/slau356d/slau356d.pdf [Accessed: May 15, 2015]

[33] M. Jahan, M. Khan, “Sub-vocal Phoneme-Based EMG Pattern Recognition and its application in Diagnosis,” *IEEE Xplore Digital Library*, 2015. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7443564 [Accessed: May 28, 2016]

[34] Y. Bandi, R. Sangani, A. Shah, A. Pandey, and A. Varia, “Subvocal Speech Recognition System based on EMG Signals,” *IEEE Xplore Digital Library*, 2015. [Online]. Available: http://research.ijcaonline.org/icct2015/number7/icct201592.pdf [Accessed: May 28, 2016]

[35] Md. R. Ahsan, Muhammad I. Ibrahimy, Othman O. Khalifa, “The use of Artificial Neural Network in the Classification of EMG Signals,” *IEEE Xplore Digital Library*, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6305853/ [Accessed: May 14, 2016]

[36] B. Champaty, B. K. Biswal, K. Pal, and D. N. Tibarewala, “Random Forests Based Sub-Vocal Electromyogram Signal Acquisition and Classification for Rehabilitative Applications,” *IEEE Xplore Digital Library*, 2014. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6808012 [Accessed: May 28, 2016]

[37] M. Khan and M. Jahan, “The Application of AR Coefficients and Burg Method in Sub-vocal EMG Pattern Recognition,” *IEEE Xplore Digital Library*, 2015. [Online]. Available: http://www.krishisanskriti.org/vol_image/07Jul201502073653%20%20%20%20%20%20%20%20Mosarrat%20Jahan___________ap%20%20%20%20%20%20%20%20%20%20%20%20%20%20813-815.pdf [Accessed: May 28, 2016]

[38] “How the capacitor works (in a debouncing circuit),” *Stack Exchange*, 2013. [Online]. Available: http://electronics.stackexchange.com/questions/66764/how-the-capacitor-works-in-a-debouncing-circuit [Accessed February 20, 2016]

[39] Microchip, "MCP4921," 12-bit DAC with SPI^{TM} Interface, 2004. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/21897a.pdf [Accessed: June 15, 2016]

[40] A. G. Kumar, "Multi-Frequency Range and Tunable DCO on MSP432P4xx Microcontrollers," *Texas Instruments*, 2016. [Online]. Available: http://www.ti.com/lit/an/slaa658a/slaa658a.pdf [Accessed: July 24, 2016]

[41] Fduignan, "Speedier DACs with the MSP432 Launchpad," *IOPROG*, July 2, 2016. [Online]. Available: https://ioprog.com/2016/07/02/speedier-dacs-with-the-msp432-launchpad/ [Accessed: August 1, 2016]

[42] SainSmart, "SainSmart DSO NOTE II 2016 NEW Update Version DS202 Nano ARM Portable Mini Handheld Touch Screen Digital Storage Oscilloscope, 8MB Memory Storage 2MHz 10Mps," *SainSmart*, 2016. [Online]. Available: http://www.sainsmart.com/sainsmart-dso-note-ii-oscilloscope.html [Accessed: August 3, 2016]

[43] Linear Technology, "LTC1658," *14-bit Rail-to-Rail Micropower DAC in MSOP*, 1998. [Online]. Available: http://cds.linear.com/docs/en/datasheet/1658f.pdf [Accessed: August 15, 2016]

[44] Drake, Richard L.; Vogl, Wayne; Tibbitts, Adam W.M. Mitchell, Gray's anatomy for students, Philadelphia: Elsevier/Churchill Livingstone, 2005, p. 987.

[45] Subasi, Abdulhamit; Yilmaz, Mustafa; Ozcalik, Hasan R., "Classification of EMG signals using wavelet neural network," *Journal of Neuroscience Methods*, 2006. [Online]. Available: https://pdfs.semanticscholar.org/a4c3/d9bb6c1ad98a5557e2262480f758931a3a64.pdf [Accessed: January 29, 2017]

APPENDICES

Appendix A: Code Composer Studio (Microcontroller) and Matlab Code

```
//******************************************************************************
*
//
// MSP432 main.c template - Empty main
//
//******************************************************************************

#include "msp.h"
#include <math.h>

/*
 * main.c
 */

#include <stdio.h>
#define SRSIZE 49 /*size of the decomposition xform shift register */
#define BLKSIZE 16 /* input block size--must be a power of two */
#define NUMFILT 2 /* number of lattice filters required */
#define HALFORD 3 /* one half of the order of the filter */
#define SRSIZEREC 26 /*size of the resonstruction xform shift register */
#define BUTTON BIT1 //button bit P1.1

void waveletdecom(float *decominput); /*declaring decomposition filter */
void recon(float *decomout); /*declaring reconstruction filter*/
void do_some_conversions(float *reconout);
void Drive_DAC(int *b);
//int a; //evenin and oddin counter
//int evenin, oddin; //inputs to decom
float decominput[BLKSIZE]; //input to decomposition filter
float decomout[BLKSIZE]; //output of decomposition filter
float reconout[BLKSIZE]; //output of reconstruction filter
int a; //sample counter
int b[BLKSIZE]; //float to int conversion

 int main(void) {

    WDTCTL = WDTPW | WDTHOLD;    // Stop watchdog timer

    // 16Mhz SMCLK
  /*  if (CALBC1_16MHZ==0xFF) // If calibration constant erased
    {
         while(1); // do not load, trap CPU!!
    } */

    /*set MSP432 settings (preset calibration in MSP432?*/
   /* DCOCTL = 0; // Select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_16MHZ; // Set range
    DCOCTL = CALDCO_16MHZ;  */  // Set DCO step + modulation RUN ON SMCLK FOR
a?????
```

```
    CSCTL0 = DCORSEL_4; //DCOCLK, SMCLK = 12 MHz

    //ADC14 uses MODCLK

    ADC14CTL0 = ADC14SHT1_2 + ADC14ON + ADC14IE15; // ADC14ON, interrupt
enabled, 16 clock samples

    ADC14MCTL0 = ADC14INCH_15; // input A15, P6.0

   // ADC10CTL1 = INCH_1; //+ SSEL_3 + CONSEQ_1 ;  input A1,

    //ADC10AE0 |= 0x02; // PA.1 ADC option select MSP432 doesn't have/need
one?

    // ADC10AE0 |= INCH_1 set P1.1 for ADC input

    // SPI Setup
        // clock inactive state = low,
        // MSB first, 8-bit SPI master,
        // 4-pin active Low STE, synchronous
        //
    UCB0CTL0 |= UCCKPL + UCMSB + UCMST + UCSYNC;

    UCB0CTL1 |= UCSSEL_2; /* SMCLK (input clock?) */

    UCB0BR0 |= 0x00; //no SMCLK division

    UCB0BR1 |= 0x00;

    UCB0CTL1 &= ~UCSWRST;

    /* set inputs and outputs */

  P1DIR |= BIT0; //set LED1 output P1.0
  P2DIR |= BIT2; //set Blue_LED
  P1OUT &= ~BIT0; //turn off LED1
  P2OUT &= ~BIT2; //turn off blue LED
  //  P6DIR |= 0x01; //set P6.0 output
    P1DIR |= BIT6; //set P1.6 output

    TA0CCTL0 = CCIE;                            // CCR0 interrupt enabled
    TA0CCR0 = 12000;
    TA0CTL = TASSEL_2 + MC_2;                 // SMCLK, contmode, time_A0
control

    //enable interrupts///
    P1DIR &= ~BUTTON;                  // button is an input
    P1OUT |= BUTTON;                   // pull-up resistor
    P1REN |= BUTTON;                   // resistor enabled
    P1IES |= BUTTON;                   // interrupt on low-to-high
transition
    P1IE |= BUTTON; // P1.1 interrupt enabled

    P1IFG &= ~BUTTON; // P1.1 IFG cleared
    __enable_interrupt(); // enable all interrupts


// while(1){ }//run continuously
//      while(P1IN == 0x02){ //while button is pressed
/*for (a = 0; a < 16; a++ ) //takes 16 samples for wavelet transform
 {          //runs continuously

            ADC14CTL0 |= ADC14ENC + ADC14SC; //enable taking samples TAKE ONLY
SIXTEEN AT A TIME!!!!!

             //__bis_SR_register(CPUOFF + GIE); // LPM0, ADC10_ISR will force
exit //

              decominput = ADC14MEM15;

              waveletdecom(decominput); // run wavelet transform (more TACCR0
cycles will be added to compensate for this)

              TA0CCR0 += 1300;
              } */
       // }
 //}
   P2OUT |= BIT2; //test blue LED
   P1OUT |= BIT1; //test LED1
    return 0;
 }

 void waveletdecom(float *decominput) {

    /* set LED outputs for each stage of code for testing */

    static const int gamma[HALFORD] = {1, 0, 0}; /*using Haar coefficients to
save memory space*/
    static float beta = 0.707;
    int i, j, k, l;
    float memory[SRSIZE];  //changed to int from double
    float evenin[NUMFILT], oddin[NUMFILT], e[NUMFILT][HALFORD], din[NUMFILT -
1][HALFORD],
        dout[NUMFILT][HALFORD], evenout[NUMFILT], oddout[NUMFILT]; //changed
to int from double

    static int evenintable[NUMFILT][BLKSIZE] = {30, 45, 43, 45, 34, 45, 39,
45, 43, 45, 41, 45, 43, 45, 41, 45,
             48, 31, 48, 31, 48, 30, 48, 27, 48, 30, 48,   28,   48, 42, 48,
48};

    static int oddintable[NUMFILT][BLKSIZE] = {31, 46, 40, 46, 33, 46, 36, 46,
35, 46, 33, 46, 33, 46, 31, 46,
             48, 32, 48, 33, 48, 16, 48,      29, 48, 29,  48, 27, 48,  32,
48,   48 };

    static int evenouttable[NUMFILT][BLKSIZE] = { 22,   48,  48,  48,  16,
35,  12,  34,  48,  33,  6, 32, 48,
             31, 1, 30,  48,  22,  48, 48,  48,  16,  48,  12,  48,  48,  48,
6,  48,  48,  48,  48 };

    static int odddouttable[NUMFILT][BLKSIZE] = { 45, 48, 48, 48, 35, 45, 45,
45, 48, 45, 45, 42, 48, 38, 0, 22,
             48, 41, 48,  48,  48, 33, 48, 36, 48, 48, 48, 34, 48, 48, 48, 48
};

    static int dintable[HALFORD-1][NUMFILT][BLKSIZE] = { 44, 47, 47, 41,37,
      39,    35,    37,    47,    36,    34,    42,    47,
             43,    42,    43,    48,    43,    48,    38,    48,    36,    48,
      38,    48, 44,        48,    40,    48,    44,    48,    48,    43,    47,
      47,    40,    36,    38,
          31, 36,   47,    35,    27,    34,    47,    38,    25,    42,    48,
      42,    48,    37,    48,    35,    48,    35,    48,    33,    48, 33,
      48,    31,    48,    48 };

    static int douttable[HALFORD-1][NUMFILT][BLKSIZE] = { 44, 43, 39, 41, 37,
      39,    39,    38,    45,    44,    41,    45,    45,
             45,    48,    45,    48,    48,    48,    38,    48,    36,    48,
      48,    48,    35,    48,    48,    48,    43,    48,    48,    43,    42,
      38,    40,    36,    38,
             36,    37,    34,    36,    34,    40,    32,    44,    48,    44,
      48,    48,    48,    37,    48,    32,    48,    48,    48,    28,    48,
      48,    48,    26,    48,    48     };

   // printf (" Circular wavelet decomposition program\n");
   // printf ("\n Output wavelet data:\n");

    /* initialize shift register memory and counter */
    for (j = 0; j < SRSIZE; j++) memory[j] = 0.0;
    i = 0;

    /* input placement */

    while (memory[SRSIZE - 3] = decominput[i])/* (scanf ("%f", &memory[SRSIZE
- 3]) <= EOF) */ {  //WHAT CAUSES THIS WHILE LOOP TO END???

    /* loop over all filters */

    for (k = 0; k < NUMFILT; k++) {

       /* resolve all inputs (input schedulers) */

       evenin[k] = memory[evenintable[k][i]];
       oddin[k] = memory[oddintable[k][i]];
       for (l = 0; l < HALFORD - 1; l++)
             din[k][l] = memory[dintable[l][k][i]];

       /* N-stage lattice filters */
       for (l = 0; l < HALFORD; l++) {
             if (l == 0) { /*first rung (normal) */
                   dout[k][0] = evenin[k] + gamma[0] * oddin[k];
                   e[k][0] = evenin[k] * gamma[0] - oddin[k];
             }
```

```
            else if ((l & 1) == 0) {

                  dout[k][l] = e[k][l - 1] + gamma[l] * din[k][l - 1];
                  e[k][l] = e[k][l - 1] * gamma[l] - din[k][l - 1];
            }
            else { /* inverted rung */

                dout[k][l] = e[k][l - 1] * gamma[l] + din[k][l - 1];
                e[k][l] = e[k][l - 1] - gamma[l] * din[k][l - 1];

           }
      }
       evenout[k] = beta * e[k][HALFORD - 1];
       oddout[k] = beta * dout[k][HALFORD - 1];
    }

    /* resolve all outputs (output schedulers) */

    for (k = 0; k < NUMFILT; k++) {
    memory[evenouttable[k][i]] = evenout[k];
    memory[oddouttable[k][i]] = oddout[k];
    for (l = 0; 1 < HALFORD - 1; l++)
      memory[douttable[l][k][i]] = dout[k][l];


    }

    /* output "placement* */

    /*print ("%7.4f ", memory[0]); */
    //if ((i % 8) == 7) printf("\n");

    //P1OUT = 0x01; //LED1 output test

    decomout[i] = memory[0]; //output value to reconstruction filter
    if ((i % 8) == 7) decomout[i] = 0;  //if i = 15, no output????

    /* shift the register contents */

    for (j = 1; j < SRSIZE; j++) memory[j - 1] = memory[j];
    memory[SRSIZE - 2] = 0.0;

    i++;  /*i is a continual loop from 0 to BLKSIZE - 1. */
    i &= (BLKSIZE - 1);

    }

    /* reverse transform */
    //DRIVE_DAC(decomout); //decomposition output HAVE BUTTON ACTIVATE WHICH
OUTPUT????
    recon(decomout); //send through reconstruction DECLARED CORRECTLY????

      return ;
}
```

```
 void recon(float *decomout) {

       static const int gamma[HALFORD] = {1, 0, 0}; /*using Haar coefficients
to save memory space*/
           static float beta = 0.707;
           int i, j, k, l;
           float memory[SRSIZEREC];  //changed to int from double, 26
           float evenin[NUMFILT], oddin[NUMFILT], e[NUMFILT][HALFORD],
din[NUMFILT - 1][HALFORD],
               dout[NUMFILT][HALFORD], evenout[NUMFILT], oddout[NUMFILT];
//changed to int from double

           static int evenintable[NUMFILT][BLKSIZE] = {21, 20, 19, 19, 17, 17,
17, 15, 16, 12, 13, 17, 9, 12, 12, 22,
                    6, 10, 25, 25, 25, 16, 25, 25, 21, 21, 21, 25, 25, 25,
25, 21 };

           static int oddintable[NUMFILT][BLKSIZE] = {22, 21, 21, 21, 19, 19,
20, 20, 20, 16, 16, 21, 21, 21, 21, 23,
                  13, 13, 25, 25, 25, 20, 25, 25, 21, 21, 21, 25, 25, 25, 25,
21};

           static int evenouttable[NUMFILT][BLKSIZE] = {25, 21, 25, 25, 19,
19, 25, 20, 20, 16, 21, 2, 3, 4, 5, 25,
                  7, 8, 25, 25, 25, 25, 25, 25, 25, 25, 1, 25, 25, 25, 25, 6
};

           static int oddouttable[NUMFILT][BLKSIZE] = {25, 20, 25, 25, 17, 17,
25, 15, 16, 12, 16, 1, 2, 3, 4, 25,
                  6, 7, 25, 25, 25, 25, 25, 25, 25, 25, 0, 25, 25, 25, 25, 6
};

           static int dintable[HALFORD-1][NUMFILT][BLKSIZE] = {20, 19, 24, 17,
16, 15, 14, 13, 12, 11, 10, 13, 20, 20, 20, 24,
                  18, 17, 25, 25, 25, 24, 25, 25, 24, 9, 8, 25, 25, 25, 25,
20,    19, 18, 24, 16, 15, 14, 13, 12, 11, 10, 9,
                  12, 16, 19, 19, 24, 17, 16, 25, 25, 25, 24, 25, 25, 24, 8,
7, 25, 25, 25, 25, 19};

           static int douttable[HALFORD-1][NUMFILT][BLKSIZE] = {20, 25, 18,
17, 16, 25, 14, 13, 12, 11, 25, 21, 21, 21, 21, 21,
                  18, 25, 25, 25, 25, 15, 25, 25, 10, 9, 14, 25, 25, 25, 25,
19,   19, 25, 17, 16, 15, 25, 13, 12, 11, 10, 25,
                  17, 20, 20, 20, 20, 17, 25, 25, 25, 25, 14, 25, 25, 9, 8,
13, 25, 25, 25, 25, 18 };


          //  printf (" Circular wavelet decomposition program\n");
           //     printf ("\n Output wavelet data:\n");

               /* initialize shift register memory and coutner */
               for (j = 0; j < SRSIZEREC; j++) memory[j] = 0.0;
               i = 0;

               /* input placement */
```

```
                while (memory[SRSIZEREC - 3] = decomout[(BLKSIZE - 1) - i])
/*(scanf ("%f", &memory[SRSIZE - 3]) <= EOF)*/ { //send in input from decom in
reverse order??????
                /* loop over all filters */

                for (k = 0; k < NUMFILT; k++) {

                    /* resolve all inputs (input schedulers) */

                    evenin[k] = memory[evenintable[k][i]];
                    oddin[k] = memory[oddintable[k][i]];
                    for (l = 0; l < HALFORD - 1; l++)
                          din[k][l] = memory[dintable[l][k][i]];

                    /* N-stage lattice filters */
                    for (l = HALFORD; l > 0; l--) {        /* gamma
coefficients have been reversed */
                          if (l == HALFORD) { /*first rung (normal) */
                                dout[k][0] = evenin[k] + gamma[2] * oddin[k];
                                e[k][0] = evenin[k] * gamma[2] - oddin[k];
                          }
                          else if ((l & 2) == 0) {

                                dout[k][l] = e[k][l - 1] + gamma[l - 1] *
din[k][l - 1];
                                e[k][l] = e[k][l - 1] * gamma[l - 1] -
din[k][l - 1];
                          }
                                else { /* inverted rung */
                        dout[k][l] = e[k][l - 1] * gamma[l - 1] + din[k][l -
1];
                        e[k][l] = e[k][l - 1] - gamma[l - 1] * din[k][l - 1];

                          }
                    }
                    evenout[k] = beta * e[k][HALFORD - 1];
                    oddout[k] = beta * dout[k][HALFORD - 1];
                }

                /* resolve all outputs (output schedulers) */

                for (k = 0; k < NUMFILT; k++) {
                memory[evenouttable[k][i]] = evenout[k];
                memory[oddouttable[k][i]] = oddout[k];
                for (l = 0; 1 < HALFORD - 1; l++)
                    memory[douttable[l][k][i]] = dout[k][l];

                }

                /* output "placement* */

                /*print ("%7.4f ", memory[0]); */
               // if ((i % 8) == 7) printf("\n");
```

```
                reconout[i] = memory[0]; //final output value
                if ((i % 8) == 7) reconout[i] = 0;  //if i = 15, no output????
            // do_some_conversions(reconout); //convert float to int
           //  Drive_DAC(b); //send output to DAC


                /* shift the register contents */

                for (j = 1; j < SRSIZEREC; j++) memory[j - 1] = memory[j];
                memory[SRSIZEREC - 2] = 0.0;

                i++;  /*i is a continual loop from 0 to BLKSIZE - 1. */
                i &= (BLKSIZE - 1);

                }

                /* convert and output final output to DAC */
                do_some_conversions(reconout); //convert float to int
                Drive_DAC(b); //send output to DAC

                return ; //back to while(1) loop HAVE THIS LOOP BACK TO WHILE
(1) to RECEIVE MORE INPUTS!!!!!
 }

  void Drive_DAC(int *b){
     unsigned int DAC_Word = 0;
     int d; //counter

     for (d = 0; d < BLKSIZE; d++) {
     DAC_Word = (0x1000) | (b[d] & 0x0FFF);
     // 0x1000 sets DAC for Write
     // to DAC, Gain = 2, /SHDN = 1
     // and put 12-bit level value
     // in low 12 bits.
     P1OUT &= ~BIT4;
     // Clear P1.4 (drive /CS low on DAC)
     // Using a port output to do this for now
     UCB0TXBUF = (DAC_Word >> 8);
     // Shift upper byte of DAC_Word
     // 8-bits to right
     while
     (!(UCB0IFG & UCTXIFG));
     // USCI_A0 TX buffer ready?
     UCB0TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
     (DAC_Word & 0x00FF);
     // Transmit lower byte to DAC
     while
     (!(UCB0IFG & UCTXIFG));
     // USCI_A0 TX buffer ready?
     __delay_cycles(150);
     // Delay 150 12 MHz SMCLK periods
     // (12.5 us) to allow SIMO to complete
     P1OUT |= BIT4;
     // Set P1.4 (drive /CS high on DAC)
     }
     return
     ;
     }

  void do_some_conversions(float *reconout)
  {
         int c; //conversion counter

       for (c = 0; c < BLKSIZE; c++) {
    b[c] = (int)floorf(reconout[c]);
    b[c] = (int)ceilf(reconout[c]);
    b[c] = (int)roundf(reconout[c]);
    b[c] = (int)truncf(reconout[c]);
    b[c] = (int)rintf(reconout[c]);
    b[c] = (int)nearbyintf(reconout[c]);
    b[c] = (int)reconout[c];
    b[c] = reconout[c];
       }
  }

//#pragma vector=PORT1_VECTOR
__interrupt void Port_1(void)
{

       for (a = 0; a < 16; a++ ) //takes 16 samples for wavelet transform
        {          //runs continuously

                  ADC14CTL0 |= ADC14ENC + ADC14SC; //enable taking samples
TAKE ONLY SIXTEEN AT A TIME!!!!!

                    //__bis_SR_register(CPUOFF + GIE); // LPM0, ADC10_ISR will
force exit //

                     decominput[a] = ADC14MEM15;

                     //waveletdecom(decominput); // run wavelet transform
(more TACCR0 cycles will be added to compensate for this)

                     TA0CCR0 += 1300;
                     //P1OUT |= BIT1; //test LED1
     }

       waveletdecom(decominput); // run wavelet transform (more TACCR0 cycles
will be added to compensate for this)

//P1OUT ^= (LED0 + LED1); // P1.0 = toggle
P1IFG &= ~BUTTON; // P1.1 IFG cleared
P1IES ^= BUTTON; // toggle the interrupt edge,
// the interrupt vector will be called
// when P1.1 goes from HitoLow as well as
// LowtoHigh
}
```

Figure A.1: Initial design for microcontroller code.

```
//****************************************************************************
*
//
// MSP432 main.c template - P1.0 port toggle
//
//****************************************************************************

#include "msp.h"

void main(void)
{
    volatile uint32_t i;

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    // The following code toggles P1.0 port
    P1DIR |= BIT0;                      // Configure P1.0 as output

    while(1)
    {
        P1OUT ^= BIT0;                  // Toggle P1.0
        for(i=10000; i>0; i--);         // Delay
    }
}
```

Figure A.2: LED toggle test code for MSP432P401R microcontroller.

```
int _system_pre_init( void )
        {
          WDTCTL = WDTPW | WDTHOLD;

          return 1;
        }
```

Figure A.3: “System_pre_init” code that freezes the watchdog timer before the program initializes.

```
//*****************************************************************************
*
//
// MSP432 main.c template - Empty main
//
//*****************************************************************************

#include "msp.h"
#include <math.h>

/*
 * main.c
 */
#include <stdio.h>
#define SRSIZE 49 /*size of the decomposition xform shift register */
#define BLKSIZE 16 /* input block size--must be a power of two */
#define NUMFILT 2 /* number of lattice filters required */
#define HALFORD 3 /* one half of the order of the filter */
#define SRSIZEREC 26 /*size of the resonstruction xform shift register */
#define BUTTON1 BIT4 //button bit P2.4
#define BUTTON2 BIT0 //button bit P3.0? P4.0 now

void waveletdecom(float *decominput); /*declaring decomposition filter */
void recon(float *decomout); /*declaring reconstruction filter*/
void do_some_conversions(float *reconout);
void Drive_DAC(int *b);
//int a; //evenin and oddin counter
//int evenin, oddin; //inputs to decom
float decominput[BLKSIZE]; //input to decomposition filter
float decomout[BLKSIZE]; //output of decomposition filter
float reconout[BLKSIZE]; //output of reconstruction filter
int a; //sample counter
int b[BLKSIZE]; //float to int conversion
int change; //values sets where output goes
//extern void INT_PORT1_Handler( void);
extern void INT_PORT2_Handler (void);
//extern void INT_PORT3_Handler (void);
extern void INT_PORT4_Handler (void);
//extern void ADC14_ISR_Handler( void );


void main(void) {
      //volatile uint32_t i;

   WDTCTL = WDTPW | WDTHOLD;     // Stop watchdog timer

    // 16Mhz SMCLK
  /*  if (CALBC1_16MHZ==0xFF) // If calibration constant erased
    {
         while(1); // do not load, trap CPU!!
    } */

    /*set MSP432 settings (preset calibration in MSP432?*/
   /* DCOCTL = 0; // Select lowest DCOx and MODx settings
    BCSCTL1 = CALBC1_16MHZ; // Set range
    DCOCTL = CALDCO_16MHZ; */  // Set DCO step + modulation RUN ON SMCLK FOR
a?????
    CSCTL0 = DCORSEL_4; //DCOCLK, SMCLK = 24 MHz

    //ADC14 uses MODCLK?

    ADC14CTL0 = ADC14SHT1_2 + ADC14ON + ADC14IE15; // ADC14ON, interrupt
enabled, 16 clock samples

    ADC14MCTL0 = ADC14INCH_15; // input A15, P6.0

    UCB0CTL0 |= UCCKPL + UCMSB + UCMST + UCSYNC;

      UCB0CTL1 |= UCSSEL_2; /* SMCLK (input clock?) */

      UCB0BR0 |= 0x00; //no SMCLK division

      UCB0BR1 |= 0x00;

      UCB0CTL1 &= ~UCSWRST;

      /* set inputs and outputs */
    P4DIR |= BIT1; //set P4.1 to output to test button
    P4OUT ^= BIT1; //test external LED

    P1DIR |= BIT0; //set LED1 output P1.0
    P2DIR |= BIT2; //set Blue_LED
    P1OUT &= ~BIT0; //turn off LED1
    P2OUT &= ~BIT2; //turn off blue LED
    //  P6DIR |= 0x01; //set P6.0 output
    P1DIR |= BIT6; //set P1.6 output

      TA0CCTL0 = CCIE;                             // CCR0 interrupt enabled
      TA0CCR0 = 12000;
      TA0CTL = TASSEL_2 + MC_2;                   // SMCLK, contmode, time_A0
control

     // __delay_cycles(3000000); //delay for a short period before enabling
interrupts, as a small signal that triggers the interrupts passes during this
time

      P4DIR &= ~BUTTON2;                    // button is an input
      P4OUT |= BUTTON2;                     // pull-up resistor
      P4REN |= BUTTON2;                     // resistor enabled

                // __delay_cycles(3000000); //delay for a short period before
enabling interrupts, as a small signal that triggers the interrupts passes
during this time


      P4IES &= ~BUTTON2;                     // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
      P4IFG &= ~BUTTON2; // P2.4 IFG cleared
      P4IE |= BUTTON2; // P2.4 interrupt enabled
      //enable interrupts///
      //take data button//
      P2DIR &= ~BUTTON1;                     // button is an input
      P2OUT |= BUTTON1;                      // pull-up resistor
      P2REN |= BUTTON1;                      // resistor enabled

     // __delay_cycles(3000000); //delay for a short period before enabling
interrupts, as a small signal that triggers the interrupts passes during this
time


      P2IES &= ~BUTTON1;                      // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
      P2IFG &= ~BUTTON1; // P2.4 IFG cleared
      P2IE |= BUTTON1; // P2.4 interrupt enabled

     // P2IFG &= ~BUTTON1; // P2.4 IFG cleared

     //change output button//
    /*  P3DIR &= ~BUTTON2;                      // button is an input
      P3OUT |= BUTTON2;                      // pull-up resistor
      P3REN |= BUTTON2;                      // resistor enabled

     // __delay_cycles(3000000); //delay for a short period before enabling
interrupts, as a small signal that triggers the interrupts passes during this
time

      P3IES &= ~BUTTON2;                      // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
      P3IFG &= ~BUTTON1; // P3.0 IFG cleared
      P3IE |= BUTTON2; // P3.0 interrupt enabled */

     /* P4DIR &= ~BUTTON1;                      // button is an input
      P4OUT |= BUTTON1;                      // pull-up resistor
      P4REN |= BUTTON1;                      // resistor enabled

            // __delay_cycles(3000000); //delay for a short period before
enabling interrupts, as a small signal that triggers the interrupts passes
during this time


      P4IES &= ~BUTTON1;                      // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
      P4IFG &= ~BUTTON1; // P2.4 IFG cleared
      P4IE |= BUTTON1; // P2.4 interrupt enabled */



      //P3IFG &= ~BUTTON1; // P3.0 IFG cleared
      change = 0; //output goes to decom default

      //INTERRUPT IS BEING CALLED FOR INT_PORT3_HANDLER and INT_PORT2_HANDLER
FOR SOME REASON

     //__enable_interrupts(); // enable all interrupts

     // __delay_cycles(300000); //delay for a short period before enabling
interrupts, as a small signal that triggers the interrupts passes during this
time

      NVIC_ISER1 = 1 << ((INT_PORT4 - 16) & 31); //enable TA1_1 interrupt for
P3.0 Button (change output) SHOULD NVIC_ISER1 BE CHANGED?? MAYBE NVIC_ISER2??

      NVIC_ISER1 = 1 << ((INT_PORT2 - 16) & 31); //enable TA0_0 interrupt for
P2.4 Button (take data)

      //__delay_cycles(500);

     // NVIC_ISER1 = 1 << ((INT_PORT4 - 16) & 31); //enable TA1_1 interrupt
for P3.0 Button (change output) SHOULD NVIC_ISER1 BE CHANGED?? MAYBE
NVIC_ISER2??
  //  P1DIR |= BIT0;                      // Configure P1.0 as output

     __enable_interrupts();

   //  P4IFG &= ~BUTTON2; // P3.0 IFG cleared

     while(1){
     }



     /*   while(1)
        {
            P1OUT ^= BIT0;                  // Toggle P1.0
            for(i=10000; i>0; i--);         // Delay
        } */
}

int _system_pre_init( void )
        {
          WDTCTL = WDTPW | WDTHOLD;

          return 1;
        }


void waveletdecom(float *decominput) {

    /* set LED outputs for each stage of code for testing */

    static const int gamma[HALFORD] = {1, 0, 0}; /*using Haar coefficients to
save memory space*/
    static float beta = 0.707;
    int i, j, k, l;
    float memory[SRSIZE];  //changed to int from double
    float evenin[NUMFILT], oddin[NUMFILT], e[NUMFILT][HALFORD], din[NUMFILT -
1][HALFORD],
        dout[NUMFILT][HALFORD], evenout[NUMFILT], oddout[NUMFILT]; //changed
to int from double

    static int evenintable[NUMFILT][BLKSIZE] = {30, 45, 43, 45, 34, 45, 39,
45, 43, 45, 41, 45, 43, 45, 41, 45,
             48, 31, 48, 31, 48, 30, 48, 27, 48, 30, 48,   28,    48, 42, 48,
48};

    static int oddintable[NUMFILT][BLKSIZE] = {31, 46, 40, 46, 33, 46, 36, 46,
35, 46, 33, 46, 33, 46, 31, 46,
             48, 32, 48, 33, 48, 16, 48,      29, 48, 29,   48, 27, 48,  32,
48,   48 };

    static int evenouttable[NUMFILT][BLKSIZE] = { 22,   48,  48,  48,  16,
35,  12,  34,  48,  33,  6, 32, 48,
             31, 1, 30,  48,  22,  48, 48,  48,  16,  48,  12,  48,  48,  48,
6,  48,  48,  48,  48 };

    static int oddouttable[NUMFILT][BLKSIZE] = { 45, 48, 48, 48, 35, 45, 45,
45, 48, 45, 45, 42, 48, 38, 0, 22,
             48, 41, 48,  48,  48, 33, 48, 36, 48, 48, 48, 34, 48, 48, 48, 48
};

    static int dintable[HALFORD-1][NUMFILT][BLKSIZE] = { 44, 47, 47, 41,37,
      39,    35,    37,    47,    36,    34,    42,    47,
             43,    42,    43,    48,    43,    48,    38,    48,    36,    48,
      38,    48, 44,       48,    40,    48,    44,    48,    48,    43,    47,
      47,    40,    36,    38,
          31, 36,   47,    35,    27,    34,    47,    38,    25,    42,    48,
      42,    48,    37,    48,    35,    48,    35,    48,    33,    48, 33,
      48,    31,    48,    48 };

    static int douttable[HALFORD-1][NUMFILT][BLKSIZE] = { 44, 43, 39, 41, 37,
      39,    39,    38,    45,    44,    41,    45,    45,
             45,    48,    45,    48,    48,    48,    38,    48,    36,    48,
      48,    48,    35,    48,    48,    48,    43,    48,    48,    43,    42,
      38,    40,    36,    38,
             36,    37,    34,    36,    34,    40,    32,    44,    48,    44,
      48,    48,    48,    37,    48,    32,    48,    48,    48,    28,    48,
      48,    48,    26,    48,    48     };

   // printf (" Circular wavelet decomposition program\n");
   // printf ("\n Output wavelet data:\n");

    /* initialize shift register memory and counter */
    for (j = 0; j < SRSIZE; j++) memory[j] = 0.0;
    i = 0;

    /* input placement */
    while (memory[SRSIZE - 3] = decominput[i])/* (scanf ("%f", &memory[SRSIZE
- 3]) <= EOF) */ {  //WHAT CAUSES THIS WHILE LOOP TO END???

    /* loop over all filters */

    for (k = 0; k < NUMFILT; k++) {

      /* resolve all inputs (input schedulers) */

      evenin[k] = memory[evenintable[k][i]];
      oddin[k] = memory[oddintable[k][i]];
      for (l = 0; l < HALFORD - 1; l++)
             din[k][l] = memory[dintable[l][k][i]];

      /* N-stage lattice filters */
      for (l = 0; l < HALFORD; l++) {
             if (l == 0) { /*first rung (normal) */
                    dout[k][0] = evenin[k] + gamma[0] * oddin[k];
                    e[k][0] = evenin[k] * gamma[0] - oddin[k];
             }
             else if ((l & 1) == 0) {

                    dout[k][l] = e[k][l - 1] + gamma[l] * din[k][l - 1];
                    e[k][l] = e[k][l - 1] * gamma[l] - din[k][l - 1];
             }
             else { /* inverted rung */

                 dout[k][l] = e[k][l - 1] * gamma[l] + din[k][l - 1];
                 e[k][l] = e[k][l - 1] - gamma[l] * din[k][l - 1];

            }
      }
        evenout[k] = beta * e[k][HALFORD - 1];
        oddout[k] = beta * dout[k][HALFORD - 1];
    }

    /* resolve all outputs (output schedulers) */

    for (k = 0; k < NUMFILT; k++) {
    memory[evenouttable[k][i]] = evenout[k];
    memory[oddouttable[k][i]] = oddout[k];
    for (l = 0; 1 < HALFORD - 1; l++)
      memory[douttable[l][k][i]] = dout[k][l];


    }

    /* output "placement* */

    /*print ("%7.4f ", memory[0]); */
    //if ((i % 8) == 7) printf("\n");

    //P1OUT = 0x01; //LED1 output test

    decomout[i] = memory[0]; //output value to reconstruction filter
    if ((i % 8) == 7) decomout[i] = 0;  //if i = 15, no output????

    /* shift the register contents */

    for (j = 1; j < SRSIZE; j++) memory[j - 1] = memory[j];
    memory[SRSIZE - 2] = 0.0;

    i++;  /*i is a continual loop from 0 to BLKSIZE - 1. */
    i &= (BLKSIZE - 1);

    }

    /* reverse transform */
    //

//    while(1) {}  //test loop

    /* convert and output final output to DAC  or send to reconstruction*/
    if (change == 1) {
       // recon(decomout); //test

            do_some_conversions(decomout); //convert float to int
          Drive_DAC(b); //send output to DAC
    }
          else {
             //  do_some_conversions(decomout); //convert float to int
                //         Drive_DAC(b); //send output to DAC

    //decomposition output HAVE BUTTON ACTIVATE WHICH OUTPUT????
    recon(decomout);   //send through reconstruction DECLARED CORRECTLY????
    }
   // recon(decomout); //testing

      return ;
}

void recon(float *decomout) {

       static const int gamma[HALFORD] = {1, 0, 0}; /*using Haar coefficients
to save memory space*/
           static float beta = 0.707;
           int i, j, k, l;
           float memory[SRSIZEREC];  //changed to int from double, 26
           float evenin[NUMFILT], oddin[NUMFILT], e[NUMFILT][HALFORD],
din[NUMFILT - 1][HALFORD],
               dout[NUMFILT][HALFORD], evenout[NUMFILT], oddout[NUMFILT];
//changed to int from double

           static int evenintable[NUMFILT][BLKSIZE] = {21, 20, 19, 19, 17, 17,
17, 15, 16, 12, 13, 17, 9, 12, 12, 22,
                    6, 10, 25, 25, 25, 16, 25, 25, 21, 21, 21, 25, 25, 25,
25, 21 };

           static int oddintable[NUMFILT][BLKSIZE] = {22, 21, 21, 21, 19, 19,
20, 20, 20, 16, 16, 21, 21, 21, 21, 23,
                   13, 13, 25, 25, 25, 20, 25, 25, 21, 21, 21, 25, 25, 25, 25,
21};

            static int evenouttable[NUMFILT][BLKSIZE] = {25, 21, 25, 25, 19,
19, 25, 20, 20, 16, 21, 2, 3, 4, 5, 25,
                   7, 8, 25, 25, 25, 25, 25, 25, 25, 25, 1, 25, 25, 25, 25, 6
};

            static int oddouttable[NUMFILT][BLKSIZE] = {25, 20, 25, 25, 17, 17,
25, 15, 16, 12, 16, 1, 2, 3, 4, 25,
                   6, 7, 25, 25, 25, 25, 25, 25, 25, 25, 0, 25, 25, 25, 25, 6
};

            static int dintable[HALFORD-1][NUMFILT][BLKSIZE] = {20, 19, 24, 17,
16, 15, 14, 13, 12, 11, 10, 13, 20, 20, 20, 24,
                   18, 17, 25, 25, 25, 24, 25, 25, 24, 9, 8, 25, 25, 25, 25,
20,    19, 18, 24, 16, 15, 14, 13, 12, 11, 10, 9,
                   12, 16, 19, 19, 24, 17, 16, 25, 25, 25, 24, 25, 25, 24, 8,
7, 25, 25, 25, 25, 19};

            static int douttable[HALFORD-1][NUMFILT][BLKSIZE] = {20, 25, 18,
17, 16, 25, 14, 13, 12, 11, 25, 21, 21, 21, 21, 21,
                   18, 25, 25, 25, 25, 15, 25, 25, 10, 9, 14, 25, 25, 25, 25,
19,   19, 25, 17, 16, 15, 25, 13, 12, 11, 10, 25,
                   17, 20, 20, 20, 20, 17, 25, 25, 25, 25, 14, 25, 25, 9, 8,
13, 25, 25, 25, 25, 18 };


          //  printf (" Circular wavelet decomposition program\n");
           //     printf ("\n Output wavelet data:\n");

                /* initialize shift register memory and coutner */
                for (j = 0; j < SRSIZEREC; j++) memory[j] = 0.0;
                i = 0;

                /* input placement */

                while (memory[SRSIZEREC - 3] = decomout[(BLKSIZE - 1) - i])
/*(scanf ("%f", &memory[SRSIZE - 3]) <= EOF)*/ { //send in input from decom in
reverse order??????

                /* loop over all filters */

                for (k = 0; k < NUMFILT; k++) {

                    /* resolve all inputs (input schedulers) */

                    evenin[k] = memory[evenintable[k][i]];
                    oddin[k] = memory[oddintable[k][i]];
                    for (l = 0; l < HALFORD - 1; l++)
                          din[k][l] = memory[dintable[l][k][i]];

                    /* N-stage lattice filters */
                    for (l = HALFORD; l > 0; l--) {        /* gamma
coefficients have been reversed */
```

```
                    if (l == HALFORD) { /*first rung (normal) */
                          dout[k][0] = evenin[k] + gamma[2] * oddin[k];
                          e[k][0] = evenin[k] * gamma[2] - oddin[k];
                    }
                    else if ((l & 2) == 0) {

                          dout[k][l] = e[k][l - 1] + gamma[l - 1] *
din[k][l - 1];
                          e[k][l] = e[k][l - 1] * gamma[l - 1] -
din[k][l - 1];
                    }
                          else { /* inverted rung */
                       dout[k][l] = e[k][l - 1] * gamma[l - 1] + din[k][l -
1];
                       e[k][l] = e[k][l - 1] - gamma[l - 1] * din[k][l - 1];

                    }
                }
                evenout[k] = beta * e[k][HALFORD - 1];
                oddout[k] = beta * dout[k][HALFORD - 1];
            }

            /* resolve all outputs (output schedulers) */

            for (k = 0; k < NUMFILT; k++) {
            memory[evenouttable[k][i]] = evenout[k];
            memory[oddouttable[k][i]] = oddout[k];
            for (l = 0; 1 < HALFORD - 1; l++)
                memory[douttable[l][k][i]] = dout[k][l];


            }

            /* output "placement* */

            /*print ("%7.4f ", memory[0]); */
           // if ((i % 8) == 7) printf("\n");

            reconout[i] = memory[0]; //final output value
            if ((i % 8) == 7) reconout[i] = 0;  //if i = 15, no output????
        // do_some_conversions(reconout); //convert float to int
       //  Drive_DAC(b); //send output to DAC


            /* shift the register contents */

            for (j = 1; j < SRSIZEREC; j++) memory[j - 1] = memory[j];
            memory[SRSIZEREC - 2] = 0.0;

            i++;  /*i is a continual loop from 0 to BLKSIZE - 1. */
            i &= (BLKSIZE - 1);

            }

            P1OUT ^= BIT0;  //test, turn on LED
```

```
                __delay_cycles(30000);  //test
                P1OUT ^= BIT0;  //test, turn off LED

               // while(1){} //test loop

                /* convert and output final output to DAC */
               // do_some_conversions(reconout); //convert float to int
              //  Drive_DAC(b); //send output to DAC *?

                return ; //back to while(1) loop HAVE THIS LOOP BACK TO WHILE
(1) to RECEIVE MORE INPUTS!!!!!
}

void Drive_DAC(int *b){
    unsigned int DAC_Word = 0;
    int d; //counter

    for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x1000) | (b[d] & 0x0FFF);
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    P1OUT &= ~BIT4;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
    UCB0TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB0TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    __delay_cycles(150);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P1OUT |= BIT4;
    // Set P1.4 (drive /CS high on DAC)
    }


    P1OUT ^= BIT0;  //test, turn on LED
    __delay_cycles(30000);  //test
    P1OUT ^= BIT0;  //test, turn off LED

   // while(1) {} //test loop

    return
    ;
    }
```

```
 void do_some_conversions(float *reconout)
 {
        int c; //conversion counter

      for (c = 0; c < BLKSIZE; c++) {
   b[c] = (int)floorf(reconout[c]);
   b[c] = (int)ceilf(reconout[c]);
   b[c] = (int)roundf(reconout[c]);
   b[c] = (int)truncf(reconout[c]);
   b[c] = (int)rintf(reconout[c]);
   b[c] = (int)nearbyintf(reconout[c]);
   b[c] = (int)reconout[c];
   b[c] = reconout[c];
      }
 }

// when button is released after being pressed, the interrupt takes effect.
Change this to make it when you first press the button

void INT_PORT2_Handler( void) //interrupt take data
{
      /*volatile uint32_t i;

        while(1)
              {
                  P1OUT ^= BIT0;                     // Toggle P1.0
                  for(i=10000; i>0; i--);            // Delay
              } */


      for (a = 0; a < 16; a++ ) //takes 16 samples for wavelet transform
             {          //runs continuously

                         ADC14CTL0 |= ADC14ENC + ADC14SC; //enable taking
samples TAKE ONLY SIXTEEN AT A TIME!!!!!

                           //__bis_SR_register(CPUOFF + GIE); // LPM0,
ADC10_ISR will force exit //

                            decominput[a] = ADC14MEM15;

                            //waveletdecom(decominput); // run wavelet
transform (more TACCR0 cycles will be added to compensate for this)

                            TA0CCR0 += 1300;
                            //P1OUT |= BIT1; //test LED1
           }

        P1OUT ^= BIT0;  //test, turn on LED
                       __delay_cycles(300000);  //test
                      P1OUT ^= BIT0;  //test, turn off LED

      waveletdecom(decominput); // run wavelet transform (more TACCR0 cycles
will be added to compensate for this)
```

```
         P2IFG &= ~BUTTON1; // P2.4 IFG cleared
        // P2IES ^= BUTTON1; // toggle the interrupt edge,


    }

          void INT_PORT4_Handler (void ) //interrupt change output (HOW WILL
interrupts know which void to go to????)
          {
                P4OUT |= BIT1; //test external LED
                if (change == 1){
                      change = 0;
                }
                else {
                      change = 1;
                }


                P1OUT ^= BIT0;  //test, turn on LED

               // P3IFG &= ~BUTTON2; // P3.0 IFG cleared
             //   P3IES ^= BUTTON2; // toggle the interrupt edge,

              //  P4IFG &= ~BUTTON2; // P3.0 IFG cleared
                __delay_cycles(30000);  //test

                P1OUT ^= BIT0;  //test, turn off LED


              //  while(1){} //test loop

               // P3IFG &= ~BUTTON2; // P3.0 IFG cleared
                P4IFG &= ~BUTTON2; // P3.0 IFG cleared

          }
```

Figure A.4: State of microcontroller code after button interrupt routine adjustments.

Neural network code (wavelet transform code removed due to the large size of the code):

```
%take data, make wav file
  recObjenable = audiorecorder(1000,16,2,1);

%record and audio

%red cord = channel 1, white cord = channel 2



    disp('recording "AAAA" data in three seconds');

    pause(3); % pause for three seconds
    disp('"AAAA" data recording');

    %can't use loops unfortunately for getaudiodata....


    recordblocking(recObjenable, 1); %record one second of data
    A1 = getaudiodata(recObjenable, 'double'); %get twenty "AAAAAAA"
samples

    recordblocking(recObjenable, 1);
    A2 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A3 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A4 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A5 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A6 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A7 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A8 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A9 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A10 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A11 = getaudiodata(recObjenable, 'double');
    recordblocking(recObjenable, 1);
    A12 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A13 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A14 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A15 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A16 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A17 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A18 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A19 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    A20 = getaudiodata(recObjenable, 'double');



    disp('recording "EEEE" data in three seconds');

    pause(3); % pause for three seconds
    disp('"EEEE" data recording');


    recordblocking(recObjenable, 1); %record one second of data
    E1 = getaudiodata(recObjenable, 'double'); %get twenty "EEEEEEE"
samples

    recordblocking(recObjenable, 1);
    E2 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E3 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E4 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E5 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E6 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E7 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E8 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E9 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E10 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E11 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E12 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E13 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E14 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E15 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E16 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E17 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E18 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E19 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    E20 = getaudiodata(recObjenable, 'double');

    disp('recording "IIII" data in three seconds');

    pause(3); % pause for three sec
    disp('"IIII" data recording');


    recordblocking(recObjenable, 1); %record one second of data
    I1 = getaudiodata(recObjenable, 'double'); %get twenty "IIIIIII"
samples
```

```
recordblocking(recObjenable, 1);
I2 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I3 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I4 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I5 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I6 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I7 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I8 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I9 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I10 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I11 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I12 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I13 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I14 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I15 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I16 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I17 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
I18 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
```

```
    I19 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    I20 = getaudiodata(recObjenable, 'double');




    disp('recording "OOOO" data in three seconds');

    pause(3); % pause for five sec
    disp('"OOOO" data recording');


    recordblocking(recObjenable, 1); %record one second of data
    O1 = getaudiodata(recObjenable, 'double'); %get twenty "OOOOOOO"
samples

    recordblocking(recObjenable, 1);
    O2 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O3 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O4 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O5 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O6 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O7 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O8 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O9 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O10 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O11 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O12 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O13 = getaudiodata(recObjenable, 'double');
    recordblocking(recObjenable, 1);
    O14 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O15 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O16 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O17 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O18 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O19 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    O20 = getaudiodata(recObjenable, 'double');




    disp('recording "UUUU" data in three seconds');

    pause(3); % pause for five sec
    disp('"UUUU" data recording');


    recordblocking(recObjenable, 1); %record one second of data
    U1 = getaudiodata(recObjenable, 'double'); %get twenty "UUUUUUU"
samples

    recordblocking(recObjenable, 1);
    U2 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    U3 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    U4 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    U5 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    U6 = getaudiodata(recObjenable, 'double');

    recordblocking(recObjenable, 1);
    U7 = getaudiodata(recObjenable, 'double');
```

```
recordblocking(recObjenable, 1);
U8 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U9 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U10 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U11 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U12 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U13 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U14 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U15 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U16 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U17 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U18 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U19 = getaudiodata(recObjenable, 'double');

recordblocking(recObjenable, 1);
U20 = getaudiodata(recObjenable, 'double');


%disp(length(y(:,2)));
 OUTA = zeros(5, 20);
 OUTE = zeros(5, 20);
 OUTI = zeros(5, 20);
 OUTO = zeros(5, 20);
 OUTU = zeros(5, 20);




 for a = 1 : 20  %for A vowel samples

   %  for q = 1 : length(A1(:,2))
```

```
if a == 1

    x(:) = A1(:,2);

elseif a == 2

    x(:) = A2(:,2);

elseif a == 3

    x(:) = A3(:,2);

elseif a == 4

    x(:) = A4(:,2);

elseif a == 5

    x(:) = A5(:,2);

elseif a == 6

    x(:) = A6(:,2);

elseif a == 7

    x(:) = A7(:,2);


 elseif a == 8

    x(:) = A8(:,2);

elseif a == 9

    x(:) = A9(:,2);

elseif a == 10

    x(:) = A10(:,2);

elseif a == 11

    x(:) = A11(:,2);

elseif a == 12

    x(:) = A12(:,2);

elseif a == 13
```

```
                x(:) = A13(:,2);

            elseif a == 14

                x(:) = A14(:,2);

            elseif a == 15

                x(:) = A15(:,2);

            elseif a == 16

                x(:) = A16(:,2);

            elseif a == 17

                x(:) = A17(:,2);

            elseif a == 18

                x(:) = A18(:,2);

            elseif a == 19

                x(:) = A19(:,2);

            elseif a == 20

                x(:) = A20(:,2);


            else


         end


            INPUTa(1,a) = mad(x(:)); %mean absolute deviation of the
signal

             INPUTa(2,a) = rms(x(:)); %get root mean square of signal
%
             INPUTa(3,a) = var(x(:)); %get the variance

            INPUTa(4,a) = std(x(:)); %get the standard deviation

             INPUTa(5,a) = step(dsp.ZeroCrossingDetector, x(:)); %get
the number of zero crossings
%
             INPUTa(6,a) = length(find(abs(diff(sign(x(:))))==2)); %get
the number of slope changes
```

```
       OUTA(1,a) = 1;


end

 %  y = zeros(1);

   disp('test1');

for a = 1 : 20  %for E vowel samples

   % for q = 1 : length(E1(:,2))

    if a == 1

           x(:) = E1(:,2);

       elseif a == 2

           x(:) = E2(:,2);

       elseif a == 3

           x(:) = E3(:,2);

       elseif a == 4

           x(:) = E4(:,2);

       elseif a == 5

           x(:) = E5(:,2);

       elseif a == 6

           x(:) = E6(:,2);

       elseif a == 7

           x(:) = E7(:,2);

        elseif a == 8

           x(:) = E8(:,2);

       elseif a == 9
```

```
            x(:) = E9(:,2);
        elseif a == 10
            x(:) = E10(:,2);
        elseif a == 11
            x(:) = E11(:,2);
        elseif a == 12
            x(:) = E12(:,2);
        elseif a == 13
            x(:) = E13(:,2);
        elseif a == 14
            x(:) = E14(:,2);
        elseif a == 15
            x(:) = E15(:,2);
        elseif a == 16
            x(:) = E16(:,2);
        elseif a == 17
            x(:) = E17(:,2);
        elseif a == 18
            x(:) = E18(:,2);
        elseif a == 19
            x(:) = E19(:,2);
        elseif a == 20
            x(:) = E20(:,2);
        else

    end
```

```
            INPUTe(1,a) = mad(x(:)); %moving average of signal

             INPUTe(2,a) = rms(x(:)); %get root mean square of signal
%
             INPUTe(3,a) = var(x(:)); %get the variance
%
            INPUTe(4,a) = std(x(:)); %get the standard deviation

             INPUTe(5,a) = step(dsp.ZeroCrossingDetector, x(:)); %get
the number of zero crossings
%
             INPUTe(6,a) = length(find(abs(diff(sign(x(:))))==2)); %get
the number of slope changes



            OUTE(2,a) = 1;



     end


       % z = zeros(1);

        disp('test2');

     for a = 1 : 20  %for I vowel samples

       %  for q = 1 : length(I1(:,2))

        if a == 1

                x(:) = I1(:,2);

            elseif a == 2

                x(:) = I2(:,2);

            elseif a == 3

                x(:) = I3(:,2);

            elseif a == 4

                x(:) = I4(:,2);

            elseif a == 5

                x(:) = I5(:,2);
```

```
elseif a == 6

    x(:) = I6(:,2);

elseif a == 7

    x(:) = I7(:,2);

 elseif a == 8

    x(:) = I8(:,2);

elseif a == 9

    x(:) = I9(:,2);

elseif a == 10

    x(:) = I10(:,2);

elseif a == 11

    x(:) = I11(:,2);

elseif a == 12

    x(:) = I12(:,2);

elseif a == 13

    x(:) = I13(:,2);

elseif a == 14

    x(:) = I14(:,2);

elseif a == 15

    x(:) = I15(:,2);

elseif a == 16

    x(:) = I16(:,2);

elseif a == 17

    x(:) = I17(:,2);

elseif a == 18

    x(:) = I18(:,2);
```

```
            elseif a == 19

                x(:) = I19(:,2);

            elseif a == 20

                x(:) = I20(:,2);

            else



            end

       %  end


            INPUTi(1,a) = mad(x(:)); %moving average of signal

            INPUTi(2,a) = rms(x(:)); %get root mean square of signal
%
             INPUTi(3,a) = var(x(:)); %get the variance

            INPUTi(4,a) = std(x(:)); %get the standard deviation

             INPUTi(5,a) = step(dsp.ZeroCrossingDetector, x(:)); %get
the number of zero crossings
%
             INPUTi(6,a) = length(find(abs(diff(sign(x(:))))==2)); %get
the number of slope changes




            OUTI(3,a) = 1;

    end

   % v = zeros(1);

    disp('test3');

    for a = 1 : 20  %for O vowel samples

         if a == 1

                x(:) = O1(:,2);

            elseif a == 2

                x(:) = O2(:,2);
```

```
elseif a == 3

    x(:) = O3(:,2);

elseif a == 4

    x(:) = O4(:,2);

elseif a == 5

    x(:) = O5(:,2);

elseif a == 6

    x(:) = O6(:,2);

elseif a == 7

    x(:) = O7(:,2);

 elseif a == 8

    x(:) = O8(:,2);

elseif a == 9

    x(:) = O9(:,2);

elseif a == 10

    x(:) = O10(:,2);

elseif a == 11

    x(:) = O11(:,2);

elseif a == 12

    x(:) = O12(:,2);

elseif a == 13

    x(:) = O13(:,2);

elseif a == 14

    x(:) = O14(:,2);

elseif a == 15

    x(:) = O15(:,2);
```

```
            elseif a == 16

                x(:) = O16(:,2);

            elseif a == 17

                x(:) = O17(:,2);

            elseif a == 18

                x(:) = O18(:,2);

            elseif a == 19

                x(:) = O19(:,2);

            elseif a == 20

                x(:) = O20(:,2);

            else



            end



            INPUTo(1,a) = mad(x(:)); %moving average of signal

             INPUTo(2,a) = rms(x(:)); %get root mean square of signal
%
             INPUTo(3,a) = var(x(:)); %get the variance
%
            INPUTo(4,a) = std(x(:)); %get the standard deviation

             INPUTo(5,a) = step(dsp.ZeroCrossingDetector, x(:)); %get
the number of zero crossings
%
             INPUTo(6,a) = length(find(abs(diff(sign(x(:))))==2)); %get
the number of slope changes

            OUTO(4,a) = 1;

    end

    %r = zeros(1);

    disp('test4');

    for a = 1 : 20  %for U vowel samples
```

```
% for q = 1 : length(U1(:,2))
 if a == 1

        x(:) = U1(:,2);

    elseif a == 2

        x(:) = U2(:,2);

    elseif a == 3

        x(:) = U3(:,2);

    elseif a == 4

        x(:) = U4(:,2);

    elseif a == 5

        x(:) = U5(:,2);

    elseif a == 6

        x(:) = U6(:,2);

    elseif a == 7

        x(:) = U7(:,2);

     elseif a == 8

        x(:) = U8(:,2);

    elseif a == 9

        x(:) = U9(:,2);

    elseif a == 10

        x(:) = U10(:,2);

    elseif a == 11

        x(:) = U11(:,2);

    elseif a == 12

        x(:) = U12(:,2);

    elseif a == 13

        x(:) = U13(:,2);
```

```
            elseif a == 14

                x(:) = U14(:,2);

            elseif a == 15

                x(:) = U15(:,2);

            elseif a == 16

                x(:) = U16(:,2);

            elseif a == 17

                x(:) = U17(:,2);

            elseif a == 18

                x(:) = U18(:,2);

            elseif a == 19

                x(:) = U19(:,2);

            elseif a == 20

                x(:) = U20(:,2);

            else


            end


            INPUTu(1,a) = mad(x(:)); %moving average of signal

            INPUTu(2,a) = rms(x(:)); %get root mean square of signal
 %
            INPUTu(3,a) = var(x(:)); %get the variance

            INPUTu(4,a) = std(x(:)); %get the standard deviation   4,a

             INPUTu(5,a) = step(dsp.ZeroCrossingDetector, x(:)); %get
the number of zero crossings
%
             INPUTu(6,a) = length(find(abs(diff(sign(x(:))))==2)); %get
the number of slope changes


            OUTU(5,a) = 1;
```

```
    end

   % x = zeros(1);

    OUT = horzcat(OUTA, OUTE, OUTI, OUTO, OUTU);

    INPUT = horzcat(INPUTa, INPUTe, INPUTi, INPUTo, INPUTu);


%     net = perceptron;  %comment out the layer statements, random
%     initialization, and "patternnet" neural network to use perception
%     nueral network


    net = patternnet(20, 'trainlm'); %declare patternnet neural network
with 10 hidden nodes in the hidden layer
% using Levenberg-Marquardt algorithm

    net.layers{1}.transferFcn = 'tansig'; %declare activation function
of the neurons in hidden layer (layer 1)
% as hyperbolic tangent

    net.layers{2}.transferFcn = 'purelin'; %declare activation function
of the neuron of the output layer (layer 2) to be linear
%
  %  net.numinputs = 7;  %there are seven inputs to the neural network

   % net.numoutputs = 5; %there are five output nodes for the neural
network

    net.trainParam.epochs = 100; %train for 100 epochs

    net.trainParam.goal=1e-12; %set desired MSE to be 1E-12

    %initialize weights and biases randomly

    net.initFcn = 'initlay';
    net.layers{1}.initFcn = 'initwb';
    net.layers{2}.initFcn = 'initwb';
    net.inputWeights{1,1}.initFcn = 'rands';
    net.layerWeights{2,1}.initFcn = 'rands';
    net.biases{1,1}.initFcn = 'rands';
    net.biases{2,1}.initFcn = 'rands';


    disp('test5');

    net = configure(net, INPUT , OUT); %initialize weights and biases


initial_output = net(INPUT); %get output before training (final
samples)
```

```
net = train(net,INPUT, OUT ); %train the neural network

test_output = net(INPUT); %get final outp
```

Figure A.5: Neural network code in Matlab (no wavelet transform).

```
//*****************************************************************************
*
//
// MSP432 main.c template - Empty main
//
//*****************************************************************************

#include "msp.h"
#include <math.h>


#define BLKSIZE 16 /* input block size--must be a power of two */
int a; //sample counter
//float decominput[BLKSIZE]; //input to decomposition filter
//int b[BLKSIZE]; //float to int conversion
int b;
//extern void TA0_N_ISR_HANDLER(void);
float test;




void Drive_DAC(int test){
    unsigned int DAC_Word = 0;
 //   int d; //counter

  //  for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x1000) | (test & 0x0FFF);
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    P6OUT &= ~BIT4;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
    UCB0TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB0TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    __delay_cycles(330);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P6OUT |= BIT4;
    // Set P1.4 (drive /CS high on DAC)
 //   }
```

```
   // P1OUT ^= BIT0;  //test, turn on LED
   // __delay_cycles(30000);  //test
   // P1OUT ^= BIT0;  //test, turn off LED

   // P1OUT |= BIT6;

  //  while(1) {} //test loop

    return;
    }


//void do_some_conversions(float *decominput)
// {
//      int c; //conversion counter

//     for (c = 0; c < BLKSIZE; c++) {
//   b[c] = (int)floorf(decominput[c]);
//   b[c] = (int)ceilf(decominput[c]);
//   b[c] = (int)roundf(decominput[c]);
//   b[c] = (int)truncf(decominput[c]);
//   b[c] = (int)rintf(decominput[c]);
//   b[c] = (int)nearbyintf(decominput[c]);
//   b[c] = (int)decominput[c];
//   b[c] = decominput[c];
//     }
// }

void do_some_conversions(float test)
 {
      //  int c; //conversion counter

      //for (c = 0; c < BLKSIZE; c++) {
   b = (int)floorf(test);
   b = (int)ceilf(test);
   b = (int)roundf(test);
   b = (int)truncf(test);
   b = (int)rintf(test);
   b = (int)nearbyintf(test);
   b = (int)test;
   b = test;
      //}
 }


void TA0_N_ISR_HANDLER( void)
{
      //for (a = 0; a < 16; a++ ) //takes 16 samples for wavelet transform
                    //      {          //runs continuously

                                           ADC14CTL0 |= ADC14ENC + ADC14SC;
//enable taking samples TAKE ONLY SIXTEEN AT A TIME!!!!!

                                             //__bis_SR_register(CPUOFF + GIE); //
LPM0, ADC10_ISR will force exit //
```

```
                                          test = ADC14MEM0;

                                           // decominput[a] = ADC14MEM15;

                                            //waveletdecom(decominput); // run
wavelet transform (more TACCR0 cycles will be added to compensate for this)

                                           // TA0CCR0 += 1300;
                                            //P1OUT |= BIT1; //test LED1
                    //    }
                                  //      a = a + 1;


                                  //      while(1) {} //test loop

        //  if (a == BLKSIZE){
        //     do_some_conversions(decominput);
               do_some_conversions(test);
               Drive_DAC(b); //send output to DAC *?

        //     a = 1;
        //  }
        //  else {

        //  }
                   //do_some_conversions(decominput)
                   //Drive_DAC(b); //send output to DAC *?
          TA0CCR0 += 50000;
          TA0CCTL0 &= ~CCIFG;
}



////////////see what ADC interrupt doesss????????????????

void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    CSCTL0 |= DCOEN + DCORSEL_4; //activate DCOCLK, SMCLK = 24 MHz

     //ADC14 uses MODCLK?

    CSCTL1 |= SELS_3 + DIVS_0; //select SMCLK, no division

    CSSTAT |= SMCLK_ON + DCO_ON; //activate SMCLK and DCO clock (status)

     ADC14CTL0 = ADC14SHT0_2 + ADC14ON; //+ ADC14IE15; // ADC14ON, interrupt
enabled, 16 clock samples

     ADC14CTL1 = ADC14RES_3; //sampling resolution, 14-bit conversion.

     ADC14MCTL0 = ADC14INCH_1; // input A1, P5.4
     P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK

     UCB0CTLW0 |= UCSWRST; //put state machine in reset momentarily

     UCB0CTLW0 |= UCCKPL + UCMSB + UCMST + UCSYNC;

       UCB0CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

       UCB0BR0 |= 0x00; //no SMCLK division

       UCB0BR1 |= 0x00;

       UCB0CTL1 &= ~UCSWRST;  //initialize USCI state machine

       P6DIR |= BIT4; //set P6.4 for /CS for external DAC

      // P1DIR |= BIT6; //set P1.6 output


      // P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK

      // TA0CCTL0 = CCIE;                             // CCR0 interrupt
enabled
      // TA0CCR0 = 12000;
      // TA0CTL = TASSEL_2 + MC_2;                   // SMCLK, contmode,
time_A0 control
           //     _BIS_SR(LPM0_bits + GIE);


       __enable_interrupt(); // enable all interrupts

       NVIC_ISER0 = 1 << ((INT_TA0_N - 16) & 31); //NVIC interrupt declaration

            TA0CCTL0 = CCIE + CCIS_0;                            // CCR0
interrupt enabled, CCIxA input signal for TAxCCR0
            TA0CCR0 = 50000;
            TA0CTL = TASSEL_2 + MC_2 + TAIE;                  // SMCLK,
contmode, time_A0 control, timer_A0 interrupt enabled

            TA0CCTL0 &= ~CAP; //compare mode

            TA0R = 0;  ///timer counter set to zero

            a = 1;


            while(1) {} //test loop

        //   for (a = 0; a < 16; a++ ) //takes 16 samples for wavelet
transform
             //     {          //runs continuously

             //                 ADC14CTL0 |= ADC14ENC + ADC14SC; //enable
taking samples TAKE ONLY SIXTEEN AT A TIME!!!!!
```

```
                                 //__bis_SR_register(CPUOFF + GIE); // LPM0,
ADC10_ISR will force exit //

             //                    decominput[a] = ADC14MEM15;

                                  //waveletdecom(decominput); // run wavelet
transform (more TACCR0 cycles will be added to compensate for this)

             //                    TA0CCR0 += 1300;
                                  //P1OUT |= BIT1; //test LED1
             //     }

        //   do_some_conversions(decominput);
        //   Drive_DAC(b); //send output to DAC *?
 // return 0;
}
```

Figure A.6: Microcontroller code input and output test program progress.

```
#include "msp.h"
#include <math.h>
#include <stdio.h>

//#define BLKSIZE 16 /* input block size--must be a power of two */
int a; //sample counter
//float decominput[BLKSIZE]; //input to decomposition filter
//int b[BLKSIZE]; //float to int conversion
int b;
//extern void TA0_N_ISR_HANDLER(void);
float test;

void Drive_DAC(int b){
    unsigned int DAC_Word = 0;
 //   int d; //counter

  //  for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x9000) | (b & 0x0FFF);   //CHANGED FROM X1000 TO X3000
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    P6OUT &= ~BIT4;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
    UCB0TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB0TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB0IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    __delay_cycles(30);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P6OUT |= BIT4;
    // Set P1.4 (drive /CS high on DAC)
 //   }


   // P1OUT ^= BIT0;  //test, turn on LED
   // __delay_cycles(30000);  //test
   // P1OUT ^= BIT0;  //test, turn off LED

   // P1OUT |= BIT6;

  //  while(1) {} //test loop

    return;
    }
```

```
void do_some_conversions(float test)
 {
       //  int c; //conversion counter

       //for (c = 0; c < BLKSIZE; c++) {
   b = (int)floorf(test);
   b = (int)ceilf(test);
   b = (int)roundf(test);
   b = (int)truncf(test);
   b = (int)rintf(test);
   b = (int)nearbyintf(test);
   b = (int)test;
   b = test;
       //}
 }

void main(void)
{

 //   WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    volatile unsigned int i;

    CSCTL0 |= DCOEN + DCORSEL_4; //activate DCOCLK, SMCLK = 24 MHz

     //ADC14 uses MODCLK?

    CSCTL1 |= SELS_3 + DIVS_0; //select SMCLK, no division

    CSSTAT |= SMCLK_ON + DCO_ON; //activate SMCLK and DCO clock (status)

     ADC14CTL0 = ADC14SHT0_2 + ADC14ON + ADC14SHP; //+ ADC14IE15; // ADC14ON,
interrupt enabled, 16 clock samples

     ADC14CTL1 = ADC14RES_3; //sampling resolution, 14-bit conversion.

     ADC14MCTL0 = ADC14INCH_1; // input A1, P5.4

     ADC14IER0 |= ADC14IE0;                    // Enable ADC conv complete
interrupt

         SCB_SCR &= ~SCB_SCR_SLEEPONEXIT;           // Wake up on exit from
ISR


     P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK

     UCB0CTLW0 |= UCSWRST; //put state machine in reset momentarily

     UCB0CTLW0 |= UCCKPL + UCMSB + UCMST + UCSYNC;

       UCB0CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

       UCB0BR0 |= 0x00; //no SMCLK division
       UCB0BR1 |= 0x00;

       UCB0CTL1 &= ~UCSWRST;  //initialize USCI state machine

       P6DIR |= BIT4; //set P6.4 for /CS for external DAC

      // P1DIR |= BIT6; //set P1.6 output


      // P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK

      // TA0CCTL0 = CCIE;                            // CCR0 interrupt
enabled
      // TA0CCR0 = 12000;
      // TA0CTL = TASSEL_2 + MC_2;                  // SMCLK, contmode,
time_A0 control
           //     _BIS_SR(LPM0_bits + GIE);
     // a = 0;

       __enable_interrupt(); // enable all interrupts

       NVIC_ISER0 = 1 << ((INT_ADC14 - 16) & 31);         // Enable ADC
interrupt in NVIC module

       while (1)
           {
             for (i = 20000; i > 0; i--);            // Delay
             ADC14CTL0 |= ADC14ENC | ADC14SC;        // Start
sampling/conversion

             __sleep();

       //      __bis_SR_register(LPM0_bits | GIE);     // LPM0, ADC14_ISR will
force exit
             __no_operation();                       // For debugger
           }

}

void ADC14IsrHandler(void) {

       //DISABLE INTERRUPT?????? UNTIL driVE_dac IS DONE???

       test = ADC14MEM0;  //added for testing
   //   decominput[a] = ADC14MEM0;   //take samples

   //   a++;
        do_some_conversions(test);
                      Drive_DAC(b); //send output to DAC *?
}
```

Figure A.7: Microcontroller code input and output test program progress after changing from TIMERA interrupt routine to ADC14 interrupt routine.

```
#include "msp.h"
#include <math.h>
#include <stdio.h>
//#include "driverlib.h"

//#define BLKSIZE 16 /* input block size--must be a power of two */
int a; //sample counter
//float decominput[BLKSIZE]; //input to decomposition filter
//int b[BLKSIZE]; //float to int conversion
//int b;
//extern void TA0_N_ISR_HANDLER(void);
float test;

#define SRSIZE 49 /*size of the decomposition xform shift register */
#define BLKSIZE 16 /* input block size--must be a power of two */ //or
16??????
#define NUMFILT 2 /* number of lattice filters required */
#define HALFORD 3 /* one half of the order of the filter */
#define SRSIZEREC 26 /*size of the resonstruction xform shift register */
#define BUTTON1 BIT4 //button bit P2.4
#define BUTTON2 BIT0 //button bit P3.0? P4.0 now

float decominput[BLKSIZE]; //input to decomposition filter
float decomout[BLKSIZE]; //output of decomposition filter
float reconout[BLKSIZE]; //output of reconstruction filter
unsigned int b[BLKSIZE]; //float to int conversion
float FST_signal[BLKSIZE];
float RST_signal[BLKSIZE];

//#define number_of_decomposition_levels 4;
//#define number_of_samples 16;
int number_of_samples, level, q, number_of_decomposition_levels, e,
total_number_of_samples;
int number_of_reconstruction_levels;
unsigned int DAC_Word;
int change;
int j;

void waveletdecom(float *decominput); /*declaring decomposition filter */
void recon(float *decomout);  /*declaring reconstruction filter */

//int _system_pre_init( void )
//      {
 //       WDTCTL = WDTPW | WDTHOLD;

 //       return 1;
 //     }

void Drive_DAC(unsigned int *b) {
   // unsigned int DAC_Word = 0;
       DAC_Word = 0;

       int i;

    int d; //counter
    for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x3000) | (b[d] & 0x0FFF);   //CHANGED FROM X1000 TO X3000
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    P5OUT &= ~BIT0;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
    UCB2TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB2TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    __delay_cycles(150);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P5OUT |= BIT0;
    // Set P1.4 (drive /CS high on DAC)
   // return;


  //  while(1) {} //test loop

    for (i = 200; i > 0; i--);            // Delay

   }


   // P1OUT ^= BIT0;  //test, turn on LED
   // __delay_cycles(30000);  //test
   // P1OUT ^= BIT0;  //test, turn off LED

   // P1OUT |= BIT6;

  //  while(1) {} //test loop

    return;
    }

void do_some_conversions(float *decomout)
 {
         int c; //conversion counter

       for (c = 0; c < BLKSIZE; c++) {
   b[c] = (int)floorf(decomout[c]);
   b[c] = (int)ceilf(decomout[c]);
   b[c] = (int)roundf(decomout[c]);
   b[c] = (int)truncf(decomout[c]);
   b[c] = (int)rintf(decomout[c]);
   b[c] = (int)nearbyintf(decomout[c]);
   b[c] = (int)decomout[c];

   b[c] = (unsigned int)decomout[c]; //make it unsigned

   b[c] = decomout[c];
       }
 }


void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer

    volatile unsigned int i;

    CSKEY = 0x695A;  // unlock CS module for register access

    CSCTL0 = 0; // reset DCO settings

    CSCTL0 |= DCOEN + DCORSEL_3; ///+ 0xFFFFFFFFFFFFFC5C;// +
DCOTUNE(0xFFFFFFFFFFFFFC5C); //activate DCOCLK, SMCLK = 24 MHz


   // CS_setFrequency(20000000);


  //  __I uint32_t rDCOIR_FCAL_RSEL04; /* DCO IR mode: Frequency calibration
for DCORSEL 0 to 4 */
  //   __I uint32_t rDCOIR_FCAL_RSEL5; /* DCO IR mode: Frequency calibration
for DCORSEL 5 */
  //   __I uint32_t rDCOIR_MAXPOSTUNE_RSEL04; /* DCO IR mode: Max Positive
Tune for DCORSEL 0 to 4 */
  //   __I uint32_t rDCOIR_MAXNEGTUNE_RSEL04; /* DCO IR mode: Max Negative
Tune for DCORSEL 0 to 4 */
  //   __I uint32_t rDCOIR_MAXPOSTUNE_RSEL5; /* DCO IR mode: Max Positive Tune
for DCORSEL 5 */
  //   __I uint32_t rDCOIR_MAXNEGTUNE_RSEL5; /* DCO IR mode: Max Negative Tune
for DCORSEL 5 */
  //   __I uint32_t rDCOIR_CONSTK_RSEL04; /* DCO IR mode: DCO Constant (K) for
DCORSEL 0 to 4 */
  //   __I uint32_t rDCOIR_CONSTK_RSEL5; /* DCO IR mode: DCO Constant (K) for
DCORSEL 5 */
   // FFFFFFFFFFFFFC5C

  //  0000001110100100
     //ADC14 uses MODCLK?

    CSCTL1 |= SELA_2 | SELS_3 | SELM_3; //SELS_3 + DIVS_0; //select SMCLK, no
division
```

```
    CSSTAT |= SMCLK_ON + DCO_ON; //activate SMCLK and DCO clock (status)

  // while(1) {} //test loop

    CSKEY = 0;

    P5SEL1 |= BIT5;                          // Configure P5.4 for ADC (A1 to
A0!)
                P5SEL0 |= BIT5;


     ADC14CTL0 = ADC14SHT0_2 + ADC14ON + ADC14SHP + ADC14SSEL_4; //+
ADC14IE15; // ADC14ON, interrupt enabled, 16 clock samples

     ADC14CTL1 = ADC14RES_2; //sampling resolution, 14-bit conversion. (now
12-bit!)

     ADC14MCTL0 = ADC14INCH_0; // input A1, P5.4 (P5.5 now!)

     ADC14IER0 |= ADC14IE0;                    // Enable ADC conv complete
interrupt

         SCB_SCR &= ~SCB_SCR_SLEEPONEXIT;           // Wake up on exit from
ISR


    // P3SEL0 |= BIT6 + BIT5; //set P3.6 output for SIMO and P3.5 for UCB0CLK

     UCB2CTLW0 |= UCSWRST; //put state machine in reset momentarily

     UCB2CTLW0 |= UCCKPL + UCMSB + UCMST + UCSYNC;

     UCB2CTLW0 &= ~UCSLA10 + ~UCMM;  //no multi-master mode, address slave
with 7-bit address

       UCB2CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

       UCB2BR0 |= 0x00; //no SMCLK division

       UCB2BR1 |= 0x00;

       P3SEL0 |= BIT6 + BIT5; //set P3.6 output for SIMO and P3.5 for UCB0CLK


       UCB2CTLW0 &= ~UCSWRST;  //initialize USCI state machine

       P5DIR |= BIT0; //set P5.0 for /CS for external DAC

      // P3DIR |= BIT5; //set P3.5, output


      // P1SEL0 = BIT6 + BIT5; //set P1.6 output for SIMO and P1.5 for UCB0CLK

     //  TA0CCTL0 = CCIE;                             // CCR0 interrupt
enabled
```

```
    //   TA0CCR0 = 12000;
    //   TA0CTL = TASSEL_2 + MC_2;                  // SMCLK, contmode,
time_A0 control
           //     _BIS_SR(LPM0_bits + GIE);

   //    NVIC_ISER0 = 1 << ((INT_TA0_N - 16) & 31); //NVIC interrupt
declaration

           /*  TA0CCTL0 = CCIE + CCIS_0;                            // CCR0
interrupt enabled, CCIxA input signal for TAxCCR0
             TA0CCR0 = 100;
             TA0CTL = TASSEL_2 + MC_2; //+ TAIE;                    // SMCLK,
contmode, time_A0 control, timer_A0 interrupt enabled

             TA0CCTL0 &= ~CAP; //compare mode

             TA0R = 0;  ///timer counter set to zero */

      a = 0;

      change = 1;

       __enable_interrupt(); // enable all interrupts

       NVIC_ISER0 = 1 << ((INT_ADC14 - 16) & 31);         // Enable ADC
interrupt in NVIC module

      // NVIC_ISER0 = 1 << ((INT_TA0_N - 16) & 31); //NVIC interrupt
declaration for TIMERA interrupt

       while (1)
           {
          //   for (i = 200; i > 0; i--);            // Delay
             ADC14CTL0 |= ADC14ENC | ADC14SC;        // Start
sampling/conversion

                   if (a == 16){                      // if all 16 samples are
taken, wavelet transform them.

                      if (change == 0){              //deconstruct and
reconstruct the signal if change = 0

                             waveletdecom(decominput);
                             recon(decomout);
                             do_some_conversions(reconout); //convert float to
int
                             Drive_DAC(b); //send output to DAC

                      }
                      else {                       // if change isn't one,
take the wavelet transform

                             waveletdecom(decominput);
                             do_some_conversions(decomout); //convert float to
int
```

```
                              Drive_DAC(b); //send output to DAC

                        }

                         a = 0;

                    }
                    else {

                    }

              // __sleep();

             //  __bis_SR_register(LPM0_bits | GIE);     // LPM0, ADC14_ISR will
force exit
             //  __no_operation();                       // For debugger
             }

}

void waveletdecom(float *decominput) {


       //int number_of_samples, level, q, number_of_decomposition_levels, e;

       number_of_samples = 16;
       number_of_decomposition_levels = 4;
//     float FST_signal[number_of_samples];
//     float Signal[number_of_samples];
//     float decomout[number_of_samples];

//     while(1) {} //test loop

       //e = 0;

       //while(1) {} //test loop


       for (level = 1; level <= number_of_decomposition_levels; level++ ){
              number_of_samples = number_of_samples >> 1; //downsampling by 2

              for (q = 0; q < number_of_samples; q++ ){
                     FST_signal[q] = ((decominput[q << 1] + decominput[(q << 1)
+ 1])); //approximation coefficients, A
                     FST_signal[q] = FST_signal[q]*(1.00/2.00);
                     FST_signal[q + number_of_samples] = (decominput[q << 1] -
decominput[(q << 1) + 1]); // detail coefficients, D


             // for (f = 0; f <= e; e++){
             //       decomout[q + e] = FST_signal[q + number_of_samples]; //add
detail coefficients in order into output
             // }
```

```
                }

                //e = e + (number_of_samples >> 1);  //counter for decomout

                for (q = 0; q < number_of_samples; q++){
               decominput[q] = FST_signal[q]; //A from current level are input
signal for the next level
                }


        //      while(1) {} //test loop


        }

    //  decomout[number_of_samples] = FST_signal[0]; //add final appr0ximation
coefficient, output is done
        for (q = 0; q < BLKSIZE; q++){  //get output
           decomout[q] = FST_signal[q];
        }

  //   do_some_conversions(decomout); //convert float to int
  //   Drive_DAC(b); //send output to DAC


        return ;
}

void recon(float *decomout){

        total_number_of_samples = 16;
        number_of_samples = 16;
        number_of_reconstruction_levels = 4;

        number_of_samples = total_number_of_samples >>
number_of_reconstruction_levels;

        for (q = 0; q < total_number_of_samples; q++ ){
                RST_signal[q] = decomout[q];
        }

        for (level = 1; level <= number_of_reconstruction_levels; level++){

                for (q = 0; q < number_of_samples; q++){  //reconstruction of
coefficients

                        RST_signal[q << 1] = decomout[q] + ((decomout[q +
number_of_samples] + 1)*(1.00/2.00));
                        RST_signal[(q << 1) + 1] = decomout[q] - ((decomout[q +
number_of_samples])*(1.00/2.00));

                }

                for(q = 0; q < number_of_samples << 1; q++){ // coefficients from
current level are input signal for next level
```

```
                        decomout[q] = RST_signal[q];
                }

                number_of_samples = number_of_samples << 1; //upsampling by a
factor of 2

        }

        for (q = 0; q < BLKSIZE; q++){
                reconout[q] = RST_signal[q];
        }

        return;
}

void ADC14IsrHandler(void) {

          int i;

        //DISABLE INTERRUPT?????? UNTIL driVE_dac IS DONE???

        //test = ADC14MEM0;  //added for testing
       decominput[a] = ADC14MEM0;   //take samples

       a++;

       for (i = 200; i > 0; i--);             // Delay

//       do_some_conversions(test);
//                     Drive_DAC(b); //send output to DAC *?
}

/* void TA0_N_ISR_HANDLER( void){   //output delay timer interrupt, each
output is within the same number of cycles

        int i;

        for (j = 0; j < BLKSIZE; j++) {

                Drive_DAC(b[j]); //send output to DAC
                for (i = 200; i > 0; i--);             // Delay
        }


        TA0CCR0 += 100000;
        TA0CCTL0 &= ~CCIFG;
} */
```

Figure A.8: Microcontroller code progress after decomposition and reconstruction rework.

```
/* DriverLib Includes */
#include "driverlib.h"

/* Standard Includes */
#include <stdint.h>

#include <stdbool.h>

#include <SPI.h>

volatile uint8_t TXData = 1;
//volatile uint8_t RXData = 0;

/* UART Configuration Parameter. These are the configuration parameters to
 * make the eUSCI A UART module to operate with a 115200 baud rate. These
 * values were calculated using the online calculator that TI provides
 * at:
 * http://software-
dl.ti.com/msp430/msp430_public_sw/mcu/msp430/MSP430BaudRateConverter/index.htm
l
 */

int b;
int c;

/* SPI Master Configuration Parameter */
const eUSCI_SPI_MasterConfig spiMasterConfig =
{
        EUSCI_B_SPI_CLOCKSOURCE_SMCLK,             // SMCLK Clock Source
        12000000,                                   // SMCLK = DCO = 12MHZ
        16000000,                                    // SPICLK = 16MHZ
        EUSCI_B_SPI_MSB_FIRST,                     // MSB First
        EUSCI_B_SPI_PHASE_DATA_CHANGED_ONFIRST_CAPTURED_ON_NEXT,    // Phase
        EUSCI_B_SPI_CLOCKPOLARITY_INACTIVITY_HIGH, // High polarity
        EUSCI_B_SPI_3PIN                           // 3Wire SPI Mode

             //__delay_cycles(30000);

      //     while(1) {}
};

 void Drive_DAC(unsigned int /* *  */ Value) {
      /* Polling to see if the TX buffer is ready */

      // Value=Value & 0xfff;
      //  Value = Value | 0xc000; // Write DAC A
      //  P5OUT &= ~BIT0; // Drive CS low
        //digitalWrite(SS_PIN,LOW); // Drive CS low
      //  UCB2TXBUF=( (Value >> 8) & 0xff); // write high byte
      //  while(UCB2STATW & BIT0); // wait while SPI busy
      //  UCB2TXBUF=  ( Value  & 0xff); // write low byte
      //  while(UCB2STATW & BIT0); // wait while SPI busy
        //digitalWrite(SS_PIN,HIGH); // Drive CS High
      //  P5OUT |= BIT0; // Drive CS High
```

```
          P5OUT &= ~BIT0; // Drive CS low
          while
(!(SPI_getInterruptStatus(EUSCI_B2_BASE,EUSCI_B_SPI_TRANSMIT_INTERRUPT)));
          //  Transmitting data to slave
        SPI_transmitData(EUSCI_B2_BASE, TXData);
        P5OUT |= BIT0; // Drive CS High


    return;
   }

int main(void)
{
    volatile uint32_t ii;

    /* Halting WDT  */
    WDT_A_holdTimer();

    P5DIR |= BIT0; //set P5.0 for /CS for external DAC

    /* Selecting P3.5 P3.6 in SPI mode */
    GPIO_setAsPeripheralModuleFunctionInputPin(GPIO_PORT_P3,
            GPIO_PIN5 | GPIO_PIN6, GPIO_PRIMARY_MODULE_FUNCTION);

    /* Configuring SPI in 3wire master mode */
    SPI_initMaster(EUSCI_B2_BASE, &spiMasterConfig);

    /* Enable SPI module */
    SPI_enableModule(EUSCI_B2_BASE);

    /* Enabling interrupts */
   /* SPI_enableInterrupt(EUSCI_B2_BASE, EUSCI_B_SPI_RECEIVE_INTERRUPT);
    Interrupt_enableInterrupt(INT_EUSCIB2);
    Interrupt_enableSleepOnIsrExit(); */
    TXData = 0x01;

    c = 0 ;

            while(1) {

             for (b = 0; b < 100; b++ ) {

                    Drive_DAC(c); //send output to DAC *

                    if ( b < 50 ) {

                           TXData = 0x05;

                    }

                    else if (b >= 50 ) {

                           TXData = 0x10;
                    }

                    else {

                    }


             }


            }


    PCM_gotoLPM0();
    __no_operation();
       }
```

Figure A.9: Microcontroller program written in “MSP432DriverLib” designed to output a square wave to an external DAC.

```
#include "msp.h"
//#include "driverlib.h"

int b;
int c;
unsigned int DAC_Word;

void Drive_DAC(unsigned int /* *  */ b) {
   // unsigned int DAC_Word = 0;
       DAC_Word = 0;

       int i;

  //  int d; //counter

  //  for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x1000) | (b /*[d] */ & 0x0FFF);   //CHANGED FROM X1000 TO
X3000
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    //P5OUT |= BIT0;
    P5OUT &= ~BIT0;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
   // __delay_cycles(1500);
   // for (i = 50; i > 0; i--);            // Delay

    UCB2TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB2TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    for (i = 30; i > 0; i--);            // Delay
   // __delay_cycles(1500);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P5OUT |= BIT0;
   // P5OUT &= ~BIT0;
    // Set P1.4 (drive /CS high on DAC)
   // return;


  //  while(1) {} //test loop

   // for (i = 200; i > 0; i--);            // Delay
```

```
 //  }

    return;
    }



void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer



    volatile unsigned int i;

        CS->KEY = 0x695A;  // unlock CS module for register access

        CS->CTL0 = 0; // reset DCO settings

        CS->CTL0 |= CS_CTL0_DCOEN + CS_CTL0_DCORSEL_3 + 0x00000600;// +
DCOTUNE(0xFFFFFFFFFFFFFC5C); //activate DCOCLK, SMCLK = 24 MHz

        CS->CTL1 |= CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3; //SELS_3
+ DIVS_0; //select SMCLK, no division

        //CS->STAT |= CS_STAT_SMCLK_ON + CS_STAT_DCO_ON; //activate SMCLK and
DCO clock (status)

        CS->KEY = 0;

        UCB2CTLW0 |= UCSWRST; //put state machine in reset momentarily

        UCB2CTLW0 |= UCCKPL + UCMST + UCSYNC;

        UCB2CTLW0 &= ~UCMSB;

        UCB2CTLW0 |= UCMSB;

      //  UCB2CTLW0 &= ~UCSLA10 + ~UCMM;  //no multi-master mode, address
slave with 7-bit address

        UCB2CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

        UCB2BR0 |= 0x00; //no SMCLK division

        UCB2BR1 |= 0x00;

        P3SEL0 |= BIT6 + BIT5; //set P3.6 output for SIMO and P3.5 for UCB0CLK

        UCB2CTLW0 &= ~UCSWRST;  //initialize USCI state machine

        P5DIR |= BIT0; //set P5.0 for /CS for external DAC
        P5OUT &= ~BIT0; //start at zero
        c = 0 ;
        while(1) {
             for (b = 0; b < 1000; b++ ) {
                    Drive_DAC(c); //send output to DAC *
                    if ( b < 500 ) {
                           c = 4000;
                    }
                    else if (b >= 500 ) {
                           c = 1000;
                    }
                    else {
                    }

             }

        }

}
```

Figure A.10: Microcontroller code used for creating a square waveform from an external DAC.

```
#include "msp.h"
#include <math.h>
#include <stdio.h>

//#include "msp.h"
//#include "driverlib.h"
int a;
//int b;
int c;
//unsigned int DAC_Word;

#define SRSIZE 49 /*size of the decomposition xform shift register */
#define BLKSIZE 16 /* input block size--must be a power of two */ //or
16??????
#define NUMFILT 2 /* number of lattice filters required */
#define HALFORD 3 /* one half of the order of the filter */
#define SRSIZEREC 26 /*size of the resonstruction xform shift register */
#define BUTTON1 BIT4 //button bit P2.4
#define BUTTON2 BIT1 //button bit P3.0? P4.0, now P4.1

float decominput[BLKSIZE]; //input to decomposition filter
float decomout[BLKSIZE]; //output of decomposition filter
float reconout[BLKSIZE]; //output of reconstruction filter
 int b[BLKSIZE]; //float to int conversion

float FST_signal[BLKSIZE];
float RST_signal[BLKSIZE];

//#define number_of_decomposition_levels 4;
//#define number_of_samples 16;
int number_of_samples, level, q, number_of_decomposition_levels, e,
total_number_of_samples;
int number_of_reconstruction_levels;
 int DAC_Word;
int change;
int j;

void Drive_DAC( int  * b) {
   // unsigned int DAC_Word = 0;
      DAC_Word = 0;

      int i;

    int d; //counter

    for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x1000) | (b[d] & 0x0FFF);   //CHANGED FROM X1000 TO X3000
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    //P5OUT |= BIT0;
    P5OUT &= ~BIT0;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
```

```
  // __delay_cycles(1500);
  // for (i = 50; i > 0; i--);            // Delay

   UCB2TXBUF = (DAC_Word >> 8);
   // Shift upper byte of DAC_Word
   // 8-bits to right
   while
   (!(UCB2IFG & UCTXIFG));
   // USCI_A0 TX buffer ready?
   UCB2TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
   (DAC_Word & 0x00FF);
   // Transmit lower byte to DAC
   while
   (!(UCB2IFG & UCTXIFG));
   // USCI_A0 TX buffer ready?
   for (i = 5; i > 0; i--);            // Delay
  // __delay_cycles(1500);
   // Delay 150 12 MHz SMCLK periods
   // (12.5 us) to allow SIMO to complete
   P5OUT |= BIT0;
  // P5OUT &= ~BIT0;
   // Set P1.4 (drive /CS high on DAC)
  // return;


 //  while(1) {} //test loop

   for (i = 50; i > 0; i--);            // Delay

  }

   return;
   }

void do_some_conversions(float *decomout)
 {
        int c; //conversion counter

      for (c = 0; c < BLKSIZE; c++) {
  b[c] = (int)floorf(decomout[c]);
  b[c] = (int)ceilf(decomout[c]);
  b[c] = (int)roundf(decomout[c]);
  b[c] = (int)truncf(decomout[c]);
  b[c] = (int)rintf(decomout[c]);
  b[c] = (int)nearbyintf(decomout[c]);
  b[c] = (int)decomout[c];

  b[c] = b[c] + 3000; //add about 2.5 V to keep numbers from negative

 // b[c] = (unsigned int)decomout[c]; //make it unsigned

  b[c] = decomout[c];
      }
 }
```

```
void waveletdecom(float *decominput) {


       //int number_of_samples, level, q, number_of_decomposition_levels, e;

       number_of_samples = 16;
       number_of_decomposition_levels = 4;
//     float FST_signal[number_of_samples];
//     float Signal[number_of_samples];
//     float decomout[number_of_samples];

//     while(1) {} //test loop

       //e = 0;

       //while(1) {} //test loop


       for (level = 1; level <= number_of_decomposition_levels; level++ ){
              number_of_samples = number_of_samples >> 1; //downsampling by 2

              for (q = 0; q < number_of_samples; q++ ){
                     FST_signal[q] = ((decominput[q << 1] + decominput[(q << 1)
+ 1])); //approximation coefficients, A
                     FST_signal[q] = FST_signal[q]*(1.00/2.00);
                     FST_signal[q + number_of_samples] = (decominput[q << 1] -
decominput[(q << 1) + 1]); // detail coefficients, D


            // for (f = 0; f <= e; e++){
            //       decomout[q + e] = FST_signal[q + number_of_samples]; //add
detail coefficients in order into output
            // }


              }

              //e = e + (number_of_samples >> 1);  //counter for decomout

              for (q = 0; q < number_of_samples; q++){
             decominput[q] = FST_signal[q]; //A from current level are input
signal for the next level
              }


       //     while(1) {} //test loop


       }

   //  decomout[number_of_samples] = FST_signal[0]; //add final appr0ximation
coefficient, output is done
        for (q = 0; q < BLKSIZE; q++){  //get output
           decomout[q] = FST_signal[q];
        }
```

```
  //   do_some_conversions(decomout); //convert float to int
  //   Drive_DAC(b); //send output to DAC


       return ;
}

void recon(float *decomout){

       total_number_of_samples = 16;
       number_of_samples = 16;
       number_of_reconstruction_levels = 4;

       number_of_samples = total_number_of_samples >>
number_of_reconstruction_levels;

       for (q = 0; q < total_number_of_samples; q++ ){
              RST_signal[q] = decomout[q];
       }

       for (level = 1; level <= number_of_reconstruction_levels; level++){

              for (q = 0; q < number_of_samples; q++){  //reconstruction of
coefficients

                     RST_signal[q << 1] = decomout[q] + ((decomout[q +
number_of_samples] + 1)*(1.00/2.00));
                     RST_signal[(q << 1) + 1] = decomout[q] - ((decomout[q +
number_of_samples])*(1.00/2.00));

              }

              for(q = 0; q < number_of_samples << 1; q++){ // coefficients from
current level are input signal for next level
                     decomout[q] = RST_signal[q];
              }

              number_of_samples = number_of_samples << 1; //upsampling by a
factor of 2

       }

       for (q = 0; q < BLKSIZE; q++){
              reconout[q] = RST_signal[q];
       }

       return;
}


void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;            // Stop watchdog timer
    volatile unsigned int i;

        CS->KEY = 0x695A;  // unlock CS module for register access

        CS->CTL0 = 0; // reset DCO settings

        CS->CTL0 |= CS_CTL0_DCOEN + CS_CTL0_DCORSEL_3 + 0x00000600;// +
DCOTUNE(0xFFFFFFFFFFFFFC5C); //activate DCOCLK, SMCLK = 24 MHz

        CS->CTL1 |= CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3; //SELS_3
+ DIVS_0; //select SMCLK, no division

        //CS->STAT |= CS_STAT_SMCLK_ON + CS_STAT_DCO_ON; //activate SMCLK and
DCO clock (status)

        CS->KEY = 0;

        UCB2CTLW0 |= UCSWRST; //put state machine in reset momentarily

        UCB2CTLW0 |= UCCKPL + UCMST + UCSYNC;

        UCB2CTLW0 &= ~UCMSB;

        UCB2CTLW0 |= UCMSB;

      //  UCB2CTLW0 &= ~UCSLA10 + ~UCMM;  //no multi-master mode, address
slave with 7-bit address

        UCB2CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

        UCB2BR0 |= 0x00; //no SMCLK division

        UCB2BR1 |= 0x00;

        P3SEL0 |= BIT6 + BIT5; //set P3.6 output for SIMO and P3.5 for UCB0CLK

        UCB2CTLW0 &= ~UCSWRST;  //initialize USCI state machine

        P5DIR |= BIT0; //set P5.0 for /CS for external DAC

        P5OUT &= ~BIT0; //start at zero

        c = 0 ;

        P5SEL1 |= BIT5;                           // Configure P5.4 for ADC
(A1 to A0!)
        P5SEL0 |= BIT5;


        ADC14->CTL0 = ADC14_CTL0_SHT0_2 + ADC14_CTL0_ON + ADC14_CTL0_SHP +
ADC14_CTL0_SSEL_4; //+ ADC14IE15; // ADC14ON, interrupt enabled, 16 clock
samples
```

```
        ADC14->CTL1 = ADC14_CTL1_RES_2; //sampling resolution, 14-bit
conversion. (now 12-bit!)

        ADC14->MCTL[0] = ADC14_MCTLN_INCH_0; // input A1, P5.4 (P5.5 now!)

        ADC14->IER0 |= ADC14_IER0_IE0;                    // Enable ADC conv
complete interrupt

        SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;           // Wake up on exit
from ISR


        P4DIR &= ~BUTTON2;                          // button is an input
        P4OUT |= BUTTON2;                           // pull-up resistor
        P4REN |= BUTTON2;                           // resistor enabled

        P4IES &= ~BUTTON2;                           // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
        P4IFG &= ~BUTTON2; // P2.4 IFG cleared
        P4IE |= BUTTON2; // P2.4 interrupt enabled


        __enable_interrupt(); // enable all interrupts

        NVIC->ISER[0] = 1 << ((ADC14_IRQn) & 31);         // Enable ADC
interrupt in NVIC module

     //   NVIC->ISER[1] = 1 << ((PORT4_IRQn) & 31);

        NVIC->ISER[1] = 1 << ((PORT2_IRQn) & 31);


        change = 0;

        while(1) {

             ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;        // Start
sampling/conversion

             /* while(1){                  // stop operation until button is
pushed again

                          if(P2IN == BUTTON1){

                                //for (i = 1000; i > 0; i--);            //
Delay

                                break;
                          }
                          else{

                          }
```

```
                        } */
                //      P4IFG &= ~BUTTON2; // P3.0 IFG cleared

        //      while(1){


        /*              if(P2IN == BUTTON1){  //change output if button (P2.4) is
pressed

                                if (change == 1){
                                        change = 0;
                                }
                                else{
                                        change = 1;
                                }

                                for (i = 1000; i > 0; i--);             // Delay

                        }
                        else {

                        } */

                if (a == 16){                           // if all 16 samples are
taken, wavelet transform them.

                                        if (change == 0){
//deconstruct and reconstruct the signal if change = 0

                                                waveletdecom(decominput);
                                                recon(decomout);
                                                do_some_conversions(reconout);
//convert float to int
                                                Drive_DAC(b); //send output to DAC

                                        }
                                        else {                          // if change
isn't one, take the wavelet transform

                                                waveletdecom(decominput);
                                                do_some_conversions(decomout);
//convert float to int
                                                Drive_DAC(b); //send output to DAC

                                        }

                                          a = 0;

                                }
                                else {

                                }
```

```
                            /*    if (a == 16){                      // if all
16 samples are taken, wavelet transform them.

                                   //  waveletdecom(decominput);

                                     do_some_conversions(decominput);
//convert float to int
                                     Drive_DAC(b); //send output to DAC
                                     a = 0;

                                }
                                else {

                                } */



             }



}
   //  ADC14_IRQHandler


void ADC14IsrHandler(void) {

       int i;

       //DISABLE INTERRUPT?????? UNTIL driVE_dac IS DONE???

       //test = ADC14MEM0;  //added for testing
        decominput[a] = ADC14->MEM[0];   //take samples

        a++;

        for (i = 50; i > 0; i--);            // Delay

        //    do_some_conversions(test);
        //                  Drive_DAC(b); //send output to DAC *?
}

//void INT_PORT2_HANDLER(void){




//}
/*
void INT_PORT4_Handler(void ){

      //while(1){ }

      int i;
```

```
	for (i = 5000; i > 0; i--);		// Delay

	// while(1){			// stop operation until button is pushed
again

	//	if(P4IN == BUTTON2){
	//		break;
	//	}
	//	else{

	//	}

	//
//	P4IFG &= ~BUTTON2; // P3.0 IFG cleared

	//if(P4IN == BUTTON2){  //change output if button (P2.4) is pressed

				if (change == 1){
					change = 0;
				}
				else{
					change = 1;
				}

				for (i = 1000; i > 0; i--);		//
Delay

			//}
			//else {

	    //		}

	P4IFG &= ~BUTTON2; // P4.0 IFG cleared

}
 */
void INT_PORT2_Handler(void) {

	while(1){ }
      int i;

	 while(1){			// stop operation until button is pushed
again

		for (i = 10000000; i > 0; i--);		// Delay


				if(P2IN == BUTTON1){

					//for (i = 1000; i > 0; i--);
// Delay


					break;
				}
```

```
                                        else{

                                        }

                                }
        P2IFG &= ~BUTTON1; // P2.4 IFG cleared


}
```

Figure A.11: Code for outputting wavelet transform or reconstructed signal with working external DAC.

```
#include "msp.h"
#include <math.h>
#include <stdio.h>

//#include "msp.h"
//#include "driverlib.h"
int a;
//int b;
int c;
//unsigned int DAC_Word;

#define SRSIZE 49 /*size of the decomposition xform shift register */
#define BLKSIZE 16 /* input block size--must be a power of two */ //or
16??????
#define NUMFILT 2 /* number of lattice filters required */
#define HALFORD 3 /* one half of the order of the filter */
#define SRSIZEREC 26 /*size of the resonstruction xform shift register */
#define BUTTON1 BIT7 //button bit P2.4, P2.7 now
#define BUTTON2 BIT1 //button bit P3.0? P4.0, now P4.1, P4.2

float decominput1[BLKSIZE]; //input to decomposition filter
float decominput2[BLKSIZE];
float decominput[BLKSIZE];

float decomout[BLKSIZE]; //output of decomposition filter
float reconout[BLKSIZE]; //output of reconstruction filter
 int b[BLKSIZE]; //float to int conversion

float FST_signal[BLKSIZE];
float RST_signal[BLKSIZE];

//#define number_of_decomposition_levels 4;
//#define number_of_samples 16;
int number_of_samples, level, q, number_of_decomposition_levels, e,
total_number_of_samples;
int number_of_reconstruction_levels;
 int DAC_Word;
int change;
int j;

int k;

void Drive_DAC( int  * b) {
   // unsigned int DAC_Word = 0;
      DAC_Word = 0;

      int i;

    int d; //counter

    for (d = 0; d < BLKSIZE; d++) {
    DAC_Word = (0x1000) | (b[d] & 0x0FFF);   //CHANGED FROM X1000 TO X3000
    // 0x1000 sets DAC for Write
    // to DAC, Gain = 2, /SHDN = 1
    // and put 12-bit level value
    // in low 12 bits.
    //P5OUT |= BIT0;
    P5OUT &= ~BIT0;
    // Clear P1.4 (drive /CS low on DAC)
    // Using a port output to do this for now
   // __delay_cycles(1500);
   // for (i = 50; i > 0; i--);            // Delay

    UCB2TXBUF = (DAC_Word >> 8);
    // Shift upper byte of DAC_Word
    // 8-bits to right
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    UCB2TXBUF = (unsigned char) //output at P1.6 ???? UCB0SIMO
    (DAC_Word & 0x00FF);
    // Transmit lower byte to DAC
    while
    (!(UCB2IFG & UCTXIFG));
    // USCI_A0 TX buffer ready?
    for (i = 5; i > 0; i--);            // Delay
   // __delay_cycles(1500);
    // Delay 150 12 MHz SMCLK periods
    // (12.5 us) to allow SIMO to complete
    P5OUT |= BIT0;
   // P5OUT &= ~BIT0;
    // Set P1.4 (drive /CS high on DAC)
   // return;


  //  while(1) {} //test loop

    for (i = 50; i > 0; i--);            // Delay

   }

    return;
    }

void do_some_conversions(float *decomout)
 {
        int c; //conversion counter

      for (c = 0; c < BLKSIZE; c++) {
   b[c] = (int)floorf(decomout[c]);
   b[c] = (int)ceilf(decomout[c]);
   b[c] = (int)roundf(decomout[c]);
   b[c] = (int)truncf(decomout[c]);
   b[c] = (int)rintf(decomout[c]);
   b[c] = (int)nearbyintf(decomout[c]);
   b[c] = (int)decomout[c];

   b[c] = b[c] + 2000; //add about 2.5 V to keep numbers from negative

  // b[c] = (unsigned int)decomout[c]; //make it unsigned
```

```
   b[c] = decomout[c];
       }
 }

void waveletdecom(float *decominput) {


       //int number_of_samples, level, q, number_of_decomposition_levels, e;

       number_of_samples = 16;
       number_of_decomposition_levels = 4;
//     float FST_signal[number_of_samples];
//     float Signal[number_of_samples];
//     float decomout[number_of_samples];

//     while(1) {} //test loop

       //e = 0;

       //while(1) {} //test loop


       for (level = 1; level <= number_of_decomposition_levels; level++ ){
              number_of_samples = number_of_samples >> 1; //downsampling by 2

              for (q = 0; q < number_of_samples; q++ ){
                     FST_signal[q] = ((decominput[q << 1] + decominput[(q << 1)
+ 1]))*(1.000/1.414); //approximation coefficients, A
                     //FST_signal[q] = FST_signal[q]*(1.00/2.00);
                     FST_signal[q + number_of_samples] = (decominput[q << 1] -
decominput[(q << 1) + 1])*(1.000/1.414); // detail coefficients, D


            // for (f = 0; f <= e; e++){
            //       decomout[q + e] = FST_signal[q + number_of_samples]; //add
detail coefficients in order into output
            // }


              }

              //e = e + (number_of_samples >> 1);  //counter for decomout

              for (q = 0; q < number_of_samples; q++){
             decominput[q] = FST_signal[q]; //A from current level are input
signal for the next level
              }


       //     while(1) {} //test loop


       }
   //  decomout[number_of_samples] = FST_signal[0]; //add final appr0ximation
coefficient, output is done
       for (q = 0; q < BLKSIZE; q++){  //get output
         decomout[q] = FST_signal[q];
       }

  //   do_some_conversions(decomout); //convert float to int
  //   Drive_DAC(b); //send output to DAC


       return ;
}

void recon(float *decomout){

       total_number_of_samples = 16;
       number_of_samples = 16;
       number_of_reconstruction_levels = 4;

       number_of_samples = total_number_of_samples >>
number_of_reconstruction_levels;

       for (q = 0; q < total_number_of_samples; q++ ){
              RST_signal[q] = decomout[q];
       }

       for (level = 1; level <= number_of_reconstruction_levels; level++){

              for (q = 0; q < number_of_samples; q++){  //reconstruction of
coefficients

                     RST_signal[q << 1] = decomout[q]*(1.000/1.414) +
((decomout[q + number_of_samples] + 1)*(1.000/1.414));
                     RST_signal[(q << 1) + 1] = decomout[q]*(1.000/1.414) -
((decomout[q + number_of_samples])*(1.000/1.414));

              }

              for(q = 0; q < number_of_samples << 1; q++){ // coefficients from
current level are input signal for next level
                     decomout[q] = RST_signal[q];
              }

              number_of_samples = number_of_samples << 1; //upsampling by a
factor of 2

       }

       for (q = 0; q < BLKSIZE; q++){
              reconout[q] = RST_signal[q];
       }

       return;
}
```

```
void main(void)
{

    WDTCTL = WDTPW | WDTHOLD;           // Stop watchdog timer


    volatile unsigned int i;

        CS->KEY = 0x695A;  // unlock CS module for register access

        CS->CTL0 = 0; // reset DCO settings

        CS->CTL0 |= CS_CTL0_DCOEN + CS_CTL0_DCORSEL_3 + 0x00000600;// +
DCOTUNE(0xFFFFFFFFFFFFFC5C); //activate DCOCLK, SMCLK = 24 MHz

        CS->CTL1 |= CS_CTL1_SELA_2 | CS_CTL1_SELS_3 | CS_CTL1_SELM_3; //SELS_3
+ DIVS_0; //select SMCLK, no division

        //CS->STAT |= CS_STAT_SMCLK_ON + CS_STAT_DCO_ON; //activate SMCLK and
DCO clock (status)

        CS->KEY = 0;

        UCB2CTLW0 |= UCSWRST; //put state machine in reset momentarily

        UCB2CTLW0 |= UCCKPL + UCMST + UCSYNC;

        UCB2CTLW0 &= ~UCMSB;

        UCB2CTLW0 |= UCMSB;

      //  UCB2CTLW0 &= ~UCSLA10 + ~UCMM;  //no multi-master mode, address
slave with 7-bit address

        UCB2CTLW0 |= UCSSEL_2; /* SMCLK (input clock?) */

        UCB2BR0 |= 0x00; //no SMCLK division

        UCB2BR1 |= 0x00;

        P3SEL0 |= BIT6 + BIT5; //set P3.6 output for SIMO and P3.5 for UCB0CLK

        UCB2CTLW0 &= ~UCSWRST;  //initialize USCI state machine

        P5DIR |= BIT0; //set P5.0 for /CS for external DAC

        P5OUT &= ~BIT0; //start at zero

        c = 0 ;

        k = 0;
        P5SEL1 |= BIT5;                           // Configure P5.4 for ADC
(A1 to A0!)
        P5SEL0 |= BIT5;

        P4SEL1 |= BIT7;                           // Configure P4.7 for ADC
        P4SEL0 |= BIT7;


        ADC14->CTL0 = ADC14_CTL0_SHT0_2 + ADC14_CTL0_ON + ADC14_CTL0_SHP +
ADC14_CTL0_SSEL_4 + ADC14_CTL0_CONSEQ_1; //+ ADC14IE15; // ADC14ON, interrupt
enabled, 16 clock samples, sequence of channels mode for two ADC inputs

        ADC14->CTL1 = ADC14_CTL1_RES_2; //sampling resolution, 14-bit
conversion. (now 12-bit!)

        ADC14->MCTL[0] = ADC14_MCTLN_INCH_0; // input A1, P5.4 (P5.5 now!)

        ADC14->MCTL[6] = ADC14_MCTLN_INCH_6; // second input A15, P6.0, P6.1
now!, P4.7 now

        ADC14->IER0 |= ADC14_IER0_IE0;                    // Enable ADC conv
complete interrupt

        SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;           // Wake up on exit
from ISR


        P4DIR &= ~BUTTON2;                       // button is an input
      //  P4OUT |= BUTTON2;                        // pull-up resistor
      //  P4REN |= BUTTON2;                        // resistor enabled

        P4IES &= ~BUTTON2;                        // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
        P4IFG &= ~BUTTON2; // P4.1 IFG cleared
        P4IE |= BUTTON2; // P4.1 interrupt enabled

        P2DIR &= ~BUTTON1;                       // button is an input
       // P2OUT |= BUTTON1;                        // pull-up resistor
       // P2REN |= BUTTON1;                        // resistor enabled

        P2IES &= ~BUTTON1;                        // interrupt on low-to-high
transition NEED CIRCUIT TO STOP FINGER FROM PRESSING BUTTON TO MUCH (BUTTON
DEBOUNCE)? WHAT AMPLITUDE DOES IT NEED TO BE TO TRIGGER THIS???
        P2IFG &= ~BUTTON1; // P2.4 IFG cleared
        P2IE |= BUTTON1; // P2.4 interrupt enabled


        __enable_interrupt(); // enable all interrupts

        NVIC->ISER[0] = 1 << ((ADC14_IRQn) & 31);         // Enable ADC
interrupt in NVIC module

        NVIC->ISER[1] = 1 << ((PORT4_IRQn) & 31);         // Enable port
interrupt for button
```

```
        NVIC->ISER[1] = 1 << ((PORT2_IRQn) & 31);         // Enable port 
interrupt for button


        change = 1;

        while(1) {

             ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;        // Start 
sampling/conversion


             if (a == 16){                        // if all 16 samples are 
taken, wavelet transform them.

                                    if (change == 0){
//deconstruct and reconstruct the signal if change = 0

                                          waveletdecom(decominput1);
                                          recon(decomout);
                                          do_some_conversions(reconout); 
//convert float to int
                                          Drive_DAC(b); //send output to DAC

                                          waveletdecom(decominput2);
                                          recon(decomout);
                                          do_some_conversions(reconout); 
//convert float to int
                                          Drive_DAC(b); //send output to DAC

                                    }
                                    else {                      // if change 
isn't one, take the wavelet transform

                                          waveletdecom(decominput1);
                                          do_some_conversions(decomout); 
//convert float to int
                                          Drive_DAC(b); //send output to DAC

                                          waveletdecom(decominput2);
                                          do_some_conversions(decomout); 
//convert float to int
                                          Drive_DAC(b); //send output to DAC

                                    }

                                     a = 0;

                                 }
                                 else {

                                 }
```

```
                }


}
   //  ADC14_IRQHandler


void ADC14IsrHandler(void) {

        int i;

        //DISABLE INTERRUPT?????? UNTIL driVE_dac IS DONE???

        //test = ADC14MEM0;  //added for testing

         decominput1[a] = ADC14->MEM[0];   //take samples
         decominput2[a] = ADC14->MEM[6];  //take second samples
       //  decominput2[a] = ADC14->MEM[6];  //take second samples

        // decominput[a] = decominput1[a];
        // decominput[a + 32] = decominput2[a];

         a++;

         for (i = 50; i > 0; i--);            // Delay

         //    do_some_conversions(test);
         //                 Drive_DAC(b); //send output to DAC *?
}

//void INT_PORT2_HANDLER(void){




//}

void INT_PORT4_Handler(void ){

        //while(1){ }

        int i;

        //int u;

        //u = 0;

        //u++;

       k++;

//     while(1) {
        if (change == 1 && (k % 2) == 0){ //check if k counter is even
(interrupt is trigger twice every button press)
              change = 0;
             }

        else if ((k % 2) == 0) {
             change = 1;
             }
        else {

        }

//     }
             for (i = 1000000; i > 0; i--);            // Delay


      P4IFG &= ~BUTTON2; // P4.0 IFG cleared

}

void INT_PORT2_Handler(void) {

      //while(1){ }
     int i;

     for (i = 1000000; i > 0; i--);            // Delay

       while(1){                  // stop operation until button is pushed
again

             // for (i = 100000; i > 0; i--);            // Delay



                                 if(P2IN &= ~BUTTON1){    // exit interrupt
routine when the button is no longer being pushed

                                 //     for (i = 1000000; i > 0; i--);
// Delay

                                        break;
                                 }
                                 else{

                                 }

                          }
      P2IFG &= ~BUTTON1; // P2.4 IFG cleared


}
```

Figure A.12: Final version of microcontroller code

Appendix B: Expenses Table

Table B.1 below shows the individual prices of each component used in the production and testing of the project. The total price of each component is shown at the bottom of the table as $221.36 and datasheets are shown for certain components.

The component prices are justified by the fact that, even though the design specifications state that the price of the device must be under $50, it is important to note that more components must be purchased than necessary to create only one silent communication device as failures, modifications, and multiple design approaches are likely. Therefore, much more components must be purchased to account for multiple circuits being built.

Table B.1: Component Expenses Table (Bill of Materials)

Component Expenses				
Item	Cost per unit (dollars)	Quantity	Total Cost (dollars)	Datasheet
Breadboard	7.99	2	15.98	Acquired from calpoly IEEE office
Wire leads	0.09	50	4.50	Acquired from calpoly IEEE office
Mc33078p op-amp	0.88 (0.26 for 1 ku)	4	3.52 (1.04 for 1 ku)	http://www.mouser.com/ds/2/405/mc33078-405489.pdf http://www.ti.com/product/MC33078/samplebuy
INA128P	11.57 (6.05 for 1 ku)	2	23.14 (12.10 for 1 ku)	http://www.ti.com/lit/ds/symlink/ina128.pdf http://www.ti.com/product/INA128/samplebuy
100 resistor	0.03	2	0.06	Acquired from calpoly IEEE office
270 resistor	0.03	2	0.06	Acquired from calpoly IEEE office
337 resistor	0.03	2	0.06	Acquired from calpoly IEEE office
1K resistor	0.03	4	0.12	Acquired from calpoly IEEE office
5K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
10K resistor	0.03	5	0.15	Acquired from calpoly IEEE office
50K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
200K resistor	0.03	4	0.12	Acquired from calpoly IEEE office
27K resistor	0.03	4	0.12	Acquired from calpoly IEEE office
56K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
270K resistor	0.03	4	0.12	Acquired from calpoly IEEE office

Item	Cost per unit (dollars)	Quantity	Total Cost (dollars)	Datasheet
2700K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
135K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
160K resistor	0.03	2	0.06	Acquired from calpoly IEEE office
1M resistor	0.03	1	0.03	Acquired from calpoly IEEE office
4.7uF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
0.1uF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
22pF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
100pF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
10000pF capacitor	0.16	30	5.00	https://www.amazon.com/uxcell-0-1uF-Voltage-Ceramic-Capacitors/dp/B008DFCUFW/ref=pd_lpo_328_bs_img_2?ie=UTF8&psc=1&refRID=NH30R8S1DDQW8SNRZCG3
470pF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
0.05MF capacitor	1.07	3	3.24	Acquired from calpoly IEEE office
390pF capacitor	1.07	2	2.14	Acquired from calpoly IEEE office
Sensor cable – Electrode Pads (3 connector)	4.95	1	4.95	https://www.sparkfun.com/products/12970
Audio Jack 3.5mm	1.50	1	1.50	https://www.sparkfun.com/datasheets/Prototyping/Audio-3.5mm.pdf

Item	Cost per unit (dollars)	Quantity	Total Cost (dollars)	Datasheet
MSP432P401R LaunchPad	12.99	2	25.98	http://www.ti.com/lit/ug/slau597/slau597.pdf
MSP-EXP430G2**:** MSP430 LaunchPad Value Line Development kit	9.99	1	9.99	http://www.ti.com/lit/pdf/slau318
Breadbroad buttons (4Pin DIP Micro PCB tactile)	0.53	10	5.30	https://www.amazon.com/6x6x6mm-Momentary-Push-Button-Switch/dp/B01GN79QF8/ref=sr_1_8?s=industrial&ie=UTF8&qid=1479592532&sr=1-8&keywords=button
MCP4921	1.97 (1.44 for 1 ku)	2	3.94 (2.88 for 1 ku)	http://ww1.microchip.com/downloads/en/devicedoc/21897b.pdf
Item	Cost per unit (dollars)	Quantity	Total Cost (dollars)	Datasheet
LTC1658	5.70 (5.15 for 0.1 ku)	2	11.4 (10.3 for 0.1 ku)	http://cds.linear.com/docs/en/datasheet/1658f.pdf
iMic Griffin	36.99 (sale)	1	36.99	https://www.amazon.com/Griffin-Technology-iMic-original-Adapter/dp/B003Y5D776/ref=sr_1_1?ie=UTF8&qid=1479592838&sr=8-1&keywords=imic+griffin
Meditrace electrodes	0.00	10	0.00	Given by professor

Item	Cost per unit (dollars)	Quantity	Total Cost (dollars)	Datasheet
3M Red Dot electrodes	0.00	10	0.00	Given by professor
Model M-008(YN-010) Power Bank	22.99	1	22.99	https://www.amazon.com/INNORI-Portable-External-22400mAh-Capacity/dp/B00N1GREF4
Model TM10 High-power Automobile Emergency Mobile Power Supply	20 (sale)	1	20	http://www.lelong.com.my/high-power-automobile-emergency-mobile-power-supply-demac-170677468-2016-01-Sale-P.htm
Stereo cord	0.00	1	0.00	Given by professor
Audio Jack Breakout	0.95	1	0.95	https://www.sparkfun.com/products/10588
Biomedical Sensor Pad (10 pack)	7.95	1	7.95	https://cdn.sparkfun.com/datasheets/Sensors/Biometric/H124SG.pdf
Total Cost (dollars): 221.36				