

Running Head: Micro Pitch Detector

Micro Pitch Detector

Jordan D. Meyer

Lake Superior State University Honors Program Thesis

Dr. Andrew Jones, Honors Thesis Advisor

April 10 2009

Table of Contents

| | |
|---|-----------|
| 1. Abstract | 3 |
| 2. Introduction | 4 |
| 3. Device Overview | 4 |
| 4. Theory of Operation | 5 |
| 4.1 Fourier Transform Algorithms | 6 |
| 5. Device Implementation | 10 |
| 5.1 Signal Flow..... | 10 |
| 5.2 Components..... | 11 |
| 5.3 Hardware implementation | 12 |
| 5.4 Software Implementation | 14 |
| 6. Design Complications | 17 |
| 7. Development Cost and Timeline | 19 |
| 8. Verification and Testing | 22 |
| 9. Conclusion | 24 |
| Works Cited | 25 |
| Appendix A: FFT Algorithms | 26 |
| First Algorithm Researched..... | 26 |
| Second Algorithm Researched | 28 |
| Modified Algorithm | 32 |
| Appendix B: Software | 35 |
| Acknowledgments | 55 |

1. Abstract

The Micro Pitch Detector is a device designed to determine the pitch or frequency of an audio signal. It uses electronic circuitry, a microcontroller, and software to sample audio signals, convert them from the time domain into the frequency domain by applying a variant of the Fourier Transform, determine the frequency, and display the results on an LCD screen. The following document covers the development of the Micro Pitch Detector. A working prototype of the Micro Pitch Detector was also constructed and demonstrated to the Lake Superior State University Honors Council and students.

2. Introduction

The Micro Pitch Detector project was chosen after considering how to apply studies in computer engineering to a musical background. Having studied music throughout secondary school, it sparked a personal interest in combining aspects of computer engineering with music. After speaking to Dr. Jones, the Faculty Advisor for this project, about ideas for possible projects, the Micro Pitch Detector was born. Since both Dr. Jones and I are involved in the worship team at the local Sault Wesleyan Church, it seemed fitting to investigate the possibility of blending our common interests. Furthermore, as a computer engineer, I wanted to gain more knowledge in the area of Digital Signal Processing (DSP), as well as practically apply what I have already learned. The following paragraphs outline the Device Overview, Theory of Operation, Device Implementation, Design Complications, Development Cost and Timeline, Testing Procedures, and Conclusion.

3. Device Overview

The Micro Pitch Detector is a device that uses a microcontroller to actively sample and calculate the frequency of an auditory input, such as a note played on a piano. A microcontroller is a small-scale processor that executes code designed to control external or onboard peripherals, allowing it to interact with its surroundings. The processor used to develop the Micro Pitch Detector is a derivative of the freescale 9S12XD family. It is scaleable up to an operating frequency of 32MHz, has 32KB of RAM, and includes numerous external peripherals, such as analog to digital converters, timers, and various digital inputs and outputs.

The main input to the device is an audio signal, which is converted to an electrical signal using an external microphone and amplifier circuitry. Before the signal is processed to

determine the frequency, the electrical signal is converted to a digital signal. Once converted, the digital data is manipulated by the microcontroller and processed by a Digital Signal Processing (DSP) function called the Fast Fourier Transform (FFT). This function converts a time-based signal, such as audio, from the time domain to the frequency domain, using complex number mathematics. After applying the FFT to the original signal, the output contains information correlating to the frequencies contained in the original signal. The frequency of the original signal is found by locating the index of the data with the largest magnitude and multiplying it by a factor of the sampling rate divided by the number of samples. After detecting the frequency, the information is presented to the user using an LCD screen.

In comparison to other devices, the Micro Pitch Detector is similar to a musical instrument tuner in that it detects the pitch and name of the played note. It differs from such devices in that it does not give an indication of the sharpness or flatness of a note but serves to identify only the frequency and name. Additionally, while similar methods may be used in the instrument tuners that exist today, their exact workings and software algorithms are unknown. Through the design and construction of the Micro Pitch Detector, however, more insight was gained on the possible design and implementation of devices such as musical instrument tuners. The Theory of Operation of the Micro Pitch Detector, and other similar devices, centers on the Fourier Transform, covered next.

4. Theory of Operation

The Micro Pitch Detector is able to detect frequencies using Fourier Analysis techniques, mainly the Fourier Transform. Fourier Analysis is the process of breaking a time-varying electrical signal into an infinite summation of simpler sine and cosine signals. Using this approach, a complex signal is described by the frequency components of the simpler sine and

cosine functions. Through the Fourier Transform, which is a function that transforms a signal from the time domain into its frequency components, Fourier Analysis makes the Micro Pitch Detector possible. The exact methods for implementing the Fourier Transform on the Micro Pitch Detector are discussed next.

4.1 Fourier Transform Algorithms

The Fourier Transform is the heart of the Micro Pitch Detector and is one method that makes the frequency detection possible. However, the complex nature of the Fourier Transform and advanced mathematics involved in performing the calculations is generally studied at the graduate level. Given that I have only had a few undergraduate classes in Digital Signal Processing (DSP), implementing my own algorithm for the Fourier Transform was beyond the scope of the project and my own understanding of the material. Therefore, I researched two similar algorithms to apply the Fourier Transform and combined their techniques. Both algorithms make use of the Fast Fourier Transform (FFT), which was credited to J. W. Cooley and J. W. Tukey in 1965 (Cooley, 2009).

In both algorithms, the FFT is computed by optimizing the Discrete Fourier Transform (DFT), which is performed on finite signals, such as an audio sample. Equation 1 shows the Discrete Fourier Transform, which sums each element of the input signal multiplied by a complex factor of $e^{-\frac{jwkn}{N}}$. In this equation, n and k range from zero to $N-1$, with N representing the number of samples and k incrementing every time n cycles through its entire range. The DFT is made even faster in the researched algorithms by eliminating multiplications that result in one, breaking the sequence into smaller sequences, and using twiddle factors, which are multiplication factors that are independent of the data processed (Twiddle, 2008).

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{j\omega kn}{N}} \quad k = 0, \dots, N - 1$$

Equation 1: Discrete Fourier Transform Summation

Before using any FFT algorithm, time-varying data must be sampled at a periodic rate that allows the signal to be reconstructed without aliasing. The Nyquist-Shannon sampling theorem, developed by Harry Nyquist and Claude Shannon, states that a continuous signal sampled and converted to a discrete signal, can be reconstructed if the sampling rate is at least twice that of the highest frequency being sampled. Since the highest note of a piano occurs at 4.186 kHz, a sampling rate of 10kHz was chosen to exceed the limit of twice the highest frequency, which is 8.372 kHz. If a signal is not sampled correctly, then aliasing can occur and lead to the improper frequency being detected. When aliasing does occur, frequencies above half of the sampling are reflected and falsely represented as frequencies below half the sampling rate. For example, if a sampling rate of 7 kHz were chosen, then the maximum allowable frequency of a signal before aliasing occurs would be a 3.5 kHz signal. However, since the highest note on a piano generates a signal at 4.186 kHz, this note would be indistinguishable from a note that naturally occurs at 2.814 kHz (Nyquist).

Douglas L. Jones, from the University of Illinois at Urbana-Champaign, developed the first algorithm researched. In this algorithm, many of the techniques for optimizing the speed have been applied such as breaking the DFT into smaller sets. Additionally, this algorithm also saves processing time and memory by overwriting the input signal with the results of the DFT. Another feature of this algorithm is that it breaks the FFT results into two components, the real component and the imaginary component. This is done because complex numbers cannot be directly represented on hardware and abstracting them in software is difficult and unnecessary, especially since only the magnitudes of each are needed. Since the microcontroller has limited

memory, this algorithm is attractive because the in place replacement of the input with the output reduces the amount of storage needed. The algorithm in its original form is included in Appendix A.

The second algorithm researched was developed by João Martins and based on work that appeared in a book titled *Numeric Recipes in C* by Brian Flannery, William Press, Saul Teukolsky, and William Vetterling (1992). In this algorithm, also shown in Appendix A, many features of the first algorithm appear such as twiddle factors and the divide and conquer techniques; however, it also highlights a method for determining the frequency of the data once the results are computed. One drawback in this approach, however, is that the output data requires allocation of memory equal to twice the size of the sample rate, which is 10kHz. This means that the output data needs 20,000 floating point memory locations, each 4 bytes wide, resulting in a total memory allocation of 80,000 bytes. This is required because, unlike the first algorithm, the real and imaginary components are stored in the same array with the real data located at even indexes and the imaginary components at odd indexes. Given that the microcontroller selected only has 32,000 bytes of RAM immediately available, this algorithm could not be implemented in full. Therefore, the best of both techniques were combined; resulting in an efficient FFT algorithm that reduces the memory storage needed and directly produces the resulting frequency.

In the combined algorithm, shown in Appendix A, the FFT uses techniques from both of the above-mentioned algorithms to create an efficient FFT that could be implemented on the microcontroller given its limited memory requirements. The new algorithm uses the in-place replacement methods of the first algorithm to reduce memory consumption and a modified version of the frequency detection from the second algorithm. In addition to combining the techniques from the other algorithms, other adjustments were made to improve processing time.

One adjustment made was switching data arrays used to store the output of the FFT algorithm from floating point numbers to integer numbers without loss of data for this implementation. This improves the speed of processing because integer values only use 16 bits of data to store information, whereas floating-point numbers use 32 bits.

Another change made to the combined algorithm that is different from the other two is the sample size and the method for determining the frequency. Ideally, a sample size of at least 4096 was desired. However, the FFT algorithm cannot process this many samples in a reasonable amount of time, so a sample size of 256 was selected. This new sample size was selected because the performance of the FFT decreases by a factor of four when the sample size is halved. With 256 samples, the new algorithm produces results within an average of one second versus over five minutes for 4096 samples.

In determining the frequency, the combined algorithm uses a similar technique to that of the second algorithm researched by searching for the index of the output signal with the highest magnitude. The difference, however, is that the second algorithm is based on sample sizes that are twice the sample rate. Therefore, the location of the highest magnitude in the output directly correlates to the input frequency. In the new algorithm, a much smaller sample size of 256 is used, which is also a power of two. Because of this fact, the symmetric property of the Fourier Transform can be applied, thus eliminating the need to look at the results of the second half of the output. This allows for faster frequency detection, as only 128 indexes need to be searched. Once the index pointing to the highest magnitude is found, it is normalized by multiplying by a factor of the sampling rate, then dividing by the number of samples. The implementation the Micro Pitch Detector, including this FFT algorithm, is covered next.

5. Device Implementation

The Micro Pitch Detector required both hardware and software components to fully implement. A block diagram representing the signal flow and hardware/software layout of the device is shown in Figure 1. The function of each block is covered in the following sections, beginning with the overall Signal Flow.

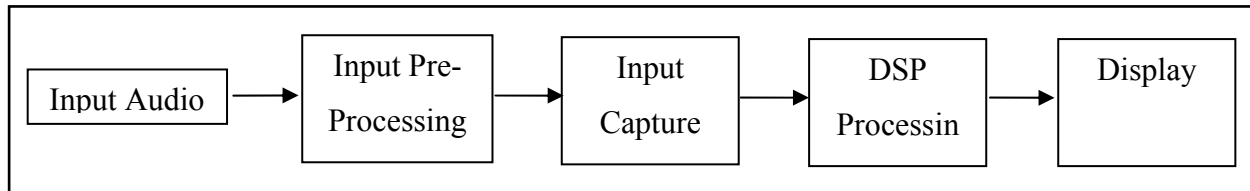


Figure 1: Micro Pitch Detector block diagram

5.1 Signal Flow

The block diagram of Figure 1 shows the overall layout and data flow of the Micro Pitch Detector. The first block is the *Input Pre-Processing Block*, which contains the microphone circuit and a simple analog amplifier. In addition to the amplifier circuitry, a voltage comparator circuit was implemented to help buffer the input from the microcontroller and prevent damage, as explained in Section 5.3, *Hardware Implementation*.

Next is the *Input Capture Block*, which represents the analog to digital (A/D) converter that is used to sample and digitize the input signal. The A/D converters are located on the microcontroller development board and sample the signal, converting the analog voltages provided by the input pre-processing block into digital values. From there the data is stored in RAM in order for the FFT algorithms to extract the frequency component.

After sampling and converting the data to digital values, the *DSP Processing Block* applies the FFT algorithm using the combine algorithm described in the previous section. After calculating the FFT, the algorithm will search the output for the location with the highest

magnitude. The index, or location, at which the highest magnitude occurs, corresponds to the frequency of the original signal by a factor of sampling rate over the number of samples.

Lastly, the information gathered from the *DSP Processing Block* is formatted and presented to the user through the *Display Block*. This block contains the hardware and software necessary to display information to the user. While the device is running, the name of the note will be displayed on the screen along with the frequency that was detected.

5.2 Components

The components selected for this project were chosen based on their functionality and cost. Approximately half of the project was constructed from purchased materials, while the remainder of the functionality was realized through programming. The first set of components required for the Micro Pitch Detector make up the first block in Figure 1, the *Input Pre-Processing Block*. This set contains a microphone, various resistors, a capacitor, an amplifier, and a voltage comparator. Together, these components act as a transducer to convert the audio energy into electrical signals and prepare the input signal for processing.

Following the *Input Pre-Processing Block* is the *Input Capture and DSP Processing Blocks*, which are made up of the microcontroller and its peripherals. The heart of this project, aside from the FFT algorithms, is the microcontroller, which functions as the main control unit of the device. It captures the input, applies the FFT algorithm, and provides feedback to the user through the LCD screen. For this project, the Motorola HC9S12X was selected because of its similarities to the Motorola cores used in lab activities of several Lake Superior State University electrical engineering courses. Additionally, the 9S12XDT512E development board containing the HC9S12X was selected because of its low cost, desired peripheral features, and scalable operating clock frequency up to 32MHz.

One of the microcontroller's peripherals is the Analog to Digital converter, or A/D. The A/D converter is used in the *Input Capture Block* and is controlled through the microcontroller by writing software to setup and control specific parameters of the A/D. The A/D is then used to sample the incoming signals and convert them to digital signals that are stored in memory of the microcontroller. As discussed in Section 4.1, *Fourier Transform Algorithms*, the sampling rate was selected to be 10kHz so that the entire frequency spectrum of the piano can be represented and captured correctly. The sampling rate is programmable and can be changed by setting the interval at which the timer interrupts. The next block in this set, the *DSP Processing Block*, was realized through software and algorithms programmed on the HC9S12 processor, also described in Section 4.1, *Fourier Transform Algorithms*.

The last major component of the Micro Pitch Detector is the LCD screen, which makes up the *Display Output Block*. The screen, which is a 20-character by 4-line display, provides the user with the name and measured frequency of the note that was detected. Messages instructing the user how to use the device may also be displayed.

5.3 Hardware implementation

The hardware required to implement the Micro Pitch Detector includes the microcontroller, a microphone circuit, an amplifier circuit, LCD screen, and various potentiometers, resistors, and capacitors. Power for the microcontroller and peripherals will be supplied through a wall transformer that provides 9V Direct Current (DC) to the development board. A 9V battery for portable operation could also easily replace the wall transformer. The LCD screen, amplifier circuit, and microphone will be powered through the development board, which contains a 5V DC power supply, accessible through the external pins.

For the original design, it was determined that the processor demand for the Micro Pitch Detector, while intense, should not require a complex or high-speed processor. Thus, the Motorola HC9S12 chosen for this project is significantly slower than modern computers and runs at only 2MHz, but is scalable up to 32MHz through a programmable clock, providing enough processing power to implement the necessary functionality. However, after implementation, the default speed of 2MHz proved to be too slow to sample the data at a reasonable rate to perform the FFT functions, so the clock was increased to 32MHz.

To implement the Micro Pitch Detector, the selected hardware was connected as shown in Figure 2. The audio signal is captured by the microphone circuit, which contains a standard PC condenser microphone, a 1K Ω resistor, and a 470nF capacitor. The microphone circuit changes pressure variations received by the microphone into voltages that mimic the audio signal being recorded. The output of the microphone circuit is then directed to an operational amplifier circuit, which boosts the amplitude of the signal by 150 times. This is necessary because the microphone generates only small voltages that are approximately 20mV to 50mV in magnitude. These small values are amplified to generate an input signal that ranges from 0V to approximately 5V.

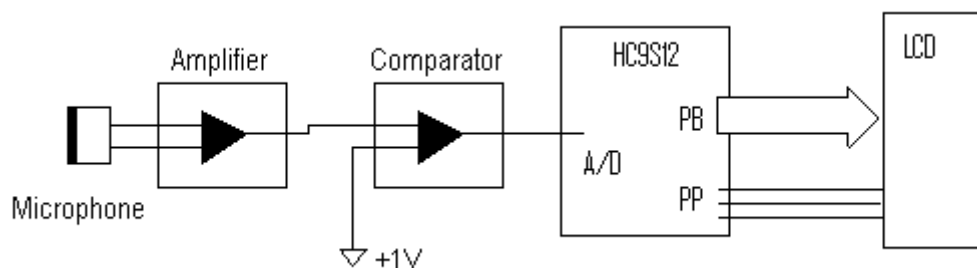


Figure 2: Micro Pitch Detector hardware schematic

After implementation of the amplifier circuit, it was discovered that audio signals at a high volume could produce an output with a magnitude of up to 7.5V. Thus, a voltage

comparator was added to the circuit to ensure that the input to the microcontroller does not exceed 5V. The voltage comparator compares its input to a reference voltage and outputs 5V if the input is above the threshold and 0V when the input is not above the threshold. This step helps to clean the input signal from noise, such as an unstable signal or background noise, and ensures that the input to the microcontroller will never exceed 5V. This is important because signal levels above 5V would damage the internal circuitry of the analog to digital converter or even the microcontroller itself. Together, the voltage comparator and amplifier produce an output signal that ranges from 0V to 5V, so that small changes in the input signal can be converted into a wide range of digital values by the A/D converter.

Once the signal is amplified, it is sampled by the A/D converter at a rate of 10kHz. This step converts the continuous analog sound wave into discrete digital values that can be processed by the microcontroller. The sampled data is then stored in memory to allow for future access by the FFT algorithm. After the algorithm has processed the data and determined what frequency was played, the information is passed on to the user through the LCD screen.

To display information on the LCD screen, it is first initialized. Afterwards, characters are printed one at a time by sending the data to the LCD driver embedded in the screen. Sending information to the screen, sampling data, and applying the FFT algorithms all occur through software, which is discussed next.

5.4 Software Implementation

The software aspect of the Micro Pitch Detector includes the functions to sample the data, the DSP algorithm that identifies the fundamental frequencies, and drivers that control information being passed to the LCD screen. In addition to these core elements of the software, various peripherals and ports are initialized before the main program runs, such as timers,

interrupts, and the A/D converter. Initializing the peripherals involves setting certain registers, or memory locations, which control the various functions of the peripherals and enables them. In addition to these peripherals, the General Purpose Input/Output (GPIO) ports must be set up in order to access them properly. For the Micro Pitch Detector, two external GPIO ports are used as outputs to send data from the microcontroller to the LCD screen. Following this configuration, the LCD screen is initialized by sending a series of commands that clear the screen, set the cursor to the home position, and configure other control registers.

Once initialization is complete, the main processing loop is executed along with a timer interrupt that allows for consistent sampling of the audio signal. This loop contains the function used to apply the FFT to the sampled data and remains idle until the sampling has been completed. The sampling occurs by a timer interrupt, which is function that is executed once every specified time interval and preempts any other processes currently running. In this manner, other less important code can continue to run while the samples are not taken, but when the timer expires, a sample is guaranteed to be taken. The use of the interrupt is necessary to ensure that an accurate sampling rate was implemented. By forcing a sample to occur every 100us, the samples are guaranteed to be evenly spaced and collected at the desired sampling rate of 10 KHz. As discussed earlier, this sample rate was chosen to avoid aliasing.

After being converted to a digital signal, the data is passed on to the FFT algorithm. Once the samples are completed, the interrupt is also disabled to prevent the processing of the FFT algorithm from being interrupted. This also prevents new samples from erasing the old samples that were not yet processed by the FFT, since it runs much slower than the sampling rate. The FFT algorithm implements the Discrete Fourier Transform, as described above in Section 4.1, *Fourier Transform Algorithms*. When transformed, the signal can be broken into

separate frequencies with the base frequency, or frequency of the signal, having the highest magnitude. This information is then passed on to the user through the LCD screen.

The LCD screen is a 20 character by 4-line display with a blue background and white text. The display is controlled through the microcontroller and the GPIO ports. After initializing the display, a message is printed to the screen by sending the message as a string of characters. The characters are sent to the screen in a loop by individually writing each character to the data register of the LCD screen. After each character is printed, the cursor is automatically incremented.

As the Micro Pitch Detector operates, it continually samples sound waves and updates the display. If a different pitch is detected, the information regarding the name of the note and the frequency that was sampled will be changed on the display. Since a smaller sample size was selected, the initial results are only accurate within 40 Hz. At higher frequencies, this difference is negligible because the difference between notes is greater than 50Hz. However, once the frequencies are below 1kHz, more accuracy is needed. Therefore, a windowing approach is used to find the exact region of the lower frequency.

To accomplish this, a lower frequency signal will be re-sampled at a rate that is one fourth the original sampling rate, or 2.5kHz, improving the accuracy to be within 10Hz. From there, a more accurate frequency can be detected. The process is repeated with a sampling rate of 1.250kHz if the frequency is determined to be lower than 500Hz and 625Hz if it is less than 300Hz, thus resulting in accuracy within 5Hz and 2.5Hz respectively. This process will iterate five or six times before returning to a sample rate of 10kHz.

The developed software discussed in this section is included in Appendix B. Throughout the software development and implementation of the Micro Pitch Detector, several complications were overcome, making it a challenging but rewarding experience, as discussed next.

6. Design Complications

Throughout the development of the Micro Pitch Detector, there were many setbacks and unexpected challenges to overcome. Through these challenges, I have become a better engineer, learned how to adapt to the challenging circumstances, and was able to apply my skills wherever possible to overcome them. For example, part of the original design for the Micro Pitch Detector included an analog filter to remove unwanted noise from the input signal before being processed by the microcontroller. This proved to be a greater challenge than originally thought because I have very little experience designing analog filters and it is not an easy or quick subject to learn. Furthermore, because of my inexperience, I was unable to design a filter with the desired outcome. This set the project back a few months as I spent a significant portion of the summer designing a filter that either did not work or did not produce results that were desired.

Through this challenge, I learned that it is better to move on to another part of the project that I do understand than invest a lot of time trying to figure out a complicated area that is above my understanding. After discussion with engineers, I discovered that the type of analog filter required to obtain the desired results would have been a multi stage filter with at least eight stages and several other parameters, which is beyond anything I could have designed on my own. Not all was lost in the summer months, however, as I did learn some filter design techniques and was able to simulate successfully some simpler single or dual stage filters.

Another challenge overcome was the implementation of the FFT algorithms. In order to use portions of the researched algorithms, a significant amount of RAM is required, as discussed in Section 4. This presented the challenge of running out of useable memory on the microcontroller, which caused about a week delay. By modifying the algorithms and expanding the memory of the microcontroller, I was able to overcome this design challenge and implement

a modified algorithm, included in Appendix B. This allowed me to use many of the desirable features of the research algorithms, including the speed, and reduced their memory usage.

To reduce the memory usage, the two algorithms were combined and modified, as discussed in Section 4.1, *Fourier Transform Algorithms*. The resulting algorithm reduces memory usage and improves performance by using a sample size of only 256 and separating the real and imaginary components. Additionally, the data used in the FFT is stored as integers to improve performance, resulting in a processing time of less than one second. In order to determine the frequency of the signal, the square of the magnitude of the real and imaginary data is calculated. However, because the sample size is an even power of two, the FFT results are symmetric and only the magnitude of the lower half needs to be calculated.

Furthermore, the square of the magnitude is used because it eliminates the need to perform a square root calculation, thus slightly improving performance. The magnitude of a complex signal is the square root of the sum of the squares of both the real and imaginary parts, as shown in Equation 2, where *re* represents the real part and *im* represents the imaginary part. The square root was eliminated because if the magnitude was the largest when calculated using Equation 2, then the square of that magnitude will also be the largest. From there, the frequency of the signal is then found by searching for the index with the highest square of the magnitude and multiplying it by the sampling rate divided by the number of samples taken. Once an approximate frequency is found, the windowing method is applied if necessary, as discussed in Section 5.4, *Software Implementation*.

$$magnitude = \sqrt{(re)^2 + (im)^2}$$

Equation 2: Magnitude of a complex number

Through this challenge, I gained a better understanding of how the FFT algorithms work and was able to implement my own approach that saved memory by extracting the useful portions of their code. I had to analyze the existing algorithms and gain an understanding of how they worked in order to modify them to suit my needs, which is a useful skill in software development, especially if I am assigned a project in my career where I am using existing code.

7. Development Cost and Timeline

Since this project was personally funded, it had a limited budget of \$200. All of the various components for this project were purchased and the budget was not exceeded. Table 1, lists the components purchased for this project and their costs, as well as the estimated software development cost had it been designed by someone else. As shown, the overall cost of hardware was \$134.25 and the estimated cost of software was \$855, resulting in a total of \$989.25. However, since I developed the software, I only incurred the cost of the hardware.

Additional costs to those listed in Table 1 include constructing an enclosure or mounting surface to house all the components and prevent them from breaking or becoming disconnected. The expected cost of this enclosure is currently unknown because the focus of the project is to construct the device and ensure that it is functioning properly. Once this occurs, the exact materials and costs for an enclosure or mounting surface will be considered, however, it is not expected that the cost of the enclosure or housing will exceed the remainder of the budget.

| Part Description | Cost |
|---|-------------|
| PC Condenser Microphone (x2) | \$11.63 |
| Motorola Freescale 9S12XDT512E Development Board | \$47.55 |
| Mixed Selection of Resistors and Capacitors | \$33.09 |
| 20x4 LCD Display | \$16.98 |
| Operational Amplifier | \$25.00 |
| Software Development Cost¹ | \$855.00 |
| Purchased Hardware Total | \$134.25 |
| Total | \$989.25 |

Table 1: Total cost and component breakdown

The project will be completed as originally planned by April 12, 2009 and the remainder of the time before the demonstration will be used for testing. Listed below in Table 2 are the approximate dates of when I worked on the Micro Pitch Detector and the estimated time spent on each task. As shown below, a significant portion of time was spent trying to implement the analog filters. Outside of that, the next largest portion of time was dedicated to constructing the sampling techniques, and researching the FFT algorithms. The areas where there are large gaps between times spent on the Micro Pitch Detector are indications of when I was busy with senior projects for engineering and other schoolwork. The remaining tasks are to optimize the FFT algorithm and improve the accuracy of the results, which is expected take less than one week of development time.

Due to the unexpected challenges I encountered, balancing time between this project and my senior engineering project, and using the voltage comparator circuit, I was unable to expand the features of the Micro Pitch Detector beyond detecting a single frequency. However, I did still learn a significant amount about signal processing, analog filter design, and managing time.

¹ Software Development Cost based on 45 hours of development time at \$18.00 per hour. See Table 2 for information regarding time of development.

Even with the core features in place, the Micro Pitch Detector was a great challenge and a great project to culminate my academic career here at LSSU.

| Date | Task | Time (hr) |
|-----------------------------|--|------------------|
| Hardware Development | | |
| September 2007 | Initial conception | 5.0 |
| October 24, 2007 | Purchased Microphone | 2.0 |
| October 30, 2007 | Purchased Microcontroller | 5.0 |
| November 2007 | Testing/constructing microphone circuit | 10.0 |
| June 2008 | Analog Filter Design/Simulations | 50.0 |
| June 2008 | Purchased electrical components (amps, resistors, capacitors) | 4.0 |
| July 2008 | Purchased LCD screen and soldered wires | 3.0 |
| February 2009 | Purchased and voltage inverter to power amplifier and built voltage comparator | 4.0 |
| Software Development | | |
| September 2008 | Developed LCD driver code | 8.0 |
| November 2008 | Added features to LCD Drivers: Print to any location on screen, shift cursor left/right, and text flows from line 1 to 2, rather than 1 – 3 as normally occurs. | 8.0 |
| March 20, 2009 | Began programming interrupt routines to sample audio | 3.0 |
| March 22, 2009 | Got interrupt and timer working at 10KHz but A/D can't complete sampling when main clock is running at 2Mhz | 4.0 |
| March 24, 2009 | A/D previous sample now read during each interrupt and next sample started after the read. Adjusted accuracy from 10bits to 8 bits for faster processing. | 2.5 |
| March 28, 2009 | Adjusted clock to 24Mhz for faster FFT and reliable sampling. Adjusted LCD Driver for with new timing. Finalized sampling sequence. Correctly samples at 10KHz & disables interrupts after data collected, re-enables interrupt after updating display | 7.0 |
| March 29, 2009 | Researched & implemented FFT algorithms. Ran out of memory but able to expand memory slightly. Attempted to run first algorithm with modifications but takes too long to process. Second algorithm does not run due to low memory | 6.5 |
| March 30, 2009 | Developed methods to reduce memory usage of algorithms and improve processing time. Yet to be implemented | 2.0 |
| April 7, 2009 | Implemented improved algorithm but did not succeed. After more research discovered FFT is symmetrical and only need to use lower half. Using this approach, increasing clock to 32MHz, and storing calculated magnitudes in larger data types created working device. Need to improve processing time & accuracy | 4.0 |
| April 9, 2009 | Implemented windowing method for lower frequencies. Still needs adjustment | 2.5 |
| Hardware Total | | 83.00 |
| Software Total | | 47.50 |
| Overall Total | | 130.50 |

Table 2: Development Timeline

8. Verification and Testing

After completing the construction and programming phases of this project, the output displayed on the screen was verified. This was accomplished by testing various aspects of the Micro Pitch Detector. Table 3 outlines the various tests that were conducted, with a description of each test following. The first four tests have been completed and the results are listed in Table 4. However, final adjustments will be made to improve the accuracy and processing time and the results of the last test will not be available before submitting this document. Therefore, a supplement providing the results of this test will be provided during the final presentation.

| Verification Test | Purpose of Test |
|------------------------------|--|
| LCD Display Test | Verify proper display of characters and positioning of cursor |
| Signal Amplification Test | Verify that Amplifier Circuit produces expected signal amplification |
| Signal Sampling Test | Verify that data is sampled correctly |
| DSP Processing Test | Verify that Digital Filters and DFT functions produce expected results |
| Overall Pitch Detection Test | Verify that proper pitches are detected 70% of the time |

Table 3: Verification Tests

| Verification Test | Test Results |
|------------------------------|--|
| LCD Display Test | Passed; Screen correctly accepts commands and prints characters at the specified location of the desired line. Code was scaled to work at any clock frequency and been tested at 2MHz, 24MHz and 32MHz |
| Signal Amplification Test | Passed; Oscilloscope was used to measure signals of varying frequencies before amplifier and after. Amplifier and voltage comparator produces square wave at the same frequency of the input without exceeding 5 Volts. |
| Signal Sampling Test | Passed; Data correctly sampled at 10kHz. Verified by blinking led whenever a sample was taken. Oscilloscope was used to measure frequency of pulsing light & was visually confirmed to be half the sampling rate. Since it takes two sample cycles to turn the led on an off this test produced correct results |
| DSP Processing Test | Passed; Debug module was used along with LCD screen to verify the frequency detected against the input frequency. FFT produces correct frequency within accuracy of 40Hz |
| Overall Pitch Detection Test | NA |

Table 4: Test Results

Each test listed above ensured that the Micro Pitch Detector performs correctly and produces the desired results. The first test listed, the *LCD Display Test*, was conducted to ensure that the functions controlling the LCD screen send data correctly and that the correct messages are received by the LCD screen. A check was also made to ensure that the cursor moved to the appropriate location after reaching the end of a line. To conduct this test, several strings of varying length were printed to the screen and a visual check verified that the words correctly wrapped to the next line. By design, the LCD screen makes characters jump from line 1-3 and from 2-4, so the LCD drivers written had to account for this and adjust the position of the cursor when it reached the end of a line. The code controlling most of the Micro Pitch Detector was written by hand, so it is imperative to ensure that the code is bug free and that it produces the expected outcome, without damaging components or causing the device to stop functioning.

Next, a *Signal Amplification Test* was conducted to ensure that the analog circuit is functioning properly. This test verified the signal integrity before and after it was amplified, as well as ensured that the amplitude of the signal remained within the bounds of the A/D converter. This test was conducted by measuring the signal before and after the amplifier and comparing the input frequency to that which was observed on the oscilloscope.

Similar to this test will be the *Signal Sampling Test*, which ensured that the data is properly sampled by the A/D converter. This test was important because without a properly sampled signal, it is impossible to determine the correct pitch of the note that was sampled. It also verified that data is stored correctly after it is sampled, so that it is ready to be sent to the DSP portion of the code. This test was conducted by using the software debugger to verify that the A/D converter was correctly sampling data and placing the results in the storage array.

Next was the *DSP Processing Test*. Testing the DSP functions of the device ensures that the proper results are obtained before the data is sent to the LCD screen. This test involved

running the code in debug mode with a known input frequency and then observing the results of the Micro Pitch Detector. This was completed using several input frequencies and the results were consistent. The FFT algorithm correctly identifies any input within 40Hz.

Last is the *Overall Pitch Detection Test*, which will be completed after submitting this document. The test will verify that the pitch detection process is working and the correct results are displayed. Additionally, it will verify the improved accuracy of the windowing method for lower frequencies and that the name of the note displayed correctly corresponds to the frequency detected. It will be conducted by playing a known audio pitch and calculating the percentage of correct detections the Micro Pitch Detectors produces, with a goal of reaching 70% accuracy.

9. Conclusion

The Micro Pitch Detector combines several aspects of computer engineering and applies them to the world of music. Benefits of the project include applying what has been studied in computer and electrical engineering courses and constructing a device that can be applied to the field of music. Through this project, I have gained knowledge and understanding in advanced Digital Signal Processing such as designing and implementing filters and the use of the Fourier Transform. Additionally, I have gained experience in overcoming challenges and setbacks and developed improved time management skills in balancing multiple projects. Furthermore, this project has given me the opportunity to complete an engineering design project on my own that challenges the knowledge I have in Computer Engineering and has offered the opportunity to expand and enhance my engineering and programming skills, such as designing electrical circuits and analyzing complex algorithms. The overall goal of this project has been met and a completed prototype of the Micro Pitch Detector will be delivered to the Honors Council and the community and April 18, 2009.

Works Cited

- Cooley-Tukey FFT Algorithm*. (2009, March 26). Retrieved April 1, 2009, from Wikipedia The Free Encyclopedia: http://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm
- Flannery, B. e. (1992). *Numerical Recipes in C - The Art of Scientific Computing*. New York: Cambridge University Press.
- Lyons, R. G. (2001). *Understanding Digital Signal Processing*. New Jersey: Prentice Hall PTR.
- Nyquist-Shannon Sampling Theorem*. (2008, October 9). Retrieved October 11, 2008, from Wikipedia The Free Encyclopedia: http://en.wikipedia.org/wiki/Nyquist-Shannon_sampling_theorem
- Twiddle Factor*. (2008, July 15). Retrieved April 2009, 1, from Wikipedia The Free Encyclopedia: http://en.wikipedia.org/wiki/Twiddle_factor

Appendix A: FFT Algorithms

First Algorithm Researched

```

/*****
/* fft.c
*/
/* (c) Douglas L. Jones
*/
/* University of Illinois at Urbana-Champaign
*/
/* January 19, 1992
*/
/*
/*
*/
/* fft: in-place radix-2 DIT DFT of a complex input
*/
/*
*/
/* input:
*/
/* n: length of FFT: must be a power of two
*/
/* m: n = 2**m
*/
/* input/output
*/
/* x: double array of length n with real part of data
*/
/* y: double array of length n with imag part of data
*/
/*
*/
/* Permission to copy and use this program is granted
*/
/* under a Creative Commons "Attribution" license
*/
/* http://creativecommons.org/licenses/by/1.0/
*/
*****/
fft(n,m,x,y)
int n,m;
double x[],y[];
{
int i,j,k,n1,n2;
double c,s,e,a,t1,t2;

j = 0; /* bit-reverse */
n2 = n/2;

```

```

for (i=1; i < n - 1; i++)
{
    n1 = n2;
    while ( j >= n1 )
    {
        j = j - n1;
        n1 = n1/2;
    }
    j = j + n1;

    if (i < j)
    {
        t1 = x[i];
        x[i] = x[j];
        x[j] = t1;
        t1 = y[i];
        y[i] = y[j];
        y[j] = t1;
    }
}

n1 = 0; /* FFT */
n2 = 1;

for (i=0; i < m; i++)
{
    n1 = n2;
    n2 = n2 + n2;
    e = -6.283185307179586/n2;
    a = 0.0;

    for (j=0; j < n1; j++)
    {
        c = cos(a);
        s = sin(a);
        a = a + e;

        for (k=j; k < n; k=k+n2)
        {
            t1 = c*x[k+n1] - s*y[k+n1];
            t2 = s*x[k+n1] + c*y[k+n1];
            x[k+n1] = x[k] - t1;
            y[k+n1] = y[k] - t2;
            x[k] = x[k] + t1;
            y[k] = y[k] + t2;
        }
    }
}

return;
}

```

Second Algorithm Researched

```

/*****
* Copyright (C) 2009
*
* Filename:  complexFFT.h
*
* Purpose: Header file to implement FFT function derived from
*          Numerical Recipes in C - Art of Scientific Computing.  Uses
*          a bit swapping method and stores results in an array that
*          is 2*sample_freq with the following form:
*          array[2*sample_freq] - an array with 2*sample_freq elements
*          (0 based)
*          index n - real part of FFT --\_ together form index 0
*          index n+1 - complex of FFT --/
*
*          The function also returns an integer that represents the
*          base frequency of the input signal.  The following
*          paragraphs describe the copyright information and from
*          where the source code was obtained.
*
* ----- Source Location and License Agreements -----
*
* The code used in this algorithm was obtained from codeproject.com in
* an article titled: "How to implement the FFT algorithm" written by
* João Martins.  It is located at the following address:
* http://www.codeproject.com/KB/recipes/howtofft.aspx
*
* In this article, João Martins references the book titled: "Numerical
* Recipes in C (1982)" and claims that his work is mostly derived from
* algorithms presented in this book.  The difference is that he has
* optimized it slightly for speed.  The Second Edition of the book,
* copyrighted in 1992, is available from the following site and
* contains a similar algorithm to João Martins:
* http://people.hnl.bcm.tmc.edu/cuixu/paper/201.pdf
*
* According to the publishers of the book, any code or ideas written
* can be used for free as long as they are typed in by hand and not
* distributed commercially.  The exact wording of their license is
* included below.  Given that my implementation derives from João
* Martins, who already modified their version, and that it will not be
* used for commercial use or made available to others, the free
* license should apply.  Below is the wording of their agreement:
*
* NR Books and Cambridge University Press Free License Agreement
* -----
* ["Immediate License"] If you are the individual owner of a copy of
* this book and you type one or more of its routines into your
* computer, we authorize you to use them on that computer for your own
* personal and noncommercial purposes.  You are not authorized to
* transfer or distribute machine-readable copies to any other person,

```

```

* or to use the routines on more than one machine, or to distribute
* executable programs containing our routines. This is the only free
* license.
*
* ----- Source Citing and References -----
*
* I am also including the reference information for the book, since I
* am indirectly referencing it through the information made available
* by João Martins. Below is the citing for the book:
*
* ----- MLA citation -----
*
* Flannery, Brian, William Press, Saul Teukolsky, and William
* Vetterling. "Fast Fourier Transform." Numerical Recipes in C - The
* Art of Scientific Computing. New York: Cambridge University Press,
* 1992. 496-536
*
*****/
#include <stdio.h>
#include <math.h>

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr
#define pi 3.1415926535897932384626433832795 //pi
#define r2 20000

//global variables
unsigned int fundamental_frequency;

//function prototype
void fft(float *d, unsigned int num_samp, int rate, int sign);

//implementation
void fft(float *d, unsigned int num_samp, int rate, int sign)
{
    //variables for the fft
    unsigned int n1, mmax, m1, j, istep, i;
    double wtemp,wr,wpr,wpi,wi,theta,tempr,tempi;

    //the complex array is real+complex so the array
    //as a size n = 2* number of complex samples
    //real part is the data[index] and
    //the complex part is the data[index+1]

    float vector[r2];

    //put the real array in a complex array
    //the complex part is filled with 0's
    //the remaining vector with no data is filled with 0's
    for(i = 0; i < rate; i++)
    {

```

```

    if(i<num_samp)
        vector[2*i] = d[i];
    else
        vector[2*i]=0;
        vector[2*i+1]=0;
}

//binary inversion (note that the indexes
//start from 0 which means that the
//real part of the complex is on the even-indexes
//and the complex part is on the odd-indexes)
n1 = rate << 1;
j=0;
for (i=0;i<n1/2;i+=2)
{
    if (j > i)
    {
        SWAP(vector[j],vector[i]);
        SWAP(vector[j+1],vector[i+1]);
        if((j/2)<(n1/4))
        {
            SWAP(vector[(n1-(i+2))],vector[(n1-(j+2))]);
            SWAP(vector[(n1-(i+2))+1],vector[(n1-(j+2))+1]);
        }
    }
    m1=n1 >> 1;

    while (m1 >= 2 && j >= m1) {
        j -= m1;
        m1 >>= 1;
    }
    j += m1;
}
//end of the bit-reversed order algorithm

//Danielson-Lanzcos routine
mmax = 2;
while (n1 > mmax) {
    istep = mmax << 1; //nmax * 2
    theta = sign*(2*pi/mmax);
    wtemp = sin(0.5*theta);
    wpr = -2.0*wtemp*wtemp;
    wpi=sin(theta);
    wr=1.0;
    wi=0.0;
    for (m1 = 1;m1<mmax;m1+=2) {
        for (i=m1;i<=n1;i+=istep) {
            j=i+mmax;
            tempr=wr*vector[j-1]-wi*vector[j];
            tempi=wr*vector[j]+wi*vector[j-1];
            vector[j-1]=vector[i-1]-tempr;
            vector[j]=vector[i]-tempi;

```

```

        vector[i-1] += tempr;
        vector[i] += tempi;
    }
    wr=(wtemp=wr)*wpr-wi*wpi+wr;
    wi=wi*wpr+wtemp*wpi+wi;
}
mmax=istep;
}
//end of the algorithm

//determine the fundamental frequency
//look for the maximum absolute value in the complex array
fundamental_frequency=0;
for(i=2; i <= rate; i+=2)
{
    if((pow(vector[i],2)+pow(vector[i+1],2)) >
        (pow(vector[fundamental_frequency],2) +
         pow(vector[fundamental_frequency+1],2)))
    {
        fundamental_frequency=i;
    }
}

//since the array of complex has the format
//[real][complex]=>[absolute value]
//the maximum absolute value must be adjusted to half
fundamental_frequency=(int)floor((float)fundamental_frequency/2);
}

```

Modified Algorithm

```

/*****
/* fft.h
/* (c) Douglas L. Jones
/* University of Illinois at Urbana-Champaign
/* January 19, 1992
/*
/* fft: in-place radix-2 DIT DFT of a complex input
/*
/* input:
/* n: length of FFT: must be a power of two
/* m: n = 2**m
/* input/output
/* re: int array of length n with real part of data
/* im: int array of length n with imag part of data
/*
/* Permission to copy and use this program is granted
/* under a Creative Commons "Attribution" license
/* http://creativecommons.org/licenses/by/1.0/
*****/

```

```

void fft()
{
    int i,j,k,n1,n2;
    float c,s,e,a;
    double t1,t2;

    j = 0; /* bit-reverse */
    n2 = n/2;
    for (i=1; i < n - 1; i++)
    {
        n1 = n2;
        while ( j >= n1 )
        {
            j = j - n1;
            n1 = n1/2;
        }
        j = j + n1;

        if (i < j)
        {
            t1 = re[i];
            re[i] = re[j];
            re[j] = t1;
            t1 = im[i];
            im[i] = im[j];
            im[j] = t1;
        }
    }
}

```

```

}

n1 = 0; /* FFT */
n2 = 1;

for (i=0; i < m; i++)
{
    n1 = n2;
    n2 = n2 + n2;
    e = (-2*pi)/n2;
    a = 0.0;

    for (j=0; j < n1; j++)
    {
        c = cos(a);
        s = sin(a);
        a = a + e;

        for (k=j; k < n; k=k+n2)
        {
            t1 = c*re[k+n1] - s*im[k+n1];
            t2 = s*re[k+n1] + c*im[k+n1];
            re[k+n1] = re[k] - t1;
            im[k+n1] = im[k] - t2;
            re[k] = re[k] + t1;
            im[k] = im[k] + t2;
        }
    }
}

//end of the algorithm

//determine the fundamental frequency
//throw away half of data because it is semetric          512
//keep DC component and nyquist component (data[0] and
//data[ceil((n+1)/2) - 1]
//compute the power, by doubling the squares of the magnitudes

//look for the maximum absolute value in the complex array
fundamental_frequency = 0.0;
index = 0;
mag[0] = ((double)re[0]*re[0]+(double)im[0]*im[0]);
                                                    //DC component

big_mag = 0;
mag[num_pt-1] = ((double)re[num_pt-1]*re[num_pt-1] +
                (double)im[num_pt-1]*im[num_pt-1]);
                                                    //Nyquist component

for(i=1; i < num_pt-1; i++)
{
    mag[i] = 2*((double)(re[i]*re[i])+(double)(im[i]*im[i]));
}

```

```

// 2*mag^2
    if(mag[i] > big_mag)
    {
        index = i;
        big_mag = mag[i];
    }
}
fundamental_frequency = f[index];
}
```

Appendix B: Software

```

/*****
*
*      Copyright (C) 2009
*
*  Filename:      main.c
*  Author:       Jordan Meyer
*  Revision:     4.0
*
*  Description:   Micro Pitch Detector Source Code.  This project
*                detects the frequency and name of a note played on a
*                piano for my honors thesis.  It uses a microphone and
*                amplifier circuit to provide input to the A/D
*                converter of the microcontroller.  The data is
*                sampled at a rate of 10KHz and the results are passed
*                to an FFT algorithm.  Once the results of the FFT
*                finish, the frequency is found by calculating the
*                magnitude of the lower half of results and looking for
*                the index of the highest magnitude.  Because the sample
*                size of 256 is an even power of two, the FFT is
*                symmetrical and only lower half of results are
*                searched.  The index is then multiplied by sampling
*                rate / sample size to find frequency.  Once detected,
*                frequency is printed to LCD screen
*
*
*  Notes:        Used in Projects Pitch_Detector_Main.mcp
*
*****/

#include <hidef.h>          /* common defines and macros */
#include <mc9s12xdt512.h>   /* derivative information */
#include <stdio.h>
#include <math.h>

#include "peripherals.h"   // Include the lcd and timer functions

#pragma LINK_INFO DERIVATIVE "mc9s12xdt512"

#define SOFTWARETRIGGER0_VEC  0x72 /* vector address= 2 * channel id
*/
#define ROUTE_INTERRUPT(vec_adr, cfdata)          \
    INT_CFADDR= (vec_adr) & 0xF0;                \
    INT_CFDATA_ARR[((vec_adr) & 0x0F) >> 1]= (cfdata)

#define pi 3.14159265 //pi
#define n 256
#define m 8
#define INTERRUPT 3200; //set interrupt interval to 100us (10khz)
#define num_pt 129 //ceil((n+1)/2)

```

```

//function prototypes
void clk_init(void);
void copy_data(volatile unsigned char *d,int size);
void calc_freq(float *f, float r); //rebuild frequency using sampling
rate
void fft(void);

//global variables

unsigned int index;
volatile int num = 0; // number of samples taken
        int count = 0;

volatile unsigned char data_10000[n]; //storage array for 10K samples
volatile unsigned char data_5000[n]; //storage array for 5K samples
volatile unsigned char data_2500[n]; //storage array for 2.5K samples
volatile unsigned char data_1250[n]; //storage array for 1.25K samples
volatile unsigned char data_625[n]; //storage array for 0.625K samples
volatile unsigned char data_312[n]; //storage array for 0.312K samples

        long double mag[num_pt]; //storage array for magnitude of
samples
        long double big_mag; //largest magnitude
        int re[n]; //real component of FFT
        int im[n]; //imaginary component of FFT
        float f[num_pt]; //frequency array
        float fundamental_frequency = 0.0;

#pragma CODE_SEG DEFAULT

void main(void)
{
    char *msg = "Welcome to the Micro Pitch Detector! Push button to
                begin.";
    char m1[21];
    int i = 0;

    //initialize peripherals and enable interrupts
    clk_init(); //scale clock to 24MHz
    EnableInterrupts;
    PeriphInit();
    atd_init(); //initialize ATD
    timer_init();
    initLCD();

    //init_data(n); //clear array

    lcd_clear();
    calc_freq(f,10000);

```

```

//print welcome message
while(*msg) {
    lcd_data(*msg++);
    delay_ms(3);
}

while(!(PORTB & 0x01)); //wait for push button

//turn on A/D so that the first sample is ready before the first
//interrupt
ATD0CTL5 = 0x00; // 8 bit, Unsigned number, left justified one
//scan

TC1 = TCNT + INTERRUPT;
TIE = 0x02; //enable interrupts
lcd_clear(); //clear screen
lcd_print("Frequency: ",0,0);
lcd_print("Note Name: ",1,0);
for (;;)
{
    if(!(TIE & 0x02))
    {
        calc_freq(f, 10000.0); //use frequency array based on 10Khz
        copy_data(data_10000,n);
        fft(); //compute fft results

        if(fundamental_frequency < 156)
        {
            copy_data(data_312,n);
            calc_freq(f,312.5);
            fft(); //calculate FFT using 5K as sample rate
        }

        else if(fundamental_frequency < 313)
        {
            copy_data(data_625,n);
            calc_freq(f,625);
            fft(); //calculate FFT using 5K as sample rate
        }

        else if(fundamental_frequency < 625)
        {
            copy_data(data_1250,n);
            calc_freq(f,1250.0);
            fft(); //calculate FFT using 5K as sample rate
        }
        else if(fundamental_frequency < 1250)
        {
            copy_data(data_2500,n);
            calc_freq(f,2500.0);
        }
    }
}

```

```

    fft(); //calculate FFT using 5K as sample rate
}
else if(fundamental_frequency < 2500)
{
    copy_data(data_5000,n);
    calc_freq(f,5000.0);
    fft(); //calculate FFT using 5K as sample rate
}

sprintf(m1,"%0.2f",fundamental_frequency);

lcd_print(m1,0,11); //print to line 1, column 12
//delay_ms(1000); //wait 1 sec
TC1 = TCNT + INTERRUPT;
TIE = 0x02; //re-enable interrupts
}

}

}

/***** Functions *****/

/*****
*
* Name      : calc_freq
* Parameters: float pointer to array, int - sample rate
* Purpose   : calculate array of frequencies for given sample rate
* Returns   : None
*
*****/
/
void calc_freq(float *f, float r)
{
    int i;
    for(i = 0; i < num_pt;i++) //compute frequency array
        f[i] = i*(r/n);
}

/*****
*
* Name      : copy_data
* Parameters: int - size of array
* Purpose   : Copies data array that holds samples from A/D into
floating
*           point array for FFT
* Returns   : None
*
*****/
/
void copy_data(volatile unsigned char *d,int i)
{

```

```

int j;
for(j = 0; j < i; j++)
{
    if(d[j] < 128)
        re[j] = 0;
    else
        re[j] = 10;
    im[i] = 0; //clear imaginary part
}
}

/***** CLOCK functions
*****/

/*****
* Initialize Clocks
* establishes bus clock to 32 MHz
*****/
void clk_init (void)
{
/*
The internal PLL clock lets us set the speed of the processor.
The default bus speed will be 2 MHz (half the OscFreq).

The math used to set the PLL frequency: (OscFreq = 4 MHz)
SYNR = 7      (PLL multiplier - 1)
REFDV = 0     (PLL divider - 1)
PLLCLK = 2 * OscFreq * (SYNR + 1) / (REFDV + 1)

The OscFreq is 4 MHz, so we have PLLCLK = 2*4*8/1 = 64 MHz
The bus clock runs at half of the PLL speed: Bus Clock = PLLCLK / 2 =
32 MHz

Currently the fastest speed this device can safely use is 24 MHz.
If you set a slower speed, it will reduce power consumption.
*/
// Set the PLL speed (**change this if you want to slow down CPU**)
    CLKSEL &= 0x7F;          // disengage PLL to system
    PLLCTL |= 0x40;        // turn on PLL
    SYNR = 0x07;           // set PLL multiplier
    REFDV = 0x00;          // set PLL divider
    asm("nop");
    asm("nop");
    while (!(CRGFLG & 0x08)); // wait for clock to sync
    CLKSEL |= 0x80;        // engage PLL to system

// Disable watchdog timer (COPCTL register)
    COPCTL = 0x40;         // COP off - RTI and COP stopped in BDM-
mode
}

```

```

/***** FFT function *****/

/*****/
/* fft.h */
/* (c) Douglas L. Jones */
/* University of Illinois at Urbana-Champaign */
/* January 19, 1992 */
/* */
/* fft: in-place radix-2 DIT DFT of a complex input */
/* */
/* input: */
/* n: length of FFT: must be a power of two */
/* m: n = 2**m */
/* input/output */
/* re: int array of length n with real part of data */
/* im: int array of length n with imag part of data */
/* */
/* Permission to copy and use this program is granted */
/* under a Creative Commons "Attribution" license */
/* http://creativecommons.org/licenses/by/1.0/ */
/*****/

void fft()
{

    int i,j,k,n1,n2;
    float c,s,e,a;
    double t1,t2;

    j = 0; /* bit-reverse */
    n2 = n/2;
    for (i=1; i < n - 1; i++)
    {
        n1 = n2;
        while ( j >= n1 )
        {
            j = j - n1;
            n1 = n1/2;
        }
        j = j + n1;

        if (i < j)
        {
            t1 = re[i];
            re[i] = re[j];
            re[j] = t1;
            t1 = im[i];
            im[i] = im[j];
            im[j] = t1;
        }
    }
}

```



```

        if(mag[i] > big_mag)
        {
            index = i;
            big_mag = mag[i];
        }
    }

    fundamental_frequency = f[index];
}

/***** ISR Functions
*****/
#pragma CODE_SEG __NEAR_SEG NON_BANKED

/*****
// Name: isr_tim1()
// Function: To capture data from A/D converter at 15.5Khz
//
*****/

interrupt void isr_t1()
{
    TC1 = TC1 + INTERRUPT; //reset interrupt interval to be 100us later
    PORTB_PB1 = ~PORTB_PB1;

    data_10000[num%n] = ATD0DR0H;

    if(!(num%2)) //use every other sample for 5K
        data_5000[(num/2)%n] = ATD0DR0H;

    if(!(num%4)) //use every fourth sample for 2.5k
        data_2500[(num/4)%n] = ATD0DR0H;

    if(!(num%8)) //use every eighth sample for 1.25k
        data_1250[(num/8)%n] = ATD0DR0H;

    if(!(num%16)) //use every 16th sample for 625Hz
        data_625[(num/16)%n] = ATD0DR0H;

    if(!(num%32)) //use every eighth sample for 312.5Hz
        data_312[(num/32)%n] = ATD0DR0H;

    ATD0CTL5 = 0x00; // 8 bit, Unsigned number, left justified single
                    //scan, start next conversion
    num++;

    if(num == (n*32-1))
    {
        TIE = 0x00; //disable interrupts while processing
        PORTB_PB1 = 0; //turn off processing light
        num = 0;
    }
}

```

```

num = num % (n*32); //wrap counter from 0 - 4095 (4096 samples)

    TFLG1 = 0x02; //clear timer ch1 interrupt flag
}

#pragma CODE_SEG DEFAULT

/***** Peripheral Control Functions *****/
/*****
*
*       Copyright (C) 2008
*
* Filename:    peripherals.h
* Author:     Jordan Meyer
* Revision:   2.0
*
* Description: Header File for peripherals and LCD driver
*
* Notes:      Used in Projects Pitch_Detector_Main.mcp
*
*****/

/* include peripheral declarations */
#include <mc9s12xdt512.h>
#include <hidef.h>      /* common defines and macros */

#define INTERRUPT 3200 //set interrupt interval to 100us (10khz)

volatile int pos = 0;
volatile int line = 0;

void PeriphInit(void);
void atd_init(void);
void timer_init(void);
void initLCD(void);
void lcd_pos(void);
void lcd_clear(void);
void lcd_print(char *msg, volatile int l,volatile int p);
void lcd_shift(int dir, int num);

void lcd_inst(unsigned char);
void lcd_data(unsigned char);
void delay_ms(int);
void delay_us(int);
void delay_64us(void);
void delay_128us(void);

/*****

```



```
}

```

```

//*****
// Name: Timer_Init
// Function: To properly set up input capture channels, scale
//           clock.
//*****
void timer_init(void)
{
    TIOS = 0x03; // Set 0 and 1 as output compare
    TSCR2 = 0x00; // Divide Clock by 1 for 24MHz Timer Clock

    TCTL1 = 0x00; // all Chs disconnected from output
    TCTL2 = 0x00;
    TCTL3 = 0x00; //
    TCTL4 = 0x00;
    TC1 = INTERRUPT; //set timer interval to interrupt every 100us
    TCNT = 0; // clear main counter
    TSCR1 = 0x80; // Turn on Timer Port, Disable Fast Flag Clear
}

```

```

//*****
* Name: initLCD
* Function: to setup control lines to LCD for write mode
* and enable screen
*
//*****/
void initLCD(void)
{
    //initialize LCD
    PTP = 0x00; //set LCD to write mode (PTP bit 0 = 0)
                //set register select for instruction mode(P bit 1 = 0)
                //set enable bit low (PTP bit 2 = 0)

    lcd_inst(0x3c); //send function set command
    delay_ms(32); //delay 32ms
    lcd_inst(0x3c);
    delay_ms(1); //delay 3ms
    lcd_inst(0x3c);
    delay_ms(1);
    lcd_inst(0x3c); //set 8 bit interface and # of lines = 2
    delay_ms(1);
    lcd_inst(0x06); //set entry mode to auto increment cursor
    delay_ms(1);
    lcd_inst(0x0C); //turn on display and cursor blink
    delay_ms(1);
    lcd_inst(0x01); // clear screen
    delay_ms(1);
}

```

```

    lcd_inst(0x80); //set DRAM address to home
    delay_ms(1);
    lcd_inst(0x02); //home cursor
    delay_ms(1);
}
/*****
*
* Name: lcd_pos
* Function: move cursor to correct line so text flows from line
*           1-4
*
*****/

void lcd_pos(void)
{
    switch (pos)
    {
        case 20:
        {
            switch(line)
            {
                case 0:
                {
                    line++;
                    pos = 0;

                    lcd_inst(0xc0); //move to second line
                    break;
                }

                case 1:
                {
                    line++;
                    pos = 0;

                    lcd_inst(0x94); //move to third line
                    break;
                }

                case 2:
                {
                    line++;
                    pos = 0;

                    lcd_inst(0xd4); //move to fourth line
                    break;
                }

                case 3:
                {
                    line = 0;

```

```

        pos = 0;

        lcd_inst(0x80); // move to first line
        break;
    }

    default:
    {
        line = 0;
        pos = 0;

        lcd_inst(0x80);
        break;
    }
}
break;
}

default:
{
    if(pos > 19)
    {
        pos = 0;
    }

    break;
}
}

}

/*****
*
* Name: lcd_clear
* Function: clear screen, move cursor home, reset counts for line
*           and character pos
*
*****/

void lcd_clear()
{
    lcd_inst(0x01); // clear screen
    delay_ms(3);
    lcd_inst(0x80); // move to first line
    delay_ms(3);
    lcd_inst(0x02); //home cursor
    delay_ms(3);
    pos = 0;
    line = 0;
}

```

```

/*****
*
* Name: lcd_inst
* Function: send instructions to LCD screen
*
*****/

void lcd_inst(unsigned char i)
{
    PTP  &= 0xf8; //set RS & RW  to instruction mode (both 0)
           // & clear 'E
    PORTA = i;    //send data to lcd
    PTP  |= 0x04; //Pulse 'E high

    delay_us(50); //wait at least 24us for instruction to be processed

    PTP  &= 0xfb; //clear 'E' to 0
}

/*****
*
* Name: lcd_data
* Function: send data to LCD screen
*
*****/

void lcd_data(unsigned char i)
{
    PTP  &= 0xfc; //clear RS and RW bits
    PTP  |= 0x02; //enable RS for data write
    PORTA = i;    //send data to lcd
    PTP  |= 0x04; // set E high for enable
    delay_us(50);
    PTP  &= 0xfb; //clear 'E' line to 0
    pos++;
    lcd_pos();
}

/*****
*
* Name: lcd_shift
* Function: shift cursor left or right and adjust cursor counter
*
*****/

void lcd_shift(int dir, int num)
{
    int i = 0;
    switch(dir)
    {
        case 0:

```

```

    {
        for(i = 0; i < num; i++)
        {
            lcd_inst(0x14); //shift cursor right
            pos++;
            if(pos > 19)
                lcd_pos();
        }
        break;
    }

case 1:
{
    for(i = 0; i < num; i++)
    {
        pos--;
        if(pos < 0)
        {
            pos = 20;
            lcd_pos();
            pos = 19;
        }

        lcd_inst(0x10); //shift cursor left
    }
    break;
}
default:
{
    for(i = 0; i < num; i++)
    {
        lcd_inst(0x14); //shift cursor right
        pos++;
        if(pos > 19)
            lcd_pos();
    }
    break;
}
}
}

/*****
*
* Name: lcd_print
* Function: print characters to designated line and cursor position
*
*****/

void lcd_print(char *msg, volatile int l, volatile int p)
{
    int i = 0;
    switch(l)

```

```

{
    case 0: {
        lcd_inst(0x80); // move to first line
        break;
    }
    case 1:{
        lcd_inst(0xc0); // move to 2nd line
        break;
    }
    case 2:{
        lcd_inst(0x94); // move to 3rd line
        break;
    }
    case 3:{
        lcd_inst(0xd4); // move to 4th line
        break;
    }
    default: {
        lcd_inst(0x80); //move to 1st line
        break;
    }
}

pos = 0;

if(p < 0)
{
    p = 0;
}

if(p > 20)
{
    p = 20;
}

lcd_shift(0,p); // shift cursor right

pos = p;
line = 1;

while(*msg)
{
    lcd_data(*msg++);
    delay_ms(3);
}

}

/*****
*
* Name: delay_ms(int t)

```

```

* Function: provide a delay lasting 1*t ms between writes to LCD.
* Integer 't' corresponds to multiplier of 1ms, where t = 1 is
* a 1 ms delay, t = 2 is 2ms, etc
*

```

```

*****/

```

```

void delay_ms(int t)
{

```

```

    int i = 0;
    if((void *)t == NULL)
    {
        t = 1;
    }

```

```

    for(i=0;i<t;i++)
    {
        TCO=TCNT+32000u;
        while(!(TFLG1 & 0x01));
        TFLG1 = 0x01;
    }
}

```

```

/*****

```

```

* Name: delay_64us
* Function: provide a delay of 64us using timer port
*

```

```

*****/

```

```

void delay_64us(void)
{

```

```

    TCO=TCNT + 2048u;
    while(!(TFLG1 & 0x01));
    TFLG1 = 0x01;
}

```

```

/*****

```

```

* Name: delay_128us
* Function: provide a delay of 128us using timer port
*

```

```

*****/

```

```

void delay_128us(void)
{

```

```

    TCO=TCNT+ 4096u;
    while(!(TFLG1 & 0x01));
    TFLG1 = 0x01;
}

```

```

/*****

```

```

* Name: delay_us

```

```

* Function: provide a delay of 2.5*t us using timer port
*
*****/
void delay_us(int t)
{
    TC0=TCNT+ (32u * t);
    while(!(TFLG1 & 0x01));
    TFLG1 = 0x01;
}

/***** ISR Vectors *****/
#include <hidef.h>
#include <start12.h>
#include "my_vectors.h"

#pragma CODE_SEG __NEAR_SEG NON_BANKED /* Interrupt section for this
module. Placement will be in NON_BANKED area. */
__interrupt void UnimplementedISR(void) {
    /* Unimplemented ISRs trap.*/
    asm BGND;
}

typedef void (*near tIsrFunc)(void);
const tIsrFunc _vect[] @0xFF10 = { /* Interrupt table */
    UnimplementedISR, /* vector 119 - Vsi */
    UnimplementedISR, /* vector 118 - Reserved 119 */
    UnimplementedISR, /* vector 117 - Reserved 118 */
    UnimplementedISR, /* vector 116 - Reserved 117 */
    UnimplementedISR, /* vector 115 - Reserved 116 */
    UnimplementedISR, /* vector 114 - Reserved 115 */
    UnimplementedISR, /* vector 113 - Reserved 114 */
    UnimplementedISR, /* vector 112 - Reserved 113 */
    UnimplementedISR, /* vector 111 - Reserved 112 */
    UnimplementedISR, /* vector 110 - Reserved 111 */
    UnimplementedISR, /* vector 109 - Reserved 110 */
    UnimplementedISR, /* vector 108 - Reserved 109 */
    UnimplementedISR, /* vector 107 - Reserved 108 */
    UnimplementedISR, /* vector 106 - Reserved 107 */
    UnimplementedISR, /* vector 105 - Reserved 106 */
    UnimplementedISR, /* vector 104 - Reserved 105 */
    UnimplementedISR, /* vector 103 - Reserved 104 */
    UnimplementedISR, /* vector 102 - Reserved 103 */
    UnimplementedISR, /* vector 101 - Reserved 102 */
    UnimplementedISR, /* vector 100 - Reserved 101 */
    UnimplementedISR, /* vector 99 - Reserved 100 */
    UnimplementedISR, /* vector 98 - Reserved 099 */
    UnimplementedISR, /* vector 97 - Reserved 098 */
    UnimplementedISR, /* vector 96 - Reserved 097 */
    UnimplementedISR, /* vector 95 - Reserved 096 */
    UnimplementedISR, /* vector 94 - Reserved 095 */
    UnimplementedISR, /* vector 93 - Reserved 094 */
    UnimplementedISR, /* vector 92 - Reserved 093 */
}

```

```

UnimplementedISR, /* vector 91 - Reserved 092 */
UnimplementedISR, /* vector 90 - Reserved 091 */
UnimplementedISR, /* vector 89 - Reserved 090 */
UnimplementedISR, /* vector 88 - Reserved 089 */
UnimplementedISR, /* vector 87 - Reserved 088 */
UnimplementedISR, /* vector 86 - Reserved 087 */
UnimplementedISR, /* vector 85 - Reserved 086 */
UnimplementedISR, /* vector 84 - Reserved 085 */
UnimplementedISR, /* vector 83 - Reserved 084 */
UnimplementedISR, /* vector 82 - Reserved 083 */
UnimplementedISR, /* vector 81 - Reserved 082 */
UnimplementedISR, /* vector 80 - Reserved 081 */
UnimplementedISR, /* vector 79 - Vxsramav */
UnimplementedISR, /* vector 78 - Vxsei */
UnimplementedISR, /* vector 77 - Vxst7 */
UnimplementedISR, /* vector 76 - Vxst6 */
UnimplementedISR, /* vector 75 - Vxst5 */
UnimplementedISR, /* vector 74 - Vxst4 */
UnimplementedISR, /* vector 73 - Vxst3 */
UnimplementedISR, /* vector 72 - Vxst2 */
UnimplementedISR, /* vector 71 - Vxst1 */
UnimplementedISR, /* vector 70 - Vxst0 */
UnimplementedISR, /* vector 69 - Vpit3 */
UnimplementedISR, /* vector 68 - Vpit2 */
UnimplementedISR, /* vector 67 - Vpit1 */
UnimplementedISR, /* vector 66 - Vpit0 */
UnimplementedISR, /* vector 65 - VReserved 65 */
UnimplementedISR, /* vector 64 - Vapi */
UnimplementedISR, /* vector 63 - Vlvi */
UnimplementedISR, /* vector 62 - VReserved 62 */
UnimplementedISR, /* vector 61 - Vsci5 */
UnimplementedISR, /* vector 60 - Vsci4 */
UnimplementedISR, /* vector 59 - Vsci3 */
UnimplementedISR, /* vector 58 - Vsci2 */
UnimplementedISR, /* vector 57 - Vpwmesdn */
UnimplementedISR, /* vector 56 - Vportp */
UnimplementedISR, /* vector 55 - Vcan4tx */
UnimplementedISR, /* vector 54 - Vcan4rx */
UnimplementedISR, /* vector 53 - Vcan4err */
UnimplementedISR, /* vector 52 - Vcan4wkup */
UnimplementedISR, /* vector 51 - VReserved 51 */
UnimplementedISR, /* vector 50 - VReserved 50 */
UnimplementedISR, /* vector 49 - VReserved 49 */
UnimplementedISR, /* vector 48 - VReserved 48 */
UnimplementedISR, /* vector 47 - VReserved 47 */
UnimplementedISR, /* vector 46 - VReserved 46 */
UnimplementedISR, /* vector 45 - VReserved 45 */
UnimplementedISR, /* vector 44 - VReserved 44 */
UnimplementedISR, /* vector 43 - Vcan1tx */
UnimplementedISR, /* vector 42 - Vcan1rx */
UnimplementedISR, /* vector 41 - Vcan1err */
UnimplementedISR, /* vector 40 - Vcan1wkup */

```

```

UnimplementedISR, /* vector 39 - Vcan0tx */
UnimplementedISR, /* vector 38 - Vcan0rx */
UnimplementedISR, /* vector 37 - Vcan0err */
UnimplementedISR, /* vector 36 - Vcan0wkup */
UnimplementedISR, /* vector 35 - Vflash */
UnimplementedISR, /* vector 34 - Veprom */
UnimplementedISR, /* vector 33 - Vspi2 */
UnimplementedISR, /* vector 32 - Vspi1 */
UnimplementedISR, /* vector 31 - Viic0 */
UnimplementedISR, /* vector 30 - VReserved 30 */
UnimplementedISR, /* vector 29 - Vcrgscm */
UnimplementedISR, /* vector 28 - Vcrgplllck */
UnimplementedISR, /* vector 27 - Vtimpabovf */
UnimplementedISR, /* vector 26 - Vtimmdcu */
UnimplementedISR, /* vector 25 - Vporth */
UnimplementedISR, /* vector 24 - Vportj */
UnimplementedISR, /* vector 23 - Vatd1 */
UnimplementedISR, /* vector 22 - Vatd0 */
UnimplementedISR, /* vector 21 - Vscil */
UnimplementedISR, /* vector 20 - Vsci0 */
UnimplementedISR, /* vector 19 - Vspi0 */
UnimplementedISR, /* vector 18 - Vtimpaie */
UnimplementedISR, /* vector 17 - Vtimpaaovf */
UnimplementedISR, /* vector 16 - Vtimovf */
UnimplementedISR, /* vector 15 - Vtimch7 */
UnimplementedISR, /* vector 14 - Vtimch6 */
UnimplementedISR, /* vector 13 - Vtimch5 */
UnimplementedISR, /* vector 12 - Vtimch4 */
UnimplementedISR, /* vector 11 - Vtimch3 */
UnimplementedISR, /* vector 10 - Vtimch2 */
UnimplementedISR, /* vector 09 - Vtimch1 */
&isr_t1, /* vector 08 - Vtimch0 */
UnimplementedISR, /* vector 07 - Vrti */
UnimplementedISR, /* vector 06 - Virq */
UnimplementedISR, /* vector 05 - Vxirq */
UnimplementedISR, /* vector 04 - Vswi */
UnimplementedISR, /* vector 03 - Vtrap */
UnimplementedISR, /* vector 02 - Vcop */
UnimplementedISR, /* vector 01 - Vclkmon */
_Startup /* Vector 00 - VReset */
};

```

Acknowledgments

Dr. Jones

Thank you for all your help and for pushing me to work through all my setbacks. You have been a great academic advisor, thesis advisor, and friend. You always have great ideas and even when you do not know the answer, you push me to think. I owe much of my future success to you.

Ashley Brauning

I do not know how I would have ever finished this thesis without you. You have always supported me and put up with the long hours I sometimes put in, both for senior projects and my honors project. You never stopped believing in me and pushed me to work hard. I can never thank you enough for your support, encouragement, and understanding.

God

I owe the most to you. You were always there for me in good times and bad, and you took care of me. I know that with you, nothing is impossible. You gave me the strength, endurance, concentration, and support from loved ones that I needed to finish this thesis. I give all the praise to you and pray that I will touch the lives of others as an engineer.