DESIGN OF A REAL-TIME DIGITAL SIGNAL PROCESSING AUDIO PROCESSING TECHNIQUE

An Honors Fellow Thesis

by

CHRISTOPHER MATTHEW JAGIELSKI

Submitted to the Office of Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

HONORS UNDERGRADUATE RESEARCH FELLOW

May 2012

Major: Electrical Engineering

DESIGN OF A REAL-TIME DIGITAL SIGNAL PROCESSING AUDIO PROCESSING TECHNIQUE

An Honors Fellow Thesis

by

CHRISTOPHER MATTHEW JAGIELSKI

Submitted to Honors and Undergraduate Research
Texas A&M University
in partial fulfillment of the requirements for the designation as

HONORS UNDERGRADUATE RESEARCH FELLOW

Approved by:

Research Advisor: Deepa Kundur
Associate Director for Honors and Undergraduate Research: Duncan MacKenzie

May 2012

Major: Electrical Engineering

ABSTRACT

Design of a Real-Time Digital Signal Processing Audio Processing Technique.
(May 2012)

Christopher Matthew Jagielski
Department of Electrical and Computer Engineering
Texas A&M University

Research Advisor: Dr. Deepa Kundur
Department of Electrical and Computer Engineering

This thesis outlines the design of a real-time digital signal processing technique for pitch detection and analysis via spectral analysis. A sound's musical pitch can be determined from its fundamental frequency, and a note that is said to be "out of tune" will have a fundamental frequency that deviates from the "in-tune" frequency by a number of Hertz. This technique determines the real-time fundamental frequency of an inputted audio sample and determines its pitch relative to standard "in-tune" frequencies. The design was first simulated in the MATLAB software environment and tested with controlled pitches before it was examined in the hardware environment to analyze unknown audio inputs in real time. The design was implemented to test feasibility with the Texas Instruments TMS320DM6437 board (DaVinci Video Processor series) and the Texas Instruments TMS320C6713 Digital Signal Processing Starter Kit using MATLAB/Simulink for software interfacing. This design is unique because of its flexibility and adaptation in a graphical user interface, which allows for rapid prototyping and ease of use.

DEDICATION

I would like to dedicate this thesis work to my fiancée, Lauren.

ACKNOWLEDGMENTS

First, I would like to thank my family for their love and their encouragement throughout my educational endeavors over the years.

Second, I owe gracious thanks to the Electrical and Computer Engineering Department at Texas A&M University and my scholarship donors for assisting me with my educational endeavors these past four years.

Lastly, I thank my advisor, Dr. Deepa Kundur, for taking me in as a research student and guiding me in this process with so much enthusiasm and support.

NOMENCLATURE

A/D	Analog-to-Digital
D/A	Digital-to-Analog
DSK	Digital Signal Processing Starter Kit
DSP	Digital Signal Processing
EVM	Evaluation Model
FFT	Fast Fourier Transform
GUI	Graphical User Interface
IFFT	Inverse Fast Fourier Transform
RAM	Random-Access Memory
ROM	Read-Only Memory
TI	Texas Instruments

TABLE OF CONTENTS

Page

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

This thesis outlines the design of an algorithm used in a digital signal processing system to detect and display the fundamental frequency of a real-time audio input signal. Digital signal processing (DSP) is a field of study involved with discrete time signals and the methods used to represent and modify them [1]. The physical world is an analog world, but computers and modern computer-based systems are digital and use discrete signals. Common DSP applications include the measuring, filtering and compressing of real-world signals [1], [2]. There are many practical uses of DSP, which can be found in the fields of audio processing, video processing, sonar and radar signal processing, digital image processing, biomedical signal processing, and processing for telecommunications, amongst others [1], [2].

Digital systems are advantageous over analog systems because of their greater control over accuracy and resolution and their predictable and reproducible behavior [1]. However, they are limited in their speed by the clock rate of the hardware available and they require extra processing via analog-to-digital (A/D) and digital-to-analog (D/A) convertors in order to interface with the real world [1], [3].

This thesis follows the style and format of *IEEE Transactions, Journals, and Letters*.

Real-time DSP involves the processing of signals in real-time such that there is no noticeable delay in the system output that is produced by the system itself. For example, the video enhancement of a live television broadcast would require that the transmitted television signal would not have a significant lag from the original video footage. For most applications, the delay needs to be minimized such that the signal's end user (i.e. a human being) does not notice its existence. In the case of a musical audio tuner, the end user would be the human ear.

Musical pitch

A sound's musical pitch can be determined from its fundamental frequency, the single frequency in a sound sample's frequency spectrum that has the highest amplitude [1], [4]. A note that is said to be "out of tune" will have a fundamental frequency that deviates from the "in-tune" frequency by any number of Hertz. Musical pitches are standardized by their frequency in Hertz [5]. See Table 1 for a list of some of the standard frequencies associated with musical pitches [5].

Table 1. Standard Pitches and Frequencies (in Hertz)

Note/pitch name	Frequency (Hz)
A	220
A sharp / B flat	233
B	247
C	262
C sharp / D flat	277
D	294
D sharp / E flat	311
E	330
F	349
F sharp / G flat	370
G	392
G sharp / A flat	415

The musical pitches rotate through these twelve note names in western music [5]. Two notes may have the same pitch name but be different frequencies; notes of the same name are mathematically related as multiples of each other. For example, the frequency 220Hz corresponds to the pitch 'A' as does the frequency 440Hz, which is the first frequency multiplied by two.

Pitch detection methods

Mathematically, an audio sample's frequency spectrum can be found by taking the Fourier Transform [1]. Typical DSP systems and hardware use the discrete Fourier Transform or Fast Fourier Transform (FFT) rather than a continuous version of the transform [1], [2]. Often the FFT algorithms are modified to optimize computing speed rather than precision in order to maximize their effectiveness in a real-time DSP application.

The continuous-time Fourier transform can be mathematically shown by the following formula [1]:

$$F(\omega) = \int_{-\infty}^{\infty} f(t) e^{-2\pi i \omega t}\, dt$$

where $f(t)$ is the time domain signal and $F(\omega)$ is the frequency domain signal. The resulting frequency domain signal is a series of sinusoids with complex amplitudes [1]. In practical applications such as the determining of an audio sample's frequency spectrum, only the real parts of the transform are kept. Also note that digital systems use the discrete-time Fourier transform and may compute the FFT in a number of different ways based on the unique DSP processor's architecture [2].

Real-time DSP design considerations

Real-time DSP systems require both high speed and high accuracy. Initial design considerations of a real-time system include the algorithm used, the input signal's

sampling rate, the speed required to meet the application's throughput, and the precision and dynamic range of the data representation [3]. Also there are always practical requirements for size, cost, and power consumption that must be met in any particular system [3].

In addition, real-time DSP systems are usually supported with specialized microprocessors [2], [3]. This allows for the implementation of state-of-the-art algorithms that can be very computationally intensive. Many digital signal processors are programmable such that they can be modified for more specialized capability [2], [3]. A processor will have local random-access memory (RAM), read-only memory (ROM), and other on-chip registers to store results [3].

In this design, two specific commercially available DSP hardware systems will be used. This removes many system design requirements such that only the algorithm and its implementation will be designed, tested, and modified as needed.

The specific hardware selected for this design application allows for rapid prototyping and ease of use. The developed algorithm will be loaded onto the DSP board, able to use its built in A/D and D/A convertors, its RAM and its ROM [6].

Motivation

Technology in the modern world incorporates a great deal of sensing and understanding, work that is becoming more common and sophisticated with advanced computing systems. Automatic pitch detection plays a central role as a fundamental block of digital audio processing, and many common applications, such as voice recognition, would not be possible without this important processing capability.

This design approach focuses on a lightweight algorithm for real-time implementation. The algorithm targets the frequency domain for its analysis, as this is the best domain for audio pitch detection because of its unique ability to separate individual musical pitches. The use of lightweight techniques in the design allows for simpler programming and faster execution, both of which are very helpful attributes in a real-time DSP system.

As a whole, this design procedure draws upon the recent advancements of DSP technology. Programing the DSP board will be accomplished via higher level software programs and not through assembly language, as is the case for older DSP boards and processors [2], [7]. As previously mentioned, this capability allows for a much simpler, faster, and easier design process. It also mimics standard industry procedures that allow for rapid prototyping, an attribute that is largely advantageous while testing the feasibility of real-time implementation on two different commercially available DSP kits [2], [7].

Organization of the thesis

There are four chapters in this thesis, which provides introduction, methods, results, and summary and conclusions. Chapter I introduces DSP and real-time processing applications and also explains the musical theory of pitch detection and the basic methodology for its analysis. The chapter then explores the motivation for this type of advanced digital design work.

Chapter II presents the detailed information regarding the designing, testing, and verifying of the pitch detection algorithm and its software and hardware implementations. Progress is shown at each stage of the development process, and unique design procedures and considerations are also deliberated.

Chapter III discusses the results of the pitch detection algorithm's implementation in multiple environments. This section outlines the many testing procedures used to verify the algorithm and explains any alterations that were made in the design process to handle unexpected setbacks.

Chapter IV summarizes the entire design process and draws conclusions from its successes and failures. Each step of the design process is scrutinized to discuss both their positive and negative aspects. Future work is then presented to provide insight into potential advancements that can be made from this particular audio design and from the DSP programming design process itself.

CHAPTER II

METHODS

This chapter provides the design methodology of a real-time digital signal processing application in MATLAB/Simulink software and its hardware implementation via the Texas Instruments TMS320DM6437 Evaluation Model (EVM) and the Texas Instruments (TI) TMS320C6713 Digital Signal Processing Starter Kit (DSK). These design techniques have been adapted from the lab manuals of Dr. Deepa Kundur, Associate Professor in the Electrical & Computer Engineering Department at Texas A&M University, found in Reference [3].

Initial MATLAB software design

The pitch detection algorithm was first developed in software before being implemented in hardware in order to foster fast and easy testing. Systems in a software setting will process known inputs with expected (and perhaps even known) outputs in non-real time; this enables the designer to fully scrutinize a system and ensure that it functions as required before testing it on a more complex and unfamiliar real-time input.

The first step in this design process was to develop the actual pitch detection algorithm. The algorithm was coded in MATLAB, a powerful numerical computing environment, and can be seen in Appendix A. The MATLAB *.m* file's code in Appendix A is

organized into three main sections: generating an input audio sample, obtaining the sample's fundamental frequency, and finally assigning a musical pitch to that frequency.

To generate the input audio sample, a simple sinusoid is created with a known and arbitrarily chosen frequency. In the code found in Appendix A, that frequency is 789 Hertz. The sinusoid propagates in time in accordance with a 1 second long time vector that is sampled $2^{13} = 8192$ times. This sampling rate was initially chosen to be 8000 samples per second, but was later modified to be 8192 samples per second in order for the generated sinusoids to become allowable inputs for the next simulation stage in this design.

After the generated sinusoid is saved as a *.wav* audio file, it is quantized and discretized as it is read back in as a new input in MATLAB. To obtain the frequency spectrum of the audio sample, the FFT is then applied to the digital time-domain signal. The absolute value of the result is taken next to ensure that only real values of the spectrum are kept, since musical frequencies are real values and not imaginary.

Figure 1 on the next page shows a time-domain sinusoid generated in the manner prescribed. It is sampled 8192 times over its duration of one second and has a chosen frequency of 88 Hertz. Mathematically, the sinusoid can be written as:

$$x(t) = \sin(2\pi * 88 * t).$$

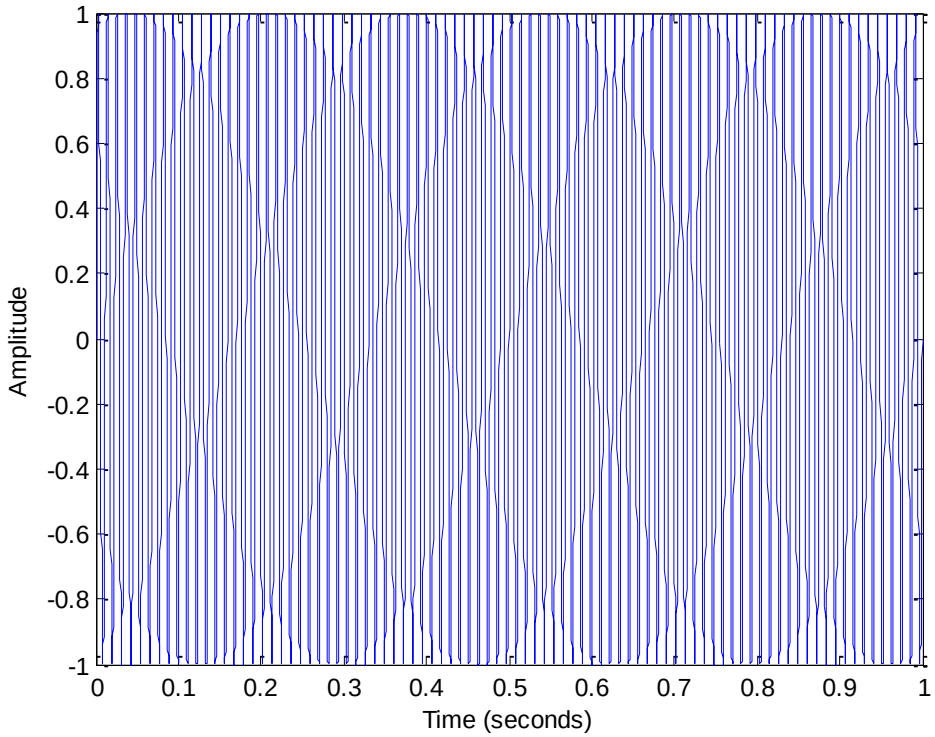

Fig. 1. Time-domain sinusoid of frequency 88 Hz over 1 second

Although the image in Figure 1 looks like a continuous sine wave, the wave is actually discretized and quantized. There are exactly 8192 points between time 0 seconds and time 1 second where the sinusoid's amplitude is plotted; these points are connected by lines to create a more graphically appealing and realistic image.

Figure 2 on the next page depicts the frequency spectrum of the sinusoid depicted in Figure 1. Note that the highest peak in Figure 2, found at 88 Hz, is also doubled at 7916 Hertz due to wrap-around error.

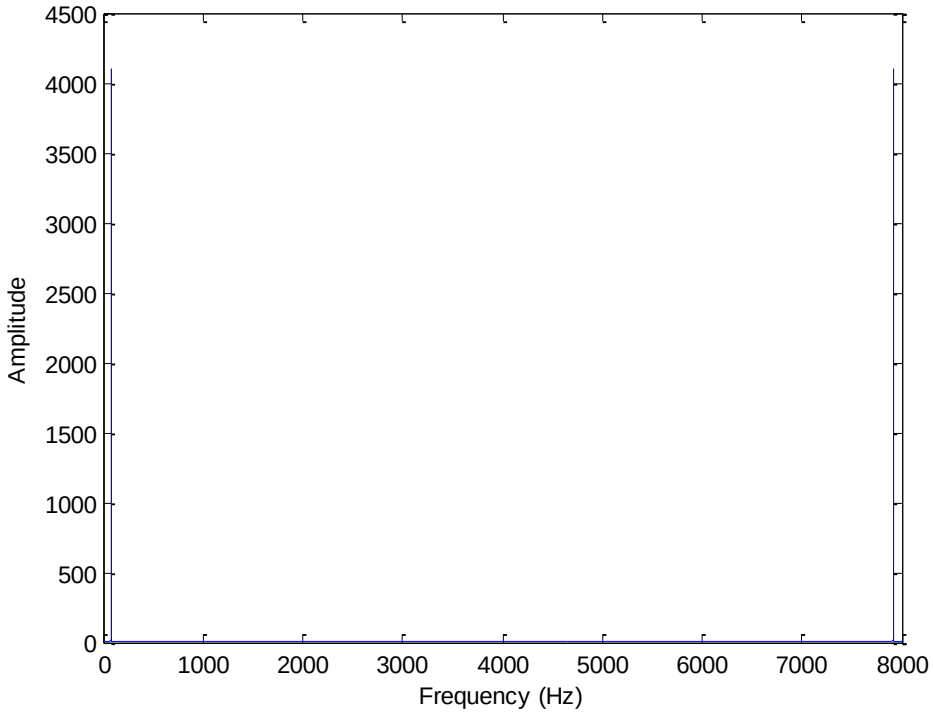

Fig. 2. Frequency spectrum of 88 Hz sinusoid

Once the fundamental frequency is found from the audio file's frequency spectrum, it is cross-referenced against a single set of musical pitch ranges. The frequency number is doubled or halved until it falls into the proper range of 212 Hertz to 425 Hertz. This range was determined based on the frequencies in Table 1.

Adaptation into Simulink

The Simulink software environment allows for user friendly software design and implementation in hardware of a DSP system. Simulink works with MATLAB to provide a graphical user interface (GUI) for Digital Signal Programming and other

applications [2]. The next step in this design is to create a non-real-time Simulink model that can mimic the results of the MATLAB code in Appendix A.

The completed Simulink model for this design stage can be seen below in Figure 3.

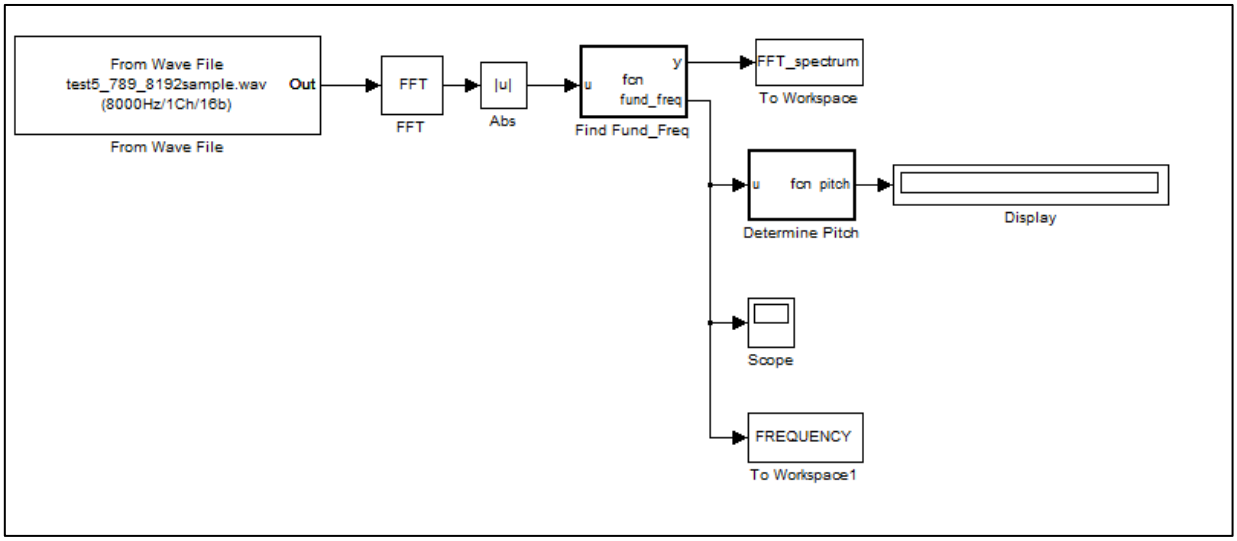

Fig. 3. Non-real-time Simulink Model

The model in Figure 3 is essentially a graphical block-diagram of the MATLAB *.m* file found in Appendix A. First, a *.wav* audio file is selected as an audio input for the model. The input parameters for this input block can be seen in Figure 4.

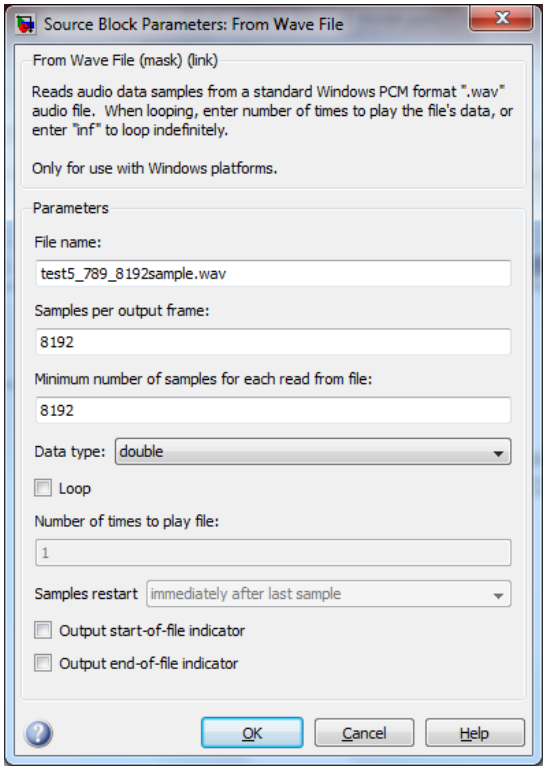

Fig. 4. Simulink input wave file source block parameters

The input audio file is first passed through Simulink's FFT block. As seen in Figure 5, the FFT algorithm used in this application is optimized for speed (rather than for computational accuracy). Note that the wave file parameters in Figure 4 show a sampling rate of 8192 Hertz. This sampling rate is used (rather than 8000, or any other arbitrary number for that matter) since, as described in Figure 5, the FFT block's input must have a power-of-2 width. Although a different power-of-two number could have been used, 8192 was chosen since it contains enough points to ensure accuracy with the FFT but not too many as to significantly slow down the simulation's computing time.

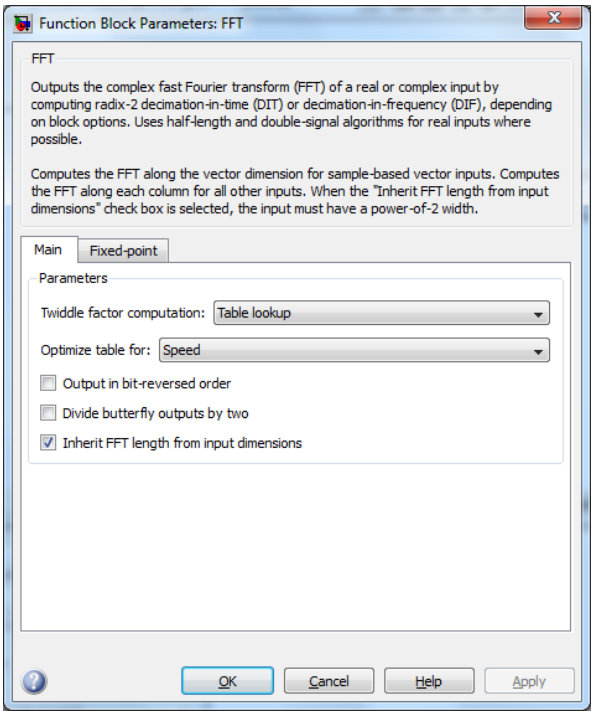

Fig. 5. Simulink FFT parameters

In order to ensure that only the real parts of the frequency spectrum are kept after the FFT is computed, the data is passed through an absolute value block.

The next sequential block found in Figure 3, titled “Find Fund_Freq”, takes the frequency spectrum as an input and produces the spectrum (unchanged) and its fundamental frequency as outputs. The algorithm to compute the fundamental frequency is based off of the previously created MATLAB code. The specific code found inside the “Find Fund_Freq” block can be seen in Appendix B.

At this point, the fundamental frequency of the input audio file has been computed. It is then displayed to the user in a number of ways. The fundamental frequency, a scalar integer, is sent to the MATLAB workspace for the user to analyze after running the Simulink model. Note that in Figure 3 the FFT spectrum is also sent out to the workspace; this provides helpful diagnostic capabilities to ensure that all of the blocks in the model are working correctly. After the model has run, this data will be available for viewing and manipulation as desired.

The integer fundamental frequency can be immediately seen by the user via an output scope placed on its data line. Figure 6 shows the output graph provided by this scope.

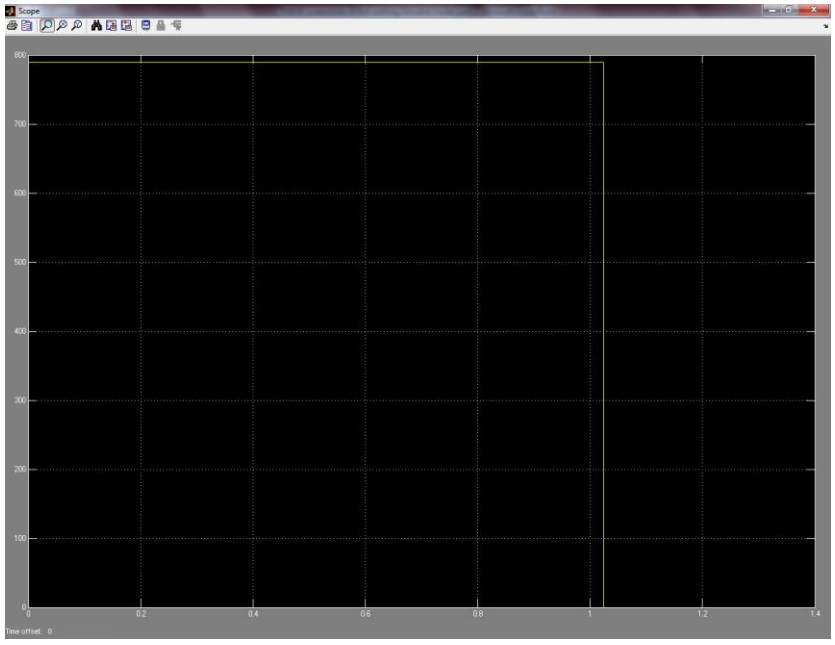

Fig. 6. Scope output showing the fundamental frequency (vertical axis, in Hertz) over time (horizontal axis, in seconds)

A zoomed-in view of the scope's output can be seen in Figure 7. This image is provided to show detail of the scope's vertical axis scale and its value of 789 Hertz at time zero.

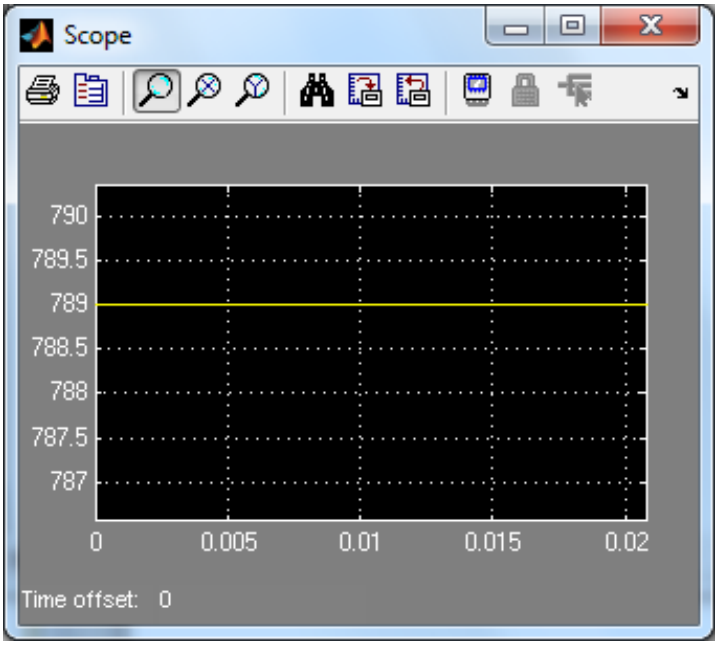

Fig. 7. Zoomed-in scope of fundamental frequency value (vertical axis, in Hertz) over time (horizontal axis, in seconds)

It is noteworthy to explain that the fundamental frequency graph seen in Figure 6 holds two values: 789 Hertz and 0 Hertz. This first value, the correct value for the input audio file, lasts for approximately 1 second in duration; this occurs because the input audio file is also 1 second in length. After this time limit, the value for the fundamental frequency drops to zero because there is no longer any input audio. Mathematically this implies

that a frequency of zero dominates the spectrum, which also holds in the time-domain since $\sin(2\pi * 0) = 0$ and there is no longer input amplitude.

Using the frequency-to-pitch conversions found in Table 1, the last step in the algorithm is to determine the unique musical pitch that corresponds to the calculated fundamental frequency. Another Simulink block of embedded MATLAB code is used to perform this; the algorithm code can be found in Appendix C. In order to properly display the pitch, since only number values are allowed, the string text is converted to type double and manipulated in a certain fashion as to display the pitch as a type double integer and the presence of the word "sharp" as a decimal.

Hardware implementation via Simulink

In order to test the feasibility of the algorithm in real-time, two Spectrum Digital Incorporated DSP boards with Texas Instruments hardware were used. New Simulink models were created and the Code Composer Studio software program was used to interface between the DSK and the host computer.

First, the Texas Instruments DM6437 EVM, with the Texas Instruments DaVinci processor series, is tested with the Simulink model. Then, the TI C6713 DSK is tested with the model. Because of unresolvable software compatibility issues, the C6713 kit is worked with more fully in this final stage of the design. In addition, the TI DM6437 board has added functionality for video processing that is not necessary for this specific

design; therefore, it is not imperative that the DM6437 EVM be used specifically for this real-time pitch detection system.

To interface with the C6713 DSK, Simulink's add-on blockset called "Target for TI C6000" is used. This blockset allows the designer to easily connect with the board's A/D and D/A convertors and to use them as the input and output, respectively, in the Simulink model. The C6713 DSK has a built-in stereo codec (TLV320AIC23) that combines the A/D and D/C convertors with 24-bits per sample and a sample frequency of 96000 Hertz [2]. The DSK has many programmable switches and LEDs which can be used for creative user interaction with the board.

Below in Figure 8 is the updated Simulink model to work with the C6713 DSK in real-time. The model is very similar to the model created in the previous design stage (seen in Figure 3), but has new additions and changes to enable real-time functionality.

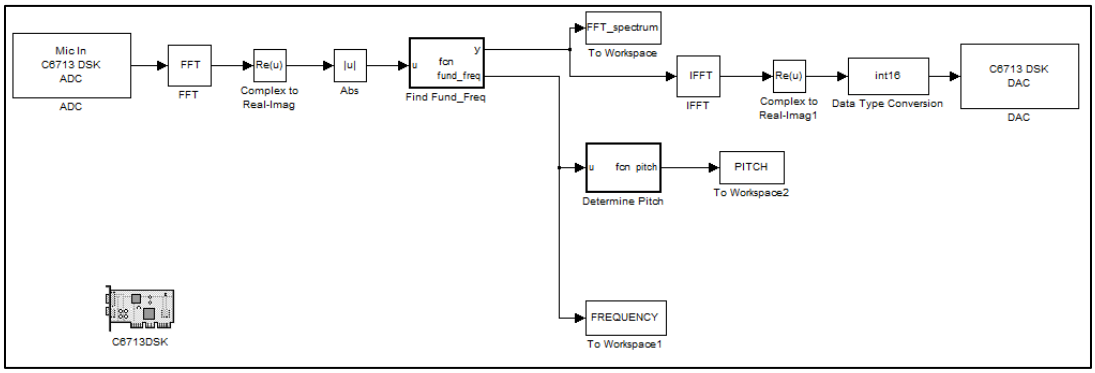

Fig. 8. Real-time Simulink model using C6713 DSK

The new input for this model is the on-board A/D convertor (labeled “ADC”) and the output is the on-board D/A convertor (labeled “DAC”). These Simulink blocks are specialized for use with the C6713 DSK and are ready to interact with the hardware without any further efforts. Shown below in Figure 9 are the source block parameters for the DSK’s A/D convertor with the actual settings used for implementation.

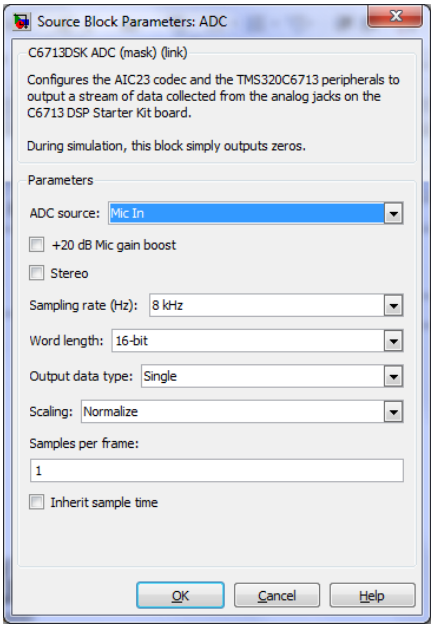

Fig. 9. C6713 DSK’s A/D convertor settings in Simulink

The other necessary addition for real-time processing is the floating “C6713DSK” block as seen in Figure 8. This highly specialized block in Simulink will set the configuration parameters for Simulink to correctly work with Code Composer Studio and the DSK [3]. In the Simulink simulation configuration parameters, it is necessary to make the following changes. First, in the “Solver” section, set stop time to ‘inf’, choose fixed-

step, choose discrete solver, set the fixed-step size to 1/8000, and choose Single Tasking as the tasking mode for periodic sample times [3]. Then in the "Real-Time Workshop" section select the target selection to be "ccslink_grt.tlc" [3]. Finally, under the "Link for CCS" section, select custom project options and type "-o2 -q" in the compiler options string, set the system stack size to 8192, and under runtime set the build action to 'Build and Execute' [3].

The last changes made to the Simulink model are the additions of the complex-to-real blocks and the Inverse FFT blocks. The inverse FFT is used to recreate the time-domain audio sample from the frequency spectrum so that it can be passed through the board's D/A convertor and outputted to the user via speakers or headphones.

Note that the real-time model's two embedded MATLAB blocks contain the exact same code as those in the non-real-time model. Thus the respective codes can be found in Appendices B and C.

Figure 10, shown on the next page, displays a screenshot of Code Composer Studio Version 3.3. This version of the software was used because of its robust performance in the Windows 7 operating system. The Simulink models, depicted in Figure 3 and Figure 8, were created with MATLAB version R2009B.

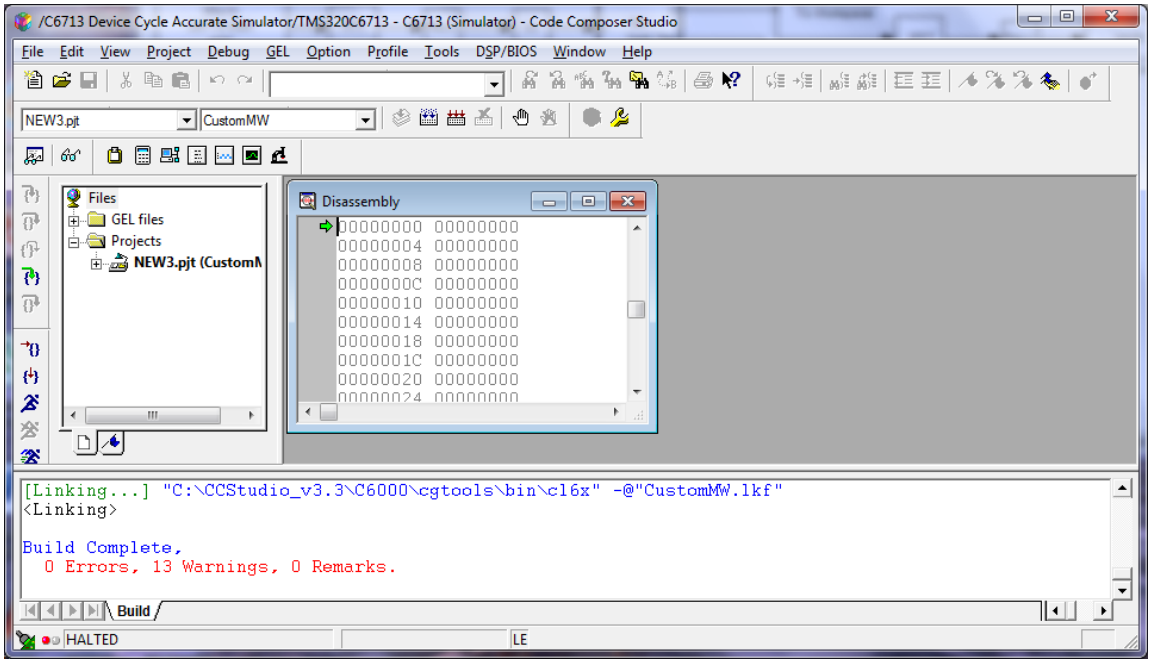

Fig. 10. Code Composer Studio Version 3.3 after real-time build

Note in Figure 10 that the build is complete, meaning that the Simulink model is converted into lower level programming code and loaded onto the DSK. A fully functional build, unlike what is observed in Figure 10, would enable real-time execution of the pitch detection algorithm allowing the user to speak/sing into a microphone attached to the DSK and have the note's pitch be shown on the computer screen.

CHAPTER III

RESULTS

Overall, both the software and hardware implementations of the pitch determining algorithm are largely operational and effective. Although feasibility is not shown with a real-time hardware configuration, the algorithm is designed and realized as desired. Uncontrollable and unexpected computer software conflicts hinder the final goal of establishing a working real-time system working through a specialized programmable DSP hardware system.

MATLAB simulation

The application's first implementation, confined entirely in a MATLAB *.m* file, is entirely successful. The code is written to handle an input *.wav* audio file and correctly analyze its frequency spectrum to identify the fundamental frequency. A simple network of conditional statements is then used to accurately determine the unique pitch associated with that frequency.

Many sample inputs are used to test this code. The most basic is a simple sinusoid with no noise modeled by the function

$$x(t) = \sin(2\pi \times freqency \times t)$$

where the $freqency$ variable is chosen at random to determine the fundamental frequency of the sample.

Various types of noise are added to the aforementioned sinusoid model in MATLAB. A random number generator is used to add varying levels of noise to the sinusoid at every point in time, thereby slightly modifying the signal. White Gaussian noise is also added via MATLAB's command $awgn(,)$ with a Signal-to-Noise Ratio of 10dB. This almost doubles the range of values for the sinusoid, from [-0.9902:0.9998] to [-1.7067:2.7949] for a generated sinusoid of frequency 836 Hz that already has random noise added to the time domain signal.

Additive noise is also tested in the frequency domain, after the audio file has been processed and converted via the FFT in MATLAB. However, this noise is found to be largely unnoticeable due to the nature of the signal's power spectrum. Since the fundamental frequency's amplitude in the spectrum is sufficiently larger than any other frequency, any random or uniform additive noise does not significantly alter the relative amplitudes in the spectrum; the fundamental frequency is always easily discernible in the spectrum.

Overall, the MATLAB code is easy to understand, executes quickly, and allows for incredibly reliable testing. Unfortunately, however, using the MATLAB file by itself is not an effective method to mimic real-time pitch detection. Inputs to the code must have been previously created and stored on the computer, and the code's design prevents it

from handling longer audio samples that could potentially have multiple toned pitches occurring sequentially through time.

Simulink simulation

Since the algorithm is implemented successfully as a MATLAB program, the first attempt in Simulink is to embed the entire previous code as a single block in the model. Unfortunately, simply reusing the code in Simulink is not effective due to the numerous idiosyncrasies of the Simulink software environment.

Once the particular design parameters of Simulink are taken into account, the new model is able to run correctly run and output the desired frequency information.

Figure 11 below shows an example display output of the Simulink model shown in Figure 3. The numbers in the display correspond to the letter "G" as determined by the algorithm's code found in Appendix C. This display result is obtained from an input audio file with a fundamental frequency of 789 Hertz.

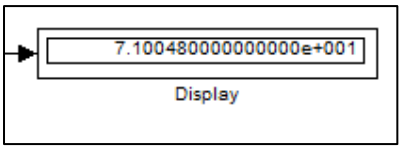

Fig. 11. Display of "G" corresponding to 789 Hertz

Test audio files are again used to verify the Simulink model; the same *.wav* files that have been created for the MATLAB *.m* file are again inputted for this process. As with the MATLAB testing, the Simulink model yields accurate results for the predetermined noisy sinusoidal inputs.

Real-time hardware implementation

To implement the algorithm on the Texas Instruments C6713 DSK, first a diagnostics test is processed on the hardware. Results of a passed test can be seen in Figure 12 below.

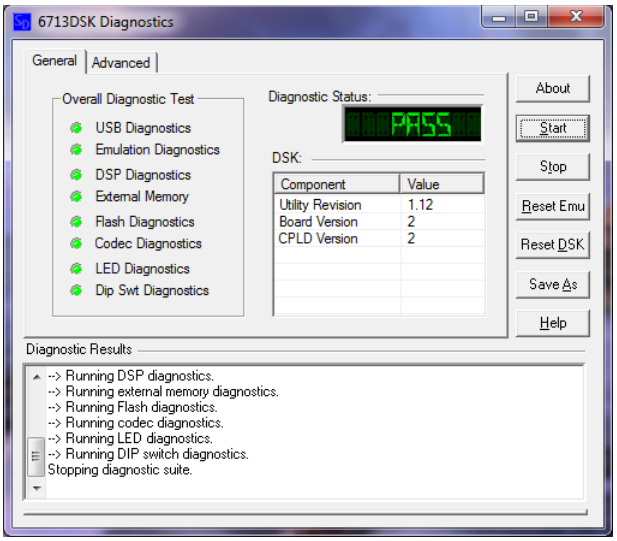

Fig. 12. Completed 6713DSK diagnostics test

Code Composer Studio version 3.3 is then opened in order to completely connect the DSK with the computer software. After this, MATLAB and Simulink can be opened in

order to open and build the real-time Simulink model (pictured in Figure 8). This process can take minutes to complete depending on the speed of the host computer that is running these programs.

While designing the real-time Simulink model, it was difficult to determine how to output the fundamental frequency, an integer scalar, or its corresponding pitch, of programming type string, as an output from the board. Normally the inputs and outputs of the DSK would be the same type of data, i.e. both audio. The best resolution was to have the pitch determined and sent to the MATLAB workspace to be watched and analyzed as the user's leisure. Unfortunately, this approach remains unverified because of the problems encountered with the real-time implementation.

After the real-time Simulink model is built, Code Composer Studio displays that the build is complete and has zero errors, much like the message seen in Figure 10. Unfortunately, obtaining results at this stage does not imply that the DSK and host computer are fully communicating correctly. Complete real-time operation with the DSK is found to be unfeasible due to unfixable software differences related to TI's "Real Time Data eXchange" protocol created for alternate debugging methods.

In order to bypass these negative results, multiple software changes were made on the host computer. The C6713 DSK was tested with Code Composer Studio versions 3.1, 3.3, and 4.0. Unfortunately, these changes did not change the outcome of the build.

Neither the C6713 DSK nor the DM6437 EVM is compatible with Code Composer Studio version 4, as they are originally designed to work with versions 3.1 and 3.3, respectively. In the best case scenario, Code Composer Studio version 3.3 ran with administrator privileges in compatibility mode for Windows XP with Service Pack 3. The only computers available for use have the Windows 7 operating system, so it is unknown what role the operating system truly plays in this situation.

Tests using different versions of MATLAB (and hence also Simulink) were also performed but did not produce successful results. Simulink models produced in MATLAB R2011b are completely incompatible with Code Composer Studio versions 3.1 and 3.3. Furthermore, Code Composer Studio version 3.1 would not run or open under any circumstances on a Windows 7 operating system.

CHAPTER IV

SUMMARY AND CONCLUSIONS

Summary

This design created an algorithm for a real-time pitch detection method using the Simulink software environment. First, the pitch detection algorithm was created and tested in a MATLAB *.m* file to ensure its precision. Then the algorithm was converted for simulation in a Simulink model. Finally the Simulink model was modified to allow for the testing of real-time processing using the TI C6713 DSK and the TI DM6437 EVM.

The first step in the design process, creating the MATLAB algorithm found in Appendix A, is 100% successful for its purpose. The algorithm passes every test and remains reliable and robust for a variety of input signals with varying amounts of noise. Drawbacks to this stage in the design include the lack of real-time processing and the necessity for the algorithm to run in the MATLAB software environment rather than on a portable and specialized DSP system.

Like the first step in the design process, the second step is also very productive; however, the two stages suffer from many of the same disadvantages. The Simulink model, seen in Figure 3, can only handle input audio files that have previously been created with a sampling rate of a power of 2 (in Hertz) in order to comply with the FFT

block's parameters. A new problem also emerges with the Simulink model regarding the output of the fundamental frequency and pitch. It is simple to graphically display the fundamental frequency's value over time, but unlike the previous MATLAB code there is no intuitive or pleasing method on how to share the name of the frequency's associated musical pitch with the user. As mentioned in Chapter II, a partial resolution has been created to convert the type string pitch name text into a type double numerical value to be displayed (see Figure 11 for an example). Overall, the Simulink model's operation as a prototype pitch detection system is complete as designed.

The third stage in this design is to test the real-time functionality of the audio pitch detection algorithm. Although progress is hindered by computer software incompatibility issues, the previously designed algorithm and Simulink model for interfacing are complete and without error. Due to time and financial constraints, the number of software packages made available for real-time implementation and testing has been limited. The TI DM6437 Evaluation Model from Spectrum Digital was first tested with the real-time Simulink model; unfortunately, the DM6437 could not pass the initial diagnostics test due to irreconcilable incompatibilities between the board's drivers, the host computer's operating system, and the available version of TI's Code Composer Studio software.

Using the C6713 DSK for real-time implementation of the algorithm yielded much better results than those of the DM6437, but the system still has slight software

incompatibility problems. The Simulink model seen in Figure 8 shows the final model that is downloaded to the DSK for real-time processing. The model builds effectively in Code Composer Studio with no errors, but real-time implementation does not occur. Without alternative software versions for MATLAB/Simulink, Code Composer Studio, and the Windows operating system it has become impractical to further troubleshoot the errors with real-time performance on the TI C6713 DSK.

These results show that digital signal programming is much more user friendly now and that many new applications can be easily developed. Hardware programming is no longer necessary for complicated and unique algorithms to be realized, as software can be developed and loaded onto a DSP board to create new functionalities in a system.

Conclusions

Overall, the methodology used in this design is helpful and successful. Creating the algorithm in a MATLAB *.m* file was instrumental in verifying its correctness and was a solid starting base to help create the Simulink models. Developing the algorithm in Simulink required some time to experience the learning curve to ensure that it was functioning correctly and as well as possible.

Unfortunately, ultimate feasibility was not shown for a real-time programmable DSP hardware system or kit; however, with the proper host computer configurations it is

undeniably expected that the pitch detection algorithm which was developed through the aforementioned methods will execute flawlessly and produce the expected results.

Future Work

The future applications and extensions of this work are numerous and varied in nature. First, it would be advantageous to start fresh with this design on a new computer running Windows XP. From there, the most compatible software versions that were designed for the C6713 DSK could be loaded onto the computer. The same design steps would be taken to create the three stages; hopefully this would help find a root cause to the problems experienced and would result in a fully functioning real-time DSP hardware based automatic pitch detection system.

To extend the work of this design, advanced techniques related to the full audio spectrum of the input audio sample could be created. It would be interesting to note and modify the entire spectrum of the audio sample (i.e. make a violin sound like a trumpet) rather than focus on one single frequency.

In general, it seems that DSP programming will progress with the creation of more user friendly interfacing systems. In the future there may be more powerful and complex software systems that provide designers greater flexibility in creating computationally intense algorithms without being limited by the architecture and functionality of the digital signal processor and its on-board ROM, RAM, A/D and D/A convertors, etc.

One would expect more projects similar to this design to be realized as the DSP hardware and software systems become more popular, commercially available, and used in academic settings [7], [8].

REFERENCES

[1] J. G. Proakis and D. G. Manolakis, Digital Signal Processing. Upper Saddle River, New Jersey: Pearson Prentice Hall, 2007.

[2] T. B. Welch, C. H. G. Wright and M. G. Morrow, *Real-Time Digital Signal Processing*. Boca Raton, FL: CRC Press, 2006.

[3] D. Kundur. ECEN 448: Real-time DSP. 2012(April 4), 2012. Available: http://www.ee.tamu.edu/~deepa/ecen448/.

[4] Yuh-Shyang Wang, Ting-Yao Hu and Shyh-Kang Jeng. "Automatic transcription for music with two timbres from monaural sound source," presented at the IEEE International Symposium on Multimedia (ISM), Taichung, Taiwan, December 13, 2010, pp. 314–317.

[5] M. Kennedy. The Oxford Dictionary of Music online. 2012(April 4), 2012. Available: http://www.oxfordmusiconline.com/subscriber/book/omo_t237.

[6] Texas Instruments. TMS320C6713 DSP starter kit (DSK). 2012(April 4), 2012. Available: http://www.ti.com/tool/tmdsdsk6713.

[7] L. Atlas and P. Duhamel. “Recent developments in the core of digital signal processing,” *IEEE Signal Processing Magazine*, vol 16, pp. 16–31, January, 1999.

[8] Z. Yuxi, K. Li, W. Jun, S. Jinping and W. Zulin. “Methods and experience of using Matlab and FPGA for teaching practice in digital signal processing,” presented at the 2010 International Conference on Education and Management Technology (ICEMT), Cairo, Egypt, November 3, 2010, pp. 414–417.

APPENDIX A

MATLAB SIMULATION CODE

```
clc;
clear all; close all;

%% Information
%
% Chris Jagielski
% Senior Thesis - TAMU Research Fellows Program
% B.S.E.E. Class of 2012
%
%% Create Audio Sample
% Create a sinusoidal audio sample & others that can be analyzed

Fs = 8192-1; %sampling frequency of 2^13 Hz
t = [0:1/Fs:1.0]; %time vector; note lasts 1 second
noise = rand([1,Fs+1]); %random noise
ssoid = sin(2*pi*789*t) + noise; %generate sinusoid & add noise
v = awgn(ssoid,10); %Additive White Gaussian noise
wavwrite(v, 'test5_789_8192sample.wav'); %save sample pitch as audio
file

%% Fundamental Frequency

[y Freq nbits] = wavread('test5_789_8192sample.wav'); %input sound file
data

L = length(y);
Y = fft(y); %take FFT
Yf = abs(Y); %obtain real values only

x = linspace(0,Freq,L-1); %used for plotting

figure; plot(x,Yf(1:L-1)) % Plot single-sided amplitude spectrum
title('Single-Sided Amplitude Spectrum of sound file')
xlabel('Frequency (Hz)')
ylabel('|Y(f)|')
figure; plot(x,y(1:L-1)) % Plot ORIGINAL sound file (time domain)

value = max(Yf); %find peak in spectrum
fund_freq = find(Yf==value,1) -1; %finds only 1 value that matches

%% Tuner
% Purpose: to determine the audio pitch

pitch = ('Unknown'); %initialize pitch value

f = round(fund_freq); %round to nearest integer
```

```
i=0; %index for iterations of the following loop
while ((f < 212) || (f > 425))
    if f <= 210 %lower octave
        f = 2*f;
        i = i+1;
    elseif (f > 430)
        f = 0.5*f;
        i = i+1;
    end
end

f = round(f); %make sure it's an integer (again)

%Set the pitch based on the fundamental frequency
if ((212 <= f) && (f <= 226))
    pitch = ('A');
elseif ((227 <= f) && (f <= 240))
    pitch = ('A sharp');
elseif ((241 <= f) && (f <= 254))
    pitch = ('B');
elseif ((255 <= f) && (f <= 269))
    pitch = ('C');
elseif ((270 <= f) && (f <= 285))
    pitch = ('C sharp');
elseif ((286 <= f) && (f <= 302))
    pitch = ('D');
elseif ((303 <= f) && (f <= 320))
    pitch = ('D sharp');
elseif ((321 <= f) && (f <= 339))
    pitch = ('E');
elseif ((340 <= f) && (f <= 359))
    pitch = ('F');
elseif ((360 <= f) && (f <= 381))
    pitch = ('F sharp');
elseif ((382 <= f) && (f <= 403))
    pitch = ('G');
elseif ((404 <= f) && (f <= 425))
    pitch = ('G sharp');
end
```

APPENDIX B

EMBEDDED MATLAB CODE FOR "FIND FUND_FREQ"

```
function [y,fund_freq] = fcn(u)
%#eml (EMBEDDED FUNCTION #1)

y = u;

value = max(u);
a = find(u==value,1) -1;
fund_freq = a(1,1); %a scalar
```

APPENDIX C

EMBEDDED MATLAB CODE FOR "DETERMINE PITCH"

```
function pitch = fcn(u)
%#em2

pitch = double(0.0); %initialize pitch value

% STANDARD PITCHES (frequencies all in Hz)
% A = 220;
% A1 = 233; % A sharp
% B = 247;
% C = 262; % 'middle C'
% C1 = 277;
% D = 294;
% D1 = 311;
% E = 330;
% F = 349;
% F1 = 370;
% G = 392;
% G1 = 415;

f = round(u); %round to nearest integer

if (f==0) %THIS IS AN ERROR!
    pitch = 9.99999999999999;
elseif (f ~=0)

i=0; %index for iterations of the following loop
while ((f < 212) || (f > 425))
    if f <= 210 %lower octave
        f = 2*f;
        i = i+1;
    elseif (f > 430)
        f = 0.5*f;
        i = i+1;
    elseif (i > 9999) %prevent infinite loop
        break;
    end
end

f = round(f); %make sure it's an integer (again)

if ((212 <= f) && (f <= 226))
    pitch = double('A') + .0001*double('0'); %A
elseif ((227 <= f) && (f <= 240))
    pitch = double('A') + .0001*double('1'); %A sharp ("1" => "sharp")
elseif ((241 <= f) && (f <= 254))
    pitch = double('B') + .0001*double('0');
elseif ((255 <= f) && (f <= 269))
```

```
    pitch = double('C') + .0001*double('0');
elseif ((270 <= f) && (f <= 285))
    pitch = double('C') + .0001*double('1');
elseif ((286 <= f) && (f <= 302))
    pitch = double('D') + .0001*double('0');
elseif ((303 <= f) && (f <= 320))
    pitch = double('D') + .0001*double('1');
elseif ((321 <= f) && (f <= 339))
    pitch = double('E') + .0001*double('0');
elseif ((340 <= f) && (f <= 359))
    pitch = double('F') + .0001*double('0');
elseif ((360 <= f) && (f <= 381))
    pitch = double('F') + .0001*double('1');
elseif ((382 <= f) && (f <= 403))
    pitch = double('G') + .0001*double('0');
elseif ((404 <= f) && (f <= 425))
    pitch = double('G') + .0001*double('1');
end

end
```

CONTACT INFORMATION

Name:	Christopher Matthew Jagielski
Professional Address:	c/o Dr. Deepa Kundur Department of Electrical and Computer Engineering 111D Zachry Engineering Center Texas A&M University College Station, TX 77843-3128 USA
Email Address:	chrisjagielski@gmail.com
Education:	B.S., Electrical Engineering, Texas A&M University, May 2012
Honors:	Cum Laude Honors Undergraduate Research Fellow University Honors Distinction Engineering Scholars Program Distinction President of Eta Kappa Nu engineering honor society Member of Tau Beta Pi engineering honor society