11-2007

A voice operated musical instrument.

Travis R. Gault
University of Louisville

A Voice Operated Musical Instrument

By

Travis R. Gault
B.S. ECE, University of Louisville, 2004 (*magna cum laude*)
B.S. CECS, University of Louisville, 2006 (*magna cum laude*)

A Thesis
Submitted to the Faculty of the
University of Louisville
Speed School of Engineering
in Partial Fulfillment of the Requirements
for the Professional Degree of

MASTER OF ENGINEERING

Department of Electrical and Computer Engineering
University of Louisville
Louisville, Kentucky

November 28, 2007

A Voice Operated Musical Instrument

By

Travis R. Gault
B.S. ECE, University of Louisville, 2004
B.S. CECS, University of Louisville, 2006

A Thesis Approved on

November 28, 2007

By the following Thesis Committee:

____
Dr. Thomas Cleaver, Thesis Director

____
Dr. Barry Horowitz

____
Dr. Ahmed Desoky

DEDICATION

To my friends, both students and faculty, whose support has been tremendous throughout my college career, especially to my parents who have supported me since day zero, and who have been wholly committed to the belief that I may actually finish school someday and “get a J-O-B”.

ACKNOWLEDGEMENTS

I would like to thank all the mentors that have given so much over most of the past decade, namely the faculty and staff at Speed School. I would also like to extend a special gratitude toward my Thesis Director (Dr. Cleaver) and Thesis Committee (Drs. Horowitz and Desoky) and Tom Carroll in the Electronics Shop for taking extra time from their days to assist and guide me on this challenging path. Thank you.

ABSTRACT

Many mathematical formulas and algorithms exist to identify pitches formed by human voices, and this has continued to be popular in the fields of music and signal processing. Other systems and research perform real time pitch identification implemented by using PCs with system clocks faster than 400MHz. This thesis explores developing an embedded RPTI system using the average magnitude difference function (AMDF), which will also use MIDI commands to control a synthesizer to track the pitch in near real time.

The AMDF algorithm was simulated and its performance analyzed in MATLAB with pre-recorded sound files from a PC. Errors inherent to the AMDF and the hardware constraints led to noticeable pitch errors. The MATLAB code was optimized and its performance verified for the Motorola 68000 assembly language. This stage of development led to realization that the original design would have to change for the processing time required for the AMDF implementation. Hardware was constructed to support an 8MHz Motorola 68000, analog input, and MIDI communications. The various modules were constructed using Vectorbord© prototyping board with soldered tracks, wires and sockets. Modules were tested individually and as a whole unit. A design flaw was noticed with the final design, which caused the unit to fail during program execution while operating in a stand-alone mode.

This design is a proof of concept for a product that can be improved upon with newer components, more advanced algorithms and hardware construction, and a more aesthetically pleasing package. Ultimately, hardware limitations imposed by the available equipment in addition to a hidden design flaw contributed to the failure of this stand-alone prototype.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I – INTRODUCTION

For anyone who has watched the first few episodes of each season of "American Idol" on the Fox television channel, it should be obvious that there are a number of Americans with singing difficulties. This results from the would-be singers lacking the ability or training to distinguish between small variations in pitch (frequency), and correct their vocal output accordingly. The vernacular calls this condition tone-deafness. However this term is actually incorrect, because tone-deafness by definition implies that one cannot hear tones, when in fact the difficulty lies in hearing the difference between tones. "Research has shown that some people, termed 'amusic', can neither produce nor perceive music." [Stewart, 2006] This condition called amusia is analogous to color blindness.

The original goal of this research was "to develop a device that will use the human voice to operate an electronic musical instrument" [Cleaver, 2000]. However, it was determined through calculation and experimentation that the original requirements of the design prototype would have to change due to hardware limitations (see Section 3.2.2). However, the original design will be presented in this section. Modifications will be presented and justified in subsequent sections where relevant.

The human voice will be sampled via a microphone and an analog to digital converter, which will be connected to a microprocessor that shall be programmed to operate as a digital signal processor. "The principal frequency[1] of the input will be extracted, corrected for pitch and then used to operate the musical instrument. The implementation for the musical instrument will use the principal frequency data to

[1] The words "principal frequency" will herein be referred to as "fundamental frequency".

provide keying commands to a commercial synthesizer keyboard" [Cleaver, 2000].

To use this product, the user must have access to a synthesizer capable of accepting and interpreting the MIDI protocol. The user must be able to make an audible tone from his/her vocal chords to activate the device. The device will be easy to operate in that there are only three setup requirements including: connecting the MIDI cable to the device and synthesizer, placing the microphone in an area close to the user's mouth, and supplying power to the synthesizer and device. Once these criteria are met, the user need only sing to operate the device.

The device must be able to identify vocal inputs in near real time so as not to be audibly noticeable. The device must be able to extract and output the fundamental frequency of the singing voice. The device shall be able to accommodate male and female singers. The device must be able to acquire the vocal signal without ambient noise, including the output of the synthesizer. The device shall be powered via an external power supply. The device must be able to communicate to a synthesizer via the MIDI protocol through a standard MIDI cable. The device must be able to communicate to a computer running a dummy terminal for development purposes. The device must be as small as reasonably possible. The device developed will be a functional prototype only and not a device suitable for manufacture, because the parts to be used to construct the prototype are no longer manufactured. However, the concepts that will be used to create the prototype can be expanded through further research to implement this device on modern technology.

The user interface for this device will consist of some knobs, switches, and indicators. The user shall use this interface to adjust the amplification settings of the microphone and operate the device. A possible user interface is show below in Figure 1.1, representing a possible view from the top of the device.

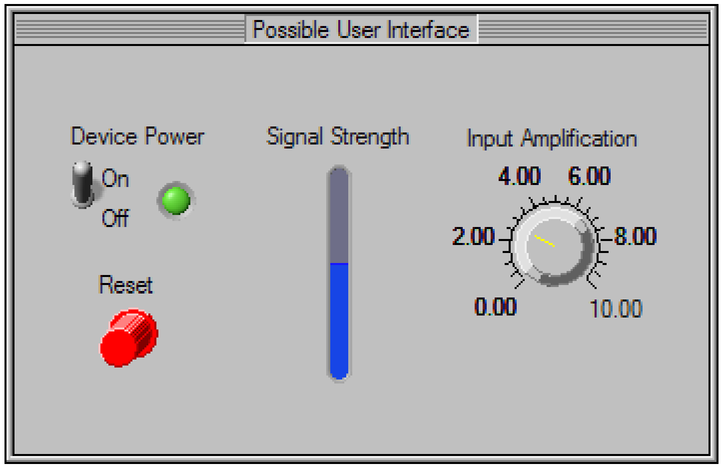


Figure 1.1: Possible device user interface.

The different items of interest are: the device power switch, the reset switch, the signal strength indicator, and the input amplification knob. The device power switch will be used to set the on or off state of the device. The (pushbutton) reset switch will primarily be used in development and emulation, but will be left for the user as alternate means to reinitialize the device. The signal strength indicator will indicate the amplitude of the amplified input signal from the microphone after the amplification and filtering stages. The input amplification control will consist of a knob that the user will adjust in conjunction with reading the signal strength indicator to optimize the input amplification of the microphone.

CHAPTER II – LITERATURE REVIEW

The tasks involved in completing this project involve two main components: identifying the fundamental frequency and using the MIDI protocol to communicate appropriately to the synthesizer. In this chapter, the necessary background information is presented to complete these tasks. In Section 2.1, the elements of pitch and its perception are discussed. This is followed by a discussion of the impact of the human vocal range on this design in Section 2.2. Sections 2.3 and 2.4 discuss the techniques directly involved in identifying the pitch and communicating with the synthesizer, respectively.

2.1 What is pitch?

According to the ANSI standard for acoustical terminology, pitch, in a general sense can be defined as:

> "…that attribute of auditory sensation in terms of which sounds may be ordered on a scale from low to high. Pitch depends mainly on the frequency content of the sound stimulus, but it also depends on the sound pressure and the waveform of the stimulus." [ANSI 1994]

The key words for this thesis are actually in the second sentence. Particularly, the dependence on the frequency content is of interest. However, the awareness that the stimulus and sound pressure can influence the pitch is also of interest, because this generally vague definition supports that the overall task of extracting the physical representation of the pitch (called the fundamental frequency) is subject to errors. The effects of some of these additional influences are discussed in Section 2.3.1.

In music, the dominant fields of study pertaining to this research are psychoacoustics and music psychology. According to Merriam-Webster's Medical Dictionary, the former is "a branch of science dealing with hearing, the sensations produced by sounds, and the problems of communication" while the latter is the study of how humans perceive musical elements and the feelings evoked by musical stimuli [Scheirer 2000].

> "The fundamental frequency of a periodic signal is the inverse of its period, which may be defined as the smallest positive member of the infinite set of time shifts that leave the signal invariant. This definition applies strictly only to a perfectly periodic signal, an uninteresting object because it cannot be switched on or off or modulated in any way without losing its perfect periodicity." [Scheirer 2000]

Usually, the subjective recognition of a pitch associated with a sound depends on the fundamental frequency. However, there are exceptions to this generalization because some sounds may be periodic but have no pitch, while other sounds may not be periodic yet have a pitch [de Cheveigné 2003]. The classic example of the latter is that of a bell, which has a pitch, but no fundamental frequency [Gerhard 2003]. However, the relationship between the existence of pitch and a fundamental frequency is usually one-to-one insofar as the words "pitch" and "fundamental frequency" (f_0) are used interchangeably in the field and in this document. A similar relationship exists between the terms "f_0 estimation methods" and "pitch detection algorithms" [de Cheveigné 2003].

2.2 The human vocal range and the ideal choice of sampling rate

The sampling rate is a crucial element in this design. According to the Shannon Sampling Theorem, the sampling rate must greater than or equal to twice the maximum frequency to be sampled. This minimum sampling rate is referred to as the Nyquist rate [Weeks 2007]. According the library of music at Yale University, the standard human vocal range is from E2 (bass singing at 82.407Hz) to A5 (soprano singing at 880.000Hz)

[Yale 2005] and is illustrated in Figure 2.1. If this design is to accommodate the full spectrum of the human voice, the ideal sampling rate must then be $f_s = 880 \cdot 2 = 1760\,Hz$.

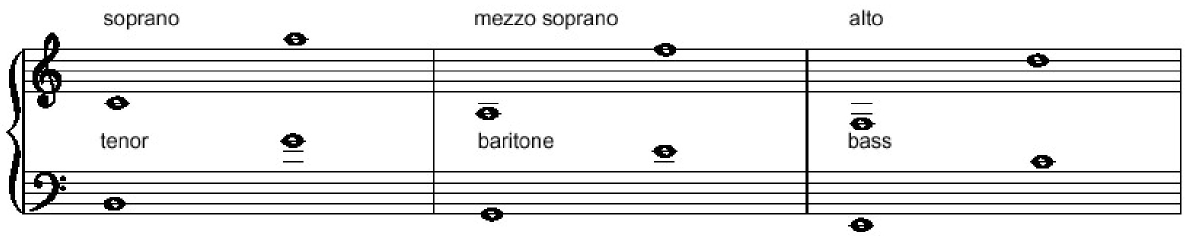


Figure 2.1: Vocal ranges for singers. Taken from [Yale 2005].

If the minimum sampling rate must be 1760Hz, a sampling rate of 2000Hz would be chosen for this design for two reasons. First this is an easy number to implement in hardware via clock dividers, and this number is a convenient figure to use for multiplication. However, this sampling rate will be too high for this embedded software implementation to handle because of the system clock speed. These difficulties are discussed in greater detail in Section 3.1.

2.3 Frequency Identification (Pitch-Tracking) Techniques

There is a broad range of literature regarding extracting pitch information using a variety of different techniques as well as some research on real time vocal analysis and synthesis on a MIDI capable synthesizer (see [Ryynänen 2004, Saul 2002, and Shimamura 2001]). However, these algorithms and designs are designed to run on a personal computer (PC) and use sampling rates of 11.025 kHz or higher. This device is unique in that it runs on an embedded system running at a much slower clock rate than a PC or its equivalent. The processor chosen for this project is the Motorola 68000 because of its availability in the lab. This processor can accept a clock in the neighborhood of 4-16MHz; therefore, it is crucial that the algorithm be speed-efficient and fairly accurate.

There are a large variety of methods available to identify a frequency from a sample dataset. However, the hardware limitations govern the choice between the methods considered. This particular processor model does not support floating-point operations or complex mathematical functions such as: the exponential function, logarithms, sine, and cosine. Therefore, methods and algorithms requiring the use of such functions were dismissed. Many other methods other than those presented were reviewed but rejected for consideration because of these hardware limitations. Some of these methods are listed in Section 2.3.6, while other methods that warranted serious consideration are discussed in this section. The methods presented in this literature review are the primary methods investigated for this design following a brief discussion of some of the difficulties inherent to pitch tracking.

2.3.1 Problems Associated with Pitch Detection of Vocal Signals

In general, finding the frequency or period of a perfectly periodic waveform is relatively simple. However, measuring the pitch, or fundamental frequency, from a voiced signal is considered a difficult task mainly because when the glottis[2] produces a waveform, the waveform is not a perfect composition of periodic pulses. It is also difficult to measure the interaction between the vocal tract and the glottal excitation

[2] The glottis is the space between one of the true vocal cords and the arytenoid cartilage on one side of the larynx and those of the other side [Merriam-Webster's Medical Dictionary].

because the vocal tract formants[3] can sometimes alter the glottal waveform structure. These interactions cause the most difficulty when the articulators change rapidly and also when the vocal formants themselves change rapidly [Rabiner 1976].

It is also difficult to determine the beginning and end of each pitch period in a voiced segment. This generally leads to the arbitrary choice of the pitch period beginning and ending times. An example of this arbitrary choice is shown in Figure 2.2. In this

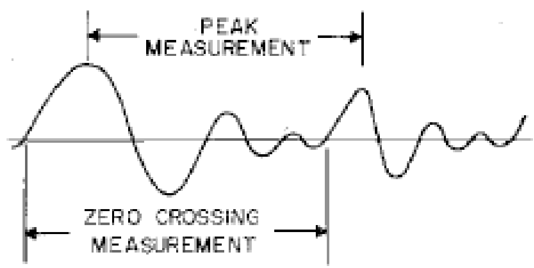


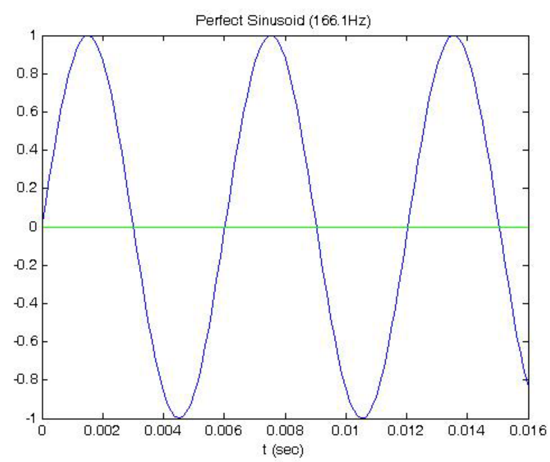
Figure 2.2: Two waveform measurements which can be used to define pitch markers. Taken from [Rabiner 1976].

figure, the two candidates for defining the period beginning/end are the maximum value and the zero-crossings prior to the maximum during each period. The only requirement with these measurements is that the locations be consistent from period-to-period, else spurious pitch estimates may result. In Figure 2.2, the period associated with the peak measurement will result in a higher frequency than the zero-crossing measurement. Discrepancies of this nature occur often because the speech waveform is quasi-periodic and because peak measurements are sensitive to formants, noise, and any DC level in the waveform. Another related difficulty arises from sorting between unvoiced speech and weakly voiced speech. This is problematic because the transitions between these two signal types are difficult to identify [Rabiner 1976].

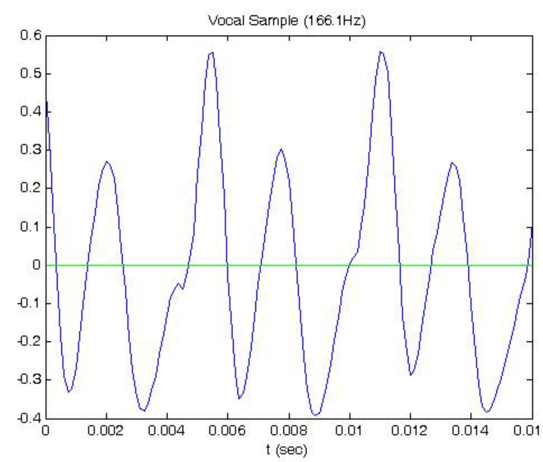
[3] A formant is any of several frequency regions of relatively great intensity in a sound spectrum, which together determine the characteristic quality of a vowel sound [The American Heritage® Dictionary].

2.3.2 Counting Zero-Crossings

One of the simplest methods to measure frequency in the time domain is to measure the time between zero-crossings of the periodic signal. The reciprocal of the period corresponds to the frequency of interest. This is by far the easiest method to implement in hardware or software. However, this method is very susceptible to noise on the channel and any DC offset generated by the amplification or filtering stages in hardware, as well as the quantization error in the analog-to-digital (A/D) converter. This method is ideal for simple sinusoids, but a poor choice for complex waveforms with harmonics or distortions, such as vocal signals. To illustrate this point, look at Figure 2.3 below. On the left is a perfect sinusoid with frequency equal to 166.1Hz. On the right is a vocal sample at approximately the same frequency.



(a)



(b)

Figure 2.3: Perfect sinusoid (a) and vocal sample (b) at the same frequency.

If counting the number of zero crossings were used for the second figure, the frequency would be approximately twice the original frequency, because of the additional harmonics in the human voice. This method has been discarded from consideration;

however, research is still performed today on finding better ways to use the zero-crossing method for pitch identification [Gerhard 2003].

2.3.3 Frequency Domain Analysis using the Discrete Fourier Transform (DFT)

Perhaps the more intuitive approach to identifying a fundamental frequency is to use the frequency domain for the analysis. The DFT "plays an important role in the analysis, design and implementation of discrete-time signal-processing algorithms and systems" [Oppenheim 1999]. The advantage of using the DFT is that the DFT spectrum is identical to samples of the continuous case of the Fourier transform with N spectra samples occurring at uniformly spaced frequencies, where the input signal is truly band-limited. Many digital applications use the more efficient version of the DFT called the Fast Fourier Transform (FFT).

Frequency domain analysis is used frequently in conjunction with tools such as a spectrum analyzer or an oscilloscope with an FFT implementation, which typically have some form of a fast digital signal processing IC to perform the calculations. Other systems similar to this research may use variations of the FFT, such at decimation in time, decimation in frequency, or other special implementations such as Cooley-Tukey's, the Prime-factor, Bruun's, Rader's, or Bluestein's FFT algorithms to find spectral results [FFT, 2007].

In theory, the implementation of this algorithm would find the maxima of the power spectra to identify the frequency once the data are transformed into the frequency domain. Figure 2.4 represents the vocal data from Figure 2.3b in the frequency domain. In this figure, it is clearly obvious that the three harmonics have sufficient power to be identified as the fundamental frequency. In fact, the middle spike has the highest power, but it does not correlate with the pitch produced by the test subject.

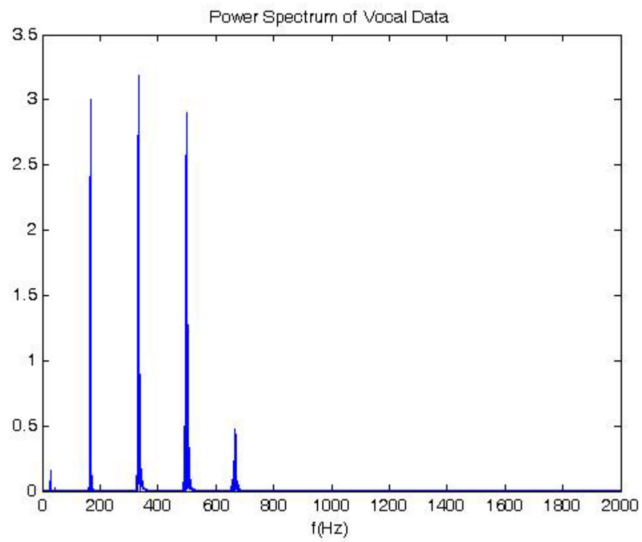


Figure 2.4: Power spectrum of the vocal sample from Figure 2.3b.

The problem with this method is that the number of sample points, N, would have to be a 512 or 1024 point DFT/FFT to get a 1-2Hz resolution on the frequency axis, which would take too long to compute in real time. The Motorola website had an FFT code example that was written by Ron Williams from Ohio University, but based on code appearing in *Byte Magazine* in 1979. As of November 27, 2007, the code is currently available at <http://www.embeddedrelated.com/groups/m68hc11/show/2125.php>. This code is designed to run on the Motorola 68HC11 (2MHz system clock) and computes a 256 point 8-bit integer FFT. The reported execution time of this code is 350ms. Performance estimation for the 68000 based on this code is possible with the proper considerations. First, the execution time can be divided by four since the 68000's system clock is 8MHz. However, the time must be rescaled to accommodate for the larger FFT. The computational complexity of the FFT is O(*N log N*). So if $87.5 = N \log N$, then $N =$

26.653 and $2N = 53.306$. Thus $53.306 \cdot \log(53.306) = 211.949\,ms$, which is too long to execute a 512 point FFT for real time applications.

2.3.4 Auto-Correlation Function

Shimamura and Kobayashi have done research in the area of extracting pitches from noisy speech signals by using the auto-correlation function (ACF) and the average magnitude-difference function (AMDF) [Shimamura 2001]. The ACF is based in the time domain and is defined by Equation 2.1:

$$\phi(\tau) = \frac{1}{N}\sum_{n=0}^{N-1} x(n)x(n+\tau) \tag{2.1}$$

where:

x(n) = vocal signal sample;
τ = the lag number, or time shift;
n = the time for a discrete signal

Sliding a small window of the sampled signal with the whole of the sampled data, in essence, forms the ACF results. As periodic segments overlap the similar segments of the sampled data, $\phi(\tau)$ assumes a large value at integer multiples of the signal's fundamental period (T_0). The fundamental period is calculated from the differences in τ corresponding to the peaks of $\phi(\tau)$; dividing the sampling rate by the period yields the fundamental frequency (f_0).

The advantages of the ACF are obvious in that the ACF does not require the use of exponentials, logarithms or sinusoidal functions to perform the calculation. The math is straightforward and simple. Shimamura and Kobayashi prove for a large N, that if a noisy speech signal x(n) is composed of a clean speech signal, s(n) and additive white

Gaussian noise, w(n), s(n) will not correlate with w(n). Therefore, an added advantage of the ACF is that it performs well in noisy environments.

The problem with the ACF is that sometimes the peak located at the second multiple of T_0 is much larger than the first peak. This can lead to a half-pitch error in frequency identification. In instances where the first peak occurs at a time of $\tau < T_0$, a double-pitch error in frequency identification may occur [Shimamura 2001].

2.3.5 Average Magnitude-Difference Function (AMDF)

The AMDF is similar to the ACF in that the results are obtained by sliding a small window of the sampled signal with the whole of the sampled data. Equation 2.2 represents the AMDF:

$$\psi(\tau) = \frac{1}{N}\sum_{n=0}^{N-1}\left|x(n) - x(n+\tau)\right| \tag{2.2}$$

where x(n), τ and n correspond to their counterparts in the definition of the ACF in Equation 2.1.

The primary difference between the two formulas is that as periodic segments overlap the similar segments of the sampled data, $\psi(\tau)$ assumes small values at integer multiples of the signal's fundamental period (T_0). The fundamental period is calculated from the differences in τ corresponding to the notches of $\psi(\tau)$; dividing the sampling rate by the period once again yields the fundamental frequency (f_0).

Shimamura and Kobayashi also prove for a large N, that if a noisy speech signal x(n) is composed of a clean speech signal and additive white Gaussian, the signal and noise are independent. Thus, the AMDF also performs well in noisy environments. The AMDF shares the advantages of the ACF as described above [Shimamura 2001].

However, the AMDF is advantageous over the ACF for this design because the main operation is subtraction and the absolute value function, whereas the main operation in the ACF is multiplication. On the 68000, and most processors, multiplication requires more clock cycles to execute than subtraction. This processor's execution time varies for mathematical functions depending on values of the input arguments. The worst-case scenario for implementing the subtraction and absolute value functions (20 clock cycles) is less than the best-case scenario (multiplying zero by zero) for unsigned multiplication (38 clock cycles) [Motorola 1993]. This means that the ACF, at its best will require 90% more time to execute. Additionally, the AMDF has a sharper pitch resolution when compared to the results of the ACF [Kim 1998].

$$AMDF(n) = \sum_{k=0}^{N-1} \left| s(k) - s(k+n) \right| \tag{2.3}$$

Therefore, the ADMF will be implemented for pitch detection in this design, with the exception that it will not be normalized with respect to the total number of samples (N) and is defined in Equation 2.3. The reason the normalization step is excluded is that dividing by N wastes clock cycles, but more importantly, the 68000 does not handle floating-point numbers as easily as 16-bit integers. Since scaling the data essentially has no added effect for this application, its removal is justified. Although this change would more effectively lend the algorithm to be called the Sum of Magnitudes Difference Function, it will still be referred to as the Average Magnitude Difference Function to be consistent with the literature available.

2.3.6 Some Other Techniques Worth Mentioning

In 1976, Lawrence Rabiner et al. authored a paper for the IEEE that comparatively studied several pitch detection algorithms. Their studies included both performance in terms of accuracy of the detection as well as computation time. Each algorithm was implemented on the same data sets consisting of a low-pitched male (LM), two male speakers (M1 and M2), two female speakers (F1 and F2), a child (C1) and a diplophonic[4] speaker (D1). All filtering, signal conditioning and signal processing was handled digitally on a Nova 800 minicomputer. The algorithms they studied were the ACF using clipping (AUTOC), the cepstrom method (CEP), the simplified inverse filtering technique (SIFT), the data reduction method (DARD), the parallel processing method (PPROC), the spectral equalization LPC method using Newton's transformation (LPC) and the previously discussed AMDF.

Table 2.1 shows the computational performance results for the different algorithms studied by Rabiner et al. For each algorithm, the speed was computed from processing a one second sample set. In these results, the AMDF had the third best performance. However, Tables 2.2 and 2.3 show that the AMDF significantly outperforms the faster two algorithms in terms of accuracy. The remaining algorithms, although more accurate, require significantly more time to compute the results, these algorithms were not considered for implementation.

[4] diplophonia is a condition in which the voice simultaneously produces two sounds of a different pitch [Dictionary.com]

Pitch Detector	Speed/s of Speech	Arithmetic Type	Down-sampling Used	Dependence on Sampling Rate
DARD	5 s	Integer	No	Linear
PPROC	7.5 s	Integer	No	Linear
AMDF	50 s	Floating point	No[a]	Quadratic
AUTOC	120 s	Integer	No[a]	Quadratic
SIFT	250 s	Floating point	Yes	Quadratic
LPC	300 s	Floating point	Yes	Quadratic
CEP	400 s	Mixed	No	Linear

[a]These algorithms could easily incorporate downsampling.

Table 2.1: Computational considerations for the seven pitch detectors on the Nova 800 minicomputer. Taken from [Rabiner 1976].

Speaker		AUTOC	CEP	SIFT	DARD	PPROC	LPC	AMDF
LM	M	15.3	0.5	0.6	5.8	10.0	4.4	12.8
	T	26.1	1.1	4.5	5.8	11.0	5.6	15.6
	W	19.5	1.3	4.5	13.8	23.8	13.0	23.8
	Sum	60.9	2.9	9.6	25.4	44.8	23.0	52.2
M1	M	0.6	0.1	0.0	5.9	2.0	0.1	0.3
	T	3.4	0.1	0.8	6.3	3.0	0.8	0.8
	W	2.8	0.5	3.0	23.5	6.0	0.8	2.8
	Sum	6.8	0.7	3.8	35.7	11.0	1.7	3.9
M2	M	6.1	0.4	1.3	15.9	4.9	2.9	7.3
	T	9.9	0.6	3.4	4.0	5.8	4.0	9.8
	W	7.3	1.3	5.3	26.8	12.3	5.5	8.5
	Sum	23.3	2.3	10.0	46.7	23.0	12.4	25.6
F1	M	1.9	9.1	4.4	7.3	4.0	2.4	0.5
	T	1.6	8.5	1.8	6.3	2.8	1.4	0.0
	W	0.0	29.0	8.0	0.8	4.0	2.0	0.0
	Sum	3.5	46.0	14.2	14.4	10.8	5.8	0.5
F2	M	0.4	1.4	2.1	7.1	2.4	1.6	0.6
	T	0.6	2.0	1.5	5.6	1.5	1.5	1.0
	W	2.0	2.5	3.8	8.5	5.0	2.0	1.8
	Sum	3.0	5.9	7.4	21.2	8.9	5.1	3.4
C1	M	1.0	13.6	65.3	6.1	7.8	8.3	10.6
	T	1.9	14.8	62.6	12.9	9.0	12.3	9.1
	W	0.0	12.5	40.8	3.0	7.3	5.5	6.5
	Sum	2.9	40.9	168.7	22.0	24.1	26.1	26.2

Table 2.2: Number of gross pitch errors – unsmoothed. Taken from [Rabiner 1976].

Speaker	AUTOC	CEP	SIFT	DARD	PPROC	LPC	AMDF	Sum
LM	4	1	2	3	4	3	4	21
M1	2	1	1	3	2	1	1	11
M2	3	1	2	4	3	2	3	18
F1	1	4	2	2	2	1	1	13
F2	1	1	2	3	2	1	1	11
C1	1	3	5	3	3	3	3	21
Sum	12	11	14	18	16	11	13	

Code: (0–6) = 1, (6–12) = 2, (12–42) = 3, (42–90) = 4, (90–) = 5.

Table 2.3: Performance scores based on sum of gross pitch errors – unsmoothed. Taken from [Rabiner 1976].

2.3.7 Anticipated Problems with the AMDF

Based on the literature review, the AMDF is susceptible to three main problems with this application. The first is the sampling rate. Although the results by Rabiner et al. show the AMDF is fairly accurate, they also used pre-recorded data and a sampling frequency of 11.025 kHz to perform their study. Even though the AMDF can be down-sampled successfully [Rabiner 1976], the sampling frequency implemented in this design is extremely small compared to that used in other research in the field. Thus it is anticipated that errors will occur as a result of this reduced sampling rate.

The second concern is that the AMDF is known to produce octave, or pitch-doubling errors when a fixed threshold for period detection is used. This is also a problem with other algorithms and can easily been seen again by recalling Figure 2.4. In [Kim 1998], they demonstrate that an adaptive threshold can be used to determine when periods occur. By using a weighting factor and comparing other suitable pitches by doubling, tripling, quadrupling, etc… the proposed fundamental frequency, they see if each multiple of the original pitch falls under the proposed threshold. If so, a new pitch is identified [Kim 1998]. Although ideal, implementing an adaptive threshold will not be possible in this design because it requires too much time to execute.

Regarding octave errors, the real time processing can be advantageous because this concern can be mitigated by the fact that this design is to be used by amateur singers who are less likely to jump more than an octave unintentionally. Tracking the numerical difference between MIDI note numbers will easily reveal if a singer tries to jump more than an octave at a time, which will help in preventing octave errors. However, if the initial pitch detection is too high, the user will not be able to jump to a lower octave.

The last known problem is identifying a pitch when the user changes between notes. It is difficult to know when one note ends, and if the sampling routines acquire a sample set during a transition, the reported pitch will be inaccurate.

2.4 The MIDI Protocol

Composers and musicians have used the Musical Instrument Digital Interface (MIDI) protocol since its development in 1983. The protocol was originally designed so musicians could connect synthesizers together. Today, the protocol is used to supplement audio in gaming and multimedia applications due to the extremely small file size required to create a MIDI file versus a sampled audio file [MMA 2001]. The size of the files is roughly analogous to comparing a vector drawn image versus a bitmap image of the same object.

The MIDI protocol makes it available not only for communication between synthesizers, but also between other sound modules, wind controllers, guitars and the modern personal computer [MMA 2007]. This design will make use of the small packet size to control a synthesizer, and will use unidirectional communication between the prototype and the synthesizer.

2.4.1 Message Format

The MIDI message format is quite simple. The beginning of a new message contains a status nibble, followed by a channel number nibble (where $0\text{-}15_{dec}$ corresponds to channels $1\text{-}16_{dec}$). The remaining size of the transmitted data depends on the command used. The MIDI protocol specifies that any other transmitted data must only utilize the lower seven bits of a byte. Placing this restriction ensures that the MIDI device can always detect a new command, '`S`' (see below), which always utilizes the most significant bit of the upper nibble. Only three of the commands are used in this design. They are *Note On*, *Note Off*, and *Program Change*.

The packet format for turning a note on or off is as follows:

```
sc nn vv

 s = Status Nibble (Command)
     8 = Note Off
     9 = Note On
 c = Channel Number (0-15dec)
nn = MIDI Note Number (0-127dec)
vv = Note Velocity
```

Example MIDI Packet (hex numbers):

`90 45 7F` - Play pitch A4 (440Hz) on channel 1 with velocity 127

`80 3C 00` - Stop pitch C4 on channel 1, velocity number irrelevant

The packet format for changing the MIDI instrument is as follows:

```
sc pp

s = Status Nibble (Command)
     12dec = program change
 c = Channel Number (0-15dec)
pp = New instrument number (0-127dec)
```

Example MIDI Packet (hex numbers):

`C0 00` - Change current instrument on channel 1 to Acoustic Grand Piano

There are seven total MIDI commands that are represented in hexadecimal per the table below. The last hexadecimal value, F, is reserved for future development and will be ignored by the MIDI device if transmitted [MMA 1995],[MMA 1995]. Table 2.4 contains the relevant commands for this design.

Hex	Command
8	Note off
9	Note on
A	After touch (key pressure)
B	Control change
C	Program (instrument) change
D	Channel pressure
E	Pitch wheel

Table 2.4: MIDI commands

2.4.2 Electrical Specifications

The MIDI data transmission protocol uses the RS-232 standard. The bit-rate is 31250 bps, and the voltages range from 0-5V representing a logic 0-1, respectively [MMA 2001]. The MIDI cable itself requires a shielded cable and cannot exceed a length of 50ft (15m) [MMA 1985] and consists of two different connectors (MIDI In and MIDI Out), and sometimes a third (MIDI Thru) [MMA 2001]. Cables designed to interface with a PC sound card using a DB15 connection, and contain an adapter with components similar to the Figure 2.5.

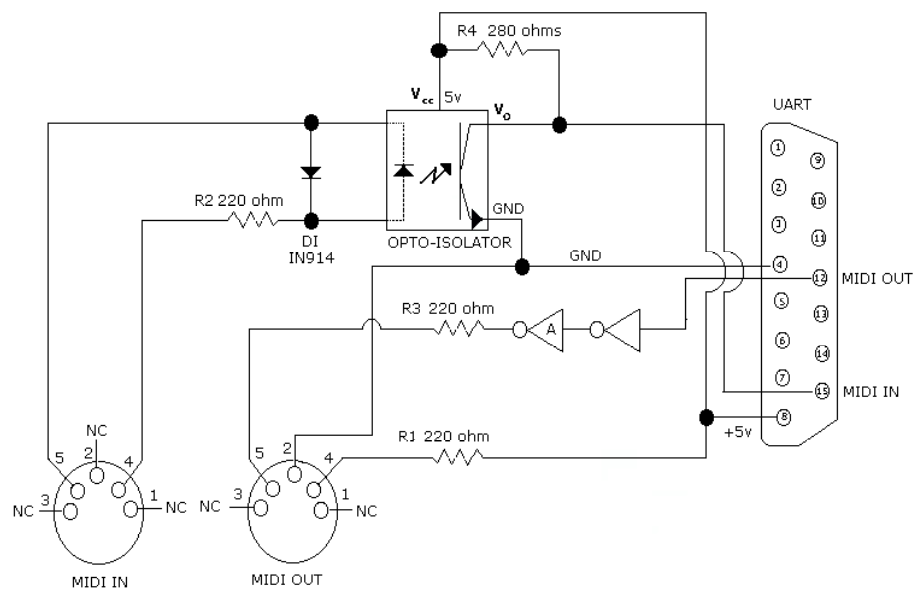


Figure 2.5: Computer Sound Card Game Port to Standard MIDI Connector [MMA 1985].

This design makes use of the DB15 connector cable above, as it requires the least amount of external components to implement.

2.4.3 Timing Considerations

MIDI was designed to convey musical performance data and therefore preserves rhythmic integrity of the music by using accurate timing. A standard RS232 packet consists of ten bits to transmit one byte of data (consisting of one start bit, eight data bits, one stop bit, and no parity) for this application. Sending a simple three-byte message for a Note On or Note Off command will only take 0.96 ms to transmit. Even with larger packets, the delay between sending the packet and hearing the sound (latency) is usually 3 ms or less, depending on the size of a packet. Research has shown that 20-30 ms latency is usually imperceptible, so long as the variation in latency (the jitter) is small [Lago 2004]. Usually, the jitter associated with the MIDI protocol is less than 1 ms.

However, it is important in this design to consider the time it takes to transmit the MIDI data. In this application, only one musical part is produced at once with small data packets, therefore the MIDI protocol timing should be quick enough and should be rhythmically accurate [MMA 2001].

CHAPTER III – SYSTEM DESIGN

3.1 Description of the Hardware

The hardware for this design can be broken down into four basic modules consisting of the hardware responsible for sampling, serial communications, MIDI communications, and memory management. Each of these large modules consists of smaller, supporting modules, and each of the larger modules is connected with the 68000. Each of the major modules is briefly described here and the detailed description will follow.

The first module is the sampling module. It consists of the microphone, amplification stage, filtering stage, the A/D converter, and the clock providing the conversion rate for the A/D converter. Another clocking mechanism is directly interfaced with the 68000 to trigger the sampling routine.

The second module is the serial communications interface (SCI) module. This module was originally designed for a developer. As implementation problems developed, the concept of adding a developer software interface was abandoned, but the fully functional hardware was left on the final prototype; this is the only unused interface in this design. This module consists of an asynchronous communications interface adapter (ACIA), a bit-rate generator, and a voltage level shifter.

The third module is the MIDI communications module, which is used to interface to the synthesizer directly. This module also consists of an ACIA and uses a simple clock divider (counter) connected to the system clock to act as a bit-rate generator. The serial output of the MIDI ACIA is connected to the MIDI OUT input to the synthesizer.

The final module is the memory management unit (MMU). It is responsible for enabling the various devices for read and/or write access as well as indicating to the 68000 when a data transmission is acknowledged (DTACK’) and when valid peripherals request use of the address bus (VPA’).

3.1.1 M68000 Connections

The Motorola M68000 is the microprocessor used in this design. There are two practical choices for a processor for this design based on hardware and test equipment that are readily available at the engineering school. These two choices are the Motorola M68HC11 microcontroller and the Motorola M68000. The latter was chosen over the Motorola M68HC11 because of speed, memory capacity, and that the M68000 easily facilitates connecting a large number of external devices.

The address and data line connections are described in detail in the memory management unit and memory sections in sections 3.1.2 and 3.1.3, respectively, as well as other sections where appropriate. However, address lines A17-A23 are left disconnected, as they are not used.

The address strobe (AS)’ is an output that indicates the address bus is ready for use. It is connected to the memory management unit (MMU) and interrupt acknowledge (IACK’) circuitry to enable communications with memory and to identify interrupt requests, respectively. The data transmission acknowledge (DTACK’) input is connected to DTACK’ from the MMU (Section 3.1.4), and the valid peripheral address (VPA’) input is connected to the VPA’ circuitry (Section 3.1.5) to indicate when memory and peripherals are ready to transmit/receive data. The upper and lower data strobes (UDS’/LDS’) and the read/write (R/W’) line are connected to the MMU to indicate whether an even or odd address is being accessed and whether a read or write cycle is engaged. The EN clock, which divides the system clock by ten, is connected as ECLK to the two M6800 series peripheral devices (M6850s) and the A/D converter clock. Finally, the valid memory address (VMA') output is negated and is used to signal to the two peripheral addresses that the 68000 has recognized a valid peripheral address (VPA') created by selecting one of those M6800 series peripherals (Sections 3.1.7 and 3.1.8).

The system clock (CLK) input comes directly from an 8MHz TTL oscillator. The function code pins (FC0-FC2) are connected to the interrupt acknowledge (IACK') circuitry to indicate an interrupt cycle on the 68000 (Section 3.1.11). These three pins are all asserted when an interrupt request is acknowledged. IPL0' is connected to the sampling frequency (FS') clock to trigger a level one interrupt request (Section 3.1.10). The HALT' and RST' lines are connected to the main reset signal (RESET') generated by the power-on reset circuitry (Section 1.1.12). The IPL1', IPL2', BERR', BGACK' and BR' lines are unused and are tied high to V_{CC} via a 4.7kΩ resistor. The bus grant (BG') output is left disconnected.

3.1.2 The Memory Management Unit (MMU)

The MMU, shown in Figure 3.1, is responsible for selecting between two NVRAMs, one latch, one A/D converter, and two ACIAs for read and/or write operations. The MMU consists of two 74LS138 decoders and a Schmitt triggered inverter (74LS14) to perform device selection and some AND gates to create the DTACK' signal (Section 3.1.4).

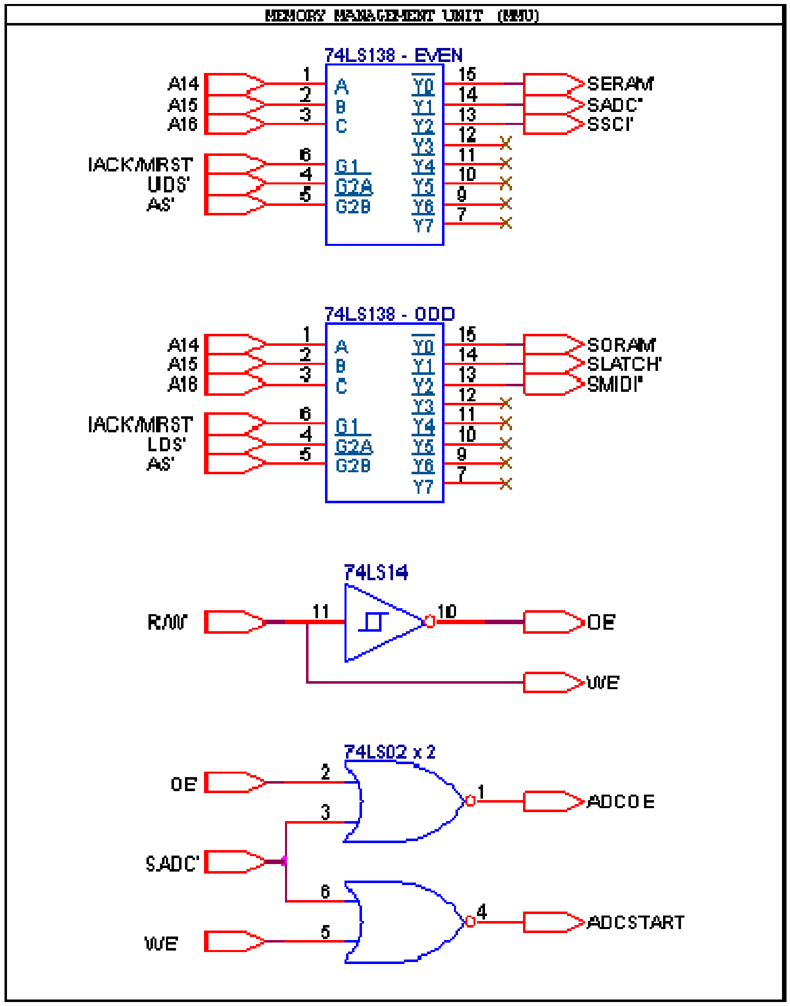


Figure 3.1: Memory management unit.

Address lines A14-A16 are connected from the 68000 to the select inputs A-C, respectively, on each decoder. When the decoders are enabled, these lines are responsible for selecting between the six devices previously mentioned. The active-high enable (G1), of each decoder is connected to the active-low IACK'/MRST' signal. This signal serves to disable the MMU during reset and to prevent erroneous data from being written to the NVRAMs during power-on. This signal also disables the MMU during the 68000's interrupt acknowledge sequence and is discussed in Section 3.1.11. Address strobe (AS') from the 68000 is connected to G2B' of each decoder. Finally, the upper data strobe (UDS') and lower data strobe (LDS') signals are connected to G2A' of the even and odd decoders, respectively. Thus, when AS' and UDS'/LDS' are asserted and IACK'/MRST' is negated, the MMU selects between devices for I/O operations.

On the even decoder (top of Figure 3.1), the active-low outputs Y0', Y1' and Y2' become the control signals SERAM', SLATCH', and SSCI', respectively. These control signals respectively select between the even RAMs, the latch used for operating the LEDs and the serial communications interface. On the odd decoder (below the even decoder in Figure 3.1), the active-low outputs Y0', Y1' and Y2' become the control signals SORAM', SADC', and SMIDI', respectively. These control signals respectively select between the odd RAMs, the A/D converter, and the MIDI ACIA. All other outputs (Y3'-Y7') on both decoders are left disconnected.

The R/W' signal from the 68000 is inverted to create the active-low output enable signal (OE') for use on the NVRAMs. The Schmitt inverter package is used for the power-on reset, and is used here to save board space since the functionality is essentially the same for standard logic levels and operations. The active-low write enable signal (WE') is directly connected to the R/W' signal on the 68000.

All the signals output from the decoders are active-low. However, the A/D converter requires active-high control signals and some special handling since there is no chip enable input on the A/D converter. Therefore, OE' and SADC' from the MMU are NORed and form the A/D output enable (ADCOE) signal. Similarly, WE' and SADC' from the MMU are NORed and for the A/D start conversion (ADCSTART) signal. With this configuration, the A/D converter will begin a conversion when any byte of its address space is written. The results can be accessed when any bytes of its address space is read.

3.1.3 Memory

Two Dallas Semiconductor DS1225AD-85 NVRAMs are used to act as both the program space and temporary memory space in this design. This is advantageous over using a pair of ROMs and a pair of static RAMs in that the NVRAMs can be programmed using the 68000 emulator and that the board space required for the larger NVRAMs is smaller. These NVRAMs also have an 85ns read and write time, which is optimal for the 8MHz system clock (125ns period) driving the 68000, because no additional timing circuitry is required to delay the 68000 read/write cycles.

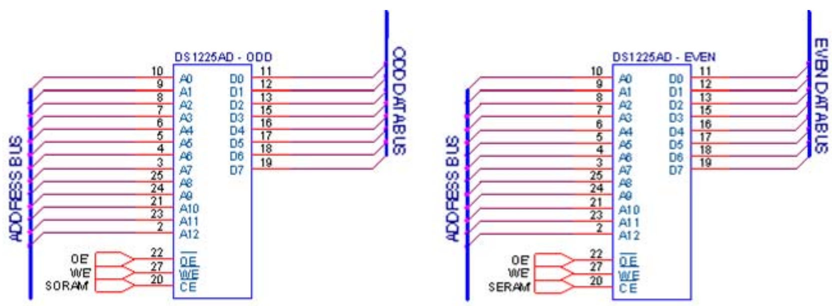


Figure 3.2: (a) Even NVRAM and (b) Odd NVRAM and connections.

Address lines A1-A13 on the 68000 (now referred to as AB1-AB13) are connected to A0-A12 on each of the NVRAMs. Data lines D0-D15 on the 68000 (now referred to as DB0-DB15) are connected to D0-D7 on the memories depending on the address space for each memory. The memory occupying odd addresses is connected to the lower byte of the data bus (DB0-DB7) while the memory occupying even addresses is connected to the upper data bus (DB8-DB15). The data lines are connected in this fashion because the memory is interleaved to make use of the data strobes (UDS' and

LDS') on the 68000, thereby allowing 16-bit executing from 8-bit devices. The control signals OE' and WE' are respectively connected to the corresponding NVRAM pins 22 and 27. SERAM' and SORAM' from the MMU are respectively connected to the even and odd NVRAM chip enables (CE') at pin 20.

3.1.4 Data Transmission Acknowledge (DTACK') Signal

An external device asserts DTACK' to signal the 68000 that data has been placed on the data bus by the device during a read cycle or that data has been read by the device from the data bus during a write cycle. Generally, the 68000 takes four clock cycles to perform a byte- or word-length read or write operation. In the event that the memory or another external device requires more time, DTACK' is used to delay 68000 instruction execution until the devices' timing requirements are met. Hence, the logic used to implement DTACK' is often referred to as a timer. DTACK' is also used to identify a fully-vectored interrupt request during an interrupt acknowledgment; however, fully-vectored interrupts are not implemented in this design.

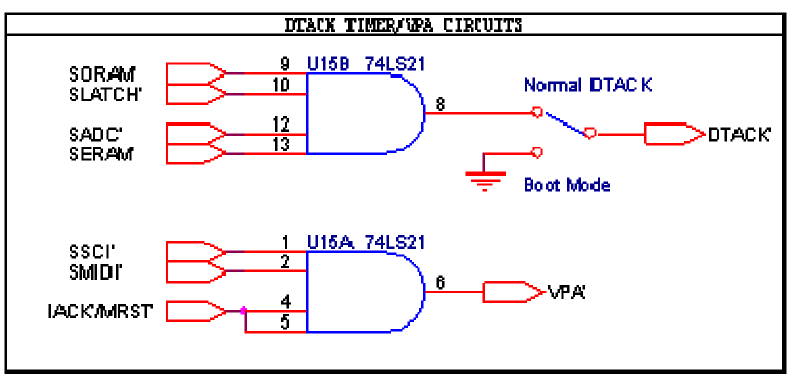


Figure 3.3: Data transfer acknowledge/valid peripheral address circuitry.

In this design, DTACK' is created by ANDing the active low chip select lines connected to the NVRAMs (SERAM' and SORAM'), A/D converter (SADC') and register (SLATCH'). Each chip select on each device is active-low, therefore, ANDing all the chip select inputs will cause a logic low to be produced whenever any of the previous four devices is selected. As previously mentioned, some designs require that a timer be implemented to operate DTACK', especially when interfacing to a slow memory, such as an EEPROM without a DTACK' output. Such a timer is not required here because the NVRAMs are rated for an 85ns read/write cycle which is faster than the period of one 8MHz clock cycle (125ns).

3.1.5 Valid Peripheral Address (VPA') Signal

The VPA' signal is used by the 68000 to interface to the older 6800 series peripherals, such as the Motorola 6850 universal asynchronous receiver/transmitter (UART). VPA' is also used to signal the 68000 that an auto-vectored interrupt is occurring. Both VPA' functions are utilized in this design. However, there are two devices that can generate the VPA' signal for normal operation and a third device indicating the auto-vectored interrupt.

The two devices using the normal functionality of VPA' are the two Motorola M6850s (ACIAs). One of these is used for communication to the synthesizer and the other for terminal communications. The third device is the 8-bit binary down counter, which indirectly asserts VPA' when an interrupt is acknowledged (Section 3.1.11) through the IACK'/MRST' signal. The chip-selects for the two UARTS (SSCI' and SMIDI') and IACK'/MRST' are ANDed to create VPA'.

3.1.6 Bar Graph LED Package and Output Data Register (74LS273)

The 74LS273 8-bit register with clear is used to illuminate the bar graph LEDs. The register is placed on the lower data bus (DB0-DB7) and occupies all odd addresses in the range of $4000-7FFF, inclusive. Data lines DB0-DB7 are connected to register data inputs D1-D8, respectively. This particular register does not have an output or chip enable, so a rudimentary chip enable is constructed by NORing the SLATCH' and (8MHz) CLK signals. When SLATCH' is asserted, the inverted CLK signal is allowed to propagate through the NOR gate and serve as a clock to the register on pin 11. When the clock pulses, the data on pins D1-D8 is latched and output on Q1-Q8. The RESET' signal is connected to CLR' on pin 1.

The output pins, Q1-Q8 are connected in series via a current limiting resistor to the cathode of the bar graph LEDs, which are configured in a common anode configuration. The anodes of the LEDs are connected to V_{CC} (5V). This configuration is used because the register can sink more current on an output low than it can source on an output high. An LED will be illuminated when one of the output pins are driven low.

The bar-graph LED package actually contains ten LEDs. Since only eight LEDs are used with the register, the remaining two LEDs are used to indicate the device is powered.

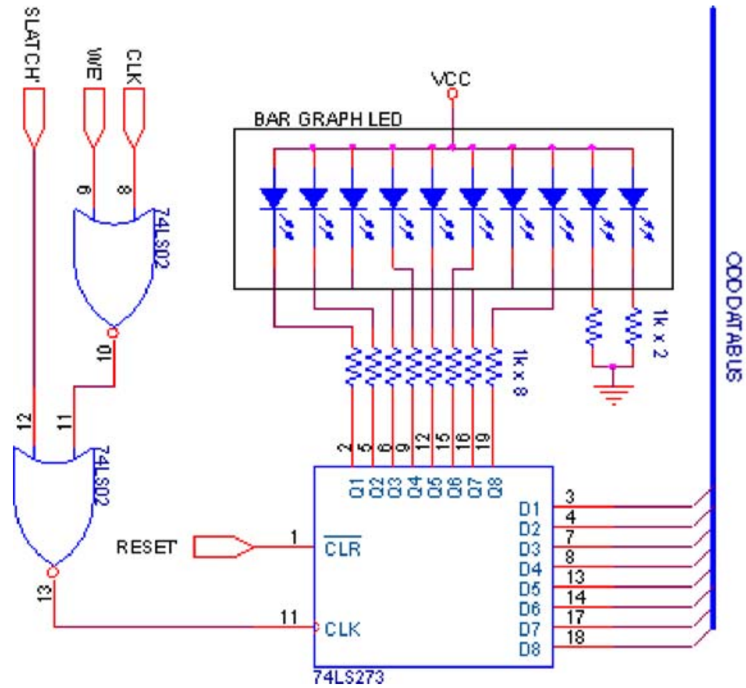


Figure 3.4: LEDs and registers.

3.1.7 Serial Communications Interface (SCI) for Terminal Communications

The device responsible for serial communications is the Motorola 68B50 Asynchronous Communications Interface Adapter (ACIA). This ACIA is connected to the MMU and is configured to operate when even memory locations in the range of $8000-BFFF are accessed for read and write operations. The ACIAs have two register pairs and AB1 on the 68000 is used as the ACIA register select (RS) signal, selecting between the transmit/receive data and status/control registers. Since this ACIA operates on even addresses, the data lines on this ACIA are connected to the upper byte in the data bus on the 68000. Other signals, such as the SSCI' signal from the MMU and the negated VMA' signal (VMA) from the 68000 are respectively connected to CS2' and CS1 on this ACIA; VMA is also connected to CS0. The read/write (R/W') line on the 68000 is connected directly to the read/write line on the ACIA. This allows the 68000 to select between the status/receive and control/transmit registers. The E-clock (EN) from the 68000 is connected to E on the ACIA to synchronize operations with the 68000. The clear to send (CTS') and data carrier detect (DCD') inputs are unused and tied to ground. The interrupt request (IRQ') and request to send (RTS') outputs are unused are disconnected.

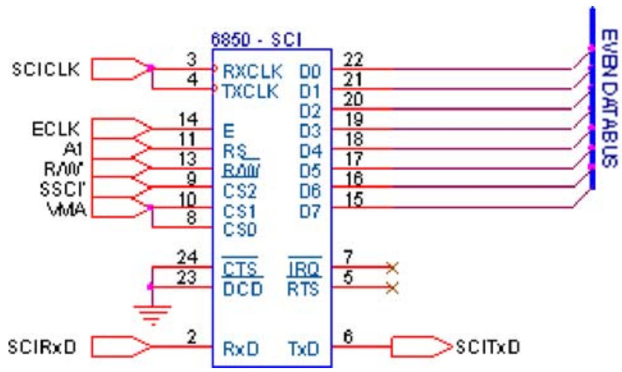


Figure 3.5: SCI asynchronous communications interface adapter.

The ACIA also requires a bit-rate generator to act as a clocking mechanism for shifting data in and out serially. The bit-rate generator (MC14411) takes a standard crystal oscillator wired in parallel with a 15MΩ resistor across the inputs X1 and X2. Different frequency divisions are output on F1-F16. The different frequencies are generated with various divide-by ratios (prescalers). The primary prescaler is set by asserting or negating RSA and/or RSB; both are tied high via a 4.7kΩ resistor in this design. The input frequency from the crystal oscillator to the bit rate generator is 1.8432 MHz and the final output is 614.4 kHz on the F1 output. This output is connected to this ACIA's RXCLK and TXCLK inputs. 614.4 kHz is used because the ACIA is set to divide the input clock frequency by 16 to prevent framing errors; this sets the transmit/receive bandwidth to 38400 bits/second. The bit-rate generator is disabled during the 68000 reset cycle which is realized by connecting RST’ to the main reset signal (RESET’). All other outputs (F2-F16) are not connected.

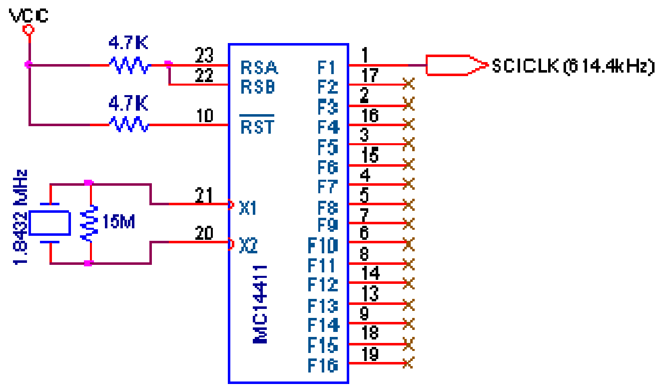


Figure 3.6: Bit-rate generator.

This ACIA also requires a voltage level shifter (MAX232) to shift the incoming RS232 voltages to standard TTL voltage levels, and vice versa for outgoing TTL signals. This device is connected per the revised specifications and diagrams provided by Maxim and can be seen in Figure 3.7. The external capacitors are used to drive a charge pump responsible for the voltage level conversion. The RS232 cable transmit (TxD) and receive (RxD) lines are connected to R1IN and T1OUT’ respectively on the MAX232. The ACIA’s TTL transmit (SCITxD) and receive (SCIRxD) lines are connected to T1IN and R1OUT’ on the MAX232. The remaining inputs/outputs are left disconnected.

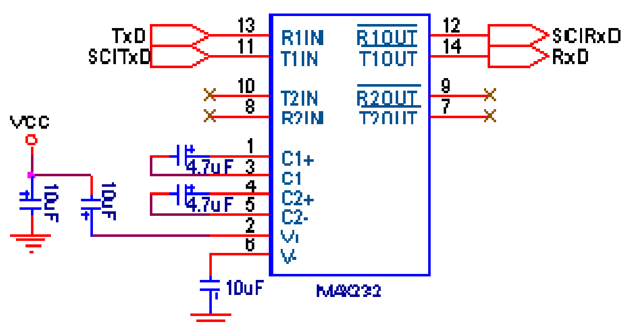


Figure 3.7: Voltage level shifter.

3.1.8 MIDI Interface

The MIDI protocol uses the RS232 protocol at a bandwidth 31250 bps for communications. Therefore an extra ACIA is used for MIDI communications. The MIDI ACIA is connected to the lower data bus and occupies odd addresses in the range of $8000-BFFF. AB1, CS0, CS1, RTS', IRQ', DCD', CTS' and E are connected in the same manner as the SCI ACIA. CS2' is connected to SMIDI' from the MMU. RxD is connected to MIDI in and TxD is connected to MIDI out on the MIDI cable.

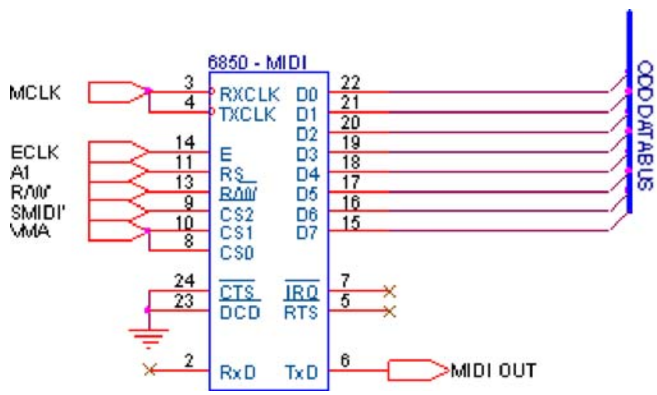


Figure 3.8: MIDI asynchronous communications interface adapter.

The TxCLK and RxCLK inputs receive a square wave at 500 kHz, as this ACIA is also configured to divide the incoming clock frequency by 16. A 74LS393 dual 4-bit counter, shown in Figure 3.9, is used to generate the 500 kHz signal by connecting the Q_D to MCLK. Q_D is also connected to Q_A of the second counter in case the extra frequency divisions are necessary. This would be true if the system clock were increased from 8 MHz to 16 MHz. The active high clear signals on both counters are connected to ground so that the counters will operate continuously. All other outputs are left disconnected.

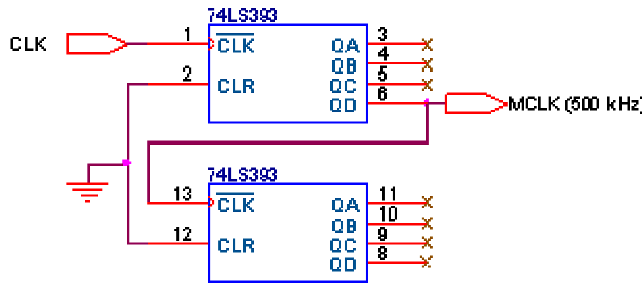
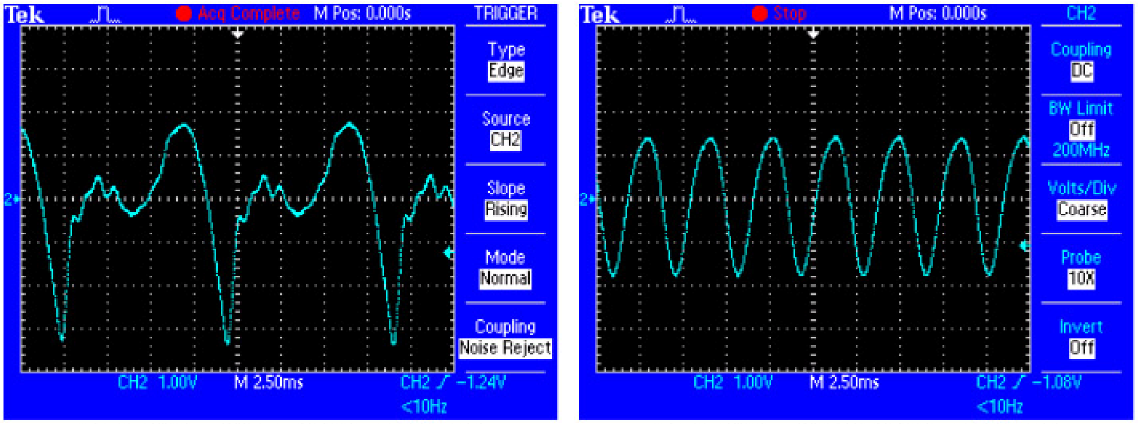


Figure 3.9: MIDI clock divider.

<u>3.1.9 Microphone Input Amplification and Filtering</u>

The microphone for this design is a Sony cardioid microphone. Although it outputs a strong signal when connected to an oscilloscope (see Figure 3.10), the output of the microphone will have to be amplified so the A/D converter can use the smaller amplitude signal. Figure 3.10 shows two separate live vocal samples taken from the microphone used in this design, which were obtained by directly connecting the microphone to the oscilloscope probe. The two samples were acquired from the same subject, but the figure on the left represents a raspy tone, whereas the figure on the right represents more of a pure tone.



(a) (b)

Figure 3.10: (a) Raspy vocal sample and (b) smooth vocal sample captured on a Tektronix TDS 2024 Oscilloscope.

The amplifier chosen for this design is the LM386 audio amplifier, with operation frequency range from 20Hz-10kHz. This was chosen primarily because it is designed for audio applications and that the power supply voltage runs at +5V. A simple amplification arrangement is made by connecting the amplifier output to the non-inverting terminal through a 15 kΩ potentiometer (R2), which is the connected to the input from the microphone through a 1 kΩ resistor (R1). This arrangement amplifies the input signal by $1+\frac{R_2}{R_1}$, and allows the user to adjust the amplification settings. The non-inverting terminal is connected to ground, and the other pins are left disconnected.

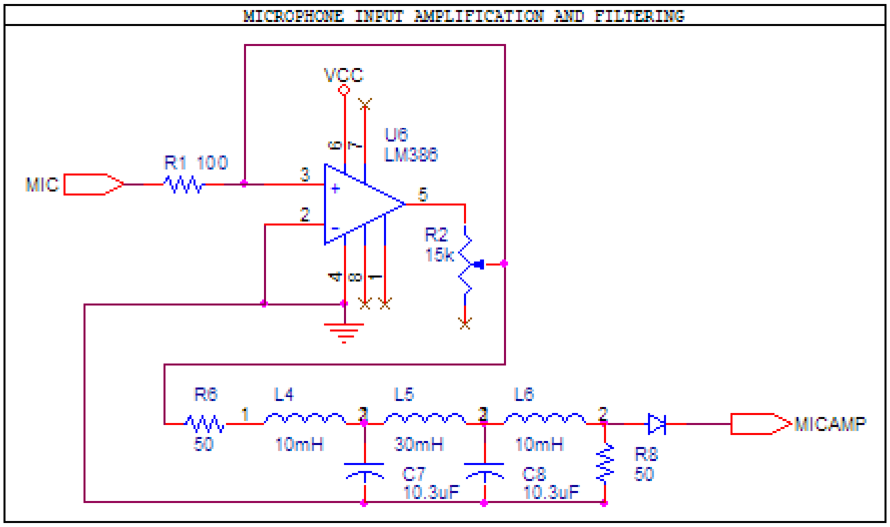


Figure 3.11: Amplification and filtering stages.

One of the problems associated with pitch identification is removing the additional harmonics associated with the fundamental frequency. Figure 2.4 illustrates this well. Therefore, filtering will be implemented on the hardware to remove some of the additional harmonics above a certain frequency. A low-pass or band-pass filter could be used for this application, however, the low-pass filter will be used because the lower-end of the spectrum is close to 0Hz, and less hardware will be required for implementation.

The Butterworth low-pass (maximally flat) approximation is used because it yields the best fit to an ideal low-pass filter at the lower end of the pass-band. However, the tradeoff is that the difference between the approximation and the ideal filter increases greatly toward the high end of the pass-band [Budak 1991]. The cutoff frequency for this design is set at 500Hz to accommodate the hardware limitations imposed by the 68000 (see Section 3.2.2).

The output of the amplifier is also connected to a 5th order Butterworth filter, which is also connected to a ½ wave rectifier to remove noise above 500Hz and also to protect the A/D converter. The filter design was taken from the online design utility at http://www-users.cs.york.ac.uk/~fisher/lcfilter/. However, the filter with the suggested components was simulated using Spice (see Section 5.3.8 for more details) before continuing with the design. The simulations proved successful and the design was accepted. The final output of this filter is referred to as MICAMP on the schematic and is the input to the A/D converter.

3.1.10 A/D Converter (ADC0809) and Sampling Frequency Generator

The A/D converter is the device responsible for sampling the amplified vocal signal and converting the sample to an 8-bit number and resides in even memory addresses in the range $4000-7FFF. The sampling occurs when the 68000 engages a write cycle to the A/D converter, which asserts the STARTADC signal. The sampling rate is determined by a 74HC40103 8-bit binary down counter and the interrupt routine on the 68000.

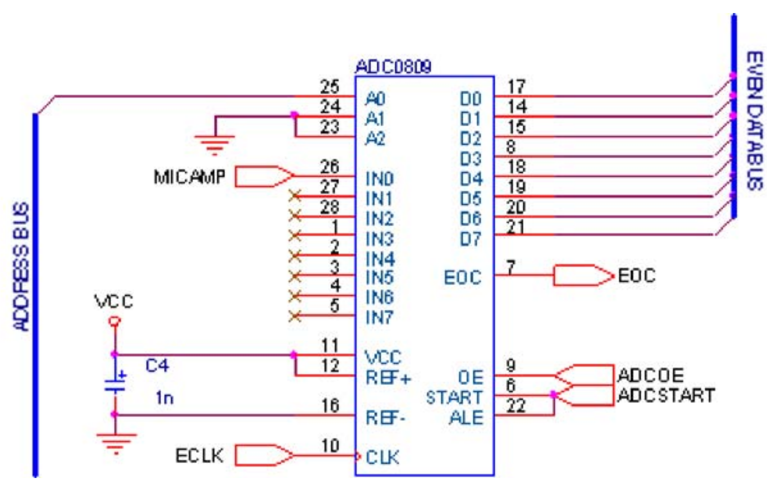


Figure 3.12: A/D converter and connections.

The A/D converter is capable of sampling eight different signals. Only one channel (IN0) is used to sample the amplified and filtered microphone signal (MICAMP); the other channels (IN1-IN7) are left disconnected. The lowest bit of the address bus

(AB1) is connected to A0 of the A/D converter. The other two lines (A1-A2) of the A/D converter are grounded to save wiring space since only one A/D channel is used. The data lines are connected to the upper data bus (DB8-15) on the 68000, so the device occupies even addresses in memory. The A/D converter lacks a chip enable signal, so the ADCOE signal from the MMU is used to indicate a read cycle from the data bus. The ADCSTART signal from the MMU is used during a write cycle, and signals the A/D converter to take a sample. The A/D converter reference voltages REF+ and REF- are respectively connected to V_{CC} (5V) and ground.

The A/D converter conversion rate is determined by the ECLK signal (800 kHz) and is not to be confused with the sampling frequency used by this design. This frequency happened to be in the middle of the acceptable frequency range and requires no additional conditioning.

The next device of interest is the adjustable sampling frequency generator as seen in Figure 3.14. This generator is created using a 74HC40103 8-bit down-counter. The counter uses the ICLK signal, which is the ECLK divided by 16 (see figure 3.13), to count down from 49_{dec}-0_{dec} (50 cycles) when \$31 is input on the preload inputs. The preload inputs are tied to ground via eight 4.7kΩ resistors. The divide-by number can be set by entering the new number in binary via an 8-bit DIP switch. When the switches are left open, the logic low value is input on the preload inputs, else they connect 5V to the preload inputs. Once the counter counts down to 0_{dec}, TC’ is toggled low for one ICLK period.

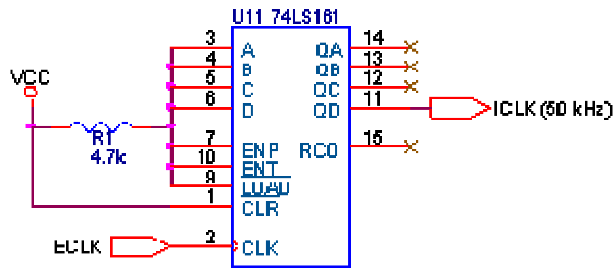


Figure 3.13: Divide-by 16 for sampling frequency generator.

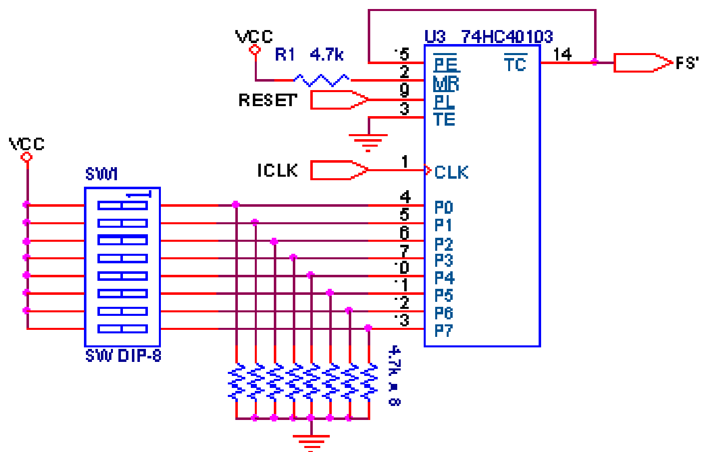


Figure 3.14: Adjustable sampling frequency generator.

TC' is renamed to FS', which is connected to IPL0' on the 68000, and PE' (synchronous preload) on the counter. This causes the sampling interrupt routine to be executed, and the value on the DIP switches to be loaded into the counter again. The asynchronous preload (PL') is connected to RESET'. Master reset (MR') on the counter is unused because the assertion of this signal causes the counter to star counting from 255_{dec}, thus it is tied high via a 4.7kΩ resistor. The remaining control input, TE' is used to enable counting, and is connected to ground because sampling will be controlled in software.

3.1.11 A/D Interrupt Acknowledgement (IACK') and MMU Reset (MRST')

When the 68000 receives an interrupt request on the IPLx' lines, it acknowledges the interrupt request by asserting each of FC0-FC2 simultaneously. Then, the inverted values appearing on inputs IPL0'-IPL2' are placed on AB1-AB3 to indicate the priority level of the interrupt request. The 68000 asserts address strobe (AS') and waits for the assertion of DTACK' or VPA' by the interrupting device, respectively indicating a fully- or auto-vectored interrupt request.

The primary difference between a fully- and an auto-vectored interrupt is that the auto-vectored interrupts only use the interrupt priority level in conjunction with a predetermined vector table to determine where the interrupt service routine is located. With auto-vectored interrupts, there is only one vector table entry associated with each priority level. Conversely, fully-vectored interrupts require the interrupting device to supply the vector table address (divided by four) on the lower data bus. The 68000 will take this address, multiply it by four internally and then searches for the interrupt service routine address in this vector table location. This design utilizes auto-vectored interrupts because there is only one interrupting device and the resulting hardware implementation is much simpler. Table 3.1 shows the addressing scheme for the various types of interrupts (or exceptions).

Vectors Numbers		Address		Space[6]	Assignment
Hex	Decimal	Dec	Hex		
0	0	0	000	SP	Reset: Initial SSP[2]
1	1	4	004	SP	Reset: Initial PC[2]
2	2	8	008	SD	Bus Error
3	3	12	00C	SD	Address Error
4	4	16	010	SD	Illegal Instruction
5	5	20	014	SD	Zero Divide
6	6	24	018	SD	CHK Instruction
7	7	28	01C	SD	TRAPV Instruction
8	8	32	020	SD	Privilege Violation
9	9	36	024	SD	Trace
A	10	40	028	SD	Line 1010 Emulator
B	11	44	02C	SD	Line 1111 Emulator
C	12[1]	48	030	SD	(Unassigned, Reserved)
D	13[1]	52	034	SD	(Unassigned, Reserved)
E	14	56	038	SD	Format Error[5]
F	15	60	03C	SD	Uninitialized Interrupt Vector
10–17	16–23[1]	64	040	SD	(Unassigned, Reserved)
		92	05C		—
18	24	96	060	SD	Spurious Interrupt[3]
19	25	100	064	SD	Level 1 Interrupt Autovector
1A	26	104	068	SD	Level 2 Interrupt Autovector
1B	27	108	06C	SD	Level 3 Interrupt Autovector
1C	28	112	070	SD	Level 4 Interrupt Autovector
1D	29	116	074	SD	Level 5 Interrupt Autovector
1E	30	120	078	SD	Level 6 Interrupt Autovector
1F	31	124	07C	SD	Level 7 Interrupt Autovector
20–2F	32–47	128	080	SD	TRAP Instruction Vectors[4]
		188	0BC		—
30–3F	48–63[1]	192	0C0	SD	(Unassigned, Reserved)
		255	0FF		—
40–FF	64–255	256	100	SD	User Interrupt Vectors
		1020	3FC		—

Table 3.1: Exception vector assignments. Taken from [Motorola 1993].

In this design, IPL0’ is the only signal permitted to change, indicating a level interrupt request. This interrupt is auto-vectored and it is not necessary to distinguish it as such to the interrupting hardware, which greatly simplifies the circuitry needed to create the interrupt acknowledge signal (IACK’).

Based on the interrupt acknowledge sequence described above, IACK’ is formed by NANDing FC0-FC2 and the negated AS’ (AS). IACK’ is then ANDed with RESET’ to form IACK’/MRST’. RESET’ is included here to provide a means to disable the

MMU during reset, because all other MMU control inputs are used. IACK'/MRST' is connected to the VPA' circuitry, to indicate an auto-vectored interrupt has occurred after the A/D converter finishes its conversion.

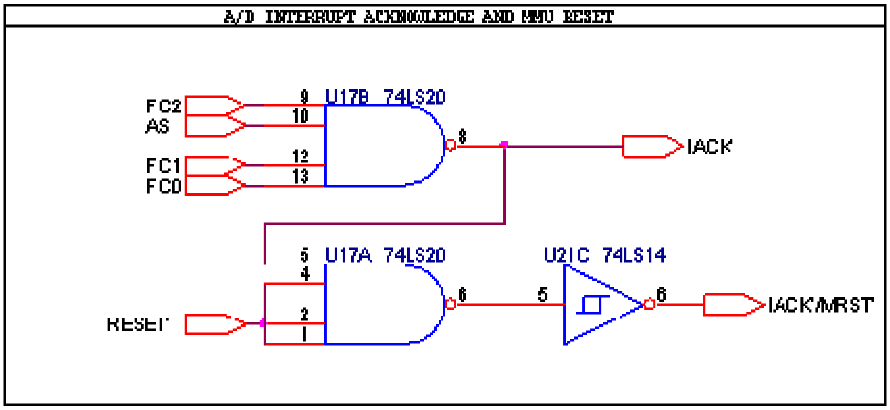


Figure 3.15: Interrupt acknowledge and MMU reset circuitry.

Originally FC0-FC2, AB1, and the end of conversion (EOC) flag from the A/D converter were to be ANDed to form IACK'. This would have also worked because EOC would normally be asserted until the A/D converter starts the conversion, at which time EOC would be negated. By the time the 68000 would start the interrupt acknowledge cycle, EOC would still be negated. Then when EOC was would be reasserted, IACK' would be asserted. This design was rejected because the 68000 would have to wait ~75µs each time for the conversion to complete, leaving only ~50µs until the next sample, and only 400, 8MHz clock cycles to perform the ADMF calculations.

3.1.12 Power-On Reset Circuit

This hardware design has the added feature of a power-on reset circuit. This feature has been added so that when power is applied to the board, the user will not have to press a reset switch to initialize the 68000. The power-on reset circuit is a simple circuit constructed using a simple RC filter with a diode connected in parallel to the resistor, with the cathode connected to V_{in}, and the anode connected to V_{out}. V_{out} is also connected to two Schmitt inverters to convert the slow rise time of the RC circuit to an oscillation-free step signal. The analysis and calculations for determining the resistance and capacitance are shown below.

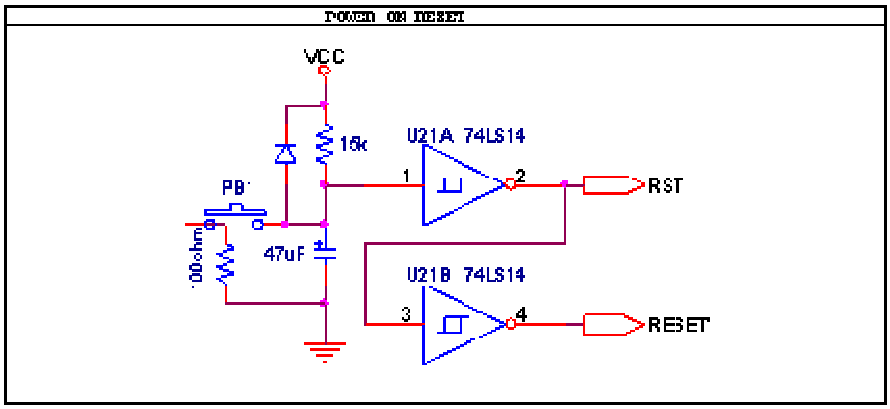


Figure 3.16: Power-on reset circuit.

The Laplace transform and the s-domain are used in this calculation since $V_{in}(t)$ is linear, time invariant, causal, and memory-less.

$$H(s) = \frac{V_{out}(s)}{V_{in}(s)} = \frac{1}{s \cdot C \cdot \left(R + \frac{1}{s \cdot C}\right)} = \frac{1}{s \cdot R \cdot C + 1} = \frac{1}{R \cdot C \cdot \left(s + \frac{1}{R \cdot C}\right)} \tag{3.1}$$

The inverse Laplace transform is applied to the transfer function H(s) to find the impulse response of the system h(t).

$$h(t) = \frac{e^{-t/(R \cdot C)}}{R \cdot C}$$

The input to the system is a step function with amplitude equal to the final desired voltage (V_F) across the capacitor. Therefore, convolution can be used to find the voltage across the capacitor as a function of time.

$$V_{out}(t) = h(t) * V_{in}(t) = \frac{e^{-t/(R \cdot C)}}{R \cdot C} * V_F \cdot u(t) = \frac{V_F}{R \cdot C} \int_0^t e^{-\lambda/(R \cdot C)} d\lambda = -V_F \cdot e^{-\lambda/(R \cdot C)} \Big|_{\lambda=0}^{t} = V_F \cdot \left(1 - e^{-t/(R \cdot C)}\right) \tag{3.2}$$

According to the Schmitt triggered inverter specifications for the 74LS14 provided by Texas Instruments, the minimum input voltage for the trigger to turn to the on state is 1.5V. Setting V_F equal to 5V, the capacitance (C) can be solved for in terms of resistance (R) and time (t). Equations 3.3-3.6 illustrate these steps.

$$V_{out} = 1.5 = V_F \cdot \left(1 - e^{-t/(R \cdot C)}\right) \tag{3.3}$$

$$1 - \frac{1.5}{V_F} = \frac{7}{10} = e^{-t/(R \cdot C)} \tag{3.4}$$

$$\frac{t}{R \cdot C} = -\ln(\frac{7}{10}) \tag{3.5}$$

$$C = -\frac{t}{R \cdot \ln(0.7)} \tag{3.6}$$

The hold time for the 68000 to go into reset is 100ms. A time of 250ms is chosen to allow enough time for the 68000 to go into and remain in reset, and to allow for part tolerances and adjusting part values to match the industrial standards. A resistance of 15kΩ is chosen so that the capacitor will have a smaller value. The values for R and t are substituted into Equation 3.6; solving for C in Equation 3.7 yields 46.73μF.

$$C = -\frac{0.25}{15000 \cdot \ln(0.7)} = 46.728 \mu F \tag{3.7}$$

Since 46.73 micro Farads is not an industrial standard for capacitance, the capacitance is adjusted to 47 micro Farads. Solving Equation 3.7 for time (t), and substituting in R = 15kΩ and C = 47μF yields a reset time of at least 251.5ms before the Schmitt Triggers will toggle the signal back to a logic high.

$$t = -R \cdot C \cdot \ln(0.7) = -(15000)(47 \cdot 10^{-6})\ln(0.7) = 251.456ms \quad (3.8)$$

Figure 3.17 shows a screen capture of this implementation with V_{CC} (green), RESET'/HALT' (yellow), and the voltage across the capacitor (blue). The delay between the time V_{CC} reaches 5V and the time RESET' is negates is approximately 170ms, which satisfies the 100ms requirement imposed by the 68000. The difference between the calculated time and the actual time is explained by the fact that the calculations assumed a unit step input, when in reality, this doesn't happen for a power supply with large capacitance. The calculations also neglect the fact that the Schmitt triggered inverter must also power up during the charging of the power supply's capacitors. This explains the "blip" on the yellow line in Figure 3.17. However, this "blip" and the error associated with the charging power supply are insufficient to cause the malfunction of the power-on reset circuit.

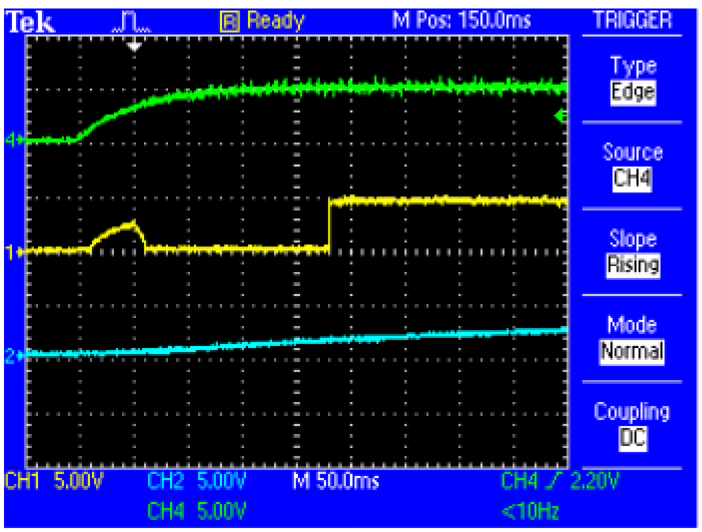


Figure 3.17: Power-on reset results.

3.2 Software Description

The software development for this project was completed in two stages once the general pitch detection algorithm was chosen. The first step involved developing a proof of concept in MATLAB. A variety of sound samples were input into the proof of concept and the results observed. Then the code segments were transformed into the Motorola 68000 assembly language for more simulation and eventual implementation. Section 3.2.1 describes the development of the proof of concept and the results, where as Section 3.2.2 describes the design constraints inherent to implementing the software due to the hardware design. The assembly level implementation of the supporting functions is discussed thoroughly in Section 4.2.

3.2.1 AMDF Proof of Concept

It felt necessary to use a high-level language to verify that the AMDF would work as expected before implementing the AMDF in assembly. MATLAB was used because of the relative ease to manipulate and plot data arrays. The MATLAB simulation code is attached in Appendix II and a variety of plots denoting the success of the AMDF algorithm are included in Section 5.1 along with a more detailed discussion of the simulations. However, the most important segment of code is included below, as this is what is used to implement the AMDF in MATLAB. In this code segment, the array of samples is s(), and the results of each iteration is stored as d(n).

```
for n=1:NMAX
    x=0;
    for k=1:KMAX
        x=x+abs(s(k)-s(k+n-1));
    end
    d(n)=x;
end
```

In this code segment, K_{MAX} and N_{MAX} are to be chosen by the developer, where N_{MAX} represents the total number of samples to be taken and K_{MAX} represents the size of the window used to identify the samples; K_{MAX} must be less than N_{MAX}. Varying these values alters the detected fundamental frequency slightly and larger values require more computation time. Experimentation revealed that setting N_{MAX} in the region of 100-200 and K_{MAX} in the region of ¼ to ½ the value of N_{MAX} yielded good results.

3.2.2 Implementing the Algorithm for the 68000

After the AMDF functionality was verified using MATLAB, the next step was to implement the ADMF in assembly. A few revisions were written, mainly to reduce the number of clock cycles. However, the functionality of the final revision was tested and verified using a fixed data array with the same values in s() from the MATLAB simulation. The routines were simulated using the EASy68K editor, assembler, and simulator and the results can be found in Section 5.2.

Originally, the hardware design revolved around an 8MHz system clock. It will be shown later that this clock will be insufficient unless changes to the design are made. However, the preliminary discussion of the software will include timings for an 8MHz system clock.

The primary algorithm for the AMDF consists of a pair of nested for-loops that creates a computational complexity of $O(K_{MAX}*N_{MAX})$. If $K_{MAX} = 96$ and $N_{MAX} =192$, this means that there will be a total of 18432 iterations through the array before considering any calculations made with the data. In reality, the loops take approximately 1.1M clock cycles according to the Easy68K simulator, which, at an 8.0MHz system clock equates to approximately 0.1375 seconds. This value includes all the steps necessary to perform the AMDF, but not to find the fundamental frequency or output the necessary commands to the synthesizer.

With these considerations in mind, there are two approaches for an interrupt-driven sampling routine. The first approach involves storing all the data from the analog-to-digital converter in an array first, then performing all the calculations after a certain number of measurements have been saved. The second approach involves storing the first M-number samples, then performing the inner loop of the MATLAB code after each consecutive reading until the number of readings is equivalent to the sum of M and N.

The two different approaches have different advantages and disadvantages. With the first approach, there is an abundant amount of time remaining between samples. Therefore, the processor can run at a slower clock frequency and sample at a higher frequency. The disadvantage to the first approach is that the bulk of the calculations are performed at the end of a sampling cycle, and the number of calculations required to perform the AMDF and identify the fundamental frequency requires approximately 1.2M clock-cycles which equates to approximately 0.15 (8 MHz clock) seconds before adding the overhead to send the MIDI information to the synthesizer. The goal, however, is to reduce the delay at less than 20-30ms so as not to be audibly noticeable by the user [Lago 2004].

The second approach performs most of the calculations for the AMDF between each sampling interrupt routine. The advantage of this approach is that the remaining calculations to process the AMDF results require approximately 22000 clock cycles, which equates to approximately 2.75ms (8 MHz clock) before adding the overhead to send the MIDI information to the synthesizer. The disadvantage with this approach, however, is that the inner-loop requires approximately 5600 clock cycles, and if the sampling rate is 3520Hz ($2 \times 1760Hz$), this leaves $\frac{8MHz}{3520Hz} = 2461$ clock cycles to perform the calculations, meaning that the routine is over budget by 3139 clock cycles.

Regardless of which approach is chosen, it is now obvious that the clock frequency must increase to accommodate the real-time calculations, or the sampling rate must significantly decrease. If the second approach is chosen, then the sampling frequency must decrease, regardless.

Later versions of the 68000 are capable of running at 12MHz and 16MHz. If a 16MHz clock is chosen and the maximum allowed vocal frequency is reduced from 1760Hz to 1250Hz (sampling rate of 2500Hz), which corresponds to a difference of six chromatic steps or ½ octave, the new sampling rate would leave 6400 clock cycles between samples to perform the inner-loop calculations. The extra clock cycles will be used to ensure the 68000 has enough time to finish all calculations during the interrupt routine and to allow for further additions to the interrupt routine for the development interface and options. Furthermore, the sampling rate of 2500Hz would be easily generated by the hardware and represents a convenient integer number for division later in the routine.

The other choice is to significantly reduce the sampling frequency to 1000Hz and leave the clock at 8MHz. This leaves 8000 clock cycles between interrupt service requests to perform the calculations described above. However, this would reduce the maximum allowed frequency to 500Hz, which corresponds to a high note of B4, (see Table 4.2). This results in a difference of eleven chromatic steps, which is almost an entire octave below the original sampling frequency. The advantage to this choice is that the lab equipment can be used to perform real-time simulations with the hardware design in its entirety. Therefore, the 8MHz clock will used in conjunction with a lowered sampling frequency.

CHAPTER IV – SYSTEM IMPLEMENTATION

This chapter explores the two necessary components of this design, namely the implementation of the hardware and software for the prototype. The hardware section focuses on the techniques used for construction; a detailed discussion of the design can be found in Section 3.1. The software section focuses on the implementation of the AMDF in assembly. Each functional component required to implement this algorithm in assembly will be discussed in detail.

4.1 Hardware

There are three methods available for assembling this hardware, each with a varying level of permanence and usability. The first method that may come to mind is bread-boarding. This method was immediately rejected because, in the lab, students typically have to reduce the typical 8MHz clock frequency to 4MHz to reduce the noise on the breadboard tracks. The next method involves laying copper tracks and sending the design to be fabricated. Once the design is received, the components would be soldered to the board. However, the final product is rather permanent and difficult to modify should there be an error in the hardware design. Therefore, the hardware for this design will be assembled using VectorBord prototyping board. The model used is double-sided and contains individual solder pads spaced 0.050 inches apart on both sides with non-plated-through holes. This method combines the advantages of both the previous methods, in that designs are somewhat permanent, but lend themselves to rework more easily than a fabricated design. The prototyping board consists of 2960 pads arranged in 40 rows by 74 columns on either side of the board, leaving a total of 5920 pads for potential use when considering using both sides of the board.

The first necessary step to implementing the hardware design is to create a board layout. This is always done for fabricated design and sometimes done for prototyping. It is easier to solder tracks, parts, and wires when the layout has been visualized and developed on paper. In industry, there exists software for board layout; however, that software is expensive and doesn't necessarily lend itself to prototyping layout as well as track layout. Therefore, Adobe Acrobat CS is used to draw the board layout. This might not be the best tool, but Photoshop files support various image layers, which has its advantages.

To start, the prototyping board was rendered to scale by creating a series of gridlines and can be seen in Figure 4.1. The gridline for every tenth pad was made thicker for a quick visual reference. No through-holes were rendered because they clutter the display. A large blue track representing the ground track was placed around the gridlines. Finally, two yellow segments were placed at the bottom of the board to represent the area on the board where the ground track is removed. This is done so that the bottom segment of the topside ground track can be used to carry the system clock across the board without as much fear of noise permeating the neighboring lines.

Figure 4.1: Topside view of the empty prototyping board.

Even before placing digital parts on the board, it was necessary to conceptualize the size of the parts in relation to the size of the board. The parts were loosely arranged based on their pinouts and size. Once the loose arrangement was loosely finalized, the parts' digital equivalents were placed on the digital grid as shown in Figure 4.2. The parts were created using copies of the pinouts taken from the various part specifications, primarily so the pin names would be included on the board layout and could be used for debugging and quick reference when looking at the physical creation.

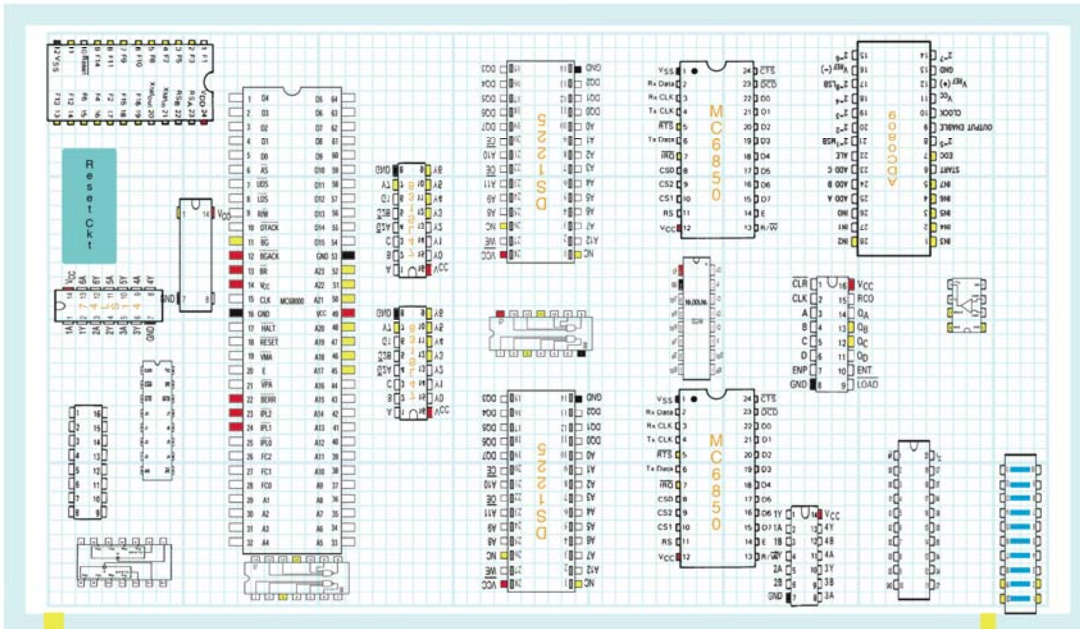


Figure 4.2: Topside view of loosely place parts on the prototyping board.

The next step was to create a convention for placing tracks, wires, and discrete components. One may also notice the additional colorings on the pinouts above. These were added to certain pins for quick reference while creating the board layout. Table 4.1 describes all conventions used in the board layout. The choices for color were mostly arbitrary.

	Item	Color
Pins	VCC	Red
	Tied High	Red
	VSS/GND	Black
	Not Connected	Yellow
Lines	Topside Track	Blue
	Topside Wire	Orange
	Backside Track	Green
	Backside Wire	Purple
	Resistors/Caps Either Side	Red

Table 4.1: Board layout color-coding conventions.

Once the conventions and the loose layout were finalized, the board layout began. For this design, a combination of solder tracks and wires were used to create the layout. For reference, connecting two or more pads on the prototyping board together by dragging the solder across the thin break between the pads creates a solder track. Conversely, soldering a wire involves soldering the wire to two separate pads on the board. There is a mixture of solder tracks and wires in this design to reduce the clutter of additional wires. The final board layout for the top and bottom sides of the board can be found in Appendix I.

4.2 Software

Now that the clock frequency and sampling rate have been finalized and the hardware design and layout have been completed, it is appropriate to discuss the methods to identify a frequency. After the 68000 initializes and all preliminary initializations in code are complete, the general approach is to sample the data, perform the AMDF calculations, find the locations of the minima, calculate the periods between the minima, calculate the average frequency, and send the data to the synthesizer.

4.2.1 Initializations

Upon coming out of reset, the 68000 code clears the data and address registers used for the counters and resets the address registers used for the sampled data array and the AMDF results array. The two ACIAs are also reset by sending $03 to their control registers. Then the ACIAs are configured to divide their clocks by 16_{dec}, to set the parity equal to none, to set the data for 8 data bits, and one stop bit for general communications. This results in a bit rate of 38400bps for the SCI ACIA and 31250bps for the MIDI

ACIA. Next, interrupt requests are enabled on the 68000 by ANDing the status register (SR) with $F8FF. At this point, the software loops by repeatedly calling the SENDMIDI routine and waits for an interrupt.

4.2.2 Sampling and performing the AMDF (ADCIRQ)

The sampling generator is the only device in this design that generates an (auto-vectored) interrupt. The sampling occurs every sampling period as adjusted by the adjustable sampling frequency generator (see Section 3.1.10), and this routine executes at that time. This interrupt service routine is perhaps the most important routine in this design because it is the top level. Figure 4.5 contains a flow diagram representing this routine. Please note that this figure represents one routine iteration. Additional iterations are performed on subsequent interrupt requests.

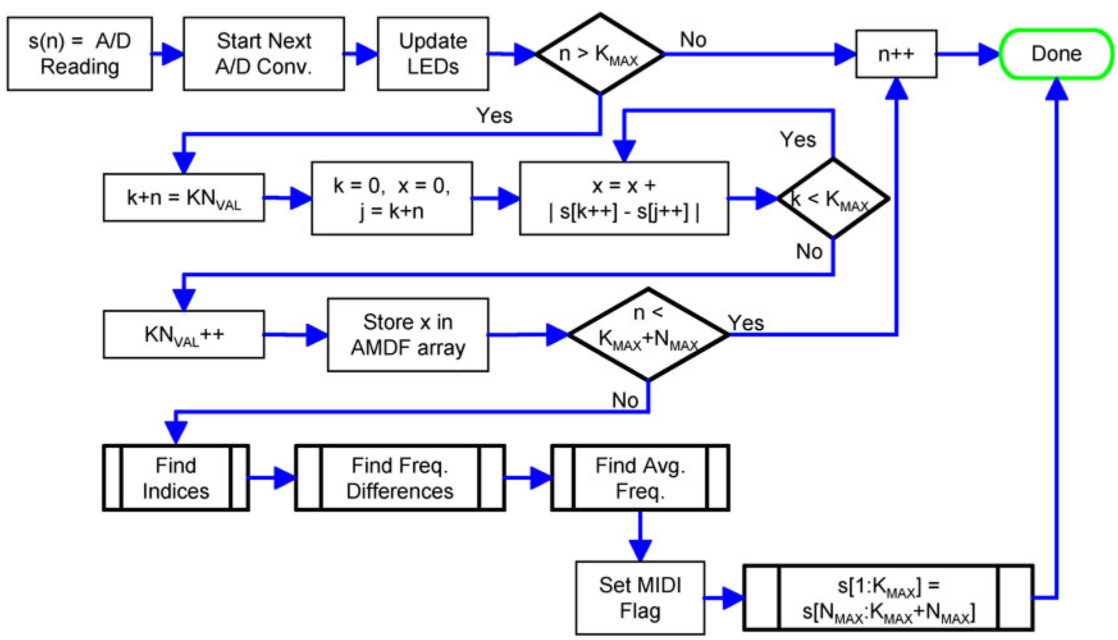


Figure 4.3: Flow diagram for interrupt service routine (ADCIRQ)

When this interrupt service routine begins execution, the data sample from the A/D converter is read and stored. Then a value is written to the A/D converter to start the next conversion. This command is placed as close to the beginning as possible because the remaining assemble commands used to calculate the AMDF array do not guarantee a fixed number of clock cycles between each execution of the routine. Therefore, placing the next write instruction immediately after the read instruction guarantees the next conversion will be complete before the next sampling period and this eliminates any sampling period discrepancy due to software timing, except the very first sample.

This service routine is also responsible for calculating the AMDF array values by executing the nested for-loops as discussed in the proof of concept section (see Section 3.2). A total of $K_{MAX}+N_{MAX}$ samples must be taken to complete the AMDF. However, at least K_{MAX} (number of samples in the sliding window) must be taken before the inner-loop calculations may occur. Thus, each time a sample is taken, the number of measurements is compared to K_{MAX}. If the number of measurements is less than K_{MAX}, the routine exits and the 68000 waits for the next sampling interrupt request to occur, else the AMDF calculations begin.

After at least K_{MAX} samples are taken, one (inner-loop) iteration of the AMDF is calculated, and the 16-bit result is stored in a new array in memory named DD, which is equivalent to the d-array in the simulations and literature. However, this will herein be referred to as the AMDF array to avoid confusion. The data stored during each inner loop calculation is $x = x + |s(k) - s(k+n)|$, where *n* and *k* are indices and *s* is the sampled data array. After K_{MAX} elements of the single iteration through the inner loop have been processed, the interrupt routine exits. This process continues until $K_{MAX}+N_{MAX}$ samples are taken, at which time interrupt requests to the 68000 are disabled in software, and the remaining functions to find the fundamental frequency are called.

4.2.3 Finding the periods between minima (FINDINDICES)

This routine searches through the ADMF results stored in the DD array and finds the corresponding AMDF array indices correlating to the local minima that fall below a defined threshold. Figure 4.6 illustrates a typical set of results from using the AMDF, the threshold value, and the identified local minima falling below the threshold. Once these array indices are identified they will be used to identify the fundamental frequency.

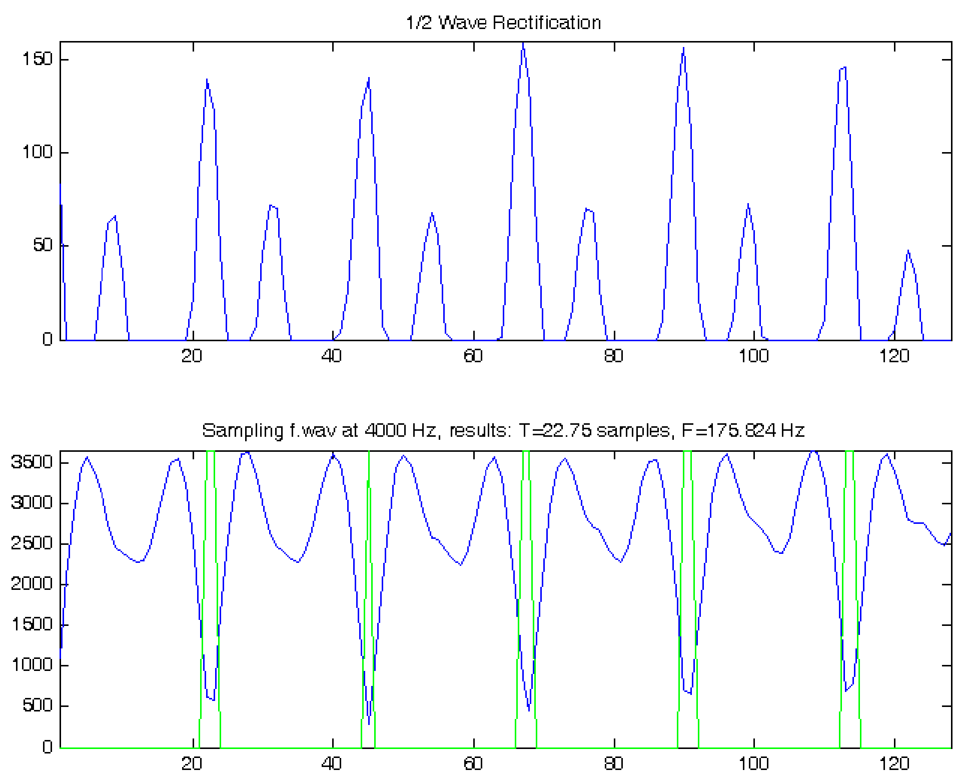


Figure 4.4: Sample AMDF results (blue) and regions where the data fall below the detection threshold (green).

The general approach to identifying the useful minima in the AMDF results array is to save all indices of minima where the AMDF samples fall below 25% of the maximum value in the AMDF array. Since there are two sets of minima each period, it is necessary to define a cutoff point for which to define a minimum. The range of the data will vary depending on the number of samples considered, thus it is necessary to redefine the threshold value each time the AMDF finishes (before this function executes). The threshold value is set to 25% of the maximum value in the AMDF array because setting the threshold to 25-30% resulted in better performance during the MATLAB simulations when performing a search. However, multiplying by 25% is equivalent to dividing by four, which is a simpler operation in assembly, thus requiring fewer precious clock cycles. Once the threshold has been identified, a sequential search for the local minima begins as described in the following paragraphs and graphically in Figure 4.7.

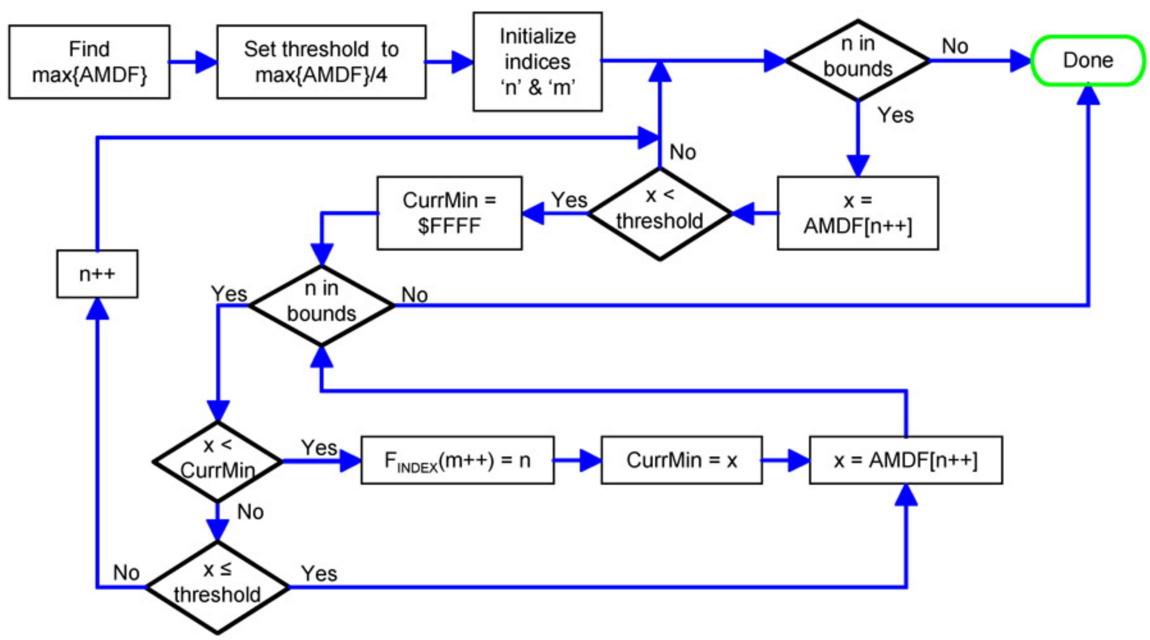


Figure 4.5: Flow diagram for finding indices.

The first step in finding the minima is to identify the maximum value in the AMDF array. A variable called MAXOFDD is initialized to zero and a sequential search through the array is executed and each value is compared to MAXOFDD. If the current value is greater than MAXOFDD, MAXOFDD assumes the current AMDF value. Once the search is complete, MAXOFDD is stored in a data register and divided by four to set the threshold for the next set of searches.

Then this routine loads the initial AMDF array index into an address register and iterates through the array until a value that is less than or equal to the threshold is identified. If the current value is less than the threshold, the routine branches to a smaller loop and finds the local minimum value in the data that fall below the threshold, while continuing to iterate through the AMDF array. Once the sequentially values rise above the threshold, the array index pointing to the minimum value in the aforementioned data segment is stored to the F_{INDEX} array. The smaller loop exits and the other loop resumes checking for values falling below the threshold. This process repeats until all values in the AMDF results array have been examined.

4.2.4 Calculating the difference between indices (FDIFF)

Once the F_{INDEX} array has been populated, the periods can be identified in units of address offsets. Subtracting the array indices will result in the number of sample periods between each minimum. This is represented by the formula: $F_{INDEX}(n) = F_{INDEX}(n+1) - F_{INDEX}(n)$. The F_{INDEX} array is re-used to save memory and because the data in its present form is not used later in the code. Figure 4.6 shows a flow diagram for this function.

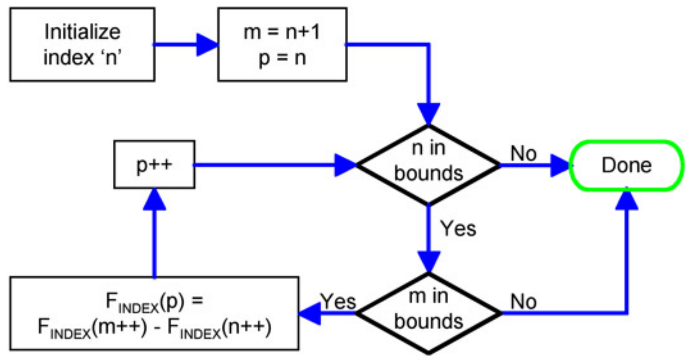


Figure 4.6: Flow diagram for finding index differences

The periods are calculated by loading the initial address of F_{INDEX} into two address registers. One address register is incremented by two to point to the next index in the array, thereby accessing $F_{INDEX}(n+1)$. The other address register is used to access $F_{INDEX}(n)$. As the array is examined, $F_{INDEX}(n+1)$ is compared to the end-of-array flag. If equal, the routine exits, else the subtraction is stored in $F_{INDEX}(n)$. An end-of-array flag is used instead of counting the number of entries in the array because the number of entries varies and can make debugging in a hex-dump more difficult, whereas an end-of-array flag is easy to spot at a glance. Before the routine exits, the last entry of the original F_{INDEX} array is replaced with a different end-of-array flag as the last entry in F_{INDEX} is no longer valid. This new end-of-array flag will be used by FAVG (see Section 4.2.5).

4.2.5 Finding the fundamental frequency (FAVG)

This routine sums all entries in F_{INDEX} after FDIFF has been executed and calculates the average frequency for the array of samples. Recall in Figure 4.6 that the AMDF produces a series of minima. Technically the difference between any two close minima will yield the fundamental period (T_0). The results, however, produce more than one of these minima, which will cause the associated period to vary. Hence, the average is used to create one fundamental period, which is converted to the equivalent fundamental frequency (f_0).

The number of entries in F_{INDEX} is multiplied by the sampling rate after the sum of the periods (stored in F_{INDEX}) is calculated. This result is divided by the sum of elements in F_{INDEX} from FDIFF. The quotient is an integer and is stored as the fundamental frequency (f_0). Equation 4.1 mathematically represents this process.

$$f_0 = \frac{f_s \cdot N}{\sum_{i=1}^{N} F_{INDEX_i}} \tag{4.1}$$

The remainder is used to determine whether to round the result up. If the remainder is greater than or equal to half of the divisor (i.e. the sum of elements in F_{INDEX}), then f_0 should be rounded up. This check is accomplished by dividing the previously calculated sum of all F_{INDEX} entries by two and by comparing the remainder from the division to this value. If the remainder is greater than or equal to the half the sum, f_0 is incremented by 1.

4.2.6 Converting frequency to MIDI (FREQ2MIDI)

This routine is actually the simplest of all the routines regarding conversion of data. The binary search is one of the first methods that come to mind to convert the frequency to a MIDI note number since the numbers can be expressed as ordered lists. This method has a computational complexity proportional to O(log N) and is memory efficient. However, execution time efficiency and consistency are of greater interest in this design. Instead of using the binary search to find the frequency in a table, interpolating between numbers, and assigning a value for the MIDI note number, the frequency (f_0) itself is used as an index offset for a very long linked list, since f_0 is an integer. This linked list contains each MIDI note number ($36\text{-}84_{dec}$) corresponding to frequencies from 63-1078Hz and Table 4.2 shows correlating integer frequencies to pitches and MIDI note numbers.

Pitch	Actual Freq. (Hz)	Valid Freqs. (Hz)	Address Offsets	MIDI Note Number
C2	65.406	63-67	0-4	36
C#2	69.296	68-71	5-8	37
D2	73.416	72-75	9-12	38
D#2	77.782	76-80	13-17	39
E2	82.407	81-84	18-21	40
F2	87.307	85-89	22-26	41
F#2	92.499	90-95	27-32	42
G2	97.999	96-100	33-37	43
G#2	103.826	101-106	38-43	44
A2	110	107-113	44-50	45
A#2	116.541	114-120	51-57	46
B2	123.471	121-127	58-64	47
C3	130.813	128-134	65-71	48
C#3	138.591	135-142	72-79	49
D3	146.832	143-151	80-88	50
D#3	155.564	152-160	89-97	51
E3	164.814	161-169	98-106	52
F3	174.614	170-179	107-116	53
F#3	184.997	180-190	117-127	54
G3	195.998	191-201	128-138	55
G#3	207.652	202-213	139-150	56
A3	220	214-226	151-163	57
A#3	233.082	227-240	164-177	58
B3	246.942	241-254	178-191	59
C4	261.626	255-269	192-206	60
C#4	277.183	270-285	207-222	61
D4	293.665	286-302	223-239	62
D#4	311.127	303-320	240-257	63
E4	329.628	321-339	258-276	64
F4	349.228	340-359	277-296	65
F#4	369.994	360-380	297-317	66
G4	391.995	381-403	318-340	67
G#4	415.305	404-427	341-364	68
A4	440	438-453	365-390	69
A#4	466.164	454-480	391-417	70
B4	493.883	481-508	418-445	71
C5	523.251	509-538	446-475	72
C#4	554.365	539-570	476-507	73
D5	587.33	571-604	508-541	74
D#4	622.254	605-640	542-577	75
E5	659.255	641-678	578-615	76
F5	698.457	679-719	616-656	77
F#5	739.989	720-761	657-698	78
G5	783.991	762-807	699-744	79
G#4	830.609	808-855	745-792	80
A5	880	856-906	793-843	81
A#5	932.328	907-960	844-897	82
B5	987.767	961-1017	898-954	83
C6	1046.502	1018-1078	955-1015	84

Table 4.2: Pitches corresponding to actual fundamental frequencies, valid integer representations from the AMDF, linked list memory offsets, and MIDI note numbers. All numbers are in base 10. Red shaded areas represent pitches outside the normal human vocal range. The blue shaded area is middle C on the piano.

This conversion method is inefficient regarding memory use because the linked list required 1016_{dec} entries, but there is plenty of memory available. However, the method is extremely efficient regarding speed because the computational complexity for finding the appropriate MIDI note number is O(1). This works because there is only one MIDI note number for each frequency, whereas there may be multiple frequencies for each MIDI note number.

4.2.7 Sending a Note to the Synthesizer (SENDMIDI)

This routine is very unique in that it is the only routine that is continually executed outside the interrupt service routine (ADCIRQ). The reason for this is simply because the ACIA itself forms a bottleneck in this design. One may recall from Section 2.4.4 that the minimum time required to send a MIDI message of three bytes length is 0.96 ms. This poses a problem because the sampling routine triggers every 1.000 ms. Therefore, this routine must be placed outside the sampling routine and called continuously in a never-ending while-loop. A simple flag set in the FAVG routine (Section 4.2.5) will be used to determine whether a note is to be sent to the synthesizer, the current note is to be turned off, and if a new note is to be sent to the synthesizer. Figure 3.9 illustrates the functionality of one routine via a flow diagram.

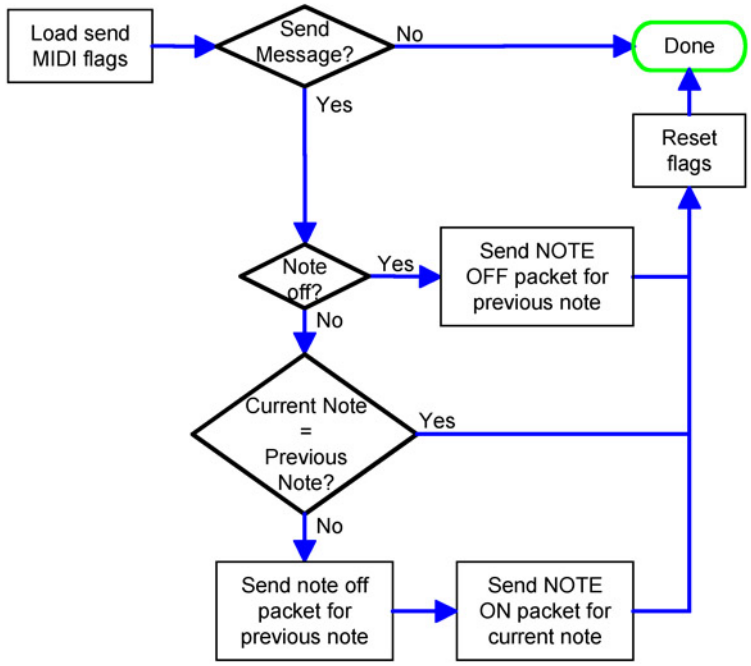


Figure 4.7: Flow diagram for sending MIDI commands.

Aside from the timing considerations, this routine analyzes a variable called MIDIFLAG to determine what packets are to be sent. The first check determines if any packets are to be sent at all. If not, then the routine exits. The second check determines if the previous note is to be disabled without sending a new note to the synthesizer. This is useful if the user quits singing. The next check determines if the previous note needs to be disabled with the intentions of sending a new note (on command) to the synthesizer.

If a note is to be enabled or disabled, the routine takes the note number specified in the MIDINOTE variable and formats a MIDI protocol packet. This packet is sent to the synthesizer using the MIDI ACIA. Recall that the packet format for turning a note on or off is as follows:

```
                    sc nn vv

  s = Status Nibble (Command)
      8 = Note Off
      9 = Note On
  c = Channel Number (0-15dec)
 nn = MIDI Note Number (0-127dec)
 vv = Note Velocity
```

Each time this routine is called with the intentions of sending a new note, the note off MIDI command is sent with the previously activated note (PREVNOTE) if the previous note differs from the current note. Then the value of the current note, stored in MIDINOTE, is loaded into a data register. The current note is sent with the note on MIDI command, and the current value of MIDINOTE is saved in the variable PREVNOTE, so that it may be turned off during the next instance of SENDMIDI. Before the routine exits, the flags are reset so no additional data is accidentally transmitted when unnecessary.

CHAPTER V – RESULTS

5.1 MATLAB AMDF Proof of Concept Simulations

The AMDF algorithm was simulated using MATLAB v7.0 and pre-recorded Windows audio (wav) files set to record 8-bit samples on a single channel at the minimum frequency the software allowed (8.000 kHz). This data was down-sampled to 4.0, 2.0, 1.6 and 1.0 kHz to by removing every second, fourth, fifth and eight sample, respectively, to better approximate the sampling rate to be used for the final design. To simulate the hardware design, the sample data was then half-wave rectified by setting all data with negative amplitude equal to zero. Although the half-wave rectification is non-linear and introduces additional harmonics, the AMDF performed similarly to the non-rectified case in the preliminary simulations. MATLAB returns wav file values in the range of minus one to positive one and so the data were scaled by 255 to simulate the output from the A/D converter. Finally, all floating-point numbers were rounded down to simulate unsigned bytes from the A/D converter.

The AMDF code obviously outputs the reported fundamental frequency for a selected data segment of length $K_{MAX}+N_{MAX}$. However, it is necessary to compare this output to another frequency measure of the same data. This is accomplished by computing the power spectrum of the same data segment by using a 1024 point FFT. Once computed, maxima that appear above a certain threshold were calculated and their frequencies recorded. These recorded frequencies were compared to the output of the AMDF. This process is repeated for 12 data segments for each file. Each new segment uses the last K_{MAX} entries of the previous data segment to better represent how the assembly code will function. Additionally, the same 12 data segments for each file were analyzed for sampling frequencies of 4.0, 2.0, 1.6 and 1.0 kHz. Finally, three files were analyzed with a male signer singing tones approximating F3 (174.614 Hz), A3 (220 Hz) and C4 (261.626). This set of simulations is by no means an exhaustive set of tests, but gives an idea of the success or failure of the algorithm. The code responsible for this can be seen in Appendix II and the results can be seen in Figures 5.1-5.4 and Tables 5.1-5.3.

The initial simulation results (not shown) were very disappointing. However, varying K_{MAX} and N_{MAX} varied performance of the AMDF. In the results presented in Tables 5.1-5.3, the values of K_{MAX} and N_{MAX} had to be reduced for the lower sampling frequencies, else the reported results had an extremely high error. Ultimately, using an adaptive window and adaptive cutoff threshold would increase the AMDF performance [Kim 1998]. Figures 5.1-5.4 graphically show simulation results for one iteration of the "A3.wav" file sampled at each of the four sampling frequencies. In each figure, there are three subfigures. The top subfigures show the half-wave rectified input signal, where the sample number is represented on the horizontal axis and signal amplitude is represented on the vertical axis. The middle subfigures show the results of the AMDF as applied to the data in the top subfigures. In the middle subfigures, the horizontal axis represents the sample number, the vertical axis represents the AMDF results, and the green line represents the threshold for identifying minima relevant to the frequency identification. The bottom subfigures show the power spectra for the data from the top subfigures, where the horizontal axis represents the frequency in Hz, and the vertical axis is the magnitude of the power spectra.

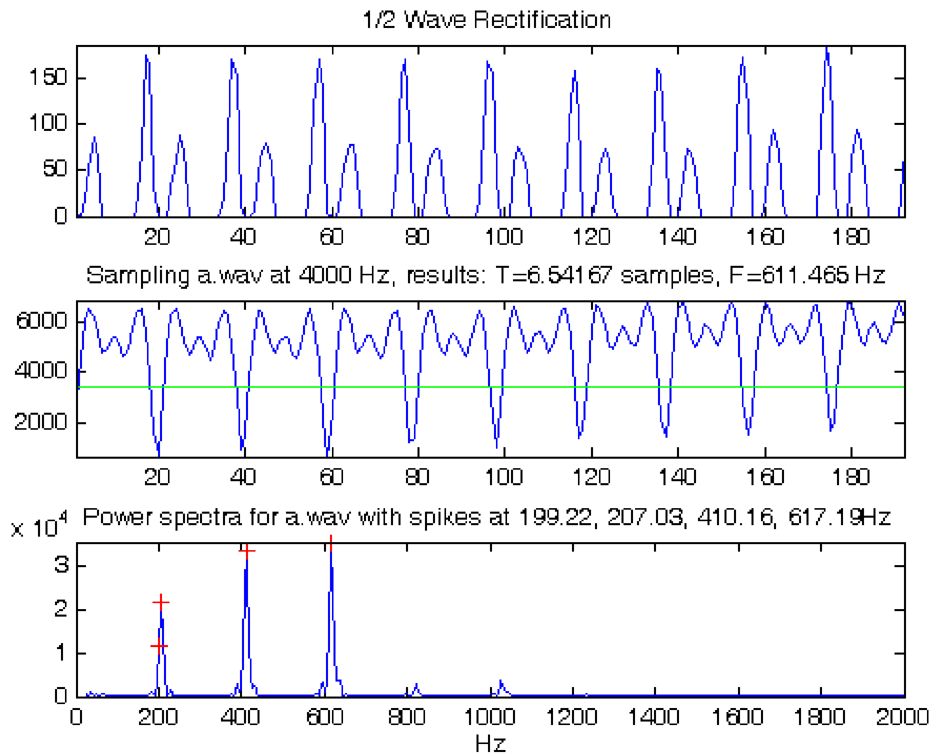


Figure 5.1: AMDF simulation results for A3, f_s = 4.0 kHz.

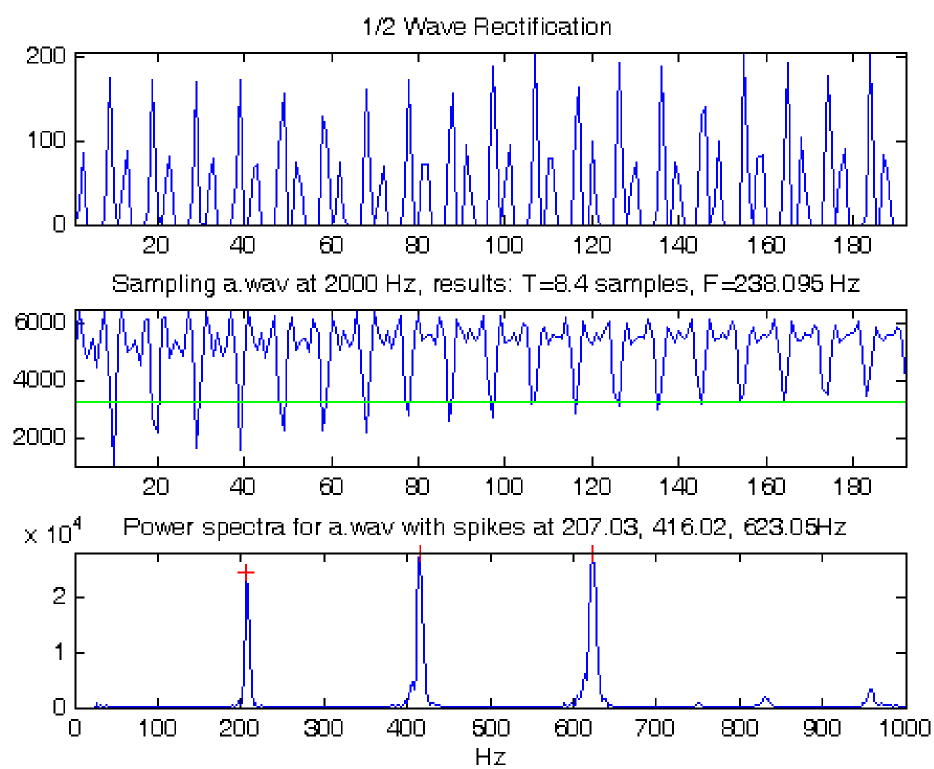


Figure 5.2: AMDF simulation results for A3, f_s = 2.0 kHz.

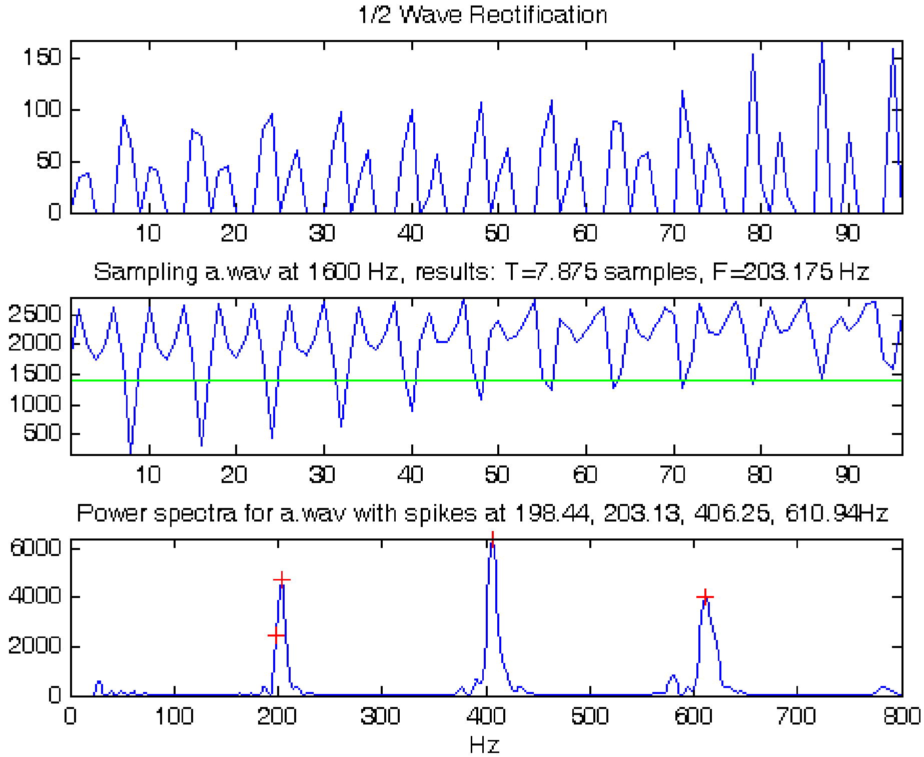


Figure 5.3: AMDF simulation results for A3, f_s = 1.60 kHz.

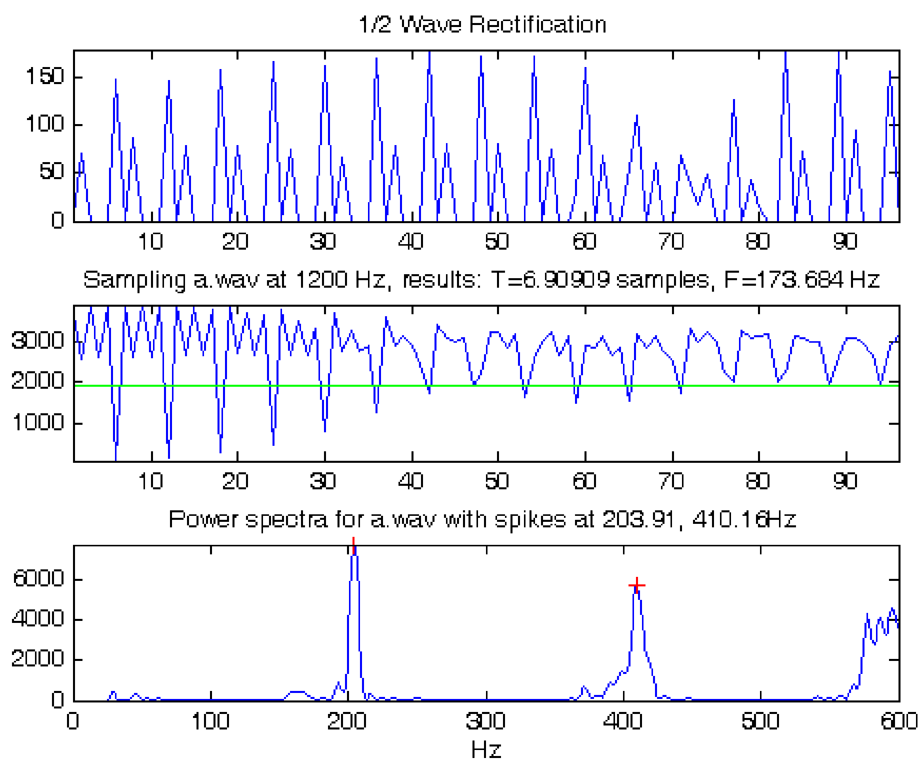


Figure 5.4: AMDF simulation results for A3, f_s = 1.0 kHz.

	Iteration	'f.wav' - Sampling Frequency (Hz) 4000	2000	1600	1000
AMDF	1	606.061	193.182	213.333	166.667
	2	602.41	237.288	256	166.667
	3	598.802	269.663	210.526	148.936
	4	550.898	239.521	231.579	150.943
	5	550.898	309.524	231.579	144.578
	6	571.429	285.714	190.476	144.578
	7	571.429	272.189	213.333	144.578
	8	571.428	331.361	207.059	144.578
	9	520.71	261.905	188.235	132.53
	10	544.379	214.286	169.412	120.482
	11	520.71	273.81	169.412	166.667
	12	497.041	292.994	169.412	83.333
FFT	1	171.88	169.92	171.88	169.92
	2	171.88	169.92	170.31	169.92
	3	167.97	167.97	170.31	168.95
	4	167.97	167.97	170.31	168.95
	5	167.97	167.97	168.75	168.95
	6	167.97	167.97	168.75	167.97
	7	167.97	167.97	168.75	167.97
	8	167.97	166.02	168.75	167.97
	9	167.97	166.02	168.75	167.97
	10	167.97	166.02	168.75	167.97
	11	167.97	166.02	168.75	166.99
	12	164.06	166.02	168.75	166.99
| Error (Hz) |	1	434.181	23.262	41.453	3.253
	2	430.53	67.368	85.69	3.253
	3	430.832	101.693	40.216	20.014
	4	382.928	71.551	61.269	18.007
	5	382.928	141.554	62.829	24.372
	6	403.459	117.744	21.726	23.392
	7	403.459	104.219	44.583	23.392
	8	403.458	165.341	38.309	23.392
	9	352.74	95.885	19.485	35.44
	10	376.409	48.266	0.662	47.488
	11	352.74	107.79	0.662	0.323
	12	332.981	126.974	0.662	83.657

* NMAX =192, KMAX = 96 for fs = 4000 and 2000 Hz.
** NMAX =96, KMAX = 48 for fs = 1600 and 1000 Hz.

Table 5.1: AMDF simulation results from MATLAB for F3.

		'a.wav' - Sampling Frequency (Hz)			
	Iteration	4000	2000	1600	1000
AMDF	1	493.827	200	200	189.873
	2	621.118	237.288	200	188.406
	3	600	285.714	222.785	180.723
	4	607.595	241.379	202.532	186.441
	5	611.465	238.095	203.175	166.667
	6	589.744	277.457	205.128	158.73
	7	554.913	263.736	205.128	181.818
	8	551.724	243.94	225.641	137.931
	9	574.713	287.293	205.128	137.931
	10	528.736	287.293	228.571	168.831
	11	574.713	266.667	225.571	151.163
	12	574.713	266.667	207.792	139.535
FFT	1	199.22	203.13	198.44	203.13
	2	199.22	205.08	198.44	199.22
	3	199.22	205.08	198.44	204.1
	4	199.22	207.03	198.44	204.1
	5	199.22	207.03	198.44	205.08
	6	199.22	207.03	198.44	205.08
	7	207.03	208.98	198.44	206.05
	8	207.03	208.98	204.69	206.05
	9	207.03	208.98	204.69	206.05
	10	207.03	210.94	206.25	207.03
	11	207.03	210.94	206.25	208.01
	12	210.94	210.94	206.25	208.01
| Error (Hz) |	1	294.607	3.13	1.56	13.257
	2	421.898	32.208	1.56	10.814
	3	400.78	80.634	24.345	23.377
	4	408.375	34.349	4.092	17.659
	5	412.245	31.065	4.735	38.413
	6	390.524	70.427	6.688	46.35
	7	347.883	54.756	6.688	24.232
	8	344.694	34.96	20.951	68.119
	9	367.683	78.313	0.438	68.119
	10	321.706	76.353	22.321	38.199
	11	367.683	55.727	19.321	56.847
	12	363.773	55.727	1.542	68.475

* NMAX =192, KMAX = 96 for fs = 4000 and 2000 Hz.
** NMAX =96, KMAX = 48 for fs = 1600 and 1000 Hz.

Table 5.2: AMDF simulation results from MATLAB for A3.

	Iteration	'c.wav' - Sampling Frequency (Hz) 4000	2000	1600	1000
AMDF	1	906.077	254.237	290.909	250
	2	491.228	285.714	256.79	142.857
	3	564.706	285.714	217.284	131.579
	4	840.909	268.657	237.037	117.647
	5	666.667	292.683	217.284	160
	6	613.333	352.941	216.216	172.414
	7	640	57.143	220	108.108
	8	601.77	205.714	240	250
	9	571.429	262.857	241.509	250
	10	666.667	148.571	241.509	81.633
	11	617.143	170.455	272.34	122.449
	12	545.455	181.818	240	58.824
FFT	1	*468.75*	*472.66*	*470.31*	241.21
	2	*468.75*	251.95	*470.31*	241.21
	3	*472.66*	253.91	*470.31*	241.21
	4	242.19	248.05	*471.88*	241.21
	5	242.19	250	*473.44*	290.04
	6	246.09	250	240.63	239.26
	7	250	251.95	240.63	254.88
	8	250	251.95	242.19	254.88
	9	253.91	251.95	243.75	253.91
	10	253.91	250	245.31	253.91
	11	253.91	250	246.88	253.91
	12	250	250	248.44	252.93
| Error (Hz) |	1	*437.327*	*218.423*	*179.401*	8.79
	2	*22.478*	33.764	*213.52*	98.353
	3	*92.046*	31.804	*253.026*	109.631
	4	598.719	20.607	*234.843*	123.563
	5	424.477	42.683	*256.156*	130.04
	6	367.243	102.941	24.414	66.846
	7	390	194.807	20.63	146.772
	8	351.77	46.236	2.19	4.88
	9	317.519	10.907	2.241	3.91
	10	412.757	101.429	3.801	172.277
	11	363.233	79.545	25.46	131.461
	12	295.455	68.182	8.44	194.106

* NMAX =192, KMAX = 96 for fs = 4000 and 2000 Hz.

** NMAX =96, KMAX = 48 for fs = 1600 and 1000 Hz.

Table 5.3: AMDF simulation results from MATLAB for C4. FFT errors are shown in italicized red.

These results indicate that the real-time implementation of this device will have errors that will likely be perceptible to the user. The better performances of the AMDF are at best +/- a musical half step from the intended frequency, while other errors are up to two octaves off. Ideally, a faster processor would yield better results because additional techniques could be implemented to correct the errors.

5.2 Easy68K Simulations

The next logical step is to simulate the assembly level implementation before using the constructed hardware. There are two ultimate goals for these simulations. The first goal is to ensure that the assembly code results match the MATLAB simulation results for the same data set, aside from rounding. The second goal is to ensure that the code execution time is short enough to operate between sampling periods.

The assembly implementation must be modified slightly to accommodate the fact that the data are not sampled in real time with a microphone. This was accomplished by saving the source data used in the MATLAB simulations to an assembly file (s.x68). The data was added to the bottom of the program space in the code. Then the interrupt routine was modified so that the data was copied from each subsequent entry in the s() array instead of from the ADC location.

Once these modifications were complete, breakpoints were set in the code and were executed to view intermediate results. Ultimately, this helped in the code debugging process, as there were a few logical problems that needed to be resolved. Once the coding errors were corrected, the results could be observed. Initial results proved promising, in that the observed frequency matched the frequency identified in

MATLAB in all cases, excepting the rounding error in the final step. All the cases described in Section 5.1 were tested and the results matched in each case.

The Easy68K also has a tool that measures the number of clock cycles between instructions. This tool is used to determine if the timing is correct between interrupt requests. Ultimately, the goal is to ensure that the number of clock cycles is less than 8000 if a 1.0 kHz sampling frequency is used or 5000 if a 1.6 kHz sampling frequency is used. Initially, these tests proved that the code would not run in the time allotted because the average number of clock cycles between samplings was around 9000 +/- 10% clock cycles. Upon further investigation, the code appeared to be inefficient, so it was rewritten with more efficient uses of commands than before. This accounted for a reduction of clock cycles to 6000 +/- 10% with the same settings for K_{MAX} and N_{MAX}, a 50% increase in efficiency.

Ultimately, this simulation tool saved countless hours of debugging in the lab. Code revisions could be implemented and simulated quickly, without the need for using the emulator or hardware. This simulator also saved time in the sense that the code was simulated while the hardware prototype was still being constructed.

5.3 Hardware Debugging with the Deneb Emulator

After constructing the physical hardware as described in Section 4.1, it was necessary to test the hardware in two stages to ensure the prototype matched the schematics and board layout before moving on to the final design implementation. The first stage involved checking all the connections formed by tracks and wires. The second stage involved checking the various modules with the memory dump feature of the Deneb Emulator software and with software segments written in assembly.

5.3.1 Checking Physical Connections

In general, soldered connections can have a few types of failures. The first is lack of continuity (open circuit), the second is a short between neighboring pads/wires/pins, and the third is internal quasi-open circuits formed by capacitance, which can be formed by "cold" solder joints[5]. Although checks of suspicious connection were made during construction, there are inevitably additional errors for a large design.

The first and most tedious task is checking all the connections between devices to ensure all part are connected according to the schematic. Performing a continuity check with a digital multi-meter (DMM) with all the parts removed from their respective sockets checks for open circuits well. Checking for short circuits is a little more difficult because each pin of one device must be checked against every other pin on every other device to check each connection. Instead of checking every combination of pins, only neighboring pins were checked for shorts because most shorts come from wires and solder crossing pads. The final test involved measuring the capacitance between connections. The lines most susceptible to error because of too much capacitance are the data and address lines; hence these were the only lines check for cold solder joints outside of a purely visual inspection of all connections.

5.3.2 Memory

Memory is perhaps the easiest component to test outside of the manual tests described above because all one has to do is connect the emulator, then read and write to the memory dump window. Once the power was applied to the board and the emulator

[5] "cold" solder joints are formed when insufficient heat is applied to the physical components soldered.

configured, the memory did not work as expected. The dump window read the values corresponding to floating values on the data lines for both the even and odd address space. This meant that either the memory was bad or the MMU was not operating properly, the latter being the most likely cause. The memory select lines (SERAM' and SORAM') were not asserting when the corresponding addresses were accessed in the dump window. Further investigation revealed that address strobe (AS') was connected to V_{CC} (5V), thereby constantly disabling both decoders. Once the connection was restored, read/write functionality was possible.

This design relies on the non-volatility of the RAM, as there is no other way to program the prototype without adding additional hardware or purchasing additional components. The only tests of interest are to (1) see if the RAM will hold values with the noise present on the board and to (2) see if the NVRAM holds its values once power is cycled. Both tests had positive results.

5.3.3 Bar graph LEDs

Upon powering the board, the LEDs light up if values are present on the data lines. Testing the LEDs is a simple as testing the memory, with the exception that the data written to the register driving the LEDs will not appear in the dump window because the hardware design would be needlessly more complicated. The other difference is that all bits written must be inverted to light the appropriate LEDs. When this test was performed, one observed inconvenience was that after the data was written, the values on the LEDs changed from the input value, mainly because the emulator attempts to read many values to update the window.

It was necessary to write a small segment of code to test the LEDs. This code is attached in Appendix III. The test simply lit a single LED, and performed a one-bit logical shift after a short delay, and repeated indefinitely. This test revealed that the first and third LEDs were shorted, a test not revealed by checking neighbors for shorts. The short was fixed and the LEDs functioned properly thereafter.

5.3.4 A/D converter

The A/D converter testing required the addition of extra hardware as seen in Figure 5.1. The hardware consisted of a simple 10k potentiometer with the end terminals connected to V_{CC} (5V) and ground. The wiper terminal was connected to IN0 on the ADC0809. By varying the position of the wiper, the voltage between the wiper terminal and ground should vary linearly between 0 and 5V. Similarly, the reading from the A/D converter readings should respectively vary between 0 and 255_{dec}.

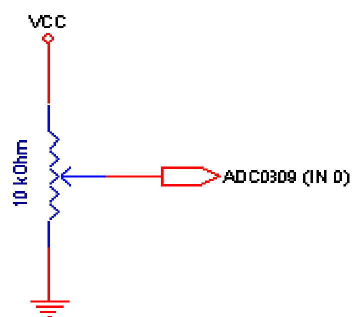


Figure 5.5: A/D converter test circuit.

This module is similar to the LEDs in that the value read from the memory dump window will not be the same as the one written. To initiate the A/D conversion, any value may be written to the device. When the emulator immediately reads the device after the write operation, the previous conversion results are displayed in the window. To perform the test, the wiper on the potentiometer is varied a number of times, the resulting voltages are measured with a DMM, and the equivalent decimal values are calculated for comparison. The A/D converter functions as expected with the potentiometer. After this test was completed, the additional hardware was removed.

5.3.5 Power-on reset circuitry, serial communications interface (SCI), and sampling frequency generator interrupt circuitry

By this point, the timing associated with the power-on reset circuit has been tested and the results are discussed in more detail in Section 3.1.12 and shown in Figure 3.16. However, it is necessary to test whether the reset switch and the power-on reset functionality cause the 68000 to reset as predicted. Any simple program should begin immediate execution once power is applied or when the reset pushbutton is depressed, so this test is coupled with some others.

An easy way to verify the interrupt and SCI circuitry is working properly is to send a small message via the SCI. This message must be small so as not to cause a backlog of interrupt service requests. It takes 10 bits to send a single byte to the terminal. At 38400 bps, a total of 3.84 bytes may be sent between interrupt requests if the interrupts occur every 1 ms. Once the 68000 comes out of reset, a message to clear the terminal screen is sent via the SCI. Then, the interrupts are enabled and a test message with the string "TST" should be sent to the terminal with each interrupt request. As expected, when power is supplied, the terminal screen is cleared and begins to fill with "TST". The same is true if the reset pushbutton is depressed at any time.

5.3.6 MIDI interface

The last major module to check is the MIDI interface. The easiest way to do this is to send properly formatted packets through the MIDI SCI. The messages selected are a program change to set the instrument to Grand Piano and a series of messages to play the C-major scale followed by a C-major I chord then a command to stop the notes. This routine is set to execute once and the code is attached in Appendix III.

Overall, this interface was the most difficult to debug in hardware because the MIDI OUT line is not connected to the MIDI IN line of the other device, but rather the lines are connected to their respective names on the opposite devices. This is different from connecting the SCI communicating with a PC terminal. Nonetheless, the MIDI interface functionality was successfully verified.

5.3.7 Proven code

The last test, outside of testing the code that was developed and simulated, was to test the hardware with a large segment of proven assembly code. Loading and executing the Game of Snake performed this test. This particular game was written in 2004 as part of the Microcomputer Design course (ECE516) taught at the University of Louisville and its functionality has long since been proven to run on a similar hardware design. The code for this game was modified to run in a different addressing scheme and to update the LEDs to reflect the player's score in binary, which is also displayed on the terminal screen. After completing the modifications, the code ran perfectly on the emulator, which suggested that the prototype was ready to be tested in stand-alone mode, as discussed in the next section.

5.3.8 Butterworth Filter

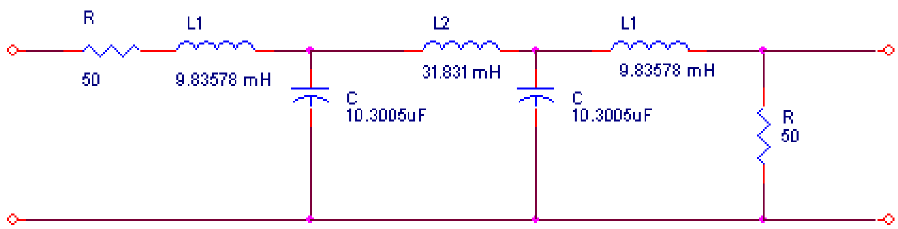


Figure 5.6: Fifth order Butterworth low pass filter design.

The fifth-order Butterworth low pass filter shown in Figure 5.6 is checked graphically measuring by the filter's amplitude response in a simulation and physically in the lab using a function generator with linear sweep and an oscilloscope with XY interpretation. The red curve in Figure 5.7 shows the results of a WinSpice simulation with the ideal values input from the website. This is not normalized, therefore the 3.01 dB point occurs at $\sqrt{2}/4$ instead of the traditional $\sqrt{2}/2$. From this waveform, it is easy to see that the filter reaches the appropriate amplitude at the desired cutoff frequency (500 Hz), and remains relatively flat until 350Hz. The source code to generate the plots can be found in Appendix II.

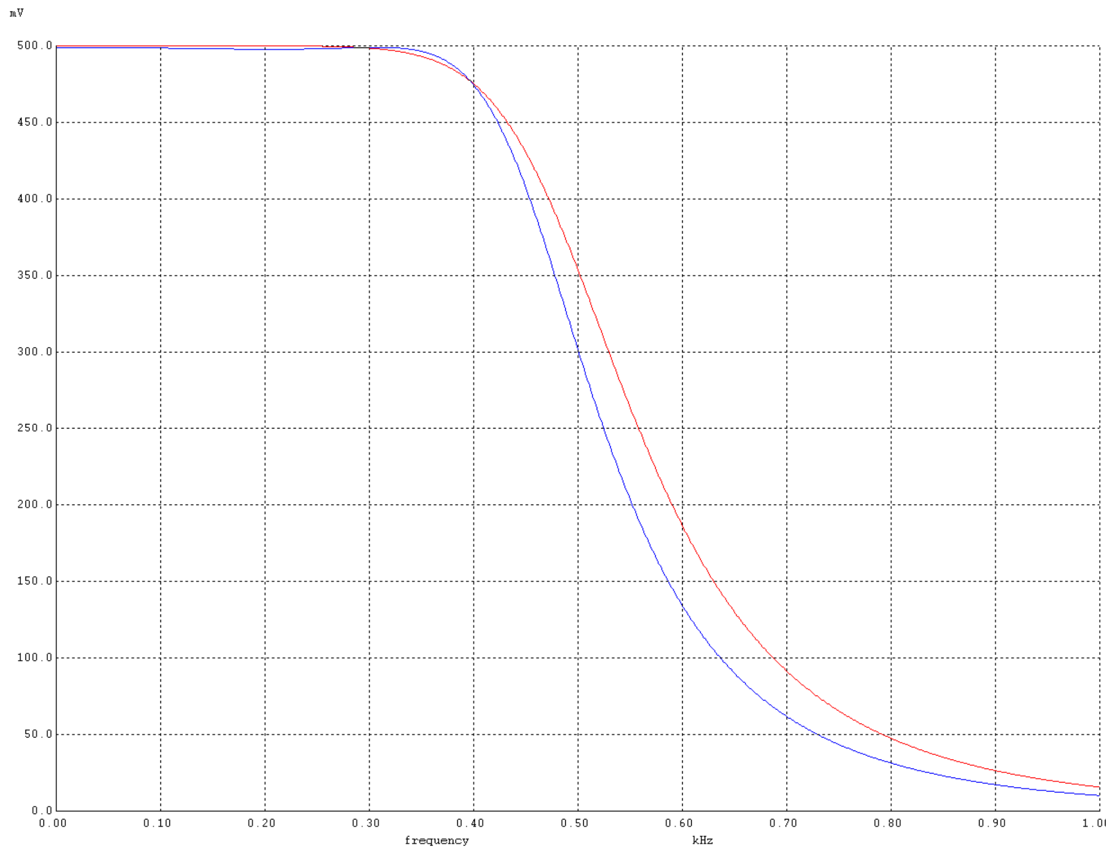


Figure 5.7: Butterworth filter frequency responses for ideal (red, gently sloped) and actual (blue, steeply sloped) component values.

The next step to test the Butterworth filter is to physically assemble the circuit and to measure the frequency response. The exact component values were not all available in the lab, so the closest approximations were used. The actual component values are R = 51.6Ω, L_1 = 11.29mH, C_1 = 11.45μF, L_2 = 31.39mH, C_2 = 11.45μF, and L_3 = 11.35mH. These values were simulated again using WinSpice and the anticipated results are shown in blue in Figure 5.7. From the results, it is anticipated that the cutoff frequency will be about 475Hz instead of 500Hz.

The filter is then constructed on the prototyping board and is tested with a function generator capable of a linear frequency sweep. The sweep is varied from 1 to 1000Hz and the results viewed in XY mode on the oscilloscope, which is shown in

Figure 5.8. In this figure, the frequency drops to the 3.01dB point at about 360 Hz, which is not consistent with the simulations.

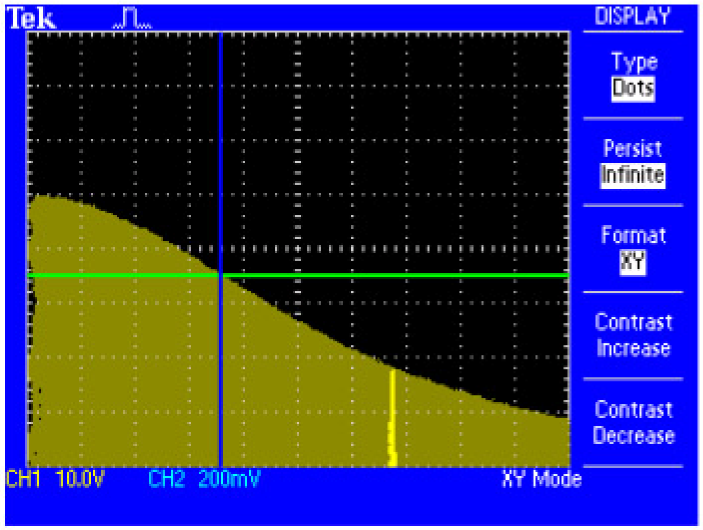


Figure 5.8: Actual Butterworth filter frequency response with 100Hz/division horizontally and 200 mV/division vertically. The intersecting green and blue lines represent the 3.01dB point.

The component values were measured again on the board itself and they matched the values reported above, except that the inductor values seem to vary by +/- 50% while connected to the bridge (Stanford Research Systems Model SR720 LCR Meter). A different bridge (GenRad 1657 RLC DIGIMETER) was used and the component values all matched except the 30mH inductor, which consistently read as 15mH when measured at 1.0 kHz. Also, an inductor can be modeled as an inductor in series with a resistor, these resistances were measured along with the inductances on the latter bridge. The resistance associated with L_1 is 6.72Ω and the resistance associated with L_2 is 94.5Ω. Another simulation was constructed which included the updated values for the resistances from the inductors to verify the simulation matched the results. Figure 5.9 shows the results of another WinSpice simulation with superimposed plots depicting the ideal values and the set with the measured inductances and resistances across the inductors.

The incorrect inductance on L_2 with the addition of the large resistance to L_2 caused the filter to perform badly.

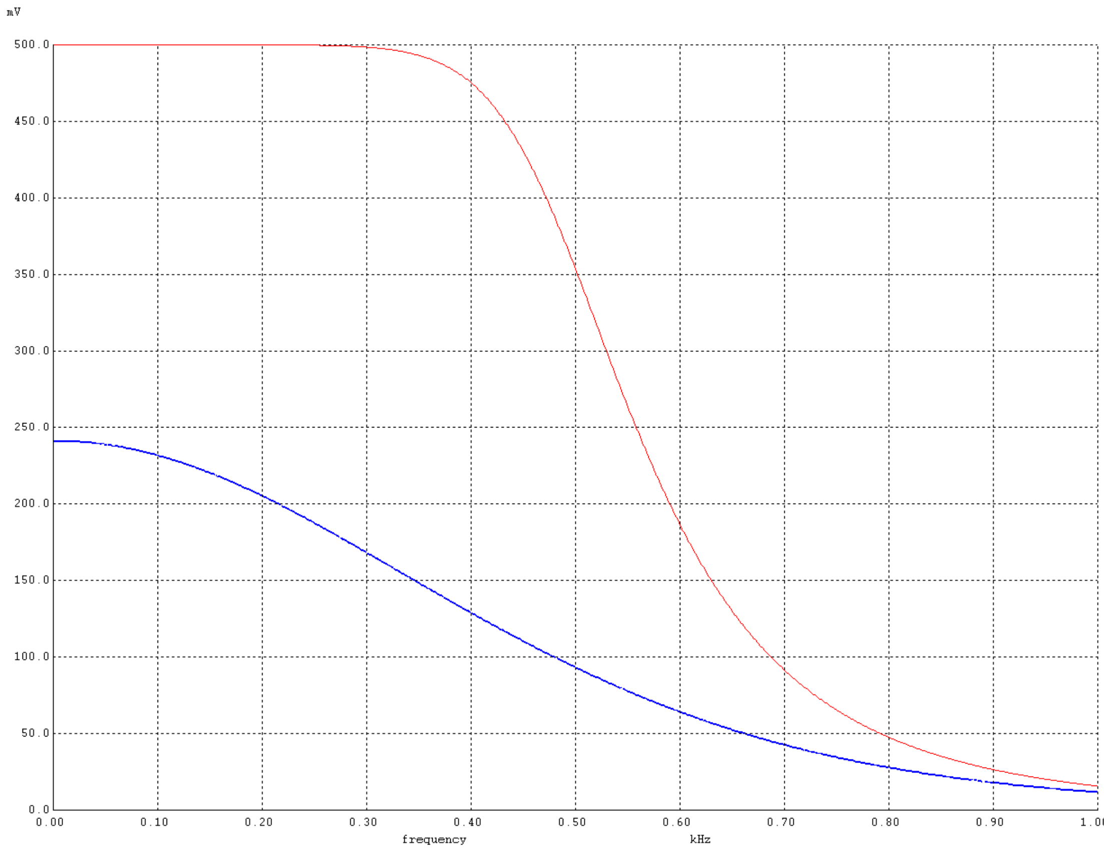


Figure 5.9: Simulated Butterworth frequency response (blue) with updated L-values accounting for resistances. The (red) curve with the overall higher magnitude is the response with the ideal values.

To correct this error, the 30mH inductor was replaced with three of the 10mH inductors connected in series, due to the low resistance of the 10mH inductors and availability of the components in the lab. This reduced the overall resistance to approximately 20Ω. Another simulation and frequency sweep was performed to compare the actual performance of the filter to the simulation. The results of the simulation (Figure 5.10) show that that filter's performance is reduced in both amplitude and cutoff frequency (360Hz).

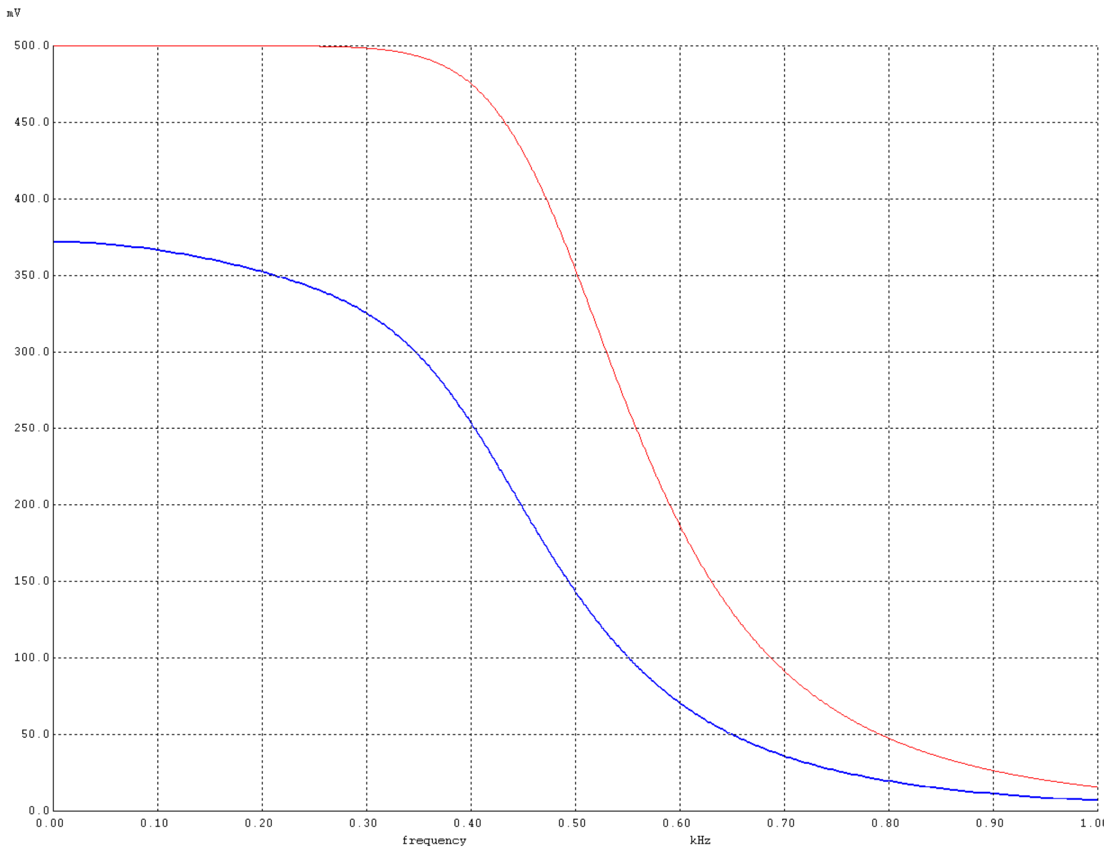


Figure 5.10: Simulated Butterworth frequency response with new L-values (blue). The (red) curve with the overall higher magnitude is the response with the ideal values.

The circuit was tested on the prototyping board with the function generator set to for a linear sweep as in the previous test. The results of this test are displayed in Figure 5.11. The results of this test are similar to the simulation, in that the cutoff frequency is the same as the results from the simulation, although it is less than the desired cutoff frequency.

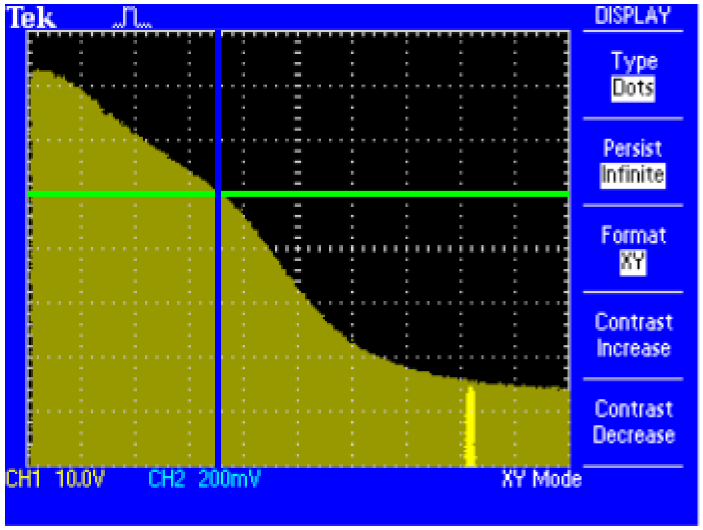


Figure 5.11: Actual Butterworth frequency response using the new L-values with 100Hz/division horizontally and 200 mV/division vertically. The intersecting green and blue lines represent the 3.01dB point.

5.4 Stand-alone Problems

The ultimate goal of this project is to develop a stand-alone prototype that can run independently of an emulator. From the tests and results in Section 5.3, it is clearly obvious that all the connections and hardware elements work properly with the emulator. The next logical step is to replace the emulator probe on the prototype with the actual microprocessor and monitor the results. Unfortunately, the prototype does not function as expected in stand-alone mode. Therefore, there has to be some hidden flaw in the hardware design or construction that is preventing the 68000 from executing code when it is placed in a socket and set to run independently. This section describes the efforts to resolve this problem.

5.4.1 68000 Controls and Inputs

Usually when the emulator is working and the actual processor is not, the problem is that one or more of the unused input lines to the processor are floating. Each of these lines (BR’, BERR’, BGACK’, IPL1’ and IPL2’) was checked with and without the hardware running to see if the lines were floating or connected improperly. Unfortunately, the problem could not be found in these connections. The connections to V_{CC}, ground (V_{SS}), HALT’, RESET’, and the system clock were also verified to be working.

5.4.2 Other Tests

Testing the load of the various devices was next in the line of logical reasoning. If a device is sourcing or sinking too much current, the device can malfunction in unpredictable ways. The easiest way to test for this on the prototype was to systematically remove parts and attempt to run test code on the emulator first, then on the stand-alone system. However, the MMU and one external interface had to remain in order to know if the system was executing the code. The best interface to leave in place was the MIDI interface because it requires the fewest components to operate. With that in mind, the A/D converter, amplifier, LEDs, data register, SCI ACIA, voltage level shifter, bit rate generator, and sampling frequency generators were removed from the system. Then the MIDI interface test code described was executed to test the functionality. The code ran fine with the emulator connected to the prototype but did not execute with the 68000. The MIDI interface was removed and the LEDs and data register were replaced to attempt another test. This test used the LED test code as described in Section 5.3.3. The results did not change. Finally, the LEDs and data register were removed and the SCI ACIA, voltage level shifter and bit rate generator were replaced. A simple “Hello World” program was executed and the results were the same as before.

5.4.3 MMU Connections

The next test was to check the various modules for functionality. The first line checked was DTACK' because DTACK' is formed by ANDing the memory select lines. Each time an address is accessed, DTACK' should be asserted. Testing revealed that this was not the case. However, it was proven that the MMU and DTACK' circuitry functioned properly in the previous tests, therefore the problem had to be with the MMU enable lines (AS', UDS', LDS', IACK'/MRST') or data input lines (AB14 - AB16).

To check these lines, DTACK' was forced low so as not to have the 68000 delay instruction execution while waiting for the assertion of DTACK'. For simple memory accesses, asserting DTACK' should pose no problem because the memory operates faster than the 68000. Probing AS', UDS' and LDS' revealed something interesting, in that each of these signals on the oscilloscope appeared to be functioning as a clock signal, or pulse train. Normally, the variety of commands associated with sending/receiving data cause delays in the assertion/negation of these lines, which never cause these lines to appear as a pulse train. However, it was easy to verify that these signals were reaching the MMU by probing the base of the MMU and the top of the MMU.

This meant that the address lines had to be causing the problem. Probing the tops of the pins on the 68000 revealed that the address lines were working properly, but the signals were not reaching the MMU. This was puzzling because the emulator worked fine and the continuity of the connections was verified earlier. On a hunch, the base of the sockets housing the 68000 was probed. This revealed that MMU data input lines were floating, meaning there had to be a mechanical connection problem. This is not too surprising because the 68000's pin diameters are smaller than the pin diameters on the emulator pod probe. When inserted into the zero insertion force (ZIF) socket, the emulator connections are more secure. The ZIF socket itself sits in a machined socket, so it is no problem to remove the ZIF socket and insert the 68000 into the machined socket when ready to execute the code stand-alone. Additionally, all other mechanical connections were examined for inconsistencies, but no others were identified.

Once this problem was identified, the proven code (The Game of Snake) was loaded and executed again. This time the code executed up to a certain point with the 68000 in place, but the 68000 locked up. This problem occurred at the same place in the code every time, and the error was independent of the duration of code execution before the error occurred. This suggested there was a problem in the memory space, although that would be unlikely because the memory works fine with the emulator. The next step was to repeat the tests from Section 5.4.2 before proceeding.

5.4.4 Checking the Memory Space

The results from performing the tests in Section 5.4.2 again were different from the first time around. In this instance, there were mixed results for the smaller code segments. In some segments, such as the LED test, the code executed correctly. In other tests, such as the interrupt request test and the MIDI interface test, the code would not execute correctly. When the proven Game of Snake code was executed, the problems were the same as before.

Essentially, these results made no sense. First, the failures seemed to be independent of memory location, which would suggest a problem with the address line or the memory itself. Second, the failures seemed to be independent of the device operated. For example, in the smaller test segments, the SCI ACIA would not output a “Hello world" message to the PC terminal, but the ACIA handled I/O correctly for the Game of Snake, which makes extensive use of the ACIA.

As stated in the previous section, the proven code always locked up in the same location in the game, although the address present on the 68000 was neither in the program nor the data space for this design. However, a test was performed to see if the error would follow the code or stay fixed to the address region where the error was occurring. Without knowing the exact location of the address, the test was performed by a number of no-op (NOP) statements at the beginning of the code. This essentially had the effect of pushing the remaining code down by two bytes for each NOP statement. Up to 20 consecutive NOP statements were added to perform this test, but the error followed the code each time. This once again made no sense because the code has been proven to work on a previous design as well as with the emulator connected to the prototype. Aside from these difficulties, this consistently reproducible error ceased to be consistent, and the code executed intermittently.

5.4.5 Testing Conclusions

These problems were analyzed, researched, and tested for months with no measurable success. The inability to consistently reproduce the errors has made debugging impossible without the ability to use the emulator. Ideally, a 32-bit logic analyzer could help in determining where the 68000 fails to execute code, but since the errors are independent of memory addresses and device, this expensive tool would prove useless. The problem itself likely comes from a problem with the soldering, noise imposed by the 8 MHz clock, and/or from a slightly excessive current draw. All of these potential problems have been tested as described in the previous sections, but no conclusive results have been obtained. Therefore, the final design will have to run on the emulator as the cost associated with trying to fix this error far outweighs any benefit for this project.

CHAPTER VI – CONCLUSIONS AND FUTURE WORK

The AMDF time-domain analysis technique was chosen to aid in real-time pitch-tracking for a voice operated musical instrument operated by the MIDI protocol. The AMDF algorithm was implemented in assembly to run on an embedded system utilizing the 68000, which was constructed on a soldered prototype board. This design consisted of four main hardware modules responsible for controlling sampling, serial communications, MIDI communications and memory management. Although the modules worked correctly individually and together with an emulator, they failed to function on a stand-alone prototype due to an unidentifiable hardware flaw.

However, the simulations and physical implementation clearly revealed that the AMDF by without additional refinements itself is insufficient to function accurately as a real-time pitch-tracking device with such a small sampling rate. Additional software refinements have been proven to enhance the functionality of the AMDF for this application, but were not implemented due to limitations imposed by the system clock. Thus, equipment limitations leading to hardware design constraints were a factor for reduced performance of this device, as well as a limiting factor for incorporating additional refinements to support the AMDF, as implemented by [Shimamura 2001 and Kim 1998].

Further work should be done to find a suitable minimum acceptable system clock capable of supporting a modified AMDF or another successful pitch detection algorithm running on an embedded system. A study similar to [Rabiner 1976] updated to include more frequency and time domain techniques including a variety of the updated FFT algorithms and zero-crossing techniques would be extremely beneficial in this field, especially if the study were a comparison of real-time performance on modern embedded hardware.

This prototype was designed to perform all signal calculations itself. Another obvious area of future work would be to implement this design using an external DSP device or an external programmable logic device using VHDL or ABEL. These design practices are implemented in industry especially when dealing with RF systems, where generic or specially designed ICs are used to modulate, demodulate, or process these high-frequency signals. Either of these concepts would alleviate computational-related performance problems on the main processor. However, these devices would likely require a system clock faster than the 8 MHz clock in this design. Either implementation would lead to a significantly more successful design of an embedded real-time voice operated musical instrument.

Ultimately, the success of a device of this nature rests with a faster system clock and a faster processor or microcontroller. In conjunction with a faster system, using an external device to handle the signal processing aspect of this application would lead to a more successful prototype and marketable product.

REFERENCES

ANSI (1997). ANSI S1.1-1994: *American national standard acoustical terminology*. Acoustical Society of America, New York, 1994.

Budak, A. (1991). *Passive and Active Network Analysis and Synthesis.* Prospect Heights, Illinois: Waveland Press, Inc. Copyright 1974. Republished 1991.

Cleaver, T.G. (2000). *Voice Operated Musical Instrument*. MEng/MS Thesis Opportunity. January 10, 2000.

Cleaver, T.G. (2004). *Needs Assessment.* 2004. Accessed 27 Nov. 2007. <http://raise.spd.uofl.edu/systom/ece599/NeedsAssessment.pps>

de Cheveigné, A. and Kawahara, H. (1998). Multiple period estimation and pitch perception model. *Speech Communications* Vol. 27, pp. 175-185. Elsevier Science B.V. July 8, 1998.

de Cheveigné, A. and Kawahara, H. (2002). YIN, a fundamental frequency estimator for speech and music. *Journal of the Acoustical Society of America*, Vol. 111, No. 4, pp. 1917-1930, April 2002.

de la Cuadra, P., Master, A., Sapp, C. (2001). Efficient Pitch Detection Techniques for Interactive Music. *Proceedings of the 2001 International Computer Music Conference*, La Habana, Cuba, September 2001.

"diplophonic." Dictionary.com Unabridged (v 1.1). Random House, Inc. Accessed 25 Jul. 2007. <Dictionary.com http://dictionary.reference.com/browse/diplophonic>.

"Fast Fourier Transform." [FFT]. Wikipedia. 2007. Wikimedia Foundation, Inc. Accessed 14 July 2007. <http://en.wikipedia.org/wiki/Fast_Fourier_transform>.

"formant." The American Heritage® Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004. Accessed 27 Nov. 2007. <Dictionary.com http://dictionary.reference.com/browse/formant>.

"glottis." Merriam-Webster's Medical Dictionary. Merriam-Webster, Inc. 25 Jul. 2007. <Dictionary.com http://dictionary.reference.com/browse/glottis>.

Lago, N.P and Kon, F. (2004). The Quest for Low Latency. *Proceedings of the International Computer Music Conference*, 2004. pp. 33-36.

Middleton, G. *Pitch Detection Algorithms*. 17 Dec. 2003. Rice University. 27 January 2006. <http://cnx.rice.edu/content/m11714/latest/>.

MIDI Manufacturers Association (MMA). *Making Music with MIDI*. Copyright 1985-2007. Accessed 10 Jan. 2007. <http://www.midi.org/about-midi/aboutmidi3.shtml>.

MIDI Manufacturers Association (MMA). *MIDI Electrical Specification Diagram & Proper Design of Joystick/MIDI Adapters.* Copyright 1985. Accessed 9 July 2007. <http://www.midi.org/about-midi/electrical.shtml>.

MIDI Manufacturers Association (MMA). *Table 1 of 3 – Summary of MIDI Messages.* Updated 1995. Accessed 9 July 2007. <http://www.midi.org/about-midi/table1.shtml>.

MIDI Manufacturers Association (MMA). *Table 2 of 3 – Expanded Messages List (Status Bytes).* Updated 1995. Accessed 9 July 2007. <http://www.midi.org/about-midi/table2.shtml>.

MIDI Manufacturers Association (MMA). *Tutorial on MIDI and Music Synthesis*. Revised 2001. Accessed 9 July 2007. <http://www.midi.org/about-midi/tutorial/tutor.shtml>.

Motorola Inc. (1993). *M68000 8-/16-/32-Bit Microprocessors User's Manual*. Ninth Edition. Copyright 1993.

Oppenheim, A.V. and Schafer, R.W. (1999). *Discrete-Time Signal Processing.* Saddle River, New Jersey: Prentice-Hall Inc. Copyright 1999.

"psychoacoustics." Merriam-Webster's Medical Dictionary. Merriam-Webster, Inc. Accessed 25 Jul. 2007. <Dictionary.com http://dictionary.reference.com/browse/psychoacoustics>.

Rabiner, L.A., Cheng, M.J., Rosenberg, A.E., & McGonegal, C.A. (1976). A Comparative Performance Study of SeveralPitch Detection Algorithms. *IEEE Transactions on Acoustics, Speech and Signal Processing*, Vol. ASSP-24, No. 5, October 1976, pp. 399-418.

Ryynänen, M. (2004). "Probabilistic Modeling of Note Events in the Transcription of Monophonic Melodies". MS thesis. Tampere University of Technology, Department of Information technology, February 11, 2004.

Saul, L.K., Lee D. D., Isbell. C.L., & LeCun, Y. (2002). Real time voice processing with audiovisual feedback: toward autonomous agents with perfect pitch. *Advances in Neural Information Processing Systems 15 (NIPS) Conference Paper*, December 9, 2002, pp 1205-1212.

Semenov, O. *PC-MIDI interface adapter (MIDI Cable) schematic and pinout.* 25 June 2005. Accessed 5 July 2007. <http://www.pinouts.ru>.

Shimamura, T. & Kobayashi, H. (2001). Weighted Autocorrelation for Pitch Extraction of Noisy Speech. *IEEE Transactions on Speech and Audio Processing*, Vol. 9, No. 7, October 2001, pp. 727-730.

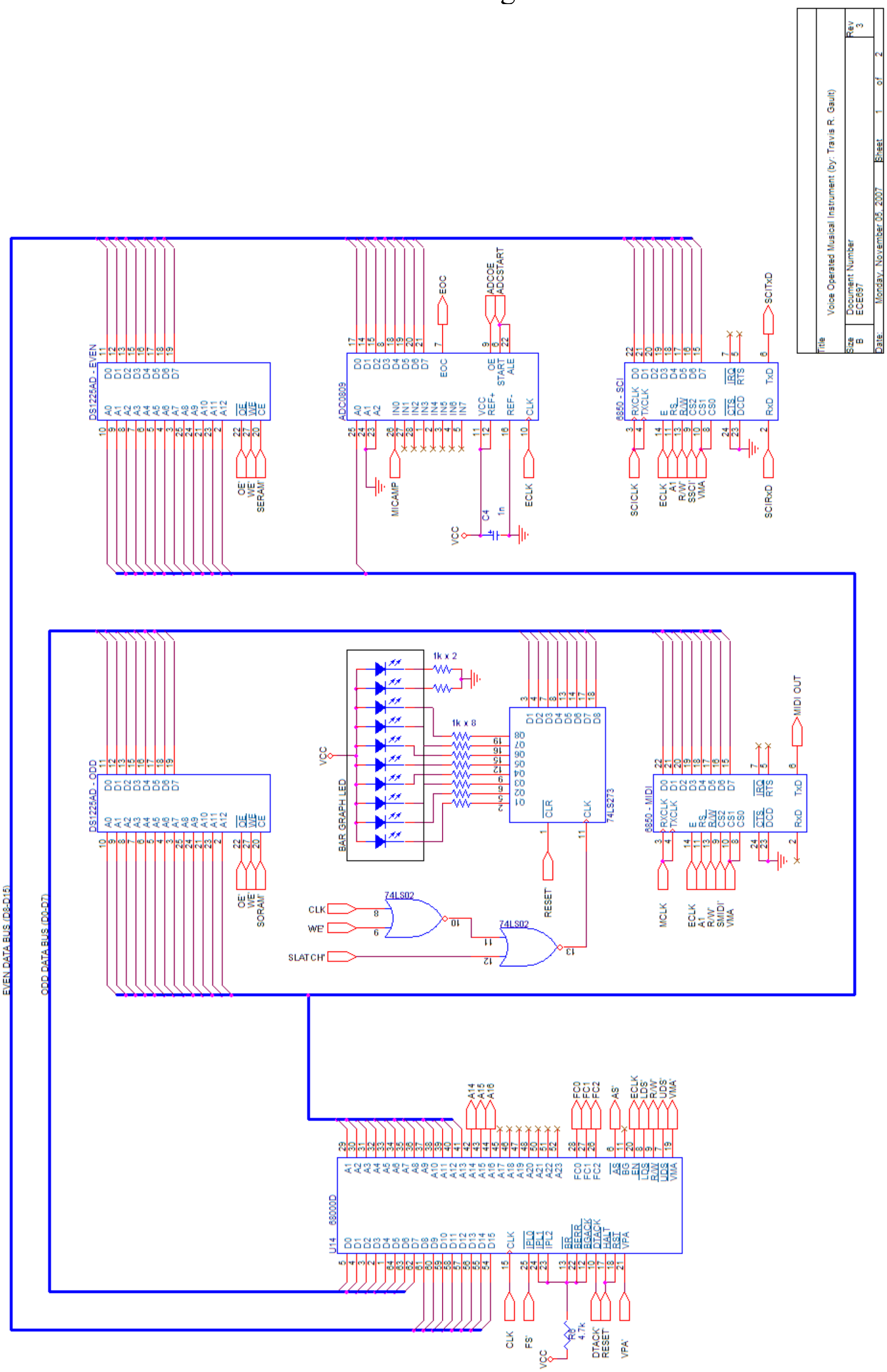
Stewart, L. "Listening Displeasure". *BBC News Magazine*. 30 Jan. 2006. Accessed 3 July 2007. <http://news.bbc.co.uk/1/hi/magazine/4655352.stm>.

Yale University Music Library. *Vocal Ranges.* March 21, 2005. Accessed 15 Oct. 2006. <http://www.library.yale.edu/cataloging/music/vocalrg.htm>.

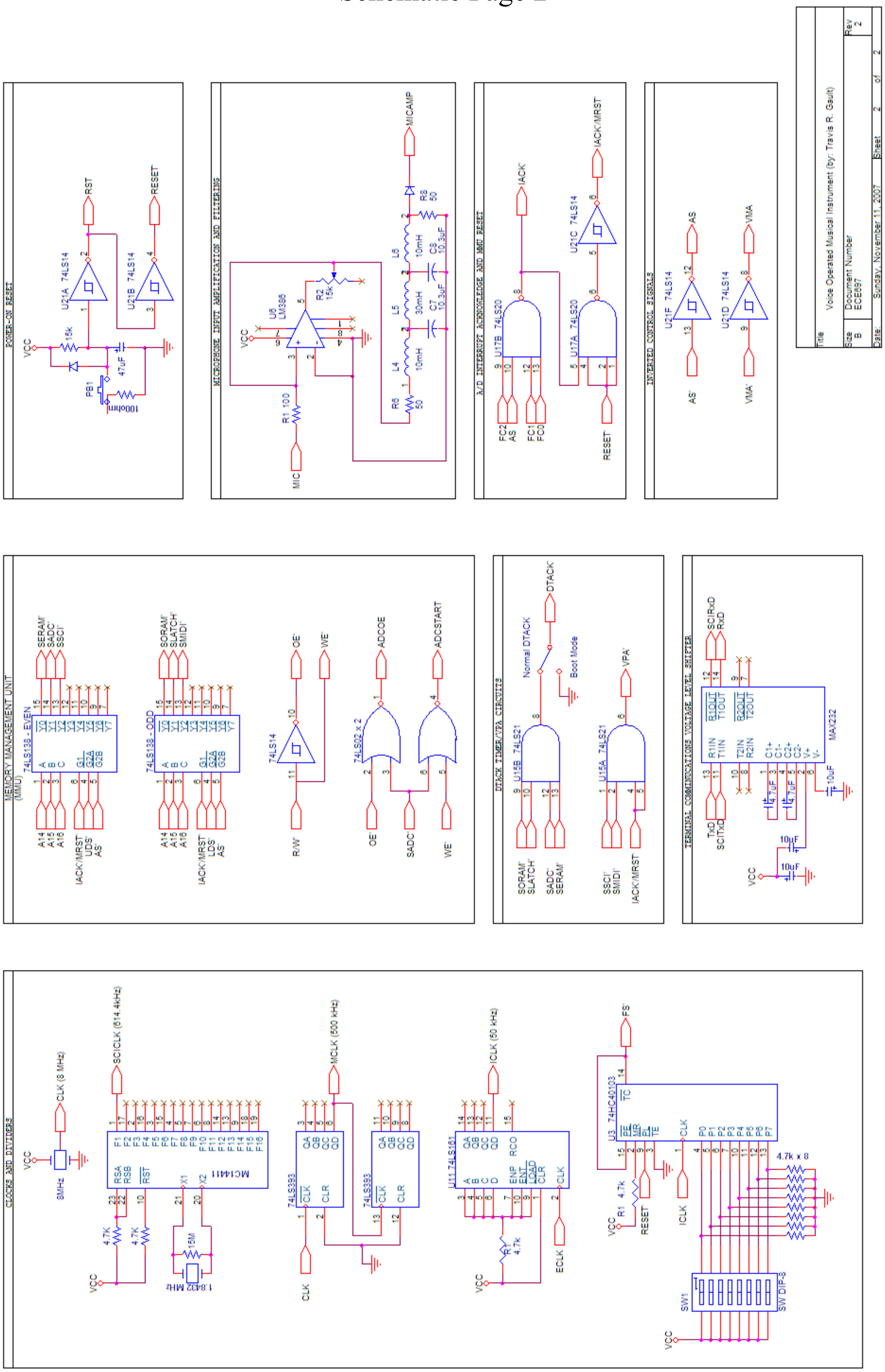
Weeks, M. (2007). *Digital Signal Processing Using MATLAB and Wavelets*. Hingham, Massachusetts: Infinity Science Press. Copyright 2007.

APPENDIX I – HARDWARE DESIGN

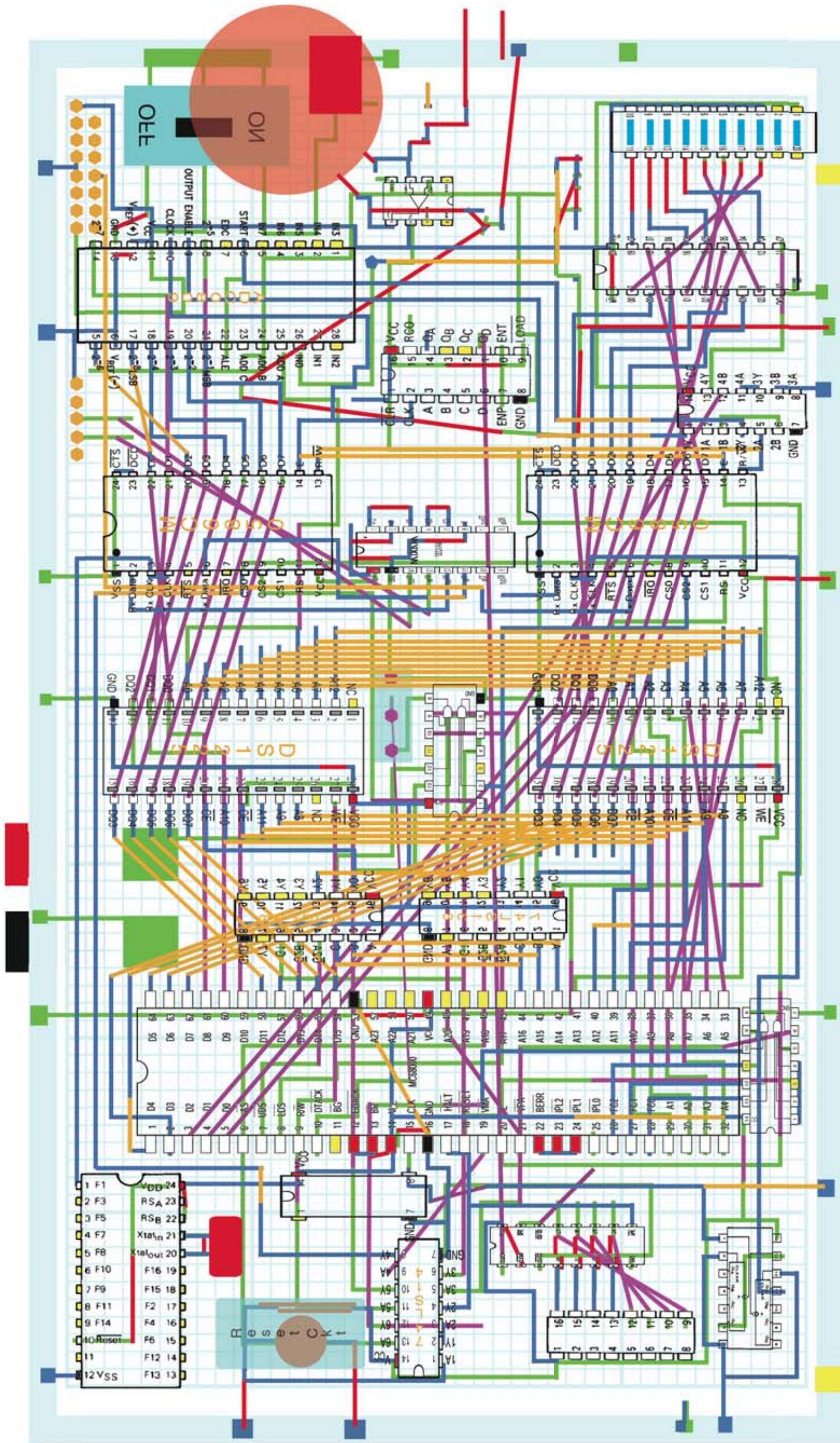
Schematic Page 1



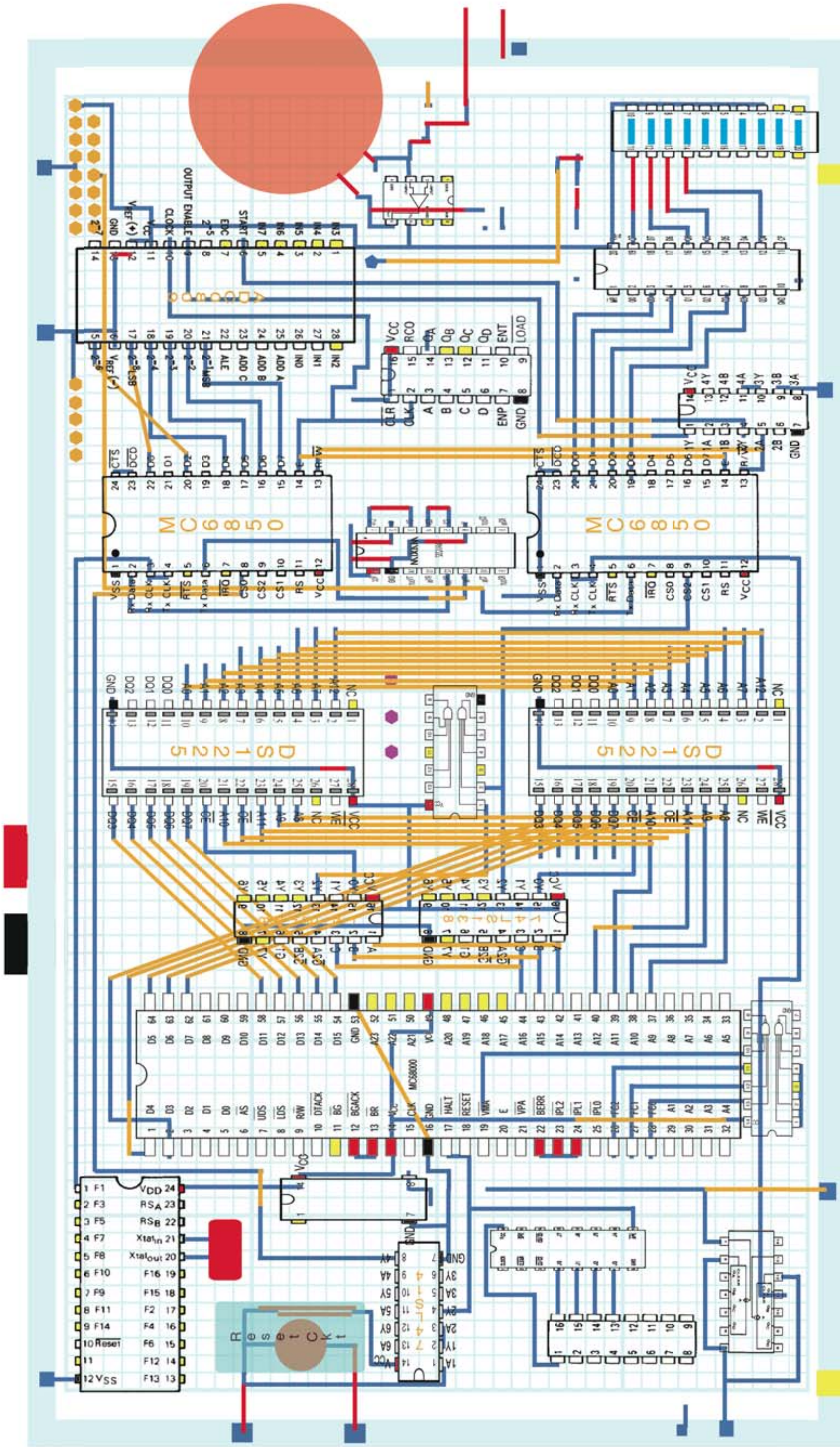
Schematic Page 2



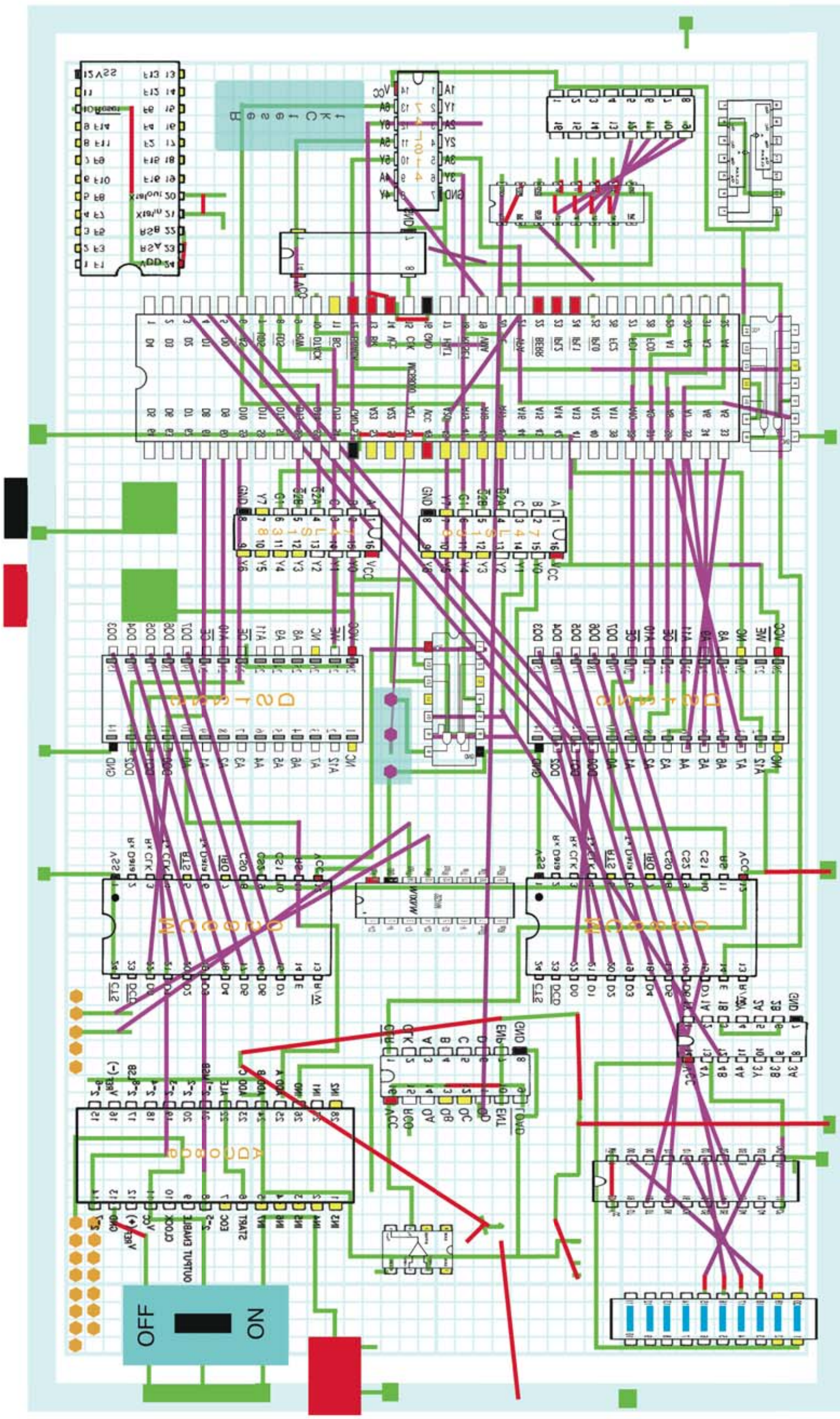
Board Layout – Top View of All Components



Board Layout – Top View of Top Components



Board Layout – Bottom View of Bottom Components



APPENDIX II – SIMULATION SOURCE CODE

Butterworth Filter Spice Simulation File for Ideal Component Values (BWLPF.spc)

```
Butterworth Filter Simulation by Travis R. Gault

.control       ; start control statements
destroy all         ; WinSpice erases all previously stored data and starts fresh.
op             ; perform an operating point analysis
tran 1e-6 2E-3 0 1E-6    ; perform a transient (time-domain) analysis
AC LIN 1000 1 1K    ; perform an ac (small-signal) frequency analysis
.endc               ; start control statements

R1   1    2    50
L1   2    3    9.83578m
C1   3    0    10.3005u
L2   3    4    31.831m
C2   4    0    10.3005u
L3   4    5    9.83578m
R2   5    0    50

Vin  1    0    SIN(0 5 1K) AC 1 DC 0

.TRAN          1u   2m

.END
```

Butterworth Filter Spice Simulation File for Actual Component Values (BWLPF-Actual Values.spc)

```
Butterworth Filter Simulation by Travis R. Gault

.control       ; start control statements
destroy all         ; WinSpice erases all previously stored data and starts fresh.
op             ; perform an operating point analysis
tran 1e-6 2E-3 0 1E-6    ; perform a transient (time-domain) analysis
AC LIN 1000 1 1K    ; perform an ac (small-signal) frequency analysis
.endc               ; start control statements

R1   1    2    51.7
L1   2    3    11.29m
C1   3    0    8.75u
L2   3    4    30.9m
C2   4    0    8.75u
L3   4    5    11.35m
R2   5    0    51.5

Vin  1    0    SIN(0 5 1K) AC 1 DC 0

.TRAN          1u   2m

.END
```

Butterworth Filter Spice Simulation File for Actual Component Values with Updated Inductance and Resistance Values (BWLPF-Actual Values with LR.spc)

```
Butterworth Filter Simulation by Travis R. Gault
.control       ; start control statements
```

```
destroy all         ; WinSpice erases all previously stored data and starts fresh.
op             ; perform an operating point analysis
tran 1e-6 2E-3 0 1E-6    ; perform a transient (time-domain) analysis
AC LIN 1000 1 1K    ; perform an ac (small-signal) frequency analysis
.endc               ; start control statements

R1   1    2    51.7
L1   2    3    11.29m
R2      3       4       6.71
C1   4    0    11.45u
L2   4    5    15.3m
R3      5 6    95.46
C2   6    0    11.45u
L3   6    7    11.35m
R4   7    8    6.74
R5   8    0    51.5

Vin  1    0    SIN(0 5 1K) AC 1 DC 0

.TRAN          1u   2m
.END
```

Butterworth Filter Spice Simulation File for Actual Component Values with New Inductance and Resistance Values (BWLPF-Actual Values with new LR.spc)

```
Butterworth Filter Simulation by Travis R. Gault
.control       ; start control statements
destroy all         ; WinSpice erases all previously stored data and starts fresh.
op             ; perform an operating point analysis
tran 1e-6 2E-3 0 1E-6    ; perform a transient (time-domain) analysis
AC LIN 1000 1 1K    ; perform an ac (small-signal) frequency analysis
.endc               ; start control statements

R1   1    2    51.7
L1   2    3    11.29m
R2      3       4       6.71
C1   4    0    11.45u
L2   4    5    39.3m
R3      5 6    21.78
C2   6    0    11.45u
L3   6    7    11.35m
R4   7    8    6.74
R5   8    0    51.5

Vin  1    0    SIN(0 5 1K) AC 1 DC 0

.TRAN          1u   2m
.END
```

AMDF MATLAB Simulation File (amdfTests.m)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% AMDF simulation
% By: Travis R. Gault
% University of Louisville
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
cd h:
clear all
close
clc

% Load wav file and downsample to 1000Hz
zNote='f.wav'
downRate = 5;
[wavSample fs nbits]=wavread(zNote);
fs=fs/downRate;

NMAX=96
KMAX=48
s=wavSample(KMAX*2:downRate:end);

% 1/2 wave rectification
s=floor(255*(s.*(s>0)));

% Perform the AMDF calculation
for n=1:NMAX
    x=0;
    for k=1:KMAX
        x=x+abs(s(k)-s(k+n));
    end
    d(n)=x;
end
dd=d;

% Find all entries below the 25% threshold
threshold=max(dd)/2;
d1=dd<floor(threshold);
ff=find(d1);

% Calculate the differences between low points
%  and find the average frequency
fdiff=ff(2:end)-ff(1:end-1);
fdiff=fdiff(2:end)
fmean=fs/mean(fdiff)

d1=d1.*max(dd);

zTitle=['Sampling ',zNote,' at ',sprintf('%g Hz, results: T=%g samples, F=%g
Hz',fs,mean(fdiff),fmean)];

subplot(2,1,1), plot(s(1:NMAX)), axis tight, title('1/2 Wave Rectification')
subplot(2,1,2)
   plot(1:length(dd),dd,'b',1:length(d1),linspace(threshold,threshold,length(d1)),'g')
   axis tight, title(zTitle);
```

APPENDIX III – MODULE TEST CODE

LEDTest.x68

```
**********************************************************************
*; DESCRIPTION:  This test code is designed to test the LED bar     ;*
*;               graph display on the board.  A single LED is lit   ;*
*;               and is shifted after a short delay to the LED in   ;*
*;               the next highest bit position, continuing forever. ;*
**********************************************************************
LATCH       EQU         $4001

            ORG         0
            DC.L        $4000,$400

            ORG         $400
START       MOVE.B      #$FF,LATCH  ;Turn all LEDs off then wait
            BSR         DELAY

            MOVE.B      #$FE,D0     ;Turn on the lsb of the LEDs, wait
            MOVE.B      D0,LATCH    ;D0 is used to shift the LEDs
            BSR         DELAY       ; rightward

SHIFTGO     LSL.B       #1,D0       ;Shift the LED and wait
            MOVE.B      D0,LATCH
            BSR         DELAY
            BRA         SHIFTGO     ;Continue forever


**********************************************************************
DELAY       MOVE.L      #15000,D5
**********************************************************************
SUBD5       SUBQ.L      #1,D5
            BNE         SUBD5
            RTS
**********************************************************************

            END         $4000
```

```
******************************************************************************
*; DESCRIPTION:  This test code is designed to test the MIDI interface by   ;*
*;               sending MIDI packets to play the C major scale and chord.  ;*
******************************************************************************

******************************************************************************
*;------------------------------ Memory Map --------------------------------;*
******************************************************************************
STATUS      EQU         $1000
NOTENBR     EQU         $1001
NOTEVEL     EQU         $1002

PROGNUM     EQU         $1001

MSGEND      EQU         $1003
MIDIINST    EQU         $1001
PCEND       EQU         $1002
PREVMIDI    EQU         $1006
MIDINOTE    EQU         $1007

ENDRAM      EQU         $4000

MIDISC      EQU         $8001
MIDITXD     EQU         $8003
MIDIRXD     EQU         $8003

******************************************************************************

******************************************************************************
*;-------------------------------- Flags -----------------------------------;*
******************************************************************************
RDRF        EQU         0           ;Receive data register full (ACIA)
TDRE        EQU         1           ;Transmit data register empty (ACIA)

******************************************************************************
*;---------------------------- Defined Values ------------------------------;*
******************************************************************************
SCICFG      EQU         $15          ;SCI set-up: 38400,8,N,1
MIDICFG     EQU         $15          ;MIDI setup; 31.25kHz,8,N,1

GRANDPIANO  EQU         0
CHURCHORGAN EQU         19
ORCHSTRINGS EQU         46

SPECCHAR    EQU         171
NOTEERR     EQU         $FF

NOTEON      EQU         $90         ;Note on channel 0
NOTEOFF     EQU         $80         ;Note off channel 0
PROGCHANGE  EQU         $C0         ;Program change on channel 0
VELOCITY    EQU         $7F         ;Max note velocity
******************************************************************************

            ORG         $00
            DC.L        ENDRAM,START

            ORG         $400
START       MOVE.B      #1,PREVMIDI

            MOVE.B      #CHURCHORGAN,MIDIINST ;Select instrument

            BSR         MIDICONFIG  ;Configure MIDI ACIA

            MOVEA.W     #SCALE,A5   ;This hex string is the C Major scale

THERE       BSR         DELAY         ;Wait a short time before sending
            MOVE.B      (A5)+,MIDINOTE ; another note.
            BEQ         SNDCHORD    ;If the end of string flag (00) is seen
```

```
            BSR         SENDMIDI    ; then send the I chord
            BRA         THERE       ;Else, send the next note in the scale

SNDCHORD    MOVEA.W     #CHORD,A5        ;This hex string is the C major chord
            MOVE.B      #NOTEON,STATUS
            MOVE.W      #STATUS,MIDISTR
CNOTES      MOVE.B      (A5)+,NOTENBR    ;Send each not in the chord, no delay
            BEQ         CHORDHOLD        ;Wait a while after all note are send

            BSR         MIDISend

            BRA         CNOTES

CHORDHOLD   BSR         DELAY       ;Play the chord fora while
            BSR         DELAY

SILENCE     MOVEA.W     #CHORD,A5   ;Silence the notes
            MOVE.B      #NOTEOFF,STATUS
            MOVE.W      #STATUS,MIDISTR

NOTESOFF    MOVE.B      (A5)+,NOTENBR
            BEQ         START

            BSR         MIDISend
            BSR         MINIDELAY

            BRA         NOTESOFF

DONE        STOP        #0          ;Stop execution
            BRA         DONE

******************************************************************************
DELAY       MOVE.L      #150000,D5
******************************************************************************
SUBD5       SUBQ.L      #1,D5
            BNE         SUBD5
            RTS
******************************************************************************

MINIDELAY   MOVE.W      #150,D5
SUBD        SUBQ.W      #1,D5
            BNE         SUBD
            RTS

******************************************************************************
SENDMIDI    MOVE.B      PREVMIDI,D0
******************************************************************************
            MOVE.B      MIDINOTE,D1 ;If the previous note and current note
            CMP.B       D0,D1       ; are the same, there is no need to send
            BEQ         DONESNDMIDI ; other MIDI packets, so exit

            MOVE.B      #NOTEOFF,STATUS   ;Else, silence the previous note
            MOVE.B      PREVMIDI,NOTENBR
            MOVE.B      #VELOCITY,NOTEVEL
            MOVE.B      #0,MSGEND
            MOVE.W      #STATUS,MIDISTR
            BSR         MIDISend


            CMPI.B      #NOTEERR,D1 ;If there was an error identifying f0, or
            BEQ         DONESNDMIDI ; the output needs to be silent, exit

            MOVE.B      #NOTEON,STATUS    ;Else send the new note and exit
            MOVE.B      MIDINOTE,NOTENBR

            MOVE.B      D1,PREVMIDI ;Now the current note is the old one

            BSR         MIDISend

DONESNDMIDI RTS
******************************************************************************
```

```
***************************************************************************
*; MIDI AND SCI INITIALIZATION ROUTINES
***************************************************************************
MIDICONFIG  MOVE.B      #$3,MIDISC
            MOVE.B      #MIDICFG,MIDISC
            MOVE.B      #PROGCHANGE,STATUS
            MOVE.B      #0,PCEND
            MOVE.W      #STATUS,MIDISTR
            BSR         MIDISend
            RTS
***************************************************************************

***************************************************************************
MIDISend    MOVEA.W     MIDISTR,A6
***************************************************************************
*; DESCRIPTION: Sends a character to out the MII Port to the synthesizer.   ;*
*;                                                                          ;*
*; PARAMETERS:  A6 - Starting address of the data to send.  Data is sent    ;*
*;              until the NULL character ($00) is found.  NULL is not sent  ;*
*;              to the terminal.                                            ;*
***************************************************************************
POLLTDR     BTST.B      #TDRE,MIDISC    ;Waiting for the previous char to go
            BEQ         POLLTDR
            MOVE.B      (A6)+,D6
            BEQ         ENDMSND         ;If NULL is detected, then exit
            MOVE.B      D6,MIDITXD      ;Else send the new char
            BRA         POLLTDR
ENDMSND     RTS

*;------------------------ MIDISend Variables -----------------------------;*
MIDISTR     DC.W        $0
***************************************************************************

SCALE       DC.B        60,60,62,64,65,67,69,71,72,72,71
            DC.B        69,67,65,64,62,60,60,$FF,0
CHORD       DC.B        36,43,48,60,64,67,72,0

            END         START
```

IRQtest.x68

```
***************************************************************************
*; DESCRIPTION:  This test code is designed to test the terminal interface  ;*
*;               and auto-vectored interrupt requests by sending a test     ;*
*;               message to the PC terminal every time the interrupt is     ;*
*;               triggered.                                                 ;*
***************************************************************************

***************************************************************************
*;------------------------------ Memory Map -------------------------------;*
***************************************************************************
AVIRQ1      EQU         $64            ;Auto-Vector Interrupt Level 1

ENDRAM      EQU         $4000

SCISC       EQU         $8000
SCITXD      EQU         $8002
SCIRXD      EQU         $8002
***************************************************************************

***************************************************************************
*;-------------------------------- Flags ----------------------------------;*
***************************************************************************
RDRF        EQU         0             ;Receive data register full (ACIA)
TDRE        EQU         1             ;Transmit data register empty (ACIA)
***************************************************************************

***************************************************************************
*;---------------------------- Defined Values -----------------------------;*
***************************************************************************
```

```
SCICFG      EQU         $15         ;SCI set-up: 38400,8,N,1
******************************************************************************


            ORG         $00
            DC.L        ENDRAM,START

            ORG         AVIRQ1      ;Set the location of the auto-vectored
            DC.L        IRQTEST     ; interrup routine

            ORG         $400
START       BSR         SCICONFIG
            BSR         ENABLEIRQ

HERE        STOP        #0
            BRA         HERE


******************************************************************************
IRQTEST     MOVE.W      #ZIRQ,SCISTR
******************************************************************************
            BSR         SCISend
            RTE
******************************************************************************



******************************************************************************
*; ENABLING AND DISABLING INTERRUPTS
******************************************************************************
ENABLEIRQ   ANDI.W      #$FEFF,SR
            RTS

DISABLEIRQ  ORI.W       #$0700,SR
            RTS
******************************************************************************



******************************************************************************
*; SCI INITIALIZATION ROUTINE
******************************************************************************
SCICONFIG   MOVE.B      #$3,SCISC
            MOVE.B      #SCICFG,SCISC
            RTS
******************************************************************************



******************************************************************************
SCISend     MOVEA.W     SCISTR,A6
******************************************************************************
*; DESCRIPTION: Sends a character to out the SCI Port to the terminal.      ;*
*;                                                                          ;*
*; PARAMETERS:  A6 - Starting address of the data to send.  Data is sent    ;*
*;              until the NULL character ($00) is found.  NULL is not sent  ;*
*;              to the terminal.                                            ;*
******************************************************************************
POLLTDRE    BTST.B      #TDRE,SCISC    ;Waiting for the previous char to go
            BEQ         POLLTDRE
            MOVE.B      (A6)+,D6
            BEQ         ENDSSND         ;If NULL is detected, then exit
            MOVE.B      D6,SCITXD       ;Else send the new char
            BRA         POLLTDRE
ENDSSND     RTS

*;-------------------------- SCISend Variables -----------------------------;*
SCISTR      DC.W        $0
******************************************************************************

ZIRQ        DC.B        'TST',0

            END         START
```

The Proven Code for the Game of Snake (nibblesB.x68)

```
*********************************************************************
*;   ECE 516 - Project 1 - The Game of Snake                       ;*
*;   Group #3                                                       ;*
*;      C. Ray Dermon                                               ;*
*;      John D. Gant                                                ;*
*;      Travis R. Gault                                             ;*
*********************************************************************

*********************************************************************
*;------------------------ Friendly Masks ------------------------;*
*********************************************************************
BIT0    EQU             $01
BIT8        EQU         $100

*********************************************************************
*;------------------------- Memory Map ---------------------------;*
*********************************************************************
XYCOORD     EQU         $1000       ;2-byte coordinate X/Y coord
X_LOC       EQU         $1000       ; from bove: the x-coord
Y_LOC       EQU         $1001       ; and the y-coord
SCICHAR     EQU         $1002       ;Character read from the terminal

SNAKELN     EQU         $1003       ;Number of yellow snake chars on screen
HEADLOC     EQU         $1004       ;X/Y-coords of the head (x_byte,y_byte)
*INUSE      EQU         $1005
TAILLOC     EQU         $1006       ;X/Y/-coords of the tail (x_byte,y_byte)
*INUSE      EQU         $1007
DIRN        EQU         $1008       ;Direction of the snake
GAMEOVR     EQU         $1009       ;Game over flag ($FF=game over)

ZSTRING     EQU         $1010       ;Four Byte String
*INUSE      EQU         $1011
*INUSE      EQU         $1012
*INUSE      EQU         $1013
NIBXY       EQU         $1014       ;Current position of the nibble
*INUSE      EQU         $1015       ; on screen (x_byte,y_byte)
ZSCORE      EQU         $1016       ;Four bytes used to display the
*INUSE      EQU         $1017       ; ASCII verion of the score on
*INUSE      EQU         $1018       ; the screen
*INUSE      EQU         $1019
DELAY       EQU         $101A       ;Delay in loops between snake
*INUSE      EQU         $101B       ; movements
RANDCTR     EQU         $101C       ;Timeout counter for the random
*                                   ; number generator

HEADPTR     EQU         $1020       ;Points to the memory location that
*INUSE      EQU         $1021       ; contains the x/y-coords of the head
TAILPTR     EQU         $1022       ;Same as the head pointer, but for
*INUSE      EQU         $1023       ; the tail

POSPTR      EQU         $1100       ;The start of our queue

ACIASC      EQU         $8000
ACIATX      EQU         $8002
ACIARX      EQU         $8002

STACK       EQU         $2000
*********************************************************************

*********************************************************************
*;---------------------------- Flags -----------------------------;*
*********************************************************************
RDRF        EQU         0           ;Receive data register full (ACIA)
TDRE        EQU         1           ;Transmit data register empty (ACIA)
*********************************************************************

*********************************************************************
*;----------------------- Defined Values -------------------------;*
```

```
***********************************************************************
SCICFG      EQU     $15         ;$09        ;SCI set-up: 9600,7,E,1
X_LOW       EQU     $2          ;Lower and upper bounds
X_HIGH      EQU     $13         ; used to check if the snake has
Y_LOW       EQU     $2          ; run into a wall.
Y_HIGH      EQU     $13

XBIT        EQU     BIT8        ;Where to inc/dec the x/y coordinates
YBIT        EQU     BIT0        ; in memory
NULL        EQU     $0          ;String termination character

UP          EQU     $35         ;ASCII chars coordinating with the
DOWN        EQU     $32         ; directions for the snake to move.
LEFT        EQU     $31         ; 5=UP, 2=DOWN, 1=LEFT, 3=RIGHT
RIGHT       EQU     $33

DLYDEC      EQU     500         ;The amt of time to speed up the game
INITDLY     EQU     25005       ;The initial delay between snake moves
MINDLY      EQU     5           ;The fastest game speed
STRTLEN     EQU     2           ;Starting length of the snake
STOPPT      EQU     $0B16       ;Cursor Stopping point
MAXTRYS     EQU     11          ;Max tries to generate a random number
***********************************************************************


***********************************************************************
            ORG     $00
            DC.L    STACK,START
***********************************************************************
            ORG     $400
START       MOVE.B  #$FF,$8001
            MOVE.B  #03,ACIASC  ;Reset the ACIA and configure
            MOVE.B  #SCICFG,ACIASC          ; the serial communications

            MOVE.B  #STRTLEN,SNAKELN ;Set the initial length of the
            MOVE.W  #POSPTR,HEADPTR         ;  snake, and initialize pointers
            MOVE.W  #POSPTR,TAILPTR         ;  to the head and tail of the snake

            MOVE.W  #INITDLY,DELAY          ;Set the initial delay between moves

            MOVE.W  #$0909,HEADLOC          ;Set the position of the head and
            MOVE.W  #$0909,TAILLOC          ; tail to (x,y)=(9,9)

            MOVEA.W #POSPTR,A0  ;Initialize the first 256 memory
NINEO       MOVE.L  #$09090909,(A0)+ ;locations in the queue to the
            CMPA.W  #$1200,A0   ; starting position of the snake
            BNE     NINEO

            MOVE.W  #$0404,NIBXY            ;Set the initial food position

            MOVE.B  #UP,DIRN    ;Set the initial direction of the
            MOVE.B  #UP,SCICHAR ; snake.


            CLR.B   GAMEOVR                 ;Clear the game over flag

            JSR     STARTGAME   ;Show the splash screen

            JSR     DRAWBRD                 ;Draw the game board

            MOVE.W  #$0404,XYCOORD          ;Draw the first nibble (food)
            JSR     COORDS                  ; on the screen
            MOVEA.W #NIBBLE,A0
            JSR     SCISEND

            MOVE.W  #$0909,XYCOORD          ;Draw the head of the snake at
            JSR     COORDS                  ; (x,y)=(9,9)
            MOVEA.W #SNAKE,A0
            JSR     SCISend

            JSR     SCORE                   ;Update the score
```

```
**********************************************************************
*;----------------- This is the main delay loop below --------------;*
**********************************************************************
LOOPS       MOVE.W      DELAY,D2    ;Load the current delay time which
LOOP        SUB.W       #1,D2                   ; changes as the snake grows, then
            BNE         LOOP                    ; execute the delay

            JSR         SCIRead                 ;See if the user changed directions

            JSR         CHKCHAR                 ;See if the direction is valid

            JSR         MOVEALG                 ;Move the snake
            CMPI.B      #$FF,GAMEOVR            ;If it didn't hit a wall or itself,
            BNE         LOOPS                   ; then continue playing

            JSR         YOULOST                 ;Else, display the GAME OVER screen

            BRA         START                   ;Let the user play again
**********************************************************************

**********************************************************************
NIBPOS      CLR.B       RANDCTR
**********************************************************************
*;DESCRIPTION:  This function generates a random number for X and Y ;*
*;              ranging in value from 0-17dec for each coordinate.  ;*
*;              After a potential random number is generated, the   ;*
*;              value is checked against the value.                 ;*
*;                                                                  ;*
*;                                                                  ;*
*; PARAMETERS:  D0 - Used as a temporary register                   ;*
*;              D1 - Used as a temporary register                   ;*
*;              D2 - Used as a temporary register                   ;*
*;              NIBXY - Previous nibble x/y-coordinates             ;*
*;                                                                  ;*
*;    RETURNS:  NIBXY - The new x/y-coordinates of the nibble       ;*
*;                                                                  ;*
**********************************************************************
NPAGAIN     ADD.B       #1,RANDCTR  ;Check the timeout counter for too
            CMP.B       #MAXTRYS,RANDCTR ;many levels of recursion
            BNE         CALCAGN

            MOVE.W      TAILLOC,D0  ;If too many levels, set the new food
            BRA         ENDCHK                  ; position at the previous tail pos.

CALCAGN     CLR.L       D0                      ;Clear all temporary registers
            CLR.L       D1
            CLR.L       D2
            MOVE.W      #18,D1                  ;USED FOR THE MODULO OPERATION XMOD18
            MOVE.W      NIBXY,D0    ;NEED 32 BITS TO GET REMAINDER
            DIVU.W      #18,D0                  ;DIVIDING QUOTIENT BY 18
            ADD.B       #1,D0                   ;ADD 1 TO THE QUOTIENT
            MOVE.W      D0,D2                   ;PUT QUOTIENT IN D2 FOR USE LATER

            SWAP        D0                      ;SWAP QUOTIENT AND REMAINDER IN DATA
REGISTER

            MOVE.W      D0,D1                   ;COPY REMAINDER INTO D1
            ADD.W       D1,D2                   ;ADDING REMAINDER(D1) TO QUOTIENT(D2)
            MULU.W      TAILLOC,D2  ;MULTIPLYING D2 BY CURRENT TAIL XY COORDINATES
            CLR.L       D0                      ;CLEARING D0
            CLR.L       D1                      ;CLEARING D1

            MOVE.B      D2,D0                   ;MOVING QUOTIENT INTO Y POSITION
            MOVE.W      D2,D1                   ;MOVING REMAINDER INTO X POSITION
            LSR.W       #8,D1                   ;RIGHT SHIFTING THE REAMINDER (D1) BY
EIGHT
            DIVU.W      #16,D0                  ;DIVIDING THE QUOTIENT BY 16
            DIVU.W      #16,D1                  ;DIVIDING THE REAMINDER BY 16

            SWAP        D1                      ;SWAPPING THE REMAINDER AND QUOTIENT OF
```

```
THE PREVIOUS DIVISON
            SWAP        D0                      ;SWAPPING THE REMIANDER AND QUOTIENT OF
THE PREVIOUS DIVISON

            LSL.W       #8,D1                   ;LEFT SHIFTING D1 AS THE X CORDINATE
            ADD.W       D1,D0                   ;PUTTING THE X COORDINATE IN D0
            ADD.W       #$202,D0

********D0 NOW HAS NEW NIBBLE POSITION*************

***************************************************
*           CHECKING NEW NIBBLE AGAINST OLD NIBBLE POS*
***************************************************
            CMP.W       NIBXY,D0
            BEQ         NPAGAIN

            MOVE.W      D0,XYCOORD
            ADD.W       D0,NIBXY

*********************************
*           NOW GO THROUGH THE STACK*
*********************************
            CLR         D2
            CLR         D1
            MOVE.W      XYCOORD,D0
            MOVE.B      SNAKELN,D2
            ADDI.B      #1,D2                   ;FOR THE TAIL
            MOVEA.W     HEADPTR,A0
MOVEAG      MOVE.W      (A0)+,D1    ;USE D1 not D0!!!!
            CMP.W       #0,D2                   ;ARE WE AT THE END OF THE STACK?
            BEQ         ENDCK                   ;IF SO YOU ARE DONE CHECKING AND IT IS AN
OK POSITION
            CMP.W       D1,D0                   ;ELSE COMPARE STACK VALUE VERSUS INPUTTED
VALUE
            BEQ         NPAGAIN                 ;IF EQUAL SET RECALC NIB POSITION
            SUB.B       #1,D2                   ;ELSE SUBTRACT FROM D2
            BRA         MOVEAG

ENDCK       MOVE.W      D0,NIBXY    ;MOVE NEW POSITION INTO RANDM


            RTS
**********************************************************************


**********************************************************************
MOVEALG     NOP
**********************************************************************
*; DESCRIPTION:  Governs the mathematical calculations in main-     ;*
*;               taining the head and tail positions of the snake   ;*
*;                                                                  ;*
*;  PARAMETERS:  D0 - Used as a temporary variable                  ;*
*;               A0 - Used to send the various parts of the snake   ;*
*;               DIRN - Direction used to calculate the snake mvmt  ;*
*;               HEADLOC - x/y-coordinates of the snake head        ;*
*;               TAILLOC - Previous location of the tail on screen  ;*
*;               DELAY - The time between snake moves               ;*
*;               NIBXY - The position of the food                   ;*
*;               SNAKELN - Snake length (# of yellow chars on screen;*
*;                                                                  ;*
*;     RETURNS:  HEADLOC - New x/y-coordinates of the head          ;*
*;               SNAKELN - New snake length                         ;*
*;               DELAY - New delay time between moves               ;*
**********************************************************************
CHKLEFT     CMPI.B      #LEFT,DIRN  ;Checking Directions:
            BNE         CHKDOWN                 ; If the snake is moving left, then
            SUB.W       #XBIT,HEADLOC           ;  decrement the x-coordinate
            BRA         HEADREF

CHKDOWN     CMPI.B      #DOWN,DIRN  ; If the snake is moving down, then
            BNE         CHKRGHT                 ;  increment the y-coordinate,
```

```
            ADD.W       #YBIT,HEADLOC           ;  because the screen is backward for
            BRA         HEADREF                 ;  the vertical axis

CHKRGHT     CMPI.B      #RIGHT,DIRN ; If the snake is moving right, then
            BNE         CHKUP                   ;  increment the x-coordinate
            ADD.W       #XBIT,HEADLOC
            BRA         HEADREF

CHKUP       CMPI.B      #UP,DIRN    ; If the snake is moving up, then
            BNE         HEADREF                 ;  decrement the y-coordinate
            SUB.W       #YBIT,HEADLOC

HEADREF     MOVE.W      HEADLOC,XYCOORD         ;Jump to the new head coordinates
            JSR         COORDS                  ; on the screen and print the head
            MOVEA.W     #SNAKE,A0   ; character
            JSR         SCISend

            MOVE.W      HEADLOC,XYCOORD         ;Check to see if the snake ran into
            JSR         CHKBNDS                 ; a wall or itself

            MOVE.W      NIBXY,D0    ;Check to see if the snake ate a
            CMP.W       HEADLOC,D0  ; nibble
            BNE         MOVIN

            ADDQ.B      #1,SNAKELN  ;If the snake ate a nibble, then
            CMPI.W      #MINDLY,DELAY           ; increase the snake length, and
            BEQ         DLYSAME                 ; decrease the delay between moves
            SUB.W       #DLYDEC,DELAY

DLYSAME     JSR         SCORE                   ;Update the score

            JSR         NIBPOS                  ;Get the coordinates of the new nibble
            MOVE.W      NIBXY,D0    ; and place it on the screen
            MOVE.W      D0,XYCOORD
            JSR         COORDS
            MOVEA.W     #NIBBLE,A0
            JSR         SCISEND


MOVIN       JSR         PSHSTK                  ;Update the queue with the new head


TAILREF     MOVE.W      TAILLOC,XYCOORD         ;Update the tail on screen
            JSR         COORDS
            MOVEA.W     #TAIL,A0    ;Print the tail char
            JSR         SCISend

            MOVE.W      TAILLOC,D0  ;If the recursion timed out in the
            CMP.W       NIBXY,D0    ; random number generator, and the
            BNE         ENDMOVE                 ; new nibble position = the old tail

            MOVE.W      NIBXY,D0    ; position, then draw the nibble on
            MOVE.W      D0,XYCOORD  ; screen again, because it was
            JSR         COORDS                  ; overwritten when the tail was
            MOVEA.W     #NIBBLE,A0  ; updated on screen
            JSR         SCISEND


ENDMOVE RTS
**********************************************************************

**********************************************************************
SendStr     MOVE.L      ZSTRING,D0
**********************************************************************
*; DESCRIPTION: Sends a character to out the SCI Port to the        ;*
*;              terminal.  Upper byte is sent first, and lower byte ;*
*;              is sent last.  The whole string is loaded into a    ;*
*;              register, rotated left by a byte, sent, then done   ;*
*;              again until all bytes are sent.                     ;*
*;                                                                  ;*
*; PARAMETERS:  zString - String (4-byte) to send.                  ;*
```

```
*;              D0 - Data buffer for sending data.                   ;*
*******************************************************************
POLTD       BTST.B      #TDRE,ACIASC            ;Waiting for the previous char to go
            BEQ         POLTD
            ROL.L       #8,D0                   ;Rotate, Send, Clear, do it again
            CMPI.B      #NULL,D0
            BEQ         ENDSSTR
            MOVE.B      D0,ACIATX
            CLR.B       D0
            BRA         POLTD
ENDSSTR     RTS
*********************************************************************


*********************************************************************
Coords      MOVE.L      #$1B5B3030,D3           ;'ESC[yy'
*********************************************************************
*; DECSRIPTION: Changes the X,Y coordinates in the terminal.        ;*
*;              Manipulates the ANSI code ('ESC[yy;xxH') in 2 parts.;*
*;              The first part handles the y-Coords and the 2nd     ;*
*;              part handles the x-Coords. See comments.            ;*
*;                                                                   ;*
*;  PARAMETERS: XYCoord - A 2-byte hex number. Upper = X, Lower = Y ;*
*;              D3 - Register for data to be sent to the terminal   ;*
*;              D4 - Data for mathematical calculations             ;*
*;                                                                   ;*
*;     RETURNS: (none) - but the cursor is place on screen          ;*
*********************************************************************
            MOVE.W      XYCOORD,D4 ;D4 = XY Coordinates
            CMPI.B      #10,D4         ;If the Y coordinate is greater than 10
            BLT         NOADDY         ; then the first number is 1 else 0
            ADD.W       #BIT8,D3
            SUB.B       #10,D4         ;Now remove the 10 and add the rest

NOADDY      ADD.B       D4,D3          ;D4 = Y, D3=first half of ESC[#;#h
            MOVE.L      D3,ZSTRING ; Send the first half to  to the terminal
            JSR         SendStr

            MOVE.L      #$3B303048,D3           ; ';xxH'
            MOVE.W      XYCOORD,D4  ;D4=XY
            LSR.W       #8,D4                   ;D4=X
            CMPI.B      #10,D4                  ;If D4>10 then add 1 to upper 0
            BLT         NOADDX                  ; else, don't add
            ADD.L       #$00010000,D3
            SUB.W       #10,D4                  ;Subtract the 10 from D4

NOADDX      LSL.W       #8,D4                   ;Shift left by 8 to put the rest
            ADD.W       D4,D3                   ; of X into the formatted string

            MOVE.L      D3,ZSTRING  ;Send the string to the terminal
            JSR         SendStr

            RTS                                 ;Outta Here!
*********************************************************************

*********************************************************************
SCORE       MOVE.L      #$30303000,D0
*********************************************************************
*; DESCRIPTION:  Prints the score at the bottom of the playing area.;*
*;                                                                   ;*
*;  PARAMETERS:  D0,D1 - Temporary variables                        ;*
*;               SNAKELN - Current length of the snake              ;*
*;               SRTRLEN - The initial length of the snake          ;*
*;               ZSCORE - Local ASCII version of the score digits   ;*
*;                                                                   ;*
*;     RETURNS:  "SCORE: xxx"  on the screen                        ;*
*********************************************************************
            CLR.L       D1
            MOVE.B      SNAKELN,D1
            SUB.B       #STRTLEN,D1
```

```
ADDAGN      ADDQ.B      #1,D1
            NEG.B       D1
            MOVE.B      D1,$4001
            NEG.B       D1
            SUBQ        #1,D1
            CMPI.B      #100,D1
            BLT         NOAD100
            ADD.L       #01000000,D0
            SUB.B       #100,D1,

NOAD100     CMPI.B      #10,D1
            BLT         NOADD10
            ADD.L       #$00010000,D0
            SUB.B       #10,D1
            BRA         ADDAGN

NOADD10     LSL.L       #8,D1
            ADD.L       D1,D0
            MOVE.L      D0,ZSCORE

            MOVE.W      #$0016,XYCOORD
            JSR         COORDS

            MOVEA.W     #SCRTXT,A0
            JSR         SCISEND

            MOVE.L      ZSCORE,D0
            MOVE.L      D0,ZSTRING
            JSR         SENDSTR

            RTS
**********************************************************************

**********************************************************************
SCISend NOP
**********************************************************************
*; DESCRIPTION: Sends a character to out the SCI Port to the        ;*
*;              terminal.                                           ;*
*;                                                                  ;*
*; PARAMETERS:  A0 - Starting address of the data to send.  Data is ;*
*;              sent until the NULL character ($00) is found.  NULL ;*
*;              is not sent to the terminal.                        ;*
**********************************************************************
POLTDRE     BTST.B      #TDRE,ACIASC           ;Waiting for the previous char to go
            BEQ         POLTDRE
            MOVE.B      (A0)+,D0
        BEQ ENDSSND         ;If NULL is detected, then exit
            MOVE.B  D0,ACIATX       ;Sending the new char
            BRA         POLTDRE
ENDSSND     RTS
**********************************************************************


**********************************************************************
SCIRead NOP
**********************************************************************
*; DESCRIPTION: Gets a single char from the terminal.               ;*
*;                                                                  ;*
*; PARAMETERS:  None                                                ;*
*;                                                                  ;*
*; RETURNS:     Character sent from terminal, stored in D0.         ;*
**********************************************************************
POLRDRF     BTST.B      #RDRF,ACIASC           ;Checking for a character
            BEQ         ENDREAD
            MOVE.B      ACIARX,SCICHAR         ;Storing it in D0
ENDREAD     RTS
**********************************************************************


**********************************************************************
DrawBRD     NOP
```

```
*********************************************************************
*; DESCRIPITION: Draws the game board, no snake, no nibble.        ;*
*;                                                                 ;*
*;   PARAMETERS: D1 - Temporary variable                           ;*
*;               A0 - Starting addresses of the various part of the ;*
*;                    board drawn on screen.                       ;*
*;
;*
*;      RETURNS: The game board on screen                          ;*
*********************************************************************

          JSR       CLRSCREEN   ;Clear the terminal screen

TOPWALL   MOVEA.W   #WALL,A0    ;Draw the top wall
          JSR       SCISEND

          MOVE.B    #18,D1

MIDDLE    MOVEA.W   #BACKGND,A0 ;Draw the 18 strips of playing area
          JSR       SCISEND
          SUBQ.B    #$01,D1
          BNE       MIDDLE

BTMWALL   MOVEA.W   #WALL,A0    ;Draw the bottom wall
          JSR       SCISEND

          RTS
*********************************************************************

*********************************************************************
CHKCHAR NOP
*********************************************************************
*; DESCRIPTION: Checks the incoming character to see if it is a     ;*
*;              valid character. A valid char is subdivided into    ;*
*;              (up, down) & (left, right). Current direction of    ;*
*;              snake is checked and compared to subdivided groups  ;*
*;                  to define which group has a valid character. If     ;*
*;                  direction is up or down then check input char to    ;*
*;                  left and right, and if the direction is left or     ;*
*;                  right the check input character to up and down. If  ;*
*;                  input character is a valid character then update the;*
*;                  direction otherwise discard input character and keep;*
*;                  old direction.                                      ;*
*;                                                                 ;*
*; PARAMETERS:  SCICHAR - The input character to be checked.       ;*
*;              DIRN - Contains current direction of snake.        ;*
*;                                                                 ;*
*;    RETURNS:  DIRN - Keeps the input character if it is valid,    ;*
*;                        otherwise it returns the last valid direction;*
*********************************************************************
          CMP.B     #UP,DIRN    ;Direction is compared to the four
          BEQ       LRCHK                   ; valid orthogonal kepad directions.
          CMP.B     #DOWN,DIRN  ;Depending on which way the snake is
          BEQ       LRCHK                   ; moving will determine which set of
          CMP.B     #LEFT,DIRN  ; of checks will be determined valid.
          BEQ       UDCHK
          CMP.B     #RIGHT,DIRN
          BEQ       UDCHK

          BRA       ENDCCHR                 ;Ignore erroneous data


UDCHK     CMPI.B    #UP,SCICHAR ;Check to determine if the new input
          BEQ       SETUP                   ; character is a new valid direction
          CMPI.B    #DOWN,SCICHAR
          BEQ       SETDWN

          BRA       ENDCCHR                 ;Discard invalid characters

LRCHK     CMPI.B    #LEFT,SCICHAR
          BEQ       SETLFT
```

```
          CMPI.B      #RIGHT,SCICHAR
          BEQ         SETRGHT

          BRA         ENDCCHR                 ;Discard invalid characters

SETLFT    MOVE.B      #LEFT,DIRN  ;If a valid direction was entered,
          BRA         ENDCCHR                 ; update the direction accordingly
SETRGHT   MOVE.B      #RIGHT,DIRN
          BRA         ENDCCHR
SETUP     MOVE.B      #UP,DIRN
          BRA         ENDCCHR
SETDWN    MOVE.B      #DOWN,DIRN

ENDCCHR   RTS
**********************************************************************


**********************************************************************
PSHSTK    NOP
**********************************************************************
*; DESCRIPTION: A queue managment routine that keeps record of all  ;*
*;              the XY positions of the snake in memory location    ;*
*;              $1100 - snake length times two plus one word, which ;*
*;              is used for a blank space trailing the snake. There ;*
*;              are two bytes of data(a word) stored per snake      ;*
*;              length due to one byte for X position and 1 byte for;*
*;              Y position. When the snake moves it pushes a value  ;*
*;              of head location onto the queue and logically shifts;*
*;              right word length snake value until the end of the  ;*
*;              queue. Words shifted outside the queue are pushed   ;*
*;              and lost. A terminating word of 'BEEF' is placed at ;*
*;              the end of the queue for debugging purposes.        ;*
*;                                                                  ;*
*;  PARAMETERS: HEADPTR - set to $1100, start of queue              ;*
*;              TAILPTR - updated to end of queue and moves towards ;*
*;                        headptr during each word shift until it   ;*
*;                        reaches the headptr value.                ;*
*;              SNAKELN - length of snake is used to calculate end  ;*
*;                        of queue (queue is variable length, based ;*
*;                        SNAKELN)                                  ;*
*;              HEADLOC - word value that contains the XY coord-    ;*
*;                        inates of the Head of the snake           ;*
*;              TAILLOC - word value that contains the XY coord-    ;*
*;                        inates of the Tail of the snake           ;*
*;              POSPTR - contains the value $1100, start of stack   ;*
*;                                                                  ;*
*;    RETURNS:  HEADLOC - word value that contains the XY coord-    ;*
*;                        inates of the Head of the snake           ;*
*;              TAILLOC - word value that contains the XY coord-    ;*
*;                        inates of the Tail of the snake           ;*
**********************************************************************

          MOVE.W      HEADPTR,D0  ;Reset the Tailptr to headptr so that
          MOVE.W      D0,TAILPTR  ; we can recalculate tailptr position

          CLR         D0                      ;Take the snake length and multiply
          MOVE.B      SNAKELN,D0  ; by 2 to set a pointer to the end
          LSL.B       #1,D0                   ; of the queue. The addition of one
          ADD.W       D0,TAILPTR  ; more word is for the blank space at
          ADDQ.W      #2,TAILPTR  ; the end of the snake

PSHTAIL   MOVEA.W     TAILPTR,A0  ;A recursive loop is set to take the
          MOVEA.W     A0,A1                   ; tailptr and logically shift right
          MOVE.W      -(A0),(A1)  ; by a word length, the snake
          MOVE.W      A0,TAILPTR  ; positions in the queue until headptr
          CMPA.W      HEADPTR,A0  ; is reached. Once headptr is reached,
          BEQ         PSHHEAD                 ; then push the new head location
          BRA         PSHTAIL         ; onto the queue.


PSHHEAD   MOVE.W      HEADLOC,POSPTR
```

```
          ADD.W       D0,TAILPTR  ;This part of the routine puts a
          MOVEA.W     TAILPTR,A0  ; terminating word at the end of
          ADDA.W      #2,A0                  ; queue. This is useful for debugging
          MOVE.W      #$FEED,(A0) ; purposes to check the queue status.

          MOVE.W      -(A0),TAILLOC          ;Return the Tail location

ENDSTPS   RTS
**********************************************************************

**********************************************************************
CHKBNDS   NOP
**********************************************************************
*; DESCRIPTION: Check the current x/y-coordinates of the snake head ;*
*;              against the walls of the playing area, and against  ;*
*;              the other values of the snake.  If a match or an    ;*
*;              intersection is found, then the Game Over flag is   ;*
*;              set, so that the main loop knows to terminate play. ;*
*;                                                                  ;*
*;  PARAMETERS: D0,D1,D2,A0 - Temporary variables                   ;*
*;              SNAKELN - Current snake length                      ;*
*;              HEADPTR - Starting location of the queue            ;*
**********************************************************************
*********************************
*         CHECKING BOUNDS ON BOARD*
*********************************
          CLR         D0
          MOVE.B      X_LOC,D0
          CMP.B       #X_LOW,D0   ;CHECKS CURRENT POSITION AGAINST X LOW
          BCS         SETFG

          CMP.B       #X_HIGH,D0  ;CHECKS CURRENT POSITION AGAINST X HIGH
          BHI         SETFG

          MOVE.B      Y_LOC,D0
          CMP.B       #Y_LOW,D0   ;CHECKS CURRENT POSITION AGAINST Y LOW
          BCS         SETFG

          CMP.B       #Y_HIGH,D0  ;CHECKS CURRENT POSITION AGAINST Y HIGH
          BHI         SETFG

*********************************
*         NOW GO THROUGH THE STACK*
*********************************
          CLR         D2
          MOVE.W      XYCOORD,D0
          MOVE.B      SNAKELN,D2
          MOVEA.W     HEADPTR,A0
MOVEAGN   MOVE.W      (A0)+,D1    ;USE D1 not D0!!!!
          CMP.W       #0,D2                  ;ARE WE AT THE END OF THE STACK?
          BEQ         ENDCHK                 ;IF SO YOU ARE DONE CHECKING
          CMP.W       D1,D0                  ;ELSE COMPARE STACK VALUE VERSUS INPUTTED
VALUE
          BEQ         SETFG                  ;IF EQUAL SET GAMEOVER=$FF
          SUB.B       #1,D2                  ;ELSE SUBTRACT FROM D2
          BRA         MOVEAGN

SETFG     MOVE.B      #$FF,GAMEOVR

ENDCHK    RTS
**********************************************************************

**********************************************************************
CLRSCREEN NOP
**********************************************************************
          MOVEA.W     #CLRSCRN,A0 ;Send the sequence to clear the screen
          JSR         SCISEND                ; and set the colors to white

          RTS
**********************************************************************
```

```
**********************************************************************
STARTGAME NOP
**********************************************************************
            JSR         CLRSCREEN   ;Clear the screen

            MOVEA.W     #STARTMENU,A0           ;Display the splash screen
            JSR         SCISEND

GETSTRT     JSR         SCIREAD                 ;Wait for a [space]
            CMPI.B      #$20,SCICHAR
            BNE         GETSTRT

            RTS
**********************************************************************

**********************************************************************
YOULOST     NOP
**********************************************************************
            JSR         CLRSCREEN   ;Clear the screen

            MOVEA.W     #GAMEOVER,A0            ;Display the GAME OVER message
            JSR         SCISEND

GETKEY      JSR         SCIREAD                 ;Wait for a [space]
            CMPI.B      #$20,SCICHAR
            BNE         GETKEY

            CLR.B       GAMEOVR                 ;Clear the game over flag


            RTS
**********************************************************************



**********************************************************************
*;------------------------- Color Schemes --------------------------;*
**********************************************************************
WALL        DC.B        $1B,'[1;0;41m                    '
            DC.B        $1B,'[0;37;47m',10,13,0
BACKGND     DC.B        $1B,'[0;31;41m ',$1B,'[30;40m                  '
            DC.B        $1B,'[0;31;41m ',$1B,'[0;37;47m',10,13,0
SNAKE       DC.B        $1B,'[0;33;43m ',$1B,'[0;30;40m',0
NIBBLE      DC.B        $1B,'[0;32;40m*',0
WHITE       DC.B        $1B,'[0;37;47m',10,13,0
TAIL        DC.B        $1B,'[0;30;40m ',$1B,'[0;37;47m',0
CLRSCRN     DC.B        $1B,'[0;37;47m',$1B,'[=3;7h',$1B,'[2J',$1B,'[0;0H',0
**********************************************************************

**********************************************************************
*;------------------------ Building Blocks -------------------------;*
**********************************************************************
SCRTXT      DC.B        $1B,'[0;34;47mSCORE: ',0
SPACE       DC.B        ' ',0
ENDSPC      DC.B        ' ',10,13,0
WALLH       DC.B        '                   ',10,13,0
BG          DC.B        '                  ',0
BLAH        DC.B        $1B,'[3;7H',0
**********************************************************************


**********************************************************************
*;------------------------ Display Screens -------------------------;*
**********************************************************************
STARTMENU     DC.B      $1B,'[0;32;47m',10,13,10,13
              DC.B      '                     .-=-.          .--.',10,13
              DC.B      '         __        .`     `.       /  " )',10,13
              DC.B      ' _     .`  `.     /   .-.   \     /  .-`\',10,13
              DC.B      '( \   / .-.  \   /   /   \   \   /  /    ^',10,13
              DC.B      ' \ `-` /   \  `-`   /     \   `-`  /',10,13
```

```
               DC.B       '  `-.-`      `.____.`        `.____.`',10,13,10,13
               DC.B       $1B,'[0;30;47m','                The Game of Snake',10,13
               DC.B       ' by: Ray Dermon, John Gant, & Travis Gault',10,13,10,13
               DC.B       ' Use the numpad to move:   5',10,13
               DC.B       '                          123',10,13
               DC.B       ' DIRECTIONS:',10,13,
               DC.B       '   5 = UP, 2 = DOWN, 1 = LEFT, 3 = RIGHT',10,13,10,13
               DC.B       ' Don',$27,'t Run into walls or yourself.',10,13,10,13
               DC.B       ' Eat the Grub to Grow',10,13,10,13
               DC.B       'Press the space key to start. ',0

GAMEOVER DC.B             10,13,10,13,$1B,'[0;31;47m'
              DC.B        ' GGG     A    M     M  EEEEE',10,13
              DC.B        'G   G   A A   MM   MM  E',10,13
              DC.B        'G      A   A  M M M M  EEE',10,13
              DC.B        'G  GG  AAAAA  M  M  M  E',10,13
              DC.B        'G   G  A   A  M     M  E',10,13
              DC.B        ' GGG   A   A  M     M  EEEEE',10,13,10,13
              DC.B        '  OOO   V   V  EEEEE  RRRR',10,13
              DC.B        ' O   O  V   V  E      R   R',10,13
              DC.B        ' O   O  V   V  EEE    R  R',10,13
              DC.B        ' O   O  V   V  E      RRR',10,13
              DC.B        ' O   O   V V   E      R  R',10,13
              DC.B        '  OOO     V    EEEEE  R   R',10,13,10,13
              DC.B        $1B,'[0;30;47mPress the space key to start over. ',0
**********************************************************************

            END           $2000
```

APPENDIX IV – DESIGN SOURCE CODE

AMDF.x68

```
******************************************************************************
*;----------------------------- Memory Map -------------------------------;*
******************************************************************************
AVIRQ1      EQU         $64         ;Auto-Vector Interrupt Level 1

F0          EQU         $1000       ;F0.W contains the fundamental frequency
*;IN USE    EQU         $1001
FDEC        EQU         $1002
*;IN USE    EQU         $1003

IRQENFLAG   EQU         $1004       ;Current status of interrupts
MIDIFLAGS   EQU         $1005       ;MIDI flags for sending notes

MIDIINST    EQU         $1006       ;Location containing Instrument Nbr.
ERRCOUNT    EQU         $1007       ;Number of times there are f0 errors

KNVAL       EQU         $14E0       ;Current value of k+n
*;IN USE    EQU         $1001
X           EQU         $14F0       ;Temporary summation variable
*;IN USE    EQU         $1001

NMAX        EQU         $14F4       ;Maximum allowable value of N
*;IN USE    EQU         $1005
KMAX        EQU         $14F6       ;Maximum allowable value of K
*;IN USE    EQU         $1007
KNMAX       EQU         $14F8       ;Maximum allowable value of K+N
*;IN USE    EQU         $1009
DDMAX       EQU         $14FA       ;Maximum array index for AMDF results
*;IN USE    EQU         $100B
MAXOFDD     EQU         $14FC       ;Max{AMDF_RESULTS}
*;IN USE    EQU         $100D
FMAX        EQU         $14FE       ;Maximum array index for FINDEX
*;IN USE    EQU         $100F

STATUS      EQU         $10FA       ;MIDI Command
NOTENBR     EQU         $10FB       ;MIDI Note number
PROGNUM     EQU         $10FB       ;MIDI Program number
NOTEVEL     EQU         $10FC       ;MIDI Note Velocity
MSGEND      EQU         $100D       ;End of Message

PREVMIDI    EQU         $10FE       ;Previous MIDI note sent
MIDINOTE    EQU         $10FF       ;Current MIDI note to send


S           EQU         $1100       ;Sampled data from ADC
DD          EQU         $1300       ;ADMF results (word lengths)
FINDEX      EQU         $1700       ;Array indices for dips in AMDF

ENDRAM      EQU         $4000       ;End of RAM memory location + 1

LATCH       EQU         $4001       ;Location of register for LEDs

ADC0        EQU         $4000       ;A/D converter channel

MIDISC      EQU         $8001       ;MIDI ACIA status/control register
MIDITXD     EQU         $8003       ;MIDI ACIA transmit data register
MIDIRXD     EQU         $8003       ;MIDI ACIA receive data register

SCISC       EQU         $8000       ;MIDI ACIA status/control register
SCITXD      EQU         $8002       ;MIDI ACIA transmit data register
SCIRXD      EQU         $8002       ;MIDI ACIA receive data register
******************************************************************************
```

```
****************************************************************************
*;------------------------------- Flags ----------------------------------;*
****************************************************************************
RDRF        EQU         0           ;Receive data register full (ACIA)
TDRE        EQU         1           ;Transmit data register empty (ACIA)

ENABLEF0    EQU         $FF
NOTEERR     EQU         $EE         ;Denotes an error finding f0
****************************************************************************


****************************************************************************
*;--------------------------- Defined Values -----------------------------;*
****************************************************************************
SCICFG      EQU         $15         ;SCI set-up: 38400,8,N,1
MIDICFG     EQU         $15         ;MIDI setup; 31.25kHz,8,N,1

FS          EQU         1000

NMAXVAL     EQU         192         ;Nbr of samples to take (outter loop)

STARTADC    EQU         $AD         ;Dummy value to start A/D conversion

FINDEND     EQU         $FEED       ;FINDEX end-of-array flag
WORDMAX     EQU         $FFFF       ;Initial value for finding minimum vlaue

GRANDPIANO  EQU         0           ;MIDI Instrument Nbr. for a piano
NOTEON      EQU         $90         ;Note on channel 0
NOTEOFF     EQU         $80         ;Note off channel 0
PROGCHNG    EQU         $C0         ;Program change on channel 0
VELOCITY    EQU         $7F         ;Max note velocity
****************************************************************************


*;----------------------------------------------------------------------;*
*; These statements tell the 68000 where the start of the program space and ;*
*; stack space begins.  Additionally, the auto-vectored interrupt service   ;*
*; routine address location is defined.                                   ;*
*;----------------------------------------------------------------------;*
            ORG         $00
            DC.L        ENDRAM,START

            ORG         AVIRQ1
            DC.L        ADCIRQ

*;----------------------------------------------------------------------;*
*; This is the start of the program, where all variables and hardware     ;*
*; devices are initialized.                                               ;*
*;----------------------------------------------------------------------;*
            ORG         $400
START       CLR.L       D0
            MOVE.L      D0,IRQENFLAG  ;Clear IRQEN and MIDI flags

            MOVE.B      #GRANDPIANO,MIDIINST

            MOVE.W      #NMAXVAL,NMAX
            ADDI.W      #NMAXVAL,NMAX     ;Nmax = highest address in S-array
            ADDI.W      #S,NMAX     ;Nmax = highest address in S-array

            MOVE.W      #NMAXVAL,D0
            ADDI.W      #S,D0     ;Kmax = highest address accessed by k in s
            MOVE.W      D0,KMAX

            MOVE.W      #NMAXVAL,KNMAX
            ADDI.W      #NMAXVAL,KNMAX
            ADD.W       D0,KNMAX

            MOVE.W      #NMAXVAL,D0
            LSL.W       #1,D0    ; Twice as long as n-max b/c word length
            ADDI.W      #DD,D0
            MOVE.W      D0,DDMAX
```

```
            MOVE.W      #NMAXVAL,D0
            LSL.W       #1,D0    ; Twice as long as n-max b/c word length
            ADDI.W      #FINDEX,D0
            MOVE.W      D0,FMAX

            BSR         INITFINDEX
            BSR         CTRCLR
            BSR         SCICONFIG
            BSR         MIDICONFIG

            MOVEA.W     #S,A0
            MOVE.W      A0,KNVAL
            ADDQ.W      #2,KNVAL

            MOVE.W      #WORDMAX,MAXOFDD

            BSR         ENABLEIRQ   ;Enable Interrupts

HERE        BSR         SENDMIDI    ;Infinite loop to check status of MIDI
            BRA         HERE        ; flags and perhaps send MIDI commands

BUTNOTHERE  BRA         START       ;Should never reach this point
*;          MOVEA.W     #S2,A5      ;Used for testing


******************************************************************************
ADCIRQ      CLR.W       D0
******************************************************************************
*; DESCRIPTION: This (auto-vectored) interrupt routine is responsible for   ;*
*;             sampling the A/D converter, illuminating the LEDs,            ;*
*;             performing the AMDF iterations and calling the necessary      ;*
*;             routines to find the fundamental frequency and output the     ;*
*;             data to the synthesizer.  The for-loop below represents the   ;*
*;             general idea of the AMDF.                                     ;*
*;                                                                           ;*
*;                          for n=1:N                                        ;*
*;                              x=0;                                         ;*
*;                              for k=1:M                                    ;*
*;                                  x=x+abs(s(k)-s(k+n));                    ;*
*;                              end                                          ;*
*;                              d(n)=x;                                      ;*
*;                          end                                              ;*
*;                                                                           ;*
******************************************************************************

            MOVE.B      ADC0,D0     ;Take the reading from the last sample
*;          MOVE.B      (A5)+,D0    ;Used for testing
            MOVE.W      D0,(A0)     ;Copy the value to the S array


*;-----------------------------------------------------------------------;*
*; This code segment does a binary search if the signal amplitude is !=0 to ;*
*; light up the LEDs.  The MSB and all lower bits of the LEDs are lit       ;*
*; corresponding to the value read from the A/D converter.  This makes log- ;*
*; scale amplitude representation, but can easilyt be modified for a linear ;*
*; scale.                                                                   ;*
*;-----------------------------------------------------------------------;*
            BNE         BINBITCHK   ;Do binary search if amplitude != 0
            MOVE.B      #$FF,LATCH  ; else turn all LEDs off and
            BRA         CMPA02KMAX    ; continue with routine

BINBITCHK   CMPI.B      #$0F,D0
            BLS         BIT03

            CMPI.B      #$3F,D0
            BLS         BIT5

BIT7        CMPI.B      #$80,D0
            BLO         BIT6
```

```
            MOVE.B      #$00,LATCH    ;Light ALL the LEDs
            BRA         CMPA02KMAX    ; and continue with the routine

BIT6        MOVE.B      #$80,LATCH  ;Light the lower 7 LEDs
            BRA         CMPA02KMAX    ; and continue with the routine

BIT5        CMPI.B      #$20,D0
            BLO         BIT4
            MOVE.B      #$C0,LATCH  ;Light the lower 6 LEDs
            BRA         CMPA02KMAX    ; and continue with the routine

BIT4        MOVE.B      #$E0,LATCH  ;Light the lower 5 LEDs
            BRA         CMPA02KMAX    ; and continue with the routine


BIT03       CMPI.B      #$03,D0
            BLS         BIT1

BIT3        CMPI.B      #$08,D0
            BLO         BIT2
            MOVE.B      #$F0,LATCH  ;Light the lower 4 LEDs
            BRA         CMPA02KMAX    ; and continue with the routine

BIT2        MOVE.B      #$F8,LATCH  ;Light the lower 3 LEDs
            BRA         CMPA02KMAX    ; and continue with the the routine

BIT1        CMPI.B      #$02,D0
            BLO         BIT0
            MOVE.B      #$FC,LATCH  ;Light the lower 2 LEDs
            BRA         CMPA02KMAX    ; and contine with the routine

BIT0        MOVE.B      #$FE,LATCH  ;Light the lowest LED


*;-----------------------------------------------------------------------;*
*;          A0 has the current address pointer for the s-array (n)
*;          If A0 >= kmaxval + nmaxval + s, then run find freq
*;-----------------------------------------------------------------------;*
CMPA02KMAX  CMPA.W      KMAX,A0     ;If n (A0) < KMAX measurements, exit
            BLO         NEXTCONV    ;  after initiating another measurement
*;-----------------------------------------------------------------------;*
*;          X is as it appears in the MATLAB for-loop above
*;-----------------------------------------------------------------------;*
            CLR.W       D5          ;Else do an ADMF cycle (x=0)
            MOVEA.W     #S,A1       ;Initialize k(A1)=0, beginning of s-vals
            MOVEA.W     KNVAL,A2    ;Initialize k+n = n, curr. location in s

*                                   ;The upper byte of D1/D2 should always = 0
            CLR.W       D1          ;Clear temp. vars as words b/c s-vals are
            CLR.W       D2          ; byte length, but cals are word length

FORK12M     MOVE.W      (A1)+,D1    ;D1 = s(k), k++
            SUB.W       (A2)+,D1    ;D1 = s(k)-s(k+n)
            BPL         NEXTSTEP    ;If result positive, go to the next step
            NEG.W       D1          ;  else negate D1 (like abs function)

NEXTSTEP    ADD.W       D1,D5       ;x = x + abs( s(k)-s(k+n) )

            CMPA.W      KMAX,A1     ;Re-iterate inner loop
ENDKLOOP    BLO         FORK12M

*;-----------------------------------------------------------------------;*
*;          Now the inner loop is finished, and it is time to store the
*;           result in the DD data array of word lengths
*;-----------------------------------------------------------------------;*
STORINGD    MOVE.W      D5,(A3)+    ;d(m)=x, m++ (~5600/loop)
            ADDQ        #2,KNVAL    ;Increment KNVAL

*;          In this code, n really runs from 0 to M+N from above
MAXDDGTX    CMPA.W      KNMAX,A0    ;If n(A0)<(kmax+nmax) then start the
            BLO         NEXTCONV    ; next conversion & exit
```

```
            BSR         DISABLEIRQ  ;Disable Interrupts

NOCHAR      BSR         CTRCLR      ;Else clear the counters

            BSR         FINDINDICES ; and find the fundamental frequency
            BSR         FDIFF
            BSR         FAVG
            BSR         INITFINDEX
            BSR         ENABLEIRQ   ;Enable Interrupts
            BRA         DONEIRQ     ;Do not increment A0
            BSR         CTRCLR      ;Clear the counters again


NEXTCONV    ADDQ.W      #2,A0       ;MOVE.B     (A0)+,DUMMY   ;n++
DONEIRQ     MOVE.B      #STARTADC,ADC0   ;Start next conversion
            RTE
*;          BRA         ADCIRQ      ;Used for testing
*;--------------------------- ADCIRQ Variables ----------------------------;*
DUMMY       DC.L        $0
******************************************************************************


******************************************************************************
FINDINDICES MOVEA.W      #DD,A0     ;D6 = max(DD)
******************************************************************************
*; DESCRIPTION: This routine searches through the ADMF values stored in the ;*
*;              DD array and looks for values that are less than or equal   ;*
*;              to 25% of maximum value in the array (already calculated).  ;*
*;              Once the threshold has been identified, the minimum value in;*
*;              the data segment below the threshold is found.  The first   ;*
*;              index that corresponds to the minimum value in the segment  ;*
*;              is recorded in the Findex array, and the processes repeat   ;*
*;              until all values in Findex are examined.  Functionality has ;*
*;              been tested and confirmed with the Easy68K simulator.       ;*
*;                                                                          ;*
*; PARAMETERS:  DD.W - Word length array containing the ADMF results from   ;*
*;                     the ADCIRQ interrrupt routine.                       ;*
*;              MAXOFDD.W - The highest numerical value in the DD array.    ;*
*;              DDMAX.W - Maximum index allowed in the DD array.            ;*
*;              WORDMAX.W - $FFFF                                           ;*
*;                                                                          ;*
*;   REGISTERS: A0 - Used to iterate through DD to look for min values.     ;*
*;              A1 - Used to iterate through FIndex to store results        ;*
*;              D1 - Holds dd(n)                                            ;*
*;              D3 - Stores the current minimum in the data segment that    ;*
*;                   contains values falling below the threshold. Once      ;*
*;                   values exceed the threshold, it is re-initialized to   ;*
*;                   WORDMAX.                                               ;*
*;              D6 - Contains MAXOFDD/4, the threshold for min-value detec- ;*
*;                   tion.                                                  ;*
*;                                                                          ;*
*;     RETURNS: FINDEX.W - The array containing the indices of where the    ;*
*;                         minpoints occur below the threshold (D6).        ;*
******************************************************************************
            CLR.W       D6

FINDMAXDD   MOVE.W      (A0)+,D3    ;Can't compare 2 mem. addresses directly
            CMP.W       D3,D6       ;Was X, If max(DD)>=X,
            BHS         CHKDDMEM    ; then next check (MAXDDGTX)
            MOVE.W      D3,D6       ;Was X, else max(DD)=X
CHKDDMEM    CMPA.W      DDMAX,A0
            BLO         FINDMAXDD

            LSR.W       #2,D6       ;Divide by 4 for threshold detection

*;------------------------------------------------------------------------;*
*;          We need to iterate.W thru DD and search for values below thresh ;*
*;           then store the indices of those values in a new array          ;*
*;------------------------------------------------------------------------;*
```

```
          MOVEA.W     #FINDEX,A1   ;Initialize m=0 for new f array

          MOVEA.W     #DD,A0
GTTHRESH  CMPA.W      DDMAX,A0   ;Make sure the array index is in range
          BHS         DONEFIND

          MOVE.W      (A0)+,D1
          CMP.W       D6,D1       ;If dd(n)<Threshold(D6)
          BLO         FINDMIN     ; then find the data segment min-point
          BRA         GTTHRESH    ;Else go to the next data point

*;--------------------------------------------------------------------;*
*;        Once the threshold is detected, only move the index of the min- ;*
*;         value into FINDEX.  This is done sequentially by examining    ;*
*;         each value and checking for a minimum value until the values  ;*
*;         exceed the threshold, at which time the loop exits.           ;*
*;--------------------------------------------------------------------;*
FINDMIN   MOVE.W      #WORDMAX,D3 ;To find the min, start with the max
FMINLOOP  CMPA.W      DDMAX,A0    ;Make sure the array index is in range
          BHS         DONEFIND

          CMP.W       D1,D3       ;If CurrentMin(D3) > current value(D1)
          BHI         CHANGEMIN   ; then change min value index

          CMP.W       D6,D1       ;ElseIf CurrValue > threshold, skip
          BHI         SKIPFMIN    ; the FindMin loop
          MOVE.W      (A0)+,D1    ;Else, load the next value
          BRA         FMINLOOP    ; and repeat the loop

SKIPFMIN  ADDQ.W      #2,A1       ;Increment array index for FINDEX
          BRA         GTTHRESH    ; & wait to fall below threshold again

CHANGEMIN MOVE.W      A0,(A1)     ;Update Findex-array
          MOVE.W      D1,D3       ;CurrentMin(D3) = Current DD-value
          MOVE.W      (A0)+,D1    ;Get next value for comparison
          BRA         FMINLOOP    ;Still in the FindMin loop

DONEFIND  RTS
******************************************************************************


******************************************************************************
FDIFF     MOVEA.W     #FINDEX,A0
******************************************************************************
*; DESCRIPTION: Takes the indices stored by FINDINDICES() and calculates    ;*
*;              the differences between them.  This provides a list of      ;*
*;              periods that should be similar in magnitude.  The formula   ;*
*;              f(n)=f(n+1)-f(n) is used, where f(n) is the current value   ;*
*;              read in the Findex array.  Functionality has been tested    ;*
*;              and confirmed with the Easy68K simulator.                    ;*
*;                                                                           ;*
*;  PARAMETERS: FINDEX - The array containing the indices of the low points ;*
*;                       generated by the ADMF function (ADCIRQ).            ;*
*;              FINDEND - F-INDex-END specifies the end location of Findex  ;*
*;              DUMMY - A dummy-variables used to help the address registers;*
*;                      increment/decrement without changing their contents.;*
*;                                                                           ;*
*;   REGISTERS: A0 - Represents n in f(n), and where to store the results   ;*
*;                   in the Findex array.                                    ;*
*;              A1 - Represents n+1 in f(n+1)                                ;*
*;              D0 - Contains a copy of f(n) used for subtraction.          ;*
*;              D1 - Contains a copy of f(n+1) used for subtraction.        ;*
*;                                                                           ;*
*;    RETURNS:  FINDEX - The array containing the sample periods from ADMF. ;*
******************************************************************************

*;--------------------------------------------------------------------;*
*;        Now that the indices are found in the FIndex array, it is time  ;*
*;         to calculate the periods by using the formula f(n)=f(n+1)-f(n).;*
*;--------------------------------------------------------------------;*
          MOVEA.W     #FINDEX,A1
```

```
          MOVEA.W     #FINDEX,A2
          ADDQ.W      #4,A1       ;Skip the first few entry because it is
          ADDQ.W      #2,A0       ; erroneous due the nature of the ADMF

          CMPI.W      #FINDEND,(A0) ;If the 1st value of FIndex != the end
          BNE         FDIFFCALC     ; of array flag, then perform fdiff()

          BRA         DONEFDIFF     ;Else exit the routine

FDIFFCALC CMPI.W      #FINDEND,(A1)
          BEQ         DONEFDIFF

          MOVE.W     (A1)+,D1       ;D1=f(n+1)
          MOVE.W     (A0)+,D0       ;D0=f(n)

          SUB.W       D0,D1         ;D1=f(n+1)-f(n)
          MOVE.W      D1,(A2)+      ;f(n)=D1
          BRA         FDIFFCALC

DONEFDIFF MOVE.W      #FINDEND,(A0)  ;Replace last 2 entries in the Findex
          MOVE.W      #FINDEND,-(A0) ; array with the end-of-array flag
          RTS
******************************************************************************


******************************************************************************
FAVG      CLR.W       D0
******************************************************************************
*; DESCRIPTION: This routine sums all entries in Findex after FDIFF has     ;*
*;              been run on Findex.  The sampling frequency (Fs) is         ;*
*;              multiplied by the number of measurements that are recorded  ;*
*;              in Findex, then the result is divided by the sum of elements;*
*;              from FDiff.  This result is stored as the fundamental       ;*
*;              frequency (F0).  Functionality has been tested and          ;*
*;              confirmed with the Easy68K simulator.                       ;*
*;                                                                          ;*
*;  PARAMETERS: FINDEX.W - The array containing the sample periods.         ;*
*;              FINDNEND - The maximum array index for the Findex array.    ;*
*;              FS - The sample frequency in Hz.                            ;*
*;                                                                          ;*
*;   REGISTERS: A0 - Used to iterate through the periods in FIndex          ;*
*;              D0 - Used to sum the entries in FDiff                       ;*
*;              D1 - Holds the number of measurements                       ;*
*;              D2 - Copy of Fs                                             ;*
*;                                                                          ;*
*;     RETURNS: F0 - The approximate fundamental frequency.                 ;*
******************************************************************************
          MOVEA.W     #FINDEX,A0
TSUM      CMPI.W      #FINDEND,(A0)
          BEQ         DONETSUM

          ADD.W       (A0)+,D0
          BRA         TSUM

DONETSUM  MOVE.W      A0,D1
          SUBI.W      #FINDEX,D1  ;D1 = Number of measurements * 2
          BEQ         DONEFAVG

*;--------------------------------------------------------------------------;*
*;        F0 = Fs(D2) * NumMeasurements(D1) / SumPeriods(D0)                ;*
*;--------------------------------------------------------------------------;*
          MOVE.W      #FS,D2
          MULU.W      D2,D1       ;D1=Fs*NumMeasurements
          DIVU.W      D0,D1       ;D1=D1/Tsum => D1.w=freq

DONEFAVG  SWAP        D1          ;D1.W =remainder
          LSR.W       #1,D0       ;Divide D0 by 2
          CMP.W       D1,D0       ;If D0/2 < REM(f0)
          BHS         SWAPF0      ; write the value as it

          SWAP        D1          ;Else round up by one
```

```
           ADDQ.W      #1,D1
           BRA         STOREF0

SWAPF0     SWAP        D1
STOREF0    MOVE.W      D1,F0        ;Writes the value for F0

ENDFAVG    RTS
*****************************************************************************


*****************************************************************************
FREQ2MIDI  MOVEA.W     F0,A0
*****************************************************************************
*; DESCRIPTION: This routine takes the frequency (F0) and uses it as an     ;*
*;              index offset for a very long linked list called MIDITABLE.  ;*
*;              This linked list contains each MIDI note number (36-84dec)  ;*
*;              corresponding to frequencies from 63-1078Hz.                ;*
*;                                                                          ;*
*;  PARAMETERS: F0 - The fundamental frequency calculated in FAVG.          ;*
*;              A0 - Address register used to access the table.             ;*
*;                                                                          ;*
*;     RETURNS: MIDINOTE - The MIDI note number later sent to the           ;*
*;                         synthesizer.                                     ;*
*****************************************************************************
           CMPA        #63,A0       ;If 62<A0<=500, then store the note
           BLO         BADNOTE      ; Else store the bad not flag in MIDINOTE
           CMPA        #500,A0
           BLS         GOODNOTE

BADNOTE    MOVE.B      #NOTEERR,MIDINOTE
           ADDQ.B      #1,ERRCOUNT ;If f0 out of range, record an error and
           CMPI.B      #3,ERRCOUNT ; set the MIDI flags to Xmit a new note.
           BLO         DONEF2M      ; Silence output if 3 consecutive errors

           MOVE.B      #NOTEOFF,MIDIFLAGS

GOODNOTE   SUBA.W      #63,A0
           ADDA.W      #MIDILIST,A0
           MOVE.B      (A0),MIDINOTE
           MOVE.B      #ENABLEF0,MIDIFLAGS
           CLR.B       ERRCOUNT

DONEF2M    RTS
*****************************************************************************


*****************************************************************************
SENDMIDI   MOVE.B      MIDIFLAGS,D2
*****************************************************************************
*; DESCRIPTION: This routine examines the MIDI flags set in this software   ;*
*;              and takes action based on these flags.  If the flags are    ;*
*;              clear, then the routine exits.  If the NOTE OFF flag is     ;*
*;              detected, then the previous MIDI note is disabled.  If any  ;*
*;              other value is detected, then a MIDI packet instructing the ;*
*;              synthesizer to change notes is sent if the current note     ;*
*;              differs from the previous note.                             ;*
*;                                                                          ;*
*;  PARAMETERS: MIDIFLAGS - Determines what action to take.                 ;*
*;              PREVMIDI - Previous MIDI note sent                          ;*
*;              MIDINOTE - Current MIDI note to evaluate                    ;*
*;                                                                          ;*
*;   REGISTERS: D0 - Contains the previous MIDI note                        ;*
*;              D1 - Contains the current MIDI note                         ;*
*;              D2 - Contains the MIDI flags                                ;*
*;                                                                          ;*
*;     RETURNS: MIDIFLAGS - Cleared upon exit under any case.               ;*
*****************************************************************************
           BEQ         DONESNDMIDI ;If the flags are clear, then exit

           CMPI.B      #NOTEOFF,D2 ;Are we to disable the previous note w/o
           BNE         NEWNOTE      ; sending another note? If not, send note
```

```
SILENCE     MOVE.B      #NOTEOFF,STATUS   ;Else, silence the previous note
            MOVE.B      PREVMIDI,NOTENBR
            MOVE.B      #VELOCITY,NOTEVEL
            MOVE.B      #0,MSGEND
            MOVE.W      #STATUS,MIDISTR
            BSR         MIDISend

            BRA         DONESNDMIDI

NEWNOTE     MOVE.B      PREVMIDI,D0
            MOVE.B      MIDINOTE,D1 ;If the previous note and current note
            CMP.B       D0,D1       ; are the same, there is no need to send
            BEQ         DONESNDMIDI ; other MIDI packets, so exit

            CMPI.B      #NOTEERR,D1 ;Exit if an error identifying f0
            BEQ         DONESNDMIDI

            MOVE.B      #NOTEOFF,STATUS   ;Else, silence the previous note
            MOVE.B      PREVMIDI,NOTENBR
            MOVE.B      #VELOCITY,NOTEVEL
            MOVE.B      #0,MSGEND
            MOVE.W      #STATUS,MIDISTR
            BSR         MIDISend

            MOVE.B      #NOTEON,STATUS    ;And send the new note and exit
            MOVE.B      MIDINOTE,NOTENBR
            BSR         MIDISend

DONESNDMIDI CLR.B       MIDIFLAGS

            RTS
*******************************************************************************


*******************************************************************************
MIDISend    MOVEA.W     MIDISTR,A6
*******************************************************************************
*; DESCRIPTION: Sends a character to out the MII Port to the synthesizer.   ;*
*;                                                                          ;*
*; PARAMETERS:  A6 - Starting address of the data to send.  Data is sent    ;*
*;              until the NULL character ($00) is found.  NULL is not sent  ;*
*;              to the terminal.                                            ;*
*******************************************************************************
POLLTDR     BTST.B      #TDRE,MIDISC    ;Waiting for the previous char to go
            BEQ         POLLTDR
            MOVE.B      (A6)+,D0
            BEQ         ENDMSND         ;If NULL is detected, then exit
            MOVE.B      D6,MIDITXD      ;Else send the new char
            BRA         POLLTDR
ENDMSND     RTS

*;------------------------- MIDISend Variables -----------------------------;*
MIDISTR     DC.W        $0
*******************************************************************************


*******************************************************************************
CTRCLR      MOVEA.W     #S,A0       ;S array index (storing samples)
*******************************************************************************
            MOVEA.W     #0,A1       ;k (accessing S)
            MOVEA.W     #0,A2       ;k+n (accessing S)
            MOVEA.W     #DD,A3      ;DD array index (storing ADMF steps)
*;   CLR.W       MAXOFDD     ;Used for max(DD);
            RTS
*******************************************************************************


*******************************************************************************
INITFINDEX  MOVEA.W     #FINDEX,A0
*******************************************************************************
```

```
            MOVE.W      FMAX,D0
FLOOP       MOVE.W      #FINDEND,(A0)+ ;Fill array w/ the end of array flag
            CMPA.W      D0,A0
            BLS         FLOOP
            RTS                     ;Exit
******************************************************************************


******************************************************************************
*; ENABLING AND DISABLING INTERRUPTS
******************************************************************************
ENABLEIRQ   ANDI.W      #$F8FF,SR
            MOVE.B      #$FF,IRQENFLAG
            RTS

DISABLEIRQ  ORI.W       #$0700,SR
            MOVE.B      #$00,IRQENFLAG
            RTS
******************************************************************************


******************************************************************************
*; MIDI AND SCI INITIALIZATION ROUTINES
******************************************************************************
SCICONFIG   MOVE.B      #$3,SCISC
            MOVE.B      #SCICFG,SCISC
            RTS

MIDICONFIG  MOVE.B      #$3,MIDISC
            MOVE.B      #MIDICFG,MIDISC
            RTS
******************************************************************************

            INCLUDE     'MIDIList.x68'

            END         START
```

MIDIList.x68

```
ORG         $2000
******************************************************
*; Pitch |  F0        | Valid F0s | Offsets   | MIDI ;*
*;-------------------------------------------------;*
*; C2    |  65.406    | 63-67     | 0-4       |  36  ;*
*; C#2   |  69.296    | 68-71     | 5-8       |  37  ;*
*; D2    |  73.416    | 72-75     | 9-12      |  38  ;*
*; D#2   |  77.782    | 76-80     | 13-17     |  39  ;*
*; E2    |  82.407    | 81-84     | 18-21     |  40  ;*
*; F2    |  87.307    | 85-89     | 22-26     |  41  ;*
*; F#2   |  92.499    | 90-95     | 27-32     |  42  ;*
*; G2    |  97.999    | 96-100    | 33-37     |  43  ;*
*; G#2   |  103.826   | 101-106   | 38-43     |  44  ;*
*; A2    |  110       | 107-113   | 44-50     |  45  ;*
*; A#2   |  116.541   | 114-120   | 51-57     |  46  ;*
*; B2    |  123.471   | 121-127   | 58-64     |  47  ;*
*; C3    |  130.813   | 128-134   | 65-71     |  48  ;*
*; C#3   |  138.591   | 135-142   | 72-79     |  49  ;*
*; D3    |  146.832   | 143-151   | 80-88     |  50  ;*
*; D#3   |  155.564   | 152-160   | 89-97     |  51  ;*
*; E3    |  164.814   | 161-169   | 98-106    |  52  ;*
*; F3    |  174.614   | 170-179   | 107-116   |  53  ;*
*; F#3   |  184.997   | 180-190   | 117-127   |  54  ;*
*; G3    |  195.998   | 191-201   | 128-138   |  55  ;*
*; G#3   |  207.652   | 202-213   | 139-150   |  56  ;*
*; A3    |  220       | 214-226   | 151-163   |  57  ;*
*; A#3   |  233.082   | 227-240   | 164-177   |  58  ;*
*; B3    |  246.942   | 241-254   | 178-191   |  59  ;*
*; C4    |  261.626   | 255-269   | 192-206   |  60  ;*
*; C#4   |  277.183   | 270-285   | 207-222   |  61  ;*
*; D4    |  293.665   | 286-302   | 223-239   |  62  ;*
*; D#4   |  311.127   | 303-320   | 240-257   |  63  ;*
*; E4    |  329.628   | 321-339   | 258-276   |  64  ;*
*; F4    |  349.228   | 340-359   | 277-296   |  65  ;*
*; F#4   |  369.994   | 360-380   | 297-317   |  66  ;*
*; G4    |  391.995   | 381-403   | 318-340   |  67  ;*
*; G#4   |  415.305   | 404-427   | 341-364   |  68  ;*
*; A4    |  440       | 438-453   | 365-390   |  69  ;*
*; A#4   |  466.164   | 454-480   | 391-417   |  70  ;*
*; B4    |  493.883   | 481-508   | 418-445   |  71  ;*
*; C5    |  523.251   | 509-538   | 446-475   |  72  ;*
*; C#4   |  554.365   | 539-570   | 476-507   |  73  ;*
*; D5    |  587.33    | 571-604   | 508-541   |  74  ;*
*; D#4   |  622.254   | 605-640   | 542-577   |  75  ;*
*; E5    |  659.255   | 641-678   | 578-615   |  76  ;*
*; F5    |  698.457   | 679-719   | 616-656   |  77  ;*
*; F#5   |  739.989   | 720-761   | 657-698   |  78  ;*
*; G5    |  783.991   | 762-807   | 699-744   |  79  ;*
*; G#5   |  830.609   | 808-855   | 745-792   |  80  ;*
*; A5    |  880       | 856-906   | 793-843   |  81  ;*
*; A#5   |  932.328   | 907-960   | 844-897   |  82  ;*
*; B5    |  987.767   | 961-1017  | 898-954   |  83  ;*
*; C6    |  1046.502  | 1018-1078 | 955-1015  |  84  ;*
******************************************************


MIDILIST    DC.B        36,36,36,36,37,37,37,37,38,38,38,38,39,39,39,39
            DC.B        39,40,40,40,40,41,41,41,41,41,42,42,42,42,42,42
            DC.B        43,43,43,43,43,44,44,44,44,44,44,45,45,45,45,45
            DC.B        45,46,46,46,46,46,46,46,47,47,47,47,47,47,47,48
            DC.B        48,48,48,48,48,48,49,49,49,49,49,49,49,49,50,50
            DC.B        50,50,50,50,50,50,50,51,51,51,51,51,51,51,51,51
            DC.B        52,52,52,52,52,52,52,52,53,53,53,53,53,53,53,53
            DC.B        53,53,54,54,54,54,54,54,54,54,54,54,54,55,55,55
            DC.B        55,55,55,55,55,55,55,55,56,56,56,56,56,56,56,56
            DC.B        56,56,57,57,57,57,57,57,57,57,57,57,57,57,57,58
            DC.B        58,58,58,58,58,58,58,58,58,58,58,58,58,59,59,59
            DC.B        59,59,59,59,59,59,59,59,59,59,59,60,60,60,60,60
```

```
DC.B        60,60,60,60,60,60,60,60,60,61,61,61,61,61,61,61
DC.B        61,61,61,61,61,61,61,61,61,62,62,62,62,62,62,62
DC.B        62,62,62,62,62,62,62,62,62,62,63,63,63,63,63,63
DC.B        63,63,63,63,63,63,63,63,63,63,63,64,64,64,64,64

DC.B        64,64,64,64,64,64,64,64,64,64,64,64,64,64,65,65
DC.B        65,65,65,65,65,65,65,65,65,65,65,65,65,65,65,65
DC.B        65,66,66,66,66,66,66,66,66,66,66,66,66,66,66,66
DC.B        66,66,66,66,66,66,67,67,67,67,67,67,67,67,67,67
DC.B        67,67,67,67,67,67,67,67,67,67,67,67,67,68,68,68
DC.B        68,68,68,68,68,68,68,68,68,68,68,68,68,68,68,68
DC.B        68,68,68,68,69,69,69,69,69,69,69,69,69,69,69,69
DC.B        69,69,69,69,69,69,69,69,69,69,69,69,69,69,70,70
DC.B        70,70,70,70,70,70,70,70,70,70,70,70,70,70,70,70
DC.B        70,70,70,70,70,70,70,70,71,71,71,71,71,71,71,71
DC.B        71,71,71,71,71,71,71,71,71,71,71,71,71,71,71,71
DC.B        71,71,71,72,72,72,72,72,72,72,72,72,72,72,72,72
DC.B        72,72,72,72,72,72,72,72,72,72,72,72,72,72,72,72
DC.B        72,73,73,73,73,73,73,73,73,73,73,73,73,73,73,73
DC.B        73,73,73,73,73,73,73,73,73,73,73,73,73,73,73,73
DC.B        74,74,74,74,74,74,74,74,74,74,74,74,74,74,74,74

DC.B        74,74,74,74,74,74,74,74,74,74,74,74,74,74,74,74
DC.B        74,75,75,75,75,75,75,75,75,75,75,75,75,75,75,75
DC.B        75,75,75,75,75,75,75,75,75,75,75,75,75,75,75,75
DC.B        75,75,75,75,75,76,76,76,76,76,76,76,76,76,76,76
DC.B        76,76,76,76,76,76,76,76,76,76,76,76,76,76,76,76
DC.B        76,76,76,76,76,76,76,76,76,76,77,77,77,77,77,77
DC.B        77,77,77,77,77,77,77,77,77,77,77,77,77,77,77,77
DC.B        77,77,77,77,77,77,77,77,77,77,77,77,77,77,77,77
DC.B        77,77,78,78,78,78,78,78,78,78,78,78,78,78,78,78
DC.B        78,78,78,78,78,78,78,78,78,78,78,78,78,78,78,78
DC.B        78,78,78,78,78,78,78,78,78,78,78,79,79,79,79,79
DC.B        79,79,79,79,79,79,79,79,79,79,79,79,79,79,79,79
DC.B        79,79,79,79,79,79,79,79,79,79,79,79,79,79,79,79
DC.B        79,79,79,79,79,79,79,79,80,80,80,80,80,80,80,80
DC.B        80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80
DC.B        80,80,80,80,80,80,80,80,80,80,80,80,80,80,80,80

DC.B        80,80,80,80,80,80,80,81,81,81,81,81,81,81,81,81
DC.B        81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81
DC.B        81,81,81,81,81,81,81,81,81,81,81,81,81,81,81,81
DC.B        81,81,81,81,81,81,81,81,81,82,82,82,82,82,82,82
DC.B        82,82,82,82,82,82,82,82,82,82,82,82,82,82,82,82
DC.B        82,82,82,82,82,82,82,82,82,82,82,82,82,82,82,82
DC.B        82,82,82,82,82,82,82,82,82,82,82,82,82,82,83,83
DC.B        83,83,83,83,83,83,83,83,83,83,83,83,83,83,83,83
DC.B        83,83,83,83,83,83,83,83,83,83,83,83,83,83,83,83
DC.B        83,83,83,83,83,83,83,83,83,83,83,83,83,83,83,83
DC.B        83,83,83,83,83,83,84,84,84,84,84,84,84,84,84,84
DC.B        84,84,84,84,84,84,84,84,84,84,84,84,84,84,84,84
DC.B        84,84,84,84,84,84,84,84,84,84,84,84,84,84,84,84
DC.B        84,84,84,84,84,84,84,84,84,84,84,84,84,84,84,84

DC.B        84,84
```