Project Number: HGR-1000

A MICROPROCESSOR BASED AUDIO FILTER

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Michael J. Pender

Robert J. Ursillo

Date: December 14, 1990

Approved:

Professor Hemant G. Rotithor, Project Advisor

ABSTRACT

An audio filtering system that removes noise from old recordings, can be used on old master recordings to create new compact disc (CD) masters. A CD master would have little distortion and would be more impervious to damage than other recording mediums. One approach to creating such a comprehensive filter is to allow for human interaction with a microprocessor based filter, which utilizes a computer's logic and speed, and human intelligence.

This project involves the design and implementation of a microprocessor based system that can interface to a music playback system. The project includes various software features that allow a user to edit the music samples spaced ten microseconds apart. With this type of system it is possible to remove most of the imperfections present in a piece of music, and then record the edited music onto a CD, where it can enjoy a theoretically infinite life.

TABLE OF CONTENTS

CHAPTER 1 - INTRODUCTION

Unlike a vinyl record which has a needle bearing down on its grooves and wearing away the medium, or a magnetic cassette tape which has a magnetic sensing device pressing down on it while the tape is moving causing irreparable bending and stretching of the medium, the compact disc (CD) has the distinct advantage of not being physically touched or altered when being played back. The CD is read by a laser beam and a sensor which detects 1's and 0's on the disc. The only objects that come in contact with the CD are harmless photons. Because of this, the life of a properly handled CD is theoretically infinite. Many older recordings are now being placed on CD's to better preserve the music on them, but two problems exist with this process. The first problem is that some of the older masters no longer exist, and the transfer of these recording to CD must be made from worn vinyl records. The second problem is that the masters were recorded on magnetic tape, a medium with inherent defects.

The types of analog filters available today cannot remove, or even minimize many of the defects found in older recordings. Also many of the defects are not consistent enough for a machine of reasonable complexity to filter out. A differentiating-integrating filter was made for comparison to the microprocessor system. But the process of differentiating magnifies high frequency noise, thereby introducing new audible noise signals. A filtering approach using a microprocessor system with human interaction is a possible approach to overcoming this difficulty.

The objective of this project is to interface a personal computer with an audio sampling and playback device. The computer samples an audio signal (i.e., records the amplitude of the output as a digital signal) at a frequency above the audible range in order to avoid any loss of sound quality. The samples are stored in the computer memory. A person can access the memory, and edit any of the audio signal samples, or apply the transfer function of any analog or digital filter to the data. The sample editing functions result in an audio signal output containing any alterations that the user desires, to a time frame as small as 10 microseconds. Various software tools can also aid the listener in the recovery and removal of unwanted signals. The resulting edited music can then be output to either audio components or a CD recording device.

The solution proposed and implemented in this project can be summarized as follows:

- At the input there is a voltage matching filter which allows for input from a CD player or any other audio signal generating device.
- The amplitude of the analog signal is converted to a digital signal by an analog to digital converter at a sampling rate of 100,000 samples per second.
- Each of these digital signals is stored in successive memory locations.
- The memory containing this digital data can be accessed and altered by the user.

- The data in the memory can then be converted back into an analog signal by a digital to analog converter.
- The signal is then sent through an analog filter which removes high frequency step noise created by the digital to analog process. Step noise is the flat areas on the output curve that occur between output samples.
- This filter can be connected to any audio device the user chooses to either listen to or record the music.

CHAPTER 2 - BACKGROUND

Before deciding whether a microprocessor based audio filter had practical applications, research had to be done on the types of noise found in old recordings, and on the varieties of filters that are available today.

2.1 DESCRIPTIONS OF NOISE IN RECORDINGS

Initial research had to be done to discover whether or not a microprocessor based filter could remove the noise found in old recordings. In order to determine this it was necessary to look at the kinds of noise found on old recordings. There are two major types.

2.1.1 DEFECTS IN VINYL RECORDS

In the case of records the most noticeable defect is a "pop". "Pops" occur is older worn records where pits in the medium cause the needle to stick momentarily. Several recordings of these "pops" were made and studied on a sample and hold oscilloscope (an oscilloscope which can hold and repeat the sample). The wave-form associated with these "pops" is a rectangular wave with significantly high amplitude. The duration (and thus frequency) of these rectangular waves varied greatly, and many were within the same frequency range as the music. A simple frequency based filter, which can only filter out a particular frequency or range of frequencies (not particular wave-forms) would therefore be useless.

2.1.2 TAPE HISS

Several blank magnetic cassette tapes (all of which produced tape hiss) were observed with a sample and hold oscilloscope. All of the tapes produced a similar waveform which was noise at or around the 1,000 hertz range.

2.2 CURRENT NOISE REDUCTION AND REMOVAL METHODS

Before determining the usefulness of a microprocessor based filter, it was necessary to determine whether or not the analog audio filters currently available could do a satisfactory job of removing the noise found on old recordings. There are two major types of analog audio filtering available today.

2.2.1 FREQUENCY FILTERING

A simple method of noise reduction that allows for listener control is that of frequency filtering through the use of an equalizer. The listener can choose the gain level for various frequency ranges by simply adjusting the controls of an equalizer. This method, however, does not selectively filter out unwanted wave-forms, and it cannot constantly adjust to its input. Because of these problems, only limited editing can be done with an equalizer.

2.2.2 DOLBY NOISE REDUCTION

Dolby noise reduction is designed for a specific purpose: eliminating tape hiss from magnetic cassette tape. Dolby noise reduction removes tape hiss upon playback of a magnetic cassette tape by ignoring the 1,000 to 3,000 hertz signal being output by the tape [1]. In this way no tape hiss (which is in this frequency range) will be sent to the output.

When recording, the Dolby system shifts all of the music at or above the 1,000 hertz frequency up about 2,000 hertz. Upon playback it ignores all signals in the 1,000 to 3,000 hertz range that are output by the tape, and shifts all the signals from 3,000 hertz or above down by 2,000 hertz and sends it to the output.

This system is effective for removing tape hiss, but this is only a very limited application. Even with regard to this single application, however, there is the drawback that when signals are shifted up 2,000 hertz, the higher frequencies of the music are sometimes clipped by the limited dynamic frequency range of the tape it is being recorded on.

CHAPTER 3 - MICROPROCESSOR BASED FILTER APPROACH

Because of the specialized and limited applications of the current types of noise reduction filters, there is a need for a more comprehensive and flexible type of filtering, one that can filter out any noise that is unpleasant to the human ear regardless of its frequency or wave-form. A possible way to filter out everything that sounds bad to a person, is to allow a person to be an interactive part of the filtering system. A microprocessor based design allows for this type of interaction.

3.1 MOTIVATION

The various advantages of the microprocessor based filter, which provided the motivation for designing and implementing it, are described in this section.

3.1.1 HUMAN INTERACTION

A microprocessor based filter has the distinct advantage of being able to post-process a signal after sampling. That is, by saving a portion of the audio signal as it is being played, a person wishing to edit an improper piece of music can pause the audio signal and search through the memory (with the help of various software routines) to find the section of music to be edited. An analog filter can merely influence the signal according to the transfer function of the circuit. The post-processing feature allows for human interaction to alter the audio signal to suit the human ear, rather than simply configure it to a the specifications of a set of unemotional transfer functions. This is important because in the end the human ear is going to listen to the signal.

3.1.2 NON-INTERACTIVE FILTERING

If human interaction is not desired or is too cumbersome, the microprocessor based filter still has the advantage of being able to run any signal processing routine that one may wish to implement, such as the Fourier transform analysis and editing that is described in chapter 5.

3.2 PROBLEMS

The microprocessor based filter is not totally without flaws. The problems associated with this kind of filter are described in this section.

3.2.1 SAMPLING SPEED

The most significant problem inherent in a microprocessor based filter is that of sampling speed. Sampling speed is the rate at which the system can record the amplitude of an analog signal in terms of digital samples per second. An analog filter has a continuous signal as an input, and a continuous signal as an output. A sampling filter can only sample an input signal at a fixed maximum sampling rate. Restrictions on this speed are imposed by the conversion time (i.e., the time it takes for an analog signal to be converted into a digital signal) of the analog to digital converter (ADC), the setting time (i.e., the time it takes for the analog output to settle at the proper voltage) of the digital to analog converter (DAC), and the processing speed of the computer. However, the present day speeds of computers and the ADC and DAC chips are such that regeneration of the signal is so accurate that the human ear cannot distinguish between a continuous signal and the sampled and filtered output.

3.2.2 MEMORY CAPACITY

A second problem, which is more of a logistical nature than a technical one, is that of memory size. At 100,000 samples per second and an eight bit sample size (representing the amplitude of the sample) one megabyte of memory will be occupied by a twenty second music sample. If an average song is assumed to be four minutes, a 12 megabyte space of memory must be allocated per song. This problem is overcome by either allotting the 12 megabytes of memory, or by playing the music through at the same time it is being stored in a memory window. In this way, the user stops the music and the storage routine when noise is encountered. The noise that the user wishes to remove will be stored in memory with music on either side of it. While in this paused mode, the user can edit the noise stored in memory before restarting the system and sending that piece of the music to the recording device. The problem with this method is that it is very difficult to synchronize the device creating the original audio signal with the computer and the device recording the output audio signal when pausing the music to do editing. All of the devices (playback device, sampler, and recorder) must come to a complete stop at exactly the same time to avoid introducing any unwanted signals into the system.

3.3 DECISION

Based on the previous discussion, it was decided that the sampling speed of the system was sufficient, and that a twenty second music sample would be enough for testing purposes (it is assumed that more memory could have been bought, but for test purposes and limited funds this would suffice). The aforementioned benefits of the microprocessor based system greatly outweighed any drawbacks associated with it, therefore the decision to implement this type of filtering system was made.

3.4 BLOCK DIAGRAM

Each of the following descriptions corresponds to one of the blocks in the block diagram shown in figure 1.

INPUT SIGNAL:

The input signal can be generated by any audio playback device. The signal is then sent to an input voltage matching filter.

INPUT VOLTAGE MATCHING FILTER:

The input voltage matching filter has two functions. The first is to rescale voltages being output by an audio output

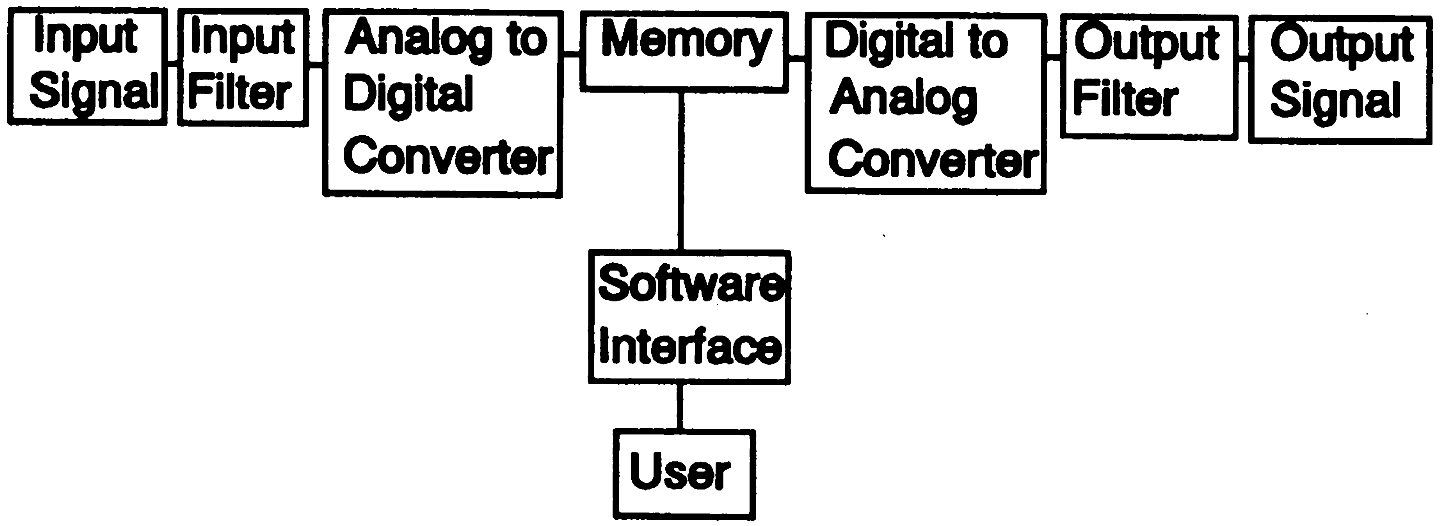


Figure 1: Block Diagram of Microprocessor Filter System

device, such as a phonograph or a CD player, to a range of +/- 5 volts that can be used by the analog to digital converter. The second is to remove the problem of aliasing by using a two pole low pass analog filter with a roll off at 20,000 hertz [2]. Aliasing occurs when the frequency of a signal is greater than half the sampling rate. The sampling device views all of the high frequency peaks as a continuous signal, and upon regeneration of the signal, would output several of these high frequency peaks as a single low frequency pulse. Though sounds at or above 20,000 hertz are not noticeable to the human ear in their original state, an aliasing problem would occur and create an audible low frequency signal if the high frequency signals were not filtered out.

ANALOG TO DIGITAL CONVERTER: (ADC)

The ADC is a single chip that is clocked at a rate of 100,000 times per second. When clocked, it tracks, holds, and converts the voltage at its input, into an eight bit digital signal. The digital signal is then sent to the computer.

COMPUTER INTERFACE:

The computer stores the digital signals that constitute a song in sequential order in the RAM (random access memory) space allocated. The next stage is optional.

SOFTWARE INTERFACE AND EDITING:

The user can select various regions of memory for either display or playback, zeroing in on the location containing the sound to be edited. Once located, the user can either implement a software editing feature, or can manually alter any of the bytes desired. This editing feature allows for alternating between loop playback and editing until the desired sound quality is attained.

DIGITAL TO ANALOG CONVERTER: (DAC)

The DAC is a single chip that is connected to the computer's data bus through latches. At a rate of 100,000 samples per second (to match the rate at which the data was sampled) the computer puts an eight bit digital signal on the data bus and then the latches. When the latches are clocked, the digital data is sent to the DAC chip which changes the digital data into a voltage that is subsequently sent to an output voltage matching filter.

OUTPUT VOLTAGE MATCHING FILTER: (OVMF)

The output voltage matching filter serves two purposes. The first is to eliminate as much of the step noise generated by the digital to analog conversion as possible. Though the step noise may not be perceptible to human ears, the high frequency and possibly high magnitude (if amplified to a great extend) may prove harmful to audio equipment such as amplifiers and speakers. Therefore the OVMF is a necessary stage. This filter is a two pole low pass analog filter with roll off at 20,000 hertz [3]. After this stage, the signal output can be handled by normal audio components such as an amplifier. The second purpose of the filter is to rescale the output voltage to +/- 2 volts RMS so that it may be used by normal audio devices such as amplifiers.

CHAPTER 4 - SYSTEM HARDWARE

This chapter gives an overview of the components used in the design and explains why they were chosen.

4.1 JUSTIFICATION AND DESCRIPTION OF COMPONENTS

LS574 LATCH CHIP

The LS574 latch chip is a high speed CMOS, eight bit, level sensitive latch device. This chip was chosen because the system uses an eight bit bus. It was necessary to use a latch that is level sensitive, as opposed to edge triggered, because the data lines are not set prior to the latched being clocked, but at the same time the latch is clocked. Another factor which was considered when choosing this chip was the pin layout, which places all of the input pins on one side of the chip. This pin configuration allowed for a more manageable wiring layout than any of the other chips considered.

LS374 LATCH CHIP

The LS374 latch chip is a low cost, eight bit, level sensitive, latch device with tri-state outputs. This chip is only used to latch a single bit, but it was necessary to use a chip with tri-state output (so it would not try to place data on the bus when the chip is not active). This chip was chosen for its tri-state output and low cost.

MICRO NETWORKS DAC80 DIGITAL TO ANALOG CONVERTER

The DAC80 is a low cost, twelve bit, digital to analog converter. The chip had a low enough logic current draw on the bus lines (from -180 to 20 microamps) to be driven by CMOS logic latch chips. The chip also had a sampling rate well within the specifications needed for audio reproduction (ie., 20,000 hertz) at 100,000 samples/second. The chip's +/- 5 volt output range also met the voltage requirements necessary for audio input, which is +/- 2 volts RMS. This chip also has the advantage of being a twelve bit high precision device which would facilitate a twelve bit oversampling function if the project were further developed. The only feature that the DAC80 did not possess, that would have made the design simpler, was the presence of built in latches, a feature on some higher priced models.

MICRO NETWORKS MN6231 ANALOG TO DIGITAL CONVERTER

The MN6231 is a low cost, twelve bit analog to digital converter. This chip configured to the 100,000 samples/second that was specified for this project. The MN6231 also worked within the +/- 2 volt RMS necessary for audio signals with a range of +/- 5 volts. This chip also had the advantage of being a twelve bit high precision device which would facilitate a twelve bit over sampling feature if the project was further developed.

4.2 DESIGN NOTES

For the construction of the circuits required in this MQP many approaches were considered. Breadboarding is well suited for circuit testing, but is inappropriate for a permanent circuit. The first attempt made was to take a board specifically designed for interfacing to an Apple computer and to mount the chips on it. Small boards would be used to mount groups of chips, and wire would be used to make the connections between boards. Before the circuit was half wired, however, there was a rat's nest of connections. Debugging such a circuit would be impractical. The next option considered was wire wrapping the circuits on a board. Wire wrapping, however, would have made an equally unmanageable rat's nest, and this technique is more suited to temporary designs. As an MQP, a project is expected to be held intact for possible use in the future. Further, the specifications for the A/D and D/A chips specifically recommend against wire wrapping to obtain the specified accuracy. Wire wrapped boards tend to pick up stray signals easily, especially when a circuit contains both digital and analog sections. The best option available was to have a custom printed circuit board generated for the circuit. To send a design out for such service would have cost hundreds of dollars for even a simple design. Because of this, using the facility at WPI for creating such boards was a viable option. The facility is located in the rear of the EE shop. With the help of Paul Nader, the process of how to manufacture the boards was soon learned. The complete process is described in detail in Appendix A.

CHAPTER 5 - SYSTEM SOFTWARE DESIGN

Various filtering functions of this system are implemented in software. This chapter includes a description of the interface between the software and hardware, the storage and retrieval algorithms used, a functional description of the software modules, and a software description using pseudocode.

5.1 FUNCTIONAL DESCRIPTION

Most of the source code for the software is contained in the file /mqp.c. This code is written in the HyperC Prodos language, but should be portable to other C environments with little effort. The first section of the file includes standard headers for file input and output, global variables which contain information about the data image on the desktop, and hardware addresses for accessing the specific configuration of the computer being used.

The first module is main, which accesses most functions of the filter system. All functions written in C, or 6502 assembly are called from main, or one of its subroutines. Analysis mode is not called from main, as the analysis functions are not written in HyperC Prodos, but in Applesoft BASIC. Control must be manually passed from the C system to the BASIC system. Therefore, the Analysis functions are run independently of main and its support functions. This triple language implementation makes the best use of the computer's various resources. Assembly is best suited to doing high speed device level operations, such as interfacing the memory card and the microprocessor filter. C interfaces easily to assembly, and provides a modular system in which to develop a large software project. Applesoft provides simple access to graphics routines, and makes use of floating point functions convenient.

Additional benefits are derived by not requiring all of code to occupy memory at the same time. The C program uses most of the 64 Kilobytes of available program memory for code space. The assembly routines use all of the one megabyte of expansion memory. The BASIC FFT routines require large arrays. The arrays use the 64 Kilobytes program space. The FFT routines also require extensive disk access, which is done from the 64 Kilobyte, slot three ramdisk.

5.1.1 MAIN

The bulk of the software code is written in Hyper C Prodos. The software is mostly menu-driven, beginning in the module main. Main presents the user with a menu of all major sub-functions, and executes the one selected.

5.1.2 INSTRUCTIONS

The purpose of this routine is to provide the user with some general information about the microprocessor-filter system. This is done by opening and reading from a text file called /mqp/mqp.helpfile, and printing the information to the screen.

5.1.3 SAMPLE

The *sample* routine is also called by *main*. This routine issues a message to the user that a key must be pressed to begin sampling, then passes control to *fastsample*.

5.1.4 FASTSAMPLE

Fastsample is an assembly language routine. Its function is to read successive values from the A/D port, and store them in memory as fast as it can, until all available expansion memory is full. A small code segment at the beginning of this routine causes it to wait before starting, ensuring that the user is ready to begin.

5.1.5 PLAYBACK

Playback gets the starting address, ending address, and speed information required to recreate a segment of the data stored on the desktop. Then it passes control to *play8*. The starting and ending addresses define the segment of data to be played, and the speed is a period multiplier between successive outputs. This means that a speed of '1' indicates playing back at the speed at which the data was sampled. A speed of '2' means to use twice the delay between samples, playing back at half the speed at which the original was sampled. The maximum delay between samples is selected by choosing speed = '0', which pauses for a delay 256 times the delay between original samples.

5.1.6 PLAY8

Play8 copies data from memory to the D/A chip. The starting address, ending address, and speed are set in playback before play8 is called. The routine repeats the segment until a key is pressed, when the loop is terminated. *Play8* is timed to reproduce data at the same rate as *fastsample*. This means that a speed value of '1' will cause playback to reproduce data at the same rate at which it was recorded. A speed value of '2' generates output at half the original speed it was recorded, etc.

5.1.7 DISPLAY

The *display* routine allows the user to view large segments of eight bit sampled data in hexadecimal form. A starting point in memory is selected by the user, the data is then displayed one page at a time. Successive pages of 384 bytes display sequential segments of the megabyte expansion memory.

5.1.8 ANALYSIS

The *analysis* function first informs the user of the steps necessary to transfer control to the external analysis routines. Next an option is presented to return to the main menu. If the user selects to continue with the procedure, the user is prompted for a starting address. This address is then loaded into the card, for use by the analysis routines. This address is used as a starting point by *fft*, and *plot*.

5.1.9 EDITOR

The *editor* function prompts the user for a starting address, then displays the contents of sixteen bytes in that area of memory. Data may then be altered byte by byte, allowing manual editing of sampled data. This is the feature that separates the microprocessor based filter from a standard analog filter. The user may individually alter the sampled data. With the system operating at full speed, one second of signal is stored in 40K of memory. An error, such as a pop noise, may be removed by replacing the error data with bytes which represent the desired performance. The desired performance is chosen by the user by manually editing the bytes in memory until the noise can no longer be heard.

5.1.10 SYSTEM

The *system* function allows the user to load or save audio image data from any prodos compatible disk device. Data is stored in an uncompressed format, allowing reproduction with no loss of the sampled data. The file format consists of the speed at which the data was recorded, and a sequential listing of the actual performance data in an eight bit format. When saving data from the desktop to a disk, the user may select the starting and end points of the data segment. When loading data from disk to the desktop, the information is copied into the expansion memory at the beginning of ram space by default.

5.1.11 TEST

To ensure proper function of the system it is helpful to have standard wave-forms available for testing purposes. The *test* routine offers the user the option to load a triangular, square, or sinusoidal waveform into the card. The ability to generate predetermined wave-forms also allows a user to check for correct output from whatever external devices may be connected, without interrupting the system. This feature is helpful for isolating faults or signal loss in the output stage.

5.1.12 STARTUP

When the BASIC.SYSTEM environment is entered, the computer looks for and attempts to execute a file named STARTUP. By choosing this name for the file it is run automatically. *Startup* offers the user a menu of the different analysis functions written in BASIC. At the user's selection the computer will execute the *fft*, *plot* or *filter* function, or return to the main menu in the C.SYSTEM.

5.1.13 FFT

A Fast Fourier Transform (FFT) is a mathematical process that decomposes a set of sampled data into harmonic frequencies. An FFT is a reversible process, the harmonic frequencies may be recombined to recreate the data of the original sample. When *fft* is called from the *startup* menu the user may select to execute an FFT of 32 to 1024 points on the data segment chosen in the *analysis* routine of mqp.c. The FFT routine used was created by Oppenheim and Schaffer [4]. After the FFT data is computed it is written to the file /RAM/FFT.DAT.

5.1.14 PLOT

The *plot* routine offers the user the option to plot several different types of information to the high resolution (hi-res) screen. Options for plotting include viewing various ranges of the data in memory, FFT data, or a reconstruction of the original data from recombining the FFT data, as described in 5.1.13. If the user selects a plot of original data, the hi-res screen is scaled automatically to use the full width of the video monitor. FFT data is read from the file /RAM/FFT.DAT and plotted at one eighth of the height of the screen by magnitude, phase, and magnitude with phase. A second option allows viewing magnitude with phase data, which allows viewing at half of the height of the screen. To reconstruct the sampled data from the FFT, data is read from disk, then an Inverse Fast Fourier Transform is performed on the data [5].

5.1.15 FILTER

Before calling the *filter* function the user must execute the *fft* routine to convert a segment of the sampled data to frequency components. The *filter* function allows the user to eliminate all frequency components of the sampled data below a certain magnitude. The "pop" type noises were found to appear as a rectangle wave, which *fft* decomposes into a primary harmonic and associated harmonics of lesser magnitude. The frequency components of tape hiss appear to be an infinite number of low magnitude frequency components. An approach to filtering that responds differently to true frequencies than noise can effectively increase the signal to noise ratio. Therefore, a copy of a recording may be made that sounds more pleasing to the user than the original. This filter is non-linear in its frequency response, as frequency components of a magnitude above the threshold level will be unaffected, while components below the threshold level will be reduced to zero magnitude.

5.2 THE MICROPROCESSOR FILTER INTERFACE

Software interface with the microprocessor filter board is made through functions written in assembly language, and linked to the software written in the C language. Assembly read/write calls are used to directly access the analog to digital (A/D) and digital to analog (D/A) converter chips on the microprocessor filter board. A segment of eight addresses in the range $COFO - $COF7 are used to control both the A/D and D/A chips. The Apple computer used has an eight bit data bus. To read or write twelve bit data provided by the A/D and D/A used requires two separate read/write cycles to access all data bits.

A Micronetworks MN6231 chip is used as the analog to digital converter. To acquire a sample the computer makes a write access to address $COF3. This access strobes the MN6231 to take a sample. The MN6231 requires ten to fifteen microseconds to complete its determination of the digital value of a sample. The computer used is capable of storing values in memory at more than twice the rate at which the converter can produce them. The MN6231 provides a STATUS line which remains high until the sampled data is ready. This line is available for a read access to software as bit seven of the address $C0F0. The STATUS line is polled, and delay cycles are executed as long as the line is high. After the STATUS line drops low, the data is read from address $C0F2. The data is then written to the expansion memory card. For sequential read or write access the card automatically increments the memory location pointer. This feature helped to simplify the software, making it unnecessary to maintain pointers to the current memory location. Reducing the number of instructions necessary helped increase execution speed of sampling, playback and disk storage routines, since less instructions needed to be executed to access each specific byte of data on the expansion card.

A Micronetworks DAC80 chip was used as the digital to analog converter. The DAC80 was indirectly controlled through two eight bit latch chips at addresses $C0F4 and $C0F6. Acoustic data is read from the memory expansion card and written to the latches. Once again the auto increment feature of the memory card allows simplifications in the software. The DAC80 chip has a two microsecond response time for full scale deflection from -5 V to +5 V, so a software delay was introduced to generate playback at the same speed as data was sampled. An optional delay loop was also added to the software to allow altering the playback speed. Additionally, playback may also be repeated as often as desired. This feature makes it simple to observe part of a sample on an oscilloscope, or to search for the occurrence of a pop or other noise.

5.3 SOFTWARE PSEUDOCODE

The pseudocode for each major function is presented in this section. Pseudocode descriptions of functions are grouped with other functions written in the same language.

5.3.1 HYPER C PRODOS (/MQP/MQP.C)

These routines handle most of the functions of the Microprocessor filter. All routines including those written in Assembly and Applesoft BASIC return to *main()*.

```
main()
{
    char done, key;
    Set80ColumnText();
    ClearTheScreen();
    IssueWarningAboutRamdisk();
    if(AskShouldContinue() == No)
    {
        Exit();
    }
    done = no;
    while(not done)
    {
        ClearTheScreen();
        IssueWarningAboutLockup();

        /* Microprocessor system may lock up if printed circuit
           board is not connected to the computer's bus */

        key = Menu()
        {
           1  View instructions;
           2  Sample mode;
           3  Playback mode;
           4  Display mode;
           5  Analysis;
           6  Editor mode;
           7  Operating system functions;
           8  Load test data;
           0  Exit program;
        }
        switch(key)
        {
            case '1':
                    instructions();
                    break;
                case '2':
                    sample();
                    break;
                case '3':
                    playback();
                    break;
                case '4':
                    display();
                    break;
                case '5':
                    analysis();
                    break;
                case '6':
                    editor();
                    break;
                case '7':
                    system();
                    break;
                case '8':
                    test();
                    break;
                case '0':
                    ClearTheScreen();
                    done = yes;
            }
        }
}

instructions()
{
    char    filename[20];
    char    c, p, count;
    FILE    fp;
    filename = FileOfInformationAboutSystem;

    /* The actual filename is system-implementation dependent */

    cnt = 0;
    fp = open (filename);
    ClearTheScreen;
    while ((c = GetCharFromFile(fp)) != ErrorReadingData)
    {
        if (c == CarriageReturn AND ++count == HeightOfScreen)
        {
            p = GetKeyPressed();
            if (p == Quit)
            {
                PrintCarriageReturn();
                Return;
            }
            cnt = 0;
        }
        if (IsPrintable(c))
            PrintCharacter(c);
    }
    close(fp);
    WaitForReturnKeypress;
}

sample()
{
    char key;
    ClearTheScreen();
    PrintMessage("Executing 8 bit sample mode");
    PrintMessage("Press any key to begin...");
    fastsample();
}

playback()
{
    char key, first, second;
    ClearTheScreen();
    PrintFunctionDescription();

    /* Issue instruction on how to use this function */

    GetStartAddress();
    GetFinishAddress();
    PrintByte(SPEED);
    if((first = GetHexValue()) != ReturnPressed)
        if((second = GetHexValue()) == ReturnPressed)
            SPEED = first;
        else
            SPEED = second + 16 * first;
    ClearTheScreen();
    PrintMessage("Executing Eight bit playback mode");
    play8();
}

display()
{
    char    done, key, temp;
    char    i, j;
    ClearTheScreen();
    PrintFunctionDescription();

    /* Issue instruction on how to use this function */

    GetStartAddress();
    InitializeRamCard();
    done = false;
    while( not done)
    {
        for(i = 0; i < 20; i++)
        {
            PrintAddress(RamCard);
            for(j = 0; j < 16; j++)
               PrintByte(DataOnCard);
            PrintCarriageReturn();
        }
        key = AskShouldContinue();
        if (key == No)  done = true;
        PrintCarriageReturn();
    }
}
```

analysis()

```
{
    ClearTheScreen();
    PrintFunctionDescription();

    /* Issue instruction on how to use this function */

    AdviseHowToUseSubfunctions();

    /* Issue instruction on how to pass control to the
       subfunctions */

    if(AskShouldContinue() == No)  Return;
    PrintCarriageReturn();
    GetStartAddress();
    PassControlToBASIC();
    Exit();
}
```

editor()

```
{
    char done, key, temp, first, second;
    char i, j;
    ClearTheScreen();
    PrintMessage("Editor mode");
    GetStartAddress();
    done = false;
    while(not done)
    {
        InitializeRamCard();
        SetAddress(START Bitwise_Anded_With 0xffff0);
        PrintAddress(RamCard);
        for(i = 0; i < 16; i++)
          PrintByte(DataOnCard);
        SetAddress(START);
        PrintAddress(START)
        PrintByte(DataOnCard);
        if((first = GetHexValue()) == ReturnPressed)
            Return;
        if((second = GetHexValue()) == ReturnPressed)
            temp = first;
        else
            temp = second + 16 * first;
        SetAddress(START);
        DataOnCard = temp;
        if(not (IncrementCardAddress) = OutOfMemory)
            done = true;
        }
}
```

system()

```
{
    char key;
    ClearTheScreen();
    key = Menu()
    {
          1    Save desktop data to file;
          2.   Load desktop data from file;
          0.   Exit to previous menu;
    }
    switch (key)
    {
        case '1' :
            GetStartAddress();
            GetFinishAddress();
            GetFilename();
            WriteFileToDisk();
            break;
        case '2' :
            GetFilename();
            ReadFileFromDisk();
            break;
    }
}
```

test()

```
{
    char key;
    static char sinusoid[32] = ImageOfSinWave;
    static char square[32]   = ImageOfSquareWave;
    static char triangle[32] = ImageOfTriangleWave;
    char *image;
    char i, j;
    ClearTheScreen();
    key = Menu()
    {
          1. Sinusoid data;
          2. Square wave data;
          3. Triangle wave data;
          0. Exit to previous menu;
    }
    PrintMessage("Loading data image...");
    switch(key)
    {
        case '1':
            image = sinusoid;
            break;
        case '2':
            image = square;
            break;
        case '3':
            image = triangle;
            break;
        case '0':
            Return;
    }
    PrintMessage("Loading 1024 data points into desktop...");
    InitializeRamCard();
    for(i = 0; i < 32; i++)
        for(j = 0; j < 32; j++)
            DataOnCard = DataPointedToBy(image + j);
    START = 0;
    FINISH = 1023;
    WaitForReturnKeypress();
}
```

5.3.2 6502 ASSEMBLY LANGUAGE (/MQP/MQP1.A)

These routines isolate the functions written in C from the details of interfacing directly to the filter system hardware. The routines have code included to allow them to work with the C system, as if they were C functions. After performing their task they return control to the calling function.

fastsample()

This routine directs the hardware card attached to perform an eight bit conversion. This routine instructs the MN6231 chip used to get a sample, then waits until the data is ready. Next the routine reads the data, and stores it on the memory expansion card at the current address. Even on its slowest loop the routine can store more than 100,000 samples per second. The converter chip used can only process up to 100,000 samples per second. This means the routine is faster than necessary. The routine is written using a wait loop that checks the status of the MN6231, and waits until the chip has completed acquiring a new sample. The routine loops until all expansion memory has been used, then returns control to the calling function.

play8()

This routine interprets the data on the memory card as eight bit audio data, and passes it to the DAC80 chip. References are made to global variables SPEED, START, and FINISH. SPEED controls the delay between successive writes of data to the DAC80 port. A SPEED value of one recreates output at the same rate at which the input signal was sampled. A SPEED value of two would recreate output at half the rate, etc. START and FINISH determine the first and last byte of data to be played.

5.3.3 APPLESOFT BASIC

These routines perform the various frequency analysis functions. These routines were written in BASIC because HyperC has poor floating point facilities, and no hi resolution graphics capability.

startup()

```
{
     char key, done;
     done = no;
     while(not done)
     {
          ClearTheScreen();
          key = Menu()
          {
               1. FFT analysis of data
               2. Plot utility
               3. Filter utility
               0. Return to main program
          }
          switch(key)
          {
               case '1':
                    fft();
                    break;
               case '2':
                    plot();
                    break;
               case '3':
                    filter();
                    break;
               case '0':
                    done = yes;
          }
     }
     ExitToMain();
}

fft()
{
     char done, key;

     GetRamPointer();
     ClearTheScreen();
     done = no;
     while(not done)
     {
          key = Menu()
          {
               1. 1024 point FFT
               2. 512 point FFT
               3. 256 point FFT
               4. 128 point FFT
               5. 64 point FFT
               6. 32 point FFT
               0. Exit
          }
          switch(key)
          {
               case '1':
                    NumberPoints = 1024;
                    break;
               case '2':
                    NumberPoints = 512;
                    break;
               case '3':
                    NumberPoints = 256;
                    break;
               case '4':
                    NumberPoints = 128;
                    break;
               case '5':
                    NumberPoints = 64;
                    break;
               case '6':
                    NumberPoints = 32;
                    break;
               case '0':
                    RestoreRamPointer();
                    ExitToStartup();
          }
          LoadArray(NumberPoints);
          PerformFFT(NumberPoints);
          WriteFFTData(NumberPoints);
     }
}
```

plot()

```
{
     char done, key;

     ProtectGraphicsPages();
     GetRamPointer();
     done = no;
     while(not done)
     {
          ClearTheScreen();
          key = Menu()
          {
               1. 1024 points
               2. 512 points
               3. 256 points
               4. 128 points
               5. 64 points
               6. 32 points
               7. Plot FFT data
               8. Only Mag. with phase
               9. View reconstruction
               0. Exit
          }
          switch(key)
          {
               case '1':
                    NumberPoints = 1024;
                    PlotData(NumberPoints);
                    Gridlines();
                    break;
               case '2':
                    NumberPoints = 512;
                    PlotData(NumberPoints);
                    Gridlines();
                    break;
                    case '3':
                         NumberPoints = 256;
                         PlotData(NumberPoints);
                         Gridlines();
                         break;
                    case '4':
                         NumberPoints = 128;
                         PlotData(NumberPoints);
                         Gridlines();
                         break;
                    case '5':
                         NumberPoints = 64;
                         PlotData(NumberPoints);
                         Gridlines();
                         break;
                    case '6':
                         Numberpoints = 32;
                         PlotData(NumberPoints);
                         Gridlines();
                         break;
                    case '7':
                         LoadFFTData();
                         PlotFFTData();
                         break;
                    case '8':
                         LoadFFTData();
                         PlotMagWithPhase();
                         break;
                    case '9':
                         LoadFFTData();
                         InverseFFT();
                         PlotData(FFTData);
                         ShouldReplaceDataOnCard();
                         break;
                    case '0':
                         done = yes;
               }
          }
          ExitToStartup();
     }

     filter()
     {
          int i;
          char threshhold;
          ClearTheScreen();
          PrintMessage("Threshold level? (0-255) ");
          threshhold = gethex();
          if(threshhold = NoChange) ExitToStartup();
          LoadFFTData();
          for (i = 1; i <= NumberPoints; i++)
               if (threshhold >  Magnitude(FFTData(i)))
                    FFTData(i) = ZeroMagnitude;
          WriteFFTData(NumberPoints);
```

```
        ExitToStartup();
}
```

CHAPTER 6 - EXPERIMENTS

Experiments were run on the microprocessor based system to determine how well it performed under various conditions. Both synthesized wave-forms and actual music were used in the experiments.

6.1 FREQUENCY RESPONSE

The following three frequency response plots indicate how the output of the system reacted to input. The type of input is stated at the top of each graph. In each case, the wave-form was fed into the system, stored in memory, and sent to the output without modification.

6.1.1 SINUSOIDAL WAVE

The system shows a flat frequency response to a sinusoidal wave, up to 5,000 hertz. Between 5,000 and 10,000 hertz the amplitude gain begins to drop. The multiple pole nature of the input and output voltage matching filters results in a flat area between 10,000 and 15,000 hertz. Beyond 15,000 hertz, the frequencies are sharply attenuated as shown in figure 2.

6.1.2 TRIANGLE WAVE

The frequency response to a triangle wave is similar to that of a sinusoidal wave up to 10,000 hertz. However, the filter responds inconsistently to the higher harmonic sinusoids which compose the triangle wave as shown in figure 3.

6.1.3 SQUARE WAVE

The response to a square wave is flat up to 10,000 hertz as expected. However, above 10,000 hertz overshoot results in the system digitizing the ringing of the input square wave. Ringing refers to the pattern of high frequency oscillation created at the beginning and end of a square wave when impedances are not perfectly matched as shown in figure 4.

6.2 TESTS ON ACTUAL MUSIC WITH NOISE

At this point the system was ready to be used for its intended purpose, removing noise from music. The two varieties of noise described in the beginning of the report were to be removed from the recording.

6.2.1 VINYL RECORD DEFECTS

The test sample was the same recording of "pops" used with the sample and hold oscilloscope. The tape was allowed to play until a "pop" was heard. The tape player and memory storage routine were both stopped. By changing the start and end memory locations for playback, the memory location in which the "pop" was stored was narrowed down. Using an oscilloscope, the memory area in which the "pop" was located was found. By using the software along with manual editing, the pop was reduced in the music until it could no longer be heard, or seen on the oscilloscope.

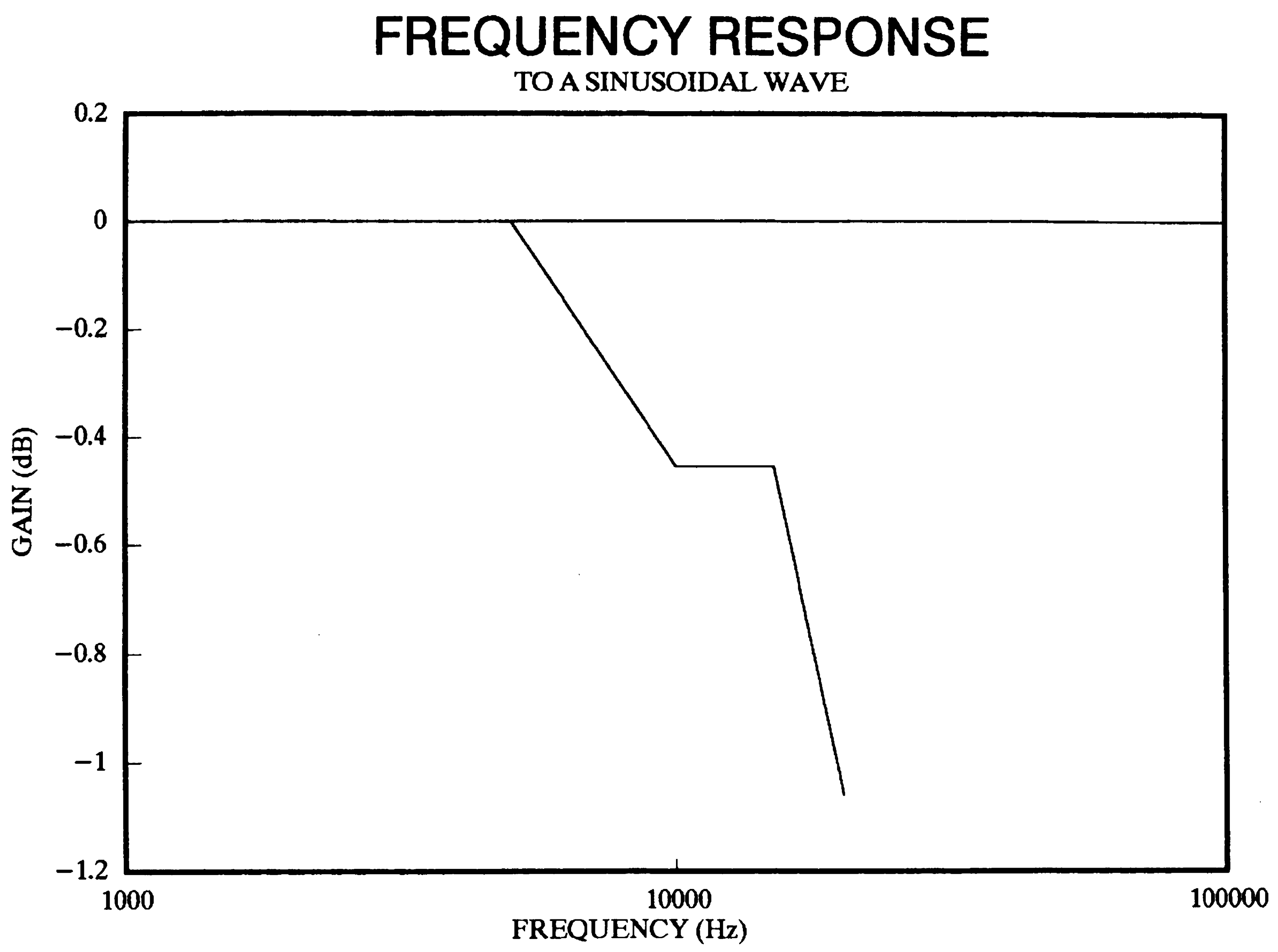


Figure 2: Frequency response of Microprocessor Filter System to a sinusoidal input waveform

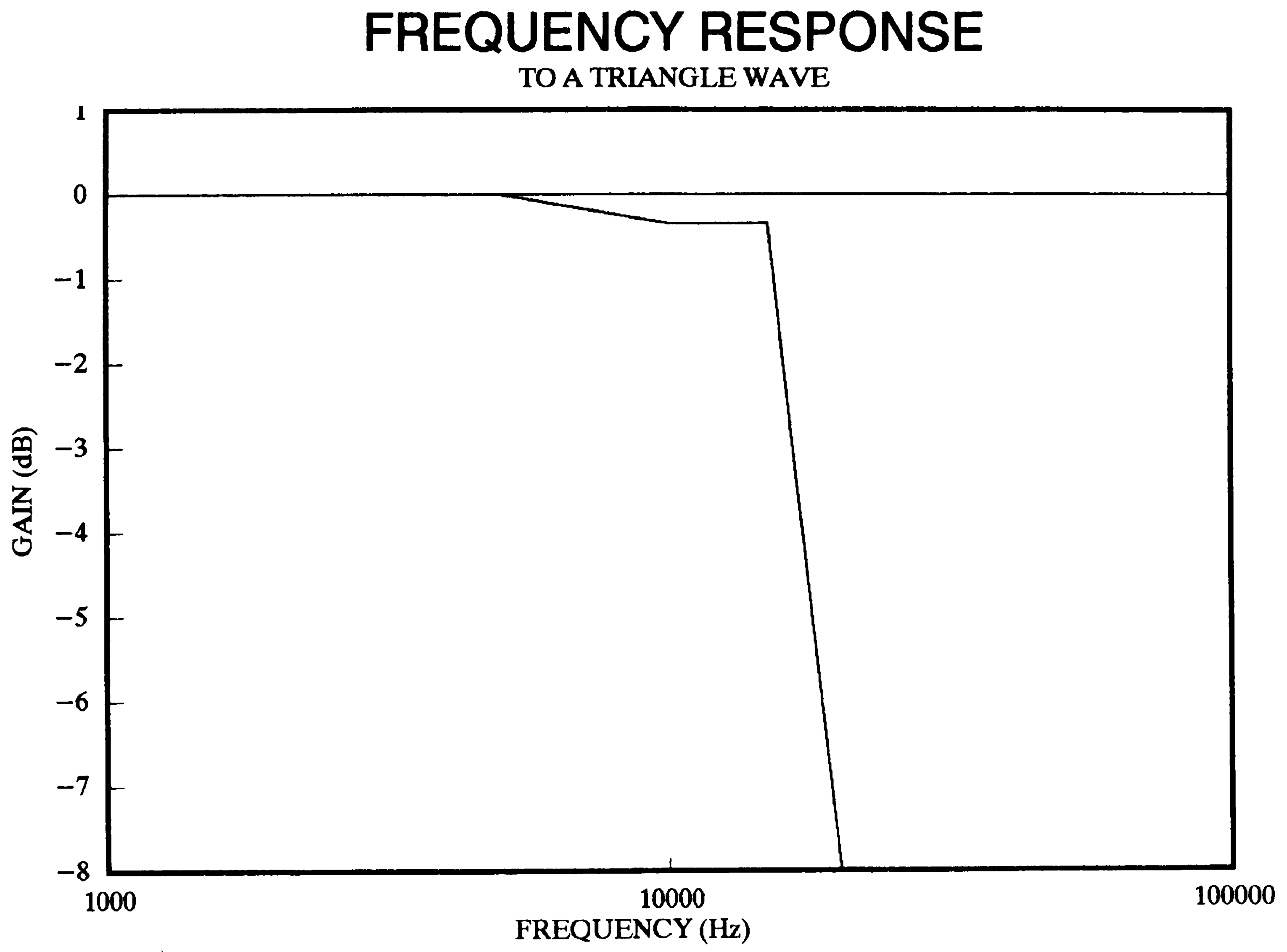


Figure 3: Frequency response of Microprocessor Filter System to a triangle wave input

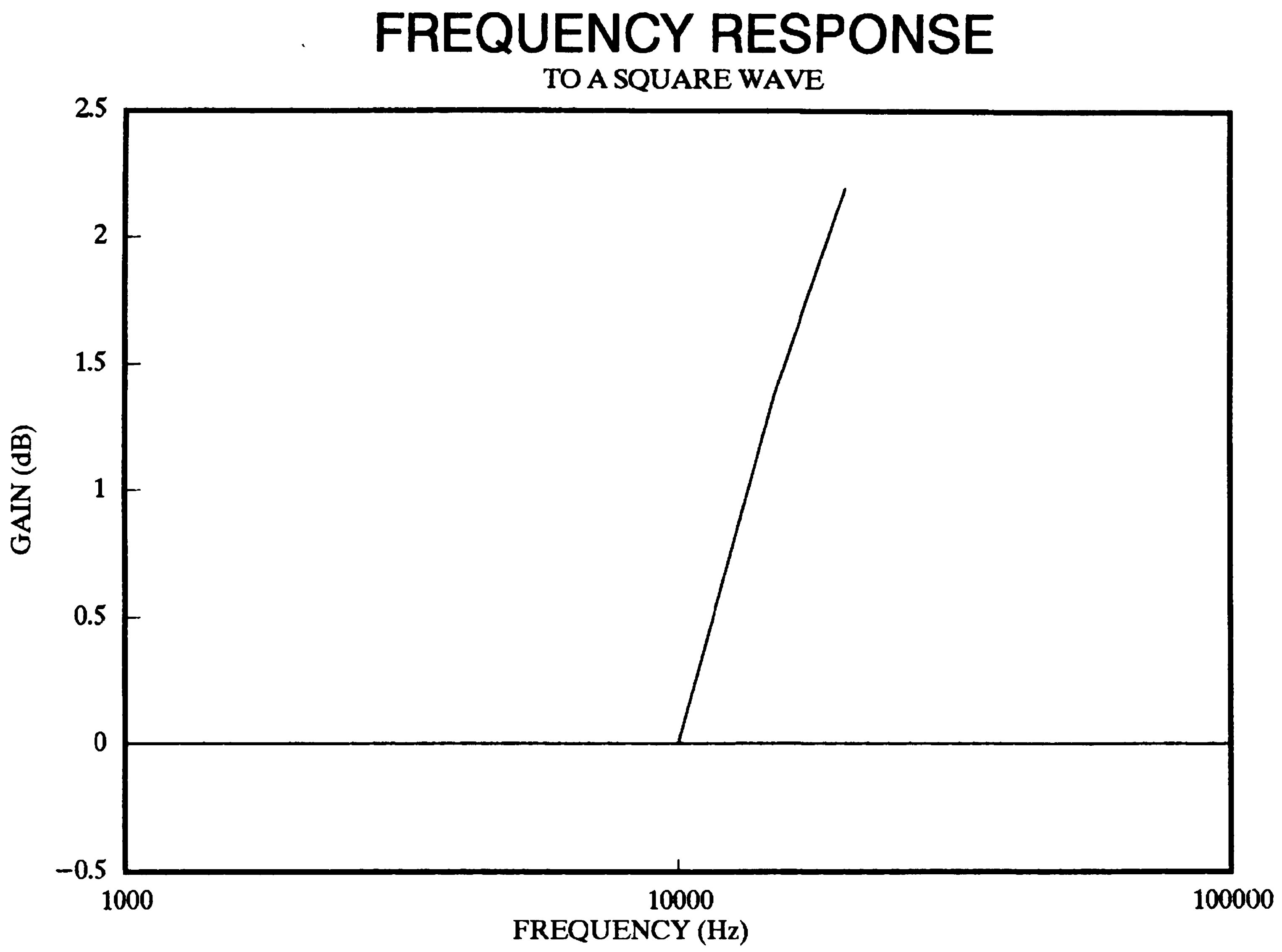


Figure 4: Frequency response of Microprocessor Filter System to a square wave input

6.2.2 TAPE HISS

Using the same tape that was used with the previous experiment, tape hiss was removed from the recording. For this test the Fourier transform analysis and editing was used. A sample of music was read into memory, and the tape player and memory storage routine were both stopped. A Fourier transformation was done on one kilobyte of memory. The transform information was then fed through a "kill filter". This filter removed all frequency harmonics below the manually chosen threshold amplitude. The threshold level selected in this case was the quantization error associated with the least significant bit of an eight bit sample: 39 millivolts. It was necessary to choose a very small amount of memory because of the processing time that a Fourier transform demands and the relatively slow processing speed of the computer being used (a 6502 with 4 Megahertz). Because of this, the sample selected for the transform (which represented one fortieth of one second) was impossible to hear. However, examination of the signal on an oscilloscope (using the repeat playback mode of the software) showed that there was a significant reduction in the amplitude of the noise present before filtering.

The first tape hiss test was run with no music in the background, just tape hiss. Initially the amplitude of the noise was 3.5 volts peak to peak. After the filtering, this noise was reduced to 1.6 volts peak to peak. This is a drop in noise of 6.8 dB. The second tape hiss test run was with music. No noticeable change occurred in the music signal (as displayed on an oscilloscope). This type of filtering resulted in a decrease in the amplitude of the noise, with no appreciable decrease in the amplitude of the music. This type of filtering, therefore, actually increases the signal to noise ratio, the true measuring stick for an audio filter.

CHAPTER 7 - CONCLUSIONS

This is a brief discussion of the success of the project versus the theoretical predictions, the problems encountered during the project, and suggestions for future directions for the project.

7.1 ACHIEVING PROJECT GOALS

One goal that was set for the microprocessor based audio filtering system was to remove various kinds of noise from old recordings, by allowing for human interaction. The actual system did successfully remove a "pop" from a recording by allowing human interaction (i.e., editing).

It was also expected that by using a flexible set of software routines, it would be possible to remove tape hiss from a tape recording. The actual system produced an increase in the signal to noise ratio (6.8 dB) in regards to tape hiss, by performing a Fourier analysis and editing on the signal.

Another goal set for the system was that it should not introduce any new audible signals into the recording. The actual system introduced no signals in the audible range into the recording during either of the noise removal processes.

The actual system achieved all of the goals set for it.

7.2 DISCUSSION OF PROBLEMS

Several problems were encountered during the implementation of the design. Most of the problems were related to the printed circuit board manufacturing process.

Initially, trying to fit the entire main circuit on a 4 in. x 6 in. board that was too small for the purpose (but of no cost), led to an extremely difficult routing task. The OrCad auto-router was of little use with a complex design in a small area, and many of the routes had to be laid out manually. After this tedious task was complete, the actual exposing and etching of the board had to be very precise to prevent tracks being broken or shorted, and to line up one layer of the board exactly with the other. This step was very time consuming. The last step of the board making process, the drilling of the holes for vias and socket mounting, turned out to be the factor that made using a board of this size impossible. There was no drill bit/drill press available that could hold the tolerances that the tightly packed copper tracks demanded. From here we realized it would be necessary to use (and pay for) a larger board for the circuit.

Once the larger board, which was 9 in. x 9 in., was created, some additional problems were found. Some etching occurred beneath the photoresist, which caused some of the tracks on the board to lift. In order to compensate for this it was necessary to lay down copper tape in several places on the board. The use of copper tape made precise soldering a difficult task. It was not until several trials of the board until all of the broken lines were discovered and repaired.

The only problem not related to the manufacturing process was the connections between the DAC and ADC chips to their respective sockets, and the sockets to the copper tracks. These two chips require a very good connection in order to properly reproduce the signal without adding any noise to the signal. Pressing on the chips during operation eliminated any noise being created by the system.

7.3 SYSTEM LIMITATIONS

One improvement that can be made to the system would be upgrading it to sixteen bit resolution. This would facilitate CD quality reproduction of input music, and lower the noise threshold of the system.

Another improvement that can be added is increasing the memory capacity to accommodate an entire song. The capacity of memory neccessary for this at sixteen bits is approximately twenty four Megabytes for a four minute song. The additional memory would allow the user to apply a transfer function to the entire song at once.

Another modification would be to increase the sampling and storage speeds of the system. Using oversampling techniques simpler input and output filters can be used. Oversampling would reduce phase distortion near the poles of the filter.

7.4 FUTURE DIRECTION

Because the system actually worked as well as the theory predicted, there is a wide variety of possible future directions for the project.

One of the more obvious paths would be to add more types of digital filtering routines, to eliminate a wider range of noise from music. Another software option would be to expand the user interface routines and allow for easier user interaction in the editing process. If a software function could be written to aid the user in distinguishing between music and noise, the utility of the system would increase greatly.

Another possible use for this system would be to keep music archives. With the proper amount of error checking, no faults should ever be introduced into the music stored.

REFERENCES

1. National Semiconductor, *Linear Databook ℓ3*, pp 1-131 - 1-135, 1988.

2. J. V. Wait, L. P. Huelsman and G. A. Korn, *Introduction to Operational Amplifier Theory and Applications*. NY: McGraw-Hill, 1975.

3. Ibid.

4. A. V. Oppenheim and R. W. Schafer, *Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, 1975.

5. Ibid.

APPENDIX A

MAKING PRINTED CIRCUIT BOARDS

The actual manufacture of a custom printed circuit board is a lengthy and complicated process. First a circuit is completed and tested. Then, with an appropriate program the schematic is prepared. The trace for the board must be laid out to 1:2 scale. A photographic process is then used to transfer the trace to a copper clad board. Finally the board is chemically etched and tinned to protect it. The process consists of about twenty steps, and proper handling at each stage is crucial to successful completion of the board. The process described in this chapter is applicable to the manufacture of one or two layer boards.

INITIAL DESIGN AND LAYOUT

Initial design is easily accomplished using a program such as OrCad, which is available for student use in the microprocessor laboratory on the second floor of Atwater Kent. Using the program is fairly straightforward, but there are a few details which are easily overlooked. The documentation is lengthy and cumbersome. Not all simple questions are easily answered.

Most of the common errors do not appear until one attempts to create a circuit layout from a schematic created with the drafting program. It is very important to assign each unique part a unique reference, this makes it much simpler to identify which components belong where, once the printed circuit board is made. When multiple parts of the same chip are used (such as the TL-084 op-amp chip which contains four distinct op-amps), this prevents the computer from assigning each part to a different chip. For transfer to a printed circuit board all contacts must be drawn out with "wire" connecting the contacts. Wire must always enter the part with a straight line perpendicular to the part, and wire must never overlap the body of a part. The key field is the part field that describes the way the part will be physically placed on the board. IC chips are easily placed, but the many variations in the size of resistors, capacitors, and other small parts requires the user to actually measure the part and compare it to the "modules" available in *pcb*. A module is an exact description of the number, size, and location of contact pads required for mounting a part. Many standard packages are already available, but there is no catalog. One must browse through the listing and find a shape that will fit their part. Then one must return to *draft* and set the name of the appropriate part field to the name of the module. The part field to be used is the one selected as the key field when one configures *draft*.

It is recommended that one become familiar with the *libedit* function when first learning to use the system. Many times a user will find it necessary to create a custom part to use in a schematic. It is much better to create a custom library of parts one needs, than to alter parts in the existing libraries. Other users may be depending on the current library configuration. To replace a part definition without altering the standard libraries is fairly simple. Go to configuration mode and place the name of the library in which the replacement part appears, before the library in which the part is normally contained. Then update the configuration. If anyone else is relying on the original part definition, they need only remove your library from the active list. It is necessary to create part descriptions for special IC chips, for pads to connect wires to the board, and for a more practical ground symbol.

It may sometimes be necessary to use *libedit* to fix parts in the normal device libraries which did not work properly with the *pcb* program. Many graphic parts have named pins instead of pin numbers. This abnormality is not apparent until a netlist is made from the schematic. In the netlist certain pins of parts are named instead of numbered. If this is the case, the *pcb* will interpret the information as connections that are not made, because *pcb* expects part pins to be numbered, not named. A user could simply edit their netlist file, changing names to numbers. A more appropriate solution is to "replace" illegally named parts with similar parts, but with numbered pins. Familiarity with *libedit* makes schematic layout much more elegant, and provides the user with a better understanding of the processes involved.

After successfully completing the schematic, one must generate a netlist. I used a batch file which sets all the appropriate options for the use of netlist. The use of a batch file makes generating a netlist very simple, just type "ornet filename". Placement of the netlist in the appropriate directory is handled automatically. If any errors occur at this level, it is better to go back to *draft* and work them out, than to go on to *pcb*. Errors at this stage mean incomplete connections, or a serious circuit design error. Once the netlist is completed, a person may select the *pcb* program, and begin laying out how parts will actually be placed.

The most important feature of *pcb* is that it can automatically route the board. One simply selects the strategy to be used, and the area to be routed. *Pcb* is not perfect however. It often misses a simpler track layout that might cause other paths to remain unconnected. But computing the "best" way of laying out an advanced circuit is an extremely time consuming task for anyone. It would be impractical to do it entirely by hand. The best solution is to use a rat's nest view of the circuits, and to place some of the tracks in the more confused areas manually, to simplify layout, and then let the computer place the remaining tracks. After all the tracks are placed, select the postscript driver, and have *pcb* write the layer image to disk. Then take the disk to CCC and print the image out on one of the laser printers. For use with the EE shop facility, *pcb* should be directed to make plots at double size.

MANUFACTURE OF PRINTED CIRCUITS

Step 1:

The image produced is then taken to the dark room, where a negative is to be made. The photographic part of the process is crucial to the proper completion of the board. The first step is to prepare the chemicals to be used. Most of the chemicals are provided in powdered form, and must be mixed with water. Since little of the chemicals is actually consumed, they may be reclaimed for future use. For this reason it is better to mix the powder with deionized water, because the tap water contains many contaminants which may lead to premature fouling. The Kodalith solutions A and B may be mixed together in equal parts. They form the developing solution.

Step 2:

The fixing solution is poured straight from the container. Also connect the rinse tray to the faucet and set the water flowing slowly. Two trays are currently available, a large round one and a large rectangular one. Since it is very important to get all of the picture under the developer at the same time, the rectangular one should be used for the developer.

Step 3:

The next step involves taking the photograph. Place the frosted glass panel in the exposure area and line up the trace behind the glass. It is important for the trace to be centered in the film. The film used is only 8" x 10", and it is necessary to leave a half inch border on each side. This means that the largest board which can be produced with the camera in the shop is roughly 7" x 9". Tape may be used to hold the image in place, but should be affixed to the glass, not the non-reflective paper used as a backing material.

Step 4:

Next turn on the small red lamp connected to the timer, and locate the film to be used. Do not open the film until all of the white and yellow light sources are turned off, and only the red lights, which will not expose the film, are turned on. Take out a piece, and examine it under the lights. One side will be very glossy, and the other has a much duller finish. It may be difficult to tell which side is which under the red light conditions but looking closely at the reflection of the ceiling lamps, one should be able to see the cracks in the lamp in the shiny side.

Step 5:

Select two pieces of film and place them dull side up inside the film carrier. Then slide the film carrier into its place on the camera, and cover the rear of the camera with the cloth. At this time one must make a point to carefully wrap and store the remaining film. The film used is very expensive, costing almost a dollar per negative, with roughly a hundred negatives per box. Every negative will be ruined if the box of film is exposed to white or yellow light.

Step 6:

Next remove the piece of plastic on the film carrier that protects it from exposure. Make sure nothing white is near the photo image surface, reflections will cause imperfections in the photograph. Also make sure that nothing at all gets between the camera and the image (hands, arms, coats,...)

Step 7:

Set the timer for the desired exposure time. The best exposure time depends on the area of the picture and the ratio of light to dark areas in the laser print. Press the metal button to start the timer and activate the exposure lamps. After the white light goes off, remove the film carrier, replace the plastic shield, flip the carrier to the other side, remove its shield, and take another picture. It is best to take two pictures because it is very difficult to judge how developed a picture is under low light conditions, and one will inevitably come out better than the other.

Step 8:

After both pictures have been exposed, take one and place it in the developing solution. Optimum developing time is discovered by trial and error. The best way to judge the state of the film is to take it out of the developer, rinse it off, and look at it under the red light. Ideally, the tracks should be perfectly clear, and the background should be dark enough not to allow any light through.

Step 9:

When an image appears that it is dark enough for use, rinse it off with water, and place it in the fixer. Leave it in the fixer for at least one minute, then rinse it with rinse. Unlike the developer, one minute or five minutes in the fixer makes little or no difference, provided the minimum time of a minute is observed. Repeat the process for the second image.

Step 10:

After the last negative has spent at least a minute in the developer it is safe to turn the room lights on again. Once the film is removed from the water be careful not to get fingerprints on the negatives, try to keep them as clean as possible. It is advisable to hang the negatives in a frame and allow the water to dry off them slowly, over the course of an hour or so. Provided neither negative has obscured tracks, the best choice is the one with the darkest background, not the one with the clearest image. It really does not matter if the hole in the center of the pad is visible, but it is crucial that copper is only left on the parts of the board where the copper is supposed to be.

Step 11:

The next step in the process is to expose the board to ultraviolet light through the negative. The boards used are sensitive to light except for certain red and yellow colors. The board must not be exposed to room lights or daylight before developing. Use only the red darkroom lights, which will not affect the board. Mark the side of the board being used in an area away from any tracks. This will prevent accidentally exposing the same side of the board twice.

Step 12:

Lay the film on top of the exposer, then lay the negative sensitized board on top of the film. It is a good idea to use a board cut to the exact size, or to place reference marks on the board layout so that one may be sure the pads on both sides of the board will align correctly. Expose the board for two minutes through the film, then expose the other side through the second negative.

Step 13:

The photoresist developer contains Xylene, which is carcinogenic, heavier than air, and highly flammable. Place the developing tray in the shielded booth. The booth vents all of the dangerous fumes to the outside. The booth fan should never be turned off. Pour the developer into a tray, and hold the board in the developer using the tweezers. Wash the surface of the board with the liquid for two minutes, raising and lowering the card within the developer to circulate the developer evenly.

Step 14:

Remove the board, pour the developer back into the container, and close the glass door to prevent fumes from entering the room. Bake the card in the oven for fifteen minutes at seventy-five degrees Celsius.

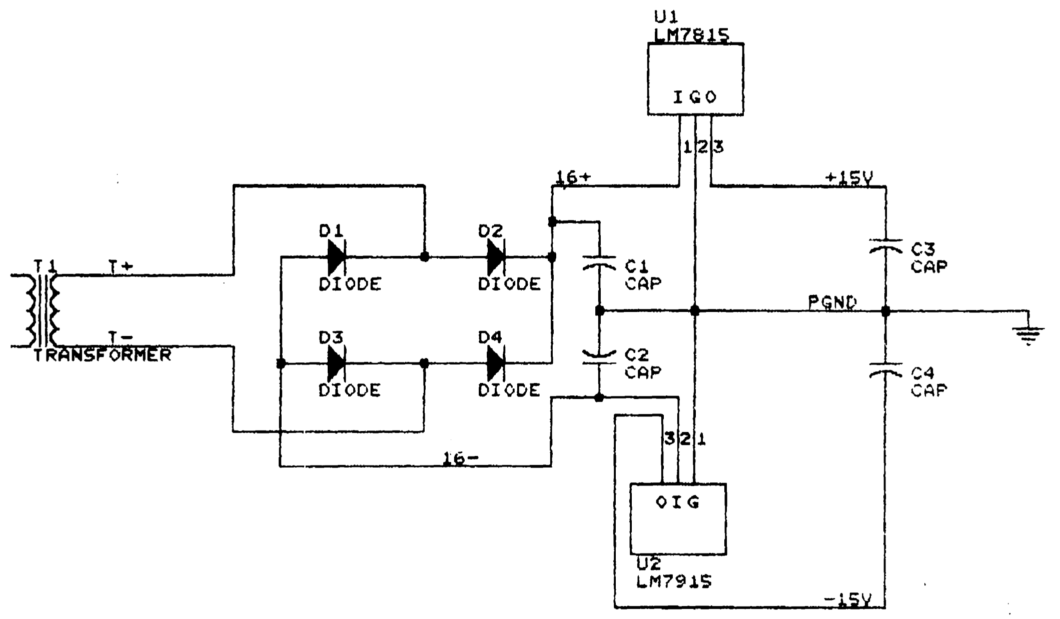
Step 15:

Place the card in the ferric chloride acid bath before it cools. If it is convenient to heat the acid as well, do so. The more rapidly the acid attacks the copper to be removed the better. The acid attacks the surface of the board, but may also begin to attack the copper beneath photoresist from the sides. This results in breaks in the tracks, or poor contacts. For large or double sided boards it is helpful to keep some powdered ferric chloride available to add as the solution begins to weaken. For double sided boards the bottom of the board should be raised above the bottom of the tray using glass rods or beads. Using beads ten millimeters or larger also has the desired effect of rolling freely across the surfaces. This agitates the acid, and helps to remove copper from the board surfaces. Occasionally removing the card from the acid and washing the surface with water also helps somewhat.

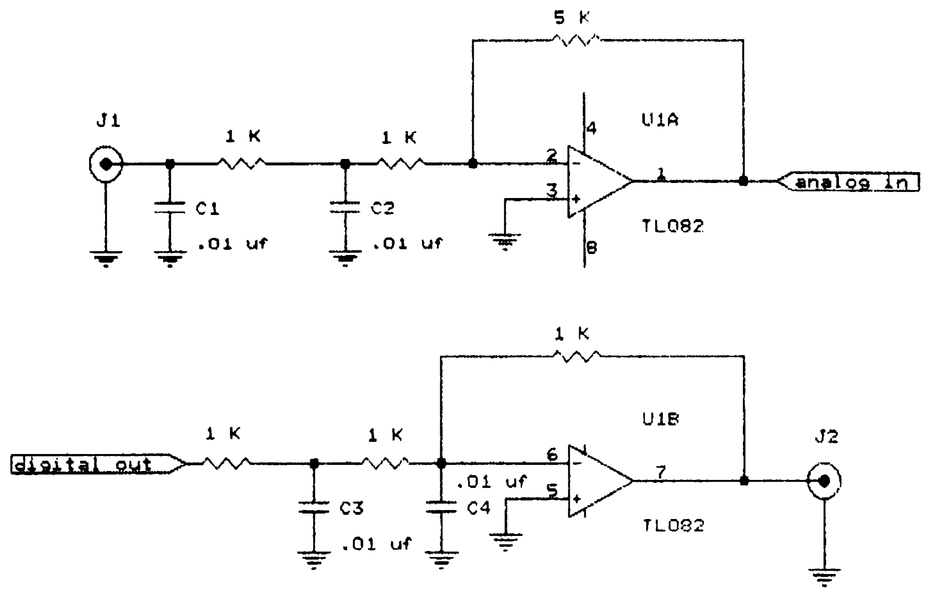
Step 16:

The final step is to drill holes in the proper locations on the board, making use of the small drill press in the dark room. With all the holes drilled, the board is ready for socket mounting. Be certain to test for broken or cross-connected tracks before and after soldering parts onto the board. Unwanted connections in the copper are easily cut and lifted out. Small wire jumpers are easily soldered across breaks in tracks. Minor defects will not render the board unusable.

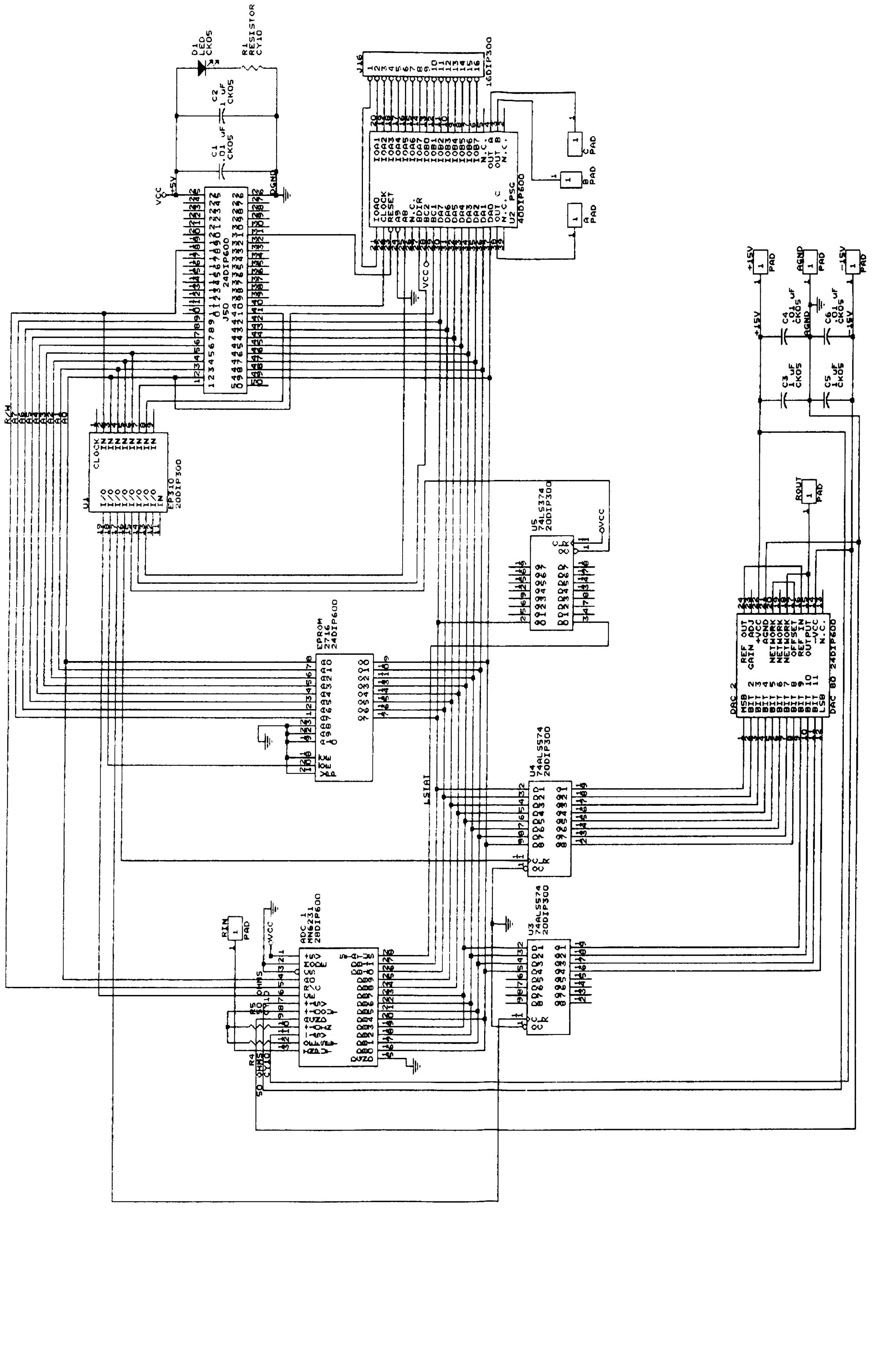
APPENDIX B



Worcester Polytechnic Institute Michael Pencer Rob Ursillo		
Title Power supply schematic		
Size A	Document Number	REV B
Date: October 15, 1990	Sheet 1 of 2	



WORCESTER POLYTECHNIC INSTITUTE MIKE PENDER AND ROB URSILLO		
Title IMPEDANCE FILTERS		
Size A	Document Number HSR 1000	REV B
Date: November 29, 1990	Sheet	of 1


D1 LED CK05
R1 RESISTOR CY10
C2 1 uF CK05
C1 .01 uF CK05
VCC
+5V
DGND
J50 24DIP600
J16 16DIP300
U2 PSG 40DIP600
A PAD
B PAD
C PAD
+15V PAD
AGND PAD
-15V PAD
C4 .01 uF CK05
C6 .01 uF CK05
C3 1 uF CK05
C5 1 uF CK05
R/W
A7
A6
A5
A4
A3
A2
A1
A0
U1 EP310 20DIP300
CLOCK
U5 74ALS374 20DIP300
ROUT PAD
EPROM 2716 24DIP600
DAC 2 DAC 80 24DIP600
U4 74ALS574 20DIP300
U3 74ALS574 20DIP300
RIN PAD
ADC 1 MN6231 28DIP600
R5 50 OHMS CY10
R4 50 OHMS CY10
LSIAI

APPENDIX C

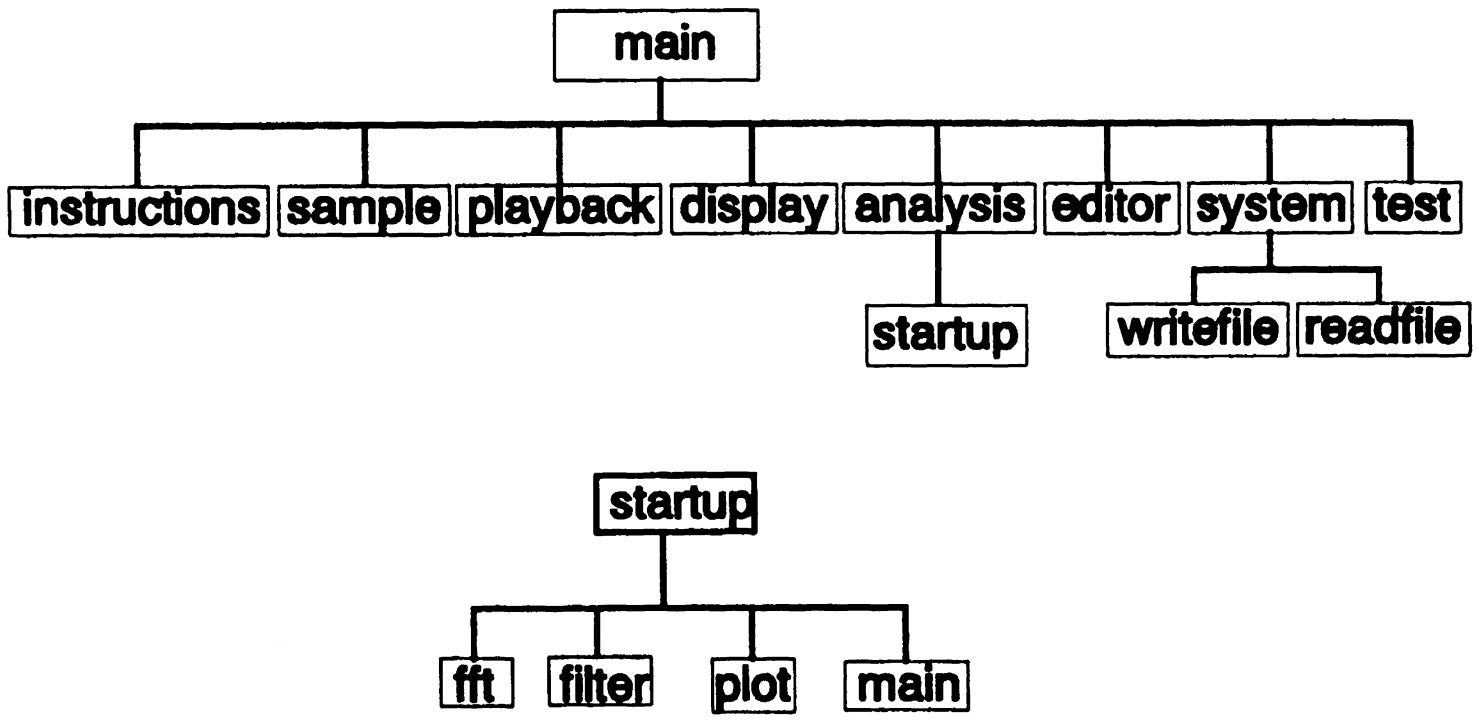


Figure 5: Block Diagram of Software

/MQP/MQP.C

```
/*
 * mqp.c
 *
 * main routine, controls various functions of microprocessor
 * filter.
 *
 */

#include <std.h>            /* File i/o macros */

#define height 24           /* Screen height */
#define false  0
#define true   1

/* global variables */

char    FNAME[64];          /* Filename to be used */
char    START[3] = { 0, 0, 0};          /* Beginning and end points of */
char    FINISH[3] = { 255, 255, 255}; /* Data segment being manipulated */
char    SPEED = 1;          /* Rate at which data was recorded */

/* hardware equates */

char    *MEMSEL = 0xc500;   /* Select Ramcard */
char    *MELOW  = 0xc0d0;   /* Low byte of address */
char    *MEMID  = 0xc0d1;   /* Middle byte of address */
char    *MEHIGH = 0xc0d2;   /* High byte of address */
char    *MEDATA = 0xc0d3;   /* Data byte address */

main ()
{
    char    key;
    settxt();
    home();
    putstr("WARNING --- This program uses the memory on the slot five\n");
    putstr("ram card.  Proceeding beyond this point will result in the\n");
    putstr("destruction of any files stored on volume /RAM5\n\n");
    putstr("Are you certain you wish to proceed? (y/n) ");
    if(getkey() == 'n')
    {
        putstr("\n\n");
        exit();
    }
    key = '\n';
    while(key != '0')
    {
        home();
        cursor(0,19);
        putstr("MQP HGR-1000   Microprocessor Based Filter\n");
        putstr("\n\tThis software is designed to work with an experimental\n");
        putstr("\tfilter card created for Apple II computers.\n");
        putstr("\tVarious features of this software can be run without the\n");
        putstr("\tboard present, but the system may lock up.\n");
        putstr("\nMain menu:\n");
        putstr("\t1.\tView instructions\n");
        putstr("\t2.\tSample mode\n");
        putstr("\t3.\tPlayback mode\n");
        putstr("\t4.\tDisplay mode\n");
        putstr("\t5.\tAnalysis\n");
        putstr("\t6.\tEditor mode\n");
        putstr("\t7.\tOperating system functions\n");
        putstr("\t8.\tLoad test data\n");
        putstr("\t0.\tExit program\n");
        key = menu('8');
        switch(key)
        {
            case '1':
                instructions();
                break;
            case '2':
                sample();
                break;
            case '3':
                playback();
                break;
            case '4':
                display();
                break;
            case '5':
                analysis();
                break;
            case '6':
                editor();
                break;
            case '7':
                system();
                break;
            case '8':
                test();
                break;
        }
    }                   /* note:  A key value of '0' terminates this loop */
    home();
}

instructions()
{
    char    filename[20] = "/mqp/mqp.helpfile";
    char    c, p, cnt;
    FILE    fp;

    cnt = 0;
    fp = open (filename);
    home();

    while ((c=getc(fp)) != -1)
    {
        if (c == '\n' && ++cnt == height)   /* uses the partial evaluation */
        {                                   /* of if statement to increment */
            p = getkey();                   /* counter */
            if (p == 'q'  || p == 'Q')
            {
                putchr('\n');
                return(1);
            }
            cnt = 0;
        }
        if (c >= ' ' && c < 0x7f || c == '\n' || c == '\t')
            putchr(c);                      /* Filter output characters */
    }
    close(fp);
    wait_return();
}

sample()
{
    char key;
    home();
    putstr("Executing 8 bit sample mode\n");
    putstr("Press any key to begin...\n");
    fastsample();
}

playback()
{
    char key, bufr[2];

/*  This routine takes data currently stored in memory and pipes it out to
    the user through the D/A port.  Start and end points are user selectable */

    home();
    putstr("This section plays data stored in memory out through the\n");
    putstr("D/A port.  Any segment of the data stored in memory may be\n");
    putstr("played back.  Use the arrow keys to change the settings.\n");
    putstr("The output will loop until a key is pressed...\n");
    getstart();
    getfinish();
    printf("\nSpeed = %x%x -->", SPEED / 16, SPEED & 0x0f);
    if((bufr[0] = get_hex()) != 0xff)
        if((bufr[1] = get_hex()) == 0xff)
            SPEED = bufr[0];
        else
            SPEED = bufr[1] + 16 * bufr[0];
    home();
    putstr("Executing Eight bit playback mode\n");
    play8();
}

display()
```

```
/*  Display takes data already stored on the memory card and shows it on
    the screen as hexadecimal data.  It is used to tour through data
    sampled and stored in memory */

{
    char    done, key, temp;
    char    i, j;

/*  This code segment verifies data is available for viewing, then gets the
    start address from the user, and loads the address into the RAM card */

    home();
    putstr("Display mode \n\n");
    putstr("\tThis function opens a window which can display\n");
    putstr("information currently stored on the desktop.\n");
    getstart();
    *MEMSEL = 0;
    *MEHIGH = START[2];
    *MEMID  = START[1];
    *MELOW  = START[0];

/*  This is the segment that actually displays the data on the card to
    the screen.  Incrementing of the data location is handled automatically
    by sequential accesses to the location MEDATA.  MEHIGH, MEMID, and
    MELOW always contain the current data location, as they are
    incremented sequentially by hardware. */

    done = false;
    while(!done)
    {
        for(i = 0; i < 20; i++)
        {
            printf("0x%x%x%x%x%x - ", *MEHIGH & 0x0f, *MEMID / 16,
                *MEMID & 0x0f, *MELOW / 16, *MELOW & 0x0f);
            for(j = 0; j < 16; j++)
            {
                temp = *MEDATA;
                printf(" %x%x", temp / 16, temp & 0x0f);
            }
            putchr('\n');
        }
        putstr("\nContinue? (y/n) ");
        key = getyn();
        if (key == 'n')
            done = true;
        putchr('\n');
    }
}

analysis()
{
    home();
    putstr("\tAnalysis menu....\n");
    putstr("Frequency analysis functions are handled by another program\n");
    putstr("written in BASIC.  This program will set up the data files\n");
    putstr("required, but you must manually run the other program.\n");
    putstr("This program will ask for the start address for analysis,\n");
    putstr("set up the ramcard, and then exit.  You must type 'bye',\n");
    putstr("then select BASIC.SYSTEM to run the analysis programs\n\n");
    putstr("We apologize for the inconvenience...\n");
    putstr("Are you sure you want to do this? (y/n) ");
    if(getyn() == 'n')
        return;
    putchr('\n');
    getstart();
    exit();
}

editor()
{
    char done, key, temp, bufr[2];
    char i, j;
    home();
    putstr("Editor mode\n\n");
    getstart();

    done = false;
    while(!done)
    {
        *MEMSEL = 0;
        *MEHIGH = START[2];
        *MEMID = START[1];
        *MELOW = START[0] & 0xf0;
        printf("\n0x%x%x%x%x%x -", *MEHIGH & 0x0f, *MEMID / 16,
                *MEMID & 0x0f, *MELOW / 16, *MELOW & 0x0f);
        for(i = 0; i < 16; i++)
        {
            temp = *MEDATA;
            printf(" %x%x", temp / 16, temp & 0x0f);
        }
        *MEHIGH = START[2];
        *MEMID = START[1];
        *MELOW = START[0];
        printf("\n0x%x%x%x%x%x = ", START[2] & 0x0f,
                START[1] / 16, START[1] & 0x0f,
                START[0] / 16, START[0] & 0x0f);
        temp = *MEDATA;
        printf("%x%x -->", temp / 16, temp & 0x0f);
        if((bufr[0] = get_hex()) == 0xff)
            return;
        if((bufr[1] = get_hex()) == 0xff)
            temp = bufr[0];
        else
            temp = bufr[1] + 16 * bufr[0];
        *MEHIGH = START[2];
        *MEMID = START[1];
        *MELOW = START[0];
        *MEDATA = temp;
        if(!(++START[0]) && !(++START[1]) && !(++START[2]))
            done = true;
        /* partial evaluation of the if clauses is used to increment the
           current position */
    }
}
system()
{
    char key;
    home();
    putstr("System Function Menu\n\n");
    putstr("\t1.\tSave desktop data to file\n");
    putstr("\t2.\tLoad desktop data from file\n");
    putstr("\t0.\tExit to previous menu\n");
    key = menu('2');
    switch (key)
    {
        case '1' :
            getstart();
            getfinish();
            getfname();
            writefile();
            break;
        case '2' :
            getfname();
            readfile();
            break;
    }
}

test() /* Loads precomputed data onto desktop for verification */
{
    char key;           /* holds value passed from menu routine */

                        /* data image for a sinusoidal waveform */
    static char sinusoid[32] =
              { 128, 152, 176, 198, 217, 233, 245, 252,
                255, 252, 245, 233, 217, 198, 176, 152,
                128, 104,  80,  58,  39,  23,  11,   4,
                  1,   4,  11,  23,  39,  58,  80, 104 };

                        /* data image for a square waveform */
    static char square[32] =
               {  0,   0,   0,   0,   0,   0,   0,   0,
                  0,   0,   0,   0,   0,   0,   0,   0,
                255, 255, 255, 255, 255, 255, 255, 255,
                255, 255, 255, 255, 255, 255, 255, 255 };

                        /* data image for a triangular waveform */
    static char triangle[32] =
               {  0,  16,  32,  48,  64,  80,  96, 112,
                128, 144, 160, 176, 192, 208, 224, 240,
                256, 240, 224, 208, 192, 176, 160, 144,
                128, 112,  96,  80,  64,  48,  32,  16 };
    char *image;        /* pointer to appropriate data image */
    char i, j;
    home();
    putstr("Test data generator menu:\n\n");
    putstr("\tThis functions allows one to load precomputed test data\n");
    putstr("onto the desktop for testing and verification.\n");
    putstr("\nTest menu:\n");
    putstr("\t1.\tSinusoid data\n");
    putstr("\t2.\tSquare wave data\n");
    putstr("\t3.\tTriangle wave data\n");
    putstr("\t0.\tExit to previous menu\n");
    key = menu('3');
    putstr("\n\tLoading data image...\n");
    switch(key)
    {
        case '1':   /* image data for a sinusoid */
            image = sinusoid;
            break;
        case '2':   /* image data for a square wave */
            image = square;
            break;
        case '3':   /* image data for a triangle wave */
            image = triangle;
            break;
        case '0':   /* exit to calling program */
            return;
            break;
    }
    putstr("\n\tLoading 1024 data points into desktop...\n");
    *MEMSEL = 0;    /* Select ram card hardware */
    *MELOW = 0;     /* Set address for card to start of space */
    *MEMID = 0;
    *MEHIGH = 0;

    for(i = 0; i < 32; i++)
        for(j = 0; j < 32; j++)
            *MEDATA = *(image + j);

    START[2] = 0;   /* Set data start point to 0x00000 */
    START[1] = 0;
    START[0] = 0;

    FINISH[2] = 0;  /* Set data end point to 0x003ff */
    FINISH[1] = 3;
    FINISH[0] = 0xff;

    wait_return();
}

char get_hex()
{
    char key, value, done;
    key = 0;
    done = false;
    while(!done)
    {
        key = getkey();
        if((key == '\n') || ((key >= '0') && (key <= '9'))
            || ((key >= 'a') && (key <= 'f')))
            done = true;
        if(key == '\n')
            value = 255;
        if((key >= '0') && (key <= '9'))
            value = key - '0';
        if((key >= 'a') && (key <= 'f'))
            value = 10 + key - 'a';
    }
    putchr(key);
    return value;
}

wait_return()
{
    char key;
    key = 0;
    putstr("\nPlease press <return> to continue...");
    while(key != '\n')
        key = getkey();
    putchr('\n');
}

char menu(options)

{
    char key;

    printf("\n\t\tYour choice? (0-%c) ", options);
    key = '\n';
    while((key < '0') || (key > options))
        key = getkey();
    printf("%c\n", key);
    return key;
}

char getyn()
{
    char keypress;
    keypress = '\n';
    while((keypress != 'n') && (keypress != 'N')
       && (keypress != 'y') && (keypress != 'Y'))
        keypress = getkey();
    switch (keypress)
    {
        case 'n':
        case 'N':
            keypress = 'n';
            break;
        case 'y':
```

```
        case 'Y':
            keypress = 'y';
            break;
    }
    return(keypress);
}

getstart()
{
/*  This routine prompts the user for a start address.  It is called by
    the display, playback, and save routines. */

    char done, index, key;
    char adrs[5];

    done = false;
    index = 0;
    adrs[0] = START[2] & 0x0f;
    adrs[1] = START[1] / 16;
    adrs[2] = START[1] & 0x0f;
    adrs[3] = START[0] / 16;
    adrs[4] = START[0] & 0x0f;

    while(!done)
    {
        cursor(10,0);
        putstr("Start Address >0x");
        printf("%x%x%x%x%x", adrs[0], adrs[1], adrs[2], adrs[3], adrs[4]);
        cursor(10,17 + index);
        key = getkey();
        switch(key)
        {
        /* Up arrow */
            case 0x0b :
                adrs[index] += (adrs[index] < 15);
                break;
        /* Down arrow */
            case 0x0a :
                adrs[index] -= (adrs[index] > 0);
                break;
        /* Right Arrow */
            case 0x15 :
                index += (index < 4);
                break;
        /* Left Arrow */
            case 0x08 :
                index -= (index > 0);
                break;
        /* Return adrs */
            case '\n' :
                done = true;
                break;
        }
    }
    START[2] = 240 + adrs[0];
    START[1] = 16 * adrs[1] + adrs[2];
    START[0] = 16 * adrs[3] + adrs[4];
    putchr('\n');
    putchr('\n');
    return;
}

getfinish()
{
/*  This routine prompts the user for an end adrs.  It is called by
    the playback, and save routines. */

    char done, index, key;
    char adrs[5];

    done = false;
    index = 0;
    adrs[0] = FINISH[2] & 0x0f;
    adrs[1] = FINISH[1] / 16;
    adrs[2] = FINISH[1] & 0x0f;
    adrs[3] = FINISH[0] / 16;
    adrs[4] = FINISH[0] & 0x0f;

    while(!done)
    {
        cursor(12,0);
        putstr("End Address   >0x");
        printf("%x%x%x%x%x", adrs[0], adrs[1], adrs[2], adrs[3], adrs[4]);
        cursor(12,17 + index);
        key = getkey();
        switch(key)
        {
        /* Up arrow */
            case 0x0b :
                adrs[index] += (adrs[index] < 15);
                break;
        /* Down arrow */
            case 0x0a :
                adrs[index] -= (adrs[index] > 0);
                break;
        /* Right Arrow */
            case 0x15 :
                index += (index < 4);
                break;
        /* Left Arrow */
            case 0x08 :
                index -= (index > 0);
                break;
        /* Return adrs */
            case '\n' :
                done = true;
                break;
        }
    }
    FINISH[2] = 240 + adrs[0];
    FINISH[1] = 16 * adrs[1] + adrs[2];
    FINISH[0] = 16 * adrs[3] + adrs[4];
    putchr('\n');
    putchr('\n');
    return;
}

writefile()
{
    FILE fp;

    fp = create(FNAME);

    putc(fp, SPEED);

    *MEMSEL = 0;        /* Prepare ramcard for sequential read */
    *MEHIGH = START[2];
    *MEMID  = START[1];
    *MELOW  = START[0];

    /* This part writes segments of 65,536 bytes */
    while(*MEHIGH != FINISH[2])
        putc(fp, *MEDATA);

    /* This part writes segments of 256 bytes */
    while(*MEMID != FINISH[1])
        putc(fp, *MEDATA);

    /* This part writes the remaining bytes to the file */
    while(*MELOW != FINISH[0])
        putc(fp, *MEDATA);

    close(fp);
    return;
}

readfile()  /* Open file for input */
{
    char c;
    FILE fp;

    fp = open(FNAME);

    SPEED = getc(fp);

    *MEMSEL = 0;
    *MEHIGH = 0;
    *MEMID  = 0;
    *MELOW  = 0;

    START[2] = 0;
    START[1] = 0;
    START[0] = 0;

    /* This part reads segments the entire file */
    while((c = getc(fp)) != -1)
        *MEDATA = c;

    FINISH[2] = *MEHIGH;
    FINISH[1] = *MEMID;
    FINISH[0] = *MELOW;

    close(fp);
    return;
}

getfname()
{
    char i;
    putstr("\nPlease type the file's complete Prodos pathname.\n");
    conRead(FNAME, 64);
    putchr('\n');
    for(i = 0; i < 64; i++)
        if(FNAME[i] == '\n')
            FNAME[i] = 0;
    return;
}
```

/MQP/MQP.HELPFILE

```
--------------
| The Theory |
--------------
```

HGR-1000   A Microprocessor Based Filter

by Michael Pender and Rob Ursillo

This MQP deals with the design and construction of a Microprocessor based filter to remove certain types of noise from old music recordings.

Conventional filters, such as a bandpass, may reduce the amount of noise at the stage of reproduction, but cannot remove errors that have become part of the recording.

Certain types of noise, such as the pop due to record needles, are very difficult to filter by analog means. When viewed on an oscilloscope they look like a step, or rectangle wave. To an analog filter they look like a primary harmonic and an infinite number of associated harmonics, spread evenly across the entire spectrum.

But using a programmable digital playback system it should be possible for a person actively listening to "edit" the waveform until it sounds right. Also, using FFT derived frequency responses from the sampled data, a person may selectively reduce the amplitude of certain frequencies.

The Hardware

The system is implemented using a 12 bit analog to digital converter, and a matched 12 bit digital to analog converter. Input and output are passed through low-pass filters (two pole, Fc = 23,000 Hz) to match impedances and voltage levels. In addition it provides a level of isolation between the chips which interface with the microprocessor and the real world.

Input sampling is performed by a Micronetworks MN6231 chip. In eight bit mode they guarantee 100,000 samples / second. Output is passed through a Micronetworks DAC80 chip. These chips are capable of producing over 200,000 values per second.

The Software

Most of the system software is written in the Hyper C Prodos language. Drivers for the hardware are written in 6502 assembly code, and interfaced to the C shell. Due to the difficulty in handling graphics or floating point calculations in C, analysis software is written in BASIC, and data points are passed between the two in the slot five ramcard.

Hardware Requirements

The software expects the microprocessor filter card to be in slot seven, and an Apple Memory Standard ramcard in slot 5. Input and output of data files is handled through standard Prodos

8 pathnames, so data images may be stored on any Prodos compatible device in any slot except the ramcard, or the slot used by the filter.

The system was implemented on a Laser 128ex, with a fully populated one megabyte ramcard in slot five, and the filter connected to slot 7. A 3.5 inch drive was also connected to slot seven. To store files on the 3.5 inch drive we just flipped the slot seven switch and locked the card hardware out.

/MQP/STARTUP

```
 100  REM startup program
 110  HOME :D$ =  CHR$ (4)
 120  PRINT D$"pr#3"
 125  PRINT : VTAB 5
 130  PRINT "Analysis main menu:"
 135  PRINT
 140  PRINT "1. FFT analysis of data"
 150  PRINT "2. Plot utility"
 160  PRINT "3. Filter utility"
 170  PRINT "0. Return to main program"
 200  VTAB 15: PRINT "Your choice? (0-3) ";
 210  GET A$: IF (A$ < "0") OR (A$ > "3") THEN 210
 220  ON  VAL (A$) + 1 GOTO 4000,1000,2000,3000
 230  END
 1000  PRINT D$"-FFT"
 1010  END
 2000  PRINT D$"-PLOT"
 2010  END
 3000  PRINT D$"-FILTER"
 3010  END
 4000  PRINT D$"-C.SYSTEM"
 4010  END
```

/MQP/MQP1.A

```
; This file contains the machine-specific information required to
; make the system software work with the hardware on an Apple II system.
; The code expects a slot 5 ramcard.  It will use as much slot five
; memory as present, up to 1 meg.

; Time intensive tasks, such as sampling and playback routines, are in this
; section.

    .nolist
sp    =    0xf4
sph   =    sp+1
fp    =    sp-2
fph   =    fp+1
pc    =    fp-2
pch   =    pc+1
r1    =    pc-2
r1h   =    r1+1
r2    =    r1-2
r2h   =    r2+1
r3    =    r2-2
r3h   =    r3+1
r4    =    r3-2
r4h   =    r4+1
jp    =    r4-2
jph   =    jp+1
smask=     jp-2
smaskh=    smask+1
dsply=     smask-32
rp    =    dsply-2

; Specific equates for dealing with hardware.

; One sets up the address to write to by setting MELOW, MEMID, and MEHIGH,
; then reading/writing MEDATA.  Addresses are automatically incremented for
; sequential read/writes.

MEMSEL  =   0xc500      ; select ramslot hardware
MELOW   =   0xc0d0      ; low address byte of 1 meg space
MEMID   =   0xc0d1      ; middle address byte of 1 meg space
MEHIGH  =   0xc0d2      ; high address byte of 1 meg space
MEDATA  =   0xc0d3      ; data address for 1 meg space

; One tells the a/d to begin taking a sample by writing to a strobe
; address.  The chip takes 10/15 microseconds from being strobed until
; data is ready.  To determine the status of the card read the STATUS
; register.  When the data is ready bit 7 of STATUS will be high,
; At which point one can read the data bits.

STATUS  =   0xc0f0      ; address for A/D channnel status
STROBE  =   0xc0f3      ; address to tell A/D to take an 8 bit sample
AD_H    =   0xc0f2      ; address for high 8 bits of a/d channel
```

```
; The D/As are dumb chips.  They must be latched, because they contain
; no latches of their own.  There is no status line to check for these.
; The D/As may take up to two microseconds to settle on the appropriate
; value for a full scale deflection, so allow at least two microseconds
; between successive write cycles.

DA_H    =    0xc0f4       ; address for high 8 bits of d/a channel

    .list

    .even
    .entry _fastsample
_fastsample:

; This routine performs an eight bit sample using only the left channel.
; Even on its slowest loop the routine can store more than 100,000 samples
; per second.  The a/d chips on the card are only designed to process up
; to 100,000 samples per second.  This means the routine is faster than
; necessary.  The routine is written using a wait loop that checks the
; status of the a/d channel and waits until the card has completed acquiring
; a new sample.

;   fast pass: 20 cc + 3 usec : 8.56 usec -> 116.9 ks
;   slow pass: 25 cc + 3 usec : 9.94 usec -> 100.6 ks

    lda     #0                ; Inform c-shell we will not be sending
    tay                       ; back any parameters
    jsr     Alnk

    sta     0xc010       ; Clear keyboard strobe
fs7:
    lda     0xc000       ; Wait for a keypress
    bpl     fs7

    lda     #0
    sta     MEMSEL       ; select 1 meg ram hardware
    sta     MELOW        ; set address = 0x00000
    sta     MEMID
    sta     MEHIGH

    ldx     #0xff
fs6:
    sta     STROBE            ; Strobe to begin taking a sample
fs5:
    lda     STATUS            ; Check channel status
    bmi     fs5               ; If a/d not ready, wait
    lda     AD_H              ; Get sample value
    sta     MEDATA            ; Store value in memory
    cpx     MEHIGH
    bne     fs6
    ldx     #0xff
fs4:
    sta     STROBE            ; Strobe to begin taking a sample
fs3:
    lda     STATUS          ; Check channel status
    bmi     fs3             ; If a/d not ready, wait
    lda     AD_H            ; Get sample value
    sta     MEDATA          ; Store value in memory
    cpx     MEMID
    bne     fs4
    ldx     #0
fs2:
    sta     STROBE          ; Strobe to begin taking a sample
fs1:
    lda     STATUS          ; Check channel status
    bmi     fs1             ; If a/d not ready, wait
    lda     AD_H            ; Get sample value
    sta     MEDATA          ; Store value in memory
    cpx     MELOW
    bne     fs2

    jmp     Artn            ; Return to caller

    .even
    .entry _play8
_play8:

; This routine interprets the data on the ramcard as eight bit data, and
; outputs it to the D/A port.  An external reference is made to a variable
; called speed, which controls the delay between successive writes to
; the D/A port.

    lda     #0              ; Inform c-shell we will not be sending
    tay                     ; back any parameters
    jsr     Alnk

    sta     0xc010
play0:
    sta     MEMSEL          ; Select ramcard hardware
    lda     _START + 2      ; Fetch the start and end points of
    ora     #0xf0
    sta     MEHIGH          ; sample segment to be played.
    lda     _START + 1
    sta     MEMID
    lda     _START
    sta     MELOW

    lda     _FINISH + 2
    ora     #0xf0
    sta     r1
    lda     _FINISH + 1
    sta     r2
    lda     _FINISH
    sta     r3

    ldx     r1
play1:
    lda     MEDATA          ; Fetch byte to be ported
```

```
        ldy     _SPEED          ; Determine playback speed

play2:
        dey                     ; Delay till timing is right
        nop
        bne     play2
        sta     DA_H            ; Write byte to port
        cpx     MEHIGH          ; Check to see if MEHIGH matches FINISH
        bne     play1           ; If not loop

        ldx     r2
play3:
        lda     MEDATA          ; Fetch next data byte
        ldy     _SPEED          ; Fetch playback delay timer

play4:
        dey                     ; Delay till timing is correct
        nop
        bne     play4
        sta     DA_H            ; Write byte to port
        cpx     MEMID           ; Check to see if MEMID matches FINISH
        bne     play3           ; If not loop

        ldx     r3
play5:
        lda     MEDATA          ; Fetch next data byte
        ldy     _SPEED          ; Fetch playback delay timer

play6:
        dey                     ; Delay till timing is correct
        nop
        bne     play6
        sta     DA_H            ; Write byte to port
        cpx     MELOW           ; Check to see if MELOW >= FINISH
        bne     play5           ; If not loop

        lda     0xc000          ; Wait for a keypress
        bpl     play0
        sta     0xc010

        jmp     Artn            ; Return to calling function


        .extern _START          ; Where to begin playing back
        .extern _FINISH         ; Where to finish playing back
        .extern _SPEED          ; Controls rate of playback
```

/MQP/FFT

```
 10  DIM ST(2):ST(0) =  PEEK (49360):ST(1) =  PEEK (49361):ST(2)
=  PEEK (49362)
 100  TEXT : HOME : PRINT "FFT routine:"
 110  PRINT : PRINT "1 - 1024 point FFT"
 115 PI = 3.1415926
 120  PRINT "2 - 512 point FFT"
 130  PRINT "3 - 256 point FFT"
 140  PRINT "4 - 128 point FFT"
 150  PRINT "5 - 64 point FFT"
 160  PRINT "6 - 32 point FFT"
 170  PRINT "0 - Exit"
 200  VTAB 12: INPUT "Your choice? ";A$
 210  ON  VAL (A$) GOTO 300,320,340,360,380,400,220
 215  GOSUB 10000
 220  PRINT  CHR$ (4)"-STARTUP"
 230  END
 300 T = 1024: GOTO 1000
 320 T = 512: GOTO 1000
 340 T = 256: GOTO 1000
 360 T = 128: GOTO 1000
 380 T = 64: GOTO 1000
 400 T = 32: GOTO 1000
 1000  DIM A(T),B(T),F(T): VTAB 15: PRINT "Generating FFT
values..."
 1010  GOSUB 10000: PRINT "...reading data points..."
 1020  FOR I = 1 TO T:A(I) =  PEEK (49363): NEXT
 1080  PRINT "...generating FFT values": GOSUB 6000
 1085  PRINT "...storing generated values"
 1086  PRINT  CHR$ (4)"OPEN /RAM/FFT.DAT"
 1087  PRINT  CHR$ (4)"CLOSE"
 1088  PRINT  CHR$ (4)"DELETE /RAM/FFT.DAT"
 1090  PRINT  CHR$ (4)"OPEN /RAM/FFT.DAT"
 1100  PRINT  CHR$ (4)"WRITE /RAM/FFT.DAT"
 1110  PRINT T: PRINT ST(0): PRINT ST(1): PRINT ST(2)
 1120  FOR I = 1 TO T: PRINT A(I): PRINT B(I): NEXT
 1130  PRINT  CHR$ (4)"CLOSE"
 1135  GOSUB 10000
 1140  PRINT  CHR$ (4)"-STARTUP"
 5000  REM Conversion of FFT routine from Oppenheim and Schaffer
 5010  REM figure 6.18
 5020  REM a and b are meant to represent one complex array.
 5030  REM u, w, and t are complex scalars
 6000 LT =  LOG (T) /  LOG (2)
 6005  DIM P2(20):P2(0) = 1: FOR I = 1 TO 20:P2(I) = 2 * P2(I -
1): NEXT I
 6010  FOR L = 1 TO LT
 6020 LE = P2(LT + 1 - L):L1 = LE / 2:UA = 1:UB = 0
 6030 WA =  COS (PI / L1):WB =  -  SIN (PI / L1)
 6040  FOR J = 1 TO L1
 6050  FOR I = J TO T STEP LE
 6060 IP = I + L1
 6070 TA = A(I) + A(IP):TB = B(I) + B(IP)
```

```
6080 VA = A(I) - A(IP):VB = B(I) - B(IP)
6090 A(IP) = VA * UA - VB * UB:B(IP) = VA * UB + VB * UA
6100 A(I) = TA:B(I) = TB: NEXT I
6110 VA = UA:VB = UB
6120 UA = VA * WA - VB * WB
6130 UB = VA * WB + VB * WA: NEXT J,L
6180 J = 1
6190  FOR I = 1 TO T - 1
6200  IF I >  = J THEN 6240
6210 TA = A(J):TB = B(J)
6220 A(J) = A(I):B(J) = B(I)
6230 A(I) = TA:B(I) = TB
6240 K = T / 2
6250  IF K >  = J THEN 6280
6260 J = J - K
6270 K = K / 2: GOTO 6250
6280 J = J + K: NEXT I
6285  FOR I = 1 TO T:A(I) = A(I) / T:B(I) = B(I) / T: NEXT
6290  RETURN
10000  POKE 50432,0: POKE 49360,ST(0): POKE 49361,ST(1): POKE
49362,ST(2): RETURN
```

/MQP/PLOT

```
 10  LOMEM: 16384
 20  DIM ST(2):ST(0) =  PEEK (49360):ST(1) =  PEEK (49361):ST(2)
=  PEEK (49362)
 100  TEXT : HOME : PRINT "PLOT Utility:"
 105 PI = 3.1415926
 110  PRINT
 140  PRINT "1 - 1024 points"
 150  PRINT "2 - 512 points"
 160  PRINT "3 - 256 points"
 165  PRINT "4 - 128 points"
 166  PRINT "5 - 64 points"
 167  PRINT "6 - 32 points"
 170  PRINT "7 - Plot FFT data"
 175  PRINT "8 - Only Mag. with phase"
 178  PRINT "9 - View reconstruction"
 180  PRINT "0 - Exit"
 185  VTAB 14: INPUT "Your choice? ";A$
 190  ON 1 +  VAL (A$) GOTO 200, 250, 300, 350, 360, 370, 380,
5000, 6000, 8000
 195  GOSUB 10000
 200  PRINT  CHR$ (4)"-STARTUP"
 250 E = 1024: GOTO 400
 300 E = 512: GOTO 400
 350 E = 256: GOTO 400
 360 E = 128: GOTO 400
 370 E = 64: GOTO 400
 380 E = 32: GOTO 400
 400 S = 1024 / E: GOSUB 3000
 410  ON A1 GOTO 1000,2000
 1000  HGR : GOSUB 10000
 1010 EC = 0:SC = 0: HCOLOR= 3
 1020  HPLOT  INT (SC / 4),128 -  INT ( PEEK (49363) / 2)
 1030 SC = SC + S:EC = EC + 1: IF EC < E THEN 1020
 1040  GOSUB 4000: HOME : VTAB 22: PRINT "Waiting for a
keypress..."
 1050  IF  PEEK (49152) < 128 THEN 1050
 1060  POKE 49168,0: RUN
 2000  HGR
 2005  GOSUB 10000
 2010 EC = 0:SC = 0: HCOLOR= 3
 2015  HPLOT  INT (SC / 4),128 -  INT ( PEEK (49363) / 2)
 2020 SC = SC + S:EC = EC + 1: IF EC >  = E THEN 2040
 2030  HPLOT  TO  INT (SC / 4),128 -  INT ( PEEK (49363) / 2):
GOTO 2020
 2040  GOSUB 4000: HOME : VTAB 22: PRINT "Waiting for a
keypress..."
 2050  IF  PEEK (49152) < 128 THEN 2050
 2060  POKE 49168,0: RUN
 3000  VTAB 17: INPUT "Connect the dots? (Y/N) ";A$: IF  NOT ((A$
= "Y") OR (A$ = "y") OR (A$ = "N") OR (A$ = "n")) THEN 400
 3010 A1 = 1 + (A$ = "Y") + (A$ = "y"): RETURN
 4000  HOME : VTAB 22: INPUT "Grid lines? (Y/N) ";A$: IF  NOT
```

```
((A$ = "Y") OR (A$ = "y") OR (A$ = "N") OR (A$ = "n")) THEN 4000
 4010  IF (A$ = "n") OR (A$ = "N") THEN  RETURN
 4020  FOR I = 0 TO 140 STEP 12.8: HPLOT 0,I TO 255,I
 4030  NEXT : HCOLOR= 2: HPLOT 0,64 TO 255,64: RETURN
 5000  GOSUB 7000:PI = 3.14159265
 5010  HGR : HCOLOR= 3:N = 1
 5020  FOR I = 0 TO 255 STEP (256 / T)
 5030  HPLOT I,33 TO I,33 -  INT ( SQR (A(N) * A(N) + B(N) *
B(N)) / 8)
 5040  IF B(N) = 0 THEN A = PI / 2 *  SGN (A(N)): GOTO 5060
 5050 A =  ATN (A(N) / B(N)) / PI * 32
 5060  HPLOT I,64 TO I,64 -  INT (A)
 5070  HPLOT I,128 TO I,128 - ( INT ( SQR (A(N) * A(N) + B(N) *
B(N)) / 8) *  SGN (A))
 5080 N = N + 1: NEXT
 5090  HOME : VTAB 21: PRINT "Magnitude / Phase / Magnitude with
Phase"
 5100  VTAB 23: PRINT "Waiting for a keypress..."
 5110  IF  PEEK (49152) < 128 THEN 5110
 5120  RUN
 6000  GOSUB 7000:PI = 3.14159265
 6010  HGR : HCOLOR= 3:N = 1
 6020  FOR I = 0 TO 255 STEP (256 / T)
 6040  IF B(N) = 0 THEN A = PI / 2 *  SGN (A(N)): GOTO 6070
 6050 A =  ATN (A(N) / B(N))
 6070  HPLOT I,96 TO I,96 - ( INT ( SQR (A(N) * A(N) + B(N) *
B(N)) / 2) *  SGN (A))
 6080 N = N + 1: NEXT
 6110  IF  PEEK (49152) < 128 THEN 6110
 6120  RUN
 7000  PRINT  CHR$ (4)"OPEN/RAM/FFT.DAT"
 7010  PRINT  CHR$ (4)"READ/RAM/FFT.DAT"
 7020  INPUT T: DIM A(T),B(T)
 7025  INPUT ST(0),ST(1),ST(2)
 7030  FOR I = 1 TO T: INPUT A(I),B(I): NEXT
 7040  PRINT  CHR$ (4)"CLOSE"
 7050  RETURN
 8000  GOSUB 7000: VTAB 18: PRINT "Regenerating function";
 8001  REM Modified FFT routine by Oppenheim and Schaffer
 8002 LT =  LOG (T) /  LOG (2): DIM P2(10):P2(0) = 1: FOR I = 1
TO 10:P2(I) = 2 * P2(I - 1): NEXT I:PI = 3.1415926535
 8003  FOR L = 1 TO LT:LE = P2(LT + 1 - L):L1 = LE / 2:UA = 1:UB
= 0
 8004 WA =  COS (PI / L1):WB =  SIN (PI / L1): FOR J = 1 TO L1
 8005  FOR I = J TO T STEP LE:IP = I + L1
 8006 TA = A(I) + A(IP):TB = B(I) + B(IP)
 8007 VA = A(I) - A(IP):VB = B(I) - B(IP)
 8008 A(IP) = VA * UA - VB * UB:B(IP) = VA * UB + VB * UA
 8009 A(I) = TA:B(I) = TB: NEXT I
 8010 VA = UA:VB = UB
 8011 UA = VA * WA - VB * WB
 8012 UB = VA * WB + VB * WA: NEXT J,L
 8018 J = 1
 8019  FOR I = 1 TO T - 1
 8020  IF I >  = J THEN 8024
 8021 TA = A(J):TB = B(J)
 8022 A(J) = A(I):B(J) = B(I)
 8023 A(I) = TA:B(I) = TB
 8024 K = T / 2
 8025  IF K >  = J THEN 8028
 8026 J = J - K
 8027 K = K / 2: GOTO 8025
 8028 J = J + K: NEXT I
 8029  FOR I = 1 TO T:A(I) = A(I):B(I) = B(I): NEXT
 8050  HGR : HCOLOR= 3: GOSUB 10000
 8060 N = 1: FOR I = 0 TO 255 STEP (256 / T)
 8070  HPLOT I,128 TO I,128 -  SQR (A(N) * A(N) + B(N) * B(N)) /
2
 8080 N = N + 1: NEXT
 8085  HOME : VTAB 22: INPUT "Replace data on card? (y/n) ";A$:
IF  NOT ((A$ = "y") OR (A$ = "Y") OR (A$ = "n") OR (A$ = "N"))
THEN 8085
 8086  IF (A$ = "Y") OR (A$ = "y") THEN  GOSUB 10000: FOR I = 1
TO T: POKE 49363,A(I): NEXT
 8090  HOME : VTAB 22: INPUT "Superimpose original? (Y/N) ";A$:
IF  NOT ((A$ = "y") OR (A$ = "Y") OR (A$ = "n") OR (A$ = "N"))
THEN 8090
 8100  IF (A$ = "n") OR (A$ = "N") THEN  RUN
 8110 E = T:S = 1024 / T: GOTO 2005
 10000  POKE 50432,0: POKE 49360,ST(0): POKE 49361,ST(1): POKE
49362,ST(2): RETURN
```

/MQP/FILTER

```
 100  HOME : PRINT "Digital kill filter utility:"
 110  PRINT : INPUT "Threshold level? (0-255) ";A$
 120 A =  VAL (A$): IF A = 0 THEN  PRINT  CHR$ (4)"-startup"
 125  PRINT "Loading data..."
 130  PRINT  CHR$ (4)"open /ram/fft.dat"
 140  PRINT  CHR$ (4)"read /ram/fft.dat"
 150  INPUT T: DIM A(T),B(T)
 160  INPUT S0,S1,S2
 170  FOR I = 1 TO T: INPUT A(I),B(I): NEXT
 171  PRINT  CHR$ (4)"close"
 173  PRINT "Filtering..."
 175  FOR I = 1 TO T
 180  IF A >  ABS (A(I)) THEN A(I) = 0
 190  IF A >  ABS (B(I)) THEN B(I) = 0
 200  NEXT I
 210  PRINT "Storing data..."
 220  PRINT  CHR$ (4)"open /ram/fft.dat"
 230  PRINT  CHR$ (4)"write /ram/fft.dat"
 240  PRINT T: PRINT S0: PRINT S1: PRINT S2
 250  FOR I = 1 TO T: PRINT A(I): PRINT B(I): NEXT
 260  PRINT  CHR$ (4)"close"
 270  PRINT  CHR$ (4)"-startup"
```