# DESIGN DOCUMENTATION OF SIMPLESCOPE: LOW COST INTEGRATED HARDWARE/SOFTWARE DATA AQCUISITION SYSTEM FOR BIOLOGY

**A Design Project Report
Presented to the Engineering Division of the Graduate School
Of Cornell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering (Electrical)**

**By
Howard J. Marron and Giles J. Peng
Project Advisor: Professor Bruce Land
Degree Date: May 2003**

## Abstract

Master of Electrical Engineering Program
Cornell University
Design Project Report

**Project Title**: SimpleScope: Low Cost Integrated Hardware/Software Data Acquisition System for Biology

**Author**: Howard Marron and Giles Peng

**Abstract**:

Biologists need tools to measure the responses of living systems. In some cases, the desired responses must be stimulated out of the system by first sending some signal into the system and then recording the output. Current data acquisition systems made for this type of biological experiment are costly even for a low end product. To alleviate this need, Howard Marron and Giles Peng have designed a low cost integrated hardware/software data acquisition system. For capturing and measuring the responses, we created an oscilloscope, and for the stimulation, we created a stimulator. We combined software, hardware, and a 16-bit sound card to synthesize the end result which we have called "SimpleScope".

The oscilloscope side of the system required a high impedance amplifier to boost the signal and a hardware component to calibrate the signal voltage as read by the software. The stimulator required two control channels to enable output isolation: an amplitude channel and a timing channel. The software GUI was modeled after Bruce Land's StimScope interface. The initial intended users were neurobiologists, specifically, a neurobiology laboratory class. But the low cost of the system makes it also ideal for high school biology classes to utilize.

Report Approved by
Project Advisor: ______________________________________ Date: _____________
Project Advisor: ______________________________________ Date: _____________

## I. Executive Summary

In the biological field of study, many experiments involve measuring the responses of a living system. This is especially true in neurobiology; experiments require that a desired signal be measured from a neural system. And these desired signals usually need to be instigated by means of a particular pulse or “stimulation signal”. Thus, by stimulating the neural system, the desired response is fired by the system and can be measured and recorded. This exact experimentation process is used in of the neurophysiological laboratory classes offered at Cornell University.

There are some basic requirements for such an oscilloscope/stimulator system. For the oscilloscope, a high impedance amplifier needs to boost the signal going into the system, and a calibrator needs to adjust the signal voltage to reflect the actual voltages being measured. The stimulation part of the system needs to be electrically isolated from the input to minimize coupling of the large stimulating pulse to the sensitive recording electrodes.

Using publicly available software as skeletons for the oscilloscope and stimulator, a software program, very similar in function to the MATLAB StimScope, was created. From shareware found on the web, Giles used the Windows API support for waveform audio to communicate and use the sound card for both input and output. For the oscilloscope, Giles interfaced the GUI controls with many of the built-in API calls to manipulate the format of the input signal. Using the event loop structure of the Deeth Stereo Oscilloscope program, Giles added more detail and features to the drawing of the input waveform in the virtual oscilloscope window. Additionally, the bytes received from the sound card input buffer had to be manipulated in order to achieve the proper 16-bit stereo values needed. The stimulator aspect of the program involved building a wave format file, creating the shape of the waveform manually, playing the wave file from memory, and the GUI controls to manipulate the characteristics of the waveform. With the exception of programming the GUI control events to function exactly as specified, the stimulator development was very straightforward compared to the oscilloscope.

For the hardware, Howard had to design and build several different circuits. The first set was designed for the oscilloscope portion of the project and required an amplifier and a calibration channel. The amplifier was designed to filter and boost biological measurements so that a small differential signal can be converted into a voltage that an audio card can sample. And the calibration channel creates a small oscillating signal that has a calibrated output voltage, allowing the software to scale the input channel voltage based on a known reference. The other set of circuits designed was for the stimulator. The stimulator hardware takes an AC coupled signal from the computer’s sound card, converts it to a DC coupled signal, and creates an isolated voltage. This is achieved by the use of 2 output signals from the sound card: an amplitude control signal and a timing control signal.

The system worked well. The input calibration and amplification created a good signal in the software interface’s virtual oscilloscope window. During testing, the calibration signal had a peak-to-peak voltage of .08V. The stimulator’s output voltage was well isolated from the input and performed according to the expected timing and amplitude inputs received from the sound card. The output voltage swings from .8V to 7.8V and is linear except for the lower power region where it’s a low order exponential.

## II. Design Problem and System of Requirements

In the biological field of study, many experiments involve measuring the responses of a living system. This is especially true in neurobiology; experiments require that a desired signal be measured from a neural system. And these desired signals usually need to be instigated by means of a particular pulse or "stimulation signal". Thus, by stimulating the neural system, the desired response is fired and can be measured and recorded. This exact experimentation process is used in one of the neurophysiological laboratory classes offered at Cornell University.

There are some basic requirements for such an oscilloscope/stimulator system. For the oscilloscope, a high impedance amplifier needs to boost the signal going into the system, and a calibrator needs to adjust the signal voltage to reflect the actual voltages being measured. For the stimulation part of the system, it needs to be electrically isolated from the input to minimize coupling of the large stimulating pulse to the sensitive recording electrodes.

Our design goal was to create an inexpensive oscilloscope and stimulator through the combination of software, a typical sound card, and calibration hardware. This inexpensive stimulator and oscilloscope, named SimpleScope, needed to be predominantly software and needed to be able to use the typical sound card located in an IBM-compatible computer running some version of Microsoft Windows. Part of the design task was to make use of the sound card already available in most Windows systems today. Senior Research Associate Bruce Land had already created a MATLAB StimScope for use with a winsound card, a sound card in a Windows system, or a NIDAQ card. However, this MATLAB version could not fulfill the need for an inexpensive oscilloscope and stimulator due to the need for a costly MATLAB license and a NIDAQ card.

Having identified the overall design task, we split it into the two primary sub-tasks at hand: writing the software and creating the calibration hardware. Giles Peng took up the coding challenge, and Howard Marron took up the hardware challenge.

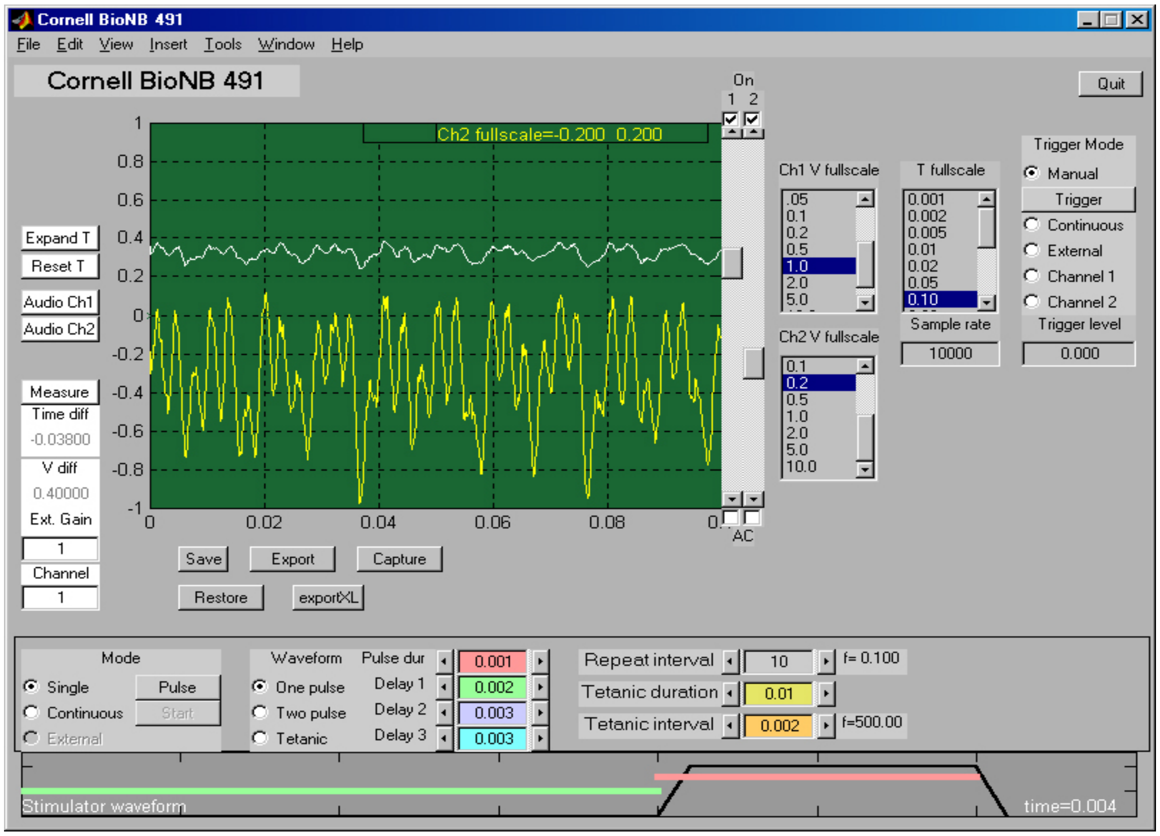
The MATLAB StimScope program consisted primarily of GUI controls and method calls to MATLAB's toolbox for winsound and data acquisition. As such, the actual code was not useful as a framework for our project. However, the GUI layout did provide the impetus for our SimpleScope's GUI.

The software in this project was modeled after the MATLAB StimScope project authored by Bruce Land. A new completely revamped version of the MATLAB program was needed because without a very expensive MATLAB license and a NIDAQ card, Professor Land's StimScope could not be used. StimScope was originally intended as an instrument for a Neurobiology Laboratory class in which nerves are stimulated and then reactions are measured using the oscilloscope. As such, the StimScope includes a stimulator, which requires the NIDAQ card to function properly. Included with our design objective of making a low cost oscilloscope was creating an accompanying stimulator for the purpose of use in the aforementioned class. This stimulator makes use of the line out port of the sound card, custom software, and calibration hardware.

So from the project description, the problem was to make a very inexpensive oscilloscope/stimulator using the sound card in a computer. This required software manipulation close to if not at the driver level of the sound card. Since both the oscilloscope and stimulator would be software driven, it was critical to select which language would be used for coding the software.

## Software

The primary software design problems were to create a GUI, which resembled the controls for a standalone oscilloscope as well as a stimulator, and to control the sound card. The design goal for the software GUI was intuitive and easy use. The MATLAB StimScope written by Bruce Land set the guidelines for the GUI. However, a few key features differed between the two specifications. The input part of the software (i.e. oscilloscope) would only have one channel as opposed to the two provided in the MATLAB program because a typical sound card only has 2 input channels and one of those channels has to be used for the voltage calibration hardware signal. The stimulator required use of 2 output channels: one for the amplitude of the output and one for the timing. These 2 signals would then be used by the output hardware to create the actual output. An image of the MATLAB StimScope GUI is included below:



Cornell BioNB 491

Ch2 fullscale=-0.200 0.200

Time diff -0.03800

V diff 0.40000

Ext. Gain 1

Channel 1

Ch1 V fullscale: .05, 0.1, 0.2, 0.5, [✓ 1.0], 2.0, 5.0

T fullscale: 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, [✓ 0.10]

Trigger Mode: [✓ Manual] Trigger, Continuous, External, Channel 1, Channel 2

Ch2 V fullscale: 0.1, [✓ 0.2], 0.5, 1.0, 2.0, 5.0, 10.0

Sample rate 10000

Trigger level 0.000

Mode: [✓ Single] Continuous External

Waveform: [✓ One pulse] Two pulse Tetanic

Pulse dur 0.001

Delay 1 0.002

Delay 2 0.003

Delay 3 0.003

Repeat interval 10 f= 0.100

Tetanic duration 0.01

Tetanic interval 0.002 f=500.00

time=0.004

Oscilloscope Interface Specifications

- Vertical voltage scale can be set from .001 to 5 volts.
- Horizontal time scale can be set from .001 to 30 seconds.
- Trigger Source:
    - **Manual:** User initiates capture of one virtual window's worth of data.
    - **Continuous:** Acquire data continuously.
    - **Channel 1:** Trigger when a certain voltage specified by the user is reached on the input channel.
- Sample Frequency can be set to 11.025 kHz, 22.050 kHz, or 44.1 kHz. The default is 11.025 kHz.
- One input channel on which an AC voltage can be captured.
- One channel to calibrate the input channel voltage.
- Using mouse clicks in the virtual oscilloscope window to measure absolute voltage and time differences.
- Exporting the data to an Excel type file with the 2 columns being time and input channel voltage. The number of points in the data set exported is the sample frequency multiplied by the per window time scale.

Stimulator Interface Specifications

- Modes:
    - **Pulse**: One pulse of the user-inputted waveform.
    - **Continuous**: Continuous repetition of the specified waveform with a user-selected repeat interval.

(Adapted from MATLAB StimScope Specifications @
http://www.nbb.cornell.edu/neurobio/land/PROJECTS/softscope/)

- The waveforms can be set to single pulse, two pulse or tetanic. A tetanic signal consists of a uniquely spaced initial pulse, a train of evenly spaced pulses, and then a uniquely spaced final pulse. **<u>In tetanic mode, the trigger continuous mode is disabled.</u>**
- All pulse delays and durations are set with a set of edit fields and left-right arrows. The stimulating waveform is drawn below the controls. The color of each time control corresponds to the color of a line on the drawing.
    - In One Pulse mode you can set "Pulse Duration" and "Trigger to Pulse" duration.
    - In Two Pulse mode you can set "Pulse Duration", "Trigger to Pulse" duration, and "1st to 2nd Pulse" duration.
    - In Tetanic mode you can set "Pulse Duration", "Trigger to Pulse" duration, "1st to 2nd Pulse" duration, and "Train to Last Pulse" duration. Additionally, you can set the "Tetanic duration", which is the length of all the pulses in the train, and the "Tetanic interval", which is the spacing of pulses within the train.
    - In all modes, you can set the repeat interval which determines how often the entire stimulus will be produced, if the trigger mode is continuous.
- The amplitude percentage for the resulting output can also be adjusted using a slider from 0% to 100%.

## Hardware

The input and calibration portions of the hardware were designed and built first. The system placed on the input side from the measured input to the audio-in jack of the computer sound card would contain two main circuits. The first, the signal path, would convert the input from a differential voltage to an single band voltage, provide some gain, and then filter out the signals outside of range of the audio capture devices and biological needs. The second channel was necessary to be able to calibrate the overall gain from the inputs to the sound card.

Another option that changed during the design was the amount of gain that would be applied. Originally there would be an optional switch that would allow the user to change the gain from 100x to 1000x. Accompanying this, the calibration signal would change in such a manner so the software could detect the change and adjust the readings accordingly. When the design was tested, the gain change was deemed not necessary for several reasons that will be discussed in the solutions section.

The purpose of the calibration channel is to give a signal to the software that allows for the compensation of variations in components placed on the board (varying gains of op-amps due to components) and variations in audio capture cards (different cards will read the same voltage at a different level). The requirements of the calibration channel remained the same throughout the project. The calibration channel had to output a square wave that was within the reading envelope of the audio card on most computers. In addition the frequency of the signal had to be chosen so that the device could sample the signal but still be fast enough so that it would not be affected by the AC coupling of the audio devices. The signal frequency had to be high enough so at least one full cycle of the calibration channel would be contained in any user-chosen time scale. This is necessary because the software will recalculate the calibration every time it recaptures the data, so there must be at least one minimum and maximum of the calibration signal to be able to get the information from the signal.

Other requirements of the input and calibration circuits were that they could be combined and the total power consumption is low enough to run for an extended period of time (i.e. months) on batteries. It was decided that a power supply of +5V and –5V would be satisfactory.

The output channel is a set of circuits that will take an audio signal from the line out of the sound card and convert this to a timed variable voltage output. The requirements for the signal from the audio card called for using both channels of the stereo sound to create the two needed signals: one for the timing of the output signal, and another one for the amplitude. To get timing, one channel will output a signal that will oscillate when the output should be on. The envelope of the signal will be used to determine if the output should be on, so the best timing that is possible is the resolution of the audio channel, about 44khz. In practice the best resolution of the signal is closer to around 10khz. The signal for amplitude would be filtered to take the envelope of the signal. The fraction of the maximum possible audio output would be the desired fraction of the output of the circuit. The timing signal would be another signal to be filtered so that on the rising and falling edges of the signal envelopes the output would turn on and off respectively.

The two signals for the output stage after being converted to a DC coupled signal will be sent to the stimulator. The stimulator is a DC-to-DC converter with a variable resistor to tune the range of voltages. The requirements of the output voltage were that the output voltage be independent of the voltage of the rest of the circuit, and the output voltage be in a range of 0V to 10V.

**Main Design Specifications and Issues**

- Inexpensive, primarily software-based oscilloscope/stimulator
- Combine software, 16-bit sound card, and hardware to synthesize this “SimpleScope”
- Tool for neurobiology laboratory class

Software-specific

- Graphical User Interface similar to the MATLAB StimScope by Bruce Land
- One input channel and one calibration channel for oscilloscope
- One output channel for amplitude and one output channel for timing (stimulator)

Hardware-specific

- Oscilloscope (input)
    - Hardware gain (vs. software gain)
    - Voltage regulation
    - Specifications for good calibration outputs
    - Input filtering
    - Power usage

- Stimulator (output)
    - How to convert audio signals to usable waveforms
    - Output isolation

## III. Solutions

### Software

A number of different languages could have been used for the software coding. Because there is such a large GUI element inherent in the design, it really didn’t make sense to choose a language that did not easily support graphics and graphical controls; this basically left four reasonable software language choices left: Visual C++, Java, and Visual Java, and Visual Basic. Of the three, I am most proficient in Java, so that seemed like a natural choice, but Java is too slow for the low level operations that needed to be done at the sound card driver level. It seemed Visual C++ would be an excellent tool to use, but in my own experience as well as that of Professor Land’s, we found visual basic to be better adapted for creating visual/GUI oriented programs. So we chose visual basic as the language for the SimpleScope.

## Oscilloscope Solutions

First, I researched what was already available in the way of a software oscilloscope in the public web domain. There were several offerings. However, a few of the possibilities were very rudimentary. However, I was able to glean a great deal of knowledge about how to interface with the sound card from these programs. During my research, I realized I would need to either develop a driver for the sound card or find a third party driver available to the public. It seemed unreasonable to write a driver for the sound card, so I went for the latter option.

Initially, I used a driver authored by He Lingsong, a professor in China. I obtained this driver from his site http://heliso.tripod.com. Though the driver seemed to be very versatile and even seemed to accommodate the output needs of our project, it turned out to be very slow. In the prototype of this solution alternative, seconds and seconds of data were missed while trying to capture data continuously.

The other driver I find in my research was a data acquisition library for visual basic. This dynamically linked library (DLL) was written in C (available from http://www.ines.com) , so in accessing the DLL, the C declarations had to be a converted to a VB format. This library seemed to be very clean, and by all appearances, I thought it would be much more efficient than the previous driver. However, upon testing a simple prototype for continuously capturing data, I found the same problem as the previous driver; chunks of data were missed entirely.

Accessing a DLL through visual basic was much too slow. The time it took for the visual basic to communicate with the DLL was greater than the time it took to capture a chunk of data, meaning filling the I/O buffer of the sound card. Consequently, chunks of data were being overwritten in the buffer before they were read out of the buffer; therefore, that data was lost. Clearly, all the software including the sound card access needed to be entirely encapsulated in visual basic. As such, this meant that a visual basic driver would have to be written, but most drivers if not all drivers in a Windows system are written in C. There was a dilemma. More research on the web through forums and shareware libraries didn't help. Then my advisor found a very obscure website which had a very, very simple oscilloscope written entirely in visual basic. The program, Deeth Stereo Oscilliscope ( http://www.fullspectrum.com/deeth/deethware/#Scope ) authored by Murphy McCauley, used the Windows API to perform basic wave functions, which could be equated to measuring voltages through the line-in port on a sound card. Testing the program, we easily saw that it was able to capture all the data in continuous mode. Using Deeth Stereo Oscilliscope as the skeleton of the input software, I coded the rest of the functionality into the code adding and removing code accordingly. It is important to note that the input was developed separately from the output code in order to keep the development logical and more importantly debuggable.

## Stimulator Solutions

The output seemed more straightforward than the input due to all the knowledge collected from the input development. However, additional research still had to be done. One means of generating a stimulator signal involved writing to the output buffer of the sound card using a circular buffer technique. Using this method, the signal would be continuously generated and written out on the fly as long as it was desired. Though this seemed to be a very elegant solution, getting it to work, especially with the visual basic and only visual basic framework proved to be a bit time consuming. More importantly, this method seemed to be unnecessary because the exact benefit of on-the-fly signal generation was not needed as specified in the stimulator design task. A fixed pulse or pulse train was to be generated every set interval. The specification indicated that the stimulator signal would be just a repetition of the same set signal.

This specification led more naturally to producing the signal once and looping it somehow. Fortunately, the windows API allows for this exact capability within its waveform audio output support section. Again, I thought there might already be a simple implementation of this publicly available. And indeed there was; I found a visual basic program along with source code to generate a waveform and to repeatedly play that waveform. This programmed named "Tones" was originally intended as a tone tester (authored by David M.Hitchner and @ http://www.thescarms.com/VBasic/tone.asp ). Refer to appendix II.4 for the code. The code allowed a waveform to be generated and either saved to file or main memory. For our design specification, it was most relevant to save the waveform, the output signal, to memory and repeatedly loop it. Using this code, which is included in the appendix, as the skeleton for the output program, I added the appropriate GUI interface for the stimulator and some additional functionality to meet the design specifications of the project. This additional functionality included generating the different signals to actually be output and generating oscillation underneath the signal envelope for purposes of hardware synchronization.

A good ways through the development of the oscilloscope software, I wondered why I didn't use visual C++ instead of visual basic. Many of the tasks would have been much more straightforward to program and indeed possible if I had used C because it is such a low level language. I really wasn't able to efficiently program a lot of low level tasks with visual basic. And I spent a considerable amount of time trying to figure out how to program a lot of functionality for which visual basic just isn't suited. C on the other hand would've been able to handle these low level procedures much more effectively. Initially, neither Professor Land nor I knew of the visual capabilities of Visual C++; in retrospect, the visual components of Visual C++ were probably as good if not better than visual basic. Visual C++ might have been a much better language choice given that I had to start from knowing so little about designing this particular software.

## Hardware

### Oscilloscope solutions

Input signal paths. The requirements of the circuits were simple and are easily created with several integrated circuits. So the solution we arrived at was a modification of Bruce Land's amplifier that he had previously designed (that project is listed here: http://www.nbb.cornell.edu/neurobio/land/PROJECTS/PREAMP/). The total gain of the final design is 40 times (10x in differential amplifier, and 2x in both remaining stages). This works because of the 16-bit input and properly placing the voltage within the inputs of audio cards. At the end of the amplifier stages, we added a 300-ohm resistor to prevent last op-amp from oscillating. We found that the large capacitive input impedance of the audio cards can cause the last stage of the preamp to osscilate, latch and destroy itself; the additional resistance prevents this from happening.

Calibration channel. There were two considerations that had to be chosen for the calibration channel: the frequency of the signal and the method to regulate the output voltage of the calibration channel. The basic design of the circuit was chosen because of its simplicity. The requirements needed a low power timer, so a 7555 timer was used instead of a 555 timer to reduce the maximum power used by the circuit.

The choice of frequency of the calibration signal had several requirements. The signal has to be fast enough to not be affected by the AC coupling of the sound cards, and slow enough to be accurately captured by the sound cards. Through experimentation we found that 1500 Hz was a good result. The design of the circuit is taken from the 7555 datasheet.

## Stimulator solutions

The computer hardware output a signal that was AC coupled with a maximum peak to peak voltage of about 1V – 1.5V. To compensate for the inability of AC coupling to send low frequency signals, a higher frequency signal is used and the envelope of the signal is used instead of the edges to determine the timing of the output. The AC-coupled signal must be converted to a pulse that is usable by digital logic. Since the peak-to-peak voltage of the AC signal was not significantly larger than the voltage threshold of integrated circuits, we used an active circuit scheme to convert the signal. This solution to the AC-DC conversion used a fast active rectifier and a gain stage to amplify to the proper levels.

The rectifier was very responsive up to the 44khz max frequency of audio signals, allowing for the rectifier to be used for both control signals of the output. The rectifier did not add noise that was unmanageable to the channels. The signal that comes out of the rectifier is properly rectified, but has to be filtered and amplified before being used in other circuits. The rectifier had a high enough frequency response that the oscillations of the input were still evident in the rectified signals. So from this stage the signals had to be amplified from the one-two volt range to the five-volt range, and the signals had to be filtered to remove the high frequency noise.

Please refer to Figure 4 in appendix I for a detailed output schematic.

## IV. Design and Implementation

Using publicly available software as skeletons for the oscilloscope and stimulator, a software program, very similar in function to the MATLAB StimScope, was created. From shareware found on the web, Giles managed to use the Windows API support for waveform audio to communicate and use the sound card for both input and output. For the oscilloscope, Giles interfaced the GUI controls with many of the built-in API calls to manipulate the format of the input signal. Using the event loop structure of the Deeth Stereo Oscilloscope program, Giles added more detail and features to the drawing of the input waveform in the virtual oscilloscope window. Additionally, the bytes received from the sound card input buffer had to be manipulated in order to achieve the proper 16-bit stereo values needed. The stimulator aspect of the program involved building a wave format file, creating the shape of the waveform manually, playing the wave file from memory, and the GUI controls to manipulate the characteristics of the waveform. With the exception of programming the GUI control events to function exactly as specified, the stimulator development was very straightforward compared to the oscilloscope.

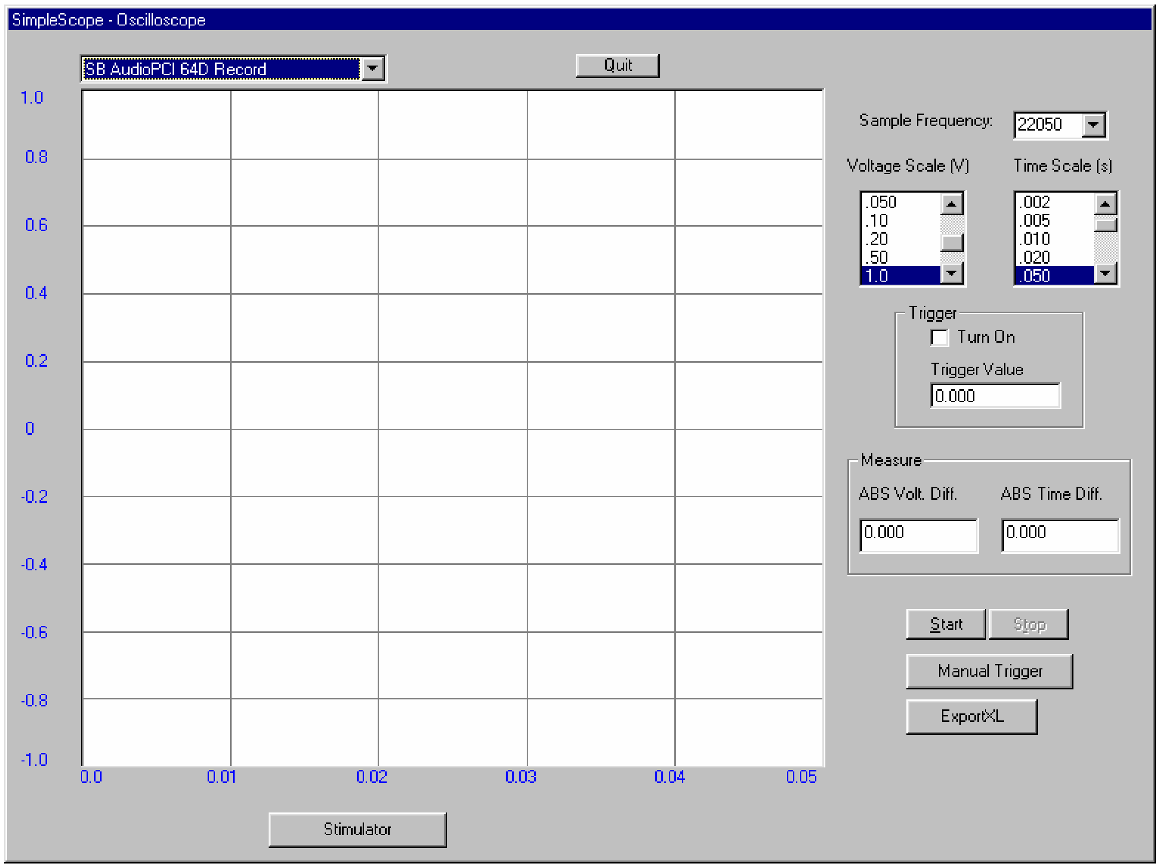
For the hardware, Howard had to design and build several different circuits. The first set was designed for the oscilloscope portion of the project and required an amplifier and a calibration channel. The amplifier was designed to intensify low frequencies so that a small differential signal can be converted into a voltage that an audio card can sample. And the calibration channel creates a small oscillating signal that has a calibrated output voltage, allowing the software to have a known voltage with values can be absolutely specified. The other set of circuits designed was for the stimulator. The stimulator hardware takes an AC coupled signal from the computer's sound card, converts it to a DC coupled signal, and creates an isolated voltage.  This is achieved by the use of 2 output signals from the sound card: an amplitude control signal and a timing control signal.

## Software

As noted earlier, the SimpleScope GUI layout is very similar to Bruce Land's StimScope. However, some important changes have been made.  The oscilloscope portion of the program can only accommodate one channel.  This is due to the hardware requiring one of the channels for voltage calibration.  This channel is transparent to the user.  Please refer to the hardware implementation of the input for more details.  The GUI is meant to be a little more intuitive than in the previous version of SimpleScope.

## GUI Design

### Oscilloscope Window Screenshot



SB AudioPCI 64D Record

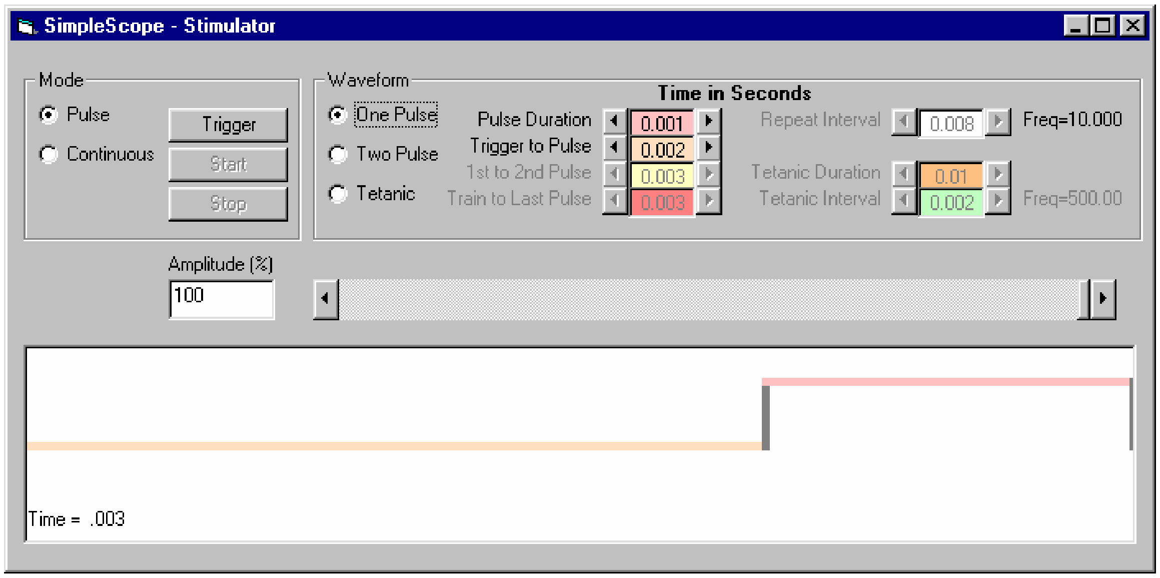
Sample Frequency: 22050

| Voltage Scale (V) | Time Scale (s) |
|---|---|
| .050 | .002 |
| .10 | .005 |
| .20 | .010 |
| .50 | .020 |
| 1.0 | .050 |

Trigger
[ ] Turn On
Trigger Value: 0.000

Measure
ABS Volt. Diff.: 0.000
ABS Time Diff.: 0.000

### Stimulator Window



Mode
[✓ Pulse]
Continuous

Waveform
[✓ One Pulse]
Two Pulse
Tetanic

Time in Seconds

| | | | | |
|---|---|---|---|---|
| Pulse Duration | 0.001 | Repeat Interval | 0.008 | Freq=10.000 |
| Trigger to Pulse | 0.002 | | | |
| 1st to 2nd Pulse | 0.003 | Tetanic Duration | 0.01 | |
| Train to Last Pulse | 0.003 | Tetanic Interval | 0.002 | Freq=500.00 |

Amplitude (%) 100

Time = .003

The list box in the upper left hand corner allows the user to choose the sound card to use for SimpleScope if more than one sound card is available for use in the computer. The quit button allows the user to exit the program only if the oscilloscope is not capturing any data and the stimulator is stopped. Sample Frequency allows the user to choose between 11025 Hz, 22050 Hz, and 44100 Hz for the frequency used for data acquisition. Voltage Scale allows the user to choose the vertical scale in the virtual scope window ranging between .001 to 5 volts. The Time Scale gives the user to the option to choose between .001 to 30 seconds for the horizontal scale in the virtual scope. The trigger box allows a trigger level to be set in continuous data acquisition mode if the trigger turn on check box is checked. The measurement box gives the user more data by providing the absolute voltage and absolute time differences between 2 user selected cursor points. The oscilloscope start button initiates continuous data acquisition while the manual trigger button captures one virtual scope window's worth of data and then stops capturing. The stop button terminates continuous data acquisition mode.

In the stimulator portion of the GUI, the 2 options in the mode box give the user the choice of a single pulse of the appropriate signal or a continuous generation of that signal. For the selection of each of the 2 modes, the irrelevant stimulator controls are grayed out and disabled and the relevant options are enabled. The trigger button which is enabled in Pulse mode allows one pulse of the appropriate waveform to be outputted upon clicking. The start button is enabled in continuous mode and initiates a continuous signal of the appropriate signal when clicked; clicking also enables the stop button, which stops the continuous signal generation.

In the waveform box of the stimulator, the three options for the signal/waveform type are one pulse, two pulse, and tetanic. The appropriate timing options are disabled and enabled according to which pulse mode is selected. The 7 time options can be adjusted when enabled by clicking on the arrow cursors or by clicking in the text box and entering the appropriate numbers. The left arrow cursor for each time halves the current time and the right arrow cursor doubles the current time.

The amplitude slider controls the amplitude of the output signal. The slider is on a relative scale meaning it runs from 0 to 100%. The waveform visual below the slider is one repetition of the signal to be generated by the stimulator. The colors of the time fields correspond to the colors in the waveform depiction so that each period of time can be visually understood if not by its description. <u>The graphic is an ideal representation of the waveform</u> which does not consider the actual fall and rise times which will occur from the hardware output.

## Oscilloscope

Visual Basic is an event driven language, so all programs must have an event driven structure. For the oscilloscope portion of the code, interacting with the visual components triggers events which in some cases have calls to other procedures. The start button for continuous data acquisition triggers a call to a procedure that contains a loop which successively waits for the sound card's input buffer to be full, reads the bytes out, and draws them in the virtual scope window. The loop is only exited upon clicking the stop button or if the program exits abnormally. This notion of a repeated event loop originates from the Deeth Stereo Oscilloscope code (refer to appendix for code and some documentation). This loop is a critical section because as happened with the DLL approach to the input, if the input buffer is overwritten by the sound card before the information can be read out, data is lost.

The evolution of the program structure provided a great learning opportunity to understand software development for a system in great detail. Starting with the Deeth Stereo

Oscilloscope code, I perused the 5 to 6 pages of code until I understood what every line did and how that might contribute to the needs I had. Indeed, much of the code proved very useful in the SimpleScope design because the original program was also an oscilloscope though a very simple one. The superfluous code was then removed, and what was left was used as the skeleton of the SimpleScope code. After adding a function or a GUI control event, I would perform a rudimentary test on its functionality before adding another function. In this way, a minimal modularization of the development process could be kept. Unfortunately, the test of the hardware with the software could not take place for the most part in this step-by-step fashion due to the need for the software to be very functional; the software needed to be fully interactive with the sound card and to be able to draw the signal in the virtual scope window. After we were able to start testing the software and hardware together, a lot of debugging and software redevelopment occurred. We were able to test that the voltage calibration channel was actually scaling the input channel correctly; these results are presented in the next section.

The program does not implement asynchronous reading of the sound card's input buffer. This proved to be extremely difficult to implement because the low level construct needed could not be easily programmed in visual basic. According to Microsoft's Development notes for the waveform audio support API, at least a double buffering scheme is needed to ensure gapless recording through the input buffers of the sound card. This double buffering scheme is exactly what proved to be difficult to synthesize in visual basic due to the primitive form of arrays and array manipulation. Additionally, the design specifications didn't specifically require gapless capturing of the input. In general, oscilloscopes don't have a gapless-stream capturing capability; they capture and process, capture and process. Though originally I had thought the SimpleScope would capture gapless streams of input, it turned out in later discussion that this feature was not needed because most of the data measured and captured by an oscilloscope is just a repetitive signal or a pulse waveform of some relatively short duration of time. These data requirements or "non-requirements" allowed the program to be in the synchronous form it's in now. So the trade-off was less code complexity to have a feature that wasn't needed anyways.

As described before, the window's API was used to take advantage of the waveform audio support for sound input. Many of the data structures needed by the functions in the interface were defined in the Globals sections of the code. The portion of the Window's API used is included in the appendix. The WaveFormatEx type included the number of channels, the samples per second, bits per sample, and average bytes per second fields so that the captured waveform could be fully specified. Each wave data chunk also needed a Wave Header to provide the sound card control mechanisms with information about the status of the input capture and the status of the actual data being captured.

The program begins with initializing many of the GUI elements when the form loads. Also the existence and capabilities of the sound card(s) in the computer is/are checked during this initialization stage. And finally, the list boxes in the application are filled in with the appropriate options. It is important to note that before initializing any data capture, the sample frequency and time scale options are set, so when a capture is started these attribute values are locked. Capturing must be halted to change these values, and the capture restarting to make use of a different sample frequency or time scale. The voltage scale as well as the channel trigger and its trigger level can be changed at any time during the capture (i.e. in mid-capture).

Visualize() and DrawData() are the key functions that actually do the capturing through the sound card and drawing the data on the scope window, respectively. When the data is captured in the visualize() function, the raw data is calibrated using the calibration channel. The number of 16-bit levels per calibration voltage is used to scale the data. Then that data is stored as the actual voltages. In the DrawData() function, the voltages are drawn on the screen by drawing a line from point to point. For small numbers of points, all the data is drawn on the screen. But as the number of data points surpasses the number of pixels available in the virtual oscilloscope window, only every n data points is plotted so that there are approximately 2 data points for each pixel. This saves large amounts of time because drawing the signal on the screen is the primary bottleneck.

The ExportXL command just writes the input data out to a tab-delimited file with an .xls extension. This outputted data set is complete, meaning all the sample points captured are included. The Mouse_Down and Mouse_Up events of the oscilloscope window facilitate the means of measuring the absolute voltage and absolute time between 2 points in the virtual scope window just like cursors on a real oscilloscope.

## Stimulator code internals

As with the input, the output's program structure was largely determined by the skeleton framework found in the Tones program. The Tones program creates a wave format sound file and plays it by means of the windows waveform audio support methods. I customized this program to create the waveforms needed for the stimulator output and to interact with the GUI.

The Tones program is encapsulated in a module for better abstraction. The basic module is named wav.bas. Initially, the wave format header has to be built before any wave data is created. So the header is built upon a call to the module. Then a call to MakeBasisWave is made in which the basis waveform for the stimulator output signal is constructed based on a single pulse, two pulses, or a tetanic pulse and the 7 durations which define the exact characteristics of the output signal. The basis waveform is generated based on a 1 kHz oscillation when the signal is high and a 0 when the signal is low. The output comes out of both channels. One channel called the synchronization has all the proper timing while the other channel called the amplitude channel outputs the scale of the output signal. These two signals are passed to the hardware where the information is used to make a properly calibrated and timed output signal. Details to what happens in the output hardware can be found in the design and implementation of the stimulator hardware. Once start or pulse is clicked, the waveform is played repeatedly or once, respectively.

With 8 bits for each channel, the output is accurate enough so that the calibration hardware can make the proper full-swing DC signal. The output code is very straightforward with the aforementioned basic structure. The comments in the source code should enlighten any of the small details unclear in the code.

## Hardware

### Oscilloscope circuits

The implementation used for the input signal circuit was a modification of Bruce Land's previously designed amplifier: Physiological AC Preamplifier (more information available at http://www.nbb.cornell.edu/neurobio/land/PROJECTS/PREAMP/). The bandwidth needed for our circuits was low enough that we were able to use the lmc7111, which is also low current. The op-amps used had a gain-bandwidth product of 40khz which provided enough bandwidth considering the sample frequency of the audio input cards was a maximum of 44khz, so maximum bandwidth of our design is 20 khz. The differential amplifier is given a gain of x10 with a 5.6kohm resistor, and the subsequent op amps were set up with a gain of 2x each using

10k and 20k ohm resistors for the second stage, and 100k , 200k ohm resistors for the last. This gives a total amplification of 40x the original signal. The gain from the INA121 is given by (1+ (50kohms/Rgain)). We decided that the optional gain created with a switch on the last op-amp gain stage was not necessary, since the computer can read the channel to 16 bits of accuracy, which translates to a software zoom of 256x for 8-bit accurate data. The power supply for the input circuits needs a positive and negative power supply. The circuits can run on power supplies from 4.5V to 6V, so standard battery packs can be used.

## Stimulator circuits

There are two signals sent from the computer to the stimulator must be converted from osscilating AC signals to the DC pulses that represent their envelopes. Two active recitifiers are used, one for each signal, both using the same design. The rectifiers design (figure 3 in the appendix) works well up to the maximum frequency output of the audio channels, and works for the low voltage of the channels.

The paths of the different signals used for the output begin to differ here. The amplitude signal can be sent through filters that have a very high time constant since the signal will not be changing on a short time scale. We want the rectified amplitude signal to be as constant as possible, so the filtering is necessary to remove the switching noise which came through the rectifiers since we used the same fast rectifiers for both channels of the stimulator input. To make sure that the user has a lot of granularity in the amplitude choices the gain of the amplitude signal must be chosen to make sure that when the inputs are changed the outputs change linearly and do not reach the maximums too early. This is accomplished by using a gain of two through an op-amp and then put through a low pass filter before being sent to the stimulator amplitude control. On the other hand, synchronization signal cannot and does not need to be linearly filtered since the signal must be transmitted fast, so the signal is just amplified and send to controls of the DC/DC (DCP010515DP) chip. Since the synchronization signal is supposed to be a binary signal on or off, we want to maximize the gain after the rectifiers to make sure the signal is at either Vdd or Vcc, so a gain of over 4 is used to ensure this.

The two signals for synchronization and amplitude are passed on to the stimulator circuit that is taken from a project by Bruce Land
(http://www.nbb.cornell.edu/neurobio/land/PROJECTS/Stimulator2/)
The idea of this circuit is to create an isolated voltage that can be controlled quickly and independently. So the Burr-Brown DCP010515DP chip creates an isolated voltage when the sync pin is allowed to have a floating voltage, created with the 4066 switches. This voltage output is sent through a voltage divider that is controlled by the H11F1 optocoupler, which isolated the inputs from the controlled resistance of its output. The stimulator's power supply runs on a supply voltage that can range from +4.5V to +6V, but a higher supply voltage will give better voltage output waveforms.

## Hardware Implementation Problems

When running the stimulator, the Burr-Brown DCP010515DP chip drew a lot more current than expected and required a 2uF capacitor to decouple the parasitics from heavily affecting the signal. The symptoms for this was excessive noise on the power rails when the device turned on and off, causing feedback that ruined the output. Even with the 2uF cap for decoupling the amplifier on the input side, the stimulator cannot be run off the same power supply as the input Vdd. So three separate power supplies are needed, two for the input, and one for the output. The input will work on either 6V or 4.5V power supplies equally well. The output will work on both, but I would recommend the 6V supplies since the output voltage is directly proportional to the supply.

After being put through the H11F1 optocoupler, the isolated voltage output was not as large as expected. A voltage of around 20V was expected since the DC/DC chip outputs around 25V, but the circuit only outputted around 9V. The culprit was the H11F1 chip not receiving enough current (~40mA) to reach its lowest resistance. To raise this the small resistor (100ohms) in this circuit should be made smaller, and the corresponding output will increase.

## V. Testing

### Software

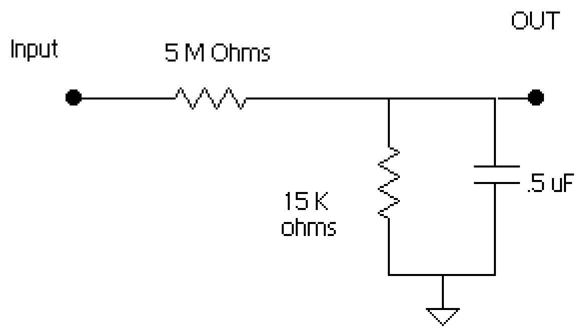
With software, testing is always taking place because a program has to be able to compile and run. Throughout development, the software has been tested for functionality and correctness. Verification is built-in to the software development process. However, all software can always be continually developing. Though initial specifications were given, when it comes to software, there is a whole range of specification-satisfying solutions.

Having that in mind, we'll take a look at the tests to verify the functionality of the software. For the oscilloscope, prototypes were made early on to make sure that some of the fundamental specifications could be met such as being able capture data continuously without any loss. This was a requisite to make any solution alternative a plausible solution for the final product; Deeth Stereo Oscilloscope fulfilled this requirement, so using this software as a base for the rest of the program ensured some degree of functionality in the final product. As each GUI element was added and its event calls associated with procedures, small tests were carried out to ensure the desired and expected functionality. For instance, changing the sample rate of the capture had to increase the accuracy of the voltage measurement on the virtual scope window, so using a standard signal, I checked the number of bytes captured for each sample rate and made sure that for each sample rate doubling bytes captured also doubled. To test the input as I developed, I used a simple square wave signal produced by a small 555 timer circuit. To test the compatibility of the software on different computers, I ran it on 4 different computers: a Windows XP system, two differently configured Windows 98 systems, and one Windows NT system. The program ran the same across 3 of the systems, but it had some major problems running on the Windows 98 system in the lab where the program would actually be used. So I needed to make sure the software ran on that Windows 98 system. The code had to be rewritten to some extent to solve this problem.

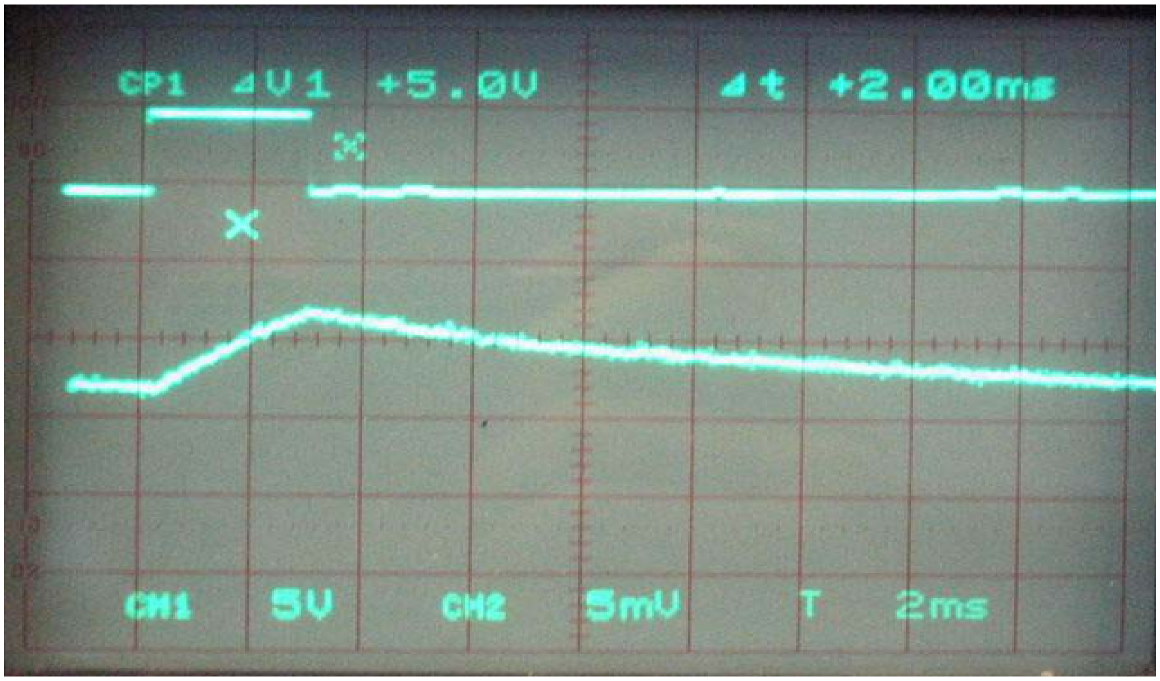
The waveInAddBuffer() API call would freeze the computer when a dynamically allocated array was used for the buffer to be added. This problem only occurred in the lab computers. So to solve this problem, I had to turn to the less elegant solution of creating the arrays associated with that API call statically and ensuring the array length was large enough to accommodate the largest buffer length needed in the program. This size ended up being 5292000 bytes. Though inelegant, this was a working solution for the lab computers.

For the stimulator, I used a hardware oscilloscope to check that the signal I expected the software to generate matched the signal coming out of the sound card. Indeed for all variations of the one pulse, two pulse, and tetanic signals, the proper functionality was being achieved. Once we began testing the stimulator software with the calibration hardware, improper software functionality was immediately obvious. I had to revise the software functionality quite a few times to achieve the best results with the hardware. We verified the signal at different stages in the hardware processing on the oscilloscope as well. As for the output before hardware processing, I thought the pulses would be very square, however, this turned out not to be the case; the pulses were actually noisy square pulses.

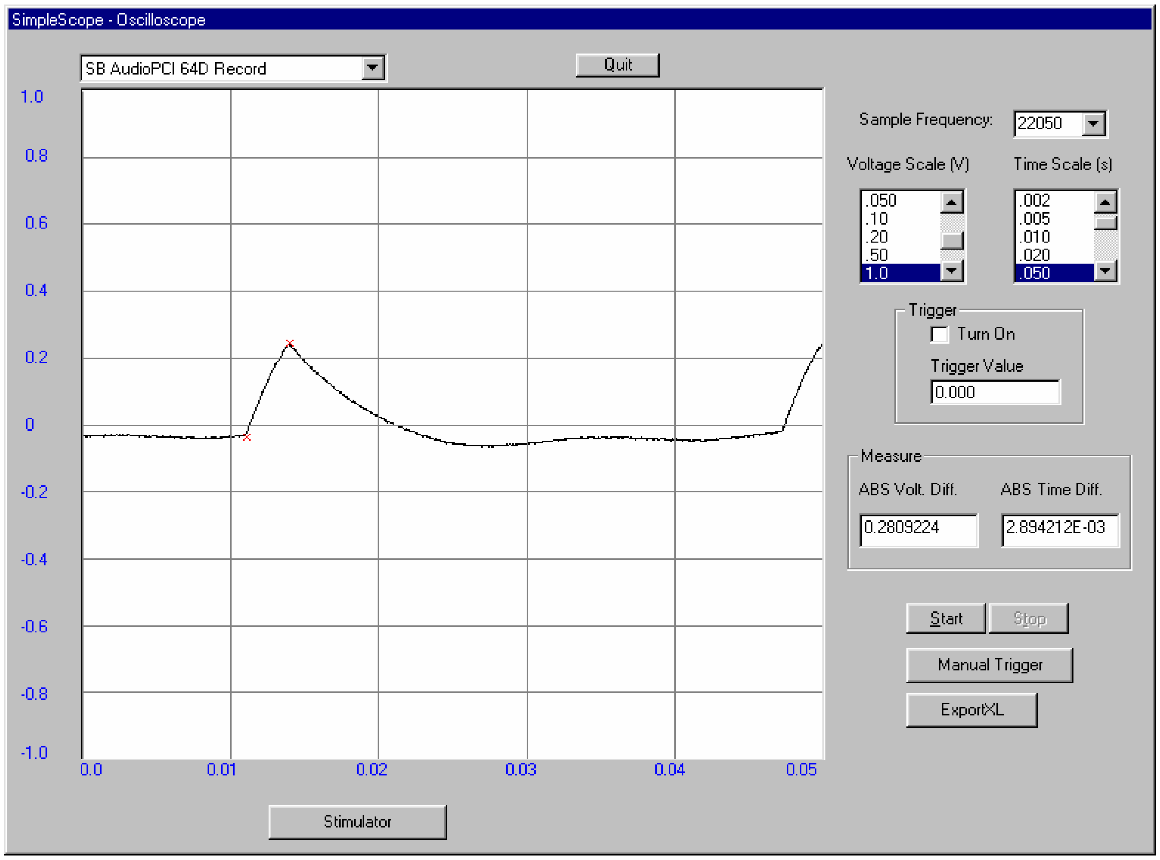
We tested the oscilloscope part of the system (software and hardware inclusive) with a function generator and a circuit representing a neural system (refer to the figure for the circuit). The expected output from square wave stimulation is shown below along with the input stimulus on the standalone oscilloscope. Comparing the SimpleScope's trace to this expected output, the results are very similar. The timing is very accurate. And the voltages are different because the software oscilloscope is reading a signal with gain. Additionally the variable resister on the calibrator can be adjusted for each system. Looking at the 2 figures, we can see that the rise time for the pulses on the SimpleScope is about 2.9ms while the stand-alone oscilloscope shows about a 2.9ms rise time as well.



Simulate neural system test circuit



The standalone scope trace of the input and the output. Square pulse put into the system generates the bottom trace. It seems to have ~2.9ms rise time.



Sample Frequency: 22050

Voltage Scale (V): .050 .10 .20 .50 1.0 (selected: 1.0)

Time Scale (s): .002 .005 .010 .020 .050 (selected: .050)

Trigger: Turn On ☐

Trigger Value: 0.000

Measure

ABS Volt. Diff.: 0.2809224

ABS Time Diff.: 2.894212E-03

The SimpleScope trace of the output from the mock-neural system. (rise time – time between the cursors is 2.9 ms- consistent with the above reading). The signal is AC coupled.

**Hardware**

Input Testing

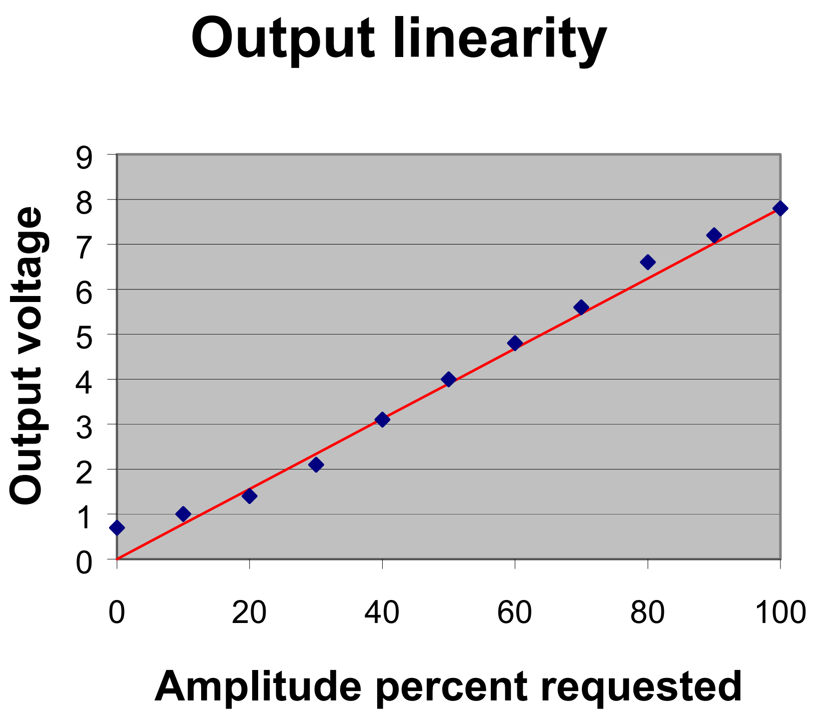
The testing of this was rather straightforward - apply an input and see if the correct amplification appeared on the output with an oscilloscope. A signal generator was used to create the input. To test for noise, the inputs were both grounded and the outputs were scoped, and there was less than 1mV of noise.

The testing of the calibration signal was different. We wanted to make sure that the signal would remain constant as the supply voltage changed, simulating dying batteries. So the output wave of the circuit was examined as the supply voltage was lowered from 5V down to around 3.5V to make sure that the signal stayed constant. The biased diode worked as expected and the calibration remained constant at a peak-to-peak voltage of .08V.

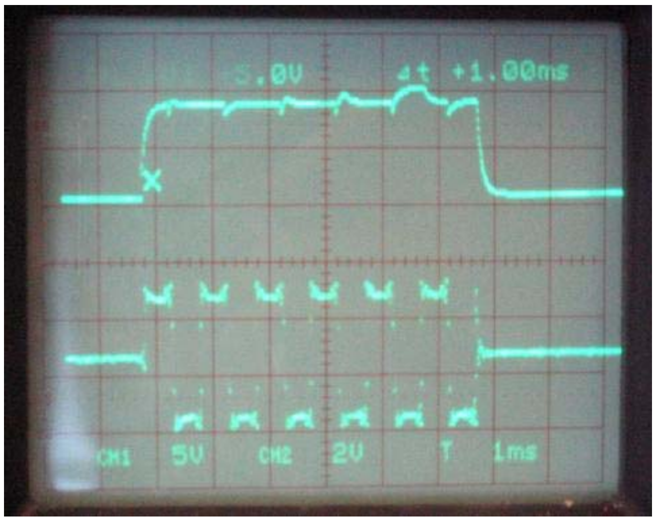
Output Testing

There were two things that we tested in the stimulator circuit: the timing of turning the output on and off, and the linearity of the output amplitude vs. input. The expected times for the rise and fall of the output was somewhere in the 50 microsecond region from the RC time constants of the devices on the circuit. The stimulator when tested had a rise time of about 100 microseconds and a fall time of about 80 microseconds.

The rise time of the output is proportional to the desired amplitude of the output. Since the H11F1 is in the resistive path of charging the output, the higher resistance used to lower output voltage will correspond to a longer time to charge the output capacitance. The falling time constant was confusing because there were no extraneous components other than one capacitor and one resistor to drain the output, together having a time constant of around 50 microseconds. The device closely follows the linear pattern that it should except near the low power region. The output voltage did not quite make the 0V to 10V swing that we expected it to make when we defined our design task. Instead, the output voltage swing ranged from ~.8 V to ~7.8 V. The chart below displays the results.



An output voltage (top trace) created from a synchronization signal (bottom trace).



## VI. Conclusions

As a design project, SimpleScope taught both of us a great deal. Initially, we didn't understand or even have the insight to know some of the tough design challenges awaiting us. We also didn't realize the less-than-theoretically perfect results that we obtained. Many times the noise levels that we encountered on the output and on the input were much worse than we expected, and as a result, we had to solve a new problem and tweak our hardware and software a little bit. This test and redesign process really was very key to establishing a working, usable final result.

Future improvements are certainly plentiful. Some of the features included in the original MATLAB software oscilloscope program were not included in this design but could very well be. These features include saving the entire state of the program, saving the captured data and being able to reload it at a later time, a software zoom, and being able to print the virtual scope signal. For the hardware, a nice package could be made for the boards so as to have a "black box" device. And in conjunction with this idea, the originally intended battery power source could be added. More gain could also be added to the output hardware if that was needed for an application.

**APPENDICES**

## APPENDIX I: HARDWARE SCHEMATICS

Circuit used for amplification and conversion of differential input.

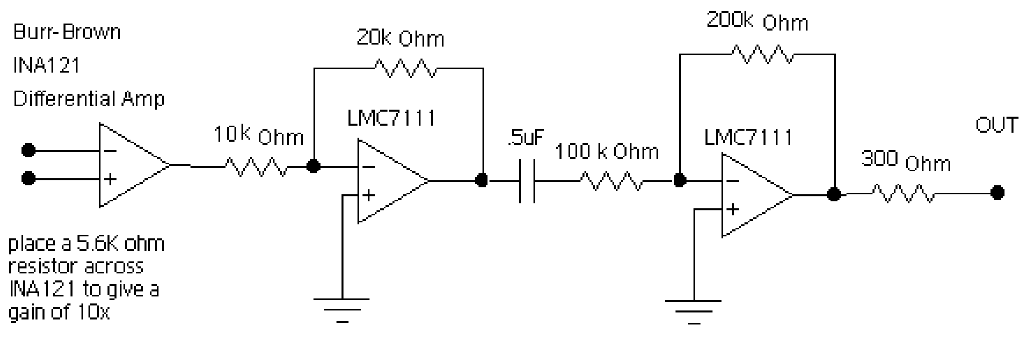


Figure 1. Preamp.

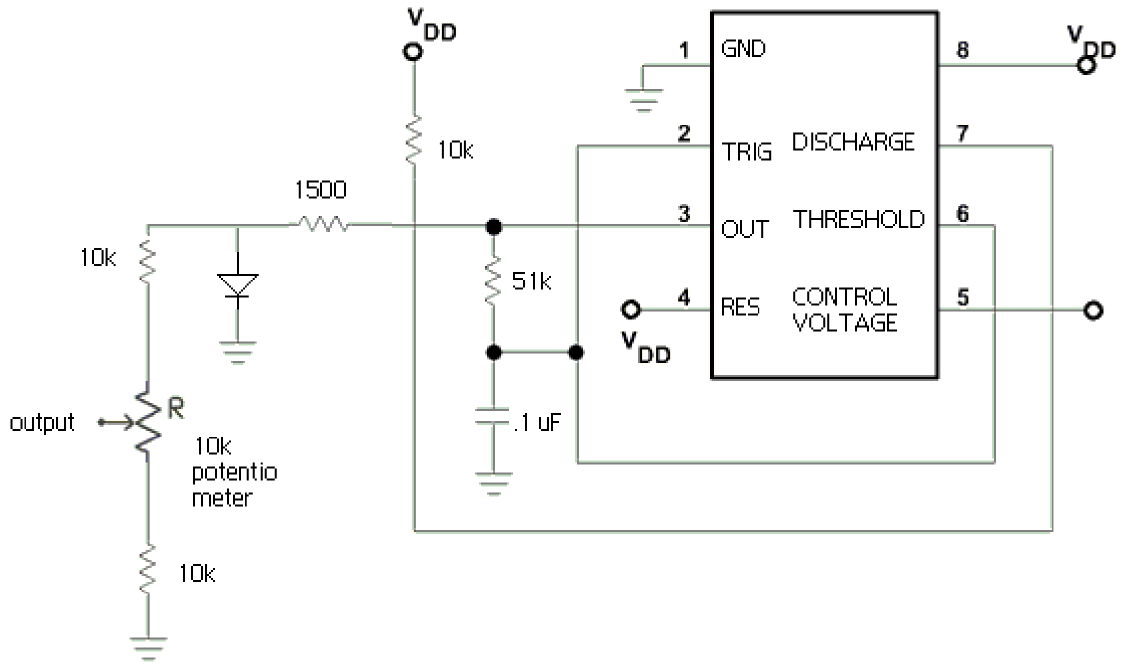


Figure 2. Calibration channel. Circuit behavior is independent of the supply voltage used.

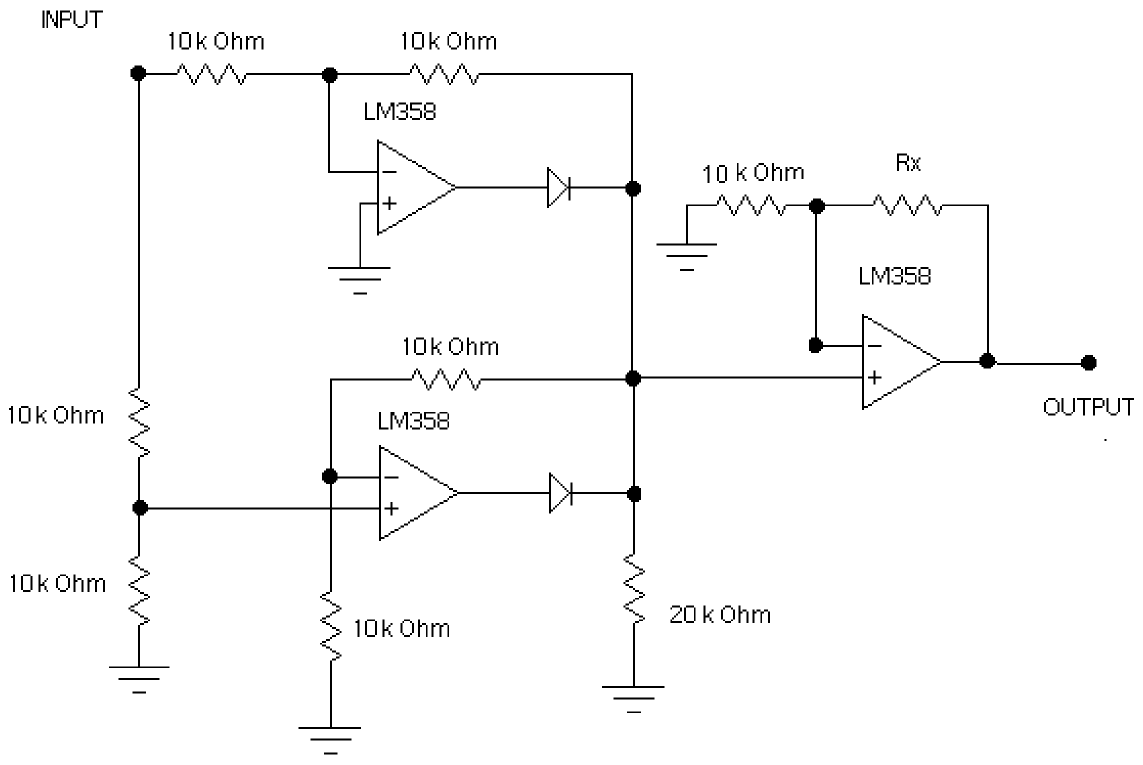


Figure 3. Rectifier diagram

The value of Rx is different for the two channels controlling the output. The synchronization channel needs more gain, so it uses Rx = 33k. The amplitude control channel needs to maximize its use of the channel so only a gain of 2 is used, Rx=10k.

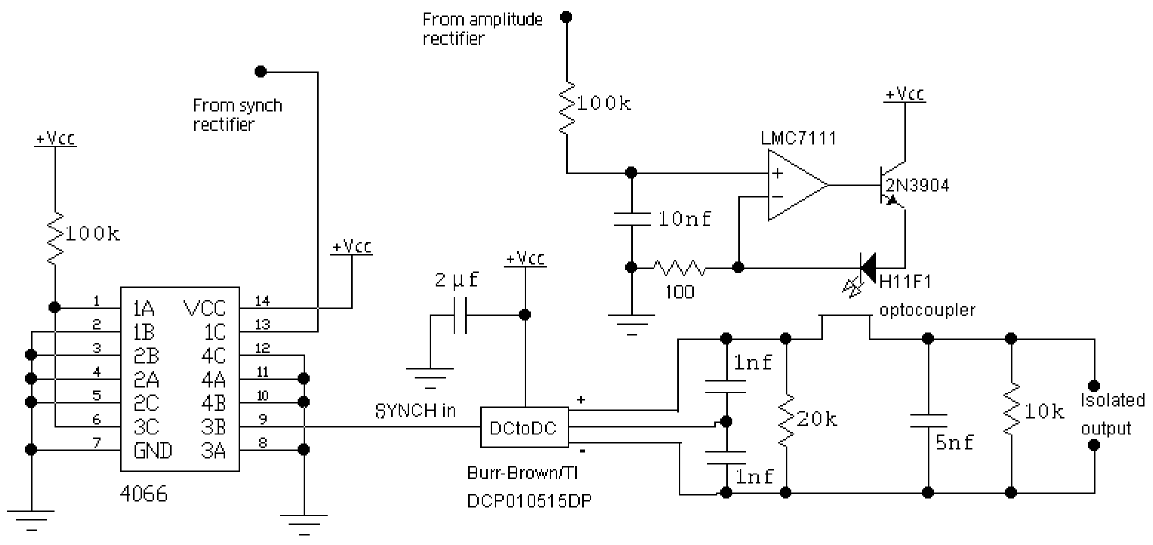


Figure 4. Stimulator diagram

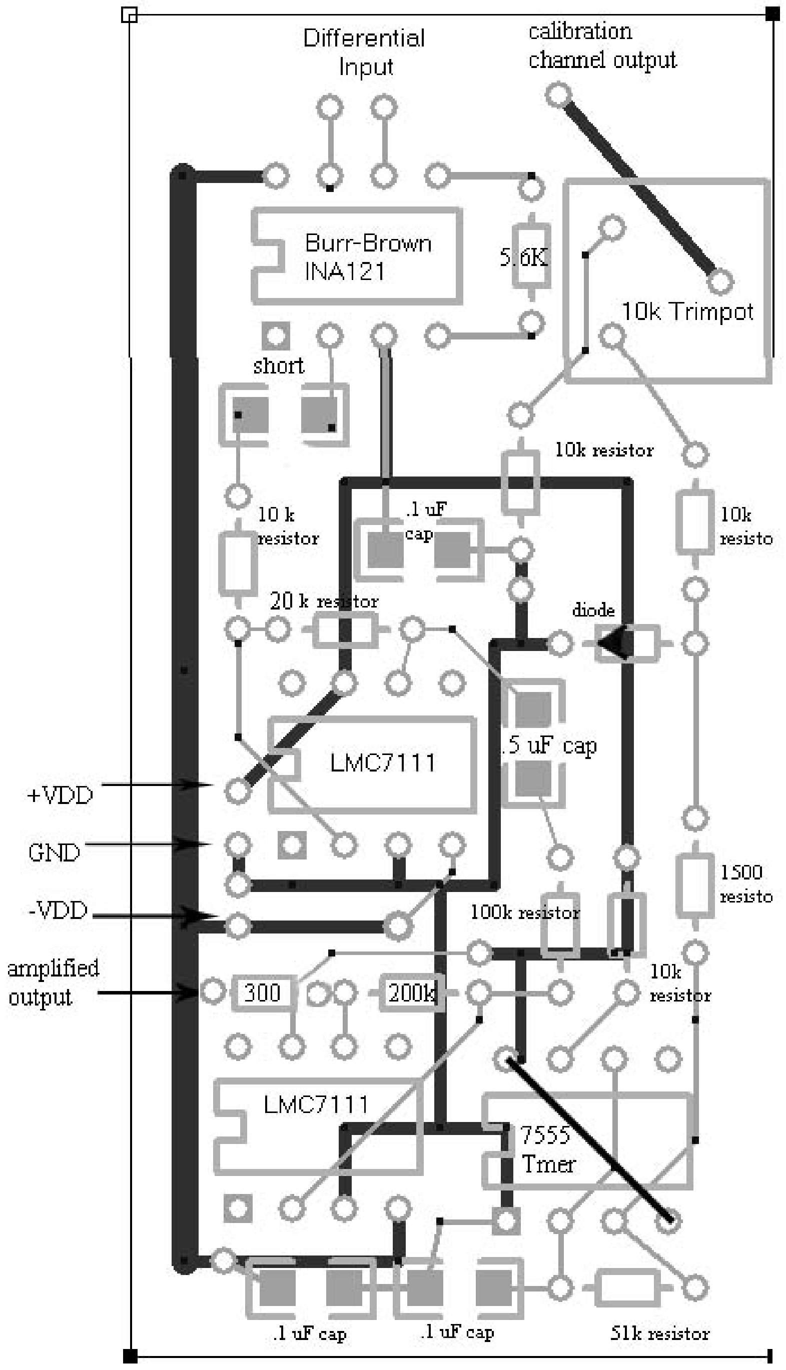


Figure 5. Part placement for input PCB board

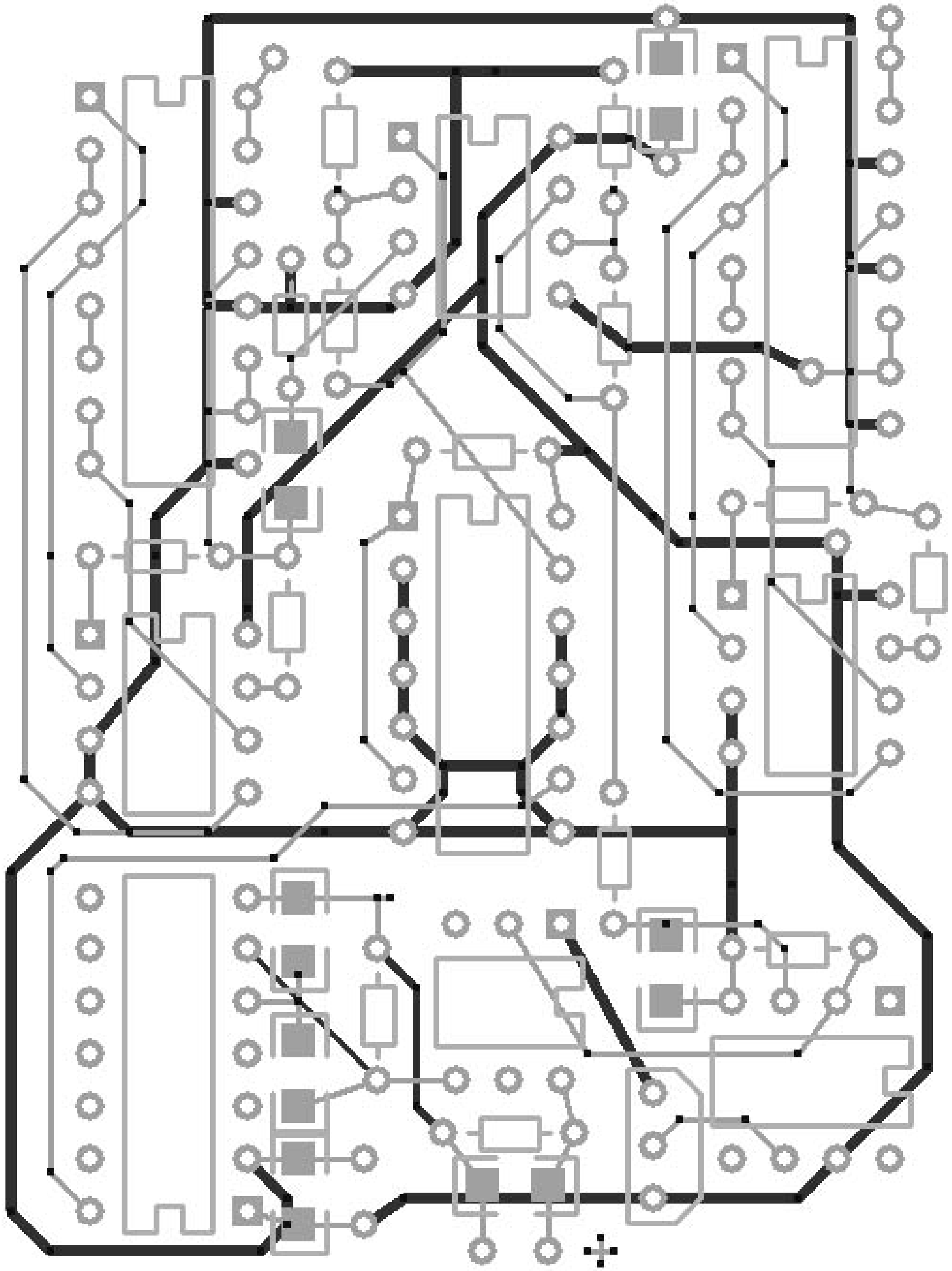


Figure 6a. PCB for stimulator.   The part placement is on next page

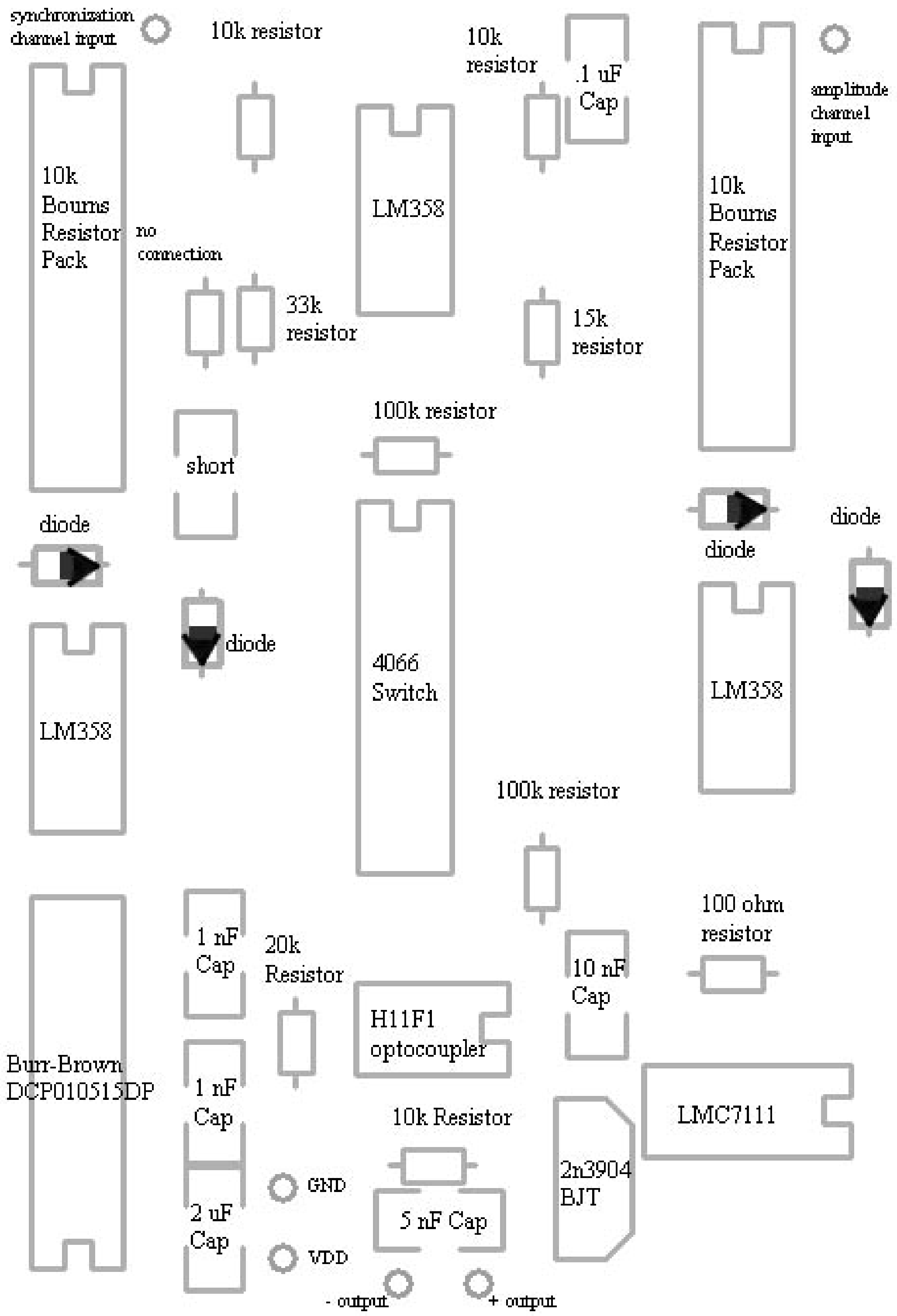
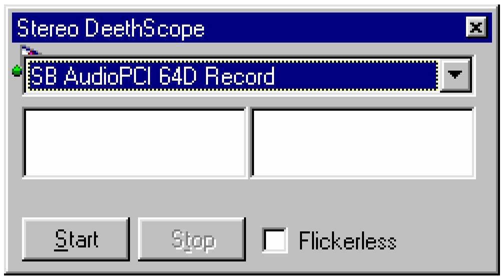


Figure 6b. Part placement for stimulator PCB board.

Note that there is a shorted capacitor connection and an unused resistor spot on the upper left part of the board. This was for a high pass filter that was taken out of the final design because it wasn't necessary.

# APPENDIX II: SOFTWARE REFERENCED/USED

## 1. Deeth Oscilloscope Program by Murphy McCauley



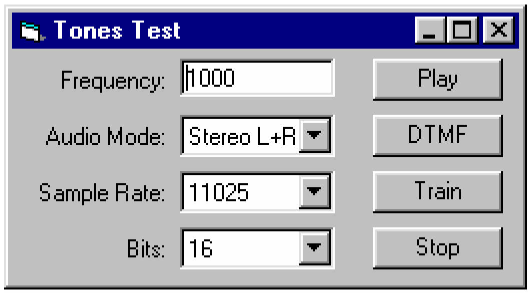
```
'-----------------------------------------------------------------------
' Deeth Stereo Oscilloscope v1.0
' A simple oscilloscope application -- now in <<stereo>>
'-----------------------------------------------------------------------
' Opens a waveform audio device for 8-bit 11kHz input, and plots the
' waveform to a window.  Can only be resized to a certain minimum
' size defined by the Shape box.
'-----------------------------------------------------------------------
' It would be good to make this use the same double-buffering
' scheme as the Spectrum Analyzer.
'-----------------------------------------------------------------------
' Murphy McCauley (MurphyMc@Concentric.NET) 08/12/99
'-----------------------------------------------------------------------

Option Explicit

Private DevHandle As Long
Private InData(0 To 511) As Byte
Private Inited As Boolean
Public MinHeight As Long, MinWidth As Long

Private Type WaveFormatEx
   FormatTag As Integer
   Channels As Integer
   SamplesPerSec As Long
   AvgBytesPerSec As Long
   BlockAlign As Integer
   BitsPerSample As Integer
   ExtraDataSize As Integer
End Type

Private Type WaveHdr
   lpData As Long
   dwBufferLength As Long
   dwBytesRecorded As Long
   dwUser As Long
   dwFlags As Long
   dwLoops As Long
   lpNext As Long 'wavehdr_tag
   Reserved As Long
End Type

Private Type WaveInCaps
   ManufacturerID As Integer      'wMid
   ProductID As Integer      'wPid
   DriverVersion As Long      'MMVERSIONS vDriverVersion
```

```
    ProductName(1 To 32) As Byte 'szPname[MAXPNAMELEN]
    Formats As Long
    Channels As Integer
    Reserved As Integer
End Type

Private Const WAVE_INVALIDFORMAT = &H0&              '/* invalid format */
Private Const WAVE_FORMAT_1M08 = &H1&                '/* 11.025 kHz, Mono,   8-bit
Private Const WAVE_FORMAT_1S08 = &H2&                '/* 11.025 kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_1M16 = &H4&                '/* 11.025 kHz, Mono,   16-bit
Private Const WAVE_FORMAT_1S16 = &H8&                '/* 11.025 kHz, Stereo, 16-bit
Private Const WAVE_FORMAT_2M08 = &H10&               '/* 22.05  kHz, Mono,   8-bit
Private Const WAVE_FORMAT_2S08 = &H20&               '/* 22.05  kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_2M16 = &H40&               '/* 22.05  kHz, Mono,   16-bit
Private Const WAVE_FORMAT_2S16 = &H80&               '/* 22.05  kHz, Stereo, 16-bit
Private Const WAVE_FORMAT_4M08 = &H100&              '/* 44.1   kHz, Mono,   8-bit
Private Const WAVE_FORMAT_4S08 = &H200&              '/* 44.1   kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_4M16 = &H400&              '/* 44.1   kHz, Mono,   16-bit
Private Const WAVE_FORMAT_4S16 = &H800&              '/* 44.1   kHz, Stereo, 16-bit

Private Const WAVE_FORMAT_PCM = 1

Private Const WHDR_DONE = &H1&            '/* done bit */
Private Const WHDR_PREPARED = &H2&        '/* set if this header has been prepared */
Private Const WHDR_BEGINLOOP = &H4&       '/* loop start block */
Private Const WHDR_ENDLOOP = &H8&         '/* loop end block */
Private Const WHDR_INQUEUE = &H10&        '/* reserved for driver */

Private Const WIM_OPEN = &H3BE
Private Const WIM_CLOSE = &H3BF
Private Const WIM_DATA = &H3C0

Private Declare Function waveInAddBuffer Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
Private Declare Function waveInPrepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
Private Declare Function waveInUnprepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer
As Long, ByVal WaveHdrStructSize As Long) As Long

Private Declare Function waveInGetNumDevs Lib "winmm" () As Long
Private Declare Function waveInGetDevCaps Lib "winmm" Alias "waveInGetDevCapsA" (ByVal uDeviceID As Long, ByVal
WaveInCapsPointer As Long, ByVal WaveInCapsStructSize As Long) As Long

Private Declare Function waveInOpen Lib "winmm" (WaveDeviceInputHandle As Long, ByVal WhichDevice As Long, ByVal
WaveFormatExPointer As Long, ByVal CallBack As Long, ByVal CallBackInstance As Long, ByVal Flags As Long) As Long
Private Declare Function waveInClose Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long

Private Declare Function waveInStart Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long
Private Declare Function waveInReset Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long
Private Declare Function waveInStop Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long


Sub InitDevices()
    Dim Caps As WaveInCaps, Which As Long
    DevicesBox.Clear
    For Which = 0 To waveInGetNumDevs - 1
        Call waveInGetDevCaps(Which, VarPtr(Caps), Len(Caps))
        'If Caps.Formats And WAVE_FORMAT_1M08 Then
        If Caps.Formats And WAVE_FORMAT_1S08 Then 'Now is 1S08 -- Check for devices that can do stereo 8-bit 11kHz
            Call DevicesBox.AddItem(StrConv(Caps.ProductName, vbUnicode), Which)
        End If
    Next
    If DevicesBox.ListCount = 0 Then
        MsgBox "You have no audio input devices!", vbCritical, "Ack!"
        End
    End If
    DevicesBox.ListIndex = 0
End Sub


Private Sub Flicker_Click()
    Scope(0).Cls
    Scope(1).Cls
    If Flicker.Value = vbChecked Then
        Scope(0).AutoRedraw = True
        Scope(1).AutoRedraw = True
    Else
        Scope(0).AutoRedraw = False
        Scope(1).AutoRedraw = False
    End If
End Sub


Private Sub Form_Load()
    Call InitDevices

    'Set MinWidth and MinHeight based on Shape...
    Dim XAdjust As Long, YAdjust As Long
    XAdjust = Me.Width \ Screen.TwipsPerPixelX - Me.ScaleWidth
    YAdjust = Me.Height \ Screen.TwipsPerPixelY - Me.ScaleHeight

    MinWidth = Shape.Width + XAdjust
    MinHeight = Shape.Height + YAdjust

    Shape.BackStyle = vbTransparent


    'Set the window proceedure to my own (which restricts the
    'minimum size of the form...
    'Comment out the SetWindowLong line if you're working with it
    'in the development environment since it'll hang in stop mode.
    MinMaxProc.Proc = GetWindowLong(Me.HWnd, GWL_WNDPROC)
    SetWindowLong Me.HWnd, GWL_WNDPROC, AddressOf WindowProc

End Sub


Private Sub Form_Resize()
    Scope(0).Cls
    Scope(1).Cls

    Stuff.Top = Me.ScaleHeight - Stuff.Height - 3
    Scope(0).Height = Me.ScaleHeight - 75
    Scope(1).Height = Scope(0).Height
    Scope(0).Width = (Me.ScaleWidth - 13) \ 2
    Scope(1).Width = Scope(0).Width
    Scope(1).Left = Scope(0).Left + Scope(0).Width + 1

    DevicesBox.Width = Me.ScaleWidth - 13

    Scope(0).ScaleHeight = 256
    Scope(0).ScaleWidth = 255
    Scope(1).ScaleHeight = 256
    Scope(1).ScaleWidth = 255
    'Make the window resize now so that it doesn't interfere with redrawing the data
    DoEvents

    'Redraw the data at the new size
    If Inited = True Then
        Call DrawData
    End If
End Sub


Private Sub Form_Unload(Cancel As Integer)
    If DevHandle <> 0 Then
        Call DoStop
    End If
End Sub


Private Sub StartButton_Click()
    Static WaveFormat As WaveFormatEx
    With WaveFormat
        .FormatTag = WAVE_FORMAT_PCM
        .Channels = 2 'Two channels -- left and right
        .SamplesPerSec = 11025 '11khz
        .BitsPerSample = 8
        .BlockAlign = (.Channels * .BitsPerSample) \ 8
        .AvgBytesPerSec = .BlockAlign * .SamplesPerSec
        .ExtraDataSize = 0
    End With

    Debug.Print "waveInOpen:"; waveInOpen(DevHandle, DevicesBox.ListIndex, VarPtr(WaveFormat), 0, 0, 0)

    If DevHandle = 0 Then
        Call MsgBox("Wave input device didn't open!", vbExclamation, "Ack!")
        Exit Sub
    End If
    Debug.Print " "; DevHandle
    Call waveInStart(DevHandle)

    Inited = True

    StopButton.Enabled = True
    StartButton.Enabled = False

    Call Visualize
End Sub


Private Sub StopButton_Click()
    Call DoStop
End Sub


Private Sub DoStop()
    Call waveInReset(DevHandle)
    Call waveInClose(DevHandle)
    DevHandle = 0
    StopButton.Enabled = False
    StartButton.Enabled = True
End Sub
```

```
Private Sub Visualize()
  Static Wave As WaveHdr

  Wave.lpData = VarPtr(InData(0))
  Wave.dwBufferLength = 512 'This is now 512 so there's still 256 samples per channel
  Wave.dwFlags = 0

  Do

    Call waveInPrepareHeader(DevHandle, VarPtr(Wave), Len(Wave))
    Call waveInAddBuffer(DevHandle, VarPtr(Wave), Len(Wave))

    Do
      'Nothing -- we're waiting for the audio driver to mark
      'this wave chunk as done.
    Loop Until ((Wave.dwFlags And WHDR_DONE) = WHDR_DONE) Or DevHandle = 0

    Call waveInUnprepareHeader(DevHandle, VarPtr(Wave), Len(Wave))

    If DevHandle = 0 Then
      'The device has closed...
      Exit Do
    End If

    Scope(0).Cls
    Scope(1).Cls

    Call DrawData

    DoEvents
  Loop While DevHandle <> 0 'While the audio device is open

End Sub


Private Sub DrawData()
  Static X As Long

  Scope(0).CurrentX = -1
  Scope(0).CurrentY = Scope(0).ScaleHeight \ 2
  Scope(1).CurrentX = -1
  Scope(1).CurrentY = Scope(0).ScaleHeight \ 2

  'Plot the data...
  For X = 0 To 255
    Scope(0).Line Step(0, 0)-(X, InData(X * 2))
    Scope(1).Line Step(0, 0)-(X, InData(X * 2 + 1)) 'For a good soundcard...

    'Use these to plot dots instead of lines...
    'Scope(0).PSet (X, InData(X * 2))
    'Scope(1).PSet (X, InData(X * 2 + 1)) 'For a good soundcard...

    'My soundcard is pretty cheap... the right is
    'noticably less loud than the left... so I add five to it.
    'Scope(1).Line Step(0, 0)-(X, InData(X * 2 + 1) + 5)
  Next

  Scope(0).CurrentY = Scope(0).Width
  Scope(1).CurrentY = Scope(0).Width
End Sub
*******************************************************************************************
'-----------------------------------------------------------------------
```

```
' This is a dopey window proceedure that restricts the size of a
' window.
'---------------------------------------------------------------------
' Murphy McCauley (MurphyMc@Concentric.NET) 08/06/99
'---------------------------------------------------------------------

Option Explicit

Declare Function GetWindowLong Lib "user32" Alias "GetWindowLongA" (ByVal HWnd As Long, ByVal nIndex As Long)
As Long
Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" (ByVal HWnd As Long, ByVal nIndex As Long,
ByVal dwNewLong As Long) As Long
Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" (ByVal lpPrevWndFunc As Long, ByVal HWnd As
Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long

Public Proc As Long
Public Const GWL_WNDPROC = (-4)

Private Const WM_GETMINMAXINFO = &H24

Declare Sub MoveMemory Lib "kernel32" Alias "RtlMoveMemory" ( _
   ByRef Destination As Any, _
   ByRef Source As Any, _
   ByVal ByteLength As Long)

Private Type POINTAPI
      X As Long
      Y As Long
End Type

Private Type MINMAXINFO
      ptReserved As POINTAPI
      ptMaxSize As POINTAPI
      ptMaxPosition As POINTAPI
      ptMinTrackSize As POINTAPI
      ptMaxTrackSize As POINTAPI
End Type


Function WindowProc(ByVal HWnd As Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
   Static MinMax As MINMAXINFO

   Select Case Msg
      Case WM_GETMINMAXINFO
         Call MoveMemory(MinMax, ByVal lParam, Len(MinMax))
         MinMax.ptMinTrackSize.X = Base.MinWidth
         MinMax.ptMinTrackSize.Y = Base.MinHeight
         Call MoveMemory(ByVal lParam, MinMax, Len(MinMax))
      Case Else
         WindowProc = CallWindowProc(Proc, HWnd, Msg, wParam, lParam)
   End Select
End Function
```

## 2. Tones by David M.Hitchner



Frequency: 1000
Audio Mode: Stereo L+R
Sample Rate: 11025
Bits: 16

```
Option Explicit

Dim bytSound() As Byte
Private Type DTMF
   bytTones() As Byte
End Type

'------------------------------------------------------------------------------------
' Create and Play all 16 DTMF Tones
'------------------------------------------------------------------------------------
Private Sub cmdDTMF_Click()
   Dim i As Integer
   Dim intAudioMode As Integer
   Dim lngSampleRate As Long
   Dim intBits As Integer
   Dim intIndex As Integer
   Dim udtSineWaves(1 To 2) As SINEWAVE

   Dim udtDTMF(16) As DTMF

   intAudioMode = cboAudioMode.ItemData(cboAudioMode.ListIndex)
   lngSampleRate = cboSampleRate.ItemData(cboSampleRate.ListIndex)
   intBits = cboBits.ItemData(cboBits.ListIndex)

   Wav_Stop

   '----------------------------------------------------
   '            DTMF Tones
   ' Freq  1209  1336  1477  1633
   '  697    1     2     3     A
   '  770    4     5     6     B
   '  852    7     8     9     C
   '  941    *     0     #     D
   '----------------------------------------------------
   For i = 1 To 16
      With udtSineWaves(1)
         .dblAmplitudeL = 0.25
         .dblAmplitudeR = 0.25
      End With

      With udtSineWaves(2)
         .dblAmplitudeL = 0.25
         .dblAmplitudeR = 0.25
      End With

      udtSineWaves(1).dblFrequency = Choose(i, _
         697, 697, 697, 697, _
         770, 770, 770, 770, _
```

```
            852, 852, 852, 852, _
            941, 941, 941, 941)

        udtSineWaves(2).dblFrequency = Choose(i, _
            1209, 1336, 1477, 1633, _
            1209, 1336, 1477, 1633, _
            1209, 1336, 1477, 1633, _
            1209, 1336, 1477, 1633)

        Wav_BuildHeader udtDTMF(i).bytTones, lngSampleRate, intBits, _
            intAudioMode, 0.5, 0.5
        Wav_MultiSineWave udtDTMF(i).bytTones, udtSineWaves, 0.25
    Next

    For i = 1 To 16
        Wav_Play udtDTMF(i).bytTones
    Next

End Sub

'-------------------------------------------------------------------------------
' Create and Play Wave based on current settings.
'-------------------------------------------------------------------------------
Private Sub cmdPlay_Click()
    Dim dblFrequency As Double
    Dim intAudioMode As Integer
    Dim lngSampleRate As Long
    Dim intBits As Integer

    dblFrequency = Val(txtFrequency)
    intAudioMode = cboAudioMode.ItemData(cboAudioMode.ListIndex)
    lngSampleRate = cboSampleRate.ItemData(cboSampleRate.ListIndex)
    intBits = cboBits.ItemData(cboBits.ListIndex)

    Wav_Stop
    Wav_BuildHeader bytSound, lngSampleRate, intBits, intAudioMode, 0.5, 0.5
    Wav_SineWave bytSound, dblFrequency

    ' Wav_WriteToFile "Test.wav", bytSound

    Wav_PlayLoop bytSound
End Sub

'-------------------------------------------------------------------------------
' Stop current wave file.
'-------------------------------------------------------------------------------
Private Sub cmdStop_Click()
    Wav_Stop
End Sub


'-------------------------------------------------------------------------------
' Create and Play Wave using frequencies of a train horn.
'-------------------------------------------------------------------------------
Private Sub cmdTrain_Click()
    Dim i As Integer
    Dim intAudioMode As Integer
    Dim lngSampleRate As Long
    Dim intBits As Integer
    Dim intIndex As Integer
    Dim udtSineWaves(1 To 3) As SINEWAVE
    Dim bytSound() As Byte
    intAudioMode = cboAudioMode.ItemData(cboAudioMode.ListIndex)
    lngSampleRate = cboSampleRate.ItemData(cboSampleRate.ListIndex)
    intBits = cboBits.ItemData(cboBits.ListIndex)

    Wav_Stop

    With udtSineWaves(1)
        .dblFrequency = 255
        .dblAmplitudeL = 0.25
        .dblAmplitudeR = 0.25
    End With

    With udtSineWaves(2)
        .dblFrequency = 311
        .dblAmplitudeL = 0.25
        .dblAmplitudeR = 0.25
    End With

    With udtSineWaves(3)
        .dblFrequency = 440
        .dblAmplitudeL = 0.25
        .dblAmplitudeR = 0.25
    End With

    Wav_BuildHeader bytSound, lngSampleRate, intBits, _
        intAudioMode, 0.5, 0.5
    Wav_MultiSineWave bytSound, udtSineWaves, 3

    Wav_Play bytSound

End Sub

Private Sub Form_Load()
    Dim i As Integer
    Dim intBitSize As Integer
    Dim lngSampleRate As Long

    txtFrequency = "1000"

    With cboAudioMode
        For i = 1 To 4
            .AddItem Choose(i, "Mono", "Stereo L+R", "Stereo L", _
                "Stereo R", "Stereo L-R")
            .ItemData(.NewIndex) = i - 1
        Next
        .ListIndex = 1
    End With

    With cboSampleRate
        For i = 1 To 8
            lngSampleRate = Choose(i, RATE_8000, RATE_11025, RATE_22050, _
                RATE_32000, RATE_44100, RATE_48000, RATE_88000, RATE_96000)
            .AddItem CStr(lngSampleRate)
            .ItemData(.NewIndex) = lngSampleRate
        Next
        .ListIndex = 1
    End With

    With cboBits
        For i = 1 To 2
            intBitSize = Choose(i, BITS_8, BITS_16)
            .AddItem CStr(intBitSize)
            .ItemData(.NewIndex) = intBitSize
        Next
        .ListIndex = 1
    End With

End Sub

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

'-------------------------------------------------------------------------------
' modWAV - WAV File Routines
'
' Written By: David M.Hitchner
'           k5dmh@bellsouth.net
'           k5dmh@arrl.net
'
' This VB module is a collection of routines to uses to create a play WAV
' format files.
'-------------------------------------------------------------------------------
Option Explicit

'-------------------------------------------------------------------------------
' Wave File Format
'-------------------------------------------------------------------------------
' RIFF Chunk   ( 12 bytes)
' 00 00 - 03  "RIFF"
' 04 04 - 07  Total Length to Follow  (Length of File - 8)
' 08 08 - 11  "WAVE"
'
' FORMAT Chunk ( 24 bytes )
' 0C 12 - 15  "fmt_"
' 10 16 - 19  Length of FORMAT Chunk  Always 0x10
' 14 20 - 21  Audio Format            Always 0x01
' 16 22 - 23  Channels                1 = Mono, 2 = Stereo
' 18 24 - 27  Sample Rate             In Hertz
' 1C 28 - 31  Bytes per Second        Sample Rate * Channels * Bits per Sample / 8
' 20 32 - 33  Bytes per Sample        Channels * Bits per Sample / 8
'                                 1 = 8 bit Mono
'                                 2 = 8 bit Stereo or 16 bit Mono
'                                 4 = 16 bit Stereo
' 22 34 - 35  Bits per Sample
'
' DATA Chunk
' 24 36 - 39  "data"
' 28 40 - 43  Length of Data          Samples * Channels * Bits per Sample / 8
' 2C 44 - End Data Samples
'            8 Bit = 0 to 255            unsigned bytes
'           16 Bit = -32,768 to 32,767    2's-complement signed integers
'-------------------------------------------------------------------------------

Public Const MODE_MONO = 0      ' Mono
Public Const MODE_LR = 1        ' Stereo L+R
Public Const MODE_L = 2         ' Stereo L
Public Const MODE_R = 3         ' Stereo R

Public Const RATE_8000 = 8000
Public Const RATE_11025 = 11025
Public Const RATE_22050 = 22050
Public Const RATE_32000 = 32000
Public Const RATE_44100 = 44100
Public Const RATE_48000 = 48000
Public Const RATE_88000 = 88000
Public Const RATE_96000 = 96000

Public Const BITS_8 = 8
Public Const BITS_16 = 16

Public Type SINEWAVE
   dblFrequency As Double
   dblDataSlice As Double
   dblAmplitudeL As Double
   dblAmplitudeR As Double
End Type

Private PI As Double
Private intBits As Integer
Private lngSampleRate As Long
Private intSampleBytes As Integer
Private intAudioMode As Integer
Private dblFrequency As Double
Private dblVolumeL As Double
Private dblVolumeR As Double
Private intAudioWidth As Integer

Private Const SND_ALIAS = &H10000
Private Const SND_ALIAS_ID = &H110000
Private Const SND_ALIAS_START = 0
Private Const SND_APPLICATION = &H80
Private Const SND_ASYNC = &H1
Private Const SND_FILENAME = &H20000
Private Const SND_LOOP = &H8
Private Const SND_MEMORY = &H4
Private Const SND_NODEFAULT = &H2
Private Const SND_NOSTOP = &H10
Private Const SND_NOWAIT = &H2000
Private Const SND_PURGE = &H40
Private Const SND_RESERVED = &HFF000000
Private Const SND_RESOURCE = &H40004
Private Const SND_SYNC = &H0
Private Const SND_TYPE_MASK = &H170007
Private Const SND_VALID = &H1F
Private Const SND_VALIDFLAGS = &H17201F

Private Declare Function PlaySoundFile Lib "winmm.dll" Alias "PlaySoundA" _
   (ByVal lpszName As String, ByVal hModule As Long, ByVal dwFlags As Long) As Long
Private Declare Function PlaySoundMemory Lib "winmm.dll" Alias "PlaySoundA" _
   (ptrMemory As Any, ByVal hModule As Long, ByVal dwFlags As Long) As Long
'-------------------------------------------------------------------------------------
' Wav_Play - Continuously plays the wav file from memory.
'-------------------------------------------------------------------------------------
Public Function Wav_PlayLoop(WavArray() As Byte) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundMemory(WavArray(0), ByVal 0&, SND_MEMORY Or SND_APPLICATION Or _
      SND_ASYNC Or SND_LOOP Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_PlayLoop = False
   Else
      Wav_PlayLoop = True
   End If
End Function
```

```
'--------------------------------------------------------------------------------
' Wav_PlayFileLoop - Continuously plays the wav file from a file.
'--------------------------------------------------------------------------------
Public Function Wav_PlayFileLoop(FileName As String) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundFile(FileName, 0, SND_FILENAME Or SND_APPLICATION Or _
      SND_ASYNC Or SND_LOOP Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_PlayFileLoop = False
   Else
      Wav_PlayFileLoop = True
   End If
End Function

'--------------------------------------------------------------------------------
' Wav_Play - Plays the wav file from memory.
'--------------------------------------------------------------------------------
Public Function Wav_Play(WavArray() As Byte) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundMemory(WavArray(0), 0, SND_MEMORY Or SND_APPLICATION Or _
      SND_SYNC Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_Play = False
   Else
      Wav_Play = True
   End If
End Function

'--------------------------------------------------------------------------------
' Wav_Play - Plays the wav file from a file.
'--------------------------------------------------------------------------------
Public Function Wav_PlayFile(FileName As String) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundFile(FileName, 0, SND_FILENAME Or SND_APPLICATION Or _
      SND_SYNC Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_PlayFile = False
   Else
      Wav_PlayFile = True
   End If
End Function
'--------------------------------------------------------------------------------
' Wav_BuildHeader - Builds the WAV file header based on the sample rate, resolution,
'               audio mode.  Also sets the volume level for other routines.
'--------------------------------------------------------------------------------
Public Sub Wav_BuildHeader(WavArray() As Byte, SampleRate As Long, _
   Resolution As Integer, AudioMode As Integer, VolumeL As Double, VolumeR As Double)
   Dim lngBytesASec As Long

   PI = 4# * Atn(1#)

   ' Save parameters.
   lngSampleRate = SampleRate
   intBits = Resolution
   intAudioMode = AudioMode
   dblVolumeL = VolumeL
```

```
dblVolumeR = VolumeR

ReDim WavArray(0 To 43)

'-----------------------------------------------------------------------------
' Fixed Data
'-----------------------------------------------------------------------------
WavArray(0) = 82   ' R
WavArray(1) = 73   ' I
WavArray(2) = 70   ' F
WavArray(3) = 70   ' F
WavArray(8) = 87   ' W
WavArray(9) = 65   ' A
WavArray(10) = 86  ' V
WavArray(11) = 69  ' E
WavArray(12) = 102 ' f
WavArray(13) = 109 ' m
WavArray(14) = 116 ' t
WavArray(15) = 32  ' .
WavArray(16) = 16  ' Length of Format Chunk
WavArray(17) = 0   ' Length of Format Chunk
WavArray(18) = 0   ' Length of Format Chunk
WavArray(19) = 0   ' Length of Format Chunk
WavArray(20) = 1   ' Audio Format
WavArray(21) = 0   ' Audio Format
WavArray(36) = 100 ' d
WavArray(37) = 97  ' a
WavArray(38) = 116 ' t
WavArray(39) = 97  ' a

'-----------------------------------------------------------------------------
' Bytes 22 - 23  Channels   1 = Mono, 2 = Stereo
'-----------------------------------------------------------------------------
Select Case intAudioMode
    Case MODE_MONO:
        WavArray(22) = 1
        WavArray(23) = 0
        intAudioWidth = 1
    Case MODE_LR:
        WavArray(22) = 2
        WavArray(23) = 0
        intAudioWidth = 2
    Case MODE_L:
        WavArray(22) = 2
        WavArray(23) = 0
        intAudioWidth = 2
    Case MODE_R:
        WavArray(22) = 2
        WavArray(23) = 0
        intAudioWidth = 2
End Select

'-----------------------------------------------------------------------------
' 24 - 27  Sample Rate            In Hertz
'-----------------------------------------------------------------------------
WavArray(24) = ExtractByte(lngSampleRate, 0)
WavArray(25) = ExtractByte(lngSampleRate, 1)
WavArray(26) = ExtractByte(lngSampleRate, 2)
WavArray(27) = ExtractByte(lngSampleRate, 3)

'-----------------------------------------------------------------------------
' Bytes 34 - 35  Bits per Sample
```

```
    '-------------------------------------------------------------------------------
    Select Case intBits
        Case 8:
            WavArray(34) = 8
            WavArray(35) = 0
            intSampleBytes = 1
        Case 16:
            WavArray(34) = 16
            WavArray(35) = 0
            intSampleBytes = 2
    End Select

    '-------------------------------------------------------------------------------
    ' Bytes 28 - 31  Bytes per Second   Sample Rate * Channels * Bits per Sample / 8
    '-------------------------------------------------------------------------------
    lngBytesASec = lngSampleRate * intAudioWidth * intSampleBytes

    WavArray(28) = ExtractByte(lngBytesASec, 0)
    WavArray(29) = ExtractByte(lngBytesASec, 1)
    WavArray(30) = ExtractByte(lngBytesASec, 2)
    WavArray(31) = ExtractByte(lngBytesASec, 3)

    '-------------------------------------------------------------------------------
    ' Bytes 32 - 33 Bytes per Sample    Channels * Bits per Sample / 8
    '                               1 = 8 bit Mono
    '                               2 = 8 bit Stereo or 16 bit Mono
    '                               4 = 16 bit Stereo
    '-------------------------------------------------------------------------------
    If (intAudioMode = MODE_MONO) And (intBits = 8) Then
        WavArray(32) = 1
        WavArray(33) = 0
    End If

    If ((intAudioMode = MODE_LR) Or (intAudioMode = MODE_L) Or _
        (intAudioMode = MODE_R)) And (intBits = 8) Then
        WavArray(32) = 2
        WavArray(33) = 0
    End If

    If (intAudioMode = MODE_MONO) And (intBits = 16) Then
        WavArray(32) = 2
        WavArray(33) = 0
    End If

    If ((intAudioMode = MODE_LR) Or (intAudioMode = MODE_L) Or _
        (intAudioMode = MODE_R)) And (intBits = 16) Then
        WavArray(32) = 4
        WavArray(33) = 0
    End If

End Sub

'-------------------------------------------------------------------------------
' Wav_WriteToFile - Writes the wav file to disk.
'-------------------------------------------------------------------------------
Public Function Wav_WriteToFile(FileName As String, WavArray() As Byte) As Long
    Static blnInitialized As Boolean

    Dim fd As Integer
    Dim i As Long, lngFileSize As Long

    Wav_WriteToFile = 0
    On Error GoTo Routine_Error

    fd = FreeFile
    Open FileName For Binary As #fd

    lngFileSize = UBound(WavArray)
    For i = 0 To lngFileSize
        Put #fd, , WavArray(i)
    Next

Routine_Exit:
    On Error Resume Next
    Close #fd
    Exit Function

Routine_Error:
    Wav_WriteToFile = Err.Number
    Resume Routine_Exit
End Function

'--------------------------------------------------------------------------------
' Wav_SineWave - Builds a sine wave that may be played in a continuous loop.
'--------------------------------------------------------------------------------
Public Sub Wav_SineWave(WavArray() As Byte, Frequency As Double)
    Dim i As Long
    Dim lngLimit As Long
    Dim lngDataL As Long, lngDataR As Long
    Dim dblDataPt As Double, blnPositive As Boolean
    Dim intCycles As Integer, intCycleCount As Integer
    Dim lngFileSize As Long
    Dim lngSamples As Long
    Dim lngDataSize As Long

    Dim dblDataSlice As Double
    Dim dblWaveTime As Double
    Dim dblTotalTime As Double
    Dim dblSampleTime As Double

    dblFrequency = Frequency

    If dblFrequency > 1000 Then
        intCycles = 100
    Else
        intCycles = 10
    End If

    dblWaveTime = 1 / dblFrequency
    dblTotalTime = dblWaveTime * intCycles
    dblSampleTime = 1 / CDbl(lngSampleRate)
    dblDataSlice = (2 * PI) / (dblWaveTime / dblSampleTime)

    lngSamples = 0
    intCycleCount = 0
    blnPositive = True
    Do
        dblDataPt = Sin(lngSamples * dblDataSlice)
        If lngSamples > 0 Then
            If dblDataPt < 0 Then
                blnPositive = False
            Else
                ' Detect Zero Crossing
                If Not blnPositive Then
                intCycleCount = intCycleCount + 1
                If intCycleCount >= intCycles Then Exit Do
                blnPositive = True
                End If
            End If
        End If
        lngSamples = lngSamples + 1
    Loop

    '-------------------------------------------------------------------------------
    ' Bytes 40 - 43  Length of Data   Samples * Channels * Bits per Sample / 8
    '-------------------------------------------------------------------------------
    lngDataSize = lngSamples * intAudioWidth * (intBits / 8)
    ReDim Preserve WavArray(0 To 43 + lngDataSize)

    WavArray(40) = ExtractByte(lngDataSize, 0)
    WavArray(41) = ExtractByte(lngDataSize, 1)
    WavArray(42) = ExtractByte(lngDataSize, 2)
    WavArray(43) = ExtractByte(lngDataSize, 3)

    '-------------------------------------------------------------------------------
    ' Bytes 04 - 07  Total Length to Follow  (Length of File - 8)
    '-------------------------------------------------------------------------------
    lngFileSize = lngDataSize + 36

    WavArray(4) = ExtractByte(lngFileSize, 0)
    WavArray(5) = ExtractByte(lngFileSize, 1)
    WavArray(6) = ExtractByte(lngFileSize, 2)
    WavArray(7) = ExtractByte(lngFileSize, 3)

    '-------------------------------------------------------------------------------
    ' Bytes 44 - End   Data Samples
    '-------------------------------------------------------------------------------

    If intBits = 8 Then
        lngLimit = 127
    Else
        lngLimit = 32767
    End If

    For i = 0 To lngSamples - 1

        If intBits = 8 Then
            '-------------------------------------------------------------------------
            ' 8 Bit Data
            '-------------------------------------------------------------------------
            ' Calculate data point.
            dblDataPt = Sin(i * dblDataSlice) * lngLimit
            lngDataL = Int(dblDataPt * dblVolumeL) + lngLimit
            lngDataR = Int(dblDataPt * dblVolumeR) + lngLimit

            ' Place data point in wave tile.
            If intAudioMode = MODE_MONO Then _
                WavArray(i + 44) = ExtractByte(lngDataL, 0)

            If intAudioMode = MODE_LR Then        'L+R stereo
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
            End If

            If intAudioMode = MODE_L Then        ' L only stereo
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = 0
            End If

            If intAudioMode = MODE_R Then       ' R only stereo
                WavArray((2 * i) + 44) = 0
                WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
            End If

        Else

            '-----------------------------------------------------------------------
            ' 16 Bit Data
            '-----------------------------------------------------------------------
            ' Calculate data point.
            dblDataPt = Sin(i * dblDataSlice) * lngLimit
            lngDataL = Int(dblDataPt * dblVolumeL)
            lngDataR = Int(dblDataPt * dblVolumeR)

            ' Place data point in wave tile.
            If intAudioMode = MODE_MONO Then
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = ExtractByte(lngDataL, 1)
            End If

            If intAudioMode = MODE_LR Then
                WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
                WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
                WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
            End If

            If intAudioMode = MODE_L Then
                WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
                WavArray((4 * i) + 46) = 0
                WavArray((4 * i) + 47) = 0
            End If

            If intAudioMode = MODE_R Then
                WavArray((4 * i) + 44) = 0
                WavArray((4 * i) + 45) = 0
                WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
                WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
            End If

        End If

      Next
    End Sub

    '-------------------------------------------------------------------------------
    ' Wav_MultiSineWave - Builds a complex wave form from one or more sine waves.
    '-------------------------------------------------------------------------------
    Public Sub Wav_MultiSineWave(WavArray() As Byte, SineWaves() As SINEWAVE, _
      Seconds As Double)

      Dim i As Long, j As Long
      Dim lngLimit As Long
      Dim lngDataL As Long, lngDataR As Long
      Dim dblDataPtL As Double, dblDataPtR As Double
      Dim dblWaveTime As Double
      Dim dblSampleTime As Double
    Dim lngSamples As Long
    Dim lngFileSize As Long, lngDataSize As Long

    Dim intSineCount As Integer

    intSineCount = UBound(SineWaves)

    For i = 1 To intSineCount
        dblWaveTime = 1 / SineWaves(i).dblFrequency
        dblSampleTime = 1 / CDbl(lngSampleRate)
        SineWaves(i).dblDataSlice = (2 * PI) / (dblWaveTime / dblSampleTime)
    Next

    lngSamples = CLng(Seconds / dblSampleTime)

    '------------------------------------------------------------------------------
    ' Bytes 40 - 43  Length of Data   Samples * Channels * Bits per Sample / 8
    '------------------------------------------------------------------------------
    lngDataSize = lngSamples * intAudioWidth * (intBits / 8)
    ReDim Preserve WavArray(0 To 43 + lngDataSize)

    WavArray(40) = ExtractByte(lngDataSize, 0)
    WavArray(41) = ExtractByte(lngDataSize, 1)
    WavArray(42) = ExtractByte(lngDataSize, 2)
    WavArray(43) = ExtractByte(lngDataSize, 3)

    '------------------------------------------------------------------------------
    ' Bytes 04 - 07  Total Length to Follow  (Length of File - 8)
    '------------------------------------------------------------------------------
    lngFileSize = lngDataSize + 36

    WavArray(4) = ExtractByte(lngFileSize, 0)
    WavArray(5) = ExtractByte(lngFileSize, 1)
    WavArray(6) = ExtractByte(lngFileSize, 2)
    WavArray(7) = ExtractByte(lngFileSize, 3)

    '------------------------------------------------------------------------------
    ' Bytes 44 - End   Data Samples
    '------------------------------------------------------------------------------

    If intBits = 8 Then
        lngLimit = 127
    Else
        lngLimit = 32767
    End If

    For i = 0 To lngSamples - 1

        If intBits = 8 Then
            '-----------------------------------------------------------------------
            ' 8 Bit Data
            '-----------------------------------------------------------------------
            dblDataPtL = 0
            dblDataPtR = 0
            For j = 1 To intSineCount
                dblDataPtL = dblDataPtL + (Sin(i * SineWaves(j).dblDataSlice) * _
                    SineWaves(j).dblAmplitudeL)
                dblDataPtR = dblDataPtR + (Sin(i * SineWaves(j).dblDataSlice) * _
                    SineWaves(j).dblAmplitudeR)
            Next

            lngDataL = Int(dblDataPtL * dblVolumeL * lngLimit) + lngLimit
            lngDataR = Int(dblDataPtL * dblVolumeR * lngLimit) + lngLimit

            If intAudioMode = MODE_MONO Then _
                WavArray(i + 44) = ExtractByte(lngDataL, 0)

            If intAudioMode = MODE_LR Then       'L+R stereo
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
            End If

            If intAudioMode = MODE_L Then       ' L only stereo
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = 0
            End If

            If intAudioMode = MODE_R Then       ' R only stereo
                WavArray((2 * i) + 44) = 0
                WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
            End If

        Else

            '-----------------------------------------------------------------------
            ' 16 Bit Data
            '-----------------------------------------------------------------------
            dblDataPtL = 0
            dblDataPtR = 0
            For j = 1 To intSineCount
                dblDataPtL = dblDataPtL + (Sin(i * SineWaves(j).dblDataSlice) * _
                    SineWaves(j).dblAmplitudeL)
                dblDataPtR = dblDataPtR + (Sin(i * SineWaves(j).dblDataSlice) * _
                    SineWaves(j).dblAmplitudeR)
            Next

            lngDataL = Int(dblDataPtL * dblVolumeL * lngLimit)
            lngDataR = Int(dblDataPtL * dblVolumeR * lngLimit)

            If intAudioMode = MODE_MONO Then
                WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((2 * i) + 45) = ExtractByte(lngDataL, 1)
            End If

            If intAudioMode = MODE_LR Then
                WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
                WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
                WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
            End If

            If intAudioMode = MODE_L Then
                WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
                WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
                WavArray((4 * i) + 46) = 0
                WavArray((4 * i) + 47) = 0
            End If

            If intAudioMode = MODE_R Then
                WavArray((4 * i) + 44) = 0
                WavArray((4 * i) + 45) = 0
                WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
                WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
            End If
        End If

    Next
End Sub
'------------------------------------------------------------------------------------
' ExtractByte - Extracts the high or low byte from a short (16 bit) VB integer.
'
'   intWord     - VB Integer from which to extract byte.
'   intByte     - Returned high or low byte.
'   intPosition - |                   Word                  |
'               | Byte = 3 | Byte = 2 | Byte = 1 | Byte = 0 |
'------------------------------------------------------------------------------------
Private Function ExtractByte(lngWord As Long, intPosition As Integer) As Byte
    Dim lngTemp As Long
    Dim intByte As Integer

    If intPosition = 3 Then
        ' Byte 2
        lngTemp = lngWord

        ' Mask off byte and shift right 24 bits.
        '   Mask  -> 2130706432 = &H7F000000
        '   Shift -> Divide by 16777216
        lngTemp = (lngTemp And 2130706432) / 16777216

        ' Cast back to integer.
        intByte = lngTemp

    ElseIf intPosition = 2 Then
        ' Byte 2
        lngTemp = lngWord

        ' Mask off byte and shift right 16 bits.
        '   Mask  -> 16711680 = &HFF0000
        '   Shift -> Divide by 65536
        lngTemp = (lngTemp And 16711680) / 65536

        ' Cast back to integer.
        intByte = lngTemp

    ElseIf intPosition = 1 Then
        ' Byte 1
        lngTemp = lngWord

        ' Mask off high byte and shift right 8 bits.
        '   Mask  -> 65290 = &HFF00
        '   Shift -> Divide by 256
        lngTemp = (lngTemp And 65290) / 256

        ' Cast back to integer.
        intByte = lngTemp
    Else
        ' Byte 0
        intByte = lngWord And &HFF
    End If

    ExtractByte = intByte
End Function
```

```
'-----------------------------------------------------------------------------------
' Wav_Stop - Stop the currently playing wav.
'-----------------------------------------------------------------------------------
Public Sub Wav_Stop()
    Dim lngStatus As Long

    lngStatus = PlaySoundMemory(ByVal 0&, ByVal 0&, SND_PURGE Or SND_NODEFAULT)
End Sub
```

# APPENDIX III: WINDOWS WAVEFORM AUDIO AND SOUND API

```
/****************************************************************************

                       Waveform audio support

****************************************************************************/

/* waveform audio error return values */
#define WAVERR_BADFORMAT      (WAVERR_BASE + 0)    /* unsupported wave format */
#define WAVERR_STILLPLAYING   (WAVERR_BASE + 1)    /* still something playing */
#define WAVERR_UNPREPARED     (WAVERR_BASE + 2)    /* header not prepared */
#define WAVERR_SYNC         (WAVERR_BASE + 3)    /* device is synchronous */
#define WAVERR_LASTERROR      (WAVERR_BASE + 3)    /* last error in range */

/* waveform audio data types */
DECLARE_HANDLE(HWAVE);
DECLARE_HANDLE(HWAVEIN);
DECLARE_HANDLE(HWAVEOUT);
#ifndef _WIN32_VXD
typedef HWAVEIN FAR *LPHWAVEIN;
typedef HWAVEOUT FAR *LPHWAVEOUT;
typedef DRVCALLBACK WAVECALLBACK;
typedef WAVECALLBACK FAR *LPWAVECALLBACK;
#endif

/* wave callback messages */
#define WOM_OPEN       MM_WOM_OPEN
#define WOM_CLOSE      MM_WOM_CLOSE
#define WOM_DONE       MM_WOM_DONE
#define WIM_OPEN       MM_WIM_OPEN
#define WIM_CLOSE      MM_WIM_CLOSE
#define WIM_DATA       MM_WIM_DATA

/* device ID for wave device mapper */
#define WAVE_MAPPER     ((UINT)-1)

/* flags for dwFlags parameter in waveOutOpen() and waveInOpen() */
#define  WAVE_FORMAT_QUERY        0x0001
#define  WAVE_ALLOWSYNC           0x0002
#if(WINVER >= 0x0400)
#define  WAVE_MAPPED              0x0004
#define  WAVE_FORMAT_DIRECT       0x0008
#define  WAVE_FORMAT_DIRECT_QUERY  (WAVE_FORMAT_QUERY | WAVE_FORMAT_DIRECT)
#endif /* WINVER >= 0x0400 */

/* wave data block header */
typedef struct wavehdr_tag {
   LPSTR       lpData;                /* pointer to locked data buffer */
   DWORD       dwBufferLength;        /* length of data buffer */
   DWORD       dwBytesRecorded;       /* used for input only */
   DWORD       dwUser;                /* for client's use */
   DWORD       dwFlags;               /* assorted flags (see defines) */
   DWORD       dwLoops;               /* loop control counter */
   struct wavehdr_tag FAR *lpNext;    /* reserved for driver */
```

```
    DWORD       reserved;               /* reserved for driver */
} WAVEHDR, *PWAVEHDR, NEAR *NPWAVEHDR, FAR *LPWAVEHDR;

/* flags for dwFlags field of WAVEHDR */
#define WHDR_DONE       0x00000001  /* done bit */
#define WHDR_PREPARED   0x00000002  /* set if this header has been prepared */
#define WHDR_BEGINLOOP  0x00000004  /* loop start block */
#define WHDR_ENDLOOP    0x00000008  /* loop end block */
#define WHDR_INQUEUE    0x00000010  /* reserved for driver */

/* waveform output device capabilities structure */
#ifdef _WIN32

typedef struct tagWAVEOUTCAPSA {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    MMVERSION vDriverVersion;      /* version of the driver */
    CHAR    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of sources supported */
    WORD    wReserved1;            /* packing */
    DWORD   dwSupport;             /* functionality supported by driver */
} WAVEOUTCAPSA, *PWAVEOUTCAPSA, *NPWAVEOUTCAPSA, *LPWAVEOUTCAPSA;
typedef struct tagWAVEOUTCAPSW {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    MMVERSION vDriverVersion;      /* version of the driver */
    WCHAR   szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of sources supported */
    WORD    wReserved1;            /* packing */
    DWORD   dwSupport;             /* functionality supported by driver */
} WAVEOUTCAPSW, *PWAVEOUTCAPSW, *NPWAVEOUTCAPSW, *LPWAVEOUTCAPSW;
#ifdef UNICODE
typedef WAVEOUTCAPSW WAVEOUTCAPS;
typedef PWAVEOUTCAPSW PWAVEOUTCAPS;
typedef NPWAVEOUTCAPSW NPWAVEOUTCAPS;
typedef LPWAVEOUTCAPSW LPWAVEOUTCAPS;
#else
typedef WAVEOUTCAPSA WAVEOUTCAPS;
typedef PWAVEOUTCAPSA PWAVEOUTCAPS;
typedef NPWAVEOUTCAPSA NPWAVEOUTCAPS;
typedef LPWAVEOUTCAPSA LPWAVEOUTCAPS;
#endif // UNICODE

#else
typedef struct waveoutcaps_tag {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    VERSION vDriverVersion;        /* version of the driver */
    char    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of sources supported */
    DWORD   dwSupport;             /* functionality supported by driver */
} WAVEOUTCAPS, *PWAVEOUTCAPS, NEAR *NPWAVEOUTCAPS, FAR *LPWAVEOUTCAPS;
#endif
```

```
/* flags for dwSupport field of WAVEOUTCAPS */
#define WAVECAPS_PITCH          0x0001   /* supports pitch control */
#define WAVECAPS_PLAYBACKRATE   0x0002   /* supports playback rate control */
#define WAVECAPS_VOLUME         0x0004   /* supports volume control */
#define WAVECAPS_LRVOLUME       0x0008   /* separate left-right volume control */
#define WAVECAPS_SYNC           0x0010
#define WAVECAPS_SAMPLEACCURATE 0x0020
#define WAVECAPS_DIRECTSOUND    0x0040

/* waveform input device capabilities structure */
#ifdef _WIN32

typedef struct tagWAVEINCAPSA {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    MMVERSION vDriverVersion;      /* version of the driver */
    CHAR    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of channels supported */
    WORD    wReserved1;            /* structure packing */
} WAVEINCAPSA, *PWAVEINCAPSA, *NPWAVEINCAPSA, *LPWAVEINCAPSA;
typedef struct tagWAVEINCAPSW {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    MMVERSION vDriverVersion;      /* version of the driver */
    WCHAR   szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of channels supported */
    WORD    wReserved1;            /* structure packing */
} WAVEINCAPSW, *PWAVEINCAPSW, *NPWAVEINCAPSW, *LPWAVEINCAPSW;
#ifdef UNICODE
typedef WAVEINCAPSW WAVEINCAPS;
typedef PWAVEINCAPSW PWAVEINCAPS;
typedef NPWAVEINCAPSW NPWAVEINCAPS;
typedef LPWAVEINCAPSW LPWAVEINCAPS;
#else
typedef WAVEINCAPSA WAVEINCAPS;
typedef PWAVEINCAPSA PWAVEINCAPS;
typedef NPWAVEINCAPSA NPWAVEINCAPS;
typedef LPWAVEINCAPSA LPWAVEINCAPS;
#endif // UNICODE

#else
typedef struct waveincaps_tag {
    WORD    wMid;                  /* manufacturer ID */
    WORD    wPid;                  /* product ID */
    VERSION vDriverVersion;        /* version of the driver */
    char    szPname[MAXPNAMELEN];  /* product name (NULL terminated string) */
    DWORD   dwFormats;             /* formats supported */
    WORD    wChannels;             /* number of channels supported */
} WAVEINCAPS, *PWAVEINCAPS, NEAR *NPWAVEINCAPS, FAR *LPWAVEINCAPS;
#endif

/* defines for dwFormat field of WAVEINCAPS and WAVEOUTCAPS */
#define WAVE_INVALIDFORMAT     0x00000000       /* invalid format */
```

```
#define WAVE_FORMAT_1M08       0x00000001       /* 11.025 kHz, Mono,   8-bit  */
#define WAVE_FORMAT_1S08       0x00000002       /* 11.025 kHz, Stereo, 8-bit  */
#define WAVE_FORMAT_1M16       0x00000004       /* 11.025 kHz, Mono,   16-bit */
#define WAVE_FORMAT_1S16       0x00000008       /* 11.025 kHz, Stereo, 16-bit */
#define WAVE_FORMAT_2M08       0x00000010       /* 22.05  kHz, Mono,   8-bit  */
#define WAVE_FORMAT_2S08       0x00000020       /* 22.05  kHz, Stereo, 8-bit  */
#define WAVE_FORMAT_2M16       0x00000040       /* 22.05  kHz, Mono,   16-bit */
#define WAVE_FORMAT_2S16       0x00000080       /* 22.05  kHz, Stereo, 16-bit */
#define WAVE_FORMAT_4M08       0x00000100       /* 44.1   kHz, Mono,   8-bit  */
#define WAVE_FORMAT_4S08       0x00000200       /* 44.1   kHz, Stereo, 8-bit  */
#define WAVE_FORMAT_4M16       0x00000400       /* 44.1   kHz, Mono,   16-bit */
#define WAVE_FORMAT_4S16       0x00000800       /* 44.1   kHz, Stereo, 16-bit */

/* OLD general waveform format structure (information common to all formats) */
typedef struct waveformat_tag {
   WORD    wFormatTag;        /* format type */
   WORD    nChannels;         /* number of channels (i.e. mono, stereo, etc.) */
   DWORD   nSamplesPerSec;    /* sample rate */
   DWORD   nAvgBytesPerSec;   /* for buffer estimation */
   WORD    nBlockAlign;       /* block size of data */
} WAVEFORMAT, *PWAVEFORMAT, NEAR *NPWAVEFORMAT, FAR *LPWAVEFORMAT;

/* flags for wFormatTag field of WAVEFORMAT */
#define WAVE_FORMAT_PCM     1

/* specific waveform format structure for PCM data */
typedef struct pcmwaveformat_tag {
   WAVEFORMAT  wf;
   WORD        wBitsPerSample;
} PCMWAVEFORMAT, *PPCMWAVEFORMAT, NEAR *NPPCMWAVEFORMAT, FAR
*LPPCMWAVEFORMAT;

#ifndef _WAVEFORMATEX_
#define _WAVEFORMATEX_

/*
 *  extended waveform format structure used for all non-PCM formats. this
 *  structure is common to all non-PCM formats.
 */
typedef struct tWAVEFORMATEX
{
   WORD        wFormatTag;         /* format type */
   WORD        nChannels;          /* number of channels (i.e. mono, stereo...) */
   DWORD       nSamplesPerSec;     /* sample rate */
   DWORD       nAvgBytesPerSec;    /* for buffer estimation */
   WORD        nBlockAlign;        /* block size of data */
   WORD        wBitsPerSample;     /* number of bits per sample of mono data */
   WORD        cbSize;             /* the count in bytes of the size of */
                                   /* extra information (after cbSize) */
} WAVEFORMATEX, *PWAVEFORMATEX, NEAR *NPWAVEFORMATEX, FAR *LPWAVEFORMATEX;
typedef const WAVEFORMATEX FAR *LPCWAVEFORMATEX;

#endif /* _WAVEFORMATEX_ */

#ifndef _WIN32_VXD
/* waveform audio function prototypes */
```

```
WINMMAPI UINT WINAPI waveOutGetNumDevs(void);

#ifdef _WIN32

WINMMAPI MMRESULT WINAPI waveOutGetDevCapsA(UINT uDeviceID, LPWAVEOUTCAPSA pwoc,
UINT cbwoc);
WINMMAPI MMRESULT WINAPI waveOutGetDevCapsW(UINT uDeviceID, LPWAVEOUTCAPSW pwoc,
UINT cbwoc);
#ifdef UNICODE
#define waveOutGetDevCaps  waveOutGetDevCapsW
#else
#define waveOutGetDevCaps  waveOutGetDevCapsA
#endif // !UNICODE

#else
WINMMAPI MMRESULT WINAPI waveOutGetDevCaps(UINT uDeviceID, LPWAVEOUTCAPS pwoc, UINT
cbwoc);
#endif

#if (WINVER >= 0x0400)
WINMMAPI MMRESULT WINAPI waveOutGetVolume(HWAVEOUT hwo, LPDWORD pdwVolume);
WINMMAPI MMRESULT WINAPI waveOutSetVolume(HWAVEOUT hwo, DWORD dwVolume);
#else
WINMMAPI MMRESULT WINAPI waveOutGetVolume(UINT uId, LPDWORD pdwVolume);
WINMMAPI MMRESULT WINAPI waveOutSetVolume(UINT uId, DWORD dwVolume);
#endif

#ifdef _WIN32

WINMMAPI MMRESULT WINAPI waveOutGetErrorTextA(MMRESULT mmrError, LPSTR pszText, UINT
cchText);
WINMMAPI MMRESULT WINAPI waveOutGetErrorTextW(MMRESULT mmrError, LPWSTR pszText, UINT
cchText);
#ifdef UNICODE
#define waveOutGetErrorText  waveOutGetErrorTextW
#else
#define waveOutGetErrorText  waveOutGetErrorTextA
#endif // !UNICODE

#else
MMRESULT WINAPI waveOutGetErrorText(MMRESULT mmrError, LPSTR pszText, UINT cchText);
#endif

WINMMAPI MMRESULT WINAPI waveOutOpen(LPHWAVEOUT phwo, UINT uDeviceID,
   LPCWAVEFORMATEX pwfx, DWORD dwCallback, DWORD dwInstance, DWORD fdwOpen);

WINMMAPI MMRESULT WINAPI waveOutClose(HWAVEOUT hwo);
WINMMAPI MMRESULT WINAPI waveOutPrepareHeader(HWAVEOUT hwo, LPWAVEHDR pwh, UINT
cbwh);
WINMMAPI MMRESULT WINAPI waveOutUnprepareHeader(HWAVEOUT hwo, LPWAVEHDR pwh, UINT
cbwh);
WINMMAPI MMRESULT WINAPI waveOutWrite(HWAVEOUT hwo, LPWAVEHDR pwh, UINT cbwh);
WINMMAPI MMRESULT WINAPI waveOutPause(HWAVEOUT hwo);
WINMMAPI MMRESULT WINAPI waveOutRestart(HWAVEOUT hwo);
WINMMAPI MMRESULT WINAPI waveOutReset(HWAVEOUT hwo);
WINMMAPI MMRESULT WINAPI waveOutBreakLoop(HWAVEOUT hwo);
WINMMAPI MMRESULT WINAPI waveOutGetPosition(HWAVEOUT hwo, LPMMTIME pmmt, UINT cbmmt);
WINMMAPI MMRESULT WINAPI waveOutGetPitch(HWAVEOUT hwo, LPDWORD pdwPitch);
WINMMAPI MMRESULT WINAPI waveOutSetPitch(HWAVEOUT hwo, DWORD dwPitch);
WINMMAPI MMRESULT WINAPI waveOutGetPlaybackRate(HWAVEOUT hwo, LPDWORD pdwRate);
WINMMAPI MMRESULT WINAPI waveOutSetPlaybackRate(HWAVEOUT hwo, DWORD dwRate);
WINMMAPI MMRESULT WINAPI waveOutGetID(HWAVEOUT hwo, LPUINT puDeviceID);

#if (WINVER >= 0x030a)
#ifdef _WIN32
WINMMAPI MMRESULT WINAPI waveOutMessage(HWAVEOUT hwo, UINT uMsg, DWORD dw1, DWORD
dw2);
#else
DWORD WINAPI waveOutMessage(HWAVEOUT hwo, UINT uMsg, DWORD dw1, DWORD dw2);
#endif
#endif /* ifdef WINVER >= 0x030a */

WINMMAPI UINT WINAPI waveInGetNumDevs(void);

#ifdef _WIN32

WINMMAPI MMRESULT WINAPI waveInGetDevCapsA(UINT uDeviceID, LPWAVEINCAPSA pwic, UINT
cbwic);
WINMMAPI MMRESULT WINAPI waveInGetDevCapsW(UINT uDeviceID, LPWAVEINCAPSW pwic, UINT
cbwic);
#ifdef UNICODE
#define waveInGetDevCaps  waveInGetDevCapsW
#else
#define waveInGetDevCaps  waveInGetDevCapsA
#endif // !UNICODE

#else
MMRESULT WINAPI waveInGetDevCaps(UINT uDeviceID, LPWAVEINCAPS pwic, UINT cbwic);
#endif

#ifdef _WIN32

WINMMAPI MMRESULT WINAPI waveInGetErrorTextA(MMRESULT mmrError, LPSTR pszText, UINT
cchText);
WINMMAPI MMRESULT WINAPI waveInGetErrorTextW(MMRESULT mmrError, LPWSTR pszText, UINT
cchText);
#ifdef UNICODE
#define waveInGetErrorText  waveInGetErrorTextW
#else
#define waveInGetErrorText  waveInGetErrorTextA
#endif // !UNICODE

#else
MMRESULT WINAPI waveInGetErrorText(MMRESULT mmrError, LPSTR pszText, UINT cchText);
#endif

WINMMAPI MMRESULT WINAPI waveInOpen(LPHWAVEIN phwi, UINT uDeviceID,
   LPCWAVEFORMATEX pwfx, DWORD dwCallback, DWORD dwInstance, DWORD fdwOpen);

WINMMAPI MMRESULT WINAPI waveInClose(HWAVEIN hwi);
WINMMAPI MMRESULT WINAPI waveInPrepareHeader(HWAVEIN hwi, LPWAVEHDR pwh, UINT cbwh);
WINMMAPI MMRESULT WINAPI waveInUnprepareHeader(HWAVEIN hwi, LPWAVEHDR pwh, UINT
cbwh);
WINMMAPI MMRESULT WINAPI waveInAddBuffer(HWAVEIN hwi, LPWAVEHDR pwh, UINT cbwh);
WINMMAPI MMRESULT WINAPI waveInStart(HWAVEIN hwi);
WINMMAPI MMRESULT WINAPI waveInStop(HWAVEIN hwi);
WINMMAPI MMRESULT WINAPI waveInReset(HWAVEIN hwi);
WINMMAPI MMRESULT WINAPI waveInGetPosition(HWAVEIN hwi, LPMMTIME pmmt, UINT cbmmt);
WINMMAPI MMRESULT WINAPI waveInGetID(HWAVEIN hwi, LPUINT puDeviceID);

#if (WINVER >= 0x030a)
#ifdef _WIN32
WINMMAPI MMRESULT WINAPI waveInMessage(HWAVEIN hwi, UINT uMsg, DWORD dw1, DWORD
dw2);
#else
DWORD WINAPI waveInMessage(HWAVEIN hwi, UINT uMsg, DWORD dw1, DWORD dw2);
#endif
#endif /* ifdef WINVER >= 0x030a */

#endif /* ifndef _WIN32_VXD */

#endif  /* ifndef MMNOWAVE */


/****************************************************************************

                                        Sound support

****************************************************************************/

#ifdef _WIN32

WINMMAPI BOOL WINAPI sndPlaySoundA(LPCSTR pszSound, UINT fuSound);
WINMMAPI BOOL WINAPI sndPlaySoundW(LPCWSTR pszSound, UINT fuSound);
#ifdef UNICODE
#define sndPlaySound  sndPlaySoundW
#else
#define sndPlaySound  sndPlaySoundA
#endif // !UNICODE

#else
BOOL WINAPI sndPlaySound(LPCSTR pszSound, UINT fuSound);
#endif

/*
 *  flag values for fuSound and fdwSound arguments on [snd]PlaySound
 */
#define SND_SYNC            0x0000  /* play synchronously (default) */
#define SND_ASYNC           0x0001  /* play asynchronously */
#define SND_NODEFAULT       0x0002  /* silence (!default) if sound not found */
#define SND_MEMORY          0x0004  /* pszSound points to a memory file */
#define SND_LOOP            0x0008  /* loop the sound until next sndPlaySound */
#define SND_NOSTOP          0x0010  /* don't stop any currently playing sound */

#define SND_NOWAIT                0x00002000L /* don't wait if the driver is busy */
#define SND_ALIAS       0x00010000L /* name is a registry alias */
#define SND_ALIAS_ID              0x00110000L /* alias is a predefined ID */
```

```
#define SND_FILENAME    0x00020000L /* name is file name */
#define SND_RESOURCE    0x00040004L /* name is resource name or atom */
#if(WINVER >= 0x0400)
#define SND_PURGE           0x0040  /* purge non-static events for task */
#define SND_APPLICATION     0x0080  /* look for application specific association */
#endif /* WINVER >= 0x0400 */


#define SND_ALIAS_START     0           /* alias base */

#ifdef _WIN32
#define     sndAlias(ch0, ch1)      (SND_ALIAS_START + (DWORD)(BYTE)(ch0) | ((DWORD)(BYTE)(ch1) <<
8))

#define SND_ALIAS_SYSTEMASTERISK        sndAlias('S', '*')
#define SND_ALIAS_SYSTEMQUESTION        sndAlias('S', '?')
#define SND_ALIAS_SYSTEMHAND            sndAlias('S', 'H')
#define SND_ALIAS_SYSTEMEXIT            sndAlias('S', 'E')
#define SND_ALIAS_SYSTEMSTART           sndAlias('S', 'S')
#define SND_ALIAS_SYSTEMWELCOME         sndAlias('S', 'W')
#define SND_ALIAS_SYSTEMEXCLAMATION     sndAlias('S', '!')
#define SND_ALIAS_SYSTEMDEFAULT         sndAlias('S', 'D')

WINMMAPI BOOL WINAPI PlaySoundA(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
WINMMAPI BOOL WINAPI PlaySoundW(LPCWSTR pszSound, HMODULE hmod, DWORD fdwSound);
#ifdef UNICODE
#define PlaySound  PlaySoundW
#else
#define PlaySound  PlaySoundA
#endif // !UNICODE

#else
BOOL WINAPI PlaySound(LPCSTR pszSound, HMODULE hmod, DWORD fdwSound);
#endif

#endif  /* ifndef _WIN32_VXD */
#endif  /* ifndef MMNOSOUND */
```

# APPENDIX IV: SIMPLESCOPE CODE LISTING

## Oscilliscope Code

```
‘fundamental.frm code
'Left channel (channel 1) is the input channel
'Right channel (channel 2) is the output channel

Option Explicit 'All variables must first be declared

'Global Variable declarations

Private DevHandle As Long

Private InData(0 To 2645999) As Integer     'Contains the raw bytes from sound card
Private InDataWord(0 To 2645999) As Long    'Raw bytes converted to values
Private Voltages(0 To 2645999) As Single    'Calibrated voltages
Private NumSampleBytes As Long              'Number of sample bytes
Private UpperBound As Long                  'Number of sampled values on both channels
Private Inited As Boolean
Private MinCal, MaxCal As Long              'Used for calibrating voltage
Public MinHeight As Long, MinWidth As Long
Private FirstTime As Single, FirstVoltage As Single
Private SecondClick As Boolean
Private DivSize As Long
Private TriggerOnce As Boolean
Private Triggered As Boolean
Private TriggerIsOn As Boolean
Private PlotInc As Long
Private OldCursors(0 To 1, 0 To 1) As Single
Private ZoomOn, ZoomBegun As Boolean        'Not Used
Private zoomrect(0 To 1, 0 To 1) As Single  'Not Used

'Calibration voltage
Private Const VOLTS_PER_DIV = 0.2

'Windows API required types

Private Type WaveFormatEx
   FormatTag As Integer
   Channels As Integer
   SamplesPerSec As Long
   AvgBytesPerSec As Long
   BlockAlign As Integer
   BitsPerSample As Integer
   ExtraDataSize As Integer
End Type

Private Type WaveHdr
   lpData As Long
   dwBufferLength As Long
   dwBytesRecorded As Long
   dwUser As Long
   dwFlags As Long
   dwLoops As Long
   lpNext As Long 'wavehdr_tag
   Reserved As Long
End Type

Private Type WaveInCaps
   ManufacturerID As Integer      'wMid
   ProductID As Integer      'wPid
   DriverVersion As Long      'MMVERSIONS vDriverVersion
```

```
    ProductName(1 To 32) As Byte 'szPname[MAXPNAMELEN]
    Formats As Long
    Channels As Integer
    Reserved As Integer
End Type

'Windows API constants

Private Const WAVE_INVALIDFORMAT = &H0&             '/* invalid format */
Private Const WAVE_FORMAT_1M08 = &H1&               '/* 11.025 kHz, Mono,   8-bit
Private Const WAVE_FORMAT_1S08 = &H2&               '/* 11.025 kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_1M16 = &H4&               '/* 11.025 kHz, Mono,   16-bit
Private Const WAVE_FORMAT_1S16 = &H8&               '/* 11.025 kHz, Stereo, 16-bit
Private Const WAVE_FORMAT_2M08 = &H10&              '/* 22.05  kHz, Mono,   8-bit
Private Const WAVE_FORMAT_2S08 = &H20&              '/* 22.05  kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_2M16 = &H40&              '/* 22.05  kHz, Mono,   16-bit
Private Const WAVE_FORMAT_2S16 = &H80&              '/* 22.05  kHz, Stereo, 16-bit
Private Const WAVE_FORMAT_4M08 = &H100&             '/* 44.1   kHz, Mono,   8-bit
Private Const WAVE_FORMAT_4S08 = &H200&             '/* 44.1   kHz, Stereo, 8-bit
Private Const WAVE_FORMAT_4M16 = &H400&             '/* 44.1   kHz, Mono,   16-bit
Private Const WAVE_FORMAT_4S16 = &H800&             '/* 44.1   kHz, Stereo, 16-bit

Private Const WAVE_FORMAT_PCM = 1

Private Const WHDR_DONE = &H1&           '/* done bit */
Private Const WHDR_PREPARED = &H2&         '/* set if this header has been prepared */
Private Const WHDR_BEGINLOOP = &H4&        '/* loop start block */
Private Const WHDR_ENDLOOP = &H8&          '/* loop end block */
Private Const WHDR_INQUEUE = &H10&         '/* reserved for driver */

Private Const WAVE_ALLOWSYNC = &H2&

Private Const WIM_OPEN = &H3BE
Private Const WIM_CLOSE = &H3BF
Private Const WIM_DATA = &H3C0

'Windows API functions

Private Declare Function waveInAddBuffer Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
Private Declare Function waveInPrepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
Private Declare Function waveInUnprepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer
As Long, ByVal WaveHdrStructSize As Long) As Long

Private Declare Function waveInGetNumDevs Lib "winmm" () As Long
Private Declare Function waveInGetDevCaps Lib "winmm" Alias "waveInGetDevCapsA" (ByVal uDeviceID As Long, ByVal
WaveInCapsPointer As Long, ByVal WaveInCapsStructSize As Long) As Long

Private Declare Function waveInOpen Lib "winmm" (WaveDeviceInputHandle As Long, ByVal WhichDevice As Long, ByVal
WaveFormatExPointer As Long, ByVal CallBack As Long, ByVal CallBackInstance As Long, ByVal Flags As Long) As Long
Private Declare Function waveInClose Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long

Private Declare Function waveInStart Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long
Private Declare Function waveInReset Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long
Private Declare Function waveInStop Lib "winmm" (ByVal WaveDeviceInputHandle As Long) As Long

'This function converts an integer into a long; since Visual Basic Integers are signed 16-bit numbers,
'any 16-bit number above 32768 is misrepresented as a negative number.  By converting to a long using this function,
'data is automatically converted to a +/- y-axis scale

Public Function MAKEINTLONG(SignedInt As Integer) As Long
```

```
   MAKEINTLONG = SignedInt + 32768
End Function

'Ensure that a sound card exists and that it can capture at least 16-bit stereo at 11kHz
Sub InitDevices()
   Dim Caps As WaveInCaps, Which As Long
   DevicesBox.Clear
   For Which = 0 To waveInGetNumDevs - 1
      Call waveInGetDevCaps(Which, VarPtr(Caps), Len(Caps))
      'If Caps.Formats And WAVE_FORMAT_1S16 Then 'Now is 1S16 -- Check for devices that can do stereo 16-bit 11kHz
      If Caps.Formats And WAVE_FORMAT_1S16 Then
         Call DevicesBox.AddItem(StrConv(Caps.ProductName, vbUnicode), Which)
      End If
   Next
   If DevicesBox.ListCount = 0 Then
      MsgBox "You have no audio input devices!", vbCritical, "Ack!"
      End
   End If
   DevicesBox.ListIndex = 0

   'initialize the graphic permanency for picture boxes
   Scope(0).AutoRedraw = True
   Scope(1).AutoRedraw = True
   Yaxis.AutoRedraw = True
   Xaxis.AutoRedraw = True

   Call CheckSFreqs(DevicesBox.ListIndex)

End Sub

'List all the sample frequencies supported by a particular sound card
Sub CheckSFreqs(ByVal DevNum As Long)
   Dim Caps As WaveInCaps

   Call waveInGetDevCaps(DevNum, VarPtr(Caps), Len(Caps))
   'These are all the 8-bit sampling frequencies, but may change to 16-bit
   If Caps.Formats And WAVE_FORMAT_1S16 Then
      Call SampleFreq.AddItem("11025", 0)
      SampleFreq.ListIndex = 0
   End If
   If Caps.Formats And WAVE_FORMAT_2S16 Then
      Call SampleFreq.AddItem("22050", 1)
      SampleFreq.ListIndex = 1
   End If
   If Caps.Formats And WAVE_FORMAT_4S16 Then
      Call SampleFreq.AddItem("44100", 2)
   End If

End Sub

'Quit out of SimpleScope

Private Sub Quit_Click()
   Unload Base
   Unload Stimulator
End Sub

'Make sure to only list sample frequencies supported by the "changed-to" sound card
Private Sub DevicesBox_Change()
   Dim DevNum As Long

   DevNum = DevicesBox.ListIndex
```

```
    Call CheckSFreqs(0)
End Sub

Private Sub Form_Load()
    'Do some value initializing here

    Tscale.AddItem "2.0"
    Tscale.AddItem "5.0"
    Tscale.AddItem "10.0"
    Tscale.AddItem "15.0"
    Tscale.AddItem "20.0"
    Tscale.AddItem "30.0"

    Tscale.ListIndex = 5
    Vscale.ListIndex = 9

    Call InitDevices

    'From Deeth Oscilloscope Program - needed to control the scaling of
    'the virtual oscilloscope window properly
    'Set the window procedure to my own (which restricts the
    'minimum size of the form...
    'Comment out the SetWindowLong line if you're working with it
    'in the development environment since it'll hang in stop mode.
     MinMaxProc.Proc = GetWindowLong(Me.HWnd, GWL_WNDPROC)
    'SetWindowLong Me.HWnd, GWL_WNDPROC, AddressOf WindowProc

    Yaxis.BackColor = BackColor
    Xaxis.BackColor = BackColor

    'VOLTS_PER_DIV = 0.7

    FirstTime = -1
    FirstVoltage = -1
    SecondClick = False

    TriggerIsOn = False
    Triggered = False

    Call DrawAxis

End Sub

'Function from Deeth Oscilloscope Program used with the MinMaxProc module
'To control virtual scope window scaling

Private Sub Form_Resize()
    Dim x As Integer

    Scope(0).Cls
    Scope(1).Cls

    Scope(0).ScaleHeight = 65536
    Scope(0).ScaleWidth = 255
    Scope(1).ScaleHeight = 65536
    Scope(1).ScaleWidth = 255

    Xaxis.ScaleWidth = 255
    Yaxis.ScaleHeight = 65536
    Yaxis.ScaleWidth = 10

    'Make the window resize now so that it doesn't interfere with redrawing the data
    DoEvents

    'For debugging
    SampleValues.Text = "Resized"

    Call DrawAxis

    'Redraw the data at the new size
    If Inited = True Then
        Call DrawData
    End If
End Sub


Private Sub Form_Unload(Cancel As Integer)
    ' make sure that I/O access is closed in case of a premature quit
    If DevHandle <> 0 Then
        Call DoStop
    End If
    Cancel = 0
End Sub

'Partial Function for software zooming

Private Sub Scope_MouseUp(Index As Integer, Button As Integer, Shift As Integer, x As Single, Y As Single)

    If ZoomOn Then
        zoomrect(1, 0) = x
        zoomrect(1, 1) = Y
        'Scope(0).Cls
        'Scope(0).Line (ZoomRect(0, 0), ZoomRect(0, 1))-(X, Y), RGB(0, 0, 0), B
        'Debug.Print Abs(zoomrect(0, 0) - X)
        'Scope(0).ScaleWidth = Abs(zoomrect(0, 0) - X)
        'Scope(0).ScaleHeight = Abs(zoomrect(0, 1) - Y)
        Call DrawZoom
    End If
End Sub

'Start continuous mode data acquisition

Private Sub StartButton_Click()
    Static WaveFormat As WaveFormatEx
    With WaveFormat
        .FormatTag = WAVE_FORMAT_PCM
        .Channels = 2 'Two channels -- left and right
        .SamplesPerSec = CLng(SampleFreq) 'Sampling Frequency
        .BitsPerSample = 16
        .BlockAlign = (.Channels * .BitsPerSample) \ 8
        .AvgBytesPerSec = .BlockAlign * .SamplesPerSec
        .ExtraDataSize = 0
    End With

    TriggerOnce = False

    'Disable quit button because we need to close the Capturing
    Quit.Enabled = False

    NumSampleBytes = CLng(CSng(SampleFreq) * CSng(Tscale)) * 4
    UpperBound = NumSampleBytes / 2

    Scope(0).ScaleWidth = UpperBound / 2 - 1
    Xaxis.ScaleWidth = UpperBound / 2 - 1

    Debug.Print UpperBound
    Debug.Print SampleFreq
    Debug.Print NumSampleBytes

    'Open the sound card recording device

    Debug.Print "waveInOpen:"; waveInOpen(DevHandle, DevicesBox.ListIndex, VarPtr(WaveFormat), 0, 0,
WAVE_ALLOWSYNC)

    If DevHandle = 0 Then
        Call MsgBox("Wave input device didn't open!", vbExclamation, "Ack!")
        Exit Sub
    End If
    Debug.Print " "; DevHandle

    Inited = True

    StopButton.Enabled = True
    StartButton.Enabled = False

    Call Visualize
End Sub
'Capture one virtual scope window's worth of data
Private Sub ManualTrigger_Click()
    Static WaveFormat As WaveFormatEx
    With WaveFormat
        .FormatTag = WAVE_FORMAT_PCM
        .Channels = 2 'Two channels -- left and right
        .SamplesPerSec = CLng(SampleFreq) 'Sampling Frequency
        .BitsPerSample = 16
        .BlockAlign = (.Channels * .BitsPerSample) \ 8
        .AvgBytesPerSec = .BlockAlign * .SamplesPerSec
        .ExtraDataSize = 0
    End With

    TriggerOnce = True

    'Disable quit button because we need to close the Capturing
    Quit.Enabled = False

    NumSampleBytes = CLng(CSng(SampleFreq) * CSng(Tscale)) * 4
    UpperBound = NumSampleBytes / 2

    Scope(0).ScaleWidth = UpperBound / 2 - 1

    Xaxis.ScaleWidth = UpperBound / 2 - 1

    'Open the sound card recording device
    Debug.Print "waveInOpen:"; waveInOpen(DevHandle, DevicesBox.ListIndex, VarPtr(WaveFormat), 0, 0,
WAVE_ALLOWSYNC)

    If DevHandle = 0 Then
        Call MsgBox("Wave input device didn't open!", vbExclamation, "Ack!")
        Exit Sub
    End If
    Debug.Print " "; DevHandle

    Inited = True

    StartButton.Enabled = False
    Call Visualize
End Sub


Private Sub StartStim_Click()
    Load Stimulator
    Stimulator.Show
End Sub

Private Sub StopButton_Click()
    Call DoStop
    Scope(0).ScaleMode = 3
    Quit.Enabled = True
End Sub

'Stop sound card capturing

Private Sub DoStop()
    Call waveInReset(DevHandle)
    Call waveInClose(DevHandle)
    DevHandle = 0
    StopButton.Enabled = False
    StartButton.Enabled = True
End Sub

'Captures the actual data and calibrates the voltages

Private Sub Visualize()
    Dim i As Integer
    Dim x As Long
    Static calmin As Long
    Static calmax As Long
    Dim w$
    Dim once As Boolean

    Static Wave As WaveHdr


    Wave.lpData = VarPtr(InData(0))
    Wave.dwBufferLength = NumSampleBytes '(in bytes) This is now 512 so there's still 256 samples per channel
    Wave.dwFlags = 0

    once = True
    Do
    If Not TriggerIsOn Or (TriggerIsOn And Triggered) Then
        Debug.Print "Wave.dwFlags:"; Wave.dwFlags
        Call waveInPrepareHeader(DevHandle, VarPtr(Wave), Len(Wave))
        Do
            Debug.Print "Wave.dwFlags:"; Hex(Wave.dwFlags)
            'Nothing -- we're waiting for the audio driver to mark
            'this wave chunk as done.
        Loop Until ((Wave.dwFlags And WHDR_PREPARED) = WHDR_PREPARED) Or DevHandle = 0
        'SampleWords.ForeColor = &HFF0000

        Call waveInAddBuffer(DevHandle, VarPtr(Wave), Len(Wave))

        If once Then
            Call waveInStart(DevHandle)
            once = False
        End If
```

```
Debug.Print "Wave.dwFlags:"; Hex(Wave.dwFlags)
Debug.Print "Right before Second Do"

Do
   Debug.Print "Wave.dwFlags:"; Hex(Wave.dwFlags)
   'Nothing -- we're waiting for the audio driver to mark
   'this wave chunk as done.
Loop Until ((Wave.dwFlags And WHDR_DONE) = WHDR_DONE) Or DevHandle = 0
'SampleWords.ForeColor = &HFF&
'If SampleWords.ForeColor = &HFF& Then
'    SampleWords.ForeColor = &HFF0000
'Else
   SampleWords.ForeColor = &HFF&
'End If

SampleWords.Text = Str(Wave.dwBytesRecorded) + " bytes captured" + Chr$(13) + Chr$(10)

Call waveInUnprepareHeader(DevHandle, VarPtr(Wave), Len(Wave))

DoEvents

If DevHandle = 0 Then
   'The device has closed...
   Exit Do
End If

Scope(0).Cls
Scope(1).Cls

'Shift the bits of each element of InData so the value is unsigned
For x = 0 To UpperBound - 1
   InDataWord(x) = MAKEINTLONG(InData(x))
Next

'Calculate the min. and max of the calibration channel (channel 2)
calmin = 32786
calmax = 32786
For x = 0 To UpperBound / 2 - 1 '(NumSampleBytes / 4) - 1
   If calmin > InDataWord(x * 2 + 1) Then
      calmin = InDataWord(x * 2 + 1)
   End If
   If calmax < InDataWord(x * 2 + 1) Then
      calmax = InDataWord(x * 2 + 1)
   End If
Next
MinCal = calmin
MaxCal = calmax

' number of 16-bit values per division
DivSize = MaxCal - MinCal

Min.Text = MinCal
Max.Text = MaxCal

' adjust the voltage scale
Scope(0).ScaleHeight = 2 * CSng(Vscale) '65536 / DivSize * VOLTS_PER_DIV
Scope(1).ScaleHeight = 2 * CSng(Vscale)
'Yaxis.ScaleHeight = 2 * CSng(Vscale)

' Calculate calibrated voltage on channel 1
For x = 0 To UpperBound - 1
   Voltages(x) = InData(x) / DivSize * VOLTS_PER_DIV + CSng(Vscale)
```

```
            If TriggerIsOn And Not Triggered Then
                If Abs(Voltages(x) - CSng(Vscale)) > CSng(TrigValue.Text) Then
                    Triggered = True
                End If
            End If
        Next

        ' Draw the data
        Call DrawData
        Call DrawAxis
    Else
        Scope(0).Cls
        Call DrawAxis
    End If
    DoEvents
    Loop While DevHandle <> 0 And Not TriggerOnce 'While the audio device is open

    If TriggerOnce Then
        Call DoStop
        Scope(0).ScaleMode = 3
        Quit.Enabled = True
    End If

End Sub

'Draw the data on-screen

Private Sub DrawData()
    Static x As Long
    Dim temp As Long
    Dim A$

    Scope(0).CurrentX = -1
    Scope(0).CurrentY = Scope(0).ScaleHeight \ 2
    Scope(1).CurrentX = -1
    Scope(1).CurrentY = Scope(0).ScaleHeight \ 2

    A$ = "Value # " + "Result" + Chr$(13) + Chr$(10)

    temp = 0
    Scope(1).PSet (0, 0)
    If Not TriggerIsOn Or (TriggerIsOn And Triggered) Then
    PlotInc = (UpperBound / 2 - 1) / 500 'Scope(0).ScaleWidth
    If PlotInc = 2 Or PlotInc = 1 Or PlotInc = 0 Then
        PlotInc = 2
    Else
        PlotInc = PlotInc
    End If
    For x = 0 To UpperBound / 2 - 1 '(NumSampleBytes / 4) - 1
        If ((x + PlotInc) Mod (PlotInc / 2) = 0) Then
            'temp = temp + 1
            Scope(0).Line Step(0, 0)-(x, Voltages(x * 2)), RGB(0, 0, 0)
            'Scope(1).Line Step(0, 0)-(X, Voltages(X * 2 + 1)), RGB(0, 0, 0) 'For a good soundcard...
            '2 channels (stereo that is)
            'printing data point values

            'A$ = A$ + Str$(X + 1) + "         " + Str$(InDataWord(X * 2)) + Chr$(13) + Chr$(10)
        End If
    Next
    End If
    SampleWords.Text = SampleWords.Text + Str(temp) + " points plotted" + Chr$(13) + Chr$(10)
    SampleValues.Text = A$
    Scope(0).CurrentY = Scope(0).ScaleHeight \ 2
    Scope(1).CurrentY = Scope(0).ScaleHeight \ 2
End Sub

'Draw the data virtual scope grid

Private Sub DrawAxis()
    Dim x As Integer

    ' y-axis grid
    For x = 1 To 9
        Scope(0).Line (0, x * Scope(0).ScaleHeight / 10)-(Scope(0).ScaleWidth, x * Scope(0).ScaleHeight / 10), RGB(128, 128,
128)
        'If X = 5 Then SampleWords.Text = Scope(0).ScaleHeight * (X / 10)
    Next
    'Scope(0).Line (128, 0)-(128, 65536)

    ' x-axis grid
    For x = 0 To 5
        Scope(0).Line (x * (Scope(0).ScaleWidth / 5), 0)-(x * (Scope(0).ScaleWidth / 5), Scope(0).ScaleHeight), RGB(128, 128,
128)
    Next

    Call PrintLabels

    'Scope(0).DrawMode
End Sub

'Labels for x and y axises

Private Sub PrintLabels()
    Dim x As Integer
    Dim Y As Integer

    Dim YaxisHalfHeight As Integer

    YaxisHalfHeight = TextHeight("0") * (Yaxis.ScaleHeight / Yaxis.Height) / 2
    Yaxis.Cls
    Xaxis.Cls

    Xaxis.CurrentX = 0
    Xaxis.CurrentY = 0

    Xaxis.ForeColor = RGB(0, 0, 255)

    ' prints the x-axis labels
    Xaxis.Print "0.0"

    For x = 1 To 4
        Xaxis.CurrentX = x * (Xaxis.ScaleWidth / 5) - 6 * TextWidth("0") * (Xaxis.ScaleWidth / Xaxis.Width) / 2
        Xaxis.CurrentY = 0
        Xaxis.Print Round(x * Tscale.List(Tscale.ListIndex) / 5, 3)
    Next

    Xaxis.CurrentX = Xaxis.ScaleWidth - 5 * TextWidth("0") * (Xaxis.ScaleWidth / Xaxis.Width)
    Xaxis.CurrentY = 0

    Xaxis.Print Round(Tscale.List(Tscale.ListIndex), 4)
    ' *******************************************************************

    Yaxis.CurrentX = 0
    Yaxis.CurrentY = 0
    Yaxis.ForeColor = RGB(0, 0, 255)

    'prints the y-axis labels
    Yaxis.Print Vscale.List(Vscale.ListIndex)

    For Y = 1 To 9
        Yaxis.CurrentX = 0
        Yaxis.CurrentY = Y * Yaxis.ScaleHeight / 10 - YaxisHalfHeight
        Yaxis.Print Round((10 - Y) * (Vscale.List(Vscale.ListIndex) / 5) - Vscale.List(Vscale.ListIndex), 4)
    Next

    Yaxis.CurrentX = 0
    Yaxis.CurrentY = Yaxis.ScaleHeight - 2 * YaxisHalfHeight

    Yaxis.Print "-" + Vscale.List(Vscale.ListIndex)

End Sub

'Function for drawing cursors when the mouse button is clicked

Private Sub Scope_MouseDown(Index As Integer, Button As Integer, Shift As Integer, x As Single, Y As Single)

    If ZoomOn Then
        zoomrect(0, 0) = x
        zoomrect(0, 1) = Y
        ZoomBegun = True
        'Scope(Index).Drag (vbBeginDrag)

    Else
     If Button = vbLeftButton Then
        'Debug.Print SMtemp
        Scope(Index).FillStyle = 0
        Scope(Index).DrawWidth = 1
        If Not SecondClick Then
            'Clear the old cursor points
            DrawCursor OldCursors(0, 0), OldCursors(0, 1), RGB(255, 255, 255)
            DrawCursor OldCursors(1, 0), OldCursors(1, 1), RGB(255, 255, 255)

            'Draw the first new cursor
            OldCursors(0, 0) = x
            OldCursors(0, 1) = Y
            DrawCursor x, Y, RGB(255, 0, 0) 'could be commented out
            FirstTime = x
            FirstVoltage = Y
            SecondClick = True
            Scope(Index).ToolTipText = Str(x / Scope(Index).ScaleWidth * CSng(Tscale)) + ", " + Str((1 - 2 * Y /
Scope(Index).ScaleHeight) * CSng(Vscale))
            'Scope(Index).ToolTipText = Str(X) + ", " + Str((Scope(Index).ScaleHeight / 2) - Y)'could be commented out
        ElseIf SecondClick Then
            'Draw the second new cursor
            OldCursors(1, 0) = x
            OldCursors(1, 1) = Y
            DrawCursor x, Y, RGB(255, 0, 0) 'could be commented out
            SecondClick = False
            Scope(Index).ToolTipText = ""
        End If
        Scope(Index).DrawWidth = 1
        CurrentX = 0 'could be commented out
        CurrentY = Scope(Index).ScaleHeight / 2 'could be commented out
        Scope(Index).FillStyle = 1 'could be commented out
     End If
     End If
End Sub

'

Private Sub Scope_MouseMove(Index As Integer, Button As Integer, Shift As Integer, x As Single, Y As Single)
    'If ZoomBegun Then
    '    Scope(0).Line (ZoomRect(0, 0), ZoomRect(0, 1))-(X, Y), RGB(0, 255, 0), B
    'End If

     If SecondClick Then
        Vdiff.Text = Abs(FirstVoltage - Y) * 2 / Scope(Index).ScaleHeight * CSng(Vscale)
        Tdiff.Text = Abs(FirstTime - x) / Scope(Index).ScaleWidth * CSng(Tscale)
     End If
     Scope(Index).ToolTipText = Str(x / Scope(Index).ScaleWidth * CSng(Tscale)) + ", " + Str((1 - 2 * Y /
Scope(Index).ScaleHeight) * CSng(Vscale))
End Sub

Private Sub DrawCursor(MidPointX As Single, MidPointY As Single, RGBColor As Long)
    Scope(0).Line (MidPointX - 2, MidPointY + 2)-(MidPointX + 3, MidPointY - 3), RGBColor
    Scope(0).Line (MidPointX - 2, MidPointY - 2)-(MidPointX + 3, MidPointY + 3), RGBColor
End Sub

'Activates channel triggering...channel only records when TrigValue is reached

Private Sub TrigOn_Click()
    If TrigOn.Value = vbChecked Then
        TriggerIsOn = True
        TrigValue.Enabled = False
    ElseIf TrigOn.Value = vbUnchecked Then
        TriggerIsOn = False
        Triggered = False
        TrigValue.Enabled = True
    End If
End Sub

'Exports the current screen's datapoints to an Excel file

Private Sub ExportXL_Click()
    Dim x As Long
    Dim timeInc As Double
    Dim fs, tf

    On Error GoTo ErrHandler
    timeInc = CDbl(1 / CDbl(SampleFreq))

    CD1.ShowSave

    Set fs = CreateObject("Scripting.FileSystemObject")
    Set tf = fs.CreateTextFile(CD1.FileName, True)
    For x = 0 To UpperBound / 2 - 1
        tf.WriteLine (Str$(CSng(timeInc * CDbl(x))) + Chr$(9) + Str$(Voltages(x)))
    Next
    tf.Close

ErrHandler:
    'User pressed Cancel button.
    Exit Sub
End Sub

'Function for printing the virtual scope capture
```

```
Private Sub Capture_Click()
    Dim FormControl As Control
    Dim debugControls
    Dim FormName As String
   Dim Msg   ' Declare variable.
   On Error GoTo ErrorHandler   ' Set up error handler.
   PrintForm   ' Print form.
   Exit Sub
ErrorHandler:
   Msg = "The form can't be printed."
   MsgBox Msg   ' Display message.
   Resume Next


    Set debugControls = CreateObject("Scripting.Dictionary")

    'Base.PrintForm

    For Each FormControl In Base.Controls
        If Not TypeOf FormControl Is CommonDialog Then
            'The control is possibly visible to the user
            If FormControl.Visible = True Then
                If Not (FormControl.Name = "Scope" Or FormControl.Name = "Xaxis" Or FormControl.Name = "Yaxis") Then
                    FormControl.Visible = False
                End If
            ElseIf FormControl.Visible = False Then
                debugControls.Add FormControl.Name, "For Debugging"
            End If
        End If
    Next FormControl

    'Scope(0).Visible = True
    'Xaxis.Visible = True
    'Yaxis.Visible = True

    'Base.PrintForm

    'Base.Name = FormName

    'Call Controls_Appear

End Sub
'Auxillary function for virtual scope window printing
Private Sub Controls_Vanish()
    Dim FormControl As Control

    For Each FormControl In Base.Controls
        If Not TypeOf FormControl Is CommonDialog Then
            'The control is possibly visible to the user
            FormControl.Visible = False
        End If
    Next FormControl
End Sub
'Auxillary function for virtual scope window printing
Private Sub Controls_Appear()
    Dim FormControl As Control

    For Each FormControl In Base.Controls
        If Not TypeOf FormControl Is CommonDialog Then
            'Make all possibly visible controls visible
            FormControl.Visible = True
        End If
    Next FormControl

    'Hide the debugging controls
    Scope(1).Visible = False
    SampleWords.Visible = False
    Max.Visible = False
    Min.Visible = False
    Shape.Visible = False
    Zoom.Visible = False
End Sub
'For zooming
Private Sub Zoom_Click()
    ZoomOn = True
    'Scope(0).ScaleMode = 0
    'ZoomOut.Enabled = True
End Sub
'For zooming
Private Sub DrawZoom()
    Static x As Long
    Dim temp As Long
    Dim A$
    Dim zoomwidth As Long
    Dim zoomstart As Long

    zoomwidth = Abs(zoomrect(0, 0) - zoomrect(1, 0)) / Scope(0).ScaleWidth * (UpperBound / 2)
    zoomstart = CLng(zoomrect(0, 0) / Scope(0).ScaleWidth * (UpperBound / 2))
    Scope(0).Cls

    Scope(0).ScaleHeight = 2 * CSng(Vscale)
    Scope(0).ScaleWidth = zoomwidth

    Scope(0).CurrentX = -1
    Scope(0).CurrentY = Scope(0).ScaleHeight \ 2
    Scope(1).CurrentX = -1
    Scope(1).CurrentY = Scope(0).ScaleHeight \ 2

    temp = 0
    Scope(1).PSet (0, 0)
    If Not TriggerIsOn Or (TriggerIsOn And Triggered) Then
    PlotInc = (UpperBound / 2 - 1) / zoomwidth 'Scope(0).ScaleWidth
    If PlotInc = 2 Or PlotInc = 1 Or PlotInc = 0 Then
        PlotInc = 2
    Else
        PlotInc = PlotInc
    End If

    For x = 0 To zoomwidth '(NumSampleBytes / 4) - 1
        'If ((X + PlotInc) Mod (PlotInc / 2) = 0) Then
            temp = temp + 1
            'Debug.Print X
            Scope(0).Line Step(0, 0)-(x, Voltages((x + zoomstart) * 2)), RGB(0, 0, 0)
            'Scope(1).Line Step(0, 0)-(X, Voltages(X * 2 + 1)), RGB(0, 0, 0) 'For a good soundcard...
            '2 channels (stereo that is)
            'printing data point values

        'End If
    Next
    End If
    SampleWords.Text = "number points is" + Str(temp) + Chr(13) + Chr(10)
    Scope(0).CurrentY = Scope(0).ScaleHeight \ 2
    Scope(1).CurrentY = Scope(0).ScaleHeight \ 2
```

```
   Debug.Print Scope(0).ScaleWidth
   Debug.Print Scope(0).ScaleHeight
   Debug.Print UpperBound
End Sub

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

Declare Sub MoveMemory Lib "kernel32" Alias "RtlMoveMemory" ( _
  pDest As Any, pSource As Any, ByVal dwLength As Long)

'*----------------------------------------------------------*
'* Name       : MAKELONG                                   *
'*----------------------------------------------------------*
'* Purpose    : Combines two integers into a long integer.  *
'*----------------------------------------------------------*
'* Parameters : wLow   Required. Low WORD.                  *
'*            : wHigh  Required. High WORD.                 *
'*----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*            : language MAKELONG macro.                    *
'*----------------------------------------------------------*
Public Function MAKELONG(wLow As Long, wHigh As Long) As Long
  MAKELONG = LOWORD(wLow) Or (&H10000 * LOWORD(wHigh))
End Function


'*----------------------------------------------------------*
'* Name       : MAKELPARAM                                 *
'*----------------------------------------------------------*
'* Purpose    : Combines two integers into a long integer.  *
'*----------------------------------------------------------*
'* Parameters : wLow   Required. Low WORD.                  *
'*            : wHigh  Required. High WORD.                 *
'*----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*            : language MAKELPARAM macro.                  *
'*----------------------------------------------------------*
Public Function MAKELPARAM(wLow As Long, wHigh As Long) As Long
  MAKELPARAM = MAKELONG(wLow, wHigh)
End Function


'*----------------------------------------------------------*
'* Name       : MAKEWORD                                   *
'*----------------------------------------------------------*
'* Purpose    : Combines two integers into a 16-bit unsigned*
'*            : integer (word).                             *
'*----------------------------------------------------------*
'* Parameters : wLow   Required. Low BYTE.                  *
'*            : wHigh  Required. High BYTE.                 *
'*----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*            : language MAKEWORD macro.                    *
'*----------------------------------------------------------*
Public Function MAKEWORD(wLow As Long, wHigh As Long) As Long
  MAKEWORD = LOBYTE(wLow) Or (&H100& * LOBYTE(wHigh))
End Function


'*----------------------------------------------------------*
'* Name       : LOWORD                                     *
'*----------------------------------------------------------*
```

```
'* Purpose    : Returns the low 16-bit integer from a 32-bit*
'*           : long integer.                            *
'*-----------------------------------------------------------*
'* Parameters : dwValue Required. 32-bit long integer value.*
'*-----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*           : language LOWORD macro.                   *
'*-----------------------------------------------------------*
Public Function LOWORD(dwValue As Long) As Integer
  MoveMemory LOWORD, dwValue, 2
End Function


'*-----------------------------------------------------------*
'* Name       : HIWORD                                   *
'*-----------------------------------------------------------*
'* Purpose    : Returns the high 16-bit integer from a      *
'*           : 32-bit long integer.                      *
'*-----------------------------------------------------------*
'* Parameters : dwValue Required. 32-bit long integer value.*
'*-----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*           : language HIWORD macro.                   *
'*-----------------------------------------------------------*
Public Function HIWORD(dwValue As Long) As Integer
  MoveMemory HIWORD, ByVal VarPtr(dwValue) + 2, 2
End Function


'*-----------------------------------------------------------*
'* Name       : LOBYTE                                   *
'*-----------------------------------------------------------*
'* Purpose    : Returns the low 8-bit byte from a low word  *
'*           : of 32-bit long integer.                  *
'*-----------------------------------------------------------*
'* Parameters : dwValue Required. 32-bit long integer value.*
'*-----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*           : language LOBYTE macro.                   *
'*-----------------------------------------------------------*
Public Function LOBYTE(dwValue As Long) As Byte
  MoveMemory LOBYTE, LOWORD(dwValue), 1
End Function


'*-----------------------------------------------------------*
'* Name       : HIBYTE                                   *
'*-----------------------------------------------------------*
'* Purpose    : Returns the high 8-bit byte from a low word *
'*           : of 32-bit long integer.                  *
'*-----------------------------------------------------------*
'* Parameters : dwValue Required. 32-bit long integer value.*
'*-----------------------------------------------------------*
'* Description: This function is equivalent to the 'C'      *
'*           : language HIBYTE macro.                   *
'*-----------------------------------------------------------*
Public Function HIBYTE(dwValue As Long) As Byte
  MoveMemory HIBYTE, ByVal VarPtr(LOWORD(dwValue)) + 1, 1
End Function
*********************************************************************************************

'-----------------------------------------------------------------------
```

```
' This is a dopey window proceedure that restricts the size of a
' window.
'----------------------------------------------------------------------
' Murphy McCauley (MurphyMc@Concentric.NET) 08/06/99
'----------------------------------------------------------------------

Option Explicit

Declare Function GetWindowLong Lib "user32" Alias "GetWindowLongA" (ByVal HWnd As Long, ByVal nIndex As Long)
As Long
Declare Function SetWindowLong Lib "user32" Alias "SetWindowLongA" (ByVal HWnd As Long, ByVal nIndex As Long,
ByVal dwNewLong As Long) As Long
Declare Function CallWindowProc Lib "user32" Alias "CallWindowProcA" (ByVal lpPrevWndFunc As Long, ByVal HWnd As
Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long

Public Proc As Long
Public Const GWL_WNDPROC = (-4)

Private Const WM_GETMINMAXINFO = &H24

Declare Sub MoveMemory Lib "kernel32" Alias "RtlMoveMemory" ( _
   ByRef Destination As Any, _
   ByRef Source As Any, _
   ByVal ByteLength As Long)

Private Type POINTAPI
      x As Long
      Y As Long
End Type

Private Type MINMAXINFO
      ptReserved As POINTAPI
      ptMaxSize As POINTAPI
      ptMaxPosition As POINTAPI
      ptMinTrackSize As POINTAPI
      ptMaxTrackSize As POINTAPI
End Type


Function WindowProc(ByVal HWnd As Long, ByVal Msg As Long, ByVal wParam As Long, ByVal lParam As Long) As Long
   Static MinMax As MINMAXINFO

   Select Case Msg
      Case WM_GETMINMAXINFO
         Call MoveMemory(MinMax, ByVal lParam, Len(MinMax))
         MinMax.ptMinTrackSize.x = Base.MinWidth
         MinMax.ptMinTrackSize.Y = Base.MinHeight
         Call MoveMemory(ByVal lParam, MinMax, Len(MinMax))
      Case Else
         WindowProc = CallWindowProc(Proc, HWnd, Msg, wParam, lParam)
   End Select
End Function
```

## Stimulator Code

```
‘stimulator.frm code
Option Explicit

'CONSTANTS
Private Const ONE_PULSE_TYPE = 1
Private Const TWO_PULSE_TYPE = 2
Private Const TETANIC_TYPE = 3
Private Const CONTINUOUS = 0
Private Const once = 1

'Global Variables
Private DevHandle As Long
Private WaveFormType As Integer
Private Increment(0 To 6) As Single
Private OldTimeValue(0 To 6) As Single
Private OldTimeScroll(0 To 6) As Long
Private byteSound() As Byte

Private Type WaveFormatEx
   FormatTag As Integer
   Channels As Integer
   SamplesPerSec As Long
   AvgBytesPerSec As Long
   BlockAlign As Integer
   BitsPerSample As Integer
   ExtraDataSize As Integer
End Type

Private Type WaveHdr
   lpData As Long
   dwBufferLength As Long
   dwBytesRecorded As Long
   dwUser As Long
   dwFlags As Long
   dwLoops As Long
   lpNext As Long 'wavehdr_tag
   Reserved As Long
End Type

Private Type WaveOutCaps
   ManufacturerID As Integer      'wMid
   ProductID As Integer       'wPid
   DriverVersion As Long       'MMVERSIONS vDriverVersion
   ProductName(1 To 32) As Byte 'szPname[MAXPNAMELEN]
   Formats As Long
   Channels As Integer
   Reserved As Integer
   Support As Long           'functionality supported by driver
End Type

'flags for Support field of WAVEOUTCAPS
Private Const WAVECAPS_PITCH = &H1          '/* supports pitch control */
Private Const WAVECAPS_PLAYBACKRATE = &H2    '/* supports playback rate control */
Private Const WAVECAPS_VOLUME = &H4           '/* supports volume control */
Private Const WAVECAPS_LRVOLUME = &H8         '/* separate left-right volume control */
Private Const WAVECAPS_SYNC = &H10
Private Const WAVECAPS_SAMPLEACCURATE = &H20
Private Const WAVECAPS_DIRECTSOUND = &H40

'Access the Windows API library functions used for outputting sound

Private Declare Function waveOutGetNumDevs Lib "winmm" () As Long
Private Declare Function waveOutGetDevCaps Lib "winmm" Alias "waveOutGetDevCapsA" (ByVal uDeviceID As Long,
ByVal WaveOutCapsPointer As Long, ByVal WaveOutCapsStructSize As Long) As Long

Private Declare Function waveInAddBuffer Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
```

```
Private Declare Function waveInPrepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer As
Long, ByVal WaveHdrStructSize As Long) As Long
Private Declare Function waveInUnprepareHeader Lib "winmm" (ByVal InputDeviceHandle As Long, ByVal WaveHdrPointer
As Long, ByVal WaveHdrStructSize As Long) As Long
'Ensures that 100% is the max amplitude that can be entered
Private Sub amp_Change()
    If amp.Text > 100 Then
        amp.Text = 100
    End If
End Sub

'Changes the amplitude of the amplitude signal

Private Sub AmpScroll_Change()
    amp.Text = AmpScroll.Value
    Wav_Stop
    CreateWaveForm (CONTINUOUS)
    Wav_PlayLoop byteSound
End Sub

'--------------------------------------------------------------------------------
' Stop the current playing wave.
'--------------------------------------------------------------------------------
Private Sub cmdStop_Click()
    Wav_Stop
    cmdStop.Enabled = False
    StartPulses.Enabled = True
End Sub


Private Sub Form_Load()
    Dim i As Integer
    'Initialize Time Increment Amounts for each time
    Increment(0) = 0.001    'Pulse Duration
    Increment(4) = 0.1      'Repeat Interval - determines how often the entire waveform shown will be repeated in continuous
trigger mode
    Increment(1) = 0.001    'Trigger to Pulse
    Increment(2) = 0.001    '1st to 2nd Pulse (in tetanic mode, a.k.a 1st pulse to train)
    'Applies only to Tetanic Waveform Type
    Increment(3) = 0.001    'Train to Last Pulse
    Increment(5) = 0.01     'Tetanic Duration - time of the pulse train from 1st rising edge to last falling edge
    Increment(6) = 0.001    'Tetanic Interval - spacing of pulses within the train

    'Initialize the OldTimeValue array with current Time values
    'and the scroll bar values with the current Time values represented in the # of corresponding increments
    For i = 0 To 6
        OldTimeValue(i) = CSng(Time(i).Text)
        OldTimeScroll(i) = CInt(CSng(Time(i).Text) / Increment(i))
        TimeScroll(i).Value = OldTimeScroll(i)
    Next

    AmpScroll.Value = 100

    cmdStop.Enabled = False

    Waveform.ScaleWidth = 100
    Waveform.ScaleHeight = 30

End Sub

Private Sub Form_Unload(Cancel As Integer)
    ' make sure that the wavefor has stopped playing if it hasn't
    Call Wav_Stop
    Cancel = 0
End Sub

Private Sub Cont_Click()
    If (Cont.Value = True) Then
        StartPulses.Enabled = True
        Trigger1pulse.Enabled = False
        Tetanic.Visible = False
        WaveFormAll True
        ReClickWaveFormType
    End If
End Sub


Private Sub Pulse_Click()
    If (Pulse.Value = True) Then
        Trigger1pulse.Enabled = True
        StartPulses.Enabled = False
        WaveFormAll True
        Time(4).Enabled = False
        TimeScroll(4).Enabled = False
        TimeLabel(4).Enabled = False
        'Tetanic.Visible = True
        ReClickWaveFormType
    End If
End Sub

Private Sub OnePulse_Click()
    If (OnePulse.Value = True) Then
        AllChangeableTimes (False)
        WaveFormType = ONE_PULSE_TYPE
        Call DrawWaveForm
    End If
End Sub

Private Sub StartPulses_Click()
    StartPulses.Enabled = False
    CreateWaveForm (CONTINUOUS)
    cmdStop.Enabled = True
    Wav_PlayLoop byteSound
    Wav_PlayLoop byteSound
End Sub

Private Sub TimeScroll_Change(Index As Integer)
    Dim ScrollChange As Single
    Dim TimeValue As Single

    TimeValue = CSng(Time(Index).Text)
    ScrollChange = Round(TimeScroll(Index).Value - OldTimeScroll(Index), 3)

    If (Not (OldTimeValue(Index) = TimeValue)) Then 'Rare case of the time being manually entered
                                                    'And then changing it's value using the l/r scroll arrows
        If (ScrollChange < 0) Then
            Time(Index).Text = Round(TimeValue / 2, 3)
            OldTimeValue(Index) = Round(OldTimeValue(Index) / 2, 3)
        ElseIf (ScrollChange > 0) Then
            Time(Index).Text = Round(TimeValue * 2, 3)
            OldTimeValue(Index) = Round(OldTimeValue(Index) * 2, 3)
        End If
    Else  'The usual case
        If (ScrollChange < 0) Then
```

```
            OldTimeValue(Index) = Round(OldTimeValue(Index) / 2, 3)
            Time(Index).Text = OldTimeValue(Index)
        ElseIf (ScrollChange > 0) Then
            OldTimeValue(Index) = Round(OldTimeValue(Index) * 2, 3)
            Time(Index).Text = OldTimeValue(Index)
        End If
    End If

    OldTimeScroll(Index) = TimeScroll(Index).Value

End Sub

Private Sub Trigger1pulse_Click()
    CreateWaveForm (once)
    Wav_Play byteSound
End Sub

Private Sub TwoPulse_Click()
    If (TwoPulse.Value = True) Then
        AllChangeableTimes (False)
        Time(2).Enabled = True
        TimeScroll(2).Enabled = True
        TimeLabel(2).Enabled = True
        WaveFormType = TWO_PULSE_TYPE
        Call DrawWaveForm
    End If
End Sub

Private Sub Tetanic_Click()
    If (Tetanic.Value = True) Then
        AllChangeableTimes (True)
        WaveFormType = TETANIC_TYPE
        Call DrawWaveForm
    End If
End Sub

Private Sub Time_Validate(Index As Integer, KeepFocus As Boolean)
   ' If the value is not a valid numeric time value, keep the focus.
   If Not IsNumeric(Time(Index).Text) Then
      KeepFocus = True
      MsgBox "Please insert a valid decimal number in the text box.", , "Invalid Number"
   End If
End Sub

'***************Element Enabling and Disabling Helper Functions*************************
Private Sub WaveFormAll(Exists As Boolean)
    Dim i As Integer
    Wavetype.Enabled = Exists
    OnePulse.Enabled = Exists
    TwoPulse.Enabled = Exists
    Tetanic.Enabled = Exists
    TetanicFreq.Enabled = Exists
    Freq.Enabled = Exists
    TIS.Enabled = Exists
    For i = 0 To 6
        Time(i).Enabled = Exists
        TimeLabel(i).Enabled = Exists
        TimeScroll(i).Enabled = Exists
    Next
End Sub

Private Sub AllChangeableTimes(Exists As Boolean)
```

```
    Dim i As Integer
    ' The Interval, Delay settings which can change between the three pulse types
    For i = 2 To 3
        Time(i).Enabled = Exists
        TimeScroll(i).Enabled = Exists
        TimeLabel(i).Enabled = Exists
    Next
    For i = 5 To 6
        Time(i).Enabled = Exists
        TimeScroll(i).Enabled = Exists
        TimeLabel(i).Enabled = Exists
    Next
    TetanicFreq.Enabled = Exists
End Sub

Private Sub ReClickWaveFormType()
    Select Case WaveFormType
        Case ONE_PULSE_TYPE
            OnePulse_Click
        Case TWO_PULSE_TYPE
            TwoPulse_Click
        Case TETANIC_TYPE
            Tetanic_Click
        Case Else
            OnePulse.SetFocus 'Default to One Pulse WaveForm Type
    End Select
End Sub

Private Sub Time_Change(Index As Integer)
    Call DrawWaveForm
End Sub

Private Sub CreateWaveForm(OutputMode As Integer)
    Dim dblFrequency As Double
    Dim intAudioMode As Integer
    Dim lngSampleRate As Long
    Dim intBits As Integer
    Dim Times() As Double
    Dim i As Integer
    Dim amplitude As Long

    If OutputMode = CONTINUOUS Then
        dblFrequency = 1 / Val(Time(4).Text)
    Else 'OutputMode is manual trigger (so calculate the appropriate length for the wave file)
        Select Case (WaveFormType)
            Case ONE_PULSE_TYPE
                dblFrequency = 1 / (Val(Time(1)) + Val(Time(0)))
            Case TWO_PULSE_TYPE
                dblFrequency = 1 / (Val(Time(1)) + Val(Time(2)) + Val(Time(0)))
            Case TETANIC_TYPE
                dblFrequency = 1 / (Val(Time(1)) + Val(Time(2)) + Val(Time(5)) - Val(Time(6)) + Val(Time(3)) + Val(Time(0)))
                Debug.Print (Val(Time(1)) + Val(Time(2)) + Val(Time(5)) - Val(Time(6)) + Val(Time(0)) + Val(Time(3)))
        End Select
    End If

    intAudioMode = 1
    lngSampleRate = RATE_88200
    intBits = 8
    amplitude = CLng((amp.Text / 100) * 110)


    'Store the appropriate times to pass to the waveform making function
    Select Case WaveFormType
        Case ONE_PULSE_TYPE
            ReDim Times(1)
            For i = 0 To 1
                Times(i) = CDbl(Time(i).Text)
            Next
        Case TWO_PULSE_TYPE
            ReDim Times(2)
            For i = 0 To 2
                Times(i) = CDbl(Time(i).Text)
            Next
        Case TETANIC_TYPE
            ReDim Times(6)
            For i = 0 To 6
                Times(i) = CDbl(Time(i).Text)
            Next
    End Select

    Wav_Stop
    Wav_BuildHeader byteSound, lngSampleRate, intBits, intAudioMode, 0.5, 0.5
    MakeBasisWave byteSound, dblFrequency, WaveFormType, Times, amplitude, CDbl(ampFreq.Text)

End Sub

Private Sub DrawWaveForm()
    Dim PulseDuration As Single
    Dim SingleTime(0 To 6) As Single
    Dim i As Integer

    Waveform.Cls

    Waveform.CurrentX = 0
    Waveform.CurrentY = 15 '(0, 0) is in the upper left corner

    For i = 0 To 6
        SingleTime(i) = CSng(Time(i).Text)
    Next

    PulseDuration = SingleTime(0)

    'Draw the appropriate waveform on the bottom of the form
    Select Case WaveFormType
        Case ONE_PULSE_TYPE
            Waveform.ScaleWidth = SingleTime(0) + SingleTime(1)

            Waveform.Line Step(0, 0)-Step(SingleTime(1), 0), RGB(&HFF, &HE0, &HC0)
            Call DrawPulse(PulseDuration)
        Case TWO_PULSE_TYPE
            Waveform.ScaleWidth = SingleTime(0) + SingleTime(1) + SingleTime(2)

            Waveform.Line Step(0, 0)-Step(SingleTime(1), 0), RGB(&HFF, &HE0, &HC0)
            Call DrawPulse(PulseDuration)
            Waveform.Line Step(-PulseDuration, 0)-Step(SingleTime(2), 0), RGB(&HFF, &HFF, &HC0)
            Call DrawPulse(PulseDuration)

        Case TETANIC_TYPE
            Waveform.ScaleWidth = SingleTime(1) + SingleTime(2) + SingleTime(5) - SingleTime(6) + SingleTime(3) +
SingleTime(0)

            Waveform.Line Step(0, 0)-Step(SingleTime(1), 0), RGB(&HFF, &HE0, &HC0)
            Call DrawPulse(PulseDuration)
            Waveform.Line Step(-PulseDuration, 0)-Step(SingleTime(2), 0), RGB(&HFF, &HFF, &HC0)
            Waveform.Line (SingleTime(1) + SingleTime(2), 10)-(SingleTime(1) + SingleTime(2) + SingleTime(5), 10),
RGB(&HFF, &HC0, &H80)
            Waveform.CurrentX = SingleTime(1) + SingleTime(2)
            Waveform.CurrentY = 15

            Call RiseTime(0)
            Waveform.Line Step(0, 0)-Step(PulseDuration, 0), RGB(&HC0, &HFF, &HC0)
            Call FallTime(0)
            For i = 1 To CLng(SingleTime(5) / SingleTime(6)) - 1
                Waveform.Line Step(0, 0)-Step((SingleTime(6) - PulseDuration), 0), RGB(&HC0, &HFF, &HC0)
                Call RiseTime(0)
                Waveform.Line Step(0, 0)-Step(PulseDuration, 0), RGB(&HC0, &HFF, &HC0)
                Call FallTime(0)
            Next
            Waveform.Line Step(-PulseDuration, 0)-Step(SingleTime(3), 0), RGB(&HFF, &H80, &H80)
            Call DrawPulse(PulseDuration)

    End Select

    Waveform.CurrentX = 0
    Waveform.CurrentY = 25
    Waveform.Print "Time = "; Str(Round(Waveform.ScaleWidth, 6))

End Sub
Private Sub DrawPulse(ByVal PulseDuration As Single)
    Call RiseTime(0)
    Waveform.Line Step(0, 0)-Step(PulseDuration, 0), RGB(&HFF, &HC0, &HC0)
    Call FallTime(0)
End Sub

Private Sub RiseTime(Rise As Integer)
    Waveform.Line Step(0, 0)-Step(Rise, -10), RGB(128, 128, 128)
End Sub

Private Sub FallTime(Fall As Integer)
    Waveform.Line Step(0, 0)-Step(Fall, 10), RGB(128, 128, 128)
End Sub

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

'--------------------------------------------------------------------------------
' modWAV - WAV File Routines
'
' Originally Written By: David M.Hitchner
'           k5dmh@bellsouth.net
'           k5dmh@arrl.net
' Modified by Giles Peng
'
' This VB module is a collection of routines to create a play WAV
' format files.
'--------------------------------------------------------------------------------
Option Explicit

'--------------------------------------------------------------------------------
' Wave File Format
'--------------------------------------------------------------------------------
' RIFF Chunk   ( 12 bytes)
' 00 00 - 03  "RIFF"
' 04 04 - 07  Total Length to Follow  (Length of File - 8)
' 08 08 - 11  "WAVE"
'
```

```
' FORMAT Chunk ( 24 bytes )
' 0C 12 - 15  "fmt_"
' 10 16 - 19  Length of FORMAT Chunk  Always 0x10
' 14 20 - 21  Audio Format            Always 0x01
' 16 22 - 23  Channels                1 = Mono, 2 = Stereo
' 18 24 - 27  Sample Rate             In Hertz
' 1C 28 - 31  Bytes per Second        Sample Rate * Channels * Bits per Sample / 8
' 20 32 - 33  Bytes per Sample        Channels * Bits per Sample / 8
'                                 1 = 8 bit Mono
'                                 2 = 8 bit Stereo or 16 bit Mono
'                                 4 = 16 bit Stereo
' 22 34 - 35  Bits per Sample
'
' DATA Chunk
' 24 36 - 39  "data"
' 28 40 - 43  Length of Data          Samples * Channels * Bits per Sample / 8
' 2C 44 - End Data Samples
'           8 Bit = 0 to 255              unsigned bytes
'          16 Bit = -32,768 to 32,767    2's-complement signed integers
'-------------------------------------------------------------------------------------

Public Const MODE_MONO = 0      ' Mono
Public Const MODE_LR = 1        ' Stereo L+R
Public Const MODE_L = 2         ' Stereo L
Public Const MODE_R = 3         ' Stereo R

Public Const RATE_8000 = 8000
Public Const RATE_11025 = 11025
Public Const RATE_22050 = 22050
Public Const RATE_32000 = 32000
Public Const RATE_44100 = 44100
Public Const RATE_88200 = 88200
Public Const RATE_48000 = 48000
Public Const RATE_88000 = 88000
Public Const RATE_96000 = 96000

Public Const BITS_8 = 8
Public Const BITS_16 = 16

Public Type SINEWAVE
   dblFrequency As Double
   dblDataSlice As Double
   dblAmplitudeL As Double
   dblAmplitudeR As Double
End Type

Private PI As Double
Private intBits As Integer
Private lngSampleRate As Long
Private intSampleBytes As Integer
Private intAudioMode As Integer
Private dblFrequency As Double
Private dblVolumeL As Double
Private dblVolumeR As Double
Private intAudioWidth As Integer
Public amp As Long

'CONSTANTS
Private Const ONE_PULSE_TYPE = 1
Private Const TWO_PULSE_TYPE = 2
Private Const TETANIC_TYPE = 3
Private Const SND_ALIAS = &H10000
Private Const SND_ALIAS_ID = &H110000
Private Const SND_ALIAS_START = 0
Private Const SND_APPLICATION = &H80
Private Const SND_ASYNC = &H1
Private Const SND_FILENAME = &H20000
Private Const SND_LOOP = &H8
Private Const SND_MEMORY = &H4
Private Const SND_NODEFAULT = &H2
Private Const SND_NOSTOP = &H10
Private Const SND_NOWAIT = &H2000
Private Const SND_PURGE = &H40
Private Const SND_RESERVED = &HFF000000
Private Const SND_RESOURCE = &H40004
Private Const SND_SYNC = &H0
Private Const SND_TYPE_MASK = &H170007
Private Const SND_VALID = &H1F
Private Const SND_VALIDFLAGS = &H17201F

Private Declare Function PlaySoundFile Lib "winmm.dll" Alias "PlaySoundA" _
   (ByVal lpszName As String, ByVal hModule As Long, ByVal dwFlags As Long) As Long
Private Declare Function PlaySoundMemory Lib "winmm.dll" Alias "PlaySoundA" _
   (ptrMemory As Any, ByVal hModule As Long, ByVal dwFlags As Long) As Long
'--------------------------------------------------------------------------------
' Wav_Play - Continuously plays the wav file from memory.
'--------------------------------------------------------------------------------
Public Function Wav_PlayLoop(WavArray() As Byte) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundMemory(WavArray(0), ByVal 0&, SND_MEMORY Or SND_APPLICATION Or _
      SND_ASYNC Or SND_LOOP Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_PlayLoop = False
   Else
      Wav_PlayLoop = True
   End If
End Function

'--------------------------------------------------------------------------------
' Wav_PlayFileLoop - Continuously plays the wav file from a file.
'--------------------------------------------------------------------------------
Public Function Wav_PlayFileLoop(FileName As String) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundFile(FileName, 0, SND_FILENAME Or SND_APPLICATION Or _
      SND_ASYNC Or SND_LOOP Or SND_NODEFAULT)

   If lngStatus = 0 Then
      Wav_PlayFileLoop = False
   Else
      Wav_PlayFileLoop = True
   End If
End Function

'--------------------------------------------------------------------------------
' Wav_Play - Plays the wav file from memory.
'--------------------------------------------------------------------------------
Public Function Wav_Play(WavArray() As Byte) As Boolean
   Dim lngStatus As Long

   lngStatus = PlaySoundMemory(WavArray(0), 0, SND_MEMORY Or SND_APPLICATION Or _
        SND_SYNC Or SND_NODEFAULT)

    If lngStatus = 0 Then
        Wav_Play = False
    Else
        Wav_Play = True
    End If
End Function

'--------------------------------------------------------------------------------
' Wav_Play - Plays the wav file from a file.
'--------------------------------------------------------------------------------
Public Function Wav_PlayFile(FileName As String) As Boolean
    Dim lngStatus As Long

    lngStatus = PlaySoundFile(FileName, 0, SND_FILENAME Or SND_APPLICATION Or _
        SND_SYNC Or SND_NODEFAULT)

    If lngStatus = 0 Then
        Wav_PlayFile = False
    Else
        Wav_PlayFile = True
    End If
End Function
'--------------------------------------------------------------------------------
' Wav_BuildHeader - Builds the WAV file header based on the sample rate, resolution,
'                 audio mode.  Also sets the volume level for other routines.
'--------------------------------------------------------------------------------
Public Sub Wav_BuildHeader(WavArray() As Byte, SampleRate As Long, _
    Resolution As Integer, AudioMode As Integer, VolumeL As Double, VolumeR As Double)
    Dim lngBytesASec As Long

    PI = 4# * Atn(1#)

    ' Save parameters.
    lngSampleRate = SampleRate
    intBits = Resolution
    intAudioMode = AudioMode
    dblVolumeL = VolumeL
    dblVolumeR = VolumeR

    ReDim WavArray(0 To 43)

    '----------------------------------------------------------------------------
    ' Fixed Data
    '----------------------------------------------------------------------------
    WavArray(0) = 82   ' R
    WavArray(1) = 73   ' I
    WavArray(2) = 70   ' F
    WavArray(3) = 70   ' F
    WavArray(8) = 87   ' W
    WavArray(9) = 65   ' A
    WavArray(10) = 86  ' V
    WavArray(11) = 69  ' E
    WavArray(12) = 102 ' f
    WavArray(13) = 109 ' m
    WavArray(14) = 116 ' t
    WavArray(15) = 32  ' .
    WavArray(16) = 16  ' Length of Format Chunk
    WavArray(17) = 0   ' Length of Format Chunk
    WavArray(18) = 0   ' Length of Format Chunk
    WavArray(19) = 0   ' Length of Format Chunk
    WavArray(20) = 1   ' Audio Format
    WavArray(21) = 0   ' Audio Format
    WavArray(36) = 100 ' d
    WavArray(37) = 97  ' a
    WavArray(38) = 116 ' t
    WavArray(39) = 97  ' a

    '-----------------------------------------------------------------------------
    ' Bytes 22 - 23  Channels   1 = Mono, 2 = Stereo
    '-----------------------------------------------------------------------------
    Select Case intAudioMode
        Case MODE_MONO:
            WavArray(22) = 1
            WavArray(23) = 0
            intAudioWidth = 1
        Case MODE_LR:
            WavArray(22) = 2
            WavArray(23) = 0
            intAudioWidth = 2
        Case MODE_L:
            WavArray(22) = 2
            WavArray(23) = 0
            intAudioWidth = 2
        Case MODE_R:
            WavArray(22) = 2
            WavArray(23) = 0
            intAudioWidth = 2
    End Select

    '-----------------------------------------------------------------------------
    ' 24 - 27  Sample Rate            In Hertz
    '-----------------------------------------------------------------------------
    WavArray(24) = ExtractByte(lngSampleRate, 0)
    WavArray(25) = ExtractByte(lngSampleRate, 1)
    WavArray(26) = ExtractByte(lngSampleRate, 2)
    WavArray(27) = ExtractByte(lngSampleRate, 3)

    '-----------------------------------------------------------------------------
    ' Bytes 34 - 35  Bits per Sample
    '-----------------------------------------------------------------------------
    Select Case intBits
        Case 8:
            WavArray(34) = 8
            WavArray(35) = 0
            intSampleBytes = 1
        Case 16:
            WavArray(34) = 16
            WavArray(35) = 0
            intSampleBytes = 2
    End Select

    '-----------------------------------------------------------------------------
    ' Bytes 28 - 31  Bytes per Second   Sample Rate * Channels * Bits per Sample / 8
    '-----------------------------------------------------------------------------
    lngBytesASec = lngSampleRate * intAudioWidth * intSampleBytes

    WavArray(28) = ExtractByte(lngBytesASec, 0)
    WavArray(29) = ExtractByte(lngBytesASec, 1)
    WavArray(30) = ExtractByte(lngBytesASec, 2)
    WavArray(31) = ExtractByte(lngBytesASec, 3)

    '-----------------------------------------------------------------------------
```

```
    ' Bytes 32 - 33 Bytes per Sample     Channels * Bits per Sample / 8
    '                                    1 = 8 bit Mono
    '                                    2 = 8 bit Stereo or 16 bit Mono
    '                                    4 = 16 bit Stereo
    '-------------------------------------------------------------------------------
    If (intAudioMode = MODE_MONO) And (intBits = 8) Then
        WavArray(32) = 1
        WavArray(33) = 0
    End If

    If ((intAudioMode = MODE_LR) Or (intAudioMode = MODE_L) Or _
        (intAudioMode = MODE_R)) And (intBits = 8) Then
        WavArray(32) = 2
        WavArray(33) = 0
    End If

    If (intAudioMode = MODE_MONO) And (intBits = 16) Then
        WavArray(32) = 2
        WavArray(33) = 0
    End If

    If ((intAudioMode = MODE_LR) Or (intAudioMode = MODE_L) Or _
        (intAudioMode = MODE_R)) And (intBits = 16) Then
        WavArray(32) = 4
        WavArray(33) = 0
    End If

End Sub

'-------------------------------------------------------------------------------
' Wav_WriteToFile - Writes the wav file to disk.
'-------------------------------------------------------------------------------
Public Function Wav_WriteToFile(FileName As String, WavArray() As Byte) As Long
    Static blnInitialized As Boolean

    Dim fd As Integer
    Dim i As Long, lngFileSize As Long

    Wav_WriteToFile = 0
    On Error GoTo Routine_Error

    fd = FreeFile
    Open FileName For Binary As #fd

    lngFileSize = UBound(WavArray)
    For i = 0 To lngFileSize
        Put #fd, , WavArray(i)
    Next

Routine_Exit:
    On Error Resume Next
    Close #fd
    Exit Function

Routine_Error:
    Wav_WriteToFile = Err.Number
    Resume Routine_Exit
End Function

'-------------------------------------------------------------------------------
' MakeBasisWave - Builds the basis waveform that can be played in a continuous loop.
'-------------------------------------------------------------------------------
```

```
Public Sub MakeBasisWave(WavArray() As Byte, Frequency As Double, Wavetype As Integer, Times() As Double, amplitude
As Long, ampFreq As Double)

    Dim i As Long
    Dim lngLimit As Long
    Dim lngDataL As Long, lngDataR As Long
    Dim dblDataPt As Double, blnPositive As Boolean
    Dim intCycles As Integer, intCycleCount As Integer
    Dim lngFileSize As Long
    Dim lngSamples As Long
    Dim lngDataSize As Long
    Dim lngDataPoints() As Long
    Dim Time1_Samples As Long, NumHigh_Samples As Long
    Dim NumTetanicSamples As Long
    Dim dblAmpliSignalTime As Double
    Dim ampSamples As Long
    Dim tempInitI As Long

    Dim dblWaveTime As Double
    Dim dblSampleTime As Double

    dblFrequency = Frequency

    dblWaveTime = 1 / dblFrequency
    'dblTotalTime = dblWaveTime * intCycles
    dblSampleTime = 1 / CDbl(lngSampleRate)
    'dblDataSlice = (2 * PI) / (dblWaveTime / dblSampleTime)
    lngSamples = CLng(dblWaveTime / dblSampleTime)
    ReDim lngDataPoints(lngSamples - 1)
    'Debug.Print "lngDataPoints = "; lngSamples

    'Right Channel Signal
    dblAmpliSignalTime = 1 / ampFreq
    ampSamples = CLng(dblAmpliSignalTime / (dblSampleTime))
    amp = amplitude

    'Create the waveform based on the times array and the WaveType
    Select Case Wavetype
        Case ONE_PULSE_TYPE
            Time1_Samples = CLng(Times(1) / dblSampleTime)
            'Debug.Print "Time1_Samples"; Time1_Samples
            For i = 0 To Time1_Samples - 1
                lngDataPoints(i) = 0
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i
            For i = i To lngSamples - 1
                lngDataPoints(i) = 0
            Next
        Case TWO_PULSE_TYPE
            Time1_Samples = CLng(Times(1) / dblSampleTime)
            'Debug.Print "Time1_Samples"; Time1_Samples
            For i = 0 To Time1_Samples - 1
                lngDataPoints(i) = 0
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i

            For i = i + 1 To CLng((Times(1) + Times(2)) / dblSampleTime) - 1
                lngDataPoints(i) = 0
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i

            For i = i + 1 To lngSamples - 1
                lngDataPoints(i) = 0
            Next
        Case TETANIC_TYPE
            Debug.Print (Times(1) + Times(2) + Times(5) - Times(6) + Times(0) + Times(3))
            NumHigh_Samples = CLng(Times(0) / dblSampleTime)
            NumTetanicSamples = CLng(Times(6) / dblSampleTime)
            For i = 0 To CLng(Times(1) / dblSampleTime) - 1
                lngDataPoints(i) = 0
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i

            For i = i + 1 To CLng((Times(1) + Times(2)) / dblSampleTime)
                lngDataPoints(i) = 0
            Next

            tempInitI = i + 1
            For i = i + 1 To i + CLng((Times(5) - Times(6)) / dblSampleTime) ' Tetanic duration
                If (((i + NumTetanicSamples - tempInitI) Mod NumTetanicSamples) <= NumHigh_Samples) Then
                'Pulse is High: Generate Oscillation under Pulse Envelope
                    GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i
                Else
                    lngDataPoints(i) = 0   'Pulse is low
                End If
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i

            For i = i + 1 To (i + CLng(Times(3) - Times(0) / dblSampleTime))
                lngDataPoints(i) = 0
            Next
            'Generate Oscillation under Pulse Envelope
            GenerateOsc lngDataPoints, CLng(Times(0) / dblSampleTime), dblSampleTime, i

            For i = i + 1 To lngSamples - 1
                lngDataPoints(i) = 0
            Next
    End Select

    '-------------------------------------------------------------------------------
    ' Bytes 40 - 43  Length of Data   Samples * Channels * Bits per Sample / 8
    '-------------------------------------------------------------------------------
    lngDataSize = lngSamples * intAudioWidth * (intBits / 8)
    ReDim Preserve WavArray(0 To 43 + lngDataSize)

    WavArray(40) = ExtractByte(lngDataSize, 0)
    WavArray(41) = ExtractByte(lngDataSize, 1)
    WavArray(42) = ExtractByte(lngDataSize, 2)
    WavArray(43) = ExtractByte(lngDataSize, 3)

    '-------------------------------------------------------------------------------
    ' Bytes 04 - 07  Total Length to Follow  (Length of File - 8)
    '-------------------------------------------------------------------------------
    lngFileSize = lngDataSize + 36

    WavArray(4) = ExtractByte(lngFileSize, 0)
    WavArray(5) = ExtractByte(lngFileSize, 1)
    WavArray(6) = ExtractByte(lngFileSize, 2)
    WavArray(7) = ExtractByte(lngFileSize, 3)
```

```
'-------------------------------------------------------------------------------
' Bytes 44 - End   Data Samples
'-------------------------------------------------------------------------------

If intBits = 8 Then
   lngLimit = 127

Else
   lngLimit = 0 '32767
End If

For i = 0 To lngSamples - 1

   If intBits = 8 Then
      '-----------------------------------------------------------------------
      ' 8 Bit Data
      '-----------------------------------------------------------------------
      ' Calculate data point.
      dblDataPt = lngDataPoints(i) * lngLimit
      lngDataL = Int(dblDataPt * dblVolumeL) + lngLimit
      'lngDataR = lngDataL

      ' Extract amplitude channel here
      If (i Mod ampSamples < (ampSamples / 2)) Then
         lngDataR = amplitude
      Else
         lngDataR = 0
      End If

      ' Place data point in wave tile.
      If intAudioMode = MODE_MONO Then _
         WavArray(i + 44) = ExtractByte(lngDataL, 0)

      If intAudioMode = MODE_LR Then       'L+R stereo
         WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
         WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
      End If

      If intAudioMode = MODE_L Then       ' L only stereo
         WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
         WavArray((2 * i) + 45) = 0
      End If

      If intAudioMode = MODE_R Then       ' R only stereo
         WavArray((2 * i) + 44) = 0
         WavArray((2 * i) + 45) = ExtractByte(lngDataR, 0)
      End If

   Else

      '-----------------------------------------------------------------------
      ' 16 Bit Data - THIS IS CURRENTLY UNUSED, BUT COULD EASILY BE INCORPORATED
      '              IN CASE A MORE ACCURATE OUTPUT SIGNAL IS REQUIRED.
      '-----------------------------------------------------------------------
      ' Calculate data point.
      dblDataPt = lngDataPoints(i) * lngLimit
      lngDataL = Int(dblDataPt) ' * dblVolumeL)
      lngDataR = Int(dblDataPt) ' * dblVolumeR)

      ' Place data point in wave tile.
      If intAudioMode = MODE_MONO Then
```

```
            WavArray((2 * i) + 44) = ExtractByte(lngDataL, 0)
            WavArray((2 * i) + 45) = ExtractByte(lngDataL, 1)
        End If

        If intAudioMode = MODE_LR Then
            WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
            WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
            WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
            WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
        End If

        If intAudioMode = MODE_L Then
            WavArray((4 * i) + 44) = ExtractByte(lngDataL, 0)
            WavArray((4 * i) + 45) = ExtractByte(lngDataL, 1)
            WavArray((4 * i) + 46) = 0
            WavArray((4 * i) + 47) = 0
        End If

        If intAudioMode = MODE_R Then
            WavArray((4 * i) + 44) = 0
            WavArray((4 * i) + 45) = 0
            WavArray((4 * i) + 46) = ExtractByte(lngDataR, 0)
            WavArray((4 * i) + 47) = ExtractByte(lngDataR, 1)
        End If

    End If

  Next
End Sub


'-------------------------------------------------------------------------------------
' ExtractByte - Extracts the high or low byte from a short (16 bit) VB integer.
'
'   intWord     - VB Integer from which to extract byte.
'   intByte     - Returned high or low byte.
'   intPosition - |                    Word                  |
'               | Byte = 3 | Byte = 2 | Byte = 1 | Byte = 0 |
'-------------------------------------------------------------------------------------
Private Function ExtractByte(lngWord As Long, intPosition As Integer) As Byte
  Dim lngTemp As Long
  Dim intByte As Integer

  If intPosition = 3 Then
    ' Byte 2
    lngTemp = lngWord

    ' Mask off byte and shift right 24 bits.
    '   Mask  -> 2130706432 = &H7F000000
    '   Shift -> Divide by 16777216
    lngTemp = (lngTemp And 2130706432) / 16777216

    ' Cast back to integer.
    intByte = lngTemp

  ElseIf intPosition = 2 Then
    ' Byte 2
    lngTemp = lngWord

    ' Mask off byte and shift right 16 bits.
    '   Mask  -> 16711680 = &HFF0000
    '   Shift -> Divide by 65536
```

```
        lngTemp = (lngTemp And 16711680) / 65536

        ' Cast back to integer.
        intByte = lngTemp

    ElseIf intPosition = 1 Then
        ' Byte 1
        lngTemp = lngWord

        ' Mask off high byte and shift right 8 bits.
        '   Mask  -> 65290 = &HFF00
        '   Shift -> Divide by 256
        lngTemp = (lngTemp And 65290) / 256

        ' Cast back to integer.
        intByte = lngTemp
    Else
        ' Byte 0
        intByte = lngWord And &HFF
    End If

    ExtractByte = intByte
End Function

'-----------------------------------------------------------------------------------
' Wav_Stop - Stop the currently playing wav.
'-----------------------------------------------------------------------------------
Public Sub Wav_Stop()
    Dim lngStatus As Long

    lngStatus = PlaySoundMemory(ByVal 0&, ByVal 0&, SND_PURGE Or SND_NODEFAULT)
End Sub

'Number of samples that can fit into the envelope of this pulse
Public Sub GenerateOsc(lngDataPoints() As Long, numSamples As Long, dblSampleTime As Double, i As Long)
    Dim modFreq As Double
    Dim modInterval As Double
    Dim intervalSamples As Long
    Dim x As Long

    modFreq = 1000
    modInterval = 1 / modFreq
    intervalSamples = CLng(modInterval / dblSampleTime)
    x = i

    For i = i + 1 To i + numSamples - 1
        If (i Mod intervalSamples < (intervalSamples / 2)) Then
            lngDataPoints(i) = 10
       Else
            lngDataPoints(i) = -10
        End If
    Next

End Sub
```

# APPENDIX V: SOFTWARE SETUP GUIDE

Before beginning to use the SimpleScope Software, the following configuration must be performed:

- The SimpleScope hardware should be attached to the sound card. Line-in is for input (what is to be seen on the virtual oscilloscope window) and line-out (or stereo-out) for the output (stimulator pulses).

- If the recording line on the sound card can be changed (i.e. could be microphone or line-in), set the recording line to line-in. This can be adjusted by a widget in the control panel.

- The master volume and the wave volume should be adjusted to max with the volume control provided by Windows.

- Additionally ensure that both the master volume and the wave volume is unmuted.

- The master volume and wave volume should be balanced between left and right channels (50/50)

- Mute the line-in volume.

## Oscilloscope Operation

- Vertical voltage scale can be set from .001 to 5 volts.
- Horizontal time scale can be set from .001 to 30 seconds.
- Trigger Source:
    - **Manual Trigger:** User initiates capture of one window's worth of data.
    - **Start:** Acquire data continuously.
    - **Trigger Check Box:** Trigger when a certain voltage specified by the user is reached on the input channel.
- Sample Frequency can be set to 11.025 kHz, 22.050 kHz, or 44.1 kHz. The default is 11.025 kHz.
- Use mouse clicks in the virtual oscilloscope window to measure absolute voltage and time differences.
- Export the data to an Excel type file with the 2 columns being time and input channel voltage. The number of points in the data set exported is the sample frequency multiplied by the per window time scale.

**Stimulator Operation**

- Started by clicking the stimulator button in the oscilloscope window
- Modes:
    - **Pulse**: One pulse of the user-inputted waveform.
    - **Continuous**: Continuous repetition of the specified waveform with a user-selected repeat interval.
- The waveforms can be set to single pulse, two pulse or tetanic. A tetanic signal consists of a uniquely spaced initial pulse, a train of evenly spaced pulses, and then a uniquely spaced final pulse.
- All pulse delays and durations are set with a set of edit fields and left-right arrows. The stimulating waveform is drawn below the controls. The color of each time control corresponds to the color of a line on the drawing.
    - In One Pulse mode you can set "Pulse Duration" and "Trigger to Pulse" duration.
    - In Two Pulse mode you can set "Pulse Duration", "Trigger to Pulse" duration, and "1st to 2nd Pulse" duration.
    - In Tetanic mode you can set "Pulse Duration", "Trigger to Pulse" duration, "1st to 2nd Pulse" duration, and "Train to Last Pulse" duration. Additionally, you can set the "Tetanic duration", which is the length of all the pulses in the train, and the "Tetanic interval", which is the spacing of pulses within the train.
    - In all modes, you can set the repeat interval which determines how often the entire stimulus will be produced, if the trigger mode is continuous.
- The amplitude percentage for the resulting output can also be adjusted using the slider from 0% to 100%.