# A PORTABLE SYSTEM FOR

# DETECTING INFRASOUND

By

STEVEN BERGREN

Bachelor of Science in Electrical Engineering

Oklahoma State University

Stillwater, OK

2001

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2018

A PORTABLE SYSTEM FOR
DETECTING INFRASOUND

Thesis Approved:

Dr. Carl Latino

Thesis Adviser

Dr. George Scheets

Dr. Keith Teague

ACKNOWLEDGEMENTS

I would not have been able to complete this thesis without the help and encouragement of Dr. Carl Latino. Though I doubted my ability to complete this, his advice and guidance along the way allowed me to finish. Thanks also go to Dr. Charles Abramson. The idea for this device started with him and I only hope I was able start a work that others may be able to build upon, and that it may possibly be used to save the lives of zookeepers.

I would like to thank the other members of the committee, and the entire faculty of the school of Electrical Engineering, as their ideas to solve problems that occurred during the course of this project were extremely helpful.

Several members of the Parallax Forums, especially Jon McPhalen, were vital to my understanding of the Propeller microcontroller and its intricacies. I would not have been able to make this device do anything at all if it wasn't for their insight and suggestions for solving the many issues I had during the course of this effort.

Lastly, my thanks go to my wife, Cindy, who has had to bear with me for the past year and a half, and the long days and nights I was cloistered away in my office accomplishing class work or this thesis. Her patience, encouragement, and belief in me is remarkable.



Acknowledgements reflect the views of the author and are not endorsed by committee members or Oklahoma State University.

Name: STEVEN BERGREN

Date of Degree: MAY, 2018

Title of Study: A PORTABLE SYSTEM FOR DETECTING INFRASOUND

Major Field: ELECTRICAL ENGINEERING

Abstract: The purpose of this project is to create a device to detect infrasound communication from elephants in order to inform handlers of possible impending aggressive behavior. Elephants often communicate using infrasound which is low-frequency sound below the threshold of human hearing. Elephants may be trying to communicate with zookeepers but the handlers are unable to hear their call. Knowing that an elephant is communicating may give handlers time to move to safety. A device is designed and prototyped that is capable of monitoring an input signal for infrasound and produces a warning alarm for handlers. This device can also record audio for long periods of time to a digital storage device. It can be utilized for other areas of study with some modification. The device is low-cost so it would be able to be procured more easily and in higher quantities than more expensive laboratory monitoring equipment. This device could also be used as an educational and research device for students studying animal behavior in the field and laboratory. Infrasound is not limited to only elephants, but hippopotamuses, rhinoceroses and giraffes also communicate with infrasound. Environmental infrasound from sources such as wind turbines, sonic booms, explosions, tornadoes, and earthquakes can also be monitored. Test results showed that the device accurately recorded low-frequency input signals. The device also was able to detect infrasound frequencies and triggered an alarm.

TABLE OF CONTENTS

Chapter Page

Chapter Page

# LIST OF TABLES

LIST OF FIGURES

# CHAPTER I

## INTRODUCTION

The purpose of this project is to create a device to detect infrasound from elephants in order to inform handlers of possible impending aggressive behavior. Detection of elephant infrasonic communication with real-time feedback will aid zookeepers in remaining safe during elephant-handling activities. A small and inexpensive system could be employed by elephant keepers as a safety device to warn them of aggressive elephant actions. This device could also be used as an educational and research device for students studying animal behavior in the field and laboratory.

Elephants were chosen as the focus animal for this device for several reasons. Safety is the primary concern of this project, and elephant interactions can be dangerous. Elephants can turn aggressive seemingly without warning, so a device that can make interactions safer is a desired product. In 2010, a Toledo, Ohio zookeeper was critically injured when the elephant he had worked with almost daily for seven years suddenly turned on him with seemingly no warning (1). In 2013, a Springfield, Missouri keeper was killed while feeding an elephant he worked with for over 23 years. (2) The latter zookeeper was a friend of Oklahoma State professor Charles Abramson and was one of the inspirations of this project.

Secondly, elephants, as a popular exhibit animal, are readily available at zoos throughout the world for observation, including the Tulsa and Oklahoma City zoos. This availability also makes them easier to study compared to other animals that are rare and secluded. As a popular attraction to children and many other zoo patrons, increasing the safety of their care ensures a more valuable experience for the observers.

Elephants in the wild are a concern for villages in Asia and Africa. A group of elephants will raid a village with little warning. A system that detects infrasound can provide early warning to humans in the area, giving them a chance to move to safety before the elephants attack.

Lastly, in addition to safety, students can learn more about elephant communication by incorporating the device into the elephant enclosure and to other zoological exhibits in which animals emit infrasound.

**Elephant Communication**

Elephants utilize many senses for communication – hearing, vision, smell, and touch – as do all social mammals. Many of their sounds are audible to human hearing, especially those produced through their trunks. However, elephants also communicate through very-low-frequency sounds, most often referred to as rumbles. These sounds fall into a range known as infrasound, which is a range of sound below the threshold of audible human hearing, generally defined as below 30 Hz. Therefore, any communication an elephant attempts toward humans in this manner would be unheard. The ability of an elephant keeper to be warned when infrasonic communication is taking place could assist the keeper in remaining safe should the elephant emit warning signals.

Elephants are extremely large and powerful animals, and though human-elephant interactions are often safe, accidents do still occur, often with disastrous results for the humans involved**.** Though declining since World War II, elephants are still used in Asia as beasts of burden. Their chores include clearing trees and performing other heavy labor tasks. This requires very close contact with their human handlers, putting the handlers in dangerous situations daily (3). In zoos throughout the world, there are multiple elephant and human accidents yearly. Many of these accidents occur between elephants and keepers that have interacted together without incident for many years, only to have one poor interaction that results in the severe injury or death of a keeper. The deaths mentioned previously are just two of the approximately 500 that occur each year when humans interact with elephants (4). Any system that may warn keepers about impending aggressive behavior would be an improvement and possibly save lives.

### Thesis Objective

The objective of this thesis is three-fold. First: to design and construct a low-cost device that is able to monitor infrasound frequencies and warn zookeepers of possibly aggressive behavior by the elephants in their care. Second: to have the ability to record for extended periods infrasound data that can be analyzed in the future. Lastly: to have a device that can serve as an educational tool and a basis for further studies as research equipment is expensive.

The primary function of the device would be to constantly monitor frequencies in the infrasound range and produce an audible alarm to signal zookeepers that the elephant is communicating in infrasound and that they should move to a safer place. This warning system could potentially save lives; therefore, it is the most important objective. However, the information that could be gathered through this device could assist in future elephant communication research, so other functions will be included to assist in that effort. The device should also be able to record the sounds to a storage device for later analysis.

### Design Considerations

As a safety device, two requirements need to be established in order for it to be effective. First, the device needs to be small, light-weight, and battery-operated; the availability of electrical outlets and the needed placement of the device would vary between different elephant enclosures, zoos, and facilities, as would the voltages provided in different countries. Though data acquisition systems are readily available for laptop computers, their size and power consumption does not lead them to be used because they do not meet the portability and low-cost objectives. Secondly, it needs to be easy to use with little to no setup required to ensure reliable function during each and every use. A complicated system that requires adjustment or multiple button presses each time the system is powered on could accidently be configured incorrectly which could result in the alarm function disabled and unable to properly monitor and warn zookeepers.

In addition to safety, another consideration is cost. First, the proposed device will often be used in outdoor animal enclosures. The device could easily be damaged by rain and other environmental causes. Additionally, the elephants themselves could damage the device, no matter how hardened a casing it could possibly have. By using a low-cost device, the device could be replaced easily without much expense, especially compared to a laptop-based system. Secondly, due to the size of enclosures, having multiple warning devices in several locations would increase the ability to properly detect the infrasound signals and ensure that a device was near enough to a keeper to hear the alarm. Lastly, producing the device at low cost would allow researchers to utilize these in poorer areas of the world that cannot afford expensive equipment. As infrasound travels across long distances, villagers could be warned well in advance of an elephant raid, giving them time to seek shelter. Given the right sensor, there is no need for the elephant to be extremely close to trigger the device.

Though designed as a safety device, the system could be easily modified to be used for a variety of applications beyond that of elephant-human interaction safety. Animal behavioral scientists could utilize the device~~s~~ in many scenarios to record animal sound in nature as well as in laboratory experiments. Commercial recording systems are often expensive. These are out of the budget of public schools, and university systems may have some recording devices, but this device could potentially allow many more students to have access to recording devices that could be used to study animal sounds and behaviors.

Chapter 2 explores the literature involving infrasound measurement and recording, tools for controlling animal behavior experimentation, and elephant calls. This review will explain concepts used in the design of the device. Chapter 3 covers the methodology and design of the device, including the sensor, controller, interface, and software. Chapter 4 reports the test results of the device. Chapter 5 concludes with future work and offers suggestions on how the capabilities of the device can be extended.

# CHAPTER II

## REVIEW OF LITERATURE

### Elephant Listening Project

The study of elephants and their behaviors are a common occurrence in their native lands of Africa and Asia. The Elephant Listening Project (ELP), part of The Cornell Laboratory of Ornithology at Cornell University, has been studying the sounds of elephant calls for over 30 years. Katharine Payne and her team worked mostly with African elephants in western Africa. The fundamental frequencies of the calls they recorded were between 15-35 Hz with pressure levels up to 117 dB at a range of 1 meter from the source (4). These low-frequency sounds can travel over long distances because neither the air nor the ground will attenuate the signals significantly. These long-distance communication methods explain elephant behavior which was not previously understood. Elephant groups separated by several kilometers would simultaneously perform similar movements for several hours. They could also, still separated by kilometers, turn and head directly towards each other. Additionally, though males and females spend most of their time apart from each other, they could find each other easily during mating seasons. The researchers tested these theories by playing back the recorded sounds while observing elephant groups. Elephants seen 2 kilometers away from the sound source would freeze and/or turn and walk towards the source and often respond to these prerecorded calls the majority of the time.

The Elephant Listening Project has recorded over 700,000 hours of sounds in the areas of west-central Africa (5). Self-contained recording devices were designed to record for up to 6 months at a time. One particular study analyzed 5 years of data – from 2007 to 2012. The goal of the analysis was to determine if elephant calls could be detected electronically rather than manually. Manual detection consisted of having a person listen to find the elephant calls on recordings. The recordings used for data analysis were at 16-bit resolution at either a 2000 or 4000 Hz sampling rate. The rumbles they found ranged from 8-180 Hz and lasted from 2 to 8 seconds. The fundamental frequency of the rumble was between 8 and 34 Hz with a median of 15 Hz.

Members of the ELP listened to the recordings and tagged true elephant calls in order to score whether the algorithm correctly identified a true elephant call. The algorithm would divide the recordings into 100 millisecond frames to then score with the likelihood of containing an elephant rumble. All audio was converted to a 2 kHz sampling rate and a short-time Fourier Transform was produced over 1024 points. These spectrographs were split into 20 second segments. Though elephant rumbles are highly varied, they typically have constant power at a specific frequency throughout a rumble. Looking at these frequency and power characteristics, 46 elements were identified in a vector for each time frame.

They found that the first harmonic, 2·F_0, which ranged from 20-24 Hz, was more prominent than the fundamental frequency. They were unsure as to the reason for this but suspected the microphones' inability to capture the lower frequencies without loss was the cause. These harmonic features were a factor in their scoring. Image filters were applied to the 20 second frames to look at the horizontal characteristics of the signal to determine how well they matched with elephant calls, as well as to reject noise that came from the hard drive used. Utilizing machine learning algorithms, they were able to detect and classify the sounds found on the recordings. Across about 4000 hours of recordings, the algorithm correctly identified an elephant call between 80-90% of the time and a false positive rate of less than 9%. When the threshold was lowered to reach a 90% detection rate, the false positive rate exceeded 40%. This analysis was performed on an 8 core Dell laptop with 16 GB of RAM, and 24 hours of recordings were analyzed in 9 minutes.

Similar research was performed in India. The researchers performed a comparable spectral analysis and a neural network classification of the signals, having only 21 elements in their vector. Their results were similar to the ELP project, detecting the signal 90% of the time, but having false positives 30% of the time (6).

**Low-cost Recording System**

The recordings captured by the ELP utilize autonomous recording devices placed in trees in elephant habitats in Africa. These are specialized devices with long recording time of at least 6 months; therefore, the battery and storage capabilities are great. These often come at great cost. Researchers in Sri Lanka developed a way to record elephant infrasound much more economically (7). Their goal was to develop a recording system that was easy to use and could be made at a fairly low-cost. In their country, as in much of southwest Asia, elephants are common and often used as work animals; therefore, elephant-human conflict is also a common occurrence. According to the Sri Lankan Department of Wildlife Conservation, an average of 150 elephants and 65 people die each year due to these confrontations. Some of these deaths occur from elephant raids on villages. Electric fences, built for village protection, are not able to keep the elephants out. A warning system was needed to detect elephants. Keying in on infrasound communications was the goal of their system.

Their system was based around a PC application that would record the elephant sounds. They needed to develop an analog input, an amplifier, an anti-aliasing filter, and an analog-to-digital converter. The output of the analog to digital converter was then sent to the PC that would record the digital representation of the elephant rumbles. For the analog sensor, they utilized a speaker which was mounted on a stand and directed toward the elephants at the Dehiwala zoo. The speaker was used instead of a microphone because of its greater sensitivity to lower frequencies and because there was no need for additional circuitry in order to power the device, as a condenser microphone would require. Though no specifications on the speaker are given, pictures of the device show it to be approximately 2-3 inches in diameter. The amplifier they selected was a simple op amp inverting amplifier. The resistances chosen gave the amp a fixed gain of 200 V/V. The anti-aliasing filter used was a second-order Butterworth filter with a cutoff of 100 Hz. The Butterworth was selected to keep a constant gain across the passband without ripples. For the analog to digital converter, a microcontroller was selected that had an onboard 10-bit converter. The microcontroller was then connected to the parallel port on the PC, where a software application recorded the data and provided for a simple graphical interface to allow users to see the frequency components of the recorded sounds. The sampling rate utilized was 7500 Hz.

They tested their device at a zoo and captured signals that contained infrasound characteristics of elephant calls mentioned previously. Fourier transforms of these recordings showed strong results in the 7-15 Hz range with duration of about 7 seconds. Their recording system was also tested on a diesel engine, finding the dominant frequency for it to be below 1 Hz. They showed that their system was able to record infrasound reliably, and the cost for the hardware, from speaker input to microcontroller output (excluding the PC and application) was only $35.

### The Propeller Experimental Controller

Microcontrollers are being utilized in other behavioral laboratory experiments. The Laboratory of Comparative Psychology and Behavioral Biology at Oklahoma State University experimented with the use of microcontrollers, specifically, the Parallax Propeller, rather than the much more expensive laboratory experiment controllers marketed by several instrument companies. In order to engage students in live animal studies, an easier way to get more students involved was needed. Computer simulations were not providing the learning experience that instructors desired, and the expense, size, and setup of the commercial experiment controllers limited the number of people that could be involved in the live animal experiments (8).

The laboratory found the Propeller easy to setup and program, and it could be easily connected to various sensors and outputs, such as switches, thermometers, lights/light sensors, and audio devices. The Propeller is also capable of generating video signals to see real-time data. They found the Propeller Spin programming language to be the easiest to use of any they had experienced. The Propeller is a multi-core microcontroller, with 8 independent cores. The cores, called "cogs", allowed several unrelated experiments to be run simultaneously without any interference between them. The ability to utilize already-written modules available at the Parallax Object Exchange, as well as the help from the community forums also at the Parallax website, allowed users to write their programs faster and accomplished tasks that may be beyond their programming skill set.

The portability of the device was also noted. Only a few inches in each dimension, and easily powered by USB, the device can be taken and setup nearly anywhere quite simply. This microcontroller has been used for automatic control and measurements that would be laborious if not impossible to accomplish manually in a reliable fashion. The laboratory went so far as to write experiment controller software that they have made available for other behavior laboratories to use.

# CHAPTER III

# METHODOLOGY

The design of the device began with identifying components capable of meeting the requirements of a portable, infrasound-detecting, and low-cost device. A microcontroller is required to manage the operations of the device. A signal input device, such as a microphone, is required to capture the infrasound frequencies. That input signal would then need to be amplified and filtered. An analog to digital converter (ADC) is needed to convert the analog signal into a digital representation for the microcontroller to work with. An external storage device is needed in order to store the recordings. A digital to analog converter (DAC) is required in order to take the sampled signal and play it back for the user. Finally, a display and interface control buttons and switches are required to operate the device.

## Signal Input

Typical microphones used to record voices or musical instruments have poor low frequency response. Common microphones, such as the Shure SM57 and SM58 (Shure Americas, Niles, IL), have a low frequency response that resides at about 0 dB near 100 Hz and continually drops, with frequency response charts showing a -10 dB loss at 50 Hz (9). The charts end there, as these microphones are intended for human hearing ranges, so there is neither a need to show lower frequencies, nor a need to test microphones to respond to such low frequencies.

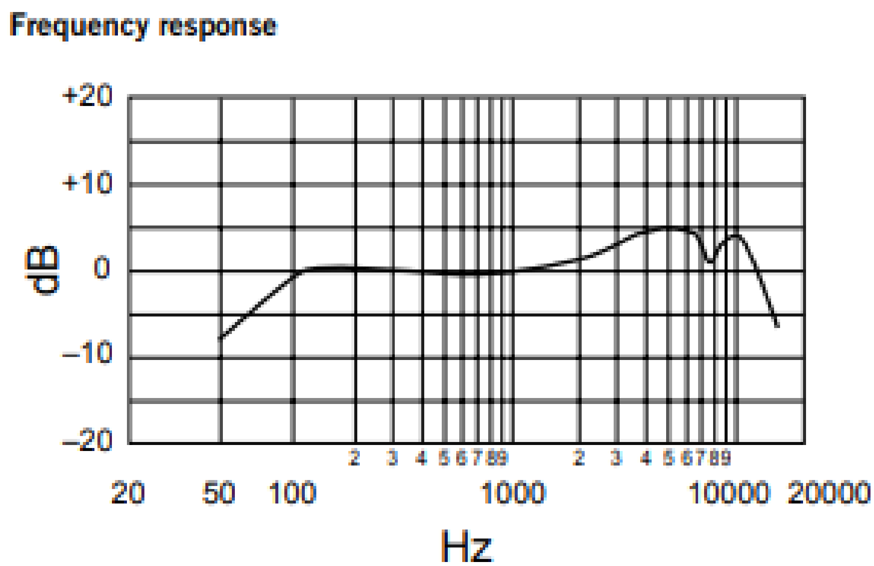



**Figure 1- Shure SM58 Frequency Response (9)**

Extrapolating from that decline, however, shows that infrasound will not be detected well by these microphones. Clearly a different kind of microphone or sensor will be required in order to detect the infrasound. Additionally, these kinds of vocal microphones cost about $100, which detracts from the low-cost intent of the device (10).

Specialty microphones designed for infrasound are manufactured. The intended applications for the PCB 378A07 (PCB Piezotronics, Depew, NY) are environmental testing, including wind turbine, sonic boom, explosion, tornado, and earthquake monitoring (11). They are also used to test machinery noise levels, such as industrial equipment and engines, and for the GRAS 41AC-2 (GRAS Sound & Vibration, Holte, Denmark), aircraft and community noise (12). The frequency response for these microphones is very low, some losing only 3 dB at 0.1 Hz. The characteristics of the frequency response are ideal for the purposes of the infrasound recording device.

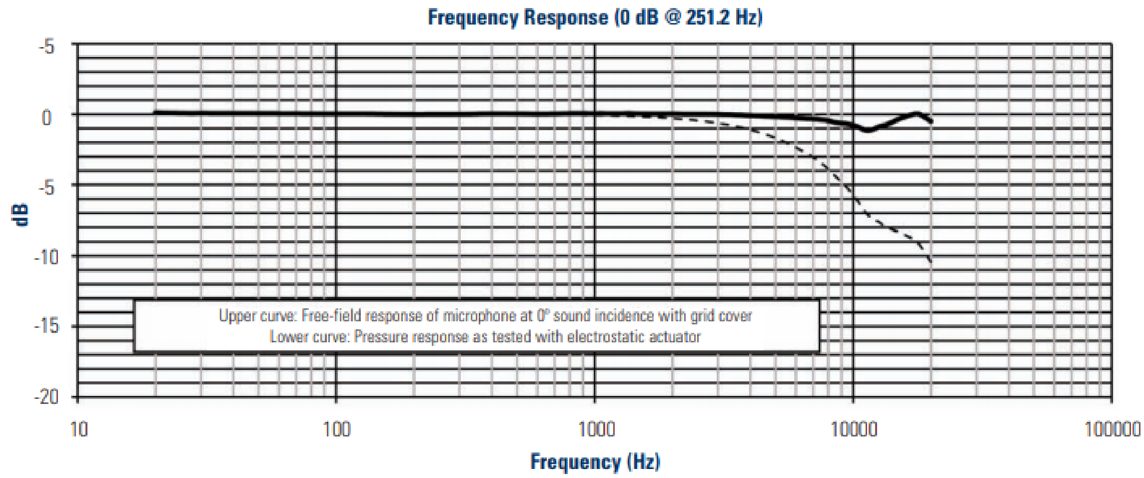



**Figure 2 – PCB Piezotronics 378A07 Frequency Response (13)**

However, several other characteristics prevent these types of microphones from being used. First, these microphones are often of a free-field design. Free-field microphones work best when pointed directly at the sound source. Sounds coming from other angles are greatly affected. In the operation of this recording device, it is doubtful that a person would be constantly moving the microphone to aim it directly at the elephants, as the most likely need of this device is during movement procedures. Though the effects on this angle of incidence are less for low-frequency signals, this is a factor to be considered. Secondly, these microphones are designed to work with preamplifiers that require at least a 28V power supply (11). That need violates the requirement that this device is small and battery operated. Lastly, these microphones typically cost greater than $1000 each, and often much more. This also violates the goal of being a low-cost device.

A larger diaphragm in a microphone is better suited to detecting lower frequencies. De Silva and De Zoyza designed their own microphone using a speaker. (7) The underlying concepts between microphones and speakers are the same – sound pressure is converted to electrical signals, and vice-versa. A diaphragm with a coil of wire attached is suspended in a frame. A large magnet is placed behind this coil to induce a magnetic field. When the diaphragm vibrates, the movement of the coils in the magnetic field induces a current in the wire proportional to the vibrations. This concept is feasible and would be sufficient. It does, however, suffer from the same problem as any microphone in that it needs to be directed toward the sound source in order to be most effective. A second problem that plagues microphones and speakers is wind noise. Wind can cause turbulence at the microphone diaphragm which results in signal generation at very low frequencies (14). These wind-generated signals can be difficult to separate from the actual infrasound intended to be recorded.

Another sensor of a different type is to be considered. Though it is not known with certainty how infrasound signals are detected by elephants, is it suspected that they can detect the signals through their feet (15). This implies that the pressure waves travel through the ground in a seismic fashion. A seismometer is a relatively simple device that can detect vibrations in the earth and convert them to electrical signals. One of the simplest seismographs is very much like the microphone and speaker described previously, but instead of a diaphragm reacting to sound pressure, an inertial mass suspended via a spring reacts to the vibrations. The mass itself is often a magnet, which then induces a current in a coil, effectively turning the vibrations into electrical signals. Unlike a microphone, there is no need to point the seismograph toward the source of the sound, as it is travelling up from the ground into the seismograph, just like it would through elephant's feet.

Seismometers are often used in petroleum exploration. One such item, a Mark Products L-25 (formerly Mark Products, now Sercel, Nantes, France) was available to use for this project. It is a cylindrical device approximately 1.5" in diameter and 2.25" tall, weighing less than a pound.

**Figure 3 - Mark Products L-25**

No specifications were available for this device, but it is very similar to more recent and available devices. This type of small seismometer is called a geophone. Geophones are used on the ground surface, whereas seismometers are typically much larger and heavier and are buried in order to better sense the very small vibrations of distant earthquakes. Geophones have a spike attached to them which is driven into the ground in order to hold them in place and transfer the ground vibrations to the device.

Due to the lack of specifications, some experimentation was required in order to understand its capabilities. The device was connected to an oscilloscope and then placed on a table. The table was tapped by hand and the measurements recorded. The recorded waveform of an average-strength tap on the table shows that the signal peaks at 50mv and with a frequency of about 29 Hz.

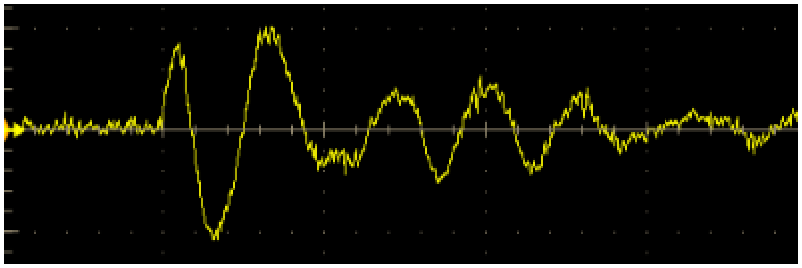


**Figure 4 - Signal from L-25 due to tap on table (100 mV/div, 50 ms/div)**

The oscillation is due to the nature of the device, with the spring stretching and contracting, dampening to a small amplitude in about 6 periods. Very faint writing on the side of the device was found that looks like it says 28-30 Hz, so this must mean it is the natural frequency of the mass and spring. Specifications for other geophones that are commercially available, such as the Sercel JF-20DX (Sercel, Nantes, France), show that a commonly-used natural frequency for geophones is 28Hz, so it is likely that the L-25 has a similar design (16). Sercel is also formerly known as Mark Products, further increasing that probability. A specification for that device shows that peak output occurs at a frequency of 28 Hz.

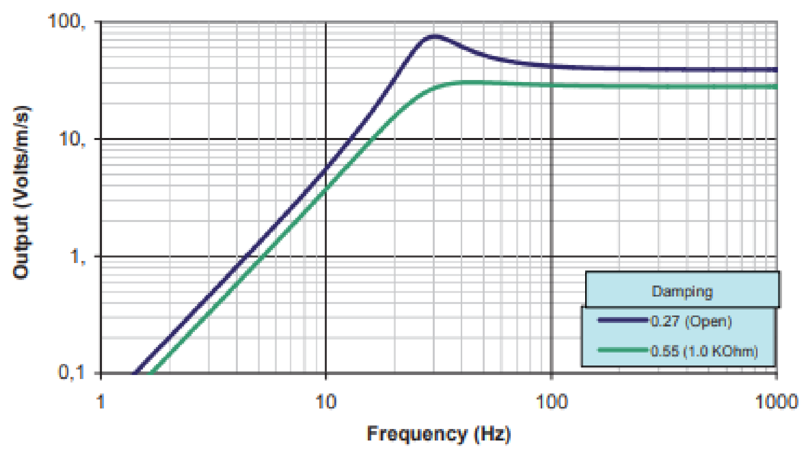



**Figure 5 - Sercel JF-20DX 28 Hz Geophone Amplitude Response (16)**

Additional testing consisted of patterns of taps on a table to simulate specific frequencies. A 4 Hz table tap (with a stronger tap at each 1-second mark, or every 4 taps, for points of reference) was recorded. The same pattern of dampening oscillations was observed.

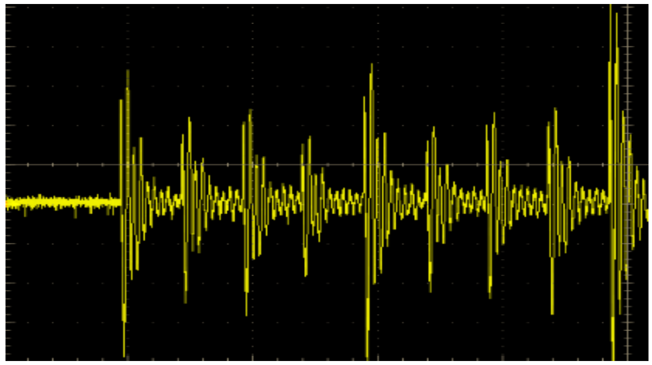


**Figure 6 - L-25 response to 4 Hz table taps (10 mV/div, 500 ms/div)**

The same 29 Hz oscillations are seen clearly when the 4 Hz signal is zoomed in. It takes until the 5th oscillation for the amplitude to be significantly less than the first oscillation. A method of filtering out the 29 Hz signal could be devised that would leave only the 4 Hz signal.

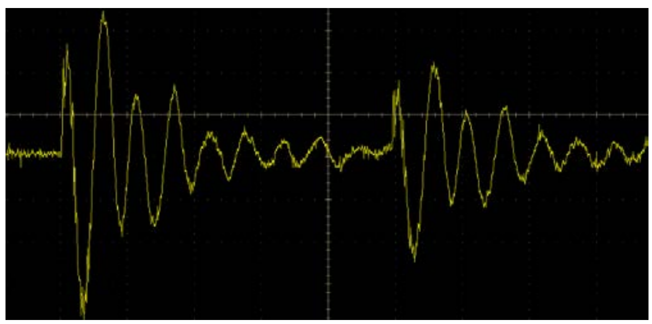


**Figure 7 - L-25 response to 4 Hz table taps (10 mV/div, 50 ms/div**

When the tapping rate was increased to about 10 Hz, the signals from the taps were unable to be distinguished from each other. As was seen on the 4 Hz signal, it took until about the 5th oscillation for the amplitude of the signal to be significantly lower than the first oscillation. At 29 Hz, that comes out to be about 116 ms. Therefore, any signal with a period smaller than about 116 ms will be hard to distinguish from the natural oscillations of the spring mechanism.

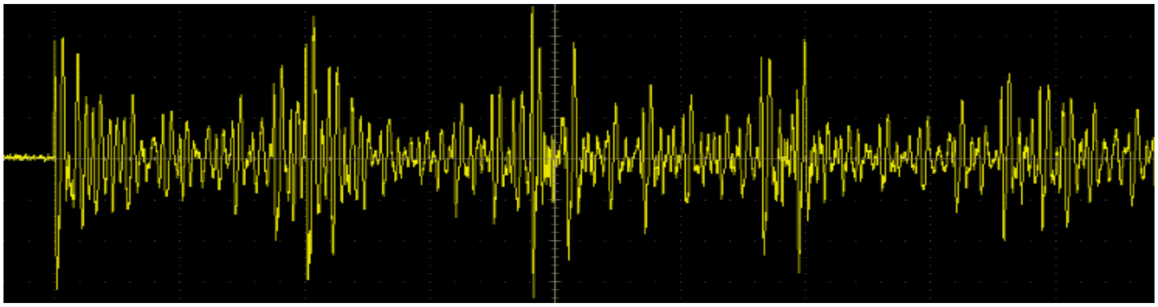


**Figure 8 - L-25 response to 10 Hz table taps (30 mV/div, 500 ms/div)**

At 10 Hz, the period is 100 ms, and the signals all seemed to mesh together with no way to differentiate between true signal and oscillation. In figure 8, stronger taps were generated at 1 Hz for points of reference. Though those 1 Hz signals are able to be identified, there is no clear 10 Hz signal to be seen under it.

A zoomed-in figure of the same oscilloscope capture shows the difficulty in separating the signals. There is a 1 second difference between the two peaks with the highest amplitude in the figure. Though there are some stronger peaks in between that may be part of the 10 Hz signal, it is just too intertwined to differentiate. The oscillations of the mass have not diminished enough by the time another tap was registered for the new signal to stand out from the oscillations.

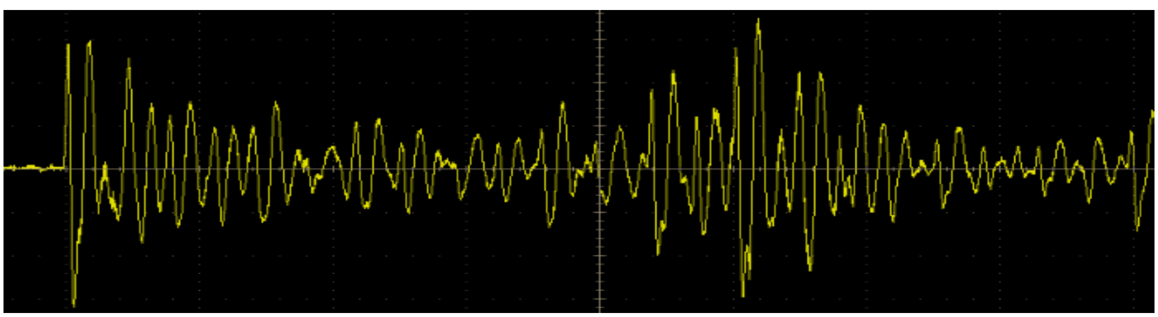


**Figure 9 - L-25 response to 10 Hz table taps (30 mV/div, 300 ms/div)**

Any infrasound that was picked up by this geophone would be hard to identify specifically by frequency and amplitude for signals above 10 Hz. Therefore, this sensor would not be a good choice for anyone wishing to capture the true audio signal of an elephant call.

However, the sensor may still be effective for simply capturing whether there is any infrasound at all in a location. For the purposes of sensing infrasound as a safety device, this geophone type of sensor may be sufficient for triggering an alarm at the presence of an elephant call.

### Microcontroller

The primary component of this device is the microcontroller. Selecting the microcontroller required estimating the microcontroller speed needed, the core bit size, and the amount of IO pins required. As this device is intended to be low cost and small, a microcontroller would be the most likely solution to meet those requirements. The clock frequency required is tied to the highest sampling speed desired. The goal of this device is to sample infrasound, which is less than 30 Hz, so a slow microcontroller would be sufficient. However, the additional objective of making this device usable in many applications requires a much higher sampling rate, such as the CD-quality sampling rate of 44,100 Hz. Each sample will take dozens of clocks for the operations, not to mention the additional processes of storing the data to an external storage device and any other device functions occurring simultaneously, such as human interface monitoring. A "safe" clock rate would be at least 1000 clocks per sample, therefore a microcontroller clock greater than 44MHz would be the best bet in order to meet the required sampling speed.

The core size of the microcontroller would need to be sufficient to handle the bit depth of the analog to digital converter. An 8-bit core could only handle a -128/127 value from the ADC which would only provide a signal to quantization noise ratio of 48.16 dB. A greater bit depth is desired, so the core of the microcontroller should be more than 8 bits.

The microcontroller also requires enough IO pins to interface with all peripherals. Connections need to be made to at least the ADC, DAC, display, interface buttons, and external storage device. This does not even include any spare pins that should be made available to future uses or expansion. These requirements allow for no less than 16 IO pins be available on the microcontroller.

There is a wide range of microcontrollers that meets these requirements. Any one of them would be sufficient for the purposes of this device. An additional factor that would help the choice of microcontroller is the programming language. As these devices are intended to be used by individuals who are not necessarily frequent programmers, utilizing a programming language and interface that is easier to understand would make the device more usable by a wider range of individuals. Though some functions may reside in assembly-level instructions, the portions of the code that may need to be changed by future users in order to meet their application should be in a higher-level language that is much easier to access.

To that end, the Parallax Propeller microcontroller meets the desired requirements. In addition to meeting all of the technical requirements stated above, the company has its own programming language called Spin made just for the Propeller. There is also the Propeller Assembly (PASM) language which allows for more detailed control of the microcontroller. This microcontroller has already been used in the Comparative Psychology and Behavioral Biology Laboratory at Oklahoma State University. Dr. Charles Abramson suggested this microcontroller due to its ease of use in his laboratory. Psychology students are not focused on programming yet have successfully used the Propeller to control experiments in their laboratories. Utilizing a Propeller microcontroller for this device increases the likelihood of maintainability and easier modification by individuals who are not strong programmers.

The Propeller is a multiprocessor microcontroller. It contains 8 individual processors called cogs. The starting and stopping of the cogs is controlled by a hub which manages all operations. The cogs process in parallel, but access to main RAM takes place in a round-robin fashion, allotting each cog exclusive access on a regular cycle. (17 p. 21) All cogs have access to the IO pins and the system clock at the same time. These characteristics allow the Propeller to be run without the need for interrupts to control event-handling. Different cogs can be free to run their dedicated function, such as ADC processing, writing to external storage, display, etc. The processor speed can be multiplied to 80 MHz using a phase-locked loop method.

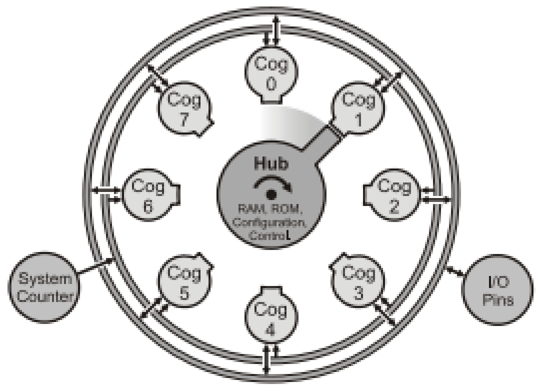



**Figure 10 - Propeller Hub and Cog Interaction (17)**

The Propeller can be purchased by itself, but also comes in a variety of development boards that contain voltage regulation, USB connections for programming and terminal access, and other peripherals. Determining which development board to use, if any, required selecting what other components were required in order to meet the device requirements.

### Amplifier

No matter the sensor type, the input signal would need to be amplified and/or buffered. The signal coming from the input sensor will be AC in nature and in the millivolt range. This signal will need to be adjusted to fit within the range of the analog to digital converter. This range is approximately 0-5 volts. Many amplifying circuits are capable of meeting this requirement. A non-inverting op amp circuit was chosen to accomplish this. The op amp is also powered with 0 and 5 volts. The input signal is sent through a coupling capacitor and biased to the midpoint of the range (2.5VDC.) The circuit also incorporates a potentiometer to allow for a variable gain. This allows the device to accommodate a wide range of input sensors. The design for this amplifier is modified from the MPLAB Starter Kit from Microchip Technology (18 p. 43). The gain of the amplifier is 2-229 V/V. The end result of this amplifier is a signal with a maximum allowable amplitude of 2.5 VAC centered at 2.5 VDC.

An anti-aliasing filter is required prior to sending the signal to the ADC. The filter is a low-pass filter than has a cutoff less than half of the sampling frequency. Because the signals in question are less than 30 Hz, the Nyquist frequency could be as low as 60 Hz. However, to prevent any loss of signal, the cutoff frequency should be a bit higher than the highest frequency desired to be captured so that is it not attenuated. To minimize ripple in the passband, a Butterworth filter was selected. The frequency roll off of a Butterworth is slow. Therefore, the sampling frequency would need to be higher than double the frequency where magnitude is deemed filtered sufficiently to ensure there is no aliasing.

**Analog to Digital Converter**

Similar to the point made in the microcontroller section, an 8-bit ADC would not produce a signal with the precision easily capable with common components, nor with the fidelity desired in order to capture a signal faithful to the original. Those 8 bits could only produce 256 points of quantization. Therefore, a higher-order ADC is desired.

ADCs typically come in the range of 8 to 24 bits of resolution. As 8 bits has been deemed too few bits, 16 bits is then considered. Sixteen bits provides for 65,536 points of quantization. If the maximum range of this device of 5 volts, the least significant bit would represent around 0.076 millivolts. This is much smaller than the device would need to differentiate, so that depth of resolution is not needed. Midway between the two, a 12-bit ADC, provides 4096 points of quantization, which results in 1.2 millivolts at the least significant bit. This number is more in line with the capabilities of the rest of the device.

### Interface

The device requires an interface in order to control the functions to be executed. A display is required for visual feedback, as well as buttons for control. The number of top-level functions is fairly small, e.g. Monitor, Record, Playback; therefore, a tiered menu-style control setup suffices to completely control the device. Four momentary switches for control are all that is needed (Back, Up, Down, and Select.) This limits some kinds of input, such as character and number input, but keeps the interface simple.

The display chosen is a Parallax 2x16 character LCD (#27922) with backlight and a piezospeaker (19). This display is small enough to meet the small design requirement of the device yet provides enough information to properly use the device. Being a Parallax product, it also interfaces well with the Propeller, so controlling the display is simple. The speaker is loud enough that it can be used to create an alarm to warn zookeepers to infrasound detection.

A 5-LED block is used to show the audio level coming into the device (PN SSA-LXB525-G2YAID, Lumex.) It is similar to many used in audio meter devices. Three LEDs are green, yellow, amber, and red. The input into the ADC is analyzed and the LEDs are lit depending on the maximum value detected over a period of time. This feedback is needed to help the use in setting the gain level on the potentiometer as there is no audio feedback available to inform the user as to the audio level coming into the device. Though the maximum value for a 12-bit DAC is 4095 bits, the input signal needs to be biased at the midpoint to avoid the incoming audio signal from being cutoff with negative-peaked signals. Therefore, when deciding how to light the LEDs, the minimum value is 2047 and the maximum is 4095. The LEDs are programmed to light at 20%, 40%, 60%, 80%, and 90% of maximum value, respectively from green to red.

The device would need to be battery-powered. Optionally, an external DC adapter could also be considered if a power source were available where the device is to be placed.

### External Storage

An external storage device must be included in order to capture the recorded audio. This memory should be non-volatile so that the memory is retained when powered down. It also should be low power, and does not need to be excessively large, as these audio files will rarely exceed several megabytes. Raw data audio file types, such as the WAV format, perform no compression on the audio data, and so each sample is stored exactly is it is measured. The WAV format has 40 bytes of header information; the remaining data is all audio samples. Therefore, the file size is: 40 bytes + # of bytes per sample × # of samples (20). Assuming a 32-bit microcontroller, the largest file that could be handled would be $2^{31}$ bytes, or 2GB. A 2-byte data word sampled at 10,000 Hz could be as long as 107,374 seconds, which is 29 hours and 49 minutes. Elephant calls last less than 10 seconds, and the sampling rate could be much lower due to the low frequencies being recorded. Theoretically, a single file could record continuously for nearly two weeks at 900 Hz. There is clearly no need for large external storage devices. 2GB of storage would be more than enough.

### Digital to Analog Converter

A Digital to Analog Converter (DAC) was included in order to have playback of recordings. The DAC selected is a 12-bit DAC that essentially reverses the process that the ADC and recording procedures executed. The data is read from the SD card WAV files and buffers are filled, with the data being sent to the DAC at the rate specified in the WAV file header. Because the output of the DAC is in the range 0-5 VDC, the output is connected to the headphone jack via a coupling capacitor to block DC current so that only the AC component transfers to the headphones.

Because infrasound is below human hearing thresholds, a simple playback of the WAV file is not helpful if desiring to hear the recorded elephant call. The signal would need to be modified in order for the user to be able to hear any infrasound that had been recorded. A simple software solution to this problem is to provide a method for the user to increase the speed of the playback by lowering the number of clocks between inputs to the DAC. A ten-fold increase in the playback speed would allow an inaudible signal to be heard, e.g., 15 Hz output is 150 Hz.

**Software**

The Propeller Spin and PASM languages are object-based, meaning that different functions can be separated into modules whose methods can then be called from a higher-level module, the highest level of which is called the top object. Many objects for performing particular functions, such as serial terminal control, SD card file handling, etc., are available on the Parallax website in a section called the Object Exchange (obex.parallax.com.) The exchange is a community-supported, "open source" depository of code written by users and by Parallax employees. It is a relatively small collection of only about 800 objects. These objects range from simple modules as listed above, that are not stand-alone programs themselves, to complete programs that perform various functions, such as an IR Remote Decoder, a function generator, and a pulse-width modulated motor driver. These objects are programmed in Spin, PASM, and some in C. Perhaps even more helpful are the forums where comments and questions can be posted and other users and experts can answer questions and help with debugging code. During the course of this project, the members of the forum were extremely helpful in my attempts to understand the Propeller and how to use it to accomplish my goals with it.

Many of the objects available were written to function with specific development boards from Parallax. Because of that, most modules require modification in order to utilize the code for development boards of a different configuration. Sometimes the changes are as simple as changing the pin numbers for connection to a device, such as connections between the Propeller and micro SD card slot are different on different development boards. Often, however, the code from several different objects would need to be merged in a way to get the desired result. For example, a module designed to use the Propeller Board of Education to record WAV files may use the on-board microphone, send it to the Propeller and use sigma-delta analog to digital conversion, and write that to an SD card. Though much of the code could be reused, it would need to be modified to use a different input source and different type of ADC.

Most modules built in Spin have one particular purpose. Keeping them single focused allows many modules to be used together. There is always a main module that is then used to call methods, Parallax's term for a sub-procedure. These methods can be in the main module or in other modules. The diagram of software module relationships is as follows:

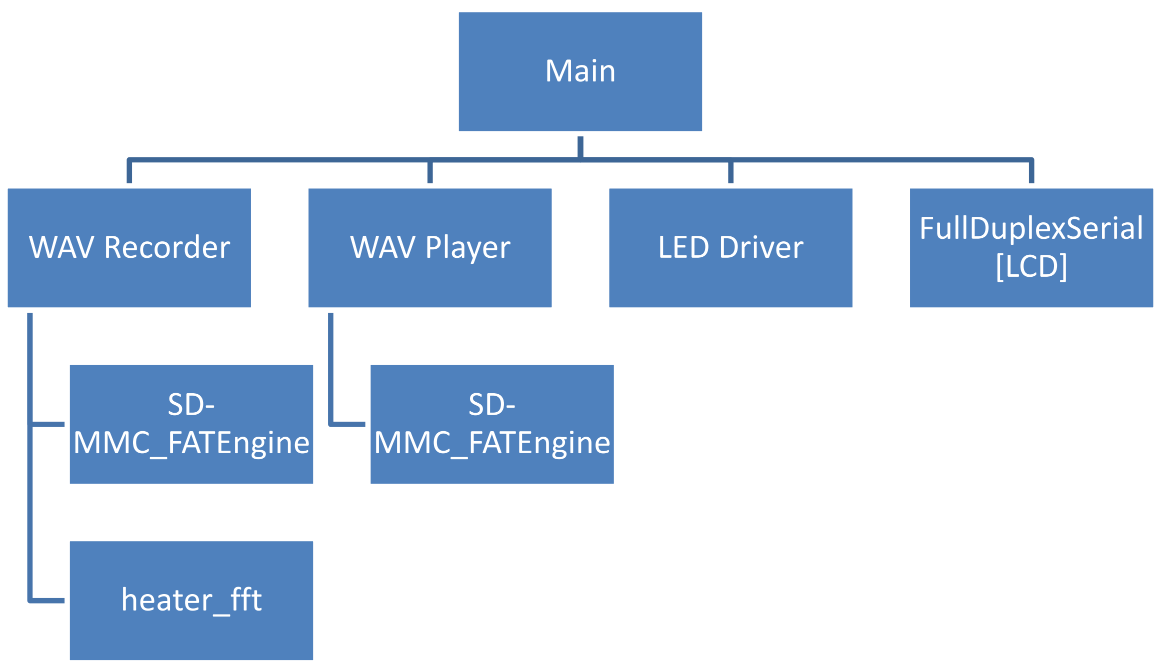



**Figure 11 - Software Module Block Diagram**

The main module sends parameters, including pointers to memory locations, that the other modules can then act on. The methods are run sequentially when called by the main module unless a new cog is created to run them. If a new cog is initialized that then calls the methods, the main module continues executing while the new cog and its methods run separately in parallel. The Main and LED Driver modules were created for this project. FullDuplexSerial is a module supplied by Parallax with their compiler. The remaining modules were acquired from the Parallax Object Exchange. WAV Recorder and WAV Player were heavily modified, however; for instance, the original code used a sigma-delta analog to digital conversion and was replaced with code for the dedicated ADC chip, among other changes. Heater_fft was not functional when first downloaded so errors had to be corrected. SD-MMC_FATEngine was used without modification except for the removal of unused methods in order to reduce module size.

**Implementation**

Based on the design considerations, the proposed device should be able to be handheld. Finding a Propeller board that incorporates some, if not all of the desired component would simplify the design. The smallest board is the Propeller Mini. Though containing the Propeller, a crystal, program storage, and voltage regulators, all connections must be soldered. A device called the Project Board is a Mini with a USB connector and an area with through holes and pads to allow for prototyping. Next is the Quickstart. In addition to the components of the Mini, it has a USB connection for programming and power, 8 programmable LEDs, and 8 resistive touch buttons. All IO pins are available via sockets. An optional Human Interface Board can connect to the Quickstart. It contains a micro SD card slot, two PS/2 ports, an infrared transceiver, and multiple audio and video output connectors. Though it has some additional components that may be helpful, it has many that are unneeded.

The Propeller ASC+ contains a micro SD slot, a 12-bit ADC, and an external DC supply power jack. This board contains many of the additional components desired with the same footprint of the Quickstart, so it is the board that was selected as the basis for this device. There are additional Propeller boards, such as the Board of Education and the Activity Board WX, but they are larger, more expensive, contain additional features not required, and are not as easily arranged and accessible for usage inside of an enclosure.

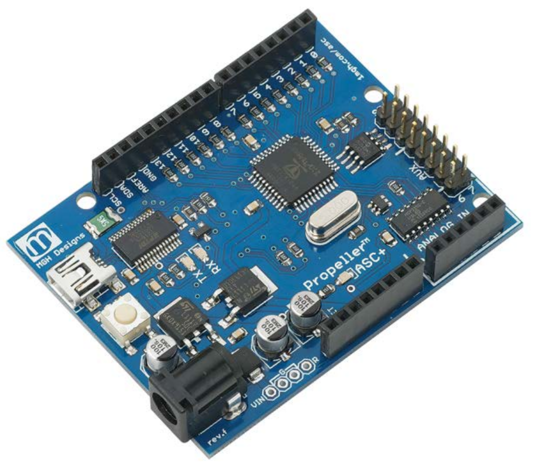


**Figure 12 - Propeller ASC+ (21)**

The geophone described in the previous chapter was chosen for further experimentation. A low frequency vibration is easier to generate than a low-frequency sound, so it will be easier to test the function of the device using the geophone as an input.

The design for the amplifier is modified from the MPLAB Starter Kit (Microchip Technology, Chandler, AZ) (18). Several changes were made to the design in order for it to meet the needs of this device. First, the MPLAB circuit had circuitry at the input that could provide power for a condenser microphone, if that was going to be used. That circuitry was removed. Secondly, the input signal was biased at 2.5VDC before going to the op amp. The same model op amp from the MPLAB circuit was used, an MCP6024, a Rail-to-Rail Input/Output, 10 MHz Op Amp. The same resistor values were selected, and the potentiometer is the same as the MPLAB circuit at 500kohms, which provides a gain from 2 to 229 V/V.

The anti-aliasing filter was designed using FilterLab 2.0 software from Microchip Technology (22). An anti-aliasing filter wizard is included in the tool that will design an operational amplifier filter given certain parameters. The tool asks the user for the cut-off frequency, the sampling frequency, the resolution in bits of the analog-to-digital converter being used, and the desired signal-to-noise ratio of the final signal. On a final summary page, the software shows the parameters, and gives the user a choice between Butterworth and Chebyshev, with the filter order and the stopband attenuation listed. Once selected, the tool presents a schematic for the designed filter and a frequency response graphic for the magnitude and phase.

Infrasound is sound less than 30 Hz, therefore no signals greater in frequency than that are desired to be recorded or detected. For the filter tool, a cut-off frequency of 40 Hz was selected. Using 40 Hz instead of 30 Hz helped to keep the magnitude at 30 Hz the same as lower frequencies, i.e. no -3dB drop at 30 Hz.

With desired signals less than 40 Hz, the sampling frequency could be extremely low, e.g. 80 Hz using the Nyquist theorem. However, the design of a filter to reduce the magnitude far enough to prevent aliasing at 80Hz, yet maintain unity magnitude at 30 Hz, is extremely difficult. This software tool could not do it, stating it would be a filter order of greater than 8, which was out of the bounds of this tool to generate. It also would be hardware intensive, requiring much more space and power than desired. Researchers in (5) and (6) utilized sampling frequencies of 1 to 2 kilohertz in their infrasound research. This higher sampling frequency allows for a lower-order anti-aliasing filter to be designed because the magnitude of the output does not have to meet the stopband threshold until 500 Hz for a 1000 Hz sampling frequency. The researchers were also interested in capturing multiple harmonics of the infrasound, resulting in a passband of at least 150 Hz. This would also drive an increase in sampling frequency. Though 40 Hz was the selected cutoff frequency for this filter, a 1 kHz sampling frequency was selected to allow for a low order filter to be designed. The Propeller ASC+ ADC is 12 bits, so there was no ability to change that parameter in the filter tool. Setting the signal-to-noise ratio parameter to -65 dB created a filter order of 3, which only needed two op amps and 3 each of resistors and capacitors.

A Butterworth filter was selected because of the smooth frequency response in the passband. The tool then output the schematic and frequency response graphics for this filter.

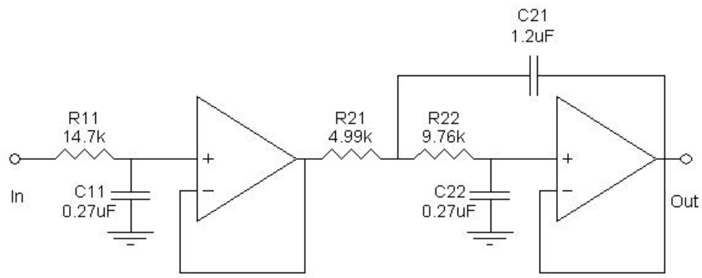



**Figure 13 - Schematic of Anti-aliasing filter using FilterLab**

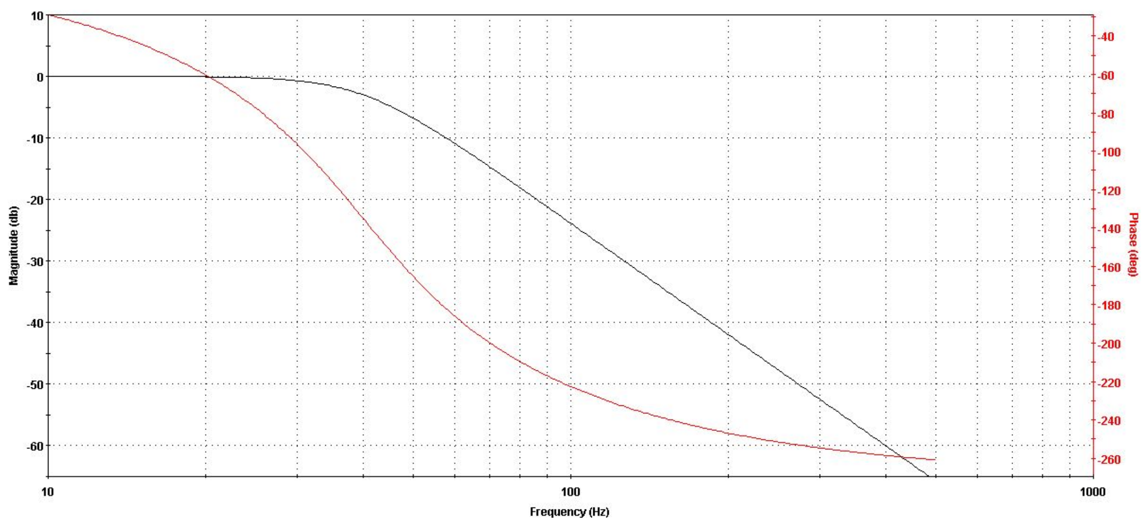



**Figure 14 - Frequency Response of Anti-aliasing filter using FilterLab**

The format for data storage on the SD card needed to be determined. Data could be written in human-readable form, such as ASCII characters. This could be the integer values that came directly from the ADC, or the calculated voltage levels based off of the ADC reference voltage. These values could be comma-separated values in a spreadsheet-style file stored to the SD card. The other option would be to store the raw data in a binary file. This could be accomplished by just writing the data as 16-bit words, or utilizing some other data format, such as a WAVE file. The human-readable versions would take more processing to turn the raw data into ASCII characters; therefore, it may not be able to be done at the speeds desired.

To test the speed of the ASCII-character writing, a program was written in Spin to time a write sequence. A new file was opened on the SD card. The current system counter value, CNT, was then stored to memory. One-hundred values were written to the SD card in ASCII integers, each separated by a new line, and the system counter was captured again. The average time to complete a write for a single value using this method was 148,241 clocks, or less than 539 samples per second. This human-readable value method of storing data is not compatible with recording audio data. The process of turning the raw ADC value into ASCII characters is much too time consuming.

The same experiment was performed, this time writing the raw data directly to the SD card. Using this method, the average number of clocks needed per sample was 435 clocks, which correlates to over 183,000 samples per second. A raw data format is the needed storage method.

The WAVE file format, a Microsoft and IBM audio file standard, is a fairly simple, uncompressed format for audio data (20). The file header contains information such as the sample rate, number of bits per sample, number of channels, and other information needed to properly describe the data in the file. Starting at byte 44, the raw data is written through until the end of the file. This file format will be the easiest to work with without requiring additional processing.

Before continuing with a WAVE recording program, it is important to understand how the Propeller's timing and cogs work. The Propeller has a 32-bit system counter (`CNT`) that increments every system clock cycle. All cogs have access to this value via the `CNT` register. Functions that need to be synchronized on different cogs can sync off of the `CNT` value. There is no need to have a cog set aside that only performs timing for the rest of the cogs. The Spin and assembly languages both have a command called `WAITCNT`(*cnt value*) which halts the processor until `CNT` equals *cnt value*. The cog will sit idle, waiting for the system counter to reach that value. Once it does, execution will resume. Command usage such as `WAITCNT(50000 + cnt)` would read the current `CNT` value, add 50000 to that number, and then wait 50000 cycles before continuing (17).

The second way to use `WAITCNT`, and more valuable to this project, is for synchronized delay timing, such as what would be needed to accurately capture audio samples at evenly spaced intervals. The command looks more like `WAITCNT(Time += 50000)`. This usage requires the variable `Time` to have been set previously. The cog waits until `Time`, but then continues execution and adds 50000 to the value of `Time`. That command would then be followed by statements, such as capturing the ADC value. It would then be looped and return to the `WAITCNT` command, where it waits for the next 50000 clock interval. This form evenly spaces each `WAITCNT` by exactly 50000 clocks, therefore keeping perfect time between samples. Care should be taken to ensure that the statements following the `WAITCNT` command do not exceed 50000 clocks themselves. If they should, then the system counter would have passed the value of `Time`, and the `WAITCNT` procedure would halt all execution on this cog until the system counter loops and returns to the value of Time in the `WAITCNT` command. With a 32-bit counter, even at the Propeller maximum speed of 80MHz, it would take 53 seconds for the counter to loop back around to the "missed" value.

Cog function is another important aspect of the function of the Propeller. The Propeller does not have an interrupt function. Each cog on a Propeller is an independent processor core, complete with its own registers and RAM. Because of this multi-core design, there is no need to stop execution of the main program in order to execute an interrupt routine. What would be the interrupt routine can just be run on another cog, leaving the main program to continue execution without interruption. This ensures more deterministic timing, as there is no stopping and starting of the main program for interrupt events. There are several ways for this "interrupt routine" to happen at the needed time. As mentioned previously, the `WAITCNT` command will wait until a specific system counter value is reached before continuing execution which works well for synchronized delay events. `WAITPEQ` halts execution until an input IO pin or set of pins matches an expected state. Execution of the rest of the code then commences. This function, as well as its inverse (`WAITPNE`), work well for halting execution until input values change, such as buttons being pressed or asynchronous inputs beginning. The input pins are checked every clock cycle, so it is a very responsive command. There is also a command, `WAITVID`, for use with the Propeller's video generation hardware. Lastly, a more brute force method is to read memory locations in Main RAM and wait for a certain condition to be met. A variable may get set in the Main program/cog, and a cog that is monitoring that variable's memory location could begin execution of code if that variable state changes. This is not quite as responsive as the previous commands, however, because of the way cogs get memory access. The Hub of the Propeller controls access to mutually exclusive resources, such as the Main RAM, to ensure system integrity. The hub runs at half of the system clock rate and gives access to each cog in a round robin fashion; therefore, there are 16 clock cycles between RAM access opportunities by a single cog. There could be up to 15 clocks of waiting before a cog can get access to the RAM if it missed the request window by 1 clock. Hub instructions also take more time to execute (8 cycles) so a RAM access takes between 8 and 23 clocks.

In order to accomplish high-bandwidth operations, a separate cog is started to perform that function. Cog startup, however, takes a bit of time itself. When a new cog is started, instructions and data from Main RAM are copied to the cog RAM. If assembly language is either the starter or what is started, it takes less than 9000 clocks, or a little less than 100us at 80 MHz. If Spin is launching a Spin cog, this can take about 25,000 clocks or 300 us (23). In either case, it is best to have a cog started and waiting for a trigger rather than try and start a cog the instant you need it to perform a function. This time delay could cause a `WAITCNT` to miss its mark and need another 53 seconds to wrap the system counter around again. This issue caused many early problems when developing the audio sampling procedures. It could work correctly at sampling rates less than 3000 Hz, but when using higher sampling rates, it took many minutes to get very small blocks of data sampled. 3000 Hz is 333us, which is very near the cog startup time of 300us. Increasing the sampling frequency much more than that caused the `WAITCNT` value to be passed by the time the cog was started, so the system counter had to loop around.

The proper way to handle cogs is to start a cog before you need it, and have it wait for a trigger of some sort – either an IO value changing or even a variable set by another cog, whatever is practical for the function of the cog. Other cogs can run continuously, if needed. For instance, an object available for the ADC from the Parallax Object Exchange starts a cog in which the ADC runs continuously, taking samples and writing to a single Main RAM address as quickly as the cog can process it. Other cogs that want the current ADC value simply read from that memory address.

Now that the SD card data format and cog function is understood, the software can be written in order to access the SD card files. Several objects are available at the Object Exchange for writing to SD cards. Each has different capabilities and different levels of error-checking and file system handling, but the basics of reading and writing data are all the same. These objects typically have a start method which is called in order to create a new cog devoted to file handling. This cog can be started as the device is powered up and sit ready, waiting for calls to its methods to write and read data to the SD card. One of these objects called SD-MMC_FATEngine.spin.

Before beginning work on the WAV recorder, the accuracy of sampling needed to be tested; could the microcontroller capture a value at the exact same time interval every time? To test this, a program was written in Spin to loop 20 times. During each loop, the command `WAITCNT(Time += Period)` was called, and then the current system counter value was stored to a variable. The period was 10000 clocks, or 8000 Hz with the 80 MHz clock. After the loop completed, the difference between sample values was calculated and the results sent to the terminal display on the PC. The difference between two values was 10000 clocks for all 19 intervals. Therefore, the `WAITCNT` command does exactly as claimed, and accurately spaced each sample.

Of the many objects available in the Object Exchange, there was a WAV recorder. However, this recorder used a sigma-delta analog-to-digital conversion technique that did not require a separate chip. To use the 12-bit ADC chip, the code had to be changed significantly to use a different method of analog-to-digital conversion. However, the overall structure of the code remained the same. Similar to the SD card object, this object also starts a new cog to initialize some settings, such as the sample rate. It also starts the ADC sampling, which continuous samples at the specified rate and writes the values to a data block in Main RAM. However, no files are being written to the SD card at this time. Once a method is called to start recording, a call to the SD card object opens a new file on the SD card, writes the header to the file, and then starts writing the data to the file. The data block to be written is broken into two 512-byte segments. These 512 bytes are the same size as a complete sector on the SD card. Flags are set by the ADC cog to signal which memory segment is being written to at the time. Once a segment gets filled, the flag is toggled, and the first cog then calls the `WRITEDATA` command to copy the segment to the file on the SD card. This process continues until a signal is given to stop the recording. At that time, the final file size is determined, appropriate data is written to the file header, and the file is closed. The SD card cog and ADC cog, however, are still always running and ready to begin another recording at any time.

This program was tested using the output from the headphone jack of an iPhone connected to the ADC input. The sampling rate was set to 22,050 Hz, or half of CD quality. A song was played from the phone, recorded to the SD card, and moved to a PC to hear the result. The song played back correctly, though with some noise. This test was repeated later with the amplifier circuit.  There was headroom in the prior test because the iPhone output did not drive the ADC input to its limits.  With the amplifier circuit, it was able to be amplified to the maximum unclipped setting prior to recording, resulting in a louder sound when played back.

Writing data to a display is one of the easier aspects of using the Propeller.  This device needed to remain small, so a 2-line, 16-character display was chosen.  It was also designed by Parallax, so the interface is just as simple to control.  ASCII DEC characters 32-127 are supported, as well as music tones using its built-in piezoelectric speaker.  Whole character strings can be sent with a single command.  A simple serial interface object written by Parallax and available on the Object Exchange can be used to communicate with the LCD display.

**Frequency Analysis**

A key component of the device is the ability to determine the frequencies included in the input signal.  This analysis must be performed in real time in order to produce a timely alarm for elephant handlers in the event of elephant infrasound communication.  Performing a Discrete Fourier Transform (DFT) on sampled input data can turn the sampled data into frequency information.  Additionally, the utilization of the Cooley-Tukey algorithm, which follows, increases the efficiency and speed of performing the operation.

The complex Fourier series

$$X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{kn} \quad W_N = e^{-j\frac{2\pi}{N}} \quad k = 0, 1, ..., N-1$$

would normally require $N^2$ complex multiplications and additions.  The algorithm developed by Cooley and Tukey can reduce the computations to less than $2N \log_2 N$ operations without the need for additional memory locations (24).

A Radix-2 FFT using decimation-in-time can accomplish this. Splitting the Fourier series into two sequences of even and odd indices results in

$$X[k] = \sum_{n\ even} x[n]W_N^{kn} + \sum_{n\ odd} x[n]W_N^{kn}$$

$$X[k] = \sum_{m=0}^{(N/2)-1} x[2m]W_N^{2mk} + \sum_{m=0}^{(N/2)-1} x[2m+1]W_N^{k(2m+1)}$$

With the following substitutions made:

$$W_N^2 = W_{N/2},\ f_1[m] = x[2m],\ f_2[m] = x[2m+1]$$

The series can be rewritten

$$X[k] = \sum_{m=0}^{N/2-1} f_1[m]W_{N/2}^{km} + W_N^k \sum_{m=0}^{N/2-1} f_2[m]W_{N/2}^{km}$$

$$= F_1[k] + W_N^k F_2[k], \quad k = 0, 1, ..., N-1$$

$F_1$ and $F_2$ are the N/2 DFTs of $f_1[k]$ and $f_2[k]$. Because of the periodicity of the summations and the symmetry of $W_N$, the series can be written

$$X[k] = F_1[k] + W_N^k F_2[k], \quad k = 0, 1, ..., N/2-1$$

$$X[k+\frac{N}{2}] = F_1[k] - W_N^k F_2[k], \quad k = 0, ..., \frac{N}{2}-1$$

This rearrangement of the equation cut the number of multiplies needed in half (25 p. 457). The recursive nature of this method then allows the N/2 DFTs to be decimated themselves, repeating the process until there is only a 2-point DFT to compute. Most importantly, this allows computations to occur in-place without the need for additional memory. Once the calculations are made there is no need for the original input pair. These computations are called butterflies due to their diagram. Given complex input pair (a,b), the calculation is

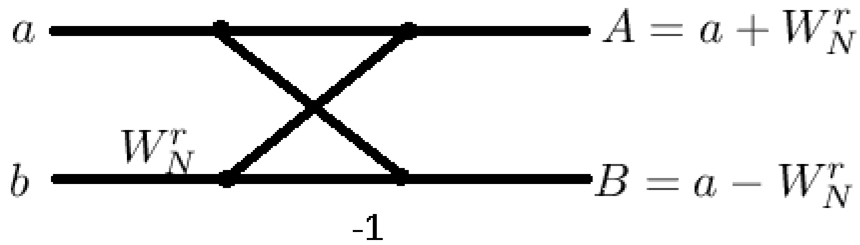



**Figure 15 - Butterfly diagram of a Fast Fourier Transform**

This Fast Fourier Transform (FFT) method has been created in a module available on the Parallax forums (https://forums.parallax.com/discussion/128292/heater-s-fast-fourier-transform.) The module uses the Cooley-Tukey method to perform the transform. It is written to take a 1024 sample input. It is capable of using a different sample size; however, changes would have to be made to the twiddle factors. To use the module, the addresses to two long-sized buffers are sent: **bx**, which contains the samples to transform, and **by**, used to perform the complex calculations and must be initialized to zeros. The steps to perform are also sent to the module via pre-defined command bits. These commands include CMD_DECIMATE, CMD_BUTTERFLY, and CMD_MAGNITUDE. All operations of the module are executed using assembly code, which greatly increases the speed of computation compared to using Spin.

CMD_DECIMATE performs a radix-2 decimation in time, reversing the order of the bits as per the Cooley-Tukey method. CMD_BUTTERFLY then performs the FFT through the different stages. CMD_MAGNITUDE converts the complex output to a magnitude number stored in **bx**.

The first 512 indices of **bx** contain the magnitudes of each frequency bin. If the sampling rate is equal to the number of samples in the FFT, then the indices in **bx** equal the frequency in hertz of the magnitude results. For example, if the sampling rate is 1024 Hz, and a 1024-sample FFT is performed, then the magnitude found at index 30 of **bx** is the magnitude at 30 Hz. No additional calculations would need to be made to convert the frequency bins to a range of frequencies in Hz. To ensure the FFT performs as desired, a software-derived waveform of known frequency was loaded in the **bx** array. A 16 Hz waveform ranging in magnitude from -2047 to +2047 was input and the resultant magnitude in index 16 was 2046, with magnitudes of 0 for all other indices.

### Device Assembly

The Propeller ASC+ card is a rather small board at 2.70 x 2.10 in (68.6 x 53.3 mm). It is Arduino-shield compatible; therefore, it has headers for all IO pins as well as for VIN to the board's regulators, ground, and +5 VDC out. A perfboard with compatible headers was connected to the ASC+. This board, from here forward called the main board, contains the amplifier, anti-aliasing filter, DAC, and LED driver chips, as well as connector jacks for cables to other boards. This board has approximately the same dimensions as the ASC+.

**Figure 16 - Infrasound Detection Device**

A plastic enclosure of size 4.724" L x 3.157" W (120.00mm x 80.19mm) X 2.311" (58.70mm) contains all components. Slots and holes were cut into the enclosure to allow for connections. The connected boards were mounted in the enclosure to allow access to the USB port, the SD Card slot and DC power connector. The four control buttons were placed on perfboard and connected to the main board via a cable and affixed to the enclosure cover. Likewise, the LED bar was put on perfboard, affixed to the cover and connected to the main board via a cable. The LCD Display was also affixed to the cover and connected to the main board via cable. The potentiometer, input jack, and output jack were wired directly to the main board and attached to the enclosure. An AAAx4 battery holder was attached to the inside of the enclosure. It was connected to an external power switch and its wires connected via jumper-style pin to the VIN and GND headers on the main board.

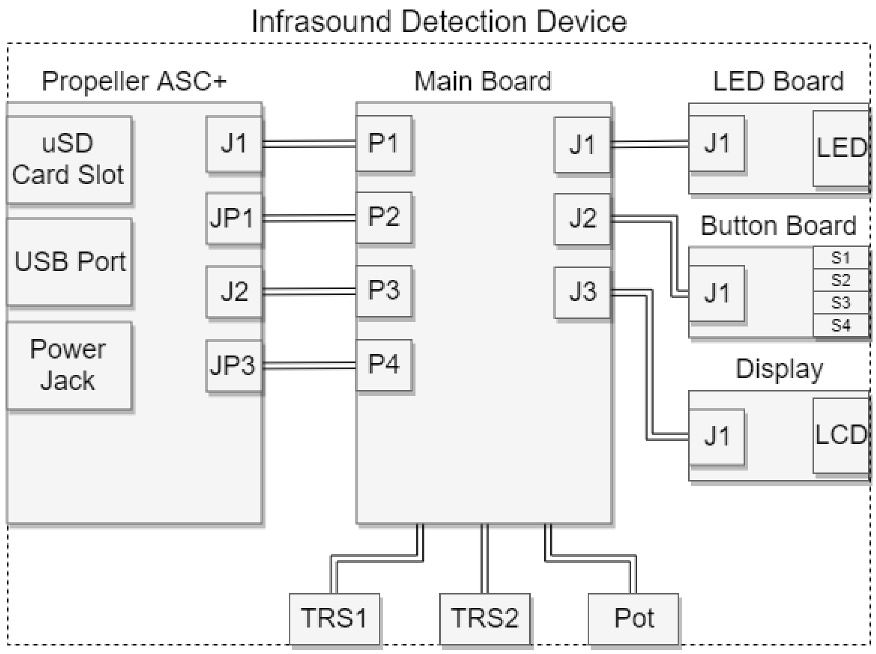



**Figure 17 - Block Diagram**

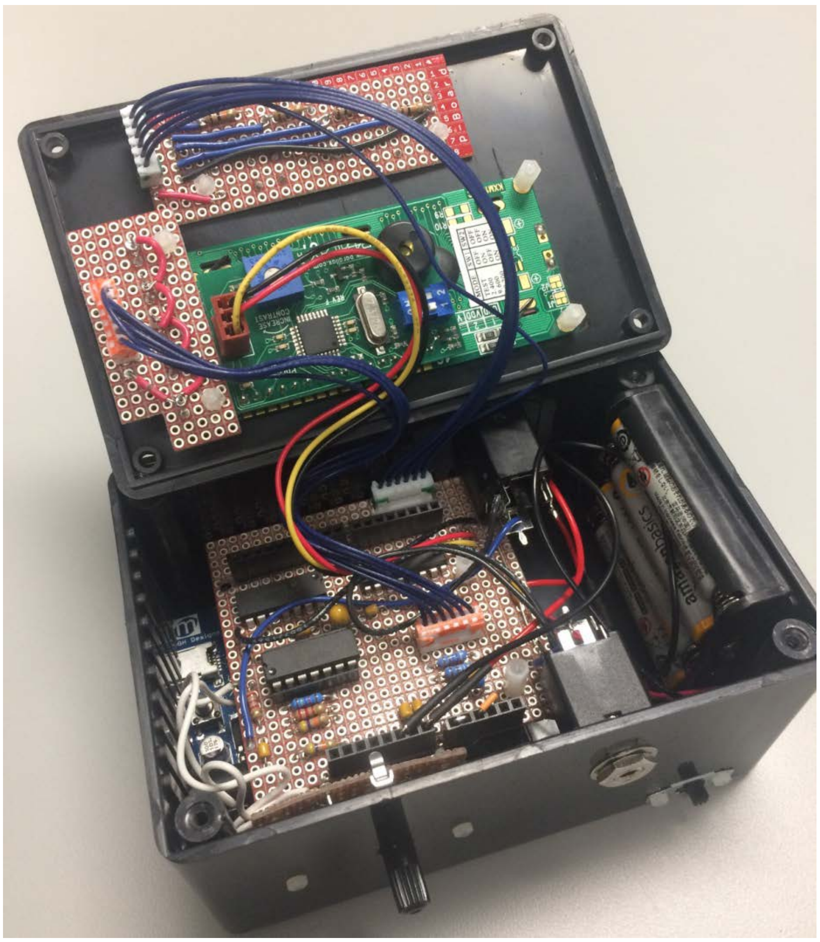


**Figure 18 - Infrasound Detection Device (Cover Open)**

**Device Specifications**

Power – 6-9 VDC external power jack; OR mini USB jack; OR 4 x AAA

Input – 3.5mm audio jack (mono); 0-1.25 peak-to-peak VAC; 2Hz – 10,000 Hz

Output – 3.5mm audio jack (mono)

Storage – micro SD card

Physical Dimensions: 4.7 in x 3.5 in x 2.5 in.

Weight: 10.5 ounces

# CHAPTER IV

# RESULTS

In this chapter the device is tested to determine if it meets the objectives. The first objective to test is the ability to accurately record infrasound. The second objective to test is the ability as a learning device and/or for field study. Last, the objective to accurately monitor infrasound frequencies will be tested.

The results showed that the device was able to record infrasound for extended periods of time. The device is easy to use and could be used for field usage, though the current battery capacity is limited. Testing also showed that infrasound was accurately detected and triggered the alarm.

## Recording

The recording capabilities of the device were tested using several input devices and two modes. The recording modes of the device are 1) Record continuous, until stopped or maximum file length is reached, or 2) Timed recording, where the recording stops after the selected timer is complete.

The maximum file length in number of bytes is $2^{31}$ bytes, or 2 GB. This is the maximum positive number that can be held in a long memory location, therefore the largest number the Propeller can track. The sampling rate on the device was set to the highest value (20,000 samples per second) and a continuous recording started. The file stopped with size 2,097,152KB, which is 2 GB. The length of the file was 14 hours, 54 minutes and 47 seconds. This test was conducted utilizing external power through the USB port.

The second test for recording was to examine the length of time the batteries could sustain recording. The device uses 4 AAA batteries providing 6 VDC. A regulator onboard the Propeller ASC+ board provides 5 VDC for the rest of the device. To conduct this test, the sampling rate was set to 1000 Hz and new Amazon Basics AAA batteries were installed. The record continuously option was started and the recording occurred until the batteries died. The resulting file stored on the SD Card was incomplete, since it could not be closed properly by the file system. However, based on the number of bytes written, it could be determined how long the device recorded before power down. This experiment was conducted three times.

| Test | File Size (KB) | Seconds | Duration |
|---|---|---|---|
| 1 | 37280 | 19087 | 5:18 |
| 2 | 37952 | 19431 | 5:23 |
| 3 | 42464 | 21742 | 6:02 |

**Table 1 - Battery Duration Test**

The results were not as good as expected. After just two hours, the display had faded completely. A DMM was connected to the device in order to monitor the current being drawn during operation. After initial startup, the device drew 58 mA from the batteries. During recording or playback operations, 80 mA were drawn. The current draw would spike to over 200 mA when the piezospeaker was being used.

The capacity of a battery in milliamp-hours (mAh) depends on the current it is discharging. An Energizer Max AAA has a capacity of 1100 mAh at a continuous current of 25 mA if dropping its output voltage to 0.8 VDC from the original 1.5 VDC. However, at 100 mA, it only has a capacity of just over 900 mAh (26). The same specification sheet contains a graph that shows the typical usage for a digital audio device at 50 mAh and the voltage vs. hours curve for that usage. The curve shows that the voltage will drop to less than 1.25 VDC in about 10 hours of use. Considering that the bare minimum usage by the device when the display is the only function executing is 58 mA, it follows that it will take even less time to drop to 1.25 VDC, which with 4 batteries gives 5 VDC, the bare minimum needed to drive the 5-volt components.

The third recording test was to judge the accuracy of the timed recording. A record time of one minute was selected. Two recordings were made at 1000 Hz and two recordings at 20,000 Hz. Once complete, the WAV files were opened on a PC using Audacity audio software. The software reported the recording length to be 1 min 0.416 sec for the 1000 Hz files and 59.507 and 59.149 sec for the 20,000 Hz. The variability can be explained because after the timer is started, the recording cog must complete a 256-sample block before it starts sending the data to the SD card. At a sampling rate of 1000 Hz, it would be over a quarter second of delay before recording. Likewise, at the end of the timer, recording is only shut down between 256-sample blocks. This could potentially add another quarter second of recording time.

The final recording test was for recording quality. The low pass filter on the device can be bypassed using a jumper. This allowed recordings in normal human aural range to be assessed for quality. The sampling frequency was set to 20,000 Hz and songs were played from an iPhone headphone jack through a cable to the input jack on the infrasound device. The gain knob was adjusted to get the amplitude in the proper range according to the LED readings. The songs were recorded and the SD card then taken to a PC to playback using its speakers. The songs were faithful representations of the original input from the iPhone. The low pass filter was included back into the signal back and recordings of signals from a function generator were made.

A 22 Hz signal from a function generator was recorded. The resulting WAV file was analyzed on a PC using Audacity software. The signal was a very clean sine wave with an exact frequency of 22Hz with no variation. The device is able to record very low frequencies accurately.

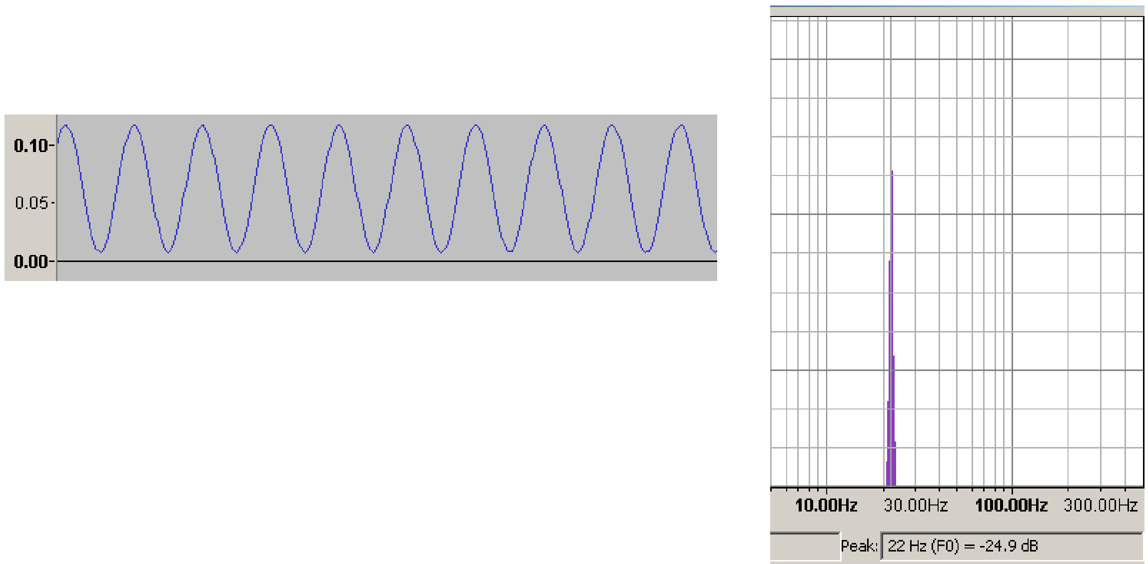



**Figure 19 - Signal and Frequency Spectrum of Recorded WAV File (22 Hz Signal Input)**

### Playback

The second objective to be tested was usability in the field. The recording portion of this objective has been shown. However, feedback is needed by the user to ensure that the recordings are in fact being made and sound proper, e.g. volume loud enough, not over-driven, etc. The recordings made previously were used in order to test the playback function of the device. Earbuds were connected to the output headphone jack. The interface was used to select a file on the SD card and played. Music files played the song accurately; however, there was a noise included in the sound which was worse on the beats of the song. There seems to be an issue with how the signal is sent from the DAC to the headphones. The only circuitry from the output of the DAC is a 0.1 uF capacitor in series to block the DC output of the DAC, since the signal is biased at 2.5 VDC.

One of the WAV files was a 44 Hz signal. Though technically it is not infrasound, it is still inaudible using earbuds because they are unable to produce low frequencies well. Pressing the 10x button on the interface played the tone back, now at 440 Hz. This matched the musical tone of “A”. The pitch matched when compared against an “A” generated by a tuner; therefore, the speed up function is accurate and allows the listener to “hear” infrasound. This, however, shortens the length of the signal, so an elephant call that is less than 10 seconds is now less than 1 second in length; however, it does let the user know that a sound was captured, which is the intent of the increased speed function. These signals are also plagued with a noise that prevents the signal from being a clean sound; however, the recorded sound is discernible from the noise.

### Infrasound Detection

The most important feature to test was the infrasound detection. Several input types were used, including pure tones, recorded elephant calls, and the geophone.

#### Pure Tones

An online tone generator (http://www.szynalski.com/tone-generator/) was used to generate sine wave signals for device input. The output of a PC was connected to the device input jack. The initial tone was set to 30 Hz, the gain adjusted, and then the frequency was changed and FFT results output to a serial terminal on the PC. The results of the experiment seem to show the weakness of the PC sound card in its ability to produce signals in the infrasound range. The frequencies detected were accurate, e.g. a 24 Hz tone was detected at 24 Hz by the device; however, the amplitudes of the signals varied greatly depending on the frequency generated, with a noticeable notch at 40 Hz.

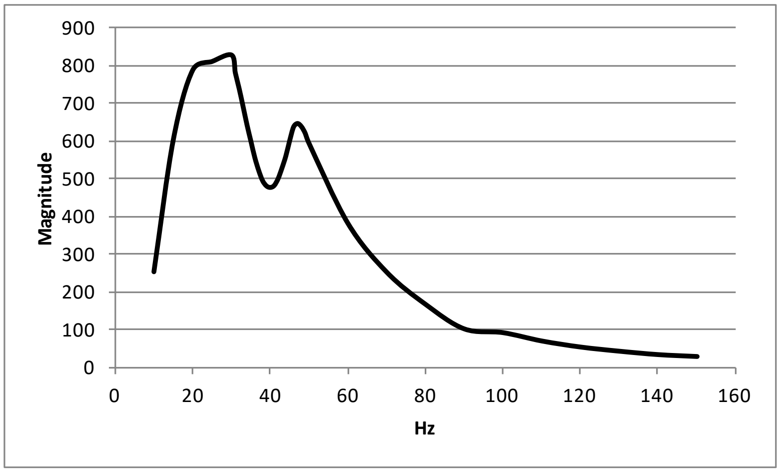



**Figure 20 - Frequency Response of Device Using Sound Card Output**

Due to the poor results of the PC tone generator, a function generator was used to provide a more accurate input. Although the notch at 40 Hz was no longer present, there was still significant signal loss starting at 40 Hz toward DC, very similar to the sound card source. It was found that the input coupling capacitor and impedance of the amplifier circuit were acting as a high-pass filter with a corner frequency of

$$f_c = \frac{1}{2\pi RC} = \frac{1}{2\pi(23440)(0.1uF)} = 67.93 \text{ Hz}$$

which was filtering out all of the infrasound, but especially the lowest frequencies. By changing the input coupling capacitor to 1.1 uF, the corner frequency was now 6.2 Hz, which allowed the desired frequencies to pass through.

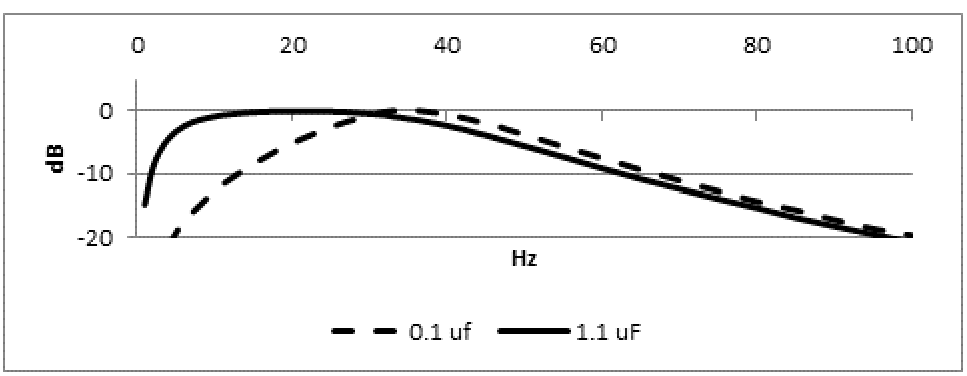



**Figure 21 - Frequency Response Using Function Generator for Two Different Coupling Capacitor Inputs**

### Elephant Calls

Despite the amplitude issues of the PC sound card noted earlier, recordings of elephant calls played into the device would provide a more realistic signal to test infrasound detection capabilities. ElephantVoices, an elephant advocacy group, has a database of elephant calls on their website (27). Their collection includes recordings of 7 rumbles, which are the calls containing infrasound. The following table gives their descriptions.

| Name | Description | Length |
|---|---|---|
| A3203414 | A relatively long, undulating rumble by a juvenile female | 0:06 |
| B1400110 | A soft, short rumble by a calf | 0:05 |
| B3304802 | A long, powerful and highly modulated rumble by an adult female | 0:06 |
| C2312431 | A long, unmodulated rumble by an adult female | 0:09 |
| F1200221 | A throaty rumble with a roaring quality by an adult female | 0:05 |
| U1605722 | A long pulsating rumble by an adult male | 0:07 |
| U1700443 | A short breathy rumble by an adult male | 0:02 |

**Table 2 - Elephant Calls**

The gain was set with the peak of the input between 60-80% of maximum magnitude with A3203414. The MP3 of the elephant call was played on a PC and sent via the headphone jack to the input of the device. The device was put into monitor mode, and the results of the FFT sent to the serial terminal and then transferred to MATLAB and plotted.

The MP3 was then input into MATLAB where it was resampled at 1024 Hz and an FFT performed on the data, which was then also plotted. The following figure shows a few comparisons of the MATLAB FFT results (on the left) to the device's FFT results (on the right.)

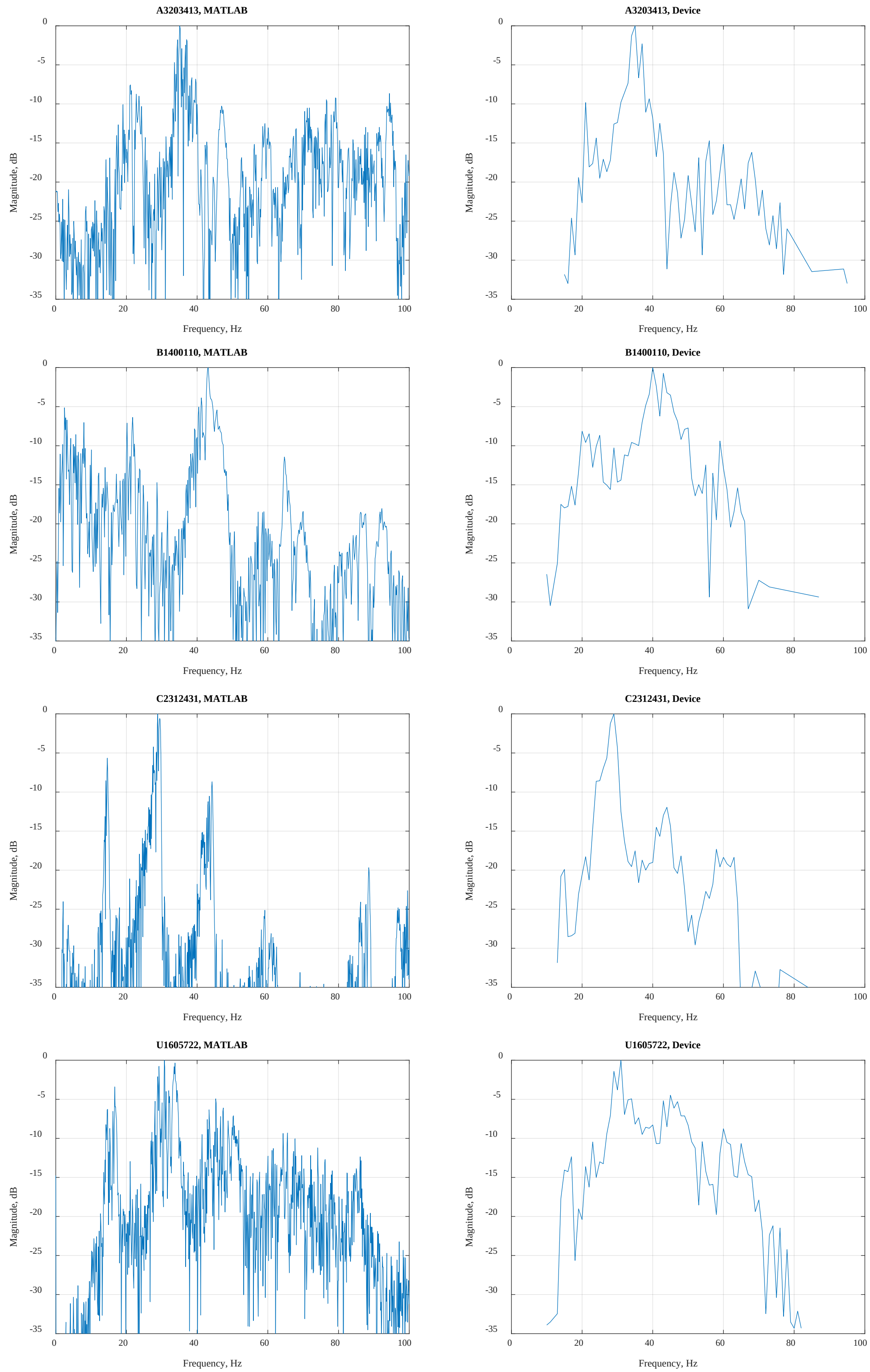



**Figure 22 - Frequency Analysis of Elephant Calls**

The MATLAB-generated FFT had more bins than the device FFT, which only had integer bins; therefore, the device FFT results look smoother. The FFT on the device accurately determined the frequencies that were prominent in the elephant call. For example, A3203413 had a peak at around 35 Hz which is clearly shown on both FFTs. The same can be shown on the other examples, as well.

When in the Monitor/Alarm mode, the device runs the FFT on 1 second of samples. If it detects a magnitude greater than the threshold (102 at a 10% detection threshold) at a frequency at or below 40 Hz, the alarm is tripped. It then displays the magnitude and frequency of the peak magnitude. The results of each elephant call were as follows:

| Name | Results | Detected? |
|---|---|---|
| A3203414 | 203 at 37 Hz, 190 at 37 Hz, 458 at 36 Hz | Yes |
| B1400110 | 117 at 26 Hz, 103 at 34 Hz, 139 at 38 Hz | Yes |
| B3304802 | 104 at 36 Hz, 146 at 30 Hz, 147 at 31 Hz | Yes |
| C2312431 | 480 at 25 Hz, 610 at 29 Hz, 423 at 28 Hz | Yes |
| F1200221 | 106 at 36 Hz, 106 at 36 Hz, 109 at 36 Hz | Yes |
| U1605722 | 116 at 40 Hz, 131 at 37 Hz, 134 at 37 Hz | Yes |
| U1700443 | 138 at 36 Hz, 103 at 36 Hz, 108 at 35 Hz | Yes |

**Table 3 - Infrasound Monitoring Results**

All 7 elephant calls were detected. The upper limit frequency is 40 Hz. Though that is not technically infrasound, low-frequencies are harder to discern. According to the ISO 226:2003 equal loudness curves, frequencies below 50 Hz would have to be over 35 dB louder than a normal conversation to have the same effective loudness. The higher-than-infrasound threshold was chosen to account for this.

### Geophone Test

To test the geophone for infrasound detection capability, a rig was constructed to suspend the geophone to allow for free movement. A small vibration motor was attached to the geophone and powered to provide a slow turn rate, less than 1200 RPM, which simulates a vibration less than 20 Hz.  The infrasound-detection device was set to monitor mode.  It triggered the alarm several times, displaying detected frequencies around 29 Hz each time.

# CHAPTER V

## CONCLUSION

In this work, it was shown that a low-cost device can be built which is capable of recording various analog inputs. The device can perform frequency analysis on those inputs and provide feedback to the user. The device is also shown to be low-cost and able to record for extended periods of time left unattended. The three objectives of this thesis – a low-cost infrasound warning device, an extended-period recording capability, and an educational tool – were met.

The total cost of components for the device was under $120. $75 were only for the Propeller board and the display; the remaining $45 were for the remaining electronics, interface, and enclosure components. If this were a production item, a designer could develop their own board instead of utilizing the ASC+ prototyping board used during this project. That board could be much less expensive because only the desired components would be included.

### Future Work

This device was only a prototype.  More work could be done to enhance some features of the device.  Using a printed circuit board to build the circuitry would enhance the look, clean operation, and reliability of the device.  Additional circuitry may need to be added and/or software changed to enhance playback to remove the noise that was present in this device.  A slightly larger enclosure to better accommodate the internal components would make the device easier to build and maintain.  That larger enclosure may also allow room for larger batteries, would help in allowing the device to be powered internally for longer periods of time.

Just simply switching to AA batteries may increase powered-on time to over 15 hours. Likewise, an external battery source could be plugged in using the DC power jack for a much longer usability period.  The device can take an input of up to 9 VDC.  This would also allow the device to stay in place without the external battery pack needing swapped in and out.

The ability to identify and eliminate constant background noise, such as an air conditioner, would be helping in preventing false alarms when infrasound from those sources would be able to surpass the alarm threshold.

The most important future work would be on the input sensor.  The experimentation with the geophone showed that it is able to detect vibration; however, that does not seem to be the most efficient sensor for infrasound, nor is it capable of recording the actually infrasound. Without a sensor that can consistently and accurately detect infrasound, the safety capability of this device would be in question.  Finding a microphone or other input sensor for low-cost would be an important piece of the continuing work.

REFERENCES

1. **Boyd-Barrett, Claudia.** Elephant attacked zoo keeper when he returned to stall second time. *Toledo Blade.* [Online] July 22, 2010. http://www.toledoblade.com/local/2010/07/22/Elephant-attacked-zoo-keeper-when-he-returned-to-stall-second-time.html.

2. **Johnston, Chuck.** Elephant kills keeper at Springfield, Missouri, zoo. *CNN.com.* [Online] October 11, 2013. https://www.cnn.com/2013/10/11/us/missouri-zoo-death/index.html.

3. **American Museum of Natural History.** Asian Elephants: Threats and Solutions. [Online] July 2007. https://www.amnh.org/explore/science-bulletins/bio/documentaries/wild-at-heart-the-plight-of-elephants-in-thailand/asian-elephants-threats-and-solutions/.

4. *African Elephants Respond to Distant Playbacks of Low-Frequency Conspecific Calls.* **Langbauer, Jr. William et al.** 1991, The Company of Biologists Limited, pp. 35-46.

5. *Automated detection of low-frequency rumbles of forest elephants: A critical tool for.* **Keen, Sara C. et al.** 2017, The Journal of the Acoustical Society of America, pp. 2715–2726.

6. *An Automatic Method to Detect the Presence of Elephant.* **Mohapatra, Arpit Sourav and Solanki, Sandeep Singh.** 2014, IEEE International Conference on Advanced Communication Control and Computing Technologies (ICACCCT), pp. 1515-1518.

7. *A Low Cost Infrasonic Recording System.* **De Silva, Girisha Durrel and Kasun De Zoyza.** 2007, 2007 4th IEEE Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications.

8. *The propeller experiment controller: low-cost automation for classroom experiments in learning and behavior.* **Varnon, Christopher A. and Charles I. Abramson.** 2013, Innovative Teaching.

9. **Shure.** SM58 Vocal Microphone Specification Sheet. *Shure.* [Online] 2015. http://www.shure.com/specification-sheets/sm58-specification-sheet-english.pdf.

10. —. SM | Microphones | Shure Americas. *Shure.* [Online] http://www.shure.com/americas/products/microphones/sm.

11. 1/2" Infrasound Prepolarized Microphone. *PCB Piezotronics.* [Online] http://www.pcb.com/microphones_preamplifiers_acoustic_accessories/specialty/low-frequency.

12. **GRAS Sound and Vibration.** GRAS 41AC-2 LEMO Outdoor Microphone with RemoteCheck for Community and Airport Noise. *GRAS Sound and Vibration A/S.* [Online] https://www.gras.dk/products/special-microphone/outdoor-microphones/product/631-41ac-2.

13. **PCB Piezotronics.** Layout 1. *PCB Piezotronics.* [Online] https://www.pcb.com/contentstore/MktgContent/LinkedDocuments/Acoustics/TM-AC-378A07_LowRes.pdf.

14. **Le Pichon, A. et al. (eds.).** *Infrasound Monitoring in Atmospheric Studies.* s.l. : Springer Science + Business Media, 2010.

15. *Elephant low-frequency vocalizations propagate in the ground and seismic playbacks of these vocalizations are detectable by wild African elephants (Loxodonta africana).* **O'Connell-Rodwell, Caitlin E. et al.** 2004, The Journal of the Acoustical Society of America, p. 2554.

16. **Sercel.** Geophones_Specifications_Sercel_EN.pdf. *Sercel - Seismic Acquisition Systems.* [Online] http://www.sercel.com/products/Lists/ProductSpecification/Geophones_specifications_Sercel_EN.pdf.

17. **Martin, Jeff.** Propeller Manual Version 1.2. *Propeller Manual | Parallax Inc.* [Online] 10 08, 2012. https://www.parallax.com/sites/default/files/downloads/P8X32A-Web-PropellerManual-v1.2.pdf.

18. **Microchip Technology.** 51700B.pdf. *MPLAB Starter Kit for dsPIC DSCs User Guide.* [Online] 2010. http://ww1.microchip.com/downloads/en/DeviceDoc/51700B.pdf.

19. **Parallax.** Parallax 2 x 16 Serial LCD (Backlit) | 27977 | Parallax Inc. *Parallax Inc.* [Online] https://www.parallax.com/product/27977.

20. **Sapp, Craig.** Microsoft WAVE soundfile format. *sapp.org.* [Online] http://soundfile.sapp.org/doc/WaveFormat/.

21. **Parallax.** Propeller ASC+. [Online] https://www.parallax.com/sites/default/files/styles/full-size-product/public/32214a_0.png?itok=u607HJZc.

22. **Microchip Technology.** Filterlab Filter Design Software. *Microchip Technology.* [Online] https://www.microchip.com/development-tools/resources/filterlab-filter-design-software.

23. **Parallax Inc.** Cogs. *Parallax Inc.* [Online] http://www.parallax.com/propeller/qna/Content/QnaTopics/QnaCogs.htm.

24. *An Algorithm for the Machine Calculation of Complex Fourier Series.* **Cooley, James W. and Tukey, John W.** 90, s.l. : American Mathematical Society, 1965, Vol. 19.

25. **Proakis, John G. and Manolakis, Dimitris K.** *Digital Signal Processing: Principles, Algorithms and Applications (3rd Edition).* s.l. : Prentice Hall, 1995.

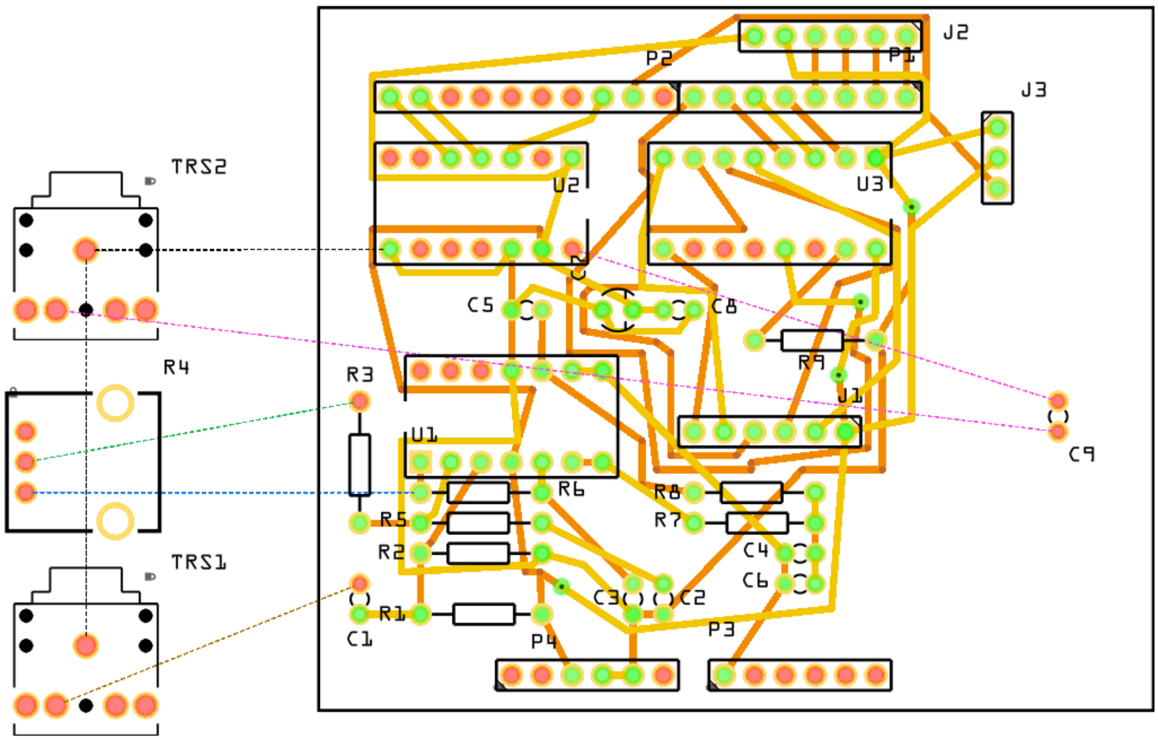
26. **Energizer.** Energizer Alkaline E92. *Energizer.com.* [Online] http://data.energizer.com/pdfs/e92.pdf.

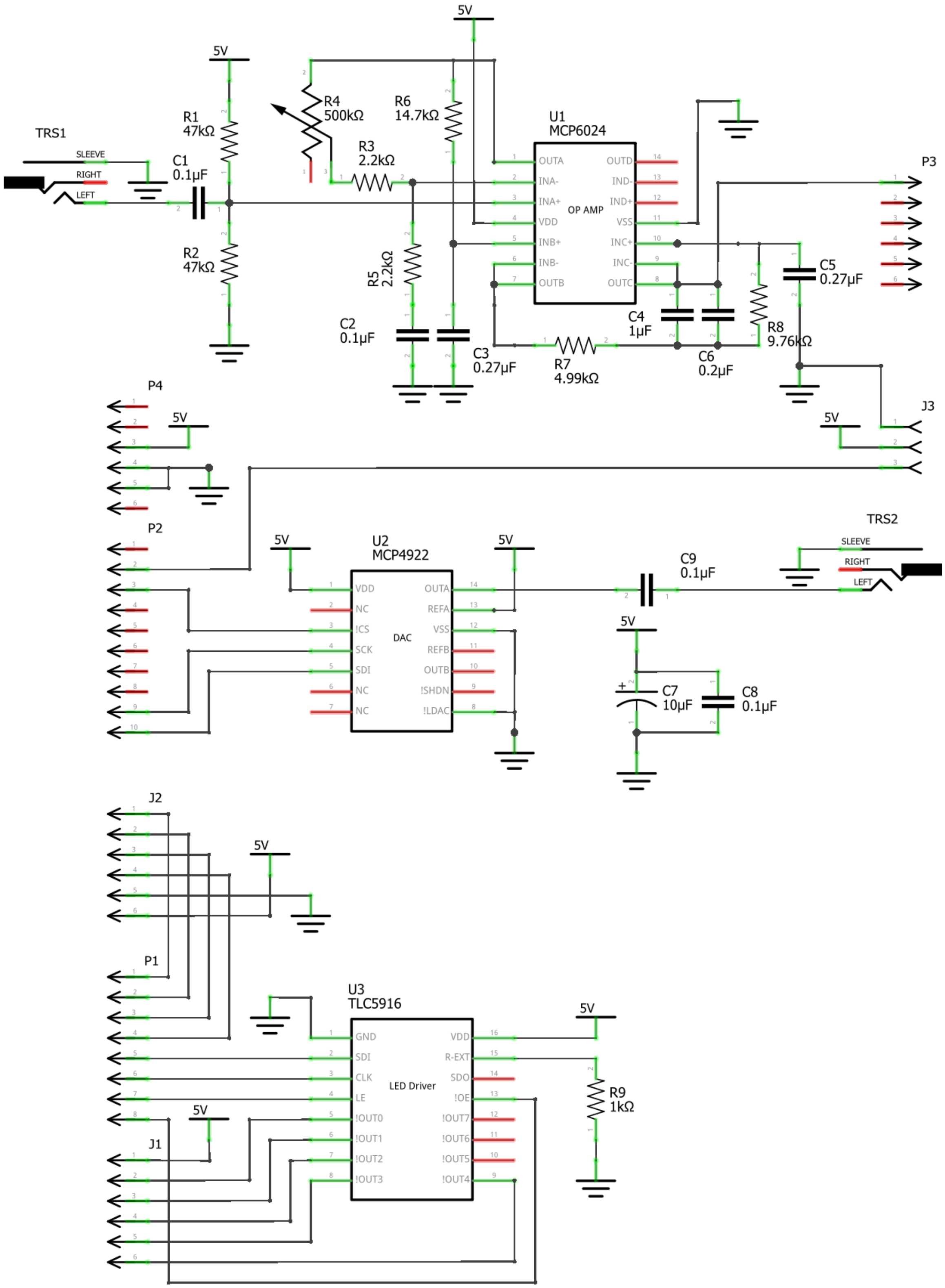
27. **Elephantvoices.** Elephant Call Types Database. *Elephantvoices.* [Online] https://www.elephantvoices.org/multimedia-resources/elephant-call-types-database.html.

# APPENDICES

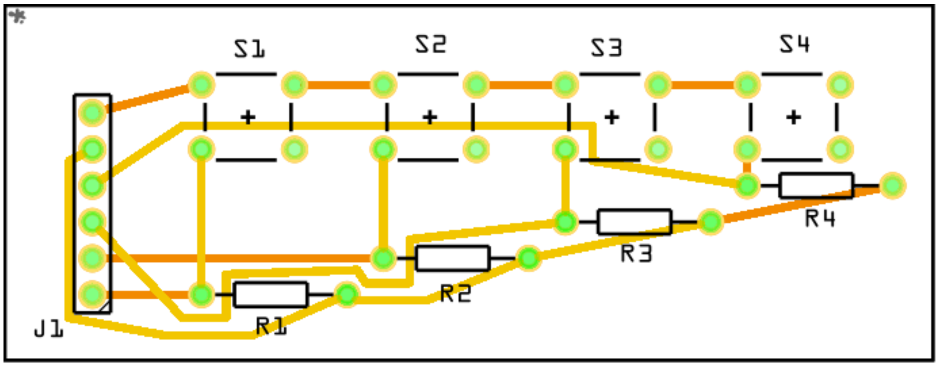
## A SCHEMATICS AND LAYOUT

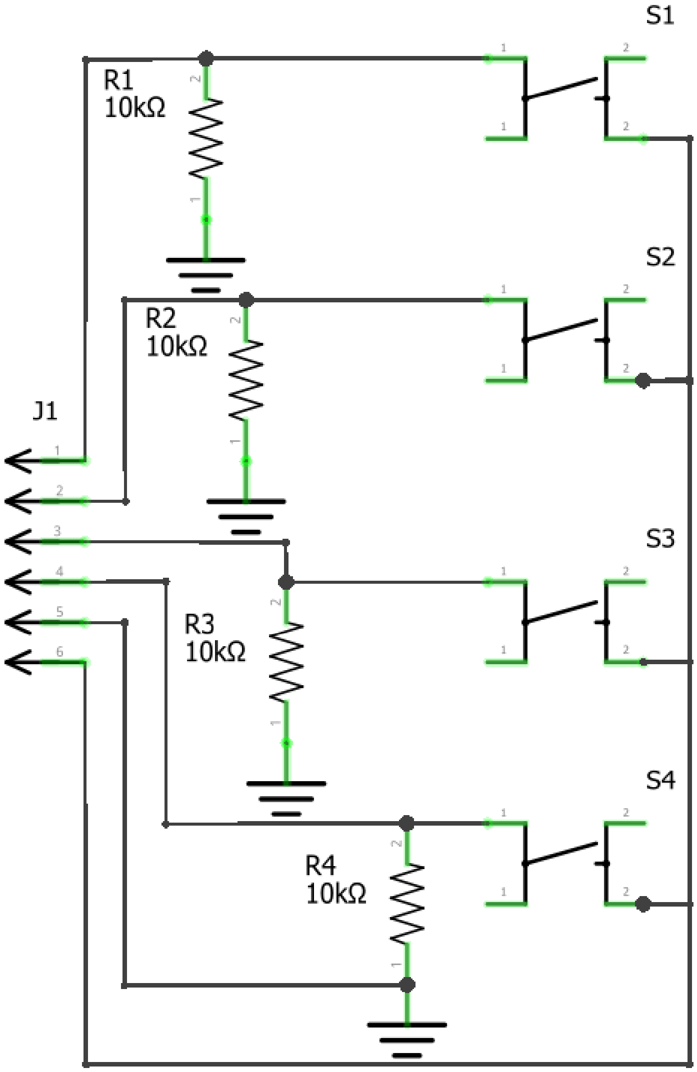
**Main Board**



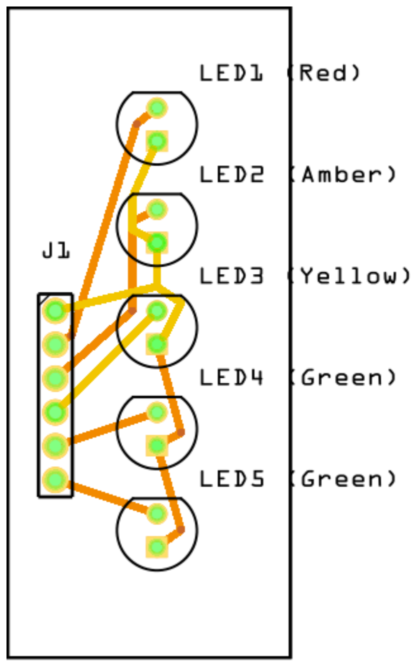

5V
R4
500kΩ
R6
14.7kΩ
U1
MCP6024
TRS1
SLEEVE
RIGHT
LEFT
R1
47kΩ
C1
0.1μF
R3
2.2kΩ
OUTA
INA-
INA+
VDD
INB+
INB-
OUTB
OP AMP
OUTD
IND-
IND+
VSS
INC+
INC-
OUTC
P3
R2
47kΩ
R5
2.2kΩ
C2
0.1μF
C3
0.27μF
R7
4.99kΩ
C4
1μF
C6
0.2μF
C5
0.27μF
R8
9.76kΩ
P4
J3
P2
U2
MCP4922
VDD
NC
!CS
SCK
SDI
NC
NC
DAC
OUTA
REFA
VSS
REFB
OUTB
!SHDN
!LDAC
C9
0.1μF
TRS2
C7
10μF
C8
0.1μF
J2
P1
U3
TLC5916
GND
SDI
CLK
LE
!OUT0
!OUT1
!OUT2
!OUT3
LED Driver
VDD
R-EXT
SDO
!OE
!OUT7
!OUT6
!OUT5
!OUT4
R9
1kΩ
J1

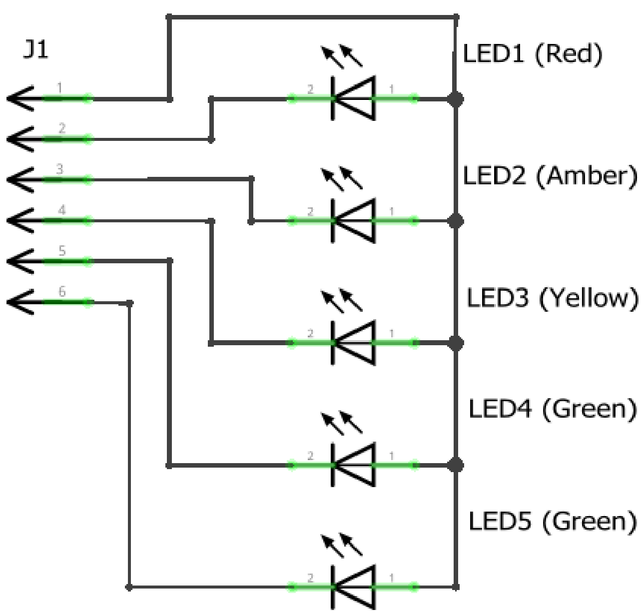
**Buttons Board**



**LED Board**


LED1 (Red)
LED2 (Amber)
J1
LED3 (Yellow)
LED4 (Green)
LED5 (Green)


J1
LED1 (Red)
LED2 (Amber)
LED3 (Yellow)
LED4 (Green)
LED5 (Green)

## B PARTS LIST

| PN | Manufacturer | Description | Qty | Price |
|---|---|---|---|---|
| 32214 | Parallax | Propeller ASC+ board | 1 | $ 49.99 |
| PRT-11417 | SparkFun Electronics | Protoboard Snappable | 1 | $ 7.95 |
| PRT-13268 | SparkFun Electronics | Arduino Stackable Header Kit | 1 | $ 1.50 |
| 1591TSBK | Hammond Manufacturing | BOX PLASTIC BLK 4.72"L X 3.16"W | 1 | $ 6.50 |
| 27977 | Parallax | Parallax 2 x 16 Serial LCD (Backlit) | 1 | $ 24.99 |
| A06KR06KR26E152A | JST Sales America | JUMPER 06KR-6S-P - 6" | 2 | $ 2.56 |
| B6B-PH-K-S(LF)(SN) | JST Sales America | CONN HEADER PH TOP 6POS 2MM | 4 | $ 1.32 |
| ACJS-MV35-3 | Amphenol | CONN JACK STEREO 3.5MM | 2 | $ 1.92 |
| SSA-LXB525-G2YAID | Lumex | LED Bars and Arrays LED Bars and Arrays 1.8x5.3mm 5 Unit LED Green/Ylw/Amb/Red | 1 | $ 3.58 |
| 1825910-7 | TE Connectivity ALCOSWITCH Switches | SWITCH TACTILE SPST-NO 0.05A 24V | 4 | $ 0.40 |
| 2482 | Keystone Electronics | HOLDER BATTERY 4CELL AAA 6" LEAD | 1 | $ 2.23 |
| TLC5916IN | Texas Instruments | IC LED DRIVER LINEAR 120MA 16DIP | 1 | $ 1.37 |
| MCP6024-E/P | Microchip Technology | IC OPAMP GP 10MHZ RRO 14DIP | 1 | $ 2.01 |
| MCP4922-E/P | Microchip Technology | IC DAC 12BIT DUAL W/SPI 14DIP | 1 | $ 2.70 |
| 296UD504B1N | CTS Electrocomponents | POT 500K OHM 0.15W CARBON LINEAR | 1 | $ 1.54 |
| CF14JT10K0 | Stackpole Electronics | RES 10K OHM 1/4W 5% AXIAL | 4 | $ 0.40 |
| CF14JT47K0 | Stackpole Electronics | RES 47K OHM 1/4W 5% AXIAL | 2 | $ 0.20 |
| MFR-25FBF52-14K7 | Yageo | RES 14.7K OHM 1/4W 1% AXIAL | 1 | $ 0.10 |
| CF14JT2K20 | Stackpole Electronics | RES 2.2K OHM 1/4W 5% AXIAL | 2 | $ 0.20 |
| MFR-25FBF52-4K99 | Yageo | RES 4.99K OHM 1/4W 1% AXIAL | 1 | $ 0.10 |
| CF18JT1K00 | Stackpole Electronics | RES 1K OHM 1/8W 5% CF AXIAL | 1 | $ 0.10 |
| MFR-25FBF52-9K76 | Yageo | RES 9.76K OHM 1/4W 1% AXIAL | 1 | $ 0.10 |
| A14042900UX0338 | Uxcell | CAP TANTALUM 10U 35V RADIAL | 1 | $ 0.67 |
| C440C105M5U5TA7200 | Kemet | CAP CER 1U 50V Z5U AXIAL | 2 | $ 0.82 |
| SA115E274MAR | AVX | CAP CER 0.27U 50V Z5U AXIAL | 2 | $ 2.10 |
| FG24X7R1H224KNT06 | TDK Corporation | CAP CER 0.22UF 50V X7R RADIAL | 1 | $ 0.29 |
| SA115C104KARC | AVX | CAP CER 0.1U 50V X7R AXIAL | 3 | $ 0.78 |
| | | | Total | $116.42 |

**!Main.spin**

```
{{
//////////////////////////////////////////////////////////////////////////////////////////
// This program is the top level module for the Infrasound detection device.  It handles
// the LCD, LED, and button interface.
//////////////////////////////////////////////////////////////////////////////////////////
}}

CON
        _clkmode = xtal1 + pll16x       'Standard clock mode * crystal frequency = 80 MHz
        _xinfreq = 5_000_000

  _dopin = 12
  _clkpin = 13
  _dipin = 11
  _cspin = 8
  _cdpin = -1 ' -1 if unused.
  _wppin = -1 ' -1 if unused.

  _rtcres1 = -1 ' -1 always.
  _rtcres2 = -1 ' -1 always.
  _rtcres3 = -1 ' -1 always.

  _rate = 1000 ' Default sample rate.
  '_rate = 8000
  '_rate = 22050

  _thresh = 10 'Default Threshold level

VAR
  long  i,j,k,SamplingFreq, AlarmThresh,FileName, maxfreq, maxamp, Stopped, CardCheck
  long  AlarmOn, RecTime, PlayFlag
  long  maxValue                                     'maximum value of current sample block
  long  Stack[100]
  long  x10

OBJ
  LCD      : "FullDuplexSerial.spin"
  adc      : "WAV-Recorder.spin"
  LED      : "LED Driver.spin"
  dac      : "WAV-Player.spin"

PUB Main
      'LCD.start(9, 9, %1000, 19_200)
      LCD.start(9, 9, %1000, 9_600)                      'initiate LCD screen communication
      Wait(1000)
      FormFeed
      LCD.tx(17)
      LCD.str(String("Starting..."))
      LCD.tx(211)                                        'Startup melody
      LCD.tx(220)
      LCD.tx(224)
      LCD.tx(227)
      LED.init
      LED.LED(4000)                                      'flash LEDs
      CustomChar
      Wait(1000)
      LED.LED(0)
      SamplingFreq := _rate                              'initialize
      AlarmThresh := _thresh                             'initialize
      Top


Pub Top
        WaitForNoButton
        repeat
          LCD.tx(18)
```

```
          FormFeed
          LCD.str(@TopMenu[i * 32])
          LCD.tx(148)
          LCD.str(String("   "))
          LCD.tx(0)
          LCD.tx(1)
          LCD.str(String(" OK:Select"))

          WaitForAnyButton
          if (ina & $F == UP)
            i += 1
            if (i > TopMenuLen)
              i := 0
          if (ina & $F == DWN)
            i -= 1
            if (i < 0)
              i := TopMenuLen
          if (ina & $F == SEL)
            case i
              0: Monitor
              1: Recording
              2: Playback
              3: Settings
          Wait(500)

Pub Monitor
        adc.ADCEngineStart(1024, @maxValue)
        WaitForNoButton
        LCD.tx(18)
        FormFeed
        LCD.str(String("Monitoring:Alarm"))
        LCD.tx(3)
        LCD.str(String(": To Stop"))
        Stopped := false
        cognew(spinMonitor(@AlarmOn, @maxfreq, @maxamp, @Stopped), @Stack)
        Wait(500)
        LCD.tx(18)
        repeat until (ina & BCK == BCK)
          if AlarmOn == true
            AlarmSignal
            AlarmOn := false
            FormFeed
            LCD.str(String("Monitoring:Alarm"))
            LCD.tx(3)
            LCD.str(String(": To Stop"))
          LED.LED(maxValue)
          maxValue := 0
          Wait(100)

        Stopped := true
        LED.LED(0)
        adc.ADCEngineStop
        Top

Pub Recording
        WaitForNoButton
        repeat
          FormFeed
          LCD.tx(2)
          LCD.str(@RecMenu[j * 32])
          brb
          LCD.str(String(" OK:Select"))
          WaitForAnyButton
          if (ina & $F == UP)
            j += 1
            if (j > RecMenuLen)
              j := 0
          if (ina & $F == DWN)
            j -= 1
            if (j < 0)
              j := RecMenuLen
```

```
          if (ina & $F == BCK)
            Top
          if (ina & $F == SEL)
            case j
              0: TimedRec
              1: RecUntilStop
          Wait(500)

Pub TimedRec | temp
        adc.ADCEngineStart(SamplingFreq, @maxvalue)
        adc.FATEngineStart
        WaitForNoButton
        k:=2
        repeat
          LCD.tx(18)
          FormFeed
          LCD.tx(2)
          LCD.tx(2)
          LCD.str(String("Record "))
          LCD.str(@RecLength[k * 7])
          brb
          LCD.str(String(" OK:Record"))
          LCD.tx(137)
          LCD.tx(24)
          WaitForAnyButton
          if (ina & $F == UP)
            k += 1
            if (k > RecValLen)
              k := 0
          if (ina & $F == DWN)
            k -= 1
            if (k < 0)
              k := RecValLen
          if (ina & $F == BCK)
            adc.unmount
            adc.ADCEngineStop
            adc.FATEngineStop
            Recording
          if (ina & $F == SEL)
            CardCheck := \adc.SDCardCheck
            if CardCheck <> true
              FormFeed
              LCD.str(String("No SD Card found"))
              Wait(2000)
            else
              Stopped := false
              RecTime := RecVal[k] * 60
              cognew(spinRecorder(@RecTime, @Stopped, @filename), @Stack)
              LCD.tx(18)
              FormFeed
              LCD.str(String("Recording "))
              LCD.str(@RecLength[k * 7])
              LCD.tx(148)
              LCD.str(String("Press OK to Stop"))
              Wait(4000)
              WaitForNoButton
              repeat until ((Stopped == true) OR (ina & SEL == SEL))
                LED.LED(maxValue)
                maxvalue := 0
                LCD.tx(128)
                LCD.str(String("Rec < "))
                if RecTime > 3600
                  LCD.dec((RecTime / 3600) + 1 )
                  LCD.str(String(" hr rem "))
                else
                  temp := RecTime / 60 + 1
                  LCD.dec(temp)
                  LCD.str(String(" m rem  "))
                Wait(200)
              Stopped := true
              FormFeed
```

```
          LCD.str(String("Recording       Stopped"))
          LED.LED(0)
          Wait(3000)
          adc.unmount
          WaitForNoButton
        Wait(500)

Pub RecUntilStop
        adc.ADCEngineStart(SamplingFreq, @maxvalue)
        adc.FATEngineStart
        WaitForNoButton
        repeat
          LCD.tx(18)
          FormFeed
          LCD.tx(2)
          LCD.tx(2)
          LCD.str(String("Rec Continuous"))
          BackSym
          LCD.str(String(" OK: Start Rec"))
          waitpne(%0000,%1001,0)             'wait for BCK or SEL button to be pressed
          if (ina & $F == BCK)
            adc.ADCEngineStop
            adc.FATEngineStop
            Recording
          if (ina & $F == SEL)
            CardCheck := \adc.SDCardCheck
            if CardCheck <> true
              FormFeed
              LCD.str(String("No SD Card found"))
              Wait(2000)
            else
              Stopped := false
              RecTime := 0
              cognew(spinRecorder(@RecTime, @Stopped, @filename), @Stack)
              LCD.tx(18)
              FormFeed
              LCD.str(String("Recording...  "))
              LCD.tx(148)
              LCD.str(String("Press OK to Stop"))
              WaitForNoButton
              repeat until ((Stopped == true) OR (ina & SEL == SEL))
                LED.LED(maxValue)
                maxValue := 0
                Wait(200)
              Stopped := true
              LED.LED(0)
              FormFeed
              LCD.str(String("Recording       Stopped"))
              Wait(3000)
              WaitForNoButton
          Wait(500)

Pub Playback
        dac.FATEngineStart
        dac.DACEngineStart(5000)
        CardCheck := \adc.SDCardCheck
            if CardCheck <> true
              FormFeed
              LCD.str(String("No SD Card found"))
              Wait(2000)
              dac.unmount
              dac.FATEngineStop
              dac.DACEngineStop
              Top
        WaitForNoButton
        filename:=\dac.ListNextFile
        repeat
          FormFeed
          LCD.tx(2)
          LCD.str(filename)
          BackSym
```

```
          LCD.tx(1)
          LCD.str(String(" OK:Play"))
          WaitForAnyButton
          if (ina & $F == DWN)
            filename:=\dac.ListNextFile
          if (ina & $F == BCK)
            dac.unmount
            dac.FATEngineStop
            dac.DACEngineStop
            Top
          if (ina & $F == SEL)
            if filename == String("No WAV files")
              FormFeed
              LCD.str(String("Not a valid file"))
            else
              x10 := false
              FormFeed
              LCD.str(filename)
              LCD.tx(148)
              cognew(spinPlayer(filename,@Stopped, @x10), @Stack)
              Wait(500)
              WaitForNoButton
              repeat until ((Stopped == true) OR (ina & SEL == SEL))
                LCD.tx(148)
                LCD.tx(0)
                LCD.str(String(":"))
                if x10 == false
                  LCD.dec(10)
                else
                  LCD.dec(1)
                LCD.str(String("x  OK:Stop "))
                if (ina & $F == UP)
                  not x10
                  WaitForNoButton
                Wait(200)

              Stopped := true
              FormFeed
              LCD.str(String("Playback Stopped"))
            Wait(2000)
            WaitForNoButton
          Wait(500)


Pub Settings
        WaitForNoButton
        repeat
          FormFeed
          LCD.tx(2)
          LCD.str(@SetMenu[j * 32])
          brb
          LCD.str(String(" OK:Select"))
          WaitForAnyButton
          if (ina & $F == UP)
            j += 1
            if (j > SetMenuLen)
              j := 0
          if (ina & $F == DWN)
            j -= 1
            if (j < 0)
              j := SetMenuLen
          if (ina & $F == BCK)
            Top
          if (ina & $F == SEL)
            case j
              0: SetSampFreq
              1: SetAlarmThresh
          Wait(500)

Pub SetSampFreq
        WaitForNoButton
```

```
        k:=1000
        if (SamplingFreq <> 0)
          k := SamplingFreq
        repeat
          FormFeed
          LCD.tx(2)
          LCD.tx(2)
          LCD.str(String("Freq: "))
          LCD.Dec(k)
          LCD.str(String(" Hz"))
          BackSym
          LCD.tx(0)
          LCD.tx(1)
          LCD.str(String(" OK:Set"))
          LCD.tx(24)
          LCD.tx(136)
          WaitForAnyButton
          if (ina & $F == UP)
            k += 1000
            if (k > 22000)
              k := 22000
          if (ina & $F == DWN)
            k -= 1000
            if (k < 1000)
              k := 1000
          if (ina & $F == BCK)
            Settings
          if (ina & $F == SEL)
            SamplingFreq := k
            FormFeed
            LCD.str(String("Frequency Set"))
            Wait(3000)
            WaitForNoButton
          Wait(500)

Pub SetAlarmThresh
        WaitForNoButton
        k:= AlarmThresh
        repeat
          FormFeed
          LCD.tx(2)
          LCD.tx(2)
          LCD.str(String("Threshold "))
          LCD.Dec(k)
          LCD.str(String("%"))
          brb
          LCD.str(String(" OK:Set"))
          LCD.tx(24)
          LCD.tx(140)
          WaitForAnyButton
          if (ina & $F == UP)
            k += 10
            if (k > 90)
              k := 90
          if (ina & $F == DWN)
            k -= 10
            if (k < 10)
              k := 10
          if (ina & $F == BCK)
            Settings
          if (ina & $F == SEL)
            AlarmThresh := k
            FormFeed
            LCD.str(String("Threshold Set"))
            Wait(3000)
            WaitForNoButton
          Wait(500)

Pub CustomChar

      LCD.tx(248)                               ' Define custom character 0 (Up arrow)
```

```
      LCD.tx(%00100)
      LCD.tx(%01110)
      LCD.tx(%11111)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)

      LCD.tx(249)                             ' Define custom character 1 (Down arrow)

      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%00100)
      LCD.tx(%11111)
      LCD.tx(%01110)
      LCD.tx(%00100)

      LCD.tx(250)                             ' Define custom character 2 (Right arrow)

      LCD.tx(%00000)
      LCD.tx(%00100)
      LCD.tx(%00110)
      LCD.tx(%11111)
      LCD.tx(%00110)
      LCD.tx(%00100)
      LCD.tx(%00000)
      LCD.tx(%00000)

      LCD.tx(251)                             ' Define custom character 3 (Left arrow)

      LCD.tx(%00000)
      LCD.tx(%00100)
      LCD.tx(%01100)
      LCD.tx(%11111)
      LCD.tx(%01100)
      LCD.tx(%00100)
      LCD.tx(%00000)
      LCD.tx(%00000)

Pub AlarmSignal
  'Displays the detected amplitude and frequency and plays tones, flashes screen
  LCD.tx(17)
  FormFeed
  LCD.Str(String("! "))
  LCD.dec(maxamp)
  LCD.Str(String(" @ "))
  LCD.dec(maxfreq)
  LCD.Str(String(" Hz !"))
  LCD.tx(212)
  AlarmOn := false
  repeat 15
    LCD.tx(225)
    LCD.tx(220)
    Wait(500)
    LCD.tx(18)
    LED.LED(0)
    Wait(500)
    LCD.tx(17)
    LED.LED(4000)
    if (ina & $F == BCK)
      quit
  WaitForNoButton
  LCD.tx(18)

PUB spinRecorder(RecTimeAddr, StoppedAddr, filenameaddr) ' Starts the recording

  adc.startRecordingWAVFile(RecTimeAddr, StoppedAddr, filenameaddr)
```

```
  cogstop(cogID)

PUB spinMonitor(AlarmOnAddr, maxfreqaddr, maxampaddr, StoppedAddr) ' Starts monitoring

  adc.Monitor(AlarmThresh, AlarmOnAddr, maxfreqaddr, maxampaddr, StoppedAddr)
  cogstop(cogID)

PUB spinPlayer(name,StoppedAddr, x10addr) ' Starts playback

  dac.startPlayingWAVFile(name, StoppedAddr, x10addr)
  cogstop(cogID)

PUB brb 'bottom row buttons

  BackSym
  LCD.tx(0)
  LCD.tx(1)

PUB BackSym '2nd line of display, space, then back symbol

  LCD.tx(148)
  LCD.tx(9)
  LCD.tx(3)

PUB FormFeed

  LCD.tx(12)
  Wait(10)
  LCD.tx(22)

PUB Wait(length) ' pause execution for 'length' msecs

  waitcnt((clkfreq / 1000 * length) + cnt)

PUB WaitForNoButton

  waitpeq(%0000,%1111,0)                          'wait until no buttons are pressed

PUB WaitForAnyButton

  waitpne(%0000,%1111,0)                          'wait for any button to be pressed

DAT
TopMenu       byte      "Monitor w/ Alarm",0[16]
              byte      "Recording",0[23]
              byte      "Playback",0[24]
              byte      "Settings",0[24]
TopMenuLen    byte      3
RecMenu       byte      "Timed Recording", 0[17]
              byte      "Rec until Stop",0[18]
RecMenuLen    byte      1
SetMenu       byte      "Sampling Freq", 0[19]
              byte      "Alarm Threshold", 0[17]
SetMenuLen    byte      1
RecLength     byte      "1 min",0[2]
              byte      "10 min",0[1]
              byte      "30 min",0[1]
              byte      "1 hr",0[3]
              byte      "2 hr",0[3]
              byte      "4 hr",0[3]
              byte      "8 hr",0[3]
              byte      "12 hr",0[2]
              byte      "24 hr",0[2]
RecValLen     byte      8
RecVal        word      1,10,30,60,120,240,480,720,1440
BCK           byte      %0001
UP            byte      %0010
DWN           byte      %0100
SEL           byte      %1000

{{
```

```
/////////////////////////////////////////////////////////////////////////////////////////
//                           TERMS OF USE: MIT License
/////////////////////////////////////////////////////////////////////////////////////////
// Permission is hereby granted, free of charge, to any person obtaining a copy of this
// software and associated documentation files (the "Software"), to deal in the Software
// without restriction, including without limitation the rights to use, copy, modify,
// merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
// permit persons to whom the Software is furnished to do so, subject to the following
// conditions:
//
// The above copyright notice and this permission notice shall be included in all copies
// or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
// INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
// PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
// HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
// CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
// OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
/////////////////////////////////////////////////////////////////////////////////////////
}}
```

**Heater_fft.spin**

```
'-----------------------------------------------------------------------------------------
' heater_fft module
' This module performs a 1024 point fft and returns the results in the first 512 cells of
' the datablock.
' Original Module "heater_fft" found on Parallax Object Exchange (OBEX) obex.parallax.com

' In place Radix-2 Decimation In Time FFT
'
' Michael Rychlik. 2011-1-25
'
'User optimization controls
'#define PASM_BUTTERFLIES       'Set this for fast PASM FFT, about 30ms
'#define USE_FASTER_MULT        'Set this for faster multiply
'#define USE_FASTER_SQRT       'Set this for faster but much bigger square root.
'-----------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------
CON
    'Specify size of FFT buffer here with length and log base 2 of the length.
    'N.B. Changing this will require changing the "twiddle factor" tables.
    '     and may also require changing the fixed point format (if going bigger)
    FFT_SIZE      = 1024
    LOG2_FFT_SIZE = 10

    CMD_DECIMATE  = %0001
    CMD_BUTTERFLY = %0010
    CMD_MAGNITUDE = %0100
    CMD_TEST      = %1000
'-----------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------
VAR
    long mailboxp
    byte cog
'-----------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------
PUB start (mailp)
'#ifdef PASM_BUTTERFLIES
    mailboxp := mailp
    LONG[mailboxp] := 0
    cog := cognew (@bfly, mailp)     'Check error?

Pub stop

   cogstop(cog)
```

```
PUB butterflies(cmd, bxp, byp)
    LONG[mailboxp + 4] := bxp                        'Address of x buffer
    LONG[mailboxp + 8] := byp                        'Address of y buffer
    LONG[mailboxp + 0] := cmd                        'Do butterflies and/or decimation
    repeat while LONG[mailboxp + 0] <> 0
'-----------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------
DAT
              org       0
bfly          mov       mb_ptr, par
              rdlong    command, mb_ptr wz           'Wait for run command in mailbox
        if_z  jmp       #bfly

              add       mb_ptr, #4                   'Fetch x array address from mbox
              rdlong    bx_ptr, mb_ptr

              add       mb_ptr, #4                   'Fetch y array address from mbox
              rdlong    by_ptr, mb_ptr
              sub       mb_ptr, #8

              test      command, #CMD_DECIMATE wz    'Bit reversal required on data?
        if_z  jmp       #:no_decimate

'Radix-2 decimation in time. (The bit reversal stage)
'Moves every sample of bx to a postion given by reversing the bits of its original array
'index. This is a direct translation of the Spin decimate above, original Spin code used
'as comments. N.B. Only the x array is bit-reversed it is up to the app to clear y.

              mov       c, fft_size_                 'repeat i from 0 to FFT_SIZE - 1
              mov       b, #0

:dloop        mov       a, b                         'revi := i >< LOG2_FFT_SIZE
              mov       rev_a, a
              rev       rev_a, #32 - LOG2_FFT_SIZE

              cmp       a, rev_a wc                  'if i < revi
        if_nc jmp       #:skip_rev

              shl       a, #2                        'Times 4 as we are reading longs
              shl       rev_a, #2

              mov       hub_ptr, bx_ptr              'tx1 := long[bxp + i * 4]
              add       hub_ptr, a
              rdlong    tx, hub_ptr

              mov       hub_rev_ptr, bx_ptr    'long[bxp + i * 4] := long[bxp + revi * 4]
              add       hub_rev_ptr, rev_a
              rdlong    ty, hub_rev_ptr
              wrlong    ty, hub_ptr

              wrlong    tx, hub_rev_ptr              'long[bxp + revi * 4] := tx1

:skip_rev     add       b, #1
              djnz      c, #:dloop

:no_decimate
              test      command, #CMD_BUTTERFLY wz   'Perform buterflies?
        if_z  jmp       #:no_butterfly

'Apply FFT butterflies to N complex samples in buffers bx and by, in time decimated order
'Resulting FFT is produced in bx and by in the correct order.
'This is a direct translation from the Spin code above, original Spin code in comments.

              mov       flight_max, fft_size_        'flight_max := FFT_SIZE / 2
              sar       flight_max, #1
              mov       wangleSkip, fft_size_        'wangleSkip := FFT_SIZE * 4
              shl       wangleSkip, #2

              mov       butterflySpan, #4            'butterflySpan := 4
```

```
              mov       butterfly_max, #1              'butterfly_max := 1
              mov       flightSkip, #4                 'flightSkip := 4

              'Loop through all the decimation levels
              mov       level, #LOG2_FFT_SIZE          'level := LOG2_FFT_SIZE
:lloop                                                 'repeat
              mov       b0x_ptr, bx_ptr                'b0x_ptr := @bx
              mov       b0y_ptr, by_ptr                'b0y_ptr := @by

              mov       b1x_ptr, b0x_ptr              'b1x_ptr := b0x_ptr + butterflySpan
              add       b1x_ptr, butterflySpan

              mov       b1y_ptr, b0y_ptr              'b1y_ptr := b0y_ptr + butterflySpan
              add       b1y_ptr, butterflySpan

              'Loop though all the flights in a level
              mov       flight, flight_max                 'flight := flight_max
:floop                                                     'repeat
{new}         mov       wangle, #0

              'Loop through all the butterflies in a flight
              mov       butterfly, butterfly_max           'butterfly := butterfly_max

              'Do the initial pass optimization, when W = [1,0] we don't need to multiply
              ' c = 1 (well, 4096/4096), d = 0
              mov       k2, #0                             'k2 := (d * (a + b)) / 4096
              rdlong    a, b1x_ptr                         'a := LONG[b1x_ptr]
              mov       k1, a                              'k1 := (a * (c + d)) / 4096
              neg       k3, a                              'k3 := (c * (b - a)) / 4096
              rdlong    b, b1y_ptr                         'b := LONG[b1y_ptr]
              add       k3, b                              'k3 := (c * (b - a)) / 4096
              jmp       #:continue_bloop

:bloop        ' repeat                                     'At last...the butterfly.
              rdlong    a, b1x_ptr                         'a := LONG[b1x_ptr]

              'Precompute the optimization for c=0, d=-1
              neg       k1, a                              'k1 := (a * (c + d)) / 4096
              neg       k2, a                              'k2 := (d * (a + b)) / 4096

              rdlong    b, b1y_ptr                         'b := LONG[b1y_ptr]

              'Precompute the optimization for c=0, d=-1
              sub       k2, b                              'k2 := (d * (a + b)) / 4096
              mov       k3, #0                             'k3 := (c * (b - a)) / 4096

              mov       c, wangle
{getcos}      add       c, sin_90                     'For cosine, add 90Â°
              test      c, sin_90       wc            'Get quadrant 2|4 into c
              test      c, sin_180      wz            'Get quadrant 3|4 into nz
              negc      c, c                          'If quadrant 2|4, negate offset
              or        c, sin_table                  'OR in sin table address >> 1
              shl       c, #1                         'Shift left to get final word address
              rdword    c, c                          'Read word sample from $E000 to $F000
              negnz     c, c                          'If quadrant 3|4, negate sample

              sar       c, #4 wz                      'Scale to +/- 4095

        if_z  jmp       #:continue_bloop              ' if c==0, we already kave k1, k2, k3

              mov       d, wangle
{getsin}      test      d, sin_90       wc            'Get quadrant 2|4 into c
              test      d, sin_180      wz            'Get quadrant 3|4 into nz
              negc      d, d                          'If quadrant 2|4, negate offset
              or        d, sin_table                  'OR in sin table address >> 1
              shl       d, #1                         'Shift left to get final word address
              rdword    d, d                          'Read word sample from $E000 to $F000
              negnz     d, d                          'If quadrant 3|4, negate sample

              sar       d, #4                         'Scale to +/- 4095
              neg       d, d                          'We want -cos
```

```
              mov       m1, c                      'k1 := (a * (c + d)) / 4096
              add       m1, d
              mov       m2, a
              call      #mult
              mov       k1, m1
              sar       k1, #15 - 3

              mov       m1, a                      'k2 := (d * (a + b)) / 4096
              add       m1, b
              mov       m2, d
              call      #mult
              mov       k2, m1
              sar       k2, #15 - 3

              mov       m1, b                      'k3 := (c * (b - a)) / 4096
              sub       m1, a
              mov       m2, c
              call      #mult
              mov       k3, m1
              sar       k3, #15 - 3

:continue_bloop

              mov       tx, k1                      'tx := k1 - k2 (part I)
              mov       ty, k1                      'ty := k1 + k3 (part I)

              rdlong    k1, b0x_ptr                 'k1 := LONG[b0x_ptr]

              sub       tx, k2                      ' (part II) moved from above to take
              add       ty, k3                      ' advantage of the hub wait times

              rdlong    k2, b0y_ptr                 'k2 := LONG[b0y_ptr]

              mov       a, k1                       'LONG[b1x_ptr] := k1 - tx
              sub       a, tx
              wrlong    a, b1x_ptr

              mov       a, k2                       'LONG[b1y_ptr] := k2 - ty
              sub       a, ty
              wrlong    a, b1y_ptr

              mov       a, k1                       'LONG[b0x_ptr] := k1 + tx
              add       a, tx
              wrlong    a, b0x_ptr

              mov       a, k2                       'LONG[b0y_ptr] := k2 + ty
              add       a, ty
              wrlong    a, b0y_ptr

              add       b0x_ptr, #4                 'b0x_ptr += 4
              add       b0y_ptr, #4                 'b0y_ptr += 4

              add       b1x_ptr, #4                 'b1x_ptr += 4
              add       b1y_ptr, #4                 'b1y_ptr += 4

              add       wangle, wangleSkip          'wangle += wangleSkip

              djnz      butterfly, #:bloop          'while --butterfly <> 0

              add       b0x_ptr, flightSkip         'b0x_ptr += flightSkip
              add       b0y_ptr, flightSkip         'b0y_ptr += flightSkip
              add       b1x_ptr, flightSkip         'b1x_ptr += flightSkip
              add       b1y_ptr, flightSkip         'b1y_ptr += flightSkip
              djnz      flight, #:floop             'while --flight <> 0

              shl       butterflySpan, #1           'butterflySpan <<= 1
              shl       flightSkip, #1              'flightSkip <<= 1

              shr       flight_max, #1              'flight_max >>= 1
```

```
              shr       wangleSkip, #1
              shr       wSkip, #1                    'wSkip >>= 1
              shl       butterfly_max, #1            'butterfly_max <<= 1
              djnz      level, #:lloop               'while --level <> 0
:no_butterfly
              test      command, #CMD_MAGNITUDE wz   'Calculate magnitudes?
        if_z  jmp       #:no_magnitude

'Calculate magnitudes from the complex results in x and y. Results placed into x

              mov       c, fft_size_                 'repeat i from 0 to FFT_SIZE
              add       c, #1                        'That is one more than half FFT_SIZE
                                                     'so as to include the Nyquist freq
              mov       b0x_ptr, bx_ptr
              mov       b0y_ptr, by_ptr

:mloop        rdlong    m1, b0x_ptr
              sar       m1, #LOG2_FFT_SIZE - 1
              mov       m2, m1
              call      #mult
              mov       input, m1

              rdlong    m1, b0y_ptr
              sar       m1, #LOG2_FFT_SIZE - 1
              mov       m2, m1
              call      #mult
              add       input, m1

              call      #sqrt

              wrlong    root, b0x_ptr                      'Write result to x array

              add       b0x_ptr, #4                        'Next x and y element and loop
              add       b0y_ptr, #4
              djnz      c, #:mloop

:no_magnitude
              mov       command, #0
              wrlong    command, mb_ptr
              jmp       #bfly
'-----------------------------------------------------------------------------------------

'-----------------------------------------------------------------------------------------
mult           'Account for sign
'#ifdef USE_FASTER_MULT
              abs       m1, m1 wc
              negc      m2, m2
              abs       m2, m2 wc
              'Make t2 the smaller of the 2 unsigned parameters
              mov       m3, m1
              max       m3, m2
              min       m2, m1
              'Correct the sign of the adder
              negc      m2, m2
{{#else
              abs       m3, m1 wc
              negc      m2, m2
#endif}}
              'My accumulator
              mov       m1, #0
              'Do the work
:mul_loop     shr       m3, #1 wc,wz           'Get the low bit of t2
        if_c  add       m1, m2                 'If it was a 1, add adder to accumulator
              shl       m2, #1                 'Shift the adder left by 1 bit
       if_nz  jmp       #:mul_loop             'Continue as long as there are no more 1's
mult_ret      ret

m1            long      0
m2            long      0
m3            long      0
'-----------------------------------------------------------------------------------------
```

```
'-----------------------------------------------------------------------------------------
'#ifdef USE_FASTER_SQRT
'Faster code square root (Chip Gracey after discussion with lonesock on Propeller Forums)
sqrt          mov       root, h40000000
              cmpsub    input, root  wc
              sumnc     root, h40000000
              shr       root, #1

              or        root, h10000000
              cmpsub    input, root  wc
              sumnc     root, h10000000
              shr       root, #1

              or        root, h04000000
              cmpsub    input, root  wc
              sumnc     root, h04000000
              shr       root, #1

              or        root, h01000000
              cmpsub    input, root  wc
              sumnc     root, h01000000
              shr       root, #1

              or        root, h00400000
              cmpsub    input, root  wc
              sumnc     root, h00400000
              shr       root, #1

              or        root, h00100000
              cmpsub    input, root  wc
              sumnc     root, h00100000
              shr       root, #1

              or        root, h00040000
              cmpsub    input, root  wc
              sumnc     root, h00040000
              shr       root, #1

              or        root, h00010000
              cmpsub    input, root  wc
              sumnc     root, h00010000
              shr       root, #1

              or        root, h00004000
              cmpsub    input, root  wc
              sumnc     root, h00004000
              shr       root, #1

              or        root, h00001000
              cmpsub    input, root  wc
              sumnc     root, h00001000
              shr       root, #1

              or        root, h00000400
              cmpsub    input, root  wc
              sumnc     root, h00000400
              shr       root, #1

              or        root, #$100
              cmpsub    input, root  wc
              sumnc     root, #$100
              shr       root, #1

              or        root, #$40
              cmpsub    input,root  wc
              sumnc     root, #$40
              shr       root, #1

              or        root, #$10
              cmpsub    input,root  wc
```

```
              sumnc     root, #$10
              shr       root, #1

              or        root, #$4
              cmpsub    input,root  wc
              sumnc     root, #$4
              shr       root, #1

              or        root, #$1
              cmpsub    input,root  wc
              sumnc     root, #$1
              shr       root, #1
sqrt_ret      ret

h10000000     long      $10000000
h04000000     long      $04000000
h01000000     long      $01000000
h00400000     long      $00400000
h00100000     long      $00100000
h00040000     long      $00040000
h00010000     long      $00010000
h00004000     long      $00004000
h00001000     long      $00001000
h00000400     long      $00000400

{{#else

'Faster code square root (Chip Gracey after discussion with lonesock on Propeller Forums)
sqrt          mov       root, #0                'Reset root
              mov       mask, h40000000         'Reset mask (constant in register)
:sqloop       or        root, mask              'Set trial bit
              cmpsub    input, root wc          'Subtract root from input if fits
              sumnc     root, mask              'Cancel trial bit, set root bit if fit
              shr       root, #1                'Shift root down
              shr       mask, #2                'Shift mask down
              tjnz      mask, #:sqloop          'Loop until mask empty
sqrt_ret      ret
#endif }}
h40000000     long      $40000000
'------------------------------------------------------------------------------------------

'------------------------------------------------------------------------------------------
'Large constants
fft_size_     long      FFT_SIZE
sin_90        long      $0800
sin_180       long      $1000
sin_table     long      $E000 >> 1               'ROM sin table base shifted right

'COG variables
level         long      0
flight        long      0
butterfly     long      0
flight_max    long      0
wSkip         long      0
butterflySpan long      0
butterfly_max long      0
flightSkip    long      0
k1            long      0
k2            long      0
k3            long      0
a             long      0
b             long      0
c             long      0
d             long      0
tx            long      0
ty            long      0
b0x_ptr       long      0
b0y_ptr       long      0
b1x_ptr       long      0
b1y_ptr       long      0
mb_ptr        long      0
```

```
bx_ptr        long      0
by_ptr        long      0
wangle        long      0
wangleSkip    long      0

rev_a         long      0
hub_ptr       long      0
hub_rev_ptr   long      0
command       long      0
root          long      0
mask          long      0
input         long      0
'------------------------------------------------------------------------------------------
              fit       496
'#endif
'------------------------------------------------------------------------------------------

'------------------------------------------------------------------------------------------
'    This file is distributed under the terms of the The MIT License as follows:
'
'    Copyright (c) 2011 Michael Rychlik
'
'    Permission is hereby granted, free of charge, to any person obtaining a copy
'    of this software and associated documentation files (the "Software"), to deal
'    in the Software without restriction, including without limitation the rights
'    to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
'    copies of the Software, and to permit persons to whom the Software is
'    furnished to do so, subject to the following conditions:
'
'    The above copyright notice and this permission notice shall be included in
'    all copies or substantial portions of the Software.
'
'    THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
'    IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
'    FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
'    AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
'    LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
'    OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
'    THE SOFTWARE.
'------------------------------------------------------------------------------------------

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

**LED Driver.spin**

```
{{ LED Driver
\\ This module initializes communication with the LED driver, as well as takes an input
\\ value, decides the LEDs to turn on and send the command to the LED driver.
}}
CON
  ValMax = 4095
  CLK = 5
  SDI = 4
  LE = 6
  OE = 7

OBJ

PUB init

  dira[CLK]~~
  dira[SDI]~~
  dira[LE]~~
  dira[OE]~~
  outa[OE] := 0
  outa[CLK] := 0
  outa[LE] := 0

PUB LED(Value) | data

  data := %00000000
  if Value > 2457
```

```
    data := %00001000
  if Value > 2867
    data := %00011000
  if Value > 3276
    data := %00111000
  if Value > 3685
    data := %01111000
  if Value > 3890
    data := %11111000

  'debug - comment out for final
  'if Value > 1000
  '  data := %11111000

  repeat 8
    outa[SDI] := data & 1                                     'set SDI to lowest data bit
    outa[CLK] := 1                                            'set CLK to 1
    outa[CLK] := 0                                            'set CLK to 0
    data >>= 1                                                'shift data right 1 bit

  outa[LE] := 1
  outa[LE] := 0
  outa[SDI] := 0

PUB Out(data)

  repeat 8
    outa[SDI] := data & 1                                     'set SDI to lowest data bit
    outa[CLK] := 1                                            'set CLK to 1
    outa[CLK] := 0                                            'set CLK to 0
    data >>= 1                                                'shift data right 1 bit

  outa[LE] := 1
  outa[LE] := 0
DAT
{{

/////////////////////////////////////////////////////////////////////////////////////////
//                           TERMS OF USE: MIT License
/////////////////////////////////////////////////////////////////////////////////////////
// Permission is hereby granted, free of charge, to any person obtaining a copy of this
// software and associated documentation files (the "Software"), to deal in the Software
// without restriction, including without limitation the rights to use, copy, modify,
// merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
// permit persons to whom the Software is furnished to do so, subject to the following
// conditions:
//
// The above copyright notice and this permission notice shall be included in all copies
// or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
// INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
// PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
// HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
// CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
// OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
/////////////////////////////////////////////////////////////////////////////////////////
}}
```

**SD-MMC_FATEngine.spin**
**Unedited from the Parallax object exchange except that many unused methods were removed to conserve program space.**

**WAV-Player.spin**

```
{{
/////////////////////////////////////////////////////////////////////////////////////////
// WAV-Player_DACEngine
// This module contains functions to play WAV files recorded by the device through the
// DAC.
// Original module from the Parallax OBEX (obex.parallax.com):
```

```
// WAV-Player - Author: Kwabena W. Agyeman
/////////////////////////////////////////////////////////////////////////////////////////
}}
CON
  dopin = 12
  clkpin = 13
  dipin = 11
  cspin = 8
  cdpin = -1 ' -1 if unused.
  wppin = -1 ' -1 if unused.

  rtcres1 = -1 ' -1 always.
  rtcres2 = -1 ' -1 always.
  rtcres3 = -1 ' -1 always.
OBJ  fat: "SD-MMC_FATEngine.spin"

VAR

  long clocksPerSample, samplesPerSecond, dataSize, filesize
  word callerPointer, callePointer, data2DAC, count, LastVal
  byte stopped, cogIdentification
  long writeResult, curpos, i, switch, linecount, playFlag
  word datablock[512]

PUB startPlayingWAVFile(filePathName, StoppedAddr, x10addr) '' 52 Stack Longs
  result := \PlayWAV(filePathName, StoppedAddr, x10addr)

PRI PlayWAV(filePathName, StoppedAddr, x10addr) | x10' 48 Stack Longs
  ifnot(fat.partitionMounted)
    fat.mountPartition(0)
  result := fat.openFile((filePathName), "R")
  fat.fileSeek(4)
  filesize := fat.readLong + 8
  fat.fileSeek(24)                                     'get samples per second
  samplesPerSecond := fat.readLong
  fat.fileSeek(40)                                     'get number of data bytes
  dataSize := fat.readLong
  clocksPerSample := (clkfreq / samplesPerSecond)      'calculate clocks to wait per samp
  fat.fileSeek(44)                                     'move to first data chunk
  stopped := false
  curpos := 0

  fat.fileSeek(44)
  callerPointer := callePointer := 0                   'init flag
  fat.readData(@datablock,512)
  not callerPointer                                         ' callerpointer = -1
  'callePointer was set to 0.  After a block of DAC PASM, it will flip to -1.
  'Wait until it equals 0 again (playing A block)
  'then continue loading data into block B (callerpointer = -1)
  repeat while (callerPointer <> callepointer)         'callerpointer=1, callepointer = 0
  repeat while(callerPointer == callePointer)          'callerpointer=1, callepointer = 1
  long[StoppedAddr] := false                           'ready to play first block
  repeat until(long[StoppedAddr] == true)
    x10 := long[x10addr]
    if x10 == true
      clocksPerSample := ((clkfreq / samplesPerSecond) / 10) #> (clkfreq / 80000)
    else
      clocksPerSample := (clkfreq / samplesPerSecond)
    playflag := 1
    fat.readData(@dataBlock[256 & callerPointer], 512)      'read next data block
    not callerPointer
    repeat while (callerpointer <> callepointer)
    curpos := fat.filetell
    if curpos >= filesize - 512
      fat.fileseek(44)                                  'loops playback
  playflag := 0

PUB DACEngineStart(sampleRate) '' 9 Stack Longs

'' /////////////////////////////////////////////////////////////////////////////////////
'' // Starts up the ADC driver running on a cog.
```

```
'' ///////////////////////////////////////////////////////////////////////////////////////
  DACEngineStop
  if(chipver == 1)
    clocksPerSampleAddress := @clocksPerSample
    dataBlockAddress := @datablock
    callePointerAddress := @callePointer
    playFlagAddress := @playFlag
    playFlag := 0
    clocksPerSample := clkfreq / sampleRate

    cogIdentification := cognew(@initialization, 0)
    result or= ++cogIdentification

PUB DACEngineStop '' 3 Stack Longs

'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Shuts down the ADC driver running on a cog.
'' ///////////////////////////////////////////////////////////////////////////////////////

  if(cogIdentification)
    cogstop(-1 + cogIdentification~)

PUB SDCardCheck
'' ///////////////////////////////////////////////////////////////////////////////////////
'' Tries to mount SD card.  If successful, returns true (-1), else returns an error code.
'' ///////////////////////////////////////////////////////////////////////////////////////
   result := false
   ifnot(fat.partitionMounted)
    fat.mountPartition(0)
   result := fat.partitionMounted
   unmount
   return

PUB unmount
  fat.unmountpartition

PUB ListNextFile | validwav, namebuffer, index
'' ///////////////////////////////////////////////////////////////////////////////////////
'' Searches through SD card and returns new WAV file name.
'' ///////////////////////////////////////////////////////////////////////////////////////
    ifnot(fat.partitionMounted)
      fat.mountPartition(0)
    validWAV := false
    count := 0
    repeat until ((validWAV == true) OR (count > 1000))
      count += 1
      nameBuffer := fat.ListEntries(String("n"))
      repeat index from 0 to 8
        if (byte[namebuffer][index] == 46)
          if((byte[namebuffer][index + 1] == 87) OR (byte[namebuffer][index + 1]  ==
119))
            if((byte[namebuffer][index + 2]  == 65) OR (byte[namebuffer][index + 2] ==
97))
              if((byte[namebuffer][index + 3]  == 86) OR (byte[namebuffer][index + 3]  ==
118))
                validWAV := true
                quit
             quit
           quit

    if validWAV == false
      namebuffer := String("No WAV files")
    return namebuffer

PUB FATEngineStart
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Starts up the SDC driver running on a cog and checks out a lock for the driver.
'' ///////////////////////////////////////////////////////////////////////////////////////
  return fat.FATEngineStart(DOPin, CLKPin, DIPin, CSPin, WPPin, CDPin, RTCRes1, RTCRes2,
RTCRes3)
```

```
PUB FATEngineStop '' 6 Stack Longs
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Shuts down the SDC driver running on a cog and returns the lock used by the driver.
'' ///////////////////////////////////////////////////////////////////////////////////////
  fat.FATEngineStop

DAT

' ////////////////////////////////////////////////////////////////////////////////////////
'                       DAC Driver
' ////////////////////////////////////////////////////////////////////////////////////////

                        org     0

' //////////////////////Initialization/////////////////////////////////////////////////////

initialization          or      outa,smask                                'CS high
                        andn    outa,cmask
                        andn    outa,dmask
                        or      dira,cmask                                'output CLK
                        or      dira,smask                                'output CS
                        or      dira,dmask                                'output D

                        mov     playerPointer,   dataBlockAddress         '

                        rdlong  playerRate,      clocksPerSampleAddress    'Setup timing

                        mov     timeCounter,     playerRate
                        add     timeCounter,     cnt

' ////////////////////////////////////////////////////////////////////////////////////////
'                       Player
' ////////////////////////////////////////////////////////////////////////////////////////

outerLoop               rdlong  outputOn,       playFlagAddress       'Playing?
                        mov     counter,         #256                 '256 Samples
                        rdlong  playerRate,     clocksPerSampleAddress

 ' //////////////////////Inner Loop//////////////////////////////////////////////////////
innerLoop               rdword  value, playerPointer    'get value to write to DAC
                        waitcnt timeCounter,     playerRate     ' Wait until next output
                        mov     command, comInit        'init command
                        test    outputOn, #1    wc      ' set c to outputOn value
                        muxc    command, shutdown       'set shutdown bit to OutputOn val
                        and     value, mask12           'trim to ensure only 12 bits
                        or      command, value          'put value in command stream
                        rev     command, #16            'reverse bits

                        andn    outa,smask              'CS low

                        mov     bits,#16                'ready 16 bits

bloop                   test    command, #1  wc         'get lowest bit
                        muxc    outa, dmask             'set data bit
                        nop
                        or      outa,cmask              'CLK high
                        nop
                        nop
                        andn    outa,cmask              'CLK low
                        nop
                        shr     command,#1
                        djnz    bits,#bloop             'next data bit

                        or      outa,smask              'CS high
                        add     playerPointer, #2                        '
                        djnz    counter,         #innerLoop                '

' //////////////////////Outer Loop//////////////////////////////////////////////////////////

                        rdword  buffer,          callePointerAddress wz   'Flip data pntr
                        sumz    buffer,          #1                       '
```

```
                        wrword  buffer,          callePointerAddress      '
if_nz                   mov     playerPointer,   dataBlockAddress         '

                        jmp     #outerLoop     wz                         ' Loop.

' ////////////////////////////////////////////////////////////////////////////////////////
'                       Data
' ////////////////////////////////////////////////////////////////////////////////////////

mask12                  long    $FFF
dmask                   long     1 << 15
cmask                   long     1 << 14
smask                   long     1 << 10
comInit                 word    $3000
shutdown                long    $1000
command                 long    0
value                   word    0

' ////////////////////Addresses/////////////////////////////////////////////////////////

clocksPerSampleAddress  long    0
dataBlockAddress        long    0
callePointerAddress     long    0
playFlagAddress         long    0

' /////////////////////Run Time Variables//////////////////////////////////////////////

buffer                  res     1
counter                 res     1
playerPointer           res     1
playerRate              res     1
timeCounter             res     1
bits                    res     1
outputOn                res     1

' ////////////////////////////////////////////////////////////////////////////////////////

                        fit     496

CON WAVFileHeaderSize = 44 ' DO NOT EDIT!

DAT WAVFileHeaderData ' DO NOT EDIT!

' ////////////////////////////////////////////////////////////////////////////////////////

  byte byte "RIFF" ' "RIFF" chunk header.
  byte long 0 ' "RIFF" chunk size = (fileSize - 8). Offset 4.
  byte byte "WAVE" ' File type.

  byte byte "fmt " ' "fmt " chunk header.
  byte long 16 ' "fmt " chunk size.
  byte word 1 ' Audio format.
  byte word 1 ' Nuber of channels.
  byte long 0 ' Sample rate.EDITED by setup function!
  byte long 0 ' Byte rate. EDITED by setup function!
  byte word 2 ' Block align.
  byte word 16 ' Bits per sample.

  byte byte "data" ' "data" chunk header.
  byte long 0 ' "data" chunk size = (fileSize - 44). Offset 40.

' ////////////////////////////////////////////////////////////////////////////////////////

{{

//////////////////////////////////////////////////////////////////////////////////////////
//                          TERMS OF USE: MIT License
//////////////////////////////////////////////////////////////////////////////////////////
// Permission is hereby granted, free of charge, to any person obtaining a copy of this
// software and associated documentation files (the "Software"), to deal in the Software
// without restriction, including without limitation the rights to use, copy, modify,
```

```
// merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
// permit persons to whom the Software is furnished to do so, subject to the following
// conditions:
//
// The above copyright notice and this permission notice shall be included in all copies
// or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
// INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
// PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
// HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
// CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
// OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
///////////////////////////////////////////////////////////////////////////////////////////
}}
```

**WAV-Recorder.spin**

```
{{
///////////////////////////////////////////////////////////////////////////////////////////
// WAV-Recorder
// This module contains functions to record from the ADC convert onboard the ASC+ board
// and call FFT functions to determine the frequency content of the input.
//
// Original module from the Parallax OBEX (obex.parallax.com):
// WAV-Recorder Analog to Digital Converter Engine
// - Author: Kwabena W. Agyeman
///////////////////////////////////////////////////////////////////////////////////////////
}}
CON
  dopin = 12
  clkpin = 13
  dipin = 11
  cspin = 8
  cdpin = -1 ' -1 if unused.
  wppin = -1 ' -1 if unused.

  rtcres1 = -1 ' -1 always.
  rtcres2 = -1 ' -1 always.
  rtcres3 = -1 ' -1 always.
MAX_LEN = 64
OBJ  fat: "SD-MMC_FATEngine.spin"
     fft: "heater_fft.spin"

VAR

  long clocksPerSample, bx[2048], by[1024], i, maxamp, maxfreq, RecTime
  long str2dec, count, TimerStack[20], idx, maxValue
  long fft_mailbox_cmd        'Command
  long fft_mailbox_bxp        'Address of x buffer
  long fft_mailbox_byp        'Address of y buffer
  word callerPointer, callePointer, samplesPerSecond, bxcallerPointer, bxcallePointer
  byte ADCcogID, nameCounter, TimerID, nstr[MAX_LEN], AlarmOn
  word datablock[512]

PUB getMaxamp

  return maxamp

PUB getmaxfreq

  return maxfreq

PUB startRecordingWAVFile(RecTimeAddr, StoppedAddr, filenameaddr) | namebuffer, index

'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Mounts SD card. Finds the largest numbered file (first 3 digits) and increments
'' // for new file name.
'' ///////////////////////////////////////////////////////////////////////////////////////
    ifnot(fat.partitionMounted)
    fat.mountPartition(0)
```

```
    'autoname the files.
    fat.listEntries("W") ' Goto the top of the directory.
    waitcnt(clkfreq + cnt)
    result := 0
    repeat while(nameBuffer := fat.listEntries("N"))
      i += 1
      str2dec := 0
      repeat index from 0 to 2                      'first 3 chars of file name are used
        str2dec *= 10
        if (byte[namebuffer][index] - "0" > 9)
          str2dec := 0
          quit
        str2dec += byte[namebuffer][index] - "0"
      result #>= str2dec                            ' Find the largest number

    nameBuffer := decf(result + 1,3)
    nameCounter := strsize(nameBuffer)

    bytemove(@AUTONAMEArray, string("000"), 3)
    bytemove((@AUTONAMEArray + (3 - nameCounter)), nameBuffer, nameCounter)
    long[filenameaddr] := @AUTONameArray

    result := \recordWAV(@AUTONAMEArray,RecTimeAddr, StoppedAddr)

    \fat.unmountPartition

PRI recordWAV(filePathName, RecTimeAddr, StoppedAddr)
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Starts recording a WAV file to the SD/MMC card.
'' ///////////////////////////////////////////////////////////////////////////////////////
  ifnot(fat.partitionMounted)
    fat.mountPartition(0)

  fat.openFile(fat.newFile(filePathName), "W")
  fat.writeData(@WAVFileHeaderData, WAVFileHeaderSize)

  long[StoppedAddr] := false
  if long[RecTimeAddr] > 0
    TimerID := cognew(Timer(RecTimeAddr, StoppedAddr), @TimerStack)     'Cog stops itself

  callerPointer := callePointer
  repeat until(long[StoppedAddr]==true or (fat.fileSize => posx))
    repeat while(callerPointer == callePointer)
    fat.writeData(@datablock[256 & callerPointer], 512)
    not callerPointer

  fat.fileSeek(4)
  fat.writeLong(fat.fileSize - 8)
  fat.fileSeek(40)
  fat.writeLong(fat.fileSize - WAVFileHeaderSize)
  long[StoppedAddr] := true

PUB Monitor(AlarmThresh, AlarmOnAddr, maxfreqaddr, maxampaddr, StoppedAddr) | temp
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Performs an FFT on incoming analog signal and sets alarm flag if results under
' //  target freq are over the alarm threshold.
'' ///////////////////////////////////////////////////////////////////////////////////////
  AlarmThresh := 2047 * AlarmThresh / 200           'take original % * max magnitude / 2
  fft.start(@fft_mailbox_cmd)
  long[StoppedAddr] := false
  callerPointer := callePointer
  bxcallerPointer := bxcallePointer
  repeat until(long[StoppedAddr] == true)
    repeat while(bxcallerPointer == bxcallePointer)
    longfill(@by, 0, 1024)
    fft.butterflies(fft#CMD_DECIMATE | fft#CMD_BUTTERFLY | fft#CMD_MAGNITUDE, @bx[1024 &
bxcallerPointer], @by)
    long[maxfreqaddr] := 0
    long[maxampaddr] := 0
    repeat i from 2 to 40    'sum 3 consecutive frequencies and test against threshold
```

```
      'temp := bx[1024 & bxcallerPointer + i] + bx[1024 & bxcallerPointer + i - 1] +
bx[1024 & bxcallerPointer + i  + 1]
      temp := bx[1024 & bxcallerPointer + i]
      if temp > AlarmThresh
        if temp > long[maxampaddr]
          long[maxfreqaddr] := i
          long[maxampaddr] := temp
        long[AlarmOnAddr] := true
    not bxcallerPointer
  fft.stop

PUB ADCEngineStart(sampleRate, maxvalueaddr)
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Starts up the ADC driver running on a cog.
'' // Returns true on success or false.
'' // SampleRate - Sample rate to record audio at.
'' // maxvalueaddr - Address of Max Value variable - used to set LED in top module
'' ///////////////////////////////////////////////////////////////////////////////////////
  ADCEngineStop
  if(chipver == 1)
    clocksPerSample := (clkfreq / (samplesPerSecond := ((sampleRate <# 44_100) #> 1)))
    bytemove((@WAVFileHeaderData + 24), @samplesPerSecond, 2)
    sampleRate := (samplesPerSecond << 1)
    bytemove((@WAVFileHeaderData + 28), @sampleRate, 4)
    clocksPerSampleAddress := @clocksPerSample
    dataBlockAddress := @datablock
    callePointerAddress := @callePointer
    bxAddress := @bx
    bxcallePointerAddress := @bxcallePointer
    maxValueAddress := maxvalueaddr
    longfill(@by, 0, 1024)
    ADCcogID := cognew(@ADCinit, 0)
    result or= ++ADCcogID

PUB ADCEngineStop
  if(ADCcogID)
    cogstop(-1 + ADCcogID~)

PUB FATEngineStart
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Starts up the SDC driver running on a cog and checks out a lock for the driver.
'' ///////////////////////////////////////////////////////////////////////////////////////
  return fat.FATEngineStart(DOPin, CLKPin, DIPin, CSPin, WPPin, CDPin, RTCRes1, RTCRes2,
RTCRes3)

PUB FATEngineStop '' 6 Stack Longs
'' ///////////////////////////////////////////////////////////////////////////////////////
'' // Shuts down the SDC driver running on a cog and returns the lock used by the driver.
'' ///////////////////////////////////////////////////////////////////////////////////////
  fat.FATEngineStop

PRI clrstr(strAddr, size)
' Clears string at strAddr
' -- also resets global character pointer (idx)
  bytefill(strAddr, 0, size)                            ' clear string to zeros
  idx~

PRI decf(value1, width) | t_val, field
'' Returns pointer to signed-decimal, fixed-width ("0" padded) string
  clrstr(@nstr, MAX_LEN)
  width := 1 #> width <# constant(MAX_LEN - 1)          ' qualify field width
  t_val := ||value1                                     ' work with absolute
  field~                                                ' clear field
  repeat while t_val > 0                                ' count number of digits
    field++
    t_val /= 10
  field #>= 1                                           ' min field width is 1
  if value1 < 0                                         ' if value is negative
    field++                                             ' bump field for neg sign
  if field < width                                      ' need padding?
    repeat (width - field)                              ' yes
```

```
      nstr[idx++] := "0"                                '   pad with space(s)
  return decstr(value1)

PRI decstr(value1) | div, z_pad
' Converts value to signed-decimal string equivalent
' -- characters written to current position of idx
' -- returns pointer to nstr
  if (value1 < 0)                                        ' negative value?
    -value1                                              '   yes, make positive
    nstr[idx++] := "-"                                  '   and print sign indicator
  div := 1_000_000_000                                  ' initialize divisor
  z_pad~                                                ' clear zero-pad flag
  repeat 10
    if (value1 => div)                                   ' printable character?
      nstr[idx++] := (value1 / div + "0")                '   yes, print ASCII digit
      value1 //= div                                     '   update value
      z_pad~~                                           '   set zflag
    elseif z_pad or (div == 1)                          ' printing or last column?
      nstr[idx++] := "0"
    div /= 10
  return @nstr

PRI Timer(RecTimeAddr, StoppedAddr) | Time, temp1
'starts a timer used to stop a timed recording.  RecTime is the rec length in seconds.
  Time := cnt
  temp1 := long[RecTimeAddr]
  repeat until temp1 =< 0 or long[StoppedAddr] == true
    waitcnt(Time += clkfreq)
    temp1--
    long[RecTimeAddr] := temp1
  long[StoppedAddr] := true
  waitcnt(clkfreq + cnt)
  cogstop(cogID)

PUB SDCardCheck
'' ////////////////////////////////////////////////////////////////////////////////
'' Tries to mount SD card.  If successful, returns true (-1), else returns an error code.
'' ////////////////////////////////////////////////////////////////////////////////
   result := false
   ifnot(fat.partitionMounted)
    fat.mountPartition(0)
   result := fat.partitionMounted
   unmount
   return

PUB unmount
  fat.unmountpartition

DAT
' /////////////////////////////////////////////////////////////////////////////////
'                       ADC Driver
' /////////////////////////////////////////////////////////////////////////////////
                        org     0
' ////////////////////Initialization////////////////////////////////////////////////
adcinit                 or      dira,cmask                                 'output CLK
                        or      dira,smask                                 'output CS

                        mov     recorderPointer, dataBlockAddress          '
                        mov     bxptr,           bxAddress
                        rdlong  recorderRate,    clocksPerSampleAddress     'Setup
timing.
                        mov     timeCounter,     recorderRate              '
                        add     timeCounter,     cnt                       '
                        mov     bxcounter, #4

' /////////////////////////////////////////////////////////////////////////////////
'                       Recorder
' /////////////////////////////////////////////////////////////////////////////////

outerLoop               mov     counter,         #256    ' 512Bytes/16Bits/1Ch = 256 Samp
```

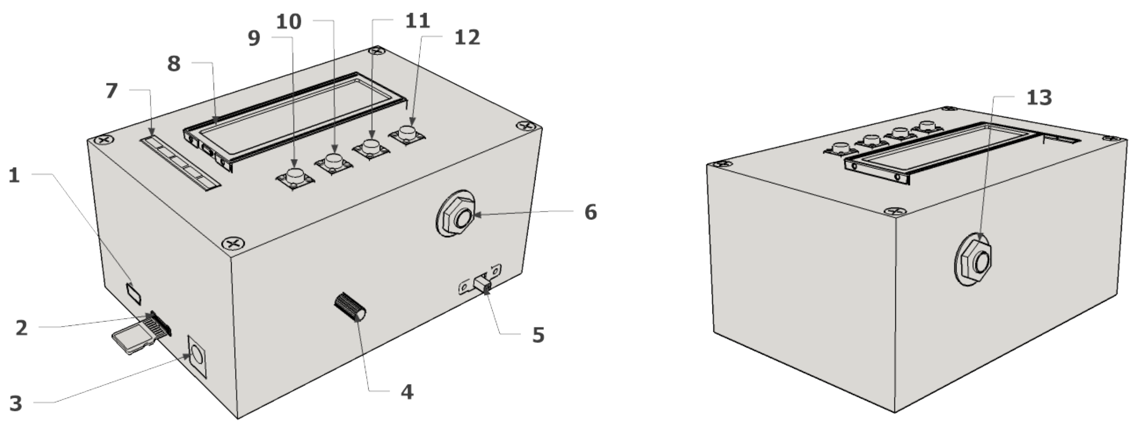
```
' //////////////////////Inner Loop/////////////////////////////////////////////////////////

innerLoop               waitcnt timeCounter,     recorderRate      'wait for next sample

main_loop               mov     command,#$10              'init command
                        mov     t2,enables                'get enables
                        mov     t3,#8                     'ready 8 channels

cloop                   shr     t2,#1           wc        'if channel disabled, skip

                        test    t2,#$80         wc        'channel enabled, get single/diff
                        muxnc   command,#$08
                        mov     stream,command

                        or      outa,smask                'CS high
                        or      dira,dmask                'make DIN/DOUT output
                        mov     bits,#20                  '(cs+1+diff+ch[3]+0+0+data[12])

bloop                   test    stream,#$20     wc        'update DIN/DOUT
                        muxc    outa,dmask

                        cmp     bits,#14        wz        'if command done, input DIN/DOUT
        if_z            andn    dira,dmask

                        andn    outa,cmask                'CLK low

                        or      outa,cmask                'CLK high

                        test    dmask,ina       wc        'sample DIN/DOUT
                        rcl     stream,#1

                        andn    outa,smask                'CS low

                        djnz    bits,#bloop               'next data bit

                        and     stream,mask12             'trim and write sample

                        wrword  stream,recorderPointer  'write data word to memory
                        rdlong  maxval, maxValueAddress 'get current maximum value
                        cmp     maxval, stream  wc
        if_c            wrlong  stream, maxValueAddress 'if stream > current max val
                        shr     stream, #1                'bx values < 2048 so divide by 2
                        wrlong  stream,bxptr              'write value to bx buffer for fft

                        add     recorderPointer, #2     'increment buffer pointers
                        add     bxptr, #4                        '

                        rdlong  recorderRate,    clocksPerSampleAddress     ' Loop.
                        djnz    counter,         #innerLoop                 '

' //////////////////////Outer Loop/////////////////////////////////////////////////////////

                        rdword  buffer,          callePointerAddress wz     ' Switch pntr
                        sumz    buffer,          #1                         '
                        wrword  buffer,          callePointerAddress        '
if_nz                   mov     recorderPointer, dataBlockAddress           '

                        sub     bxcounter, #1  wz
if_z                    call    #bxbuffer

                        jmp     #outerLoop                                  ' Loop.

bxbuffer                rdword  buffer,         bxcallePointerAddress wz    'flip bxpntr
                        sumz    buffer,         #1
                        wrword  buffer,         bxcallePointerAddress
if_nz                   mov     bxptr,          bxAddress
                        mov     bxcounter,      #4                          'reset bx cnt
bxbuffer_ret            ret                                                 '1024, 4 sets
                                                                            'of 256)
' ////////////////////////////////////////////////////////////////////////////////////////
'                       Data
```

```
' ////////////////////////////////////////////////////////////////////////////////////////
mask12                  long    $FFF
dmask                   long     1 << 26
cmask                   long     1 << 25
smask                   long     1 << 27
enables                 long     1                                 'CH0 only, single mode

' //////////////////////Addresses/////////////////////////////////////////////////////////

clocksPerSampleAddress  long    0
dataBlockAddress        long    0
callePointerAddress     long    0
bxcallePointerAddress   long    0
bxAddress               long    0
maxValueAddress         long    0

' //////////////////////Run Time Variables////////////////////////////////////////////////

buffer                  res     1
counter                 res     1
t2                      res     1
t3                      res     1
stream                  res     1
bits                    res     1
maxval                  res     1
bxcounter               res     1
recorderPointer         res     1
bxPtr                   res     1
recorderRate            res     1
timeCounter             res     1
command                 res     1

' ////////////////////////////////////////////////////////////////////////////////////////

                        fit     496

CON WAVFileHeaderSize = 44 ' DO NOT EDIT!

DAT WAVFileHeaderData ' DO NOT EDIT!

' ////////////////////////////////////////////////////////////////////////////////////////

  byte byte "RIFF" ' "RIFF" chunk header.
  byte long 0 ' "RIFF" chunk size = (fileSize - 8). Offset 4.
  byte byte "WAVE" ' File type.

  byte byte "fmt " ' "fmt " chunk header.
  byte long 16 ' "fmt " chunk size.
  byte word 1 ' Audio format.
  byte word 1 ' Nuber of channels.
  byte long 0 ' Sample rate.EDITED by setup function!
  byte long 0 ' Byte rate. EDITED by setup function!
  byte word 2 ' Block align.
  byte word 16 ' Bits per sample.

  byte byte "data" ' "data" chunk header.
  byte long 0 ' "data" chunk size = (fileSize - 44). Offset 40.


DAT AUTONAMEArray byte "000_OKC.WAV", 0
'default file name.  Change "_OKC" extension if desired.
' ////////////////////////////////////////////////////////////////////////////////////////

{{

//////////////////////////////////////////////////////////////////////////////////////////
//                           TERMS OF USE: MIT License
//////////////////////////////////////////////////////////////////////////////////////////
// Permission is hereby granted, free of charge, to any person obtaining a copy of this
// software and associated documentation files (the "Software"), to deal in the Software
// without restriction, including without limitation the rights to use, copy, modify,
```

```
// merge, publish, distribute, sublicense, and/or sell copies of the Software, and to
// permit persons to whom the Software is furnished to do so, subject to the following
// conditions:
//
// The above copyright notice and this permission notice shall be included in all copies
// or substantial portions of the Software.
//
// THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED,
// INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A
// PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
// HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF
// CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE
// OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
/////////////////////////////////////////////////////////////////////////////////////////
}}
```

**FullDuplexSerial.spin** – Unmodified code from Parallax, available in Propeller Tool 1.3.2

**Infrasound Detection Device**




1. Mini USB Connector – Used to program and/or power the device.
2. Micro SD Card Slot – Cards are inserted face down (metal contacts face up)
3. 2.1 mm Power Jack – Used to power device (6-9 VDC)
4. Gain knob – Controls amplifier gain (2-229x)
5. Power Switch
6. Output Jack – 3.5 mm mono output *** RANGE SPECS ONCE FINALIZED***
7. LED Strip – Displays input amplitude
8. Display – 2-line interface displays device instructions and status
9. Back button – Interface control – displayed as ←
10. Up Button – Interface control – displayed as ↑
11. Down Button – Interface control – displayed as ↓
12. Select Button – Interface control – displayed as **OK**
13. Input Jack – 3.5 mm mono input (0-2.5 VAC)

**To operate device:**

1. Slide power switch to ON position.
2. Display will illuminate showing `Starting`... and plays a startup tone. LED block will flash on and off.
3. Menu options will show:

The display has two lines. The top line will show information and status. The bottom line will show what button options are available. Press the UP and DOWN buttons to switch between the options. Press BACK in any menu to return to a higher-level menu. Press OK to select the option shown.

**Menu Options**

1. `Monitor w/ Alarm`
   Press OK to select. This will put the device in Monitor mode. If the signal from the input is a frequency below 40 Hz that exceeds the Alarm Threshold level, the alarm will sound and display lights will flash. Pressing BACK will return the user to the main menu and disable the monitor/alarm.
2. `Recording`
   Press OK to enter the Recording menu. Recordings are made in the WAV format. Filenames are auto-generated. The default filename is 000_OKC.WAV. Subsequent files will increment the three-digit number at the front of the filename. Two options are available:
   a. `Timed Recording`
      Press OK to select. This option allows the user to record the input signal for a specified amount of time. Pressing UP or DOWN in this menu switches the predefined time options ranging from 1 minute to 24 hours. When the desired duration is displayed, press OK to start the recording. The device will record until the time has elapsed or maximum file size has been reached, whichever is earlier.
   b. `Rec Until Stop`
      Press OK to select. The display will show Rec Continuous. This option can start a recording that will continue until either the OK button is pressed or maximum file size is reached, whichever is earlier. Press OK to start and stop the recording.
3. `Playback`
   Press OK to enter the Playback menu. Valid WAV file names will be shown on the display. Press DOWN to step through the files on the SD card. Press OK to start

playback of the selected file. While the file is playing, press DOWN to toggle between 1x and 10x playback speeds. Press OK to stop playback.

4. `Settings`
   Press OK to enter the Settings menu. Two options are available:
   a. `Sampling Freq`
      Press OK to select. This option allows setting the sampling frequency between 1000 Hz and 20000 Hz in 1000 Hz increments. Press UP and DOWN to switch between frequency settings. Press OK to set the frequency. This value is only used while the device is powered on. On reset, the sampling frequency reverts to default (1000 Hz.)
   b. `Alarm Threshold`
      Press OK to select. This option allows setting the amplitude threshold that will trigger the alarm for frequencies below 40 Hz. The valid range is between 10% and 90% of maximum amplitude. Press UP and DOWN to switch between threshold percentages in increments of 10%. Press OK to set the threshold level. This value is only used while the device is powered on. On reset, the alarm threshold value reverts to default (10%.)

**Setting Input Levels**

When monitoring or recording, the gain of the amplifier must be adjusted to account for different input levels and different input devices.

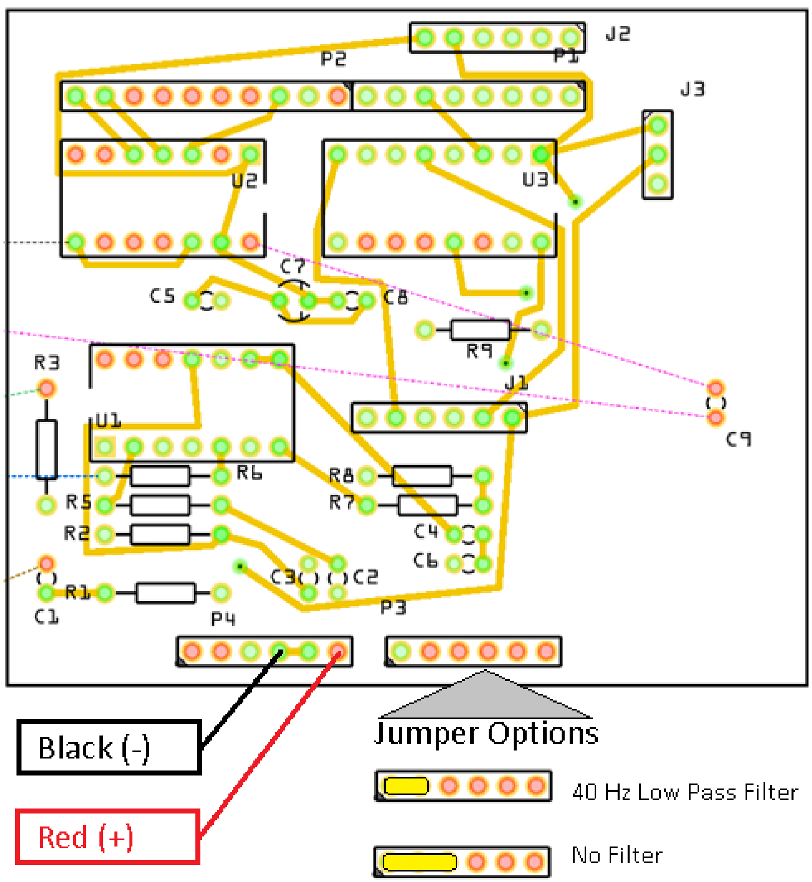
1. Put the device in `Monitoring w/ Alarm` or a `Recording` mode.
2. While in these modes, the 5-light LED bar will show the input signal level.
3. Provide an input signal of a type similar to the expected volume level of the recording. For example, if the device is a microphone and the expected volume level is similar to human speech, speak into the microphone at the approximate volume level.
4. Adjust the gain control knob while providing an input signal until the signal is consistently in the 3$^{rd}$ or 4$^{th}$ LED from the bottom. Reduce the gain if the 5$^{th}$ LED is lit constantly or often to prevent clipping the signal.
5. The gain level is now set.

**Changing Batteries**

Remove the 4 screws on the top cover. Carefully lift off the cover. The battery compartment is on the right face of the device. The device requires four AAA batteries.

**Connections**

There are three connections that could become unconnected if the cover is removed. Two of the connections are the + and – connections to the battery. The last connection is a jumper that connects the input signal to the microcontroller. The first option will connect through a low-pass filter (needed for infrasound detection.) The second option bypasses the low pass filter. Only one of these options can be connected at the same time. Ensure connections are in place before reattaching the top cover. Connections are made by pushing the jumper contact firmly into the female header slot (see figure below.)



VITA

Steven Marcus Bergren

Candidate for the Degree of

Master of Science

Thesis: A PORTABLE DEVICE FOR DETECTING INFRASOUND

Major Field: Electrical Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2018.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2001.

Experience:

Electronics Engineer, Dept. of Defense, Tinker Air Force Base, 2001-present