Smart Acoustic Whale Monitor

A Major Qualifying Project Report: submitted to the Faculty of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for
the Degree of Bachelor of Science of
Electrical and Computer Engineering

by

-------------------------------------
Matthew C Campbell

-------------------------------------
Ehsun Siddiqui

Advisor

-------------------------------------
Susan Jarvis

April 24, 2008
# Contents

1. Abstract ........................................................................... 1  
2. Introduction ...................................................................... 2  
3. Existing Technologies .......................................................... 3  
4. System Overview ................................................................. 4  
5. Theoretical Analysis .............................................................. 6  
   5.1 Preconditioning Filter ......................................................... 6  
   5.2 Moving Average ............................................................... 9  
   5.3 Automated Noise Reduction Process .................................. 10  
6. Slave Unit .......................................................................... 17  
   6.1 Communications Unit ......................................................... 17  
   6.2 DSP Microprocessor and Analog to Digital Converter ............. 19  
   6.3 GPS Unit .................................................................. 20  
   6.4 Whale Signal Identification ................................................ 21  
7. Master Unit ......................................................................... 22  
   7.1 Microprocessor ............................................................... 23  
   7.1.1 MSP430 ................................................................. 24  
   7.1.2 dsPIC ................................................................. 24  
   7.2 External ADC ............................................................... 25  
   7.3 Data memory ................................................................. 26  
   7.3.1 SPI SD Card Interface .................................................. 26  
   7.3.2 FAT Filesystem ......................................................... 27  
8. Interfaces and Technical Specifications .................................... 30  
   8.1 Communications Unit Interface - Ocean Controls HAC-UM12 .... 30  
   8.2 SD Card Interface ............................................................ 30  
   8.3 External Codec Interface - Si3000 ......................................... 31  
   8.4 GPS Interface - Trimble Lassen iQ ....................................... 31  
   8.5 Master-Slave Chip Interface ............................................... 32  
   8.6 Voltage Buffer Solution .................................................... 32  
9. Design Implementation and Testing ......................................... 35  
   9.1 Communications Unit - Ocean Controls HAC- ....................... 35  
   9.1.1 Wireless Module Subsystem Testing .................................. 37  
   9.2 SD Card Implementation and Testing ..................................... 38  
   9.2.1 Custom File System ..................................................... 39  
   9.2.2 SD Card Subsystem Testing ............................................ 40  
   9.3 GPS Implementation and Testing ......................................... 41  
   9.3.1 GPS Subsystem Testing - Trimble Lassen iQ ....................... 41
**List of Figures**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Graphical Representation of System Concept</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Slave Unit Block Diagram</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Slave Unit Block Diagram</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>Preconditioning Filter</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>Moving Average Output with 100 Sample Window</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Example Signal 1 - Before Filtering</td>
<td>10</td>
</tr>
<tr>
<td>7</td>
<td>Example Signal 2 - Before Filtering</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Low Filter Frequency Response</td>
<td>12</td>
</tr>
<tr>
<td>9</td>
<td>High Filter Frequency Response</td>
<td>12</td>
</tr>
<tr>
<td>10</td>
<td>Example Signal Before Filtering</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>Example Signal After Filtering</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>Low Filter Applied Spectrogram</td>
<td>15</td>
</tr>
<tr>
<td>13</td>
<td>High Filter Applied Spectrogram</td>
<td>15</td>
</tr>
<tr>
<td>14</td>
<td>Slave Unit Block Diagram</td>
<td>17</td>
</tr>
<tr>
<td>15</td>
<td>Ocean Controls HAC-UM96</td>
<td>18</td>
</tr>
<tr>
<td>16</td>
<td>dsPIC30f4013 PDIP Package</td>
<td>20</td>
</tr>
<tr>
<td>17</td>
<td>Lassen iq GPS Module</td>
<td>21</td>
</tr>
<tr>
<td>18</td>
<td>Master Unit Block Diagram</td>
<td>22</td>
</tr>
<tr>
<td>19</td>
<td>Sample SD Card Interface to dsPIC</td>
<td>28</td>
</tr>
<tr>
<td>20</td>
<td>Breadboarded example of SPI interface between dsPIC and SD card</td>
<td>29</td>
</tr>
<tr>
<td>21</td>
<td>3.3V to 5V Interface Schematics</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>Maxium-IC MAX3390 Level Translator Concept</td>
<td>34</td>
</tr>
<tr>
<td>23</td>
<td>Maxium-IC MAX3390 Level Package</td>
<td>34</td>
</tr>
<tr>
<td>24</td>
<td>Wireless Module Hardware Interface</td>
<td>35</td>
</tr>
<tr>
<td>25</td>
<td>SD Card SPI Pin Configuration</td>
<td>38</td>
</tr>
<tr>
<td>26</td>
<td>SD Card Example Interface Schematics</td>
<td>95</td>
</tr>
</tbody>
</table>
## List of Tables

1. Wireless Module Comparison ........................................ 19
2. dsPIC30F4013 Specifications ........................................ 20
3. Master Unit Processor Comparison ................................. 23
4. MSP430F2013 Specifications ........................................ 24
5. Si3000 Specifications ................................................ 25
6. HAC-UM12 Connections .............................................. 35
7. Wireless Protocol ...................................................... 36
8. Wireless Protocol - Event Description Byte .................... 36
10. Wireless Protocol - Data Response Packet ....................... 37
12. SD Card SPI Pin Connections ...................................... 39
13. Identified Signal Packet ........................................... 40
14. GPS Module Pin Connections ...................................... 41
15. Parts List ............................................................ 45
Abstract

The purpose of this project is to monitor whale vocalizations in an efficient manner. This involves several units monitoring sounds beneath the ocean surface using hydrophones, and analyzing them using a DSP chip. These units are designed to have a marketable version that could be located in buoys. Whale vocalizations, or signals of interest, will be identified and information about each buoy’s identification will be recorded on a single master buoy. The communication between the multiple units is handled through long range radio frequency radio units. Metrics such as time of arrival, duration, GPS time and GPS location are stored on the master unit for later analysis such as further species identification or location triangulation.
2 Introduction

Some whale monitoring systems in use today require a good deal of maintenance and frequent visiting to retrieve data. Those that record constantly must frequently be tended to, to clear up memory and to supply power. Additionally, each buoy’s data must be analyzed individually and then later compared with the results of the buoy network as a whole. After manual analysis, it is decided what species of whale was present.

However, we propose a smart monitoring system that can not only record audio data selectively and intelligently but also, this unit offers longer time-on-station in the ocean without the need to swap data storage or replace batteries. Additionally, as the data is collected in an intelligent manner the analysis process is greatly eased with more time spent on analysis on signal of interest and less on simply identifying signals in a sea of ambient noise.

This system involves several components that could be combined to fit in a buoy. The major components include a GPS unit, microprocessor, communications module, SD card slot and hydrophone. The following sections describe these components and their relations and purpose to each other in greater detail.
3 Existing Technologies

There are several such undersea monitoring technologies. One such is Sound Surveillance System (SOSUS) which is developed by the US government. This system is an array of hydrophones placed on the ocean bottom in key locations. This was at first developed as a defensive in war from naval attacks, but is now used by whale researchers for the recording of whale sounds and localization of whales. SOSUS has been developing since the 1950’s, and is now remotely accessible by computers for detailed and quick acoustic data analysis.

Another technology is the T-Pod. This was developed specifically for the monitoring of cetaceans. This piece of technology is a cylinder that is lowered into the ocean to monitor the nearby cetaceans and identify the species.

Although SOSUS has a broader range of monitoring than our application will, we will incorporate species identification. Also, we imagine that access and use of SOSUS is relatively expensive, whereas what we plan to develop is relatively inexpensive. The T-Pod does not calculate the coordinates of the source of the whale vocalization.
4 System Overview

Before understanding the details of implementation a grasp of the high level function must be obtained. As with current technologies this system is deployed in a network of buoys spread out over a section of ocean. The smallest deployable network that still is useful would contain three buoys. One and only one master buoy must be present in every network to coordinate the system. Additionally, two or more slave buoys must also be deployed along with the master to obtain any meaningful positioning data.

Figure 1 shows a graphical representation of a deployed network identifying a whale vocalization. The white lines represent the vocalizations coming from the whale and being picked up by the hydrophones connected to buoys. The three buoys on the left are slave buoys while the buoy furthest right represents a master. When the slaves get a successful identification it caches this in memory along with the GPS time and location when it was detected. The master unit continuously polls the
slave buoys wirelessly and downloads all information about identifications since its last download. This data is then recorded to the external mass storage on the master unit. The master unit contains an implementation of a slave unit in addition to the master unit. This slave implementation is almost identical to the others with just a few changes. The wireless communications is discarded for a direct wire connection. In addition to reporting GPS time and location of signal detection, this information is also used to by the slave to toggle the recording of high quality audio data on the master unit.
5 Theoretical Analysis

As seen previously, the project requires a lot of signal processing and analysis of whale vocalizations. Due to budget limitations, it would not be possible to collect live whale data from the ocean itself. Therefore, the dataset of whale sounds was taken from sets publicly posted by the Defence R&D of Canada\(^1\). It would be impossible to implement our entire project without having to use software simulations of a real-time environment. The theoretical analysis of whale vocalizations was done using MATLAB. It was convenient and quicker to implement and test algorithms in MATLAB rather than running it through the hardware without the MATLAB testing. Running an analysis can also expose errors with our algorithm implementation. These errors might go unnoticed, or might be harder to debug if only tested with the hardware, yet are much easier to understand and fix with software implementation. As an end result, the analysis quickly produced filter coefficients and we were able to automate it to run a user-friendly interface for wave file analysis.

The dataset from DRDC consisted of approximately 80 different whale vocalizations, each with a sampling rate of 1200 Hz. However, these wave files were too low in amplitude to be heard when played back even with the PC’s volume at maximum. MATLAB could only play these files when they were amplified by a factor of 10. Therefore, a loop was run to amplify and save each of these wave files so they could be adequately used. In the process, the files were resampled at 8000 Hz. For the remainder of the project, the files used were all sampled at 8000 Hz simply due to MATLAB’s default resample rate.

5.1 Preconditioning Filter

Before anything could be done with our signal, we needed to apply a preconditioning filter to eliminate prominent large low-frequency components that existed in all our sample signals. This can be seen in the frequency and spectrogram plots below.

It might seem at first glance that the low frequency component with an amplitude of 10 dB is our frequency of interest. However, this is not the case. The frequencies below 300 Hz are in fact just noise components with a very high amplitude. The actual whale vocalizations are almost completely obscured by the noise in nearly every data file. The signal of interest is often masked, but becomes more evident through a spectrogram plot.

\(^1\) http://www.atlantic.drdc-rddc.gc.ca/newsevents/workshop/workshop_03_e.shtml
5 Theoretical Analysis

Fig. 2: Slave Unit Block Diagram

Fig. 3: Slave Unit Block Diagram
It is quite clear in this image that our real signal of interest exists at frequencies around 350 Hz and above. It is also clear that there are very low frequencies with great amplitude that spread across the entire time span of the signal. This is the noise component of the signal that needs to be filtered out with a preconditioning filter.

After more detailed analysis of the set of 80 whale vocalizations, we chose a high-pass IIR Butterworth filter with a cutoff frequency of 300 Hz as our preconditioning filter. Before selecting the filter, it was decided that most of our noise content was comprised of frequencies below 300 Hz. Shown below is the magnitude and phase plot for the filter.

Fig. 4: Preconditioning Filter
5.2 Moving Average

As discussed in previous sections, our system is designed to record only the signals of interest. To accomplish this, we made use of a moving averager. The moving averager estimates the average audio amplitude and computes a threshold value based on a window of samples. The average of these accumulated samples is taken, and the value is offset by 0.075 to create the current threshold for the incoming sample. The recording can be made more or less by adjusting this offset. The moving average of a particular whale vocalization can be seen below along with the equation.

\[
S_i = \frac{1}{n} \sum_{j=i}^{i+n-1} |a_j|
\]  

(1)

Fig. 5: Moving Average Output with 100 Sample Window
The moving average threshold is in red, whereas the incoming audio signal is in blue. Any signal exceeding this threshold in the time domain is considered a signal of interest. Such a signal cues the system to record for a pre-determined time period.

5.3 Automated Noise Reduction Process

This part of the analysis is the most difficult to implement. The purpose of this process is to be able to choose an appropriate bandpass filter to apply to the incoming signal, so that all frequencies of interests are preserved, yet all other components in the frequency domain are rejected. To accomplish this, the idea was to implement multiple filters at different cutoff frequencies in parallel on the same input. The resulting signal with the greatest signal energy would be considered our best-matched signal in which the frequency of interest would ideally have been preserved. To depict the purpose of this, shown below are two different signals of interest, each in a different frequency band.

![Image](image-url)  
**Fig. 6: Example Signal 1 - Before Filtering**
The preconditioning filter will indeed reject the low frequency noise. However, the signal will not be optimally filtered for maximum noise reduction. The first data file shows four whale clicks, each ranging between 300 Hz to 2kHz. The second signal however shows a signal at a much higher frequency of 2.5kHz to 3.8kHz. Ideally, each signal would be filtered exactly to pass only those bands that contain signals of interest. Our goal is to roughly implement this in firmware with a matched-filtering concept.

Two issues developed with this implementation. The first is the question of how many filters we would run in parallel. The second issue was that in the resulting signals, the one with the highest signal energy was not necessarily our signal of interest.

We had previously noted that most whale vocalizations, if ideally filtered for maximum noise-reduction, yielded a requirement of a bandpass filter with a width of about 1.8KHz (given our 8KHz sampling frequency). A bandpass filter below this requirement risked rejecting frequencies containing our signal of interest. Shown below is a signal with an approximate 1.8kHz bandwidth for the signal of interest.

Fig. 7: Example Signal 2 - Before Filtering
This file would ideally be filtered between 2kHz and 3.8kHz.

Due to our 4 KHz bandwidth, we could only have a set of two filters. The first filter passed frequencies between 400 Hz and 2.2kHz, the second passed between 2kHz and 3.8kHz. The overlap between the two filters is desirable, in the event that a high frequency signal might dip down below the usual cutoff, or a low frequency signal rising above the cut off.

In terms of our second concern with being able to accurately determine which resulting signal contained our frequency of interest, after further testing, we discovered that the signal with the lowest energy was in fact our signal of interest. This was due to the fact that the noise component was much higher in amplitude (sometimes by a factor of 30), that our signal of interest was often shadowed. This yielded a much greater signal energy for noise components than for our filtered whale vocalizations.

Although this rule did not work all the time, it was applicable to almost all the whale vocalizations we dealt with. The following images depict the process of this implementation. When the file is first read in, the entire spectrogram shows a very high frequency audio signal.
The preconditioning filter is then immediately applied.

Fig. 10: Example Signal Before Filtering
After this, two filters are applied to the preconditioned signal. The first is the low frequency bandpass filter, passing frequencies between 400 Hz and 2.2 kHz, and a high frequency bandpass filter, passing frequencies between 2 kHz and 3.8 kHz.
Fig. 12: Low Filter Applied Spectrogram

Fig. 13: High Filter Applied Spectrogram
It can be seen that our actual signal of interest was passed with the high frequency bandpass. However, the calculation in the code (seen in the Appendix), yields a lower signal energy for the signal of interest. This is due to the fact that ocean ambient noise exists largely at the low frequency levels even after applying our preconditioning filter, as can be seen in Figure 10. This essentially means that after both the low pass and high pass filters are applied, the low-passed signal will necessarily have more energy than the high-passed signal. To accommodate for this, we would have to normalize both frequency bands to account for this excess noise energy.
The slave unit is a simple device consisting of a few components as possible. Rather than recording any audio data this unit continuously analyzes incoming audio to identify signals of interest. The time and location of these detected sounds is determined from the on-board GPS chip. This information is downloaded by the master buoy over the wireless connection and recorded on mass storage there. The main design requirements for the slave unit are simplicity, cost and power consumption.

6.1 Communications Unit

There were several choices for the communications between the slave and master units. Simple radio frequency modules can be used, which operate at frequencies of about 900 MHz. A price quote for these modules was never obtained, but it was accepted to be well over the typical MQP project budget, and for this reason was...
never considered a viable solution. Another option considered was to use simple Wireless LAN (802.11) modules to transmit information. WLAN operates at higher frequencies of 2.4 GHz. The desired range required for this project is difficult to reach using these modules. Another option considered was Bluetooth. Bluetooth uses the same frequencies as WLAN, but is easier to implement and the least expensive option of those considered. In any of the three options mentioned, an antenna will have to be appended to the module to provide for proper transmission. There are distortions due to the ocean surface. A comparison of features and cost can be seen in table 1.

The HAC-UM12 Module from Ocean Controls was chosen as the best available solution and the unit currently implemented in the system. Although this module is slightly more expensive than the Bluetooth solution, it operates at the minimum range for this project of 1km. Additionally this unit was chosen for its simple communications interface. It offers two different serial communications ports, RS-232 UART compatible and RS-424. This unit is used as a serial to wireless converter. Hardware error checking and correction are handled in hardware removing the need to implement these features in software. The simple data-in, data-out nature of this module proved to be a great aid in reducing the complexity of implementing the wireless interface. The unit is available in two baud rates, a 1200 baud version and a 9660 version. As only small amounts of data are sent over the wireless link the 1200 baud version was selected as it offers nearly double the range of the 9660 baud version.

Fig. 15: Ocean Controls HAC-UM96
### Module Comparison

<table>
<thead>
<tr>
<th>Module</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ocean Controls: HAC-UM96</td>
<td>Low power consumption</td>
<td>Greater than $60</td>
</tr>
<tr>
<td></td>
<td>High data rate</td>
<td>500m range</td>
</tr>
<tr>
<td>Ocean Controls: HAC-UM12</td>
<td>Low power consumption</td>
<td>Greater than $60</td>
</tr>
<tr>
<td></td>
<td>1000m range</td>
<td></td>
</tr>
<tr>
<td>Ezurio (bluetooth)</td>
<td>Low power consumption</td>
<td>Very low range (250m)</td>
</tr>
<tr>
<td></td>
<td>High data rate</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Less than $50</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Very easy interfacing</td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 1: Wireless Module Compareison**

### 6.2 DSP Microprocessor and Analog to Digital Converter

In the slave unit there were several concerns with the DSP chip. The design requirements call for a low power, inexpensive chip that is capable of performing real-time analysis of incoming signals. Several options were considered including an ARM based microcomputer system, a full-fledged DSP chip and Microchip’s dsPIC line of microprocessors. The ARM and full DSP chip require more external components to have a fully functioning system which raises cost and the difficulty of implementation. The dsPIC line of microprocessors are an all-in-one system on chip design that not only has all components including program and data memory on chip but also has a build in analog to digital converter. Unfortunately the dsPIC line only offers chips with built in 10 or 12 bit analog to digital converters. A 16 bit converter would be ideal. The 16 bit converter would provide much more dynamic range in the samples and thus allowing greater ability to distinguish whale signals from the ocean background noise. For this MQP the 12 bit converter will be sufficient for proof of concept and will greatly ease the development and implementation process. In the future an external 16 bit converter could be used with the dsPIC to achieve the greater resolution with minimal change to the overall design. Many dsPICS have a hardware external serial CODEC interface that is compatible with many discrete analog to digital converter solutions.

The specific dsPIC selected is less important as most members of this chip family have the required hardware. It was desirable to select a chip that is available in PDIP packaging for easy prototyping. This narrowed the dsPIC family down to eight offerings: two 40MIPS (millions of instructions per second) chips and six 30MIPS chips. While it was desired to go with a faster chip, the 30MIPS chips are offered with more general IO pins, two UART modules as well as far more RAM and program.
memory. After evaluating the options the dsPIC30F4013 was selected.

<table>
<thead>
<tr>
<th>CPU Speed</th>
<th>30MIPs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Flash</td>
<td>48 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>2048 B</td>
</tr>
<tr>
<td>Data Flash</td>
<td>1024 B</td>
</tr>
<tr>
<td>GPIO</td>
<td>30</td>
</tr>
<tr>
<td>ADC</td>
<td>13 12-bit</td>
</tr>
<tr>
<td>Interface</td>
<td>2 UART, 1 SPI, 1 I2C, 1 CAN, 1 Codec</td>
</tr>
<tr>
<td>Package</td>
<td>40 pin PDIP</td>
</tr>
<tr>
<td>Pricing</td>
<td>Volume $3.98, samples available</td>
</tr>
</tbody>
</table>

Tab. 2: dsPIC30F4013 Specifications

Fig. 16: dsPIC30f4013 PDIP Package

6.3 GPS Unit

In order to be able to triangulate the position of the whales, the exact time of arrival of the whale signal must be measured at three or more separate locations must be recorded. GPS provided an ideal solution for this. Several GPS units were researched as candidates for this system. Most GPS modules offer a simple serial interface which can be easily interfaced with the dsPIC though UART. Unfortunately, there were no GPS modules found for under fifty dollars. This would place the GPS module as one of the most expensive part of the slave unit by far. This is not an unreasonable cost however and is considered a viable option.

The GPS unit selected was the Lassen iQ GPS Receiver (P/N: 54854-00 rev A). The Lassen iQ was selected for availability and price. This unit has a simple serial interface and operates at TTL levels and doesn’t require the overhead of any other
protocol. The unit can be interfaced directly with the hardware UART unit of the dsPIC directly.

![Image of Lassen iq GPS Module](image)

**Fig. 17: Lassen iq GPS Module**

### 6.4 Whale Signal Identification

There are three characteristics of the input signal that we will use for our analysis. First, an adaptive threshold is calculated using a moving average to identify signals from background noise based on amplitude. A whale vocalization exceeds the amplitude of the average significantly. At that point, the signal will be identified by the slave units and recorded on the master unit. The frequency content will also be analyzed to determine the type of whale and reduce the amount of false positives. Implementation will be done in a three tier process. The first tier will be simply to identify signals of interest by amplitude to avoid constantly recording background noise. The second tier will be to analyze the signals’ frequency content by passing them through a series of bandpass filters. The final tier is to analyze signals against known whale patterns in the time and frequency domain. The final tier is a challenging endeavor and is not necessary to be performed on the units. This processing could be done on a computer after data has been collected. In the final implementation of this system, the third tier was considered too complicated while offering little gain in the systems viability. For this reason it was never implemented.
7 Master Unit

In a deployed network there must at minimum two slave units transmitting data to a single master unit. The master unit has an implementation of the slave unit on board to handle signal identification and GPS data on this buoy. This slave implementation functions the same as any other slave unit with the exception that data is sent to the master CPU over a physical serial line rather than a wireless link. The slave implementation on the master board also notifies the master CPU when to begin and end recordings of whale signals. The master unit also polls the network of slave buoys and downloads information about whale signals that have been identified since the last download. This data includes GPS time and location. This information is all saved along with the audio recording to external mass storage for later analysis. With this information triangulated positions of the whale signals can be calculated later on a computer.
7.1 Microprocessor

The master unit is capable of doing everything that the slave unit can do. In addition to these functions the master unit is responsible for managing the external mass storage, wirelessly communicating with every slave buoy in the network and recording audio of signals. To manage these extra tasks a second microprocessor is needed for the master station. There were several options considered for this purpose. The first possibility was to continue with the PIC line and use any of the 16 bit microprocessor options. PICs are a low power option, however they lacks in speed and available on-board peripherals. Another option considered was to design an embedded system around an ARM processor. ARM processors are powerful embedded options but do not meet the design requirements of this application such as low power consumption. Also ARM chips lack on board 16 bit A/D converters for high quality recoding. Another option researched was the MSP430. The MSP 430 is a low power, highly featured 16bit microprocessor. Also, there are models of the MSP430 available with on-board 16bit Sigma Delta A/D converters. This A/D converter is exactly what is needed to obtain the high quality audio recording for the master unit.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pic</td>
<td>Low power consumption</td>
<td>Not a powerful CPU</td>
</tr>
<tr>
<td></td>
<td>Low cost</td>
<td>No on-board 16bit A/D converter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Lack of on-board peripherals</td>
</tr>
<tr>
<td>ARM</td>
<td>Powerful CPU</td>
<td>Not as power efficient</td>
</tr>
<tr>
<td></td>
<td>Capable of many general</td>
<td>Much more complicated system design</td>
</tr>
<tr>
<td></td>
<td>computing tasks</td>
<td>No on-board 16bit A/D converter</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Higher cost</td>
</tr>
<tr>
<td>MSP430</td>
<td>Low power consumption</td>
<td>Small memory sizes</td>
</tr>
<tr>
<td></td>
<td>Inexpensive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Available with on-board</td>
<td></td>
</tr>
<tr>
<td></td>
<td>16bit A/D converter</td>
<td></td>
</tr>
</tbody>
</table>

Tab. 3: Master Unit Processor Comparison
7.1.1 MSP430

When considering versions of the MSP430 that could be used in the project there were several criteria to consider. The most important was the packaging. For ease of prototyping a PDIP package was required. Texas Instruments offers sixteen different models with an integrated 16bit sigma delta A/D. Of these sixteen only two versions are available in PDIP packaging. The two chips are identical other then their clock speed. They are available in 8MHz and 16MHz. In the interest of future expansion and the requirements of the chip, the faster unit is the obvious choice. The best MSP430 solution is the MSP430F2013. One disadvantage for all MSP430s that contain the 16bit A/D converter is the small size of RAM (128 bytes). This is extremely restrictive as this chip must buffer not only audio for high quality recording but incoming messages from slave units as well. With the size of these two buffers and the size needed for normal variable usage 128 would at the very least be a challenge if not impossible to work with.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>16MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flash</td>
<td>2 KB</td>
</tr>
<tr>
<td>RAM</td>
<td>128 B</td>
</tr>
<tr>
<td>GPIO</td>
<td>10</td>
</tr>
<tr>
<td>ADC</td>
<td>16-big sigma delta</td>
</tr>
<tr>
<td>Interface</td>
<td>USI (SPI or I2C)</td>
</tr>
<tr>
<td>Package</td>
<td>12 pin PDIP</td>
</tr>
<tr>
<td>Pricing</td>
<td>Volume $1.65, samples available</td>
</tr>
</tbody>
</table>

Tab. 4: MSP430F2013 Specifications

7.1.2 dsPIC

After considering the MSP430 as a solution and finding that it would likely not suffice the dsPIC line was considered an alternative. One of the strongest points of the MSP430 is that it is available with a 16bit sigma delta A/D converter. However after further research the MSP430 versions that include the 16bit A/D converter all had very limited RAM and program memory. The best chip selected only had 128bytes of RAM. With that small amount of memory it would be near impossible to carry out all the necessary functions of the master unit. After realizing this other chips were looked into. However, in the end the same dsPIC used for the slave unit proved to be powerful enough to perform all these tasks. The dsPIC has 2048 bytes of RAM, far more then the MSP430 and while enough to preform all
tasks for the master unit. The dsPIC selected also has a full set of I/O interfaces including SPI and UART which are crucial for communication between all devices. With this decision the master unit has two dsPIC: one for the master and one in the on board slave implementation. As stated before, the slave and master chips will not communicate over wireless since they are on the same board. The two dsPIC will be wired together with a UART serial interface. The same dsPIC model, the dsPIC30f4013, was chosen as it meets the requirements and using the same chip will simplify software development and hardware interfacing. The specifications for the dsPIC30f4013 can be seen in the Slave Unit section in Table 2.

7.2 External ADC

For the audio data recording in the master unit the possibility of using an external 16 bit A/D as opposed to the internal 12 bit A/D was considered. There are many ADC that can interface with both MSP430 and the dsPIC in a clean and efficient manor. After research the Si3000 16-bit ADC, which is aimed at telecommunications, was chosen as the high quality ADC. This chip offers not only high quality analog to digital conversion, but also low power consumption. The interface for sending digital data to a chip uses a standard CODEC interface that is present in many DSP chips and modems. This serial interface uses only four wires and keeps the overall simplicity of the project. However, for the scope of this MQP the internal 12 bit A/D was used to keep design simple. The internal 12 bit A/D is sufficient for proof of concept. The external 16 bit A/D could be added at a later date with minimal hardware and software changes. Additionally, this A/D could be used on the slave units to improve their dynamic range of signal detection.

<table>
<thead>
<tr>
<th>ADC ad DAC bit depth</th>
<th>16-bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of ADC in</td>
<td>2</td>
</tr>
<tr>
<td>Number of DAC out</td>
<td>3</td>
</tr>
<tr>
<td>Sampling Rate</td>
<td>12.000 ksps max</td>
</tr>
<tr>
<td>Supply voltage</td>
<td>3.3V to 5V</td>
</tr>
<tr>
<td>Interface</td>
<td>Serial</td>
</tr>
<tr>
<td>Package</td>
<td>16-SOIC</td>
</tr>
<tr>
<td>Price</td>
<td>$1.57</td>
</tr>
</tbody>
</table>

Tab. 5: Si3000 Specifications
7.3 Data memory

Large amounts of memory are needed to store the audio and GPS data. Rather than using more expensive embedded memory chips that would require the designs of a retrieval interface a memory card solution was used. Secure digital memory cards are inexpensive (4GB for approximately $40), easy to interface with and provide an easy way of retrieving data by simply swapping cards. SD cards can be written to with the FAT 32 file system which can be read by most current computer operating system. Additionally using the SPI interface, only four wires are needed to control the SD card making for easy implementation.

With dsPIC microprocessor there was the possibly of compressing the audio data before storing to the SD card. Several GNU GPL licensed algorithms could be used including GZIP, BZIP2 and FLAC. FLAC would be the most desirable as it is a lossless compression algorithm with the specific purpose for compressing audio which would offer the best compression ratios. However with gigabytes of data storage and the selective recording, compression may prove to be unnecessary. In the future compression could be fully implemented in software requiring no hardware changes, only a firmware update.

With the dsPIC chosen for this project, it was be beyond the chips capabilities to implement FAT32. With the FAT32 implementation not possible, data is stored in raw dumps to the chip using an extremely rudimentary block based file system (see section 9.2.1). This requires an extra program on the PC used to post-process the data, but this is considered to be an acceptable loss as it does not reduce overall system functionality.

7.3.1 SPI SD Card Interface

Secure digital standards allow for two mode of access to cards. The first is a more complicated parallel interface that is not openly documented. The second interface is implemented over SPI (serial peripheral interface). Full documentation of commands and data transfer are available on the Internet\(^2\). For this project the SPI interface is used because of its better documentation and simpler interface. With the dsPIC clocked at its maximum speed of 30MHz the SD card is not able to keep up. When an SD card is in SPI mode it can be clocked up to 25MHz maximum. Some specific SD cards can operate higher, but there is no guarantee of stable operation for all cards. For this project the SPI unit’s clock is set with a 2:1 clock divider internally on the chip. This requires no extra hardware and the SPI will run at 15MHz which

still provides plenty of throughput for data to the SD card.

SPI connections are made with only four wires; a clock, two data lines and a slave select. The simplicity of connecting only four wires is appealing from the design standpoint. The dsPIC selected for this project has built in hardware SPI modules. This also aided the rapid and easy implementation of SPI to the SD card. Much less code was needed to initialize and access the SD card using the built in SPI module of the dsPIC. Additionally there are several GNU GPL licenced projects that use SPI to access SD cards from various PIC microcontrollers. These projects include schematics and C code that aided in developing the interface for this project. One of these such projects is a simple voice recorder using the exact dsPIC model used in this project\(^3\).

One other small issue to consider with the SD card interface is the operating voltage. To achieve the full clock speed of the dsPIC it must run at TTL level 5v. SD cards are designed to work at a nominal voltage of 3.3v. Applying 5v to an SD card would not only malfunction but would likely damage the SD card. To handle this a 3.3v regulator is used to provide power to the SD card. The four SPI lines are at 5v and 3.3v for the dsPIC and SD card respectively. To handle this conversion a logic chip, such as the 74VHC08, was considered. The 74VHC08 can run the output at a different voltage from the input and in the datasheet it is cited as a useful chip for interfacing 5v/3v dual voltage systems. Although this chip is an AND gate the inputs can be tied together effectively simulating a buffer. The 74VHC08 is a quad two input AND gate which makes it ideal as only one chip would be needed for all four lines of the SPI interface. In the final design a discrete logic translator chip was used (see section 8.6).

### 7.3.2 FAT Filesystem

Although not used, it is worth documenting the possibilities of a FAT filesystem in the future as this would only require a firmware update. There are currently four major FAT variants; FAT12, FAT16, FAT32 and VFAT. FAT12 has an extreme size limitation of 32MB. FAT16 is limited to 2GB. Finally FAT32 can handle filesystems up to 8TB in size. There may not be enough memory on the dsPIC to implement FAT32. If this is the case FAT16 would be an acceptable alternative. Although SD cards are available up to 8GB and 16GB, being limited to 2GB would not be fatal to the project. Given that this device selectively records, this device could potentially still spend several days at least in the ocean. At 8kHz sampling with 16bit resolution 1.55 days of continuous audio can be stored. However since the device is selectively

\(^3\) [http://www.k9spud.com/sdcard/](http://www.k9spud.com/sdcard/)
Fig. 19: Sample SD Card Interface to dsPIC

recording audio data, the time-on-station in the ocean would be far more than this.
Fig. 20: Breadboarded example of SPI interface between dsPIC and SD card
8 Interfaces and Technical Specifications

8.1 Communications Unit Interface - Ocean Controls

HAC-UM12

The communications module used operates at a carrier frequency of 433MHz. It is based on GFSK modulation and the BER is around $10^{-6}$ to $10^{-5}$. The maximum range for this module is about 1000m at a rate of 1200bps. The module provides three interface protocols through two hardware serial ports. COM1 uses TTL level UART interface, and COM2 uses either RS-232 or RS-485. The transmitting current for this device is 40mA and the receiving current is 30ma. The sleep current is 20mA. It handles forward error correction and error detection in hardware alleviating the need to implement these in software.

The dsPIC is connected to the module through the UART interface (COM1). The connections to the COM1 port will be made from pins on the dsPIC using simple wires. There will be no additional circuitry in between these components, due to the fact that both devices use TTL voltage levels. No additional overhead in protocol for controlling the unit is necessary as it transparently transmit byte for byte the data is receives on the COM port.

8.2 SD Card Interface

The SD card communicates with the master dsPIC over SPI as stated earlier. Operating the SPI interface above 25MHz has been shown to cause unreliable performance. By configuring the prescale clock to 2:1 for the SPI port the interface will run at an acceptable 15MHz with the dsPIC core still clock at its maximum 30MIPS. However in the final design the dsPIC was run at 15MIPS and clock pre-scaling was unnecessary.

SD cards are designed to operate at a nominal voltage level of 3.3V. Although in the projects current implementation the dsPIC runs at 3.3v, research was done to devise a method of interfacing the 3.3v and 5v systems. If the dsPIC is run at 5v, all signals between the dsPIC and the SD card must be put through a voltage buffer to achieve the correct voltage conversion. Two solutions were considered. The first was the use of a simple resistor network to achieve the correct voltage change. Unfortunately the resistor networks were not able to keep the high speed digital data intact at 15MHz. The second solution was to use voltage buffer chips. The 74VHC logic family of chips is an ideal choice for this. A 74VHC08 quad two input and gate

\[^4\] http://www.k9spud.com/sdcard/
chip could solve the problem in a single chip. Additionally there are tristate buffer chips in the 74VHC family that would also serve this purpose and allow for easier multiplexing of the SPI interface if necessary. Unfortunately these chips have only been found in SSOP packaging and not in PDIP packaging. This greatly increases the effort needed to prototype this system. The complexity of prototyping with many SSOP devices led to the research into a third possibility, a discrete IC solution. A Maxim-IC chip was found that offers four high speed bidirectional buffers (see section 8.6). If the system was ever extended to have the dsPIC running at 5v this would be the chosen solution.

8.3 External Codec Interface - Si3000

In the master unit the possibility of an external 16bit CODEC was considered for greater quality when recording audio data for storage on the SD card. A standardized interface for using CODEC chips and DSP chips together is available on the Si3000 CODEC used. The dsPIC supports this interface through the use of a built in DCI module. The DCI module (data conversion interface module) is very similar to the SPI interface used for the SD card. It is a serial interface and requires only four wires to operate. With the wires correctly attached to the DCI pins of the dsPIC all that is required is to properly setup the chip for use of this interface through software. Several projects have successfully paired the dsPIC line of chips with the Si3000 voiceband codec chip in this way\(^5\). This was never implemented or tested due to the internal 12 bit A/D being sufficient and the complexity of prototyping with the SSOP CODEC chip.

8.4 GPS Interface - Trimble Lassen iQ

The GPS module offers three different serial interfaces for communications between the Lassen iQ and the dsPIC: basic UART, RS-232 and RS-422. The basic UART is used to interface with the slave unit. The dsPIC has a built in UART module that connects over two wires for data communication. The default speed of the UART interface on the GPS module is set at 9600bps, 8 data bits, odd parity, 1 stop bit and no flow control for both input and output. Deviation from this default is not necessary and this speed is used.

There are three protocols that the GPS unit speaks; TSIP (Trimble Standard Interface Protocol), TAIP (Trimble ASCII Interface Protocol) and NMEA. TSIP is the default for the UART port on the GPS module and is the manufacturers standard

binary protocol for controlling the GPS unit. This protocol provides maximum control of the GPS chip. TAIP is an ASCII implementation of the standard protocol and would only introduce increased size overhead in data transmission. NMEA is an industry standard protocol common in marine applications and is mainly used when interfacing with devices that already speak the NMEA protocol. With TSIP being the default and the most powerful this is the selected protocol for communication with the GPS unit.

8.5 Master-Slave Chip Interface

Within the master unit there is an onboard implementation of the slave unit. This slave implementation communicates with the master chip over wire rather than the wireless interface as with the other slave units. The UART system is available for simple serial communication. This interface is used at a standard speed of 9600 baud, which is more than sufficient to transmit the needed data.

The protocol spoken over this line is the same as the wireless protocol (see section 9.1) with some logistical exceptions. The slave implementation controls the beginning and ending of audio data recording on the master chip. Rather than polling the slave chip as is done with other slave units, the data is pushed out to the master by the slave and received with high priority interrupts to assure timely beginning and ending of audio recording.

8.6 Voltage Buffer Solution

In the one design there were two different voltages that subsystems operate at; 3.3V and 5V. This leads to the need for a voltage buffer interface between systems of different voltages that must communicate digital data. Two components in the system, the SD card and the GPS unit, are required to operate at 3.3V. Additionally the Si3000 codec chip which is capable of running at 5V is within spec when run at 3.3V. The dsPIC is within range when operating at 3.3V, however it is not possible to run the chip at its full 30MIPS when running at the lower voltage. If the dsPIC was run at 3.3V along with the codec the only system requiring a voltage buffer would be the communications system which would still be running at 5V to maintain maximum range. The elimination of one voltage buffer system would in theory simplify the design and reduce prototyping time.

When the Microchip dsPIC is operated at 30MIPS it runs at +5V TTL Voltages, whereas the GPS and SD Card modules operated at +3.3 TTL Voltages. This posed a communication problem between the two. The first obvious option of running the
dsPIC at +3.3V resulted a major, but not critical, loss in computational speed. This was not desirable.

Another possible option was to create a voltage amplifier and an attenuator using transistors. Figure 21 shows the schematic that were simulated and tested on a breadboard.

![3.3V to 5V Interface Schematics](image)

Fig. 21: 3.3V to 5V Interface Schematics

This circuit was an inverter, that took a 3.3V input and outputted 0V, and at 0V outputted 5V. Although this worked properly at reasonable frequencies of 5MHz, it did not perform well at higher frequencies of 15MHz. The reason for this was the capacitance of the breadboard, as well as other parasitic effects from the BJT and resistor components.

Another issue was that this circuit worked only at higher frequencies, but at DC values or much lower frequencies, it did not output a proper value. Eventually we were forced to conclude that this solution would not work, since we needed to translate voltages at a much broader range of frequencies, from DC up to 25MHz. After further research, we came upon an IC solution by Maxim. The MAX3390 was a simple bidirectional voltage translator that efficiently accomplished the solution.
for us. Figure 23 is a basic block diagram of it’s functionality. This worked well for us, and was implemented as our final solution.

Fig. 22: Maxium-IC MAX3390 Level Translator Concept

Fig. 23: Maxium-IC MAX3390 Level Package
9 Design Implementation and Testing

9.1 Communications Unit - Ocean Controls HAC-

One communications unit is connected on both the slave and master buoy. The unit runs at 5v TTL logic levels and is compatible with the dsPICs CMOS logic. No voltage buffering is needed as both the dsPIC and communications unit are running at 5v. The pin connections for the HAC-UM96 can be seen in figure 24. A table listing the connections for each pin and their connection to the dsPIC is seen in table 6.

![Wireless Module Hardware Interface](image)

**Fig. 24: Wireless Module Hardware Interface**

<table>
<thead>
<tr>
<th>Comm Unit</th>
<th>dsPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>Func.</td>
</tr>
<tr>
<td>1</td>
<td>Gnd</td>
</tr>
<tr>
<td>2</td>
<td>Vcc</td>
</tr>
<tr>
<td>3</td>
<td>Com1 Rx</td>
</tr>
<tr>
<td>4</td>
<td>Com1 Tx</td>
</tr>
<tr>
<td>5</td>
<td>Signal GND</td>
</tr>
<tr>
<td>6</td>
<td>Com2 Tx</td>
</tr>
<tr>
<td>7</td>
<td>Com2 Rx</td>
</tr>
<tr>
<td>8</td>
<td>Sleep</td>
</tr>
<tr>
<td>9</td>
<td>Reset</td>
</tr>
</tbody>
</table>

**Tab. 6: HAC-UM12 Connections**
A simple wireless protocol was designed for sending data between the master and slave units. A quick breakdown of the packet structure is seen in table 7.

Byte 1 of the protocol is a destination address byte. A value of 0x00 indicates a packet destination for the master unit. All packets sent from a slave unit are addressed to the master and will have a destination address value of 0x00. Any other value corresponds to single buoy in network as the destination. Only the master unit can communicate with slave units. Byte 2 is the event description. This byte indicates the purpose of the packet. Table 8 shows the meaning of different values of this byte. Byte 3 is the length of the data payload that follows after this byte. The final block of data transmitted is an arbitrary data payload that is of the length specified in byte 3.

This protocol is designed to never have more then one radio unit transmitting at any time. The master unit polls through all the slaves continuously requesting any data with an 0x00 request data packet (table 9). If there are one or more identified signals to report, an 0x01 data response packet is sent for each identified signal (Table 10). After all current data is sent from the slave to the master, the slave terminates the stream with an 0x02 end data response packet (Table 11).

<table>
<thead>
<tr>
<th>Byte</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Destination address</td>
</tr>
<tr>
<td>2</td>
<td>Event description</td>
</tr>
<tr>
<td>3</td>
<td>Data payload length (in bytes)</td>
</tr>
<tr>
<td>4-end</td>
<td>Data payload</td>
</tr>
</tbody>
</table>

Tab. 7: Wireless Protocol

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>Request Data (from master to slave only)</td>
</tr>
<tr>
<td>0x01</td>
<td>Data Response (from slave to master only)</td>
</tr>
<tr>
<td>0x02</td>
<td>End Data Response (from slave to master only)</td>
</tr>
<tr>
<td>0x03-0xFF</td>
<td>Unused</td>
</tr>
</tbody>
</table>

Tab. 8: Wireless Protocol - Event Description Byte
9 Design Implementation and Testing

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x01-0xFF</td>
<td>Destination address</td>
</tr>
<tr>
<td>2</td>
<td>0x00</td>
<td>Event description</td>
</tr>
<tr>
<td>3</td>
<td>0x00</td>
<td>Data payload length (in bytes)</td>
</tr>
</tbody>
</table>

Tab. 9: Wireless Protocol - Request Data Packet

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x00</td>
<td>Destination address, always master</td>
</tr>
<tr>
<td>2</td>
<td>0x01</td>
<td>Event description</td>
</tr>
<tr>
<td>3</td>
<td>0x22</td>
<td>Data payload length (in bytes)</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>Start event (0x00) or stop event (0x01)</td>
</tr>
<tr>
<td>5-13</td>
<td>-</td>
<td>Latitude Position (ascii)(ddmm.mmmm)</td>
</tr>
<tr>
<td>14-21</td>
<td>-</td>
<td>Longitude (ascii)(ddmm.mmmm)</td>
</tr>
<tr>
<td>22</td>
<td>-</td>
<td>N/S Indicator (char)</td>
</tr>
<tr>
<td>23</td>
<td>-</td>
<td>E/W Indicator (char)</td>
</tr>
<tr>
<td>24-32</td>
<td>-</td>
<td>UTC Time (ascii)(hhmmss.sss)</td>
</tr>
<tr>
<td>33-38</td>
<td>-</td>
<td>Date (ascii)(ddmmyy)</td>
</tr>
</tbody>
</table>

Tab. 10: Wireless Protocol - Data Response Packet

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x00</td>
<td>Destination address</td>
</tr>
<tr>
<td>2</td>
<td>0x02</td>
<td>Event description</td>
</tr>
<tr>
<td>3</td>
<td>0x00</td>
<td>Data payload length (in bytes)</td>
</tr>
</tbody>
</table>

Tab. 11: Wireless Protocol - End Data Response Packet

9.1.1 Wireless Module Subsystem Testing

The wireless module was first tested to see if data from the UART port could be sent in received and both directions. This was done by streaming a few bytes at a time from one unit to another and verifying their reception though viewing data memory. In addition to testing at a single bench, longer range tests were also performed. Ranges of up to 100ft were done. Even at these ranges with walls and wiring providing interference, data reception rate was still 100 percent.
9.2 SD Card Implementation and Testing

The SD card was chosen for its ease of integration with embedded systems. The SD card specifications are available freely online\textsuperscript{6}. There are two interfacing options available. The first is the SD proprietary interface. This interface is closed and licensing fees must be paid for documentation and usage. Additionally, this interface is more complicated and difficult to implement without a discrete chip to handle the interfacing from the microprocessor to the SD card. The second option is to use the freely available SPI interface. Not only is this interface easier to implement, it also is openly documented.

The SPI interface consists of the standard four wire SPI interface as well as a few extra pins specific to the SD card. The SPI interface has unidirectional transmit and receive pins, a master clock line and a chip select pin. In addition to the SPI pins there are also power and two ground pins. A diagram of the pin configuration of an SD card in SPI mode is show in figure 25. Additionally, the pin connections of both the SD card and the dsPIC can be seen in table 12.

![SD Card SPI Pin Configuration](image)

Fig. 25: SD Card SPI Pin Configuration

An implementation of SPI interfacing of an SD card and the dsPIC30f4013 was found online\textsuperscript{7}. This implementation included schematics and firmware example code freely available under the GNU GPL v2 license\textsuperscript{8}. This code showed an example of reading a single 512 byte block from an SD card. This code was adapted to allow for reading and writing 512 byte blocks to and from arbitrary address on the SD card. Additionally with these new modifications this code was adapted into a modular

\begin{itemize}
  \item http://www.cs.ucr.edu/~amitra/sdcard/ProdManualSDCardv1.9.pdf
  \item http://www.k9spud.com/sdcard/
  \item http://www.gnu.org/licenses/gpl-2.0.html
\end{itemize}
library format to allow for easy use as an SD card driver. The original schematics and code can be seen in appendix C.1 and C.2.

When the dsPIC is operating at its maximum speed of 30MIPS the frequency is outside the suggested operating range for SD cards. SD cards have a maximum operating speed of 25Mhz. A simple 2:1 clock pre-scaler is used for the dsPICs SPI clock to run the SD card at an acceptable 15Mhz.

9.2.1 Custom File System

The custom filesystem that is implemented is a basic sequential block storage system. This system only requires that data be recorded to the SD card, not read. Because of this, a complex filesystem with look up tables relating to read/write filesystems is not necessary. Data is stored in 512 byte blocks on the SD card with a two byte header that identifies the contents of that block. The two byte header is simply an unsigned integer.

For blocks containing GPS time and location data about signal identifications, the value of this header is zero. These blocks are sequently filled with packets of GPS time and location data as well as a reference to an audio recording number. The structure of these packets is shown in table reftab:gpspacket. The packets are 36 bytes in length and fourteen can fit in a single filesystem block.

For any non zero value of the header it refers to a block of audio data. The particular value is used to uniquely identify which audio recording the data belongs to. For example all audio data blocks corresponding to the first recording would have a header value of 1. The next recording would have all audio data blocks with a header value of 2 and so on.
### 9 Design Implementation and Testing

<table>
<thead>
<tr>
<th>byte</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>address of buoy received at</td>
</tr>
<tr>
<td>1</td>
<td>Start event (0x00) or stop event (0x01)</td>
</tr>
<tr>
<td>2-9</td>
<td>Latitude Position (ascii)(ddmm.mmmm)</td>
</tr>
<tr>
<td>10-17</td>
<td>Longitude (ascii)(ddmm.mmmm)</td>
</tr>
<tr>
<td>18</td>
<td>N/S Indicator (char)</td>
</tr>
<tr>
<td>19</td>
<td>E/W Indicator (char)</td>
</tr>
<tr>
<td>20-28</td>
<td>UTC Time (ascii)(hhmmss.sss)</td>
</tr>
<tr>
<td>29-34</td>
<td>Date (ascii)(ddmmyy)</td>
</tr>
<tr>
<td>35-36</td>
<td>Related audio data header (unsigned int)(0 if none)</td>
</tr>
</tbody>
</table>

**Tab. 13: Identified Signal Packet**

#### 9.2.2 SD Card Subsystem Testing

Subsystem testing of the SD card was done in several incremental stages. The first step was testing the already available firmware. This code reads a single 512 byte block from the beginning of the SD card. To test this several UNIX utilities were used to write and read data from the SD card. The first tool used was ‘dd’. dd, or data dump allows a method of low level writing and reading from files and block devices in a UNIX environment. To test the 512 byte read a 512 character ASCII file was prepared and dumped to the first block of the SD card with dd. The firmware was downloaded to the dsPIC and run. Upon completion of the read function, a break point was placed and memory content was viewed to verify the data had been read from the SD card.

The second stage of testing involved implementing and testing reading as well as writing after the code had been adapted into a modular driver. Much of the same method of testing was used here. For the read test a more robust test was done. Rather then only reading the first block of data, the entire card was read one block at a time. Each block was verified tough viewing the memory to verify it was read correctly. To test the write functionality a 512 byte ASCII string was stored on the dsPIC and then sequentially written to the entire SD card. To verify it had been written correctly the SD card was first zeroed out to ensure no data from a previous test was showing a false positive. The data from the SD card was dumped to a file and examined to verify all 512 blocks were filled with the correct ASCII string.
9.3 GPS Implementation and Testing

The GPS module, Trimble Lassen iq, offers two separate communications interfaces for systems integration. The first is a UART compatible RS232 non-level shifted interface. This is designated as COM1 or COM A. The second is an RS-242 compatible interface designated COM2 or COM b. COM1 was chosen as the communications interface as the dsPIC contains a hardware UART module that makes data sending and receiving simple.

The pin connections for the GPS module and dsPIC can be seen in table 14. The alternate UART1 pin configuration was chosen as the SD card SPI interface shares two pins with the standard UART1 pins on the master unit. Although the slave unit could use the standard pins as the SPI interface is not in use, it was decided simpler to simply use the same pin configuration and firmware code so only one version of the GPS driver needed to be maintained.

<table>
<thead>
<tr>
<th>GPS Module</th>
<th>dsPIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin</td>
<td>Func.</td>
</tr>
<tr>
<td>1</td>
<td>COM1 TX</td>
</tr>
<tr>
<td>2</td>
<td>Gnd</td>
</tr>
<tr>
<td>3</td>
<td>COM1 RX</td>
</tr>
<tr>
<td>4</td>
<td>PPS</td>
</tr>
<tr>
<td>5</td>
<td>COM2 TX</td>
</tr>
<tr>
<td>6</td>
<td>COM2 RX</td>
</tr>
<tr>
<td>7</td>
<td>Main Power</td>
</tr>
<tr>
<td>8</td>
<td>Battery Backup</td>
</tr>
</tbody>
</table>

Tab. 14: GPS Module Pin Connections

9.3.1 GPS Subsystem Testing - Trimble Lassen iQ

The GPS system testing is broken up into two blocks. The first part of the system that needed to be tested was to ensure that the module was communicating over the serial interface and commands could be sent and received without issue. The second part of the testing was to test the GPS position and time information.

The serial testing was done by sending various commands to the GPS module and verifying the correct returns packets were received. Several commands such as the ‘report device status’ and ‘report satellite signal strength’ were sent to the GPS module. The return packets were viewed in memory to verify the proper packets and information were returned.
The second part of the testing was to verify that GPS positioning and time data could be read and was correct. The first step of this process was to verify that the GPS module was receiving a signal from at least three GPS satellites. To obtain the satellite strength information the ‘report satellite signal strength’ packet was sent to the GPS module. However, the returning packet consistently reported zero of the twelve channels ever tracking a GPS satellite. Several locations were tried including inside buildings, near windows and outside in several locations. The only signal lock ever gained was on just one satellite at a time and was intermittent at best. To determine the issue with the system a second new GPS module was put in place of the first to see if hardware was the issue. The problem persisted indicating hardware failure in the module was not the issue. The next step would be to swap the antenna out and attempt the test with a new antenna. Unfortunately due to time and budget constraints a second antenna could not be obtained.

To test GPS as a viable option in the face of the GPS modules failure, a separate USB GPS receiver was used to test the locations over again. In every location at least one strong GPS satellite lock was made, and in most locations three or more. Every location gave at the very least intermittent position and time data and usually strong accurate data. This would seem to indicate the GPS antenna for the Lassen iQ is either defective or broken.

To overcome the issues with the Lassen iQ GPS module a contingency plan has been deduced. Rather then attempt to debug the already problematic Lassen iQ module a new GPS module has been selected. The USB GPS receiver tested above is a SiRF3 based GPS unit. With such major success when testing this USB module an embedded solution that is based on the SiRF3 GPS core is desired. The USGlobalSat EM-406A was selected to replace the Lassen iQ. The EM-406A offers many advantages over the Lassen iQ unit. First, this unit is a 20 channel as opposed to 12 as with the Lassen iQ. Second its operating voltage range is 4.5v to 6.5v. This allows the unit to run directly of the 5v power and removes the need for logic level translation in the slave units. A final advantage is that this unit has a built in antenna. The most likely reason for the failure of the Lassen iQ is a faulty external antenna. This unit would remove any possibility of this issue. The price of this unit is slightly higher then the Lassen iQ ($59.95 vs $49.95), however with the Lassen iQ an external antenna must also be purchased that costs well over $10.00. This puts the total cost of the Lassen iQ above the EM-406A.

Although a viable alternative to the Lassen iQ was found it was too late in the development process to purchase and prototype with the EM-406A. However the design has been altered to reflect this change and would be ready to accept the EM-406A with very little firmware and hardware change. As the format of data coming
from the unit is well documented, the communications with the unit were faked in firmware to simulate an attached unit. This would make the difference between this faked data and a live unit almost indistinguishable to the rest of the code in the firmware. Additionally the EM-406A would be connected on the same pins and same UART module of the dsPIC. The only difference in firmware would be a change in baud rate from 9600 to 4800.
9.4 Analog to Digital Converter Implementation and Testing

The analog to digital converter used is the built in 12bit converter of the dsPIC. Although the Si3000 external CODEC analog to digital converter was considered for the high quality recording, the proof of concept nature of this project made the internal analog to digital converter a faster and easier solution to implement. Additionally the loss of dynamic range due to 12bit rather then 16bit sampling proved not to be an issue as the high quality recording is made only for later analysis of the signal. In a future revision of this project the external CODEC could be integrated in both the master and slave units with minimal hardware and firmware changes.

The analog to digital converter in the dsPIC offers 12bit sampling at rates up to 120Khz. For this project 8Khz is more then sufficient due to the low frequency nature of the North Atlantic Right whales vocalizations. The data was retrieved from the analog to digital unit as unsigned 12bit integers. However, this data was stored in 16bit integers due to the 16bit architecture of the dsPIC.

When data is retrieved form the SD card on a PC it must be preprocessed before it is usable. The first step in processing the data from the SD card is to swap the high and low byte of each 16bit sample. This is necessary because the dsPIC is a big endian architecture where as x86 is little endian. The data then had to be centered around zero. As the dynamic range went from 0 to $2^{12}$ or 4096, the zero level of this data is at 2048. After the data is centered at zero, it then must be scaled between -1 to 1. This is done by dividing all samples by 2048. At this point normalized floating point samples that are zero centered and ranging from -1 to 1 are ready to be processed with MATLAB or other analysis tools. A simple C program was written to extract and normalize the data to be ready for MATLAB. This code is available in appendix D.1

9.4.1 Analog to Digital Converter Subsystem Testing

As with the other subsystem tests, the analog to digital converter testing was broken down into several incremental phases. The first step was to verify that data was being recorded at all. A sine wave was input to the dsPIC and continuously recorded to the SD card. When viewed in MATLAB it was verified that the signal was a clean sine wave at the proper frequency.

The second step was to record various sound files such as music and sounds from the whale data set. This data was played into the dsPIC and again recorded to the SD card. In addition to viewing the data in MATLAB it was also played back for human auditory analysis of the sound quality and validity of the recording.
<table>
<thead>
<tr>
<th>QTY</th>
<th>Description</th>
<th>Dist.</th>
<th>Part No.</th>
<th>MFG</th>
<th>MFG Part No.</th>
<th>Unit</th>
<th>Sub</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MPLAB Compatible ICD2</td>
<td>Sparkfun</td>
<td>PGM-000005</td>
<td>Olimex</td>
<td>N/A</td>
<td>106.95</td>
<td>106.95</td>
</tr>
<tr>
<td>1</td>
<td>Antenna GPS for Lassen IQ</td>
<td>Sparkfun</td>
<td>GPS-00178</td>
<td>Lassen</td>
<td>45336-00-AD</td>
<td>18.95</td>
<td>18.95</td>
</tr>
<tr>
<td>1</td>
<td>Wall Adapter - 9VDC 650mA</td>
<td>Sparkfun</td>
<td>TOL-00298</td>
<td>N/A</td>
<td>N/A</td>
<td>5.95</td>
<td>5.95</td>
</tr>
<tr>
<td>1</td>
<td>USB Cable A to B - 6 Foot</td>
<td>Sparkfun</td>
<td>CAB-00512</td>
<td>N/A</td>
<td>N/A</td>
<td>3.95</td>
<td>3.95</td>
</tr>
<tr>
<td>2</td>
<td>SD Card Slot</td>
<td>Sparkfun</td>
<td>PRT-00136</td>
<td>4UCON Technology</td>
<td>PRT-00136</td>
<td>3.95</td>
<td>7.9</td>
</tr>
<tr>
<td>1</td>
<td>Lassen iQ GPS</td>
<td>Sparkfun</td>
<td>GPS-00163</td>
<td>Lassen</td>
<td>N/a</td>
<td>49.99</td>
<td>49.99</td>
</tr>
<tr>
<td>4</td>
<td>Microcontroller</td>
<td>Microchip</td>
<td>DSPIC30F4013</td>
<td>Microchip</td>
<td>dsPIC4013</td>
<td>5.02</td>
<td>20.08</td>
</tr>
<tr>
<td>2</td>
<td>Oceans Control WIR-002 Modem</td>
<td>Oceans Control</td>
<td>WIR-002</td>
<td>Oceans Control</td>
<td>HAC-UM96</td>
<td>63.23</td>
<td>126.46</td>
</tr>
<tr>
<td>4</td>
<td>A/D Converter</td>
<td>Digikey</td>
<td>336-1382-ND</td>
<td>N/A</td>
<td>SI3000-C-FS</td>
<td>1.57</td>
<td>6.28</td>
</tr>
<tr>
<td>2</td>
<td>NPN Transister</td>
<td>Digikey</td>
<td>497-2598-ND</td>
<td>STMicroelectronics</td>
<td>2N2222A</td>
<td>0.95</td>
<td>1.9</td>
</tr>
</tbody>
</table>

Tab. 15: Parts List
11 Conclusion

The Smart Acoustic Whale monitor can remain self-sufficient for long periods of time, and not require labor or maintenance. Memory does not need to be swapped, nor does battery for a long time. The data is stored on a very small SD card as well, and all communications between buoys are done wirelessly.

The analysis of each whale vocalization is done onboard our system, and the data is outputted to the SD card. This saves the analyst a great deal of time. In conclusion, our project is an efficient and feasible way to monitor whale activity.
A Firmware Code

A.1 main-master-master.c

001 #include "p30f4013.h"
002 #include "dsp.h"
003 #include "sd_card.h"
004 #include "a2d.h"
005 #include "wireless.h"
006 #include "slave2master.h"
007 #define FOSC(CSW_FSCM_OFF & FRC_PLL8); // 64Mhz (8mhz internal RC * 8xPLL) (16MIPS = CLK/4)
008 _FOSC(CSW_FSCM_OFF & FRC_PLL8); // 64Mhz (8mhz internal RC * 8xPLL) (16MIPS = CLK/4)
009 _FWDT(WDT_OFF);
010 _FBORPOR(PBOR OFF & BORV 20 & PWRT 64 & MCLR_EN);
011 _FGS(CODE PROT OFF);
012 #define ISR __attribute__((interrupt, auto_psv))
013 #define ISRFAST __attribute__((interrupt, shadow, auto_psv))
014 #define LED PORTBbits.RB0
015 #define LED_DIR TRISBbits.TRISB0
016 #define kNumCoeffs 4
017 const unsigned int firTaps[ kNumCoeffs ] = {
018 0.25, 0.25, 0.25, 0.25
019};
020 FIRStruct low12bitFilter;
021 fractional _YBSS(128) delay[kNumCoeffs];
022 fractional inSample;
023 fractional outSample;
024 //unsigned char buf[512]="ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '0' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descripti";
025 int buf1[256];
026 int buf2[256];
027 int buf_num;
unsigned short indexx;
unsigned long sd_index;
unsigned int write;
int bg_level;
unsigned long energy[3];
int inbuf[10];

int ADResult1;

unsigned int record;

int main (void)
{
  energy[0]=0;
  energy[1]=0;
  energy[2]=0;
  indexx = 0;
  sd_index=0;
  buf_num=1;
  write=0;
  record=0;
  ADPCFG = 0xFFFF; // Force all ADC pins as digital I/O
  ADPCFG = 0xFFFB; // set AN2 to analog input
  for(i=0;i<256;i++)
    buf1[i]="c"+(‘d’<<8);
  TRISBbits.TRISB9 = 0;
  PORTBbits.RB9 = 1;
  // Configure output pins
  LED_DIR = 0;
  LED=1;
  init_wireless();
  //init slave to master communication
  init_slave2master();
  //init the SD card
  InitSPI();
  unsigned char status;
  status = InitSD();
  do{
    status = InitSD();
  }
A Firmware Code

081 } while(status);
082
083 //start sampling the A/D
084 init_ad();
085
086 //main loop
087 while(1) //loop and write data to card when ready
088 {
089 if(write)
090 {
091 int status;
092 do{
093 if(write==1)
094 status=SD_WriteBlock(sd_index*512,(char*)(buf1));
095 else
096 status=SD_WriteBlock(sd_index*512,(char*)(buf2));
097 }while(status);
098 write=0;
099 sd_index++;
100 }
101 }
102 return 0;
103 }
104 }
105
106 void ISRFAST_U2RXInterrupt(void)
107 {
108 while( U2STAbits.URXDA )
109 {
110 unsigned char in;
111 in=U2RXREG;
112 if(in == 0x0F)
113 {
114 LED=1;
115 record=1;
116 }
117 else if(in == 0x00)
118 {
119 LED=0;
120 record=0;
121 }
122 }
123 IFS1bits.U2RXIF=0; //clear UART1 receiver interrupt flag
124 }
125 }
void ISRFAST_ADCInterrupt(void)
{
    ADResult1 = ADCBUF0;
    // inSample = ADResult1;
    // FIR( 1, &outSample, &inSample, &low12bitFilter );
    // ADResult1 = outSample;
    /*
     * //shift in buffer
     * int i;
     * for(i=9; i>0; i--)
     *     inbuf[i] = inbuf[i-1];
     * inbuf[0] = ADResult1-1906;
     * //calculate the power level
     * unsigned long eng=0;
     * for(i=0; i<10; i++)
     *     eng+=((long)inbuf[i]*(long)inbuf[i])/10;
     * //get new power average
     * unsigned long pow_avg=eng/4;
     * for(i=0; i<3; i++)
     *     pow_avg+=energy[i]/4;
     * //shift power buffer
     * //only if not recording
     * if(!record)
     * {
     *     energy[2] = energy[1];
     *     energy[1] = energy[0];
     *     energy[0] = pow_avg;
     * }
     *
     * //test for a noise
     * if(pow_avg>10000)
     * {
     *     record=1;
     *     LED=1;
     * }
     * else if(record)
     * {
     *     record++;
     *     if(record>=1000)
     *     {
     *         LED=0;
     *         record=0;
     *     }
     * 
     * /*}
     */
}
A Firmware Code

```c
indexx=0;
if(buf_num)
    write=1;
else
    write=2;
for(i=0;i<255;i++)
{
    buf1[i]=0;
    buf2[i]=0;
}

if(record)
{
    //write to only the current buffer
    if(buf_num)
        buf1[indexx++]=ADResult1;
    else
        buf2[indexx++]=ADResult1;
    //if buffer full set write flag for current buffer and switch to other buffer
    if(indexx>255)
    {
        if(buf_num)
            write=1;
        else
            write=2;
        buf_num=!buf_num;
        indexx=0;
    }
    IFS0bits.ADIF = 0;
}

void ISR_OscillatorFail( void )
{
    while(1) ;
}

void ISR_AddressError( void )
{
```

while(1);
}

void ISR _StackError( void )
{
  while(1);
}

void ISR _MathError( void )
{
  while(1);
}
A.2 main-master-slave.c

001
002 #include "p30f4013.h"
003 #include "dsp.h"
004 #include "math.h"
005 #include "a2d.h"
006 #include "gps.h"
007 #include "slave2master.h"
008
009 #define ISR __attribute__((interrupt, auto_psv))
010 #define ISRFAST __attribute__((interrupt, shadow, auto_psv))
011
012 #define LED PORTBbits.RB0
013 #define LED_DIR TRISBbits.TRISB0
014
015 #define IN_SIZE 100
016 int inbuf[IN_SIZE];
017 int ADResult1;
018
019 unsigned int record;
020
021 int main (void)
022 {
023   record=0;
024
025   ADPCFG = 0xFFF8; // set AN2 to analog input
026
027   TRISBbits.TRISB9 = 0;
028   PORTBbits.RB9 = 1;
029
030   // Configure output pins
031   LED_DIR = 0;
032   LED = 0;
033
034   // init_gps();
035
036   //init uart to master chip
044 // init_slave2master();
045 046 //start sampling the A/D
047 init_ad();
048 049 //main loop
050 while(1)
051 {
052 }
053
054 return 0;
055 }
056
057
058 void ISRFAST_ADCInterrupt(void)
059 {
060   ADResult1 = ADCBUF0;
061
062 //shift in buffer
063   int i;
064   for(i=IN_SIZE-1;i>0;i--)
065       inbuf[i]=inbuf[i-1];
066   inbuf[0]=ADResult1-1906;
067
068 //Clear the A/D Interrupt flag bit or else the CPU will
069 //keep vectoring back to the ISR
070   IFS0bits.ADIF = 0;
071 }
072 /*
073 void ISR_OscillatorFail( void )
074 {
075   while(1);
076 }
077
078 void ISR_AddressError( void )
079 {
080   while(1);
081 }
082
083 void ISR_StackError( void )
084 {
085   while(1);
086 }
087
088 void ISR_MathError( void )
089 {
while(1);
A.3  sd_card.c

001  // -* tab-width: 4 -*
002  // SD Card Read Test
003  // For use on dsPIC30F4013 at 3.3VDC
004  //
005  // Copyright (c) 2005, K9spud LLC.
006  // http://www.k9spud.com/sdcard/
007  //
008  // This program is free software; you can redistribute it and/or
009  // modify it under the terms of the GNU General Public License
010  // as published by the Free Software Foundation; either version 2
011  // of the License, or (at your option) any later version.
012  //
013  // This program is distributed in the hope that it will be useful,
014  // but WITHOUT ANY WARRANTY; without even the implied warranty of
015  // MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
016  // GNU General Public License for more details.
017  //
018  // You should have received a copy of the GNU General Public License
019  // along with this program; if not, write to the Free Software
020  // Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307,
021  // USA.
022  //
023  // $Log: sdtest.c,v $
024  // Revision 1.2-mcc 2008/2/24 4:22:00 Matthew Campbell
025  // Modularized into library and added block write support
026  //
027  // Revision 1.2 2005/09/09 04:26:23 edwards
028  // Comments updated.
029  //
030  // Revision 1.1 2005/09/07 07:07:18 edwards
031  // First successful read from SD Card on my dsPIC30F4013!
032  #include "sd_card.h"
033  
034  /***************************************************************************/
035  // SPI Related Functions
036  /***************************************************************************/
037
038  void SPIWrite(unsigned char data)
039  {
040      // DO NOT WAIT FOR SPITBF TO BE CLEAR HERE
041      // (for some reason, it doesn’t work on this side of the write data).
042
043      // Write the data!
SPI1BUF = data;
// Wait until send buffer is ready for more data.
while(SPI1STATbits.SPITBF);

unsigned char SPIRead(void)
{
    unsigned char data;
    if(SPI1STATbits.SPIRBF)
    {
        // already have some data to return, don't initiate a read
        data = SPI1BUF;
        SPI1STATbits.SPIROV = 0;
        return data;
    }

    // We don't have any data to read yet, so initiate a read
    SPI1BUF = 0xFF;  // write dummy data to initiate an SPI read
    while(SPI1STATbits.SPITBF);  // wait until the data is finished reading
    data = SPI1BUF;
    if(data!=0xFF && data!=0x00)
        PORTBbits.RB0=1;
    SPI1STATbits.SPIROV = 0;
    return data;
}

void InitSPI(void)
{
    SD_PowerOff();
    SD_PWR_DIR = 0;  // output
    SD_PowerOff();
    SD_Disable();
    SD_CS_DIR = 0;  // output
    SD_Disable();
    SDI_DIR = 1;  // input
    SCK_DIR = 1;
    SDO_DIR = 1;
    SPI1STAT = 0x8000;  // enable SPI port
090 // set SPI port to slowest setting
091 // master mode
092 // 8 bit
093 // Idle state for Clock is high level
094 // Primary prescaler 64:1
095 // Secondary prescaler 8:1
096 SPI1CON = 0x0060;
097 }
098
099
100
101
102 //******************************************************************************
103 // SD Card Related Functions
104 //******************************************************************************
105
106 unsigned char SD_WriteCommand(unsigned char* cmd)
107 {
108    unsigned int i;
109    unsigned char response;
110    unsigned char savedSD_CS = SD_CS;
111
112
113    // SD Card Command Format
114    // (from Section 5.2.1 of SanDisk SD Card Product Manual v1.9).
115    // Frame 7 = 0
116    // Frame 6 = 1
117    // Command (6 bits)
118    // Address (32 bits)
119    // Frame 0 = 1
120
121    // Set the framing bits correctly (never change)
122    cmd[0] |= (1<<6);
123    cmd[0] &= ~(1<<7);
124    cmd[5] |= (1<<0);
125
126    // Send the 6 byte command
127    SD_Enable();
128    //wait for card to go idle
129    while(SPIRead()!=0xFF);
130    for(i = 0; i < 6; ++i)
131    {
132        SPIWrite(*cmd);
133        cmd++;
134    }
// Wait for the response
i = 0;
do {
    response = SPIRead();
    if(i > 100)
        break;
i++;
} while(response == 0xFF);

SD_Disable();

// Following any command, the SD Card needs 8 clocks to finish up its work.
// (from SanDisk SD Card Product Manual v1.9 section 5.1.8)
SPIWrite(0xFF);
SD_CS = savedSD_CS;
return(response);

unsigned char InitSD() {
    unsigned int i = 0;
    unsigned char status;

    // Turn off SD Card
    SD_Disable();
    SD_PowerOff();

    // Wait for power to really go down
    for(i = 0; i; i++);
    for(i = 0; i; i++);
    for(i = 0; i; i++);
    for(i = 0; i; i++);

    // Turn on SD Card
    SD_PowerOn();

    // Wait for power to really come up
    for(status = 0; status < 10; ++status)
        {
            for(i = 0; i; i++);
            for(i = 0; i; i++);
for(i = 0; i; i++);  
for(i = 0; i; i++);  
}  

// We need to give SD Card about a hundred clock cycles to boot up  
for(i = 0; i < 16; ++i)  
{  
    SPIWrite(0xFF); // write dummy data to pump clock signal line  
}  

SD_Enable();  

// This is the only command required to have a valid CRC  
// After this command, CRC values are ignore unless explicitly enabled  
// using CMD59  
unsigned char CMD0_GO_IDLESTATE[] = {0x00,0x00,0x00,0x00,0x00,0x95};  

// Wait for the SD Card to go into IDLE state  
i = 0;  
do  
{  
    status = SD_WriteCommand(CMD0_GO_IDLESTATE);  
    
    // fail and return  
    if(i++ > 50)  
    {  
        return 1;  
    }  
} while( status != 0x01 );  

// Wait for SD Card to initialize  
unsigned char CMD1_SEND_OP_COND[] = {0x01,0x00,0x00,0x00,0x00,0xFF};  
i = 0;  
do  
{  
    status = SD_WriteCommand(CMD1_SEND_OP_COND);  
    if(i++ > 50)  
    {  
        return 2;  
    }  
} while( (status & R1_IN_IDLESTATE) != 0 );  

// Send CMD55, required to precede all "application specific" commands  
unsigned char CMD55_APP_CMD[] = {55,0x00,0x00,0x00,0x00,0xFF};  
status = SD_WriteCommand(CMD55_APP_CMD); // Do not check response here
i = 0;
unsigned char ACMD41_SD_SEND_OP_COND[] = {41,0x00,0x00,0x00,0x00,0xFF};
do
{
    status = SD_WriteCommand(ACMD41_SD_SEND_OP_COND);
    // Might return 0x04 for Invalid Command if MMC card is connected
    if(i++ > 50)
    {
        return 3;
    }
} while( (status & R1_IN_IDLE_STATE) != 0 );

// Set the SPI bus to full speed now that SD Card is initialized in SPI mode
SD_Disable();
SPIFastClock();
return 0;

// SD Card defaults to 512 byte block size
#define BLOCK_SIZE 512
unsigned char SD_ReadBlock(unsigned long addr, unsigned char *buf)
{
    unsigned int i;
    unsigned char status;
    unsigned char CMD17_READ_SINGLE_BLOCK[] = {17,0x00,0x00,0x00,0x00,0xFF};
    CMD17_READ_SINGLE_BLOCK[1] = ((addr & 0xFF000000) >> 24);
    CMD17_READ_SINGLE_BLOCK[2] = ((addr & 0x00FF0000) >> 16);
    CMD17_READ_SINGLE_BLOCK[3] = ((addr & 0x0000FF00) >> 8);
    CMD17_READ_SINGLE_BLOCK[4] = ((addr & 0x000000FF));
    SD_Enable();
    status = SD_WriteCommand(CMD17_READ_SINGLE_BLOCK);
    if(status != 0)
    {
        // ABORT: invalid response for read single command
        return 1;
    }
// Now wait for the "Start Block" token (0xFE)
// (see SanDisk SD Card Product Manual v1.9 section 5.2.4. Data Tokens)
do {
    status = SPIRead();
} while(status != 0xFE);

// Read off all the bytes in the block
for(i = 0; i < BLOCK_SIZE; ++i)
{
    status = SPIRead();
    ∗buf = status;
    buf++;
}

// Read CRC bytes
status = SPIRead();
status = SPIRead();

//wait for non busy response
while(SPIRead()!=0xFF);
SD_Disable();
return 0;

unsigned char SD_WriteBlock(unsigned long addr, unsigned char ∗buf) {
    unsigned int i;
    unsigned char status;

    unsigned char cmd[] = {24,0x00,0x00,0x00,0x00,0xFF};
    cmd[1] = ((addr & 0xFF000000) >> 24);
    cmd[2] = ((addr & 0x00FF0000) >> 16);
    cmd[3] = ((addr & 0x0000FF00) >> 8);
    cmd[4] = ((addr & 0x000000FF));

    SD_Enable();

    // Send the write command
    status = SD_WriteCommand(cmd);
    if(status != 0)
    {
        return 1;
    }
A Firmware Code

317    }
318
319    SPIWrite(0xFF); //give the card a couple clock cycles to think about it
320    SPIWrite(0xFF);
321
322    //send block start token
323    SPIWrite(0xFE);
324    int j=0;
325    for(j=0;j<512;j++)
326        SPIWrite(buf[j]);
327    //send a dummy 16bit CRC
328    SPIWrite(0xFF);
329    SPIWrite(0xFF);
330
331    //wait for non busy response
332    while(SPIRead()==0x00); //wait for response to end
333
334    SD_Disable();
335    return 0;
336    }
337
338
339
A.4  sd_card.h

001 #ifndef __sd_card_h__
002 #define __sd_card_h__
003
004 #include "p30f4013.h"
005
006 // Set port usage with defines
007 #define SD_PWR PORTBbits.RB1
008 #define SD_PWR_DIR TRISBbits.TRISB1
009 #define SD_CS PORTBbits.RB3
010 #define SD_CS_DIR TRISBbits.TRISB3
011
012 #define SDI PORTFbits.RF2
013 #define SDI_DIR TRISFbits.TRISF2
014 #define SCK PORTFbits.RF6
015 #define SCK_DIR TRISFbits.TRISF6
016 #define SDO PORTFbits.RF3
017 #define SDO_DIR TRISFbits.TRISF3
018
019 // power and chip select macros
020 #define SD_PowerOn() SD_PWR = 1
021 #define SD_PowerOff() SD_PWR = 0
022 #define SD_Enable() SD_CS = 0 /* set low to activate SD Card chip select */
023 #define SD_Disable() SD_CS = 1 /* set high to deactivate SD Card chip select */
024
025 // SPI clock macro
026 // primary prescaler 1:1
027 // secondary prescaler 1:1
028 #define SPIFastClock() SPI1CON = 0x007F
029 // Due to slow voltage translation need to run at 7.5MHz
030 // primary prescaler 1:1 (11)
031 // secondary prescaler 8:1 (000)
032 // #define SPIFastClock() SPI1CON = 0x007E //(011 0.00 11)
033
034 // SD Card SPI Commands are defined in the SanDisk SD Card Product Manual
035 // v1.9
036 // section 5.2.2.1 (page 91 of PDF)
038 //
039 // 0 - GO_IDLE_STATE - Resets the SD Card
040 // 1 - SEND_OP_COND - Activates the card’s initialization process
041 // 9 - SEND_CSD - Asks card to send card-specific data
042 // 10 - SEND_CID - Asks card to send card identification
043 // 12 - STOP_TRANSMISSION - Forces card to stop transmission during
multi-block read

043 // 13 - SEND_STATUS  Asks card to send its status register.
044 // 16 - SET_BLOCKLEN  Selects block length for all subsequent block
commands (default is 512)
045 // 17 - READ_SINGLE_BLOCK  Reads a block of the size specified by
SET_BLOCKLEN
046 // 18 - READ_MULTIPLE_BLOCK  Continuously transfers data until interrupted by
STOP_TRANSMISSION
047 // 24 - WRITE_BLOCK  Writes a block of the size specified by SET_BLOCKLEN
048 // 25 - WRITE_MULTI_BLOCK  Continuously writes blocks of data until a stop
transmission token is sent instead of start block token.

R1 Response Codes (from SD Card Product Manual v1.9 section 5.2.3.1)

#define R1_IN_IDLE_STATE (1<<0)  // The card is in idle state and running
initializing process.
#define R1_ERASE_RESET (1<<1)  // An erase sequence was cleared before
executing because of an out of erase sequence command was received.
#define R1_ILLEGAL_COMMAND (1<<2)  // An illegal command code was detected
#define R1_COM_CRC_ERROR (1<<3)  // The CRC check of the last command
failed.
#define R1_ERASE_SEQ_ERROR (1<<4)  // An error in the sequence of erase
commands occurred.
#define R1_ADDRESS_ERROR (1<<5)  // A misaligned address, which did not
match the block length was used in the command.
#define R1_PARAMETER (1<<6)  // The command’s argument (e.g. address,
block length) was out of the allowed range for this card.
#define R1_BIT (1<<7) is always 0

 External API for SD card

unsigned char SD_WriteCommand(unsigned char *cmd);
unsigned char InitSD();
unsigned char SD_ReadBlock(unsigned long addr, unsigned char *buf);
unsigned char SD_WriteBlock(unsigned long addr, unsigned char *buf);

 External API for SPI interface

void InitSPI(void);

#endif//__sd_card_h__
A.5 gps.c

```
#include "p30f4013.h"
#include "gps.h"

void init_gps()
{
    //using uart1 module, pins 25(tx) and 26(rx)
    U1MODEbits.PDSEL = 0b10; //set parity
    /*PDSEL<1:0> value meaning (binary values)
     * 00 8-bit data, no parity
     * 01 8-bit data, even parity
     * 10 8-bit data, odd parity
     * 11 9-bit data, no parity
     */
    U1MODEbits.STSEL = 0; //set stop bits to one
    // U1BRG = 376; //set baud rate to 1200 through BRG
    U1BRG = 90; //set baud rate to 9600 through BRG
    U1MODEbits.UARTEN = 1; //turn on uart 2 module
    U1STAbits.UTXEN = 1; //enable transmission

    char j[1024];
    while(1)
    {
        //zero out buffer
        int k = 0;
        for(k=0;k<1024;k++)
            j[k]=0;

        //request satilite info
        U1TXREG = 0x10;
        while(U1STAbits.UTXBF);
        U1TXREG = 0x27;
        while(U1STAbits.UTXBF);
        U1TXREG = 0x10;
        while(U1STAbits.UTXBF);
        U1TXREG = 0x03;
        while(U1STAbits.UTXBF);
        k=0;
        while(k<200)
        {
            while(!(U1STAbits.URXDA));
            j[k] = U1RXREG; //read data
            if(j[k]==0x03 && j[k-1]==0x10)
                break;
```

045 k++;  
046 }  
047 }  
048 }  
049  
050
A.6  gps.h

    #ifndef __gps_h__
    #define __gps_h__

    void init_gps();

    #endif//__gps_h__
A.7 a2d.c

001 #include "a2d.h"
002 
003 void init_ad()
004 {
005 /∗ ADCON1bits.FORM=0b11; // set conversion format to unsigned int
006    ADCON1bits.CHOSA = 0x02; // enable sampling on AN2
007    ADCON2bits.VCFG=0x00; // set voltage ref to VSS and VDD
008    ADCON1bits.SSRC=0b111; // set aquisition to automatic
009    ADCON1bits.ASAM=0b1; // set automatic converstion
010    ADCON3bits.ADCS=39; // set aquisition time*/
011    // ADPCFG = 0xFFFF;  // Clear Bit-5, Set all others Only AN5 is Analog
012    // ADPCFG = 0xFFCF;  // AN3,4,5 analog
013    // ADPCFG = 0xFFD5;  // AN0,1,5 analog
014    ADCS = 0; // clear ADICHS
015    ADCCHSbits.CHOSA = 2; // MUX-A Positive Input is AN5; This is the input port to the AtoD
016    // ADCHSbits.CH0SA = 2; // MUX-A Positive Input is AN5; This is the input port to the AtoD
017    // ADCHSbits.CHOSA = 4;
018    // ADCHSbits.CHOSA = 5;
019    // ADCHSbits.CH123SA = 1;
020 
021    //ADCHS = 0x0007;  // MC EXAMP
022 
023    // ADCON3bits.ADCS = 39; // select the analog conversion clock
024    // ADCON3bits.ADCS = 0x3F;
025    ADCON3bits.SAMC = 14; // Auto conversion time = 14 x TAD
026 
027    // ADCON3bits.ADRC = 0; // Clock derived from system clock
028    ADCON2 = 0; // Clear the A/D Control Register 2; SetsVr+ to AVdd and Vr- to AVss
029    // Do not scan inputs
030    // Interrupts at the completion of conversion for each sample/convert sequence
031    // Buffer configured as one 16-word buffer
032    (AD1BUFn<15:0>)
033    // Always uses MUX A input multiplexer settings
034    // ADCON2 = 0x300;
035    // ADCON2 = 0x300; //MC EXAMP
036    // ADCON2 = 0x0308;
037    ADCON2bits.SMPI = 1; // interrupt after 1 in the buffer
038    // ADCON2bits.CSCNA = 1;
039
ADCSSL = 0x38;

ADCON1 = 0x8;

ADCON1 = 0x00EC; // MC EXAMP

ADCON1bits.ASAM = 1; // Sampling begins immediately after last conversion completes. SAMP bit is auto-set.
ADCON1bits.SSRC = 7; // Internal counter ends sampling and starts conversion (auto-convert)
ADCON1bits.FORM = 0; // Data format = simple 10-bit integer
ADCON1bits.ADSIDL = 1; // Stop in Idle Mode
ADCON1bits.ADON = 1; // ADC Module is Enabled

ADCON1bits.SIMSAM = 1; // Simultaneous samples

IPC3bits.ADIP = 5; // Select A/D interrupt priority

IFS0bits.ADIF = 0; // Clear interrupt flag
IEC0bits.ADIE = 1; // Enable ADC1 interrupt
ADCON1bits.SAMP = 1; // Sampling begins
A.8 a2d.h

```c
#ifndef __a2d_h__
#define __a2d_h__

#include "p30f4013.h"

void init_ad();

#endif/__a2d_h__
```

011
A.9 slave2master.c

```c
#include "p30f4013.h"
#include "slave2master.h"

void init_slave2master()
{
    //using uart2 module, pins 27(tx) and 28(rx)
    U2MODEbits.PDSEL = 0b00; //set parity
    /*PDSEL<1:0> value meaning (binary values)
    00 8-bit data, no parity
    01 8-bit data, even parity
    10 8-bit data, odd parity
    11 9-bit data, no parity
    */
    U2MODEbits.STSEL = 0; //set stop bits to one
    // U2BRG = 376; //set baud rate to 1200 through BRG
    U2BRG = 47; //set baud rate to 9600 through BRG
    // IFS1bits.U2RXIF=0; //clear UART1 receiver interrupt flag
    // IEC1bits.U2RXIE=1; //enable UART1 receiver ISR
    U2MODEbits.UARTEN = 1; //turn on uart 2 module
    U2STAbits.UTXEN = 1; //enable transmission
    /* PORTBbits.RB9 = 1; //set reset to disabled for wireless module(active low)
    //reset the unit
    PORTBbits.RB9 = 0; //active low reset
    int i;
    for(i=0;i<32000;i++); //delay
    PORTBbits.RB9 = 1; //finish reset cycle
    for(i=0;i<32000;i++); //delay let chip come up before transmitting
    */

    //unsigned char test = 0x00;
    //test write
    //while(1){
    U2TXREG = 0x55; //write dummy data
    //while(1);
    int k;
    k=1;
    // }
}
```

044
045
A.10  slave2master.h

```c
#ifndef __slave2master_h__
#define __slave2master_h__

// API for slave to master communication

void init_slave2master();

#endif/__slave2master_h__
```

A.11 wireless.c

```c
#include "wireless.h"

void init_wireless()
{
    //using uart2 module, pins 27(tx) and 28(rx)
    U2MODEbits.PDSEL = 0b00; //set parity
    /*PDSEL<1:0> value meaning (binary values)
    00 8-bit data, no parity
    01 8-bit data, even parity
    10 8-bit data, odd parity
    11 9-bit data, no parity*/
    U2MODEbits.STSEL = 0; //set stop bits to one
    U2BRG = 376; //set baud rate to 1200 through BRG
    U2MODEbits.UARTEN = 1; //turn on uart 2 module
    U2STAbits.U TXEN = 1; //enable transmission
    PORTBbits.RB9 = 1; //set reset to disabled for wireless module (active low)
    //reset the unit
    PORTBbits.RB9 = 0; //active low reset
    int i;
    for(i=0;i<32000;i++); //delay
    PORTBbits.RB9 = 1; //finish reset cycle
    for(i=0;i<32000;i++); //delay let chip come up before transmitting
    unsigned char test = 0x00;
    //test write
    while(1){
        U2TXREG = 0x55; //write dummy data
    }
}
```
A.12  wireless.h

001 #ifndef __wireless_h__
002 #define __wireless_h__
003
004 #include "p30f4013.h"
005
006 //******************************
007 // wireless API
008 //******************************
009 void init_wireless();
010
011 #endif //__wireless_h__
012
013
B MATLAB Code

B.1 Wave File Analysis

001 % Ehsun Siddiqui
002 % MQP Project: Smart Acoustic Whale Monitor
003 % Created: Mar 4, 2008
004 % Last Modified: Apr 23, 2008
005
006 % WAVE FILE ANALYSIS
007 % This program is a user friendly interface that allows the user to
008 % read in a wave file and manipulate it with an IIR Band-Pass Filter.
009
010 % NOTES:
011 % The program MUST be run under the directory where the wavefile is
012 % located. The directory must contain only wave files.
013
014 files = dir;
015 reply = 100;
016 namelen = 6;
017 while (1)
018     fileno = input('What file number? [1-76] ');
019     fprintf(1,'%s
',files(fileno+2).name(1:namelen))
020
021     [y,fs] = wavread(files(fileno+2).name);
022
023     % Calculate frequency response
024     d = 1/fs;
025     t = 0:d:(length(y)-1)*d;
026     z = abs(fft(y));
027     s = z(1:length(y)/2);
028     df = fs/length(z);
029     f = 0:df:df*(length(s)-1);
030
031     reply = 100;
032     while (reply ~= 0)
033         fprintf('%s
','
');
034         fprintf('%s
','
');
035         fprintf('%s
','
');
036         fprintf('%s
','What would you like to do?');
037         fprintf('%s
',' 0. New File');
038         fprintf('%s
',' 1. Show Frequency Plot');
039         fprintf('%s
',' 2. Save Frequency Plot');
040         fprintf('%s
',' 3. Show Spectrogram');
041         fprintf('%s
',' 4. Save Spectrogram');
042         fprintf('%s
',' 5. Play Sound');

fprintf('%s
', ' 6. Apply Filter');
fprintf('%s
', '');
reply = input('Enter 0-6: ');
if (reply == 1) % Show Frequency Plot
    plot(f,s);
end
if (reply == 2) % Save Frequency Plot
    saveas(plot(f,s),files(fileno+2).name(1:namelen),'jpg');
end
if (reply == 3) % Show Spectrogram
    [B,fst,tst]=specgram(y,1024,fs,256,192); % calculate the table of amplitudes
    bmin=max(max(abs(B)))/300; % calculate amplitude 50dB down from maximum
    imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin)); % plot top 50dB as image
end
if (reply == 4) % Save Spectrogram
    saveas(imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin)),files(fileno+2).name(1:namelen),'jpg');
end
if (reply == 5) % Play Sound
    sound(y)
end
if (reply == 6) % Apply Filter
    while (reply ~= 0)
        low = input('Low Frequency? '); % Read in low frequency
        high = input('High Frequency? '); % Read in high frequency
        fprintf('%s
', '');
        fprintf('%s
', 'Filtering...');
    end
    reply = 100;
    % BAND PASS Butterworth filter
    w1 = low/(fs/2);
    w2 = high/(fs/2);
    w = [w1 w2];
    [a b] = butter(6,w,'bandpass');
    yf = filter(a,b,y);
    % I CUT OUT OLD FILTERING TECHNIQUE. SEE MISC.m FILE
    % Calculate Frequency Response after filtering
    z2 = abs(fft(yf));
    df2 = fs/length(z2);
s2 = z2(1:length(yf)/2);
f2 = 0:df2:df2*(length(s2)-1);

while ((reply ~= 0) && (reply ~= 9))
    fprintf('%s
','What would you like to do?');
    fprintf('%s
',' 0. New File');
    fprintf('%s
',' 1. Show Frequency Plot of New Signal');
    fprintf('%s
',' 2. Show Spectrogram of New Signal');
    fprintf('%s
',' 3. Show Frequency Plot of Old Signal');
    fprintf('%s
',' 4. Show Spectrogram of Old Signal');
    fprintf('%s
',' 5. Show Frequency Response of Band Pass Filter');
    fprintf('%s
',' 6. Play New Sound');
    fprintf('%s
',' 7. Play Original Sound');
    fprintf('%s
',' 8. Apply a new filter');
    fprintf('%s
', '');
    reply = input('Enter 0-8: ');

    if (reply == 1) % Frequency Plot of New Signal
        plot(f2,s2);
    end

    if (reply == 2) % Spectrogram of New Signal
        [B,fst,tst]=specgram(yf,1024,fs,256,192); % calculate the table of amplitudes
        bmin=max(max(abs(B)))/300; % calculate amplitude 50dB down from maximum
        imagesc(t,-fst,20*log10(max(abs(B),bmin))/bmin)); % plot top 50dB as image
    end

    if (reply == 3) % Frequency Plot of Old Signal
        plot(f,s);
    end

    if (reply == 4) % Spectrogram of Old Signal
        [B,fst,tst]=specgram(y,1024,fs,256,192); % calculate the table of amplitudes
        bmin=max(max(abs(B)))/300; % calculate amplitude 50dB down from maximum
        imagesc(t,-fst,20*log10(max(abs(B),bmin))/bmin)); % plot top 50dB as image
    end

    if (reply == 5) % Frequency Response of Band Pass Filter
        freqz(a,b);
    end
if (reply == 6) % Play New Sound
    sound(yf)
end

if (reply == 7) % Play Original Sound
    sound(y)
end

fprintf('%s
', ' ');
fprintf('%s
', ' ');
fprintf('%s
', ' ');
end
end

% END LOOP
B.2 Moving Average

001 % Ehsun Siddiqui
002 % MQP Project: Smart Acoustic Whale Monitor
003 % Created: Mar 21, 2008
004 % Last Modified: Apr 23, 2008
005
006 % SMOOTHING AND AVERAGING
007 % This program is a user friendly interface for testing with moving
008 % averages and exponential averages. A wave file is read in, either
009 % exponential or moving average is selected, and a coefficient is
010 % entered. The resulting threshold is displayed along with the
011 % original signal.
012
013 % NOTES:
014 % The program MUST be run under the directory where the wavefile is
015 % located. The directory must contain only wave files.
016
017 files = dir;
018 reply = 100;
019 namelen = 6;
020 while (1)
021    fileno = input('What file number? [1-78] ');
022    fprintf(1,'%s
',files(fileno+2).name(1:namelen))
023
024    [y fs] = wavread(files(fileno+2).name);
025
026    while (reply ~= 0)
027       fprintf('%s
', ' ');  
028       fprintf('%s
', ' ');  
029       fprintf('%s
', 'What would you like to do?');
030       fprintf('%s
', ' 0. New File');
031       fprintf('%s
', ' 1. Apply MOVING Average');
032       fprintf('%s
', ' 2. Apply EXPONENTIAL Average');
033       fprintf('%s
', ' ');  
034       reply = input('Enter 0-2: ');
035
036       if (reply == 1)
037          n = input('Enter window width: ');
038          fprintf('%s
', 'Calculating...')
039          for c = n+1:length(y)-n   % Applying Moving Average
040             sum = 0;
041             for c2 = (c-n):(c+n)
042                sum = sum + abs(y(c2));
043          end
044         end
045
046         if (reply == 2)
047            beta = input('Enter exponential coefficient: ');
048            fprintf('%s
', 'Calculating...');
049            for c = 0:length(y)-1   % Applying Exponential Average
050               sum = (1 - beta)*sum + beta*y(c);
051            end
052            disp(sum)
053         end
054
055         if (reply == 3)
056            fprintf('%s
', 'Calculating...');
057            for c = 0:length(y)-1   % Applying Exponential Average
058               sum = (1 - beta)*sum + beta*y(c);
059            end
060            disp(sum)
061         end
062
063         if (reply == 4)
064            fprintf('%s
', 'Calculating...');
065            for c = 0:length(y)-1   % Applying Exponential Average
066               sum = (1 - beta)*sum + beta*y(c);
067            end
068            disp(sum)
069         end
070
071         if (reply == 5)
072            fprintf('%s
', 'Calculating...');
073            for c = 0:length(y)-1   % Applying Exponential Average
074               sum = (1 - beta)*sum + beta*y(c);
075            end
076            disp(sum)
077         end
078
079         if (reply == 6)
080            fprintf('%s
', 'Calculating...');
081            for c = 0:length(y)-1   % Applying Exponential Average
082               sum = (1 - beta)*sum + beta*y(c);
083            end
084            disp(sum)
085         end
086
087         if (reply == 7)
088            fprintf('%s
', 'Calculating...');
089            for c = 0:length(y)-1   % Applying Exponential Average
090               sum = (1 - beta)*sum + beta*y(c);
091            end
092            disp(sum)
093         end
094
095         if (reply == 8)
096            fprintf('%s
', 'Calculating...');
097            for c = 0:length(y)-1   % Applying Exponential Average
098               sum = (1 - beta)*sum + beta*y(c);
099            end
100            disp(sum)
101         end
102
103         if (reply == 9)
104            fprintf('%s
', 'Calculating...');
105            for c = 0:length(y)-1   % Applying Exponential Average
106               sum = (1 - beta)*sum + beta*y(c);
107            end
108            disp(sum)
109         end
110
111         if (reply == 10)
112            fprintf('%s
', 'Calculating...');
113            for c = 0:length(y)-1   % Applying Exponential Average
104             sum = sum + abs(y(c));
105             end
106            disp(sum)
107         end
108
109         if (reply == 11)
110            fprintf('%s
', 'Calculating...');
111            for c = 0:length(y)-1   % Applying Exponential Average
112               sum = sum + abs(y(c));
113            end
114            disp(sum)
115         end
116
117         if (reply == 12)
118            fprintf('%s
', 'Calculating...');
119            for c = 0:length(y)-1   % Applying Exponential Average
120               sum = sum + abs(y(c));
121            end
122            disp(sum)
123         end
124
125         if (reply == 13)
126            fprintf('%s
', 'Calculating...');
127            for c = 0:length(y)-1   % Applying Exponential Average
128               sum = sum + abs(y(c));
129            end
130            disp(sum)
131         end
132
133         if (reply == 14)
134            fprintf('%s
', 'Calculating...');
135            for c = 0:length(y)-1   % Applying Exponential Average
136               sum = sum + abs(y(c));
137            end
138            disp(sum)
139         end
140
141         end
142    end
avg = sum/(2*n+1);
yk(c) = avg + .075;
end
plot(y);grid
hold
plot(yk,'r');grid
hold
end

if (reply == 2)
x = input('Enter coefficient: ');
fprintf('%s\n', 'Calculating...');
yn = 0;
for n = 2:length(y)  % Applying Exponential Average
    yn(n) = (x*abs(yn(n-1)) + (1-x)*abs(y(n)));
end
yn = yn + .075;
plot(y);grid
hold
plot(yn,'r');grid
hold
end
end
end
end
B.3 Matched Filtering

001 % Ehsun Siddiqui
002 % MQP Project: Smart Acoustic Whale Monitor
003 % Created: Apr 14, 2008
004 % Last Modified: Apr 23, 2008
005
006 % MATCHED FILTERING
007 % This program reads in a wave file specified by the user, and runs
008 % two different filters on it. Based on signal energy, it determines
009 % the best-matched filter. The cut-off frequencies can be altered
010 % within the program.
011
012 % NOTES:
013 % The program MUST be run under the directory where the wavefile is
014 % located. The directory must contain only wave files.
015
016 files = dir;
017 reply = 100;
018 namelen = 6;
019 while (1)
020    fileno = input('What file number? [1-76] ');
021    fprintf(1,'%s
',files(fileno+2).name(1:namelen))
022
023    [y,fs] = wavread(files(fileno+2).name);
024
025    % Calculate frequency response
026    d = 1/fs;
027    t = 0:d:(length(y)-1)*d;
028    z = abs(fft(y));
029    s = z(1:length(y)/2);
030    df = fs/length(z);
031    f = 0:df:df*(length(s)-1);
032
033    % Preconditioning Filter
034    w = 300/(fs/2);
035    [a b] = butter(6,w,’high’);
036    yprecon = filter(a,b,y);
037
038    % Filter 1 - 400 Hz to 2200 Hz
039    w1 = 400/(fs/2);
040    w2 = 2300/(fs/2);
041    w = [w1 w2];
042    [a b] = butter(6,w,’bandpass’);
043    y1 = filter(a,b,yprecon);
% Filter 2 - 2000 Hz to 3800 Hz
w1 = 2000/(fs/2);
w2 = 3800/(fs/2);
w = [w1 w2];
[a b] = butter(6,w,'bandpass');
y2 = filter(a,b,yprecon);

% Energy Calculation
energylow = 0;
for i=1:length(y1)
    energylow = energylow + (y1(i))^2;
end
energyhigh = 0;
for i=1:length(y2)
    energyhigh = energyhigh + (y2(i))^2;
end
energylow
energyhigh

if (energylow < energyhigh)
    [B,fst,tst]=specgram(y1,1024,fs,256,192); % calculate the table of amplitudes
    bmin=max(max(abs(B)))/300; % calculate amplitude 50dB down from maximum
    imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));
    fprintf('%s
', 'Low Filter Applied.');
end
if (energyhigh < energylow)
    [B,fst,tst]=specgram(y2,1024,fs,256,192); % calculate the table of amplitudes
    bmin=max(max(abs(B)))/300; % calculate amplitude 50dB down from maximum
    imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));
    fprintf('%s
', 'High Filter Applied.');
end

reply = 100;
while (reply ~= 0)
    fprintf('%s
', 'What would you like to do?');
    fprintf('%s
', ' 0. New File');
    fprintf('%s
', ' 1. Original Spectrogram');
    fprintf('%s
', ' 2. First Filter Spectrogram');
fprintf('%s
', '3. Second Filter Spectrogram');
fprintf('%s
', '4. Preconditioning Filter Spectrogram');
reply = input('Enter: ');

if (reply == 1)  % Show Original Spec
  [B,fst,tst]=specgram(y,1024,fs,256,192);  % calculate the table of amplitudes
  bmin=max(max(abs(B)))/300;  % calculate amplitude 50dB down from maximum
  imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));  % plot top 50dB as image
end

if (reply == 2)  % Show First Filter Spec
  [B,fst,tst]=specgram(y1,1024,fs,256,192);  % calculate the table of amplitudes
  bmin=max(max(abs(B)))/300;  % calculate amplitude 50dB down from maximum
  imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));  % plot top 50dB as image
end

if (reply == 3)  % Show Second Filter Spec
  [B,fst,tst]=specgram(y2,1024,fs,256,192);  % calculate the table of amplitudes
  bmin=max(max(abs(B)))/300;  % calculate amplitude 50dB down from maximum
  imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));  % plot top 50dB as image
end

if (reply == 4)  % Show Precon Filter Spec
  [B,fst,tst]=specgram(yprecon,1024,fs,256,192);  % calculate the table of amplitudes
  bmin=max(max(abs(B)))/300;  % calculate amplitude 50dB down from maximum
  imagesc(t,-fst,20*log10(max(abs(B),bmin)/bmin));  % plot top 50dB as image
end
C Adapted Code and Schematics

C.1 SD Card Firmware

// SD Card Read Test
// For use on dsPIC30F4013 at 3.3VDC

// Copyright (c) 2005, K9spud LLC.
// http://www.k9spud.com/sdcard/

// This program is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License
// as published by the Free Software Foundation; either version 2
// of the License, or (at your option) any later version.

// This program is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.

// You should have received a copy of the GNU General Public License
// along with this program; if not, write to the Free Software
// Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307,
// USA.

#ifndef SDTEST_H
#define SDTEST_H
#endif

// Revision 1.2 2005/09/09 04:26:23 edwards
// Comments updated.

// Revision 1.1 2005/09/07 07:07:18 edwards
// First successful read from SD Card on my dsPIC30F4013!

#include "p30f4013.h"

#define LED PORTBbits.RB0
#define LED_DIR TRISBbits.TRISB0
#define SD_PWR PORTBbits.RB1
#define SD_PWR_DIR TRISBbits.TRISB1
#define SD_CS PORTBbits.RB3
```c
042 #define SD_CS_DIR TRISBbits.TRISB3
043
044 #define SDI PORTFbits.RF2
045 #define SDI_DIR TRISFbits.TRISF2
046 #define SCK PORTFbits.RF6
047 #define SCK_DIR TRISFbits.TRISF6
048 #define SDO PORTFbits.RF3
049 #define SDO_DIR TRISFbits.TRISF3
050
051 #define SD_PowerOn() SD_PWR = 1
052 #define SD_PowerOff() SD_PWR = 0
053 
054 #define SD_Enable() SD_CS = 0 /* set low to activate SD Card chip select */
055 #define SD_Disable() SD_CS = 1 /* set high to deactivate SD Card chip select */
056
057 void SPIWrite(unsigned char data)
058 {
059     // DO NOT WAIT FOR SPITBF TO BE CLEAR HERE
060     // (for some reason, it doesn’t work on this side of the write data).
061     // Write the data!
062     SPI1BUF = data;
063     // Wait until send buffer is ready for more data.
064     while(SPI1STATbits.SPITBF);
065 }
066
067 unsigned char SPIRead(void)
068 {
069     unsigned char data;
070     if(SPI1STATbits.SPIRBF)
071     {
072         // already have some data to return, don’t initiate a read
073         data = SPI1BUF;
074         SPI1STATbits.SPIROV = 0;
075         return data;
076     }
077     // We don’t have any data to read yet, so initiate a read
078     SPI1BUF = 0xFF; // write dummy data to initiate an SPI read
079     while(SPI1STATbits.SPITBF); // wait until the data is finished reading
080     data = SPI1BUF;
081     SPI1STATbits.SPIROV = 0;
```
087    return data;
088 }
089
090 void InitSPI(void)
091 {
092    SD_PowerOff();
093    SD_PWR_DIR = 0; // output
094    SD_PowerOff();
095
096    SD_Disable();
097    SD_CS_DIR = 0; // output
098    SD_Disable();
099
100    SDI_DIR = 1; // input
101    SCK_DIR = 1;
102    SDO_DIR = 1;
103    SPI1STAT = 0x8000; // enable SPI port
104    // set SPI port to slowest setting
105    // master mode
106    // 8 bit
107    // Idle state for Clock is high level
108    // Primary prescaler 64:1
109    // Secondary prescaler 8:1
110    SPI1CON = 0x0060;
111  }
112
113  // primary prescaler 1:1
114  // secondary prescaler 1:1
115  #define SPIFastClock() SPI1CON = 0x007F
116
117  // SD Card SPI Commands are defined in the SanDisk SD Card Product Manual
118  // section 5.2.2.1 (page 91 of PDF)
120  //
121  // 0 - GO_IDLE_STATE - Resets the SD Card
122  // 1 - SEND_OP_COND - Activates the card’s initialization process
123  // 9 - SEND_CSD - Asks card to send card-specific data
124  // 10 - SEND_CID - Asks card to send card identification
125  // 12 - STOP_TRANSMISSION - Forces card to stop transmission during multi-block read
126  // 13 - SEND_STATUS - Asks card to send its status register.
127  // 16 - SET_BLOCKLEN - Selects block length for all subsequent block commands (default is 512)
130 // 17 - READ_SINGLE_BLOCK  Reads a block of the size specified by SET_BLOCKLEN
131 // 18 - READ_MULTIPLE_BLOCK Continuously transfers data until interrupted by STOP_TRANSITION
132 // 24 - WRITE_BLOCK  Writes a block of the size specified by SET_BLOCKLEN
133 // 25 - WRITE_MULTI_BLOCK  Continuously writes blocks of data until a stop transmission token is sent instead of start block token.
134
135 // R1 Response Codes (from SD Card Product Manual v1.9 section 5.2.3.1)
136 #define R1_IN_IDLE_STATE  (1<<0)  // The card is in idle state and running initializing process.
137 #define R1_ERASE_RESET  (1<<1)  // An erase sequence was cleared before executing because of an out of erase sequence command was received.
138 #define R1_ILLEGAL_COMMAND  (1<<2)  // An illegal command code was detected
139 #define R1_COM_CRC_ERROR  (1<<3)  // The CRC check of the last command failed.
140 #define R1_ERASE_SEQ_ERROR  (1<<4)  // An error in the sequence of erase commands occurred.
141 #define R1_ADDRESS_ERROR  (1<<5)  // A misaligned address, which did not match the block length was used in the command.
142 #define R1_PARAMETER  (1<<6)  // The command's argument (e.g. address, block length) was out of the allowed range for this card.
143 // R1 bit (1<<7) is always 0
144 unsigned char SD_WriteCommand(unsigned char* cmd)
145 {
146    unsigned int i;
147    unsigned char response;
148    unsigned char savedSD_CS = SD_CS;
149    // SD Card Command Format
150    // (from Section 5.2.1 of SanDisk SD Card Product Manual v1.9).
151    // Frame 7 = 0
152    // Frame 6 = 1
153    // Command (6 bits)
154    // Address (32 bits)
155    // Frame 0 = 1
156    // Set the framing bits correctly (never change)
157    cmd[0] |= (1<<6);
158    cmd[0] &~ (1<<7);
159    cmd[5] |= (1<<0);
160    // Send the 6 byte command
161    SD_Enable();
162    for(i = 0; i < 6; ++i)
163       {
SPIWrite(*cmd);
cmd++;
}

// Wait for the response
i = 0;
do {
  response = SPIRead();
  if(i > 100)
    break;
i++;
} while(response == 0xFF);
SD_Disable();

// Following any command, the SD Card needs 8 clocks to finish up its work.
// (from SanDisk SD Card Product Manual v1.9 section 5.1.8)
SPIWrite(0xFF);

SD_CS = savedSD_CS;
return(response);
}

unsigned char InitSD()
{
  unsigned int i = 0;
  unsigned char status;
  // Turn off SD Card
  SD_Disable();
  SD_PowerOff();
  // Wait for power to really go down
  for(i = 0; i; i++);
  for(i = 0; i; i++);
  for(i = 0; i; i++);
  for(i = 0; i; i++);
  // Turn on SD Card
  SD_PowerOn();
  // Wait for power to really come up
for(status = 0; status < 10; ++status)
{
    for(i = 0; i; i++);
    for(i = 0; i; i++);
    for(i = 0; i; i++);
    for(i = 0; i; i++);
}

// We need to give SD Card about a hundred clock cycles to boot up
for(i = 0; i < 16; ++i)
{
    SPIWrite(0xFF); // write dummy data to pump clock signal line
}

SD_Enable();

// This is the only command required to have a valid CRC
// After this command, CRC values are ignore unless explicitly enabled
using CMD9
unsigned char CMD0_GO_IDLE_STATE[] = {0x00,0x00,0x00,0x00,0x00,0x95};

// Wait for the SD Card to go into IDLE state
i = 0;
do
{
    status = SD_WriteCommand(CMD0_GO_IDLE_STATE);
    // fail and return
    if(i++ > 50)
    {
        return 1;
    }
} while( status != 0x01 );

// Wait for SD Card to initialize
unsigned char CMD1_SEND_OP_COND[] = {0x01,0x00,0x00,0x00,0x00,0xFF};
i = 0;
do
{
    status = SD_WriteCommand(CMD1_SEND_OP_COND);
    if(i++ > 50)
    {
        return 2;
    }
} while( (status & R1_IN_IDLE_STATE) != 0 );
Adapted Code and Schematics

258  // Send CMD55, required to precede all "application specific" commands
259  unsigned char CMD55_APP_CMD[] = {55,0x00,0x00,0x00,0x00,0xFF};
260  status = SD_WriteCommand(CMD55_APP_CMD); // Do not check response here
261  
262  // Send the ACMD41 command to initialize SD Card mode (not supported by MMC cards)
263  i = 0;
264  unsigned char ACMD41_SD_SEND_OP_COND[] = {41,0x00,0x00,0x00,0x00,0xFF};
265  do
266    {
267      status = SD_WriteCommand(ACMD41_SD_SEND_OP_COND);
268      // Might return 0x04 for Invalid Command if MMC card is connected
269      if(i++ > 50)
270        {
271          return 3;
272        }
273    } while( (status & R1_IN_IDLE_STATE) != 0 );
274  
275  // Set the SPI bus to full speed now that SD Card is initialized in SPI mode
276  SD_Disable();
277  SPIFastClock();
278  
279  return 0;
280  }
281  
282  // SD Card defaults to 512 byte block size
283  #define BLOCK_SIZE 512
284  unsigned char SD_ReadBlock(unsigned long addr, unsigned char ∗buf)
285  {
286    unsigned int i;
287    unsigned char status;
288    
289    unsigned char CMD17_READ_SINGLE_BLOCK[] = {17,0x00,0x00,0x00,0x00,0xFF};
290    CMD17_READ_SINGLE_BLOCK[1] = ((addr & 0xFF000000) >> 24);
291    CMD17_READ_SINGLE_BLOCK[2] = ((addr & 0x00FFFFFF) >> 16);
292    CMD17_READ_SINGLE_BLOCK[3] = ((addr & 0x0000FFFF) >> 8);
293    CMD17_READ_SINGLE_BLOCK[4] = ((addr & 0x000000FF));
294    
295    SD_Enable();
296    
297    // Send the read command
298    status = SD_WriteCommand(CMD17_READ_SINGLE_BLOCK);
299    if(status != 0)
{ // ABORT: invalid response for read single command
  return 1;
}

// Now wait for the "Start Block" token (0xFE)
// (see SanDisk SD Card Product Manual v1.9 section 5.2.4. Data Tokens)
do{
  status = SPIRead();
} while(status != 0xFE);

// Read off all the bytes in the block
for(i = 0; i < BLOCK_SIZE; ++i)
{
  status = SPIRead();
  *buf = status;
  buf++;
}

// Read CRC bytes
status = SPIRead();
status = SPIRead();

SD_Disable();

// Following a read transaction, the SD Card needs 8 clocks after the end
// bit of the last data block to finish up its work.
// (from SanDisk SD Card Product Manual v1.9 section 5.1.8)
SPIWrite(0xFF);

return 0;
}

unsigned char buf[512];

int main (void)
{
  unsigned int i, j;
  ADPCFG = 0xFFFF; // Force all ADC pins as digital I/O
  // Configure output pins
  LED_DIR = 0;
  LED = 1;
  InitSPI();
unsigned char status;
status = InitSD();
do {
    while(status)
    {
        status = InitSD();
        LED = 0;
    }
    status = SD_ReadBlock(0, buf);
} while(status);

i = 0; //reset counter
while(1)
{
    LED = i & 1;
    if(i & 1)
    {
        for(j = 0; j < 40000; ++j);
    }
    else
    {
        for(j = 0; j < 65000; ++j);
    }
    i++;
    if(i == 15)
    {
        i = 0;
    }
}
return 0;
C.2 SD Card Schematics

Fig. 26: SD Card Example Interface Schematics
D Additional Code

D.1 Sound Sample Processing

```cpp
#include <fstream>
#include <stdio.h>

int main()
{
    FILE *fi,*fo;
    fi=fopen("./test","rb");
    int i,j;
    while(1){
        i=fgetc(fi);
        j=fgetc(fi);
        //printf("%i, %i\n",i,j);
        if(!((i!=EOF)&&(j!=EOF)))
            break;
        int q=i+(j<<8);
        q-=2048;
        float out=(float)q/2048;
        printf("%f\n",out);
    }
    fclose(fi);
    return 0;
}
```