iSlide Avalanche Receiver

Final Report

Jacob Sanders
Sean Jennings
University of Utah

Table of Contents

Project Description 3
Project Scope 4
Hardware Design and Implementation 5
Antenna 5
Finding Direction 6
Finding Distance 8
Signal Amplification 9
Preparing the Signal for Digital Conversion 12
Software Design and Implementation 14
Analog to Digital Conversion 14
Digital Signal Processing 15
High Frequency Noise Filter 15
Peak Detection Algorithm 15
iPhone Interface 16
Risk Assessment Revisited 18
Bottle-Neck Design Process, Noise, and Wireless Inexperience 18
Improper Hardware Selection and Limitations of the Standard Frequency 19
Limited Public Documentation on the iPhone 19
Cost 19
Conclusion 20
References 21
Acknowledgements 23
Appendix A1 - IOS Source Code 24
iPhone GUI files 24
iPhone Serial files 31
Appendix A2 - MCU Source Code 34
Appendix A3 - Matlab Source Code 40
Appendix B - Bill of Materials 43
Appendix C1 - Complete Circuit Schematic 44
Appendix C2 - Photograph of Final Product 45

Project Description

The intent of our project was to design an avalanche beacon accessory and application for the iPhone. Avalanche beacons are used by winter recreationalists to locate buried victims in the case of an avalanche. The user wears the device during outdoor recreation and runs in a low-power transmit mode, emitting a brief 457 kHz radio pulse about once per second [1]. If an individual is buried in an avalanche, rescuers switch their beacons to receive mode and use them in conjunction with other search techniques [2] to locate, and hopefully rescue, buried victims.

Fig. 1 Prototype of iSlide on an iPhone 4.

Originally, avalanche beacons simply transformed the input signal into an auditory tone whose intensity fluctuated proportionally with the strength of the signal, leaving the signal processing to the ear and brain of the user [3]. Most beacons on the market today implement a visual display and some form of signal processing to approximate the distance and direction to the victim [4]. Our objectives with this project were to understand how these devices work, design a compatible receiver device, and advance the existing technology by interfacing with the iPhone.

Our interest in this project stems mainly from three sources. First, we share a common interest in embedded systems. This is essentially what motivated our microcontroller-based design over a fully analog design. Secondly, it seems that for virtually every task a person might want to perform, “there’s an app for that.”[5] We're interested in applying our knowledge of software engineering to the growing field of mobile computing. And finally, the analog component of this receiver was in uncharted territory for both of us, giving us an excellent opportunity to stretch our experience and understanding.

The essential hardware component of our project is a microprocessor-based receiver circuit with a 32-pin iPhone interface. The software component included the programing of the microprocessor as well as a user-friendly iPhone application. This report describes the process of researching and implementing our project. Though we faced many challenges, we were able to successfully implement the iSlide avalanche beacon receiver.

Scope

The primary standard to which current avalanche beacon manufacturers adhere is EN 300 178, published by the European Telecommunications Standards Institute (ETSI) [1]. ETSI establishes mechanical and electrical standards for the devices as well as acceptable methods for testing their various components and functions. For the purposes of this project, we didn't attempt total compliance with ETSI standards. We did, however, make our device compatible with others on the market by adhering to the following ETSI specifications:

- Operating frequency: 457 kHz
- Bandwidth: 160 Hz
- Carrier keying (see Fig. 2):
    - On time: 70 ms minimum
    - Off time: 400 ms minimum
    - Period: 1000 ms ± 300 ms

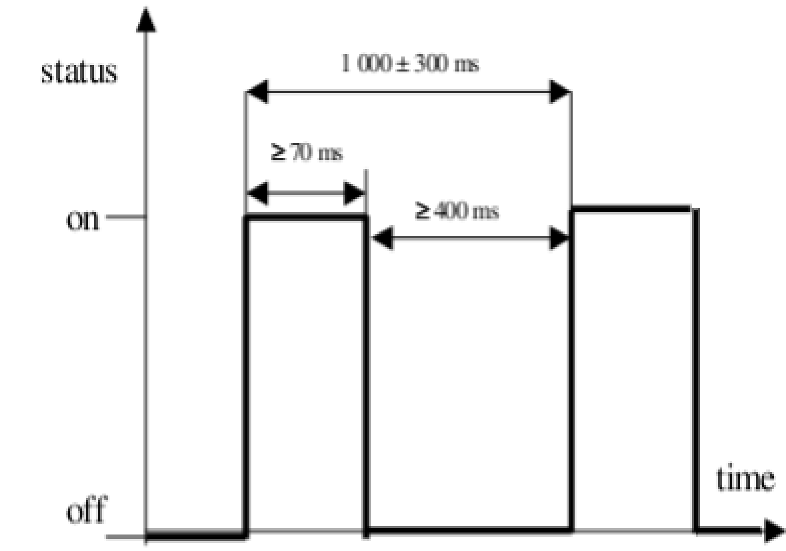


Fig. 2 Carrier keying requirements of ETSI EN 300 718.

Though there are several features we would have enjoyed implementing, the core requirement of our project was the design and implementation of a receiver, capable of directing the user to the source of a transmitting beacon. In the end, we found this to be challenging enough to consume our time and financial resources without implementing any additional frills.

Hardware Design and Implementation

We designed and simulated our analog circuitry using mathematical and graphical models in freeMat [6] and LTSpice [7]. The final circuit was was soldered on various interconnected prototyping boards as shown. The details of each component of the analog circuitry is outlined in the sections that follow.

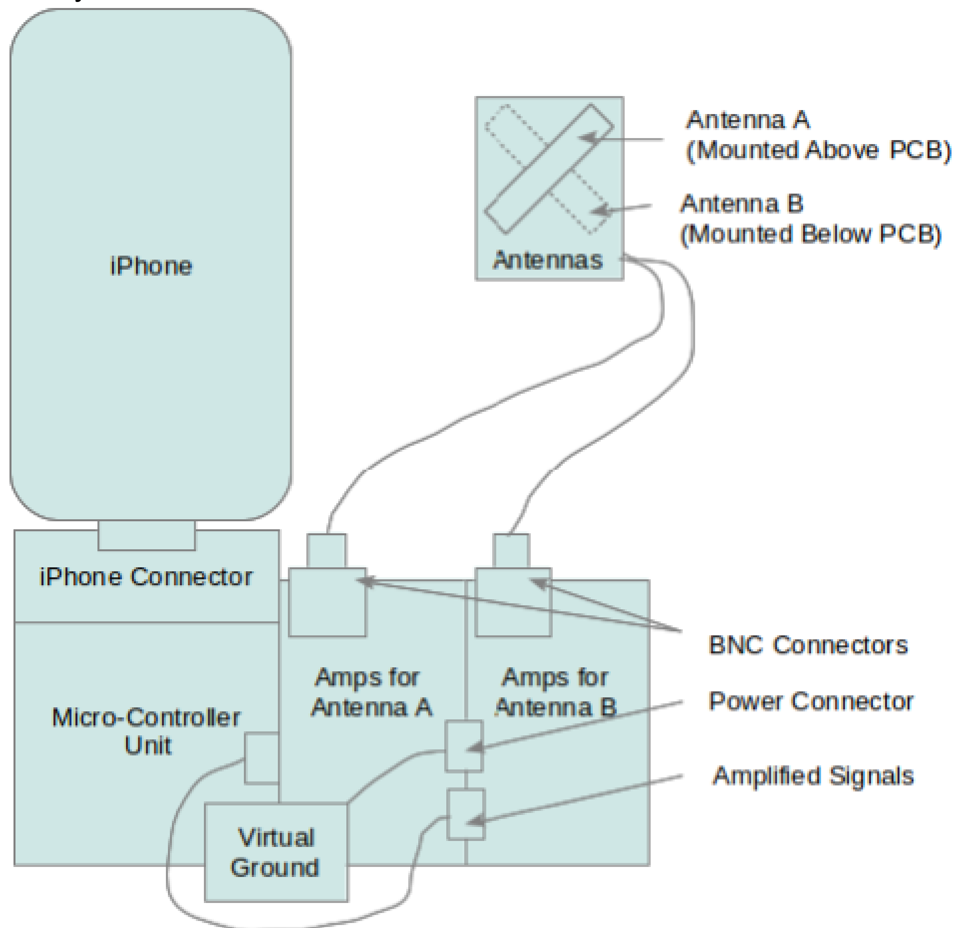


Fig. 3 High level circuit diagram.

Antenna

The key to our design is the compact and directional nature of our antennas. In general, the length of an antenna is on the same order of magnitude as the wave length of the desired frequency [8]. The wavelength of a 457-kHz signal is approximately 656 meters, [9] almost 6000 times bigger than the iPhone [10]. Obviously, a standard half- or quarter-wave antenna won't work in our design. AM radio receivers (which operate between 535-kHz and 1605-kHz [11]) address this problem using a loopstick antenna [12], which consists of an inductive coil wrapped around a ferrite rod. The ferrite rod responds to the magnetic portion of an electromagnetic signal, inducing a current on the coil [13]. The resonant frequency of the antenna can then be adjusted by adding

a capacitive impedance to form and LC circuit [14]. Though the resultant circuit may have a length of only a few centimeters, it's effective length (due to geometry, material properties, and tuning circuitry) can be much larger[15].

The resonant frequency, f, of an LC circuit is defined as:

$$C = 1 \; / \; L(2\pi f)^2 \qquad \text{Eq. 1}$$

The bandwidth, β, of the LC circuit can also be controlled by adding a parallel resistance, R, such that:

$$R = 1 \; / \; (C\beta) \qquad \text{Eq. 2}$$

There is a trade-off between gain and noise, however, since adding resistance to the circuit also adds noise [16]. One way to reduce resistance is to use litz wire to wind the antenna coil rather than a solid conductor, since litz wires has a lower DC resistance [17].

Finding Direction

Another useful characteristic of the loopstick antenna is it's figure-eight radiation pattern. The signal received by the antenna is stronger at either end of the rod and nulls toward the center. Thus, when pointing the antenna directly at a transmitter, the strength of the signal is strong, but when oriented perpendicularly the signal weakens significantly [17].

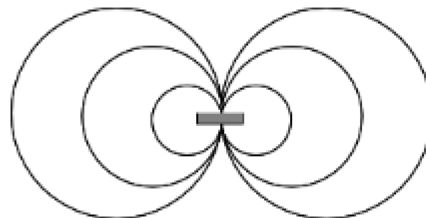


Fig.4 Figure-eight radiation pattern of a loopstick antenna.

Our design exploits this pattern, using two mutually orthogonal antennas, thus the null of one antenna aligns with the maximum of the other. Placing the antennas in an “x” orientation, such that each point away from the user at a 45˚ angle, we're able to direct the user along the figure-eight radiation pattern of a transmitting beacon. When the amplitudes of the signals from both antennas are equal, we direct the user forward. If either signal becomes stronger than the other, the user is directed to turn slightly in the direction that leads back to equilibrium.

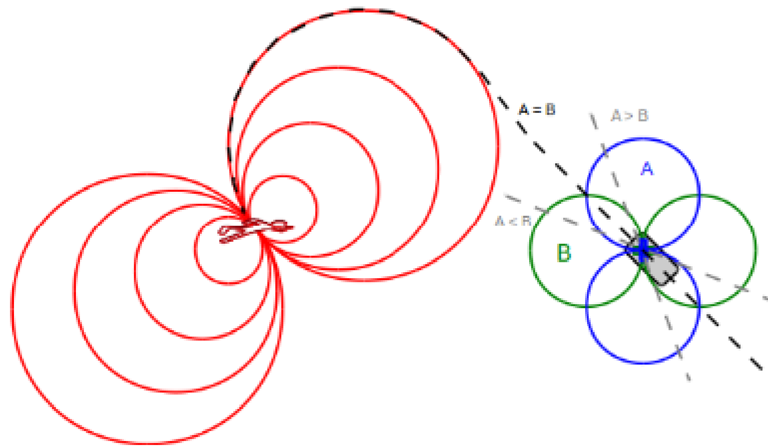


Fig. 5 Finding buried victims by exploiting the figure-eight radiation pattern of the loopstick antennas. A and B represent the amplitude of the signal detected by each of the two mutually orthogonal antennas.

We purchased several prefabricated 2.5” loopstick antennas wound using a very fine litz wire.

Fig. 6 Prefabricated loopstic antenna [30].

The nominal inductance of the antenna was 680-μH. We tuned them to 457-kHz using a parallel capacitance and resistance of 77-pF and 5.6-MΩ, respectively. Unfortunately the signal to noise ratio of the antennas wasn't very good. We theorized that two main issues caused the problem:

1. The large inductance of the antenna necessitated a very small capacitance value. This made it difficult to tune the antenna using discrete components (due to the imprecision of nominal values) and though a variable capacitor simplified the task significantly, it also introduced additional noise.

2. The inductive coil covered only 33% of the ferrite rod's surface area. Maximum efficiency is obtained by covering the entire length of the rod [18].

To overcome these issues, we rewound the antennas using thicker litz wire. Doing so, we were able to cover more of the rod's length using fewer windings, thus minimizing the inductance of the antenna. We also employed a winding method developed by the U.S. Army Signal Corps which produces better surface coverage and lower inductance. Rather than simply winding from one side of the rod to the other (as with the prefabricated antennas) we began the winding in the middle of the rod, winding counter

clockwise on one side and clockwise on the other and connecting the two ends of the winding to a common node. The result of this is equivalent to placing two inductors in parallel (thus reducing the inductance) while providing the efficiency of using the whole surface area of the rod [15].

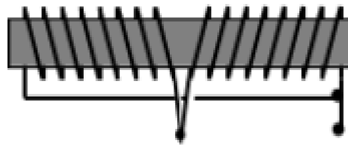


Fig. 7 An efficient method for winding loopstick antennas developed by the US Army.

These optimizations to the antenna improved our signal to noise ratio by a factor of six. Additional improvements would likely be needed to take the receiver to production, but this improvement was sufficient for our classroom demonstration, giving us a usable range of about 10-15 feet. Our custom antenna had an inductance of about 23-μH allowing for a more reasonable capacitance value of about 5300-pF and a resistance of 200-kΩ.

Finding Distance

We were also able to estimate the distance away from the transmitter using the amplitude of the signal received by each antenna. First, we compiled a table of measurements containing the head-on signal strength of each antenna at various distances from a transmitting beacon.

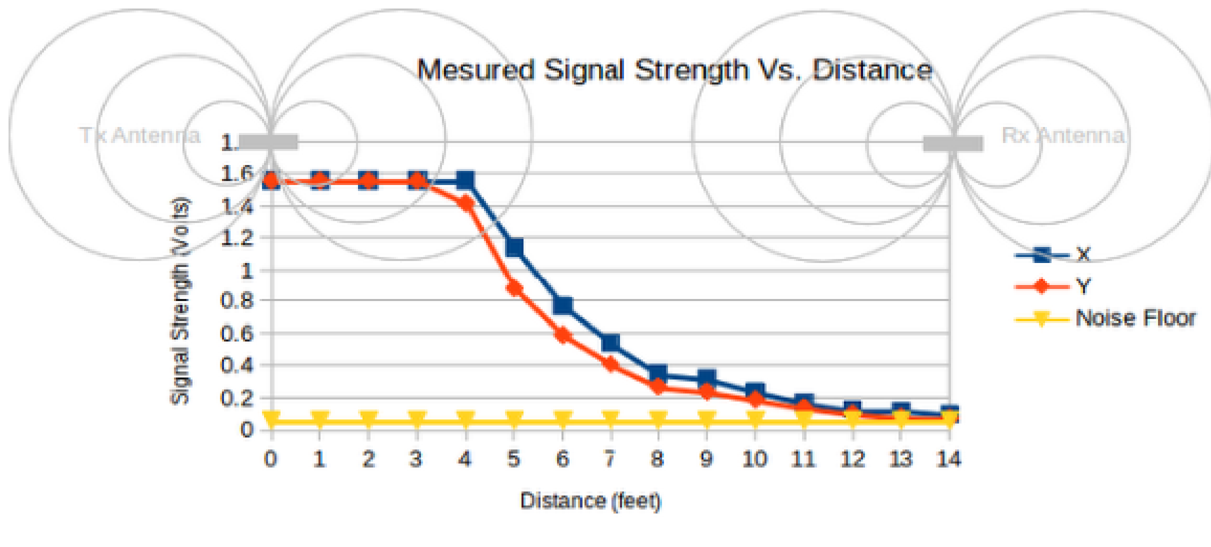


Fig. 8 Measured signal strength for each antenna at varying distances. Note that these measurements were taken using the second amplification stage which exceeds the positive supply rail at a distance of about 3 or 4 feet.

Fitting an equation to the curves of our measured values yields the following:

$$dist(v_x) = 100.8 * e^{-36.35 * vx} + 12.04 * e^{-0.526 * vx} \quad \text{Eq. 3}$$

$$dist(v_y) = 10.01 * e^{-8.127 * vy} + 8.509 * e^{-0.6099 * vy} \quad \text{Eq. 4}$$

The actual distance from the transmitter is then estimated using the Pythagorean theorem:

$$dist^2 = dist(x)^2 + dist(y)^2 \quad \text{Eq. 5}$$

Signal Amplification

As demonstrated in Fig. 8, signal strength decreases significantly with distance—it's inversely proportional to distance squared [19]. For example, a usable signal at 14 feet becomes completely saturated at 4 feet. To deal with this problem, we implemented 3 stages of signal amplification allowing the micro-controller to switch between stages as necessary. This feature was not fully implemented in our signal-processing algorithm, however, since only the second stage was really necessary for our classroom demonstration.

The major obstacle we faced in designing our amplifier stages was dealing with noise. The first stage of amplification is typically the most important in dealing with noise since the noise and signal gain from stage one is cascaded through each subsequent stage. It's important to have the highest gain and lowest noise possible on the first stage [20]. As demonstrated by Eq. 2, however, a narrow bandwidth requires a large input resistance on stage one, but adding resistance to the circuit also adds noise. Thus, by increasing resistance we increase noise but by decreasing resistance we increase the bandwidth allowing the antenna to detect more frequencies and in turn, increasing noise. To deal with this issue, we opted to use an instrumentation amplifier op amp configuration. The instrumentation amplifier provides high gain while rejecting common mode noise and providing the high input resistance inherent in the non-inverting input of the op amp, without adding additional resistors to the tuned LC circuit [21].

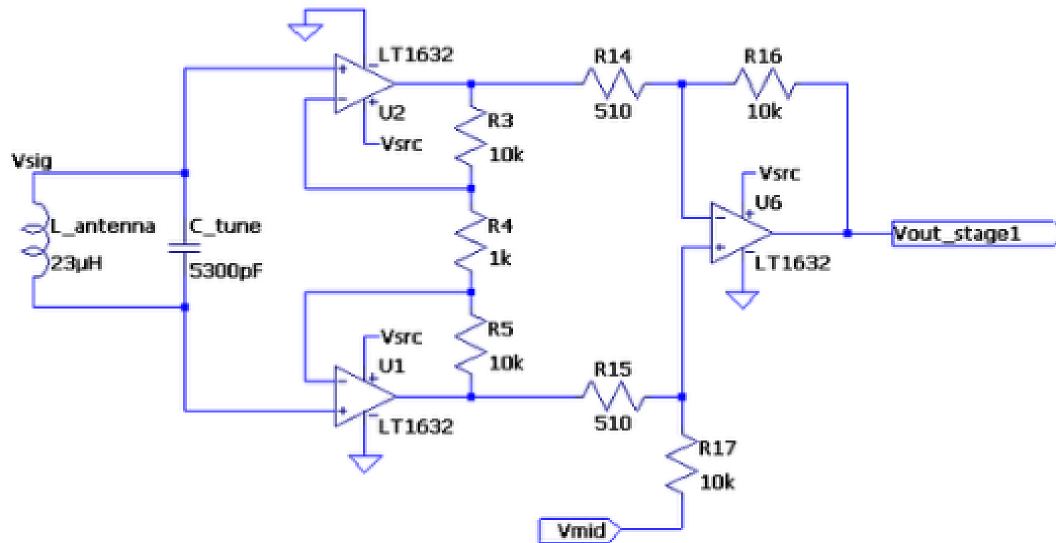


Fig. 9 Simple circuit schematic of our stage-one amplifier, including the tunned antenna. Vsrc = 3.3-V and Vmid = 1.65-V.

The total gain of this stage was about 400-V/V, as calculated using the following equation:

$$gain = (R16/R14)(1+2R3/R4) \qquad \text{Eq. 6}$$

Where, R3 and R5 are matched, as are R14 and R15, as well as R16 and R17. In order to accomplish this, we used resistors with a tolerance of only 1%. The op amps shown in Fig. 9 are configured almost to their maximum gain for that frequency. Our resistor values were chosen to be relatively small (without being impractically small) so as to minimize noise.

We chose to use the LT1632 op amp from Linear Technology because of it's high gain at 457-kHz, it's rail to rail output, and because it is easily configurable using a single power supply, allowing us to power the amplifiers using the iPhone's 3.3-V supply without a negative rail. In order to use a single supply, however, it was necessary to create a virtual ground at 1.65-V (half of the 3.3-V supply). This was done as follows [22]:

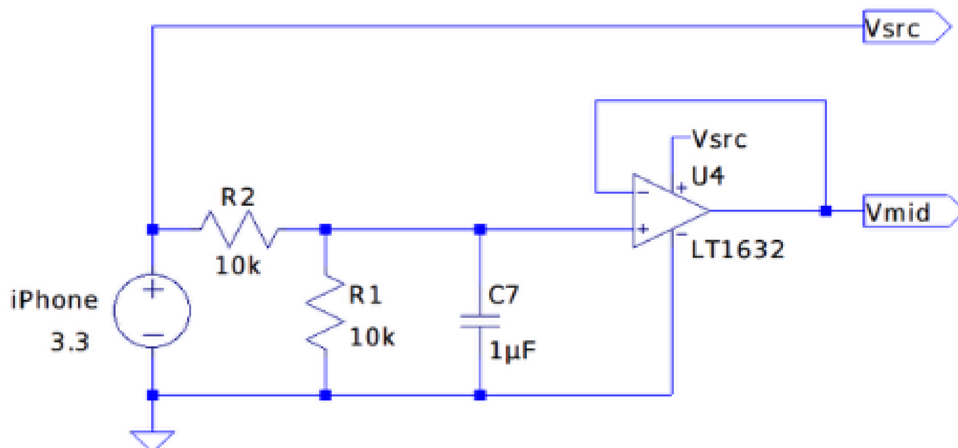


Fig. 10 This circuit creates a 1.65-V virtual ground for use by our single-supply op amps.

To further optimize the circuit and reduce noise, we added additional capacitance in conjunction with the resistors since a parallel RC (resistor-capacitor) circuit acts as a high pass filter, and a serial RC circuit acts as a low pass filter [23]. Using the following equation to calculate the capacitance value which provides a cutoff at 457-kHz, we then used a common rule of thumb, sizing the capacitors up a decade for low pass filters and down a decade for high pass filters ensuring that our signal would not be accidentally cut off.

$$C = 1 \,/\, (2R \cdot f) \qquad \text{Eq. 7}$$

We also separated the tuned antenna physically from the amplifier circuit using short coaxial cables and a BNC-type connector in order to avoid self-resonance between the antenna and the amplifier circuitry [24]. We added decoupling capacitors to the antenna and connected the virtual ground to it. This ensured that the antenna's oscillations would be centered at virtual ground. To reduce power supply noise, we connected a large capacitor across the source and ground terminals of each op amp package [25], resulting in the following circuit configuration:

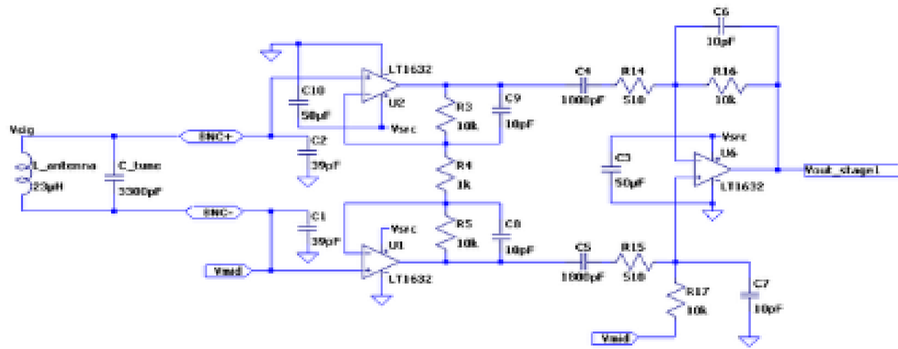


Fig. 11 Our final stage-one amplification configuration with the tunned antenna circuit connected via a BNC connector.

The result of these optimizations was a strong signal up to about 3 or 4 feet with only about 20-mVpp of noise. Note that the value of the capacitor in the tuned antenna circuit needed to be adjusted. We assume that this was necessary to account for impedance matching necessitated by the connecting the antenna to the amplifier using a coaxial cable.

Stage two and three were simple inverting amplifiers as shown below. Each was connected to the output of its predecessor. By the third stage noise once again became very limiting, but stage two provided a usable signal up to about 14 feet (as shown in Fig. 8) with only about 60-mVpp of noise. Since the amplitude of the pulse is all that we needed to calculate distance and direction, the fact that the signal might be inverted was of little importance.

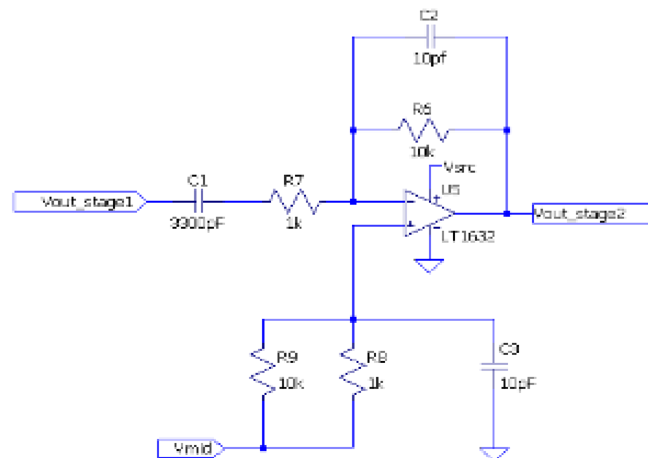


Fig. 12 An inverting amplifier providing a gain of 10-V/V. This schematic is for stage 2, though stage 3 was identical except that it was connected to the output of stage 2 rather than stage 1.

Preparing the Signal for Digital Conversion

One serious difficulty we encountered had to do with the sampling rate of our microprocessor's ADCs. In order for a signal's frequency to be accurately measured using an ADC, the sampling rate must be greater than the Nyquist frequency (twice

the frequency of interest) [26]. Our ADCs were capable of sampling the signal up to a rate of 1-MHz [27] which satisfies Nyquist's requirements, but does not provide enough resolution to confidently calculate the amplitude of the signal.

For the purposes of our receiver, however, we only need to know when a transmitting pulse is on (to verify that our direction and distance calculations are not based on the actual signal and not on noise) and what the average amplitude of the signal is. We overcame the limitations of the ADC by designing a peak detector circuit [28] for connection between the output of each amplifier and the analog inputs of the micro-controller as follows:

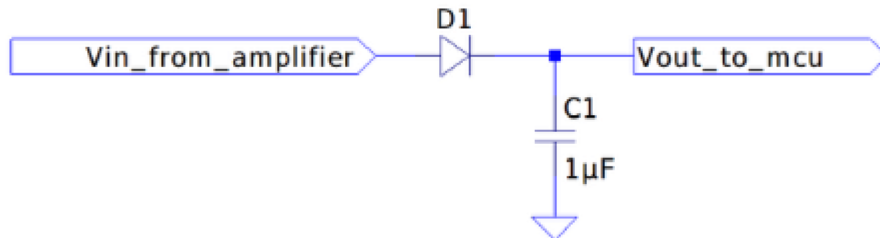


Fig. 13 Peak detection circuit place between each amplifier stage and its corresponding analog input pin on the micro-controller.

This circuit helped transform a signal like that shown in Fig. 14a to the signal shown in Fig. 14b. It also helped reduce much of the signal noise.

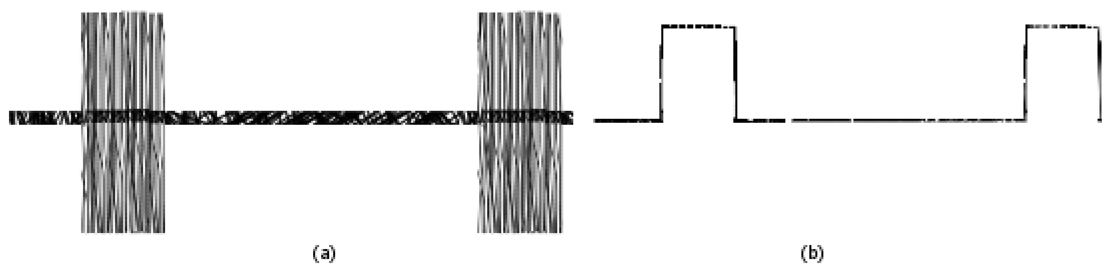


Fig. 14 A graphical representation of the signal pulse (a) without the peak detector circuit and (b) with the peak detector circuit.

The average amplitude of a 70-ms pulse was then easily sampled many times by the ADC at a rate of 1-MHz so that the signal can be processed by the microprocessor.

Software Design and Implementation

The software component of our project consists of a graphical user interface (GUI) designed using Interface Builder for Xcode IDE and a serial device driver to interface between the microcontroller and the iPhone.

Xcode supports a variety of programming languages, two of which were used for the iSlide software, Objective-C and C. Objective-C was used solely for the user interface, while C was used for the serial communication. Xcode also offers several templates for iOS applications; we chose to use the tab bar application template. The first tab is used as a splash page, and the second is the receiver enable tab which interacts with the user and directs them to the transmitting beacon.

The receiver enable tab gives the user an option to toggle receive mode, on and off and gives a graphical display of distance and direction. While receive mode is enabled the direction indicator directs the user left, right, or straight along the current path. The distance display simply displays a value received from the microcontroller below the direction indicator. Note that the iPhone is only provided with a direction and distance value that has been precomputed by the microprocessor, it has no part in the computation aside from interpreting the value received.

Interfacing with the iPhone is done by using the Rx and Tx ports on the iPhone as a standard UART serial port [29], which is described further in the iPhone interfacing section. Several files are needed for the iSlide GUI and thus are placed in Appendix A3. The analog to digital conversion (ADC) process and digital signal processing (DSP) happens before any relevant data is transmitted to the iPhone. The following two sections cover ADC and DSP.

Analog to Digital Conversion

As computer engineering majors, we wanted to have a digital aspect of our project, which is why we decided to use a microcontroller unit (MCU) rather than analog circuitry for the signal processing. After researching possible MCUs, we decided to use the LeafLabs Maple 32-bit microcontroller shown in figure 15. The LeafLabs device uses the 72 MHz STM32F 103RB processor by STMicroelectronics and operates at 3.3 V (as does the iPhone). The processor is rated for temperatures from -40° C to 85° C. The STM32 microprocessor has two ADCs with a total of sixteen channels. Each ADC has twelve bits of precision and a 14-MHz maximum ADC clock rate. We used four of the sixteen channels; one for the output of the stage 1 and stage 2

amplifiers for each antenna.

At its maximum sampling rate, 1-MHz, the ADC samples slightly above the Nyquist Frequency as previously explained. Our first attempt at sampling the output of our antenna circuit was only sampling at about 150-KHz, well under the Nyquist frequency. We were also only using positive samples, that is, only samples that were above the 1.65-V virtual ground, causing us to sample several different beacon pulses, rather than sampling the same pulse several times. Using this method, it was impossible to model the 457-kHz waveform.

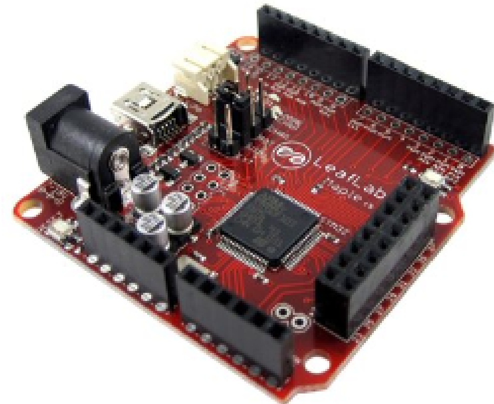


Fig.15 LeafLabs Maple 32-bit Microcontroller (r5).

After reading further into the STM32 documentation, we realized the ADC supported simultaneous sampling of both ADCs. This allowed us to simultaneously sample two channels, and the result was stored in a 32-bit integer. The upper sixteen bits contained the result of channel a, and the lower sixteen bits contained channel b. The result was stored in DMA, and we read that value when necessary. By using the DMA we were able to allow the ADC to sample freely without having to wait on computation to finish before taking the next sample. Refer to Appendix A3 for the related source code file, islide_dsp.pde.

Digital Signal Processing

After the analog signal was digitized through the ADC, we needed to process that digital value. At a maximum clock rate of 72-MHz, the microprocessor selected to compute our signal processing algorithms was sufficient. Two algorithms were used to process the digital value: a high frequency noise filter, and a peak detection algorithm.

High Frequency Noise Filter

Our original required an algorithm to filter high frequency noise because analog waveforms tend to fluctuate in voltage levels as it oscillates. More specifically, if two samples are taken consecutively, the second sample could be less than first while the waveform is rising to its positive peak, and similarly when falling to its negative peak. This algorithm places each sample into an array of size four. When the array is full the samples are averaged and we determine if that value is a peak or not by what we call a “peak detection algorithm.” Now that the array is full, each sample, *n*, replaces the $(n-5)^{th}$ element in the array and a new average is computed. This algorithm was later omitted to cut down on computation time since our analog peak detector eliminated most of the noise.

Peak Detection Algorithm

The Idea behind the peak detection algorithm was to determine whether or not the

current sample was a peak. Every sample was compared to the previous sample and stored in an array if it was larger than the previous; this was considered the current "peak." If the current sample was less than the previous, it was stored in the next element of the array, and overwritten if the following sample was a higher value. Sixteen total "peaks" for each antenna were stored into an array and averaged. Distance and direction were calculated using the computed average. This would effectively model the waveform and detect the peaks if we were sampling at a frequency well above 1-MHz. Due to the sampling rate limitations, described previously, we also had to discard this approach. Using the analog peak detector, however, we have a pulse for 70-ms at a steady voltage, allowing us to take several samples of a given peak. We choose to store sixteen samples above a 1.70-V threshold, average them, and compute distance and direction.

iPhone Interface

The only hardware we needed to gain access to the iPhone's serial pins was a PodBreakout board, which connects to the standard iPhone connector port, giving access to all 30 pins as shown in figure 16 [29]. We were then able to develop a handshake protocol as well as a robust communication protocol using serial UART communication between the iPhone and the microprocessor. The handshake and communication protocols are depicted in Fig.17. When the user presses the receive button, the serial port on the iPhone is opened and a handshake character is sent to the MCU. The MCU confirms the handshake and waits for a request from the iPhone. Meanwhile the main thread on the iPhone has been waiting for the confirmation, upon confirmation a child thread is launched and makes a request for the distance and direction packet and waits. The MCU, which has been waiting for a request sends the packet and waits for confirmation that the iPhone received the packet. The screen is now updated to display recent search information and another request is made. This cycle is repeated until the user disables receive mode, at which time a character, "!", is sent to the MCU indicating a reset request.

Fig. 16 Kinetica PodBreakout Board.

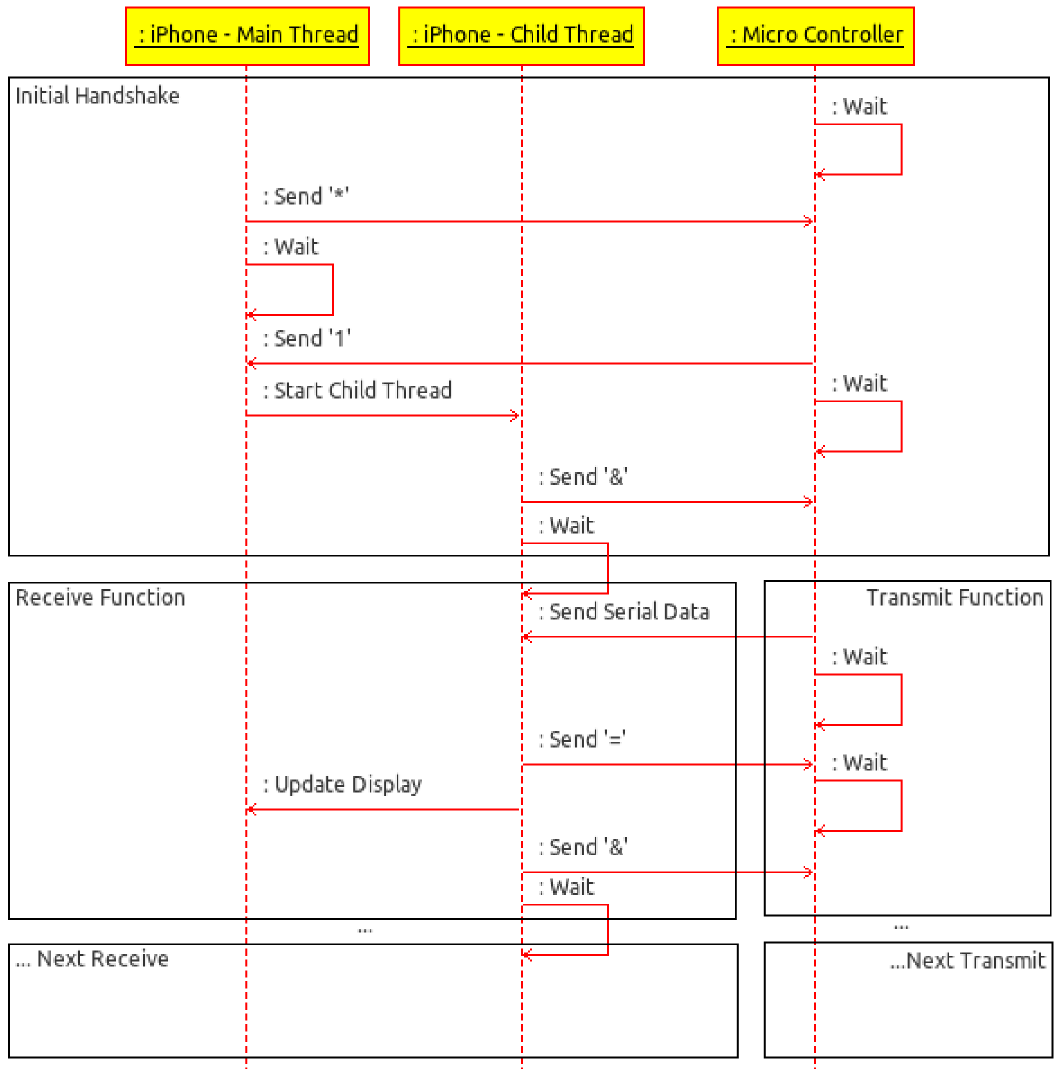


Fig. 17 Communication protocol for sending and receiving data via UART

Risk Assessment Revisited

In our project proposal, we identified the following risks and concerns:

- Our limited experience with wireless design.
- Modulation restriction of avalanche beacon standards.
- Difficulty identifying the shortest path to a transmitting beacon.
- Limited public documentation on iPhone hardware.
- Noise filtering.
- Cost.

Most of these items proved to be valid concerns. During the development process, we encountered two unexpected challenges:

- A bottle-neck design process.
- Improper hardware selection.

Many of these concerns were interrelated. We'll address their effect on our final product in the paragraphs that follow.

Bottle-Neck Design Process, Noise, and Wireless Inexperience

The most significant challenge we faced had to do with our design approach. With limited understanding of the principles of radio wave propagation and wireless communication, we planned to "figure out" the direction and distance algorithms after completing our hardware design so that we could experiment and take actual measurements to determine an effective method for identifying the distance from and direction to a transmitting beacon.

Unexpectedly, we spent much of the semester implementing the hardware only to discover that our theorized beacon-location approach was flawed. As a result, a significant portion of the software development was delayed until the final weeks of the project.

We had imagined that radio waves would propagate like perfect invisible spheres in a 3-dimensional Cartesian coordinate system, that we could capture any signal if our tuning and amplification were sufficient, and that distance and an exact, shortest-path, direction to a transmitting beacon could easily be calculated using basic geometry and trigonometry.

Our initial design consisted of about five simple BJT (Bipolar Junction Transistor) amplifier stages following the tuned antenna, and in simulation it appeared that our design would work extremely well. After building the cicuit, however, we quickly discovered the complexities and trade-offs of obtaining high gain and low noise, especially with cascading amplifiers. This motivated our op amp design and the winding of our own antennas, which helped significantly, but still left us with a usable range of only about 10 to15 feet before our signal became washed out by noise from internal and external sources. Luckily, this was sufficient for our classroom demonstration.

We were also caught off guard by the effect of the figure eight radiation pattern on the complexity of processing the signal. Additional research and guidance from a few experienced professionals lead us to the successful methodology described previously in this document. By comparing the amplitude of the signal detected by each antenna, we were able to guide the user on a circular path along the radiation pattern of the transmitting antenna toward the source beacon.

Improper Hardware Selection and Limitations of the Standard Frequency

We realized that our ADC sampling rate was insufficient to completely digitize our analog signal about 24 hours before our demonstration. Originally, we were concerned that we may not be able to modulate the signal to encode the data required for some of our add-on features. Now we faced the possibility that our micro-controller would never even get an accurate representation of the signal.

We understood that in order to recreate the signal we would need to sample at a rate of at least two times our 457-kHz frequency, or 994-kHz (the Nyquist frequency). We assumed that our 72-MHz processor would easily manage that task. It never occurred to us that the microprocessor's ADCs would operate on a slower 14-MHz clock and that each sample would require at least 14 clock cycles, though realistically we were only able to sample at a rate of more than half the Nyquist frequency.

To overcome this obstacle, we were able to find a hardware solution, to implement the averaging and peak detection algorithms as described above.

Limited Public Documentation on the iPhone

When conducting preliminary research for this project, we discovered the serial communication ports on the iPhone. This makes communicating with the iPhone somewhat trivial. The downside to using the serial port was the lack of documentation on how to use it. After a denied application for the Made For Apple (MFi) license, it was necessary to find supplemental information on how to access the serial port. Fortunately, we were able to find an open source tutorial on using the iPhone as a serial UART [29]. Using the provided information, we were able to use C libraries and functions to access the UART as in any UNIX-based software [29].

Cost

Unfortunately, this project broke the bank. Thank goodness for student loans! We had anticipated spending about $500 on the project, but as shown in the bill of materials, we actually spent over $1000, not to mention shipping, university parking fees, late night snacks, and the loss of wages caused by taking time off work. The bill of materials for the iSlide device itself, however, was only about $140.

Conclusion

Though we faced several serious challenges, and paid for it in nearly every aspect of our unbalanced lives, this was an incredible experience for us. Much additional work would be required to take this project to production, but we worked well as a team and were both stretched beyond our perceived limits at times. We learned a lot about project planning and development and about the principles of electrical and computer engineering. We were able take an idea and produce a working prototype by our target deadline. We successfully designed and implemented an avalanche beacon receiver consisting of a custom iPhone accessory and application. Our design was compatible with commercial beacons and we were able to successfully guide users, to a transmitting beacon.

References

[1] Electromagnetic compatibility and Radio spectrum Matters (ERM); Avalanche Beacons; Transmitter-receiver systems, ETSI Standard EN 300 718, 2001.

[2] “Avalanche Tranciever”; Wikipedia; Retrieved from http://en.wikipedia.org/wiki/Avalanche_transceiver#Search_techniques in July, 2011.

[3] J. Hereford *et al*, “457 kHz Electromagnetism & the Future of Avalanche Transceivers,” International Snow Science Workshop, Big Sky, MT, 2000 [Online]. Available: http://www.backcountryaccess.com/index.php?id=171

[4] “Dual Antennas,” Antennas [Online]. Retrieved from http://beaconreviews.com/transceivers/Antennas.asp in July 2011.

[5] E. Letterman, “There’s an app for that,” CCHeadliner, Nov. 2, 2011. Retrieved in December 2011 from http://ccheadliner.com/news/there-s-an-app-for-that/article_a8901100-04a1-11e1-8874-001cc4c002e0.html

[6] Open source software retrieved in August 2011 from http://freemat.sourceforge.net/.

[7] Free software retrieved in August 2011 from http://www.linear.com/designtools/software/.

[8] P. Donohoe, “Wire Antennas,” ECE 4900 Lecture Notes. Retrieved in July 2011 from http://www.ece.msstate.edu/~donohoe/ece4990notes4.pdf.

[9] Frequency Wavelength Calculator, Retrieved in august 2011 from http://www.csgnetwork.com/freqwavelengthcalc.html.

[10] iPhone 4s Technical Specification. Retrieved in December 2011 from http://www.apple.com/iphone/specs.html

[11] “United States Frequency Allocation: The Radio Spectrum,” U.S. Department of Commerce, 2003. Retrieved in December 2011 from http://www.ntia.doc.gov/files/ntia/publications/2003-allochrt.pdf.

[12] “Loop Antenna,” Wikipedia. Retrieved in August 2011 from http://en.wikipedia.org/wiki/Loop_antenna.

[13] “Ferrite Rod Antenna,” Antennas and Propogation. Retrieved in August 2011 from http://www.radio-electronics.com/info/antennas/ferrite_rod_antenna/ferrite_rod_antenna.php.

[14] “RLC Circuit,” Wikipedia. Retrieved in July 2011 from http://en.wikipedia.org/wiki/RLC_circuit.

[15] "All About Loop Antennas for VLF-LF," Radio-Electronics Magazine, June 1983, pp83. Retrieved in September from http://user.netonecom.net/~swordman/Radio/re-loop-article.htm.

[16] K. Coates, “Resistors for Noise,” eHow. Retrieved in November from http://www.ehow.com/list_7567847_resistors-noise.html.

[17] E. A. Richards el al, “Electrically Small Antenna Design for Low Frequency Systems,” 34th Annual Antenna Applications Symposium, September, 2010. Retrieved in November 2011 from http://www.q-track.com/Files/files/Electrically%20Small%20Antenna%20Design-final.pdf.

[18] “Ferrite Rods, Bars, Plates and Tubes,“ CWS Bytemark, 2002. Retrieved in December 2011 from http://www.bytemark.com/products/rod1.htm.

[19] “Radio Propagation,” Wikipedia. Retrieved in July 2011 from http://en.wikipedia.org/wiki/Radio_propagation.

[20] “Noise in Cascaded Amplifiers,” Radar Tutorial. Retrieved in November 2011 from http://www.radartutorial.eu/09.receivers/rx09.en.html.

[21] B. Carter, “A Single-Supply Op-Amp Circuit Collection,” Application Report, Texas Instruments, November 2010, pp. 11. Retrieved in November 2011 from http://www.ti.com/lit/an/sloa058/sloa058.pdf.

[22] “LT1632 / LT1633 45MHz, 45V/ms, Dual/Quad Rail-to-Rail Input and Output, Precision Op Amps,” Linear Technology, 1998. Retrieved in October 2011 from http://cds.linear.com/docs/Datasheet/16323fs.pdf.

[23] “RC Circuit,” Wikipedia. Retrieved in December 2011 from http://en.wikipedia.org/wiki/RC_circuit.

[24] J. Humes, Head of Engineering, Back Country Access, Colorado. Email correspondence on October 20th, 2011.

[25] M Eastwood, Principal, Card Access Inc., Utah. Personal interview in November 2011.

[26] K. Williston *et al*, “ADCs, DACs, and Sampliy Theory” in *Digital Signal Processing,* Burlington, MA, Newness, 2009, ch. 1, sec. 1.2, pp. 28-36.

[27] “STM32F103x8 / STM32F103xB: Medium-density performance line ARM-based 32-bit MCU with 64 or
128 KB Flash, USB, CAN, 7 timers, 2 ADCs, 9 communication interfaces,” STMicroelectronic, 2009. Retrieved in August 2011 from http://www.robotshop.com/pdf/microprocessor-stm32f103rb-datasheet-m30.pdf.

[28] “Peak Detector,” All About Circuits. Retrieved in December 2011 from http://www.allaboutcircuits.com
/vol_3/chpt_3/5.html.

[29] Meyer, C. *iPhone Serial Port Tutorial.* Retrieved March 1, 2011, from http://devdot.wikispaces.com/Iphone+Serial+Port+Tutorial

[30] Image retrieved in December 2011 from http://www.angelfire.com/electronic2/index1/680uh-Loopstick-Small.jpg.

Acknowledgements

We would like to thank the following individuals for freely offering their knowledge and experience to aide us on this project (listed in alphabetically by last name):

- Al Davis, University of Utah. Though he would prefer not to be acknowledged since “it’s his job to help us.” Thank you for mentoring us in the principles of project management and development.
- Martin Eastwood, Card Access, Inc. Thank you for helping us troubleshoot our circuit and for teaching us some useful techniques to increase gain and decrease noise.
- Jim Humes, Back Country Access. Thank you for reviewing our initial design and for offering helpful feedback.
- Nathan Mueller, Card Access, Inc. Thank you for helping us understand more about radio theory, which lead us to make significant improvements to our RF circuitry.
- Bruce Spratt, General Electric. Thank you for helping us understand some of the principles of antenna theory.

Finally, thank you to our families and employers for supporting us and allowing us the flexibility in our schedules that we needed to complete this project on time.

Appendix A1 - iOS Source Code

iPhone GUI Files

```
//
//  iSlideAppDelegate.h
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#import <UIKit/UIKit.h>

@interface iSlideAppDelegate : NSObject <UIApplicationDelegate, UITabBarControllerDelegate> {

}

@property (nonatomic, retain) IBOutlet UIWindow *window;

@property (nonatomic, retain) IBOutlet UITabBarController *tabBarController;

@end

//
//  iSlideAppDelegate.m
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//
#import "iSlideAppDelegate.h"

@implementation iSlideAppDelegate

@synthesize window=_window;

@synthesize tabBarController=_tabBarController;

- (BOOL)application:(UIApplication *)application didFinishLaunchingWithOptions:(NSDictionary *)
launchOptions
{
    // Override point for customization after application launch.
    // Add the tab bar controller's current view as a subview of the window
    self.window.rootViewController = self.tabBarController;
    [self.window makeKeyAndVisible];
    return YES;
}

- (void)applicationWillResignActive:(UIApplication *)application
{
    /*
     Sent when the application is about to move from active to inactive state. This can occur for
certain types of temporary interruptions (such as an incoming phone call or SMS message) or when
the user quits the application and it begins the transition to the background state.
     Use this method to pause ongoing tasks, disable timers, and throttle down OpenGL ES frame
rates. Games should use this method to pause the game.
     */
}

- (void)applicationDidEnterBackground:(UIApplication *)application
{
    /*
     Use this method to release shared resources, save user data, invalidate timers, and store
enough application state information to restore your application to its current state in case it
is terminated later.
```

```
     If your application supports background execution, this method is called instead of
applicationWillTerminate: when the user quits.
     */
}

- (void)applicationWillEnterForeground:(UIApplication *)application
{
    /*
     Called as part of the transition from the background to the inactive state; here you can
undo many of the changes made on entering the background.
     */
}

- (void)applicationDidBecomeActive:(UIApplication *)application
{
    /*
     Restart any tasks that were paused (or not yet started) while the application was inactive.
If the application was previously in the background, optionally refresh the user interface.
     */
}

- (void)applicationWillTerminate:(UIApplication *)application
{
    /*
     Called when the application is about to terminate.
     Save data if appropriate.
     See also applicationDidEnterBackground:.
     */
}

- (void)dealloc
{
    [_window release];
    [_tabBarController release];
    [super dealloc];
}

/*
// Optional UITabBarControllerDelegate method.
- (void)tabBarController:(UITabBarController *)tabBarController
didSelectViewController:(UIViewController *)viewController
{
}
*/
/*
// Optional UITabBarControllerDelegate method.
- (void)tabBarController:(UITabBarController *)tabBarController
didEndCustomizingViewControllers:(NSArray *)viewControllers changed:(BOOL)changed
{
}*/
@end
//
//  FirstViewController.h
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#import <UIKit/UIKit.h>


@interface FirstViewController : UIViewController {

}

@end
//
//  FirstViewController.m
```

```
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#import "FirstViewController.h"


@implementation FirstViewController

/*
// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad
{
    [super viewDidLoad];
}
*/

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}


- (void)didReceiveMemoryWarning
{
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc. that aren't in use.
}


- (void)viewDidUnload
{
    [super viewDidUnload];

    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}


- (void)dealloc
{
    [super dealloc];
}

@end

//
//  SecondViewController.h
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#import <UIKit/UIKit.h>
#import "Serial.h"
#import "Time.h"


@interface SecondViewController : UIViewController {

       UIImageView *Background;
       UIImageView *arrow;
       UILabel *distance;
```

```
        //UILabel *distance2;
        NSString *serialData;
        UITapGestureRecognizer *touchGesture;
        bool doubleTapped;
        float angle;
        UIButton *TransmitButton;
        UIButton *ReceiveButton;
        UIImage *receiveOnBtnImage;
        UIImage *receiveOffBtnImage;

        NSTimer *receiverTimer;

        int fd;         //file descriptor
        char somechar[8];
        BOOL receiverEnabled;

        //int16_t someint;
        unsigned int someint;
        //int16_t serialInt;

}

- (IBAction)transmitHandler:(id)sender;
- (IBAction)receiveHandler:(id)sender;
- (void)highlightButton:(UIButton *)button;
- (void)highlightButtonOff:(UIButton *)button;
- (void)pollSerialRx:(id)sender;

@property (nonatomic, retain) IBOutlet UIButton *TransmitButton;
@property (nonatomic, retain) IBOutlet UIButton *ReceiveButton;
@property (nonatomic, retain) IBOutlet UIImageView *Background;
@property (nonatomic, retain) IBOutlet UIImageView *arrow;
@property (nonatomic, retain) IBOutlet UILabel *distance;
//@property (nonatomic, retain) IBOutlet UILabel *distance2;
@end

//
//  SecondViewController.m
//  iSlide
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#import "SecondViewController.h"

@implementation SecondViewController
@synthesize TransmitButton;
@synthesize ReceiveButton;
@synthesize Background;
@synthesize arrow;
@synthesize distance;

#define M_PI_6 M_PI_4*(2.0/3.0)

// Implement viewDidLoad to do additional setup after loading the view, typically from a nib.
- (void)viewDidLoad
{
        touchGesture = [[UITapGestureRecognizer alloc] initWithTarget:self
action:@selector(viewWasDoubleTapped:)];
        [touchGesture setNumberOfTouchesRequired:2];
        receiveOffBtnImage = [UIImage imageNamed:@"receive_off_but.png"];
        [ReceiveButton setImage:receiveOffBtnImage forState:UIControlStateHighlighted];
        receiveOnBtnImage = [UIImage imageNamed:@"receive_on_but2.png"];
        [ReceiveButton setImage:receiveOnBtnImage forState:UIControlStateNormal];
        [[self view ] addGestureRecognizer:touchGesture];
        [touchGesture release];

        //init vars
```

```
        someint = 0;
        angle = M_PI;
        doubleTapped = false;
        receiverEnabled = 0;
        self.arrow.transform = CGAffineTransformMakeRotation(angle);
        distance.text = [NSString stringWithFormat:@"00"];
        distance.textColor = [UIColor greenColor];

        [super viewDidLoad];
}


- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation
{
    // Return YES for supported orientations
    return (interfaceOrientation == UIInterfaceOrientationPortrait);
}

- (void)didReceiveMemoryWarning
{
    // Releases the view if it doesn't have a superview.
    [super didReceiveMemoryWarning];

    // Release any cached data, images, etc. that aren't in use.
}
#pragma -
#pragma UIResponder delegate overrides
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{
        //Change something when screen is pressed
}
- (void) viewWasDoubleTapped:(UIGestureRecognizer*)sender
{
    if (sender.state != UIGestureRecognizerStateEnded)  // <---
        return;
    doubleTapped = true;

}

- (void)viewDidUnload
{
    [self setArrow:nil];
    [self setDistance:nil];
    [self setTransmitButton:nil];
        [self setReceiveButton:nil];
    [super viewDidUnload];

    // Release any retained subviews of the main view.
    // e.g. self.myOutlet = nil;
}


- (void)dealloc
{
    [arrow release];
    [distance release];
    [TransmitButton release];
        [ReceiveButton release];
    [super dealloc];
}

- (IBAction)transmitHandler:(id)sender
{

}
- (void)highlightButton:(UIButton *)button {
    [button setHighlighted:YES];
}
```

```
- (void)highlightButtonOff:(UIButton *)button {
    [button setHighlighted:NO];
}

- (IBAction)receiveHandler:(id)sender
{

	if( !receiverEnabled )
	{
		fd=OpenSerialPort(); // Open tty.iap with no hardware control, 8 bit, BLOCKING
and at 19200 baud

		[self performSelector:@selector(highlightButton:) withObject:sender
afterDelay:0.0];

		if(fd>-1)
		{
			write(fd,"*",1); // Write handshaking message over serial
			read(fd,&someint,4); // Read 4 bytes  over serial.  This will block (wait)
untill the byte has been received

			//serialData = [NSString stringWithCString:somechar
encoding:NSASCIIStringEncoding];

			distance.text = [NSString stringWithFormat:@"a%d", someint];
			if( someint == 1)
			{
				distance.text = [NSString stringWithFormat:@"Calling Thread..."];
				receiverTimer = [NSTimer scheduledTimerWithTimeInterval:0.5
target:self selector:@selector(pollSerialRx:) userInfo:nil repeats:YES];
				[receiverTimer fire];
			}
			receiverEnabled = 1;
		}
	}
	else
	{
		//tell mcu to cease data xfer
		write(fd,"!",1);
		[self performSelector:@selector(highlightButtonOff:) withObject:sender
afterDelay:0.0];
		receiverEnabled = 0;
		[receiverTimer invalidate];
		someint = 0;
		distance.text = [NSString stringWithFormat:@"00"];
		close(fd); //close serial port
	}

}

/*********************************************************************************
 pollSerialRx  reads the antenna data from the microcontroller every 1 second (as
				specified by the NSTimer thread)
*********************************************************************************/
- (void)pollSerialRx:(id)sender
{
	write(fd,"&",1);
	read(fd,&someint,4);  // Read 4 bytes  over serial.  This will block (wait) untill the
byte has been received

	//serialInt = someint;
	write(fd,"=",1);		// Confirm reception of antenna data with "=" char
	static int temp = 0;
	if ( !temp )
	{
		distance.textColor = [UIColor whiteColor];
```

```
	}
	else
	{
		distance.textColor = [UIColor greenColor];
	}
	temp = !temp;

	unsigned int direction = someint&0x3;
	unsigned int dist		= (someint>>4);

	//adjust arrow and/or distance meter accordingly
	if( (direction >= 1) || (direction<=3) )
	{
		if( direction == 1 )		// turn arrow left
			angle = 5*M_PI_6;
		else if( direction == 2 )	// set arrow straight
			angle = M_PI;
		else if( direction == 3 )	// turn arrow right
			angle = 7*M_PI_6;
		self.arrow.transform = CGAffineTransformMakeRotation(angle);
	}

	distance.text = [NSString stringWithFormat:@"%i ft.", dist];
}

@end
```

iPhone Serial Files

```
//
//  Serial.h
//
//  Created by Jacob Sanders on 6/13/11.
//  Copyright 2011 University of Utah. All rights reserved.
//

#include <stdio.h>      /* Standard input/output definitions */
#include <string.h>     /* String function definitions */
#include <unistd.h>     /* UNIX standard function definitions */
#include <fcntl.h>      /* File control definitions */
#include <errno.h>      /* Error number definitions */
#include <termios.h>    /* POSIX terminal control definitions */
#include <sys/ioctl.h>  /* General terminal interface */

int OpenSerialPort();


////////////////////////////////////////////////////////////////////////
//  Iphone Serial I/O demo.  see DevDot.wikispaces.com for more info  //
//  --------------------------------------------------------------  //
//  By Collin Meyer (TheRain)
//
//  12-09-2007 revision 1                                                    //
//  Feel free to reuse this code.                                           //
////////////////////////////////////////////////////////////////////////


#include "Serial.h"

static struct termios gOriginalTTYAttrs;

int OpenSerialPort()
{
	int                   fileDescriptor = -1;
	// int        handshake;
	struct termios  options;

	//Open the serial port read/write, with no controlling
	//terminal, and don't wait for a connection.
	//The O_NONBLOCK flag also causes subsequent I/O on
	//the device to be non-blocking.
	//See open(2) ("man 2 open") for details.

	fileDescriptor = open("/dev/tty.iap", O_RDWR | O_NOCTTY | O_NONBLOCK);

	if (fileDescriptor == -1)
	{
		printf("Error opening serial port %s - %s(%d).\n",
		           "/dev/tty.iap", strerror(errno), errno);
		goto error;
	}

	// Note that open() follows POSIX semantics: multiple
	// open() calls to the same file will succeed
	// unless the TIOCEXCL ioctl is issued. This will prevent
	// additional opens except by root-owned processes.
	// See tty(4) ("man 4 tty") and ioctl(2) ("man 2 ioctl") for details.
	int iotclRet = ioctl(fileDescriptor, TIOCEXCL);

	if (iotclRet == -1)
	{
		printf("Error setting TIOCEXCL on %s - %s(%d).\n",
		        "/dev/tty.iap", strerror(errno), errno);
		goto error;
	}
```

```
// Now that the device is open, clear the O_NONBLOCK
//  flag so subsequent I/O will block.
// See fcntl(2) ("man 2 fcntl") for details.

if (fcntl(fileDescriptor, F_SETFL, 0) == -1)
{
        printf("Error clearing O_NONBLOCK %s - %s(%d).\n",
               "/dev/tty.iap", strerror(errno), errno);
        goto error;
}

// Get the current options and save them so
//  we can restore the default settings later.
if (tcgetattr(fileDescriptor, &gOriginalTTYAttrs) == -1)
{
        printf("Error getting tty attributes %s - %s(%d).\n",
               "/dev/tty.iap", strerror(errno), errno);
        goto error;
}

// The serial port attributes such as timeouts and baud
//  rate are set by modifying the termios
//  structure and then calling tcsetattr() to cause the
//  changes to take effect. Note that the
//  changes will not become effective without the tcsetattr() call.
// See tcsetattr(4) ("man 4 tcsetattr") for details.

options = gOriginalTTYAttrs;

// Print the current input and output baud rates.
// See tcsetattr(4) ("man 4 tcsetattr") for details.

printf("Current input baud rate is %d\n", (int) cfgetispeed(&options));
printf("Current output baud rate is %d\n", (int) cfgetospeed(&options));

// Set raw input (non-canonical) mode, with reads
//  blocking until either a single character
// has been received or a one second timeout expires.
// See tcsetattr(4) ("man 4 tcsetattr")
//  and termios(4) ("man 4 termios") for details.

cfmakeraw(&options);
options.c_cc[VMIN] = 1;
options.c_cc[VTIME] = 10;


// The baud rate, word length, and handshake
//  options can be set as follows:

cfsetspeed(&options, B19200);    // Set 19200 baud
options.c_cflag |= (CS8);  // RTS flow control of input


printf("Input baud rate changed to %d\n", (int) cfgetispeed(&options));
printf("Output baud rate changed to %d\n", (int) cfgetospeed(&options));

// Cause the new options to take effect immediately.
if (tcsetattr(fileDescriptor, TCSANOW, &options) == -1)
{
        printf("Error setting tty attributes %s - %s(%d).\n",
               "/dev/tty.iap", strerror(errno), errno);
        goto error;
}

// Success
return fileDescriptor;

// Failure "/dev/tty.iap"
```

```
error:
	if (fileDescriptor != -1)
	{
		close(fileDescriptor);
	}

	return -1;
}
```

Appendix A2 - MCU Source Code

```
#include <stdlib.h>
#include "dma.h"

#define baud_rate       19200
#define SERIAL_LED_PIN  2   //serial led connected to pin 1
#define DEFAULT_LED_PIN 0  //board is on led on pin 0
#define ANTENNA_X_PIN   19    // X-Antenna connected to pin 20
#define ANTENNA_Y_PIN   20    // Y-Antenna connected to pin 19
#define RAILED_VAL     3400
#define LOW_VAL        2300
#define X_TH            2100//1769
#define Y_TH            2100//1769
#define MAX_CNT         128
#define MAX_CNT_POW     7
#define LEFT            1
#define STRAIGHT        2
#define RIGHT           3
#define OFFSET          150
#define VREF            3.3
#define DREF            4096
#define ax_co            100.8
#define bx_co            -36.35
#define cx_co            12.04
#define dx_co           -0.526
#define ay_co            10.01
#define by_co            -8.127
#define cy_co            8.509
#define dy_co           -0.6099

void init_dma_xfer(void);
void handler_adc_sample(void);
void end_of_transf_irq(void) ;
uint32 calc_adc_sequence(uint8 adc_sequence_array[6]);
void set_near(void);
void transmit( int*);

//serial vars
char incomingByte = 0;
int data_out = 0;
unsigned int buffer_size = 4;
int receiverEnabled = 0;
char antenna_data_conf = 0;

//adc vars
byte direction = 1;
int distance = 0;

//dsp vars
int avgx = 0;
int avgy = 0;

//distance vars
float dist_var = 0.0;
float x_ant = 0.0;
float y_ant = 0.0;
float v_x_ant = 0.0;
float v_y_ant = 0.0;


/*dual simultaneous sampling vars*/
  uint8 adc_length=1;     //The number of channels to be converted per ADC channel
  int16 adc1_data;        //Temporary binary data
  int16 adc2_data;        //Temporary binary data
  uint32 adc_sequence=0;  //Temporary
```

```
boolean dispReg=false;
boolean dispVolts=true;
boolean dispBin=false;
float adc1_Vdata;
float adc2_Vdata;
float voltsConvert=3.350; //Conversion to voltage. Use for fine-tuning calibration.
uint32 irq_fired = 0;     //Optional for user function.
uint32 rawDataArray[6];   //The dma temporary data array.
uint8 ADC1_Sequence2[]={  // X stage 2
      13};                /*  Set te sequence 1-6 for SQR3 (left will be first).
                              Must top up to all 6 channels with zeros  */
uint8 ADC2_Sequence2[]={  // Y Stage 2
      15};
uint8 ADC1_Sequence1[]={  // X Stage 1
      12};                /* Set the sequence 1-6 for SQR3 (left will be first).
                             Must top up to all 6 channels with zeros  */
uint8 ADC2_Sequence1[]={  // Y Stage 1
      14};
uint8 i;

int samp_cnt = 0;

byte stage1 = 0;
byte stage2 = 0;

void setup()
{
  //setup serial output pins

  pinMode(SERIAL_LED_PIN, OUTPUT);
  digitalWrite(SERIAL_LED_PIN, LOW);
  pinMode(DEFAULT_LED_PIN, OUTPUT);
  digitalWrite(DEFAULT_LED_PIN, LOW);
  //setup adc input pins
  pinMode(ANTENNA_X_PIN, INPUT_ANALOG); //ch18
  pinMode(ANTENNA_Y_PIN, INPUT_ANALOG); //ch20
  pinMode(17, INPUT_ANALOG); //ch17
  pinMode(19, INPUT_ANALOG); //ch19

  adc_init(ADC1);    //rcc_clk_enable(ADC1->clk_id), Must be the first adc command!
  adc_init(ADC2);

  adc_enable(ADC1);  //ADC_CR2_ADON_BIT = 1
  adc_enable(ADC2);

  adc_calibrate(ADC1);  //Optional
  adc_calibrate(ADC2);
  /*
   * The total conversion time is calculated as follows: Tconv = Sampling time + 12.5 cycles
   * Example, with an ADCCLK = 14 MHz and a sampling time of 1.5 cycles:
   * Tconv = 1.5 + 12.5 = 14 cycles = 1 µs
   */

  adc_set_sample_rate(ADC1, ADC_SMPR_1_5);  // Setting all channels sampling time in cycles:
  adc_set_sample_rate(ADC2, ADC_SMPR_1_5); //1,5;7,5;13,5;28,5;41,5;55,5;71,5;239,5.
   /*
   From ST Manual: In dual mode, when configuring conversion to be triggered by an external
event,
   the user must set the trigger for the master only and set a software trigger for the slave
to
   prevent spurious triggers to start unwanted slave conversion. However, external triggers
must be
   enabled on both master and slave ADCs
   */

  adc_set_exttrig(ADC1, 1);        //External trigger must be Enabled for both ADC.
  adc_set_exttrig(ADC2, 1);        //External trigger must be Enabled for both ADC.

  adc_set_extsel(ADC1,ADC_ADC12_SWSTART);   //External trigger Event, ADC1 only!
```

```
    /*
     Once the scan bit is set, ADC scans all the channels selected in the ADC_SQRx registers.
     A single conversion is performed for each channel of the group.
     After each end of conversion the next channel of the group is converted AUTOMATICALLY.
     If the DMA bit is set, the direct memory access controller is used to transfer the converted
     data of regular group channels to SRAM after each EOC.
     */

    /***SELECT BETWEEN THE FOLLOWING OPTIONS*/  //
    ADC1->regs->CR1 |= 1 << 8;  // Set scan mode
    ADC2->regs->CR1 |= 1 << 8;  // Set scan mode
    //scan_mode(ADC1,1);      //USE ONLY WITH DADC.H  (not in adc.h)
    //scan_mode(ADC2,1);      //USE ONLY WITH DADC.H  (not in adc.h)

    adc_set_reg_seqlen(ADC1, adc_length);  //The number of channels to be converted.
    adc_set_reg_seqlen(ADC2, adc_length);
    /*
     * calc_adc_sequence(ADCx_Sequence) converts the SQR3 6 channels' (each ADC1 and ADC2) list
into
     * a valid 6 X 5=30 bits sequence format. Load the sequence onto one of 3 SQR registers.
     * For more channels, repeat the same for SQR2, SQR1. (For SQR1 4 channels only!)
     */
    ADC1->regs->SQR3 |= calc_adc_sequence(ADC1_Sequence2);
    ADC2->regs->SQR3 |= calc_adc_sequence(ADC2_Sequence2);

    /***SELECT BETWEEN THE FOLLOWING OPTIONS*/  //
    ADC1->regs->CR1 |= 6 << 16;                  //Regular simultaneous mode ADC1 only !!*/
    //set_dual_mode(ADC1,DADC_MODE_6);      // Not in adc files.//USE ONLY WITH DADC.H

    ADC1->regs->CR2 |= 1 << 8; ///ADC_CR2_DMA_BIT 8=1, Use DMA for adc. ADC1 only.
  //adc_dma_enable(ADC1);  //Not in adc files       //USE ONLY WITH DADC.H

    /* Setup of the DMA for the adc data */
    init_dma_xfer();
    //set the datarate for USART
    Serial1.begin(baud_rate);
  }

  void init_dma_xfer(void)
  {

dma_init(DMA1);      // MUST initiate before ANY other setting!!!!.dma_init: rcc_clk_enable(dev-
>clk_id)

    dma_setup_transfer
    (
        DMA1,               //dma_device
        DMA_CH1,            //dma_channel
        &ADC1_BASE->DR,     //*peripheral_address,
        DMA_SIZE_32BITS,    //peripheral_size,
        rawDataArray,       //*memory_address, user defined array.
        DMA_SIZE_32BITS,    //memory_size,
        DMA_MINC_MODE       //dma mode, Auto-increment memory address
    );

    dma_set_priority(DMA1, DMA_CH1, DMA_PRIORITY_HIGH);    //Optional
    dma_set_num_transfers(DMA1,DMA_CH1,adc_length);
    dma_attach_interrupt(DMA1, DMA_CH1,end_of_transf_irq);
    dma_enable(DMA1,DMA_CH1);                //CCR1 EN bit 0
    DMA1->regs->CCR1 |= DMA_CIRC_MODE;        //Not in dma files. Set circular mode CCR1 CIRC bit
5

  } //end init_dma_xfer*/
  void end_of_transf_irq(void)     //User function
  {
    irq_fired = true;
  }
  //Loop is called repeatedly after creating setup() function
```

```
void loop()
{
  //direction = !direction;
  //digitalWrite(SERIAL_LED_PIN, LOW);
    digitalWrite(DEFAULT_LED_PIN, LOW);
    if( Serial1.available() > 0 )           //Check if data has been send from iPod
    {
      incomingByte = Serial1.read();         //Store the incoming data from iPod
      if( incomingByte =='*' )               // '*' means the receiver has been enabled
      {
       // digitalWrite(DEFAULT_LED_PIN, HIGH);  //LED means we received handshake char
        receiverEnabled = 1;
        //timer.resume();
        int confirm = 1;
        Serial1.write(confirm);              //Confirm the handshake
        while( Serial1.available() <= 0)     //wait for ipod to request data
         { }

         antenna_data_conf = Serial1.read(); //read ipod's request (& - send antenna data)
         if( antenna_data_conf == '&' )
         {

            while(1)
            {

              digitalWrite(DEFAULT_LED_PIN, LOW);
              /***SELECT BETWEEN THE FOLLOWING OPTIONS*/  //
              ADC1->regs->CR2 |= 1 << 22;                //software_start(ADC1 only)
              //software_start(ADC1,1);                  //Not in adc files USE ONLY WITH
DADC.H

              while(!(dma_get_isr_bits(DMA1,DMA_CH1)&1)) // Wait on dma transfer complete on
channel (DMA_ISR_TCIF1_BIT)
              {
                dma_clear_isr_bits(DMA1,DMA_CH1);        //Global Clear DMA1 channel1 transfer
flags */

                /***SELECT BETWEEN THE FOLLOWING OPTIONS*/    //
                ADC1->regs->SR |= 1<<1 ;                //Clear the end-of-conversion bit
*ADC_SR_EOC_BIT) = 0;
                //end_of_conversion_clear(ADC1);        //Not in adc files. //USE ONLY WITH
DADC.H
              }
              digitalWrite(SERIAL_LED_PIN, HIGH);
              adc1_data = (rawDataArray[0] & 0xFFFF);
              adc2_data = (rawDataArray[0]>>16 & 0xFFFF);
              //data_out = adc1_data<<8;

              //digitalWrite(DEFAULT_LED_PIN, HIGH);

              if( (adc1_data >= X_TH)
                &&(adc2_data >= Y_TH) )
              {

                avgx = avgx + (adc1_data>>MAX_CNT_POW);  //divide each elmt by 8 for avg
                avgy = avgy + (adc2_data>>MAX_CNT_POW);  //divide each elmt by 4 for 2*y_avg/8

                samp_cnt++;
              }

              if(samp_cnt == MAX_CNT)
              {
                digitalWrite(DEFAULT_LED_PIN, HIGH);
                if(    (avgx < (avgy + OFFSET * 4 / 5)) //*X_TH))
                    && (avgx > (avgy - OFFSET)) //*X_TH))
                    ||
                    (avgy < (avgx + OFFSET)) //*X_TH))
                    && (avgy  > (avgx - OFFSET)) )//*X_TH))   )
                {
```

```
                 direction = STRAIGHT;
                }
                else if( avgx > avgy )  // turn user right
                  direction = 3;
                else if ( avgx < avgy ) // turn user left
                  direction = 1;

                v_x_ant = (avgx)*VREF/DREF;
                v_y_ant = (avgy)*VREF/DREF;

                x_ant = ax_co*exp(bx_co*v_x_ant)+cx_co*exp(dx_co*v_x_ant);
                y_ant = ay_co*exp(by_co*v_y_ant)+cy_co*exp(dy_co*v_y_ant);

                 dist_var = sqrt(sq(x_ant)+sq(y_ant));

                if( dist_var > 8 )
                   distance = 99;
                else if( dist_var < 3)
                  distance = 0;
                else
                  distance = (int)(dist_var+0.5); //Add 0.5, or else the float to int
conversion always rounds down
                data_out = distance<<8|direction;

                // If stage 2 is railed, switch to stage 1
                // Need to switch back if it's not railed
                // Comment out until switch-back is implemented
              /*  if( ( (avgx >= RAILED_VAL) || (avgy >= RAILED_VAL) )
                  && !stage1 )
                {

                  stage1 = 1;
                  adc_disable(ADC1);
                  adc_disable(ADC2);
                  dma_disable(DMA1, DMA_CH1);
                  //switch adc to sample stage 1 channels
                  set_near();

                  // Setup of the DMA for the adc data
                  init_dma_xfer();
                }
               else if( ( (avgx <= LOW_VAL) || (avgy <= LOW_VAL) ) && !stage2)
               {
                  stage2 = 1;
                  adc_disable(ADC1);
                  adc_disable(ADC2);
                  dma_disable(DMA1, DMA_CH1);
                  //switch adc to sample stage 1 channels
                  set_far();

                  // Setup of the DMA for the adc data
                  init_dma_xfer();
               }*/

                //store direction in the bottom nibble, distance in the upper 28 bits
                data_out = ((distance<<4) | direction);
                transmit(&data_out);

                avgx = 0;
                avgy = 0;
                samp_cnt = 0;
              }
            }
          }
          else if( antenna_data_conf == '!')
          {
            serial_reset();
          }
```

```
        }
        else if( incomingByte == '!')           // '!' means turn the receiver off
        {
          serial_reset();
        }
    }
  }
  uint32 calc_adc_sequence(uint8 adc_sequence_array[1])
  {
    adc_sequence=0;

    for (int i=0;i<6;i++)     // There are 6 available sequences in each SQR3 SQR2, and 4 in
SQR1.
    {
      /*This function converts the array into one number by multiplying each 5-bits channel
numbers
       by multiplications of 2^5
       */
      adc_sequence=adc_sequence + adc_sequence_array[i]*pow(2,(i*5));
    }
    return adc_sequence;
  }   //end calc_adc_sequence

  void transmit( int *data )
  {
       Serial1.flush();

       Serial1.write(data, buffer_size);
       while( Serial1.available() <= 0)     //wait for confirmation that ipod received antenna
data
       { }
       antenna_data_conf = Serial1.read();  //read data reception status
       if( antenna_data_conf == '!')        //not implemented on ipod
       {
         serial_reset();
       }
       while( Serial1.available() <= 0)     //wait for ipod to request data
       { }
       int x = Serial1.read();
  }

  void serial_reset(void)
  {
    data_out = 0;
    receiverEnabled = 0;
    //timer.pause();
    //timer.refresh();
    //adc stuff
    stage1 = 0;

    Serial1.end();
    Serial1.begin(baud_rate);
  }
```

Appendix A3 - Matlab Code

antenna.m

Used for sizing the components of the tunned antenna.

```
f = 457e3;
B = 160*2*pi;
w0 = 2*pi*f;
L = 23e-6
% Based on H(s), s = 1/sqrt(C*L)
C = 1/(w0^2*L)
w = w0 + pi*B;
R=1/C/B
```

decoupling_cap.m

Used for sizing capacitors in RC networks.

```
% info from http://www.ecp.cc/cap-notes.html
%
% Zcap = Rin - j/wC; Zcap = 0 when Rin = -j/wC
%
Rin = 2e3
f = 457e3;
Cmin = 1 / (2 * pi * Rin)
```

signal.m

Used for displaying captured oscilloscope data and simulating the location algorithms.

```
clear all;

%% PROGRAM PARAMETERS
%
% Definitions:
%
% frequency - Operating freaquency of receiver device.
%
% steps_per_sample - Sample rate given in O-scope time steps.
%
% smooth_count - For high frequency noise reduction each sample
% is averaged with previous samples. This variable defines the
% number of previous samples that will be included in the
% calculated average.
%
% on_factor,
% avg_amp_count - For pulse detection each peak detected is
% compared with the average amplitude of the previous
% period(s). If it is greater than the average by a factor
% defined in on_factor, it is assumed that a pulse has begun.
% if it is less than the average, it is assumed that the pulse
```

```
% has come to an end. The number of previous periods averaged
% is defined by the value i avg_amp_count.
%

frequency = 457e3;
steps_per_sample = 10;
smooth_count = 10;
on_factor = 10;
avg_amp_periods = 10;

% Load sig_data generated by oscope_reader.m
load('phase360.mat');
start_file = 0;
file_count = length(phase360_count);
%file_count = 1;

for (i = 1 + start_file : start_file + file_count)
clc;

printf('Analyzing file: %d/%d', i-start_file, file_count);
t_step = phase360_step(i);
count = phase360_count(i);
sig(:,1:3) = phase360_sig(:, (i*3-2):(i*3));
period = round (1 / (frequency * t_step * steps_per_sample));
avg_amp = zeros (1, 3);
avg_amp_count = avg_amp_periods * period;
avg_amp_bucket = zeros (avg_amp_count, 3);
pulse_on = zeros(on_factor, 3);
on_time = zeros(on_factor, 3);
sig_max = zeros(1,3);
sig_max_index = zeros(1,3);
sig_max_count = zeros(1,3);
peak_count = zeros(1,3);

%% POLLING LOOP:
%
% This loop represents the infinite polling loop that will be
% implemented by the microprocessor
%
for (j = 1 : (count / steps_per_sample))
sig_sample = sig( 1 + (j - 1) * steps_per_sample, 1:3);

%% AVERAGE AMPLITUDE:
%
% Calculate the average amplitude (using the absolute value
% of the signal) of the passed "avg_amp_count" number of
% elements in the signal array.
%
if (j > 2 * avg_amp_count)
for (k = 1:3)
for (l = 2:on_factor)
if (pulse_on(l, k) == false)
if (sig_sample(k) > (avg_amp(k) * l))
pulse_on(l,k) = true;
on_time(l,k) = j;
end
end
end
end
end
index = mod (j, avg_amp_count) + 1;
avg_amp(1:3) = avg_amp(1:3) - avg_amp_bucket(index, 1:3);
avg_amp_bucket(index, 1:3) = abs (sig_sample(1:3) / avg_amp_count);
avg_amp(1:3) = avg_amp(1:3) + avg_amp_bucket(index, 1:3);

% Following used only for testing and modeling
% {
result_amp(j,1:3) = avg_amp;
% }
```

```
end
x(i) = result_amp(length(result_amp)*3/4:length(result_amp),2);
y(i) = result_amp(length(result_amp)*3/4:length(result_amp),1);
x_to_y = result_amp(:,1) ./ result_amp(:,2);
avg(i) = mean(x_to_y(length(x_to_y)*3/4:length(x_to_y)));
% maxes(i,1:3) = max (result_amp(:,1:3))
%theta(:,i) = 360 * (on_time(:,2) - on_time(:,1) + (0.25 * period)) / period;
end

%theta
plot(atan(avg));
```

oscope_parser.m

Used to parse files generated by an oscilloscope into a format easily readable by freeMat.

```
clear all;

%% O-SCOPE FILE PARSER
file_count = 85;
file_start = 0;
dest_file = 'phase360.mat';
filename_0 = 'angle_data/tek';
filename_1 = 'ALL.csv';
step_range = [6, 1, 6, 1];
count_range = [7, 1, 7, 1];
scale_range = [12, 1, 12, 3];
sig_range = [15, 1, inf, 3];
for (i = 1 : file_count)
clc;
printf('Read file(s): %d/%d', i, file_count);
file = [filename_0, num2str(file_start + (i - 1), '%04d'), filename_1];
% Get O-scope horizontal values
step = real (csvread (file, step_range(1), step_range(2), step_range));
count = real (csvread (file, count_range(1), count_range(2), count_range));
% Get O-scope vertical values
scale = real (csvread (file, scale_range(1), scale_range(2), scale_range));
sig = real (csvread (file, sig_range(1), sig_range(2), sig_range));
% Apply O-scope scalling factor to data
for (j = 1 : count)
sig (j, 1:3) = sig (j, 1:3) ./ scale;
end
phase360_sig (:, (3*i-2):(3*i)) = sig (:,1:3);
phase360_step(i) = step;
phase360_count(i) = count;
end
save (dest_file, 'phase360_step', 'phase360_count', 'phase360_sig');
clc;
printf('%d files read to %s.\n', file_count, dest_file);
```

Appendix B - Bill of Materials

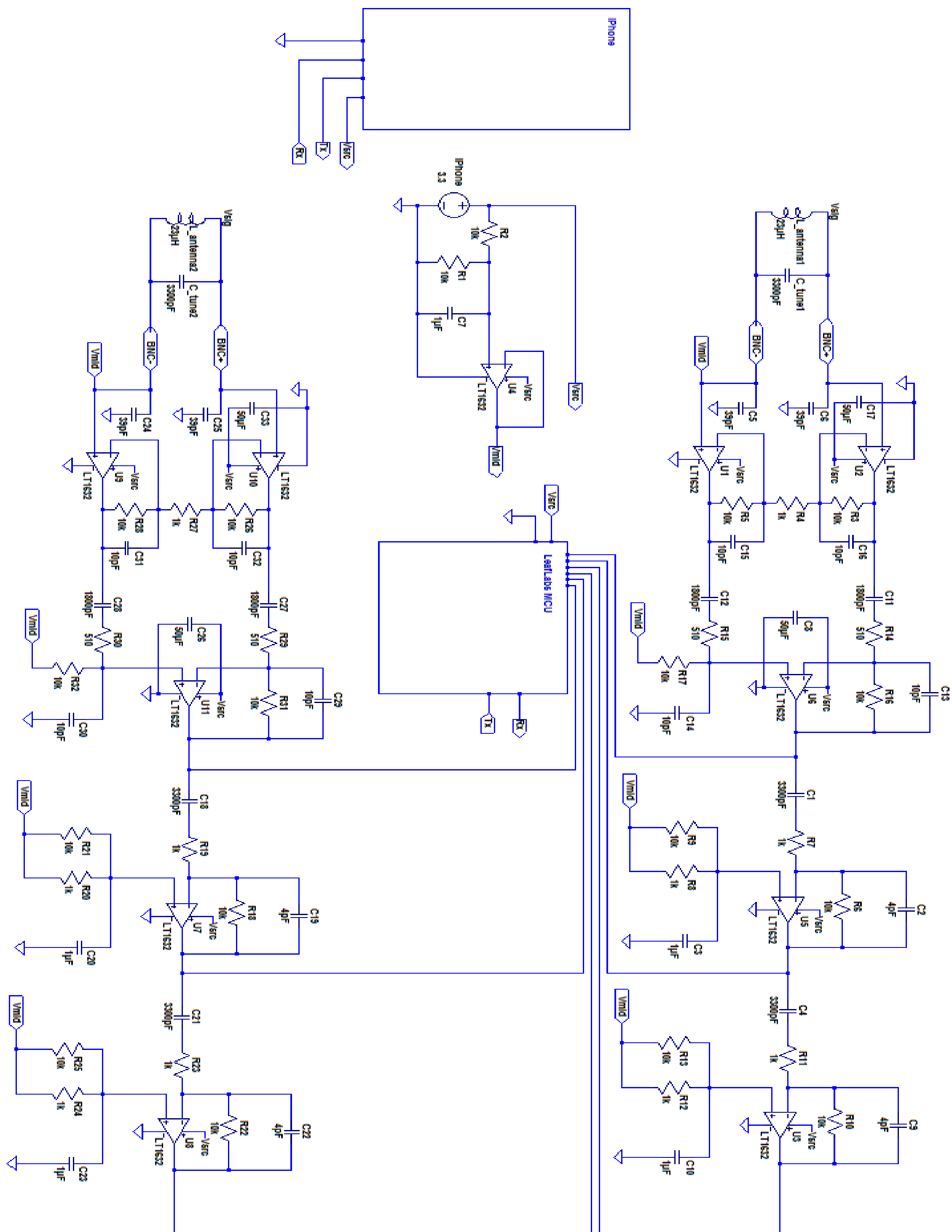
Table 1 - iSlide Bill of Materials

Part	Source	Unit Cost	Qty.	Total Cost
Pod Breakout	http://shop.kineteka.com/products/92-podbreakout-ipod-iphone-ipad-breakout-board.aspx	$14.99	1	$14.99
LeafLabs Maple 32-bit MCU	RobotShop.com	$49.99	1	$49.99
AM Rod Antennas	http://www.angelfire.com/electronic2/index1/loopstick.html	$2.47	2	$4.96
Prototype Boards	Raelco	$3	3	$9
Op Amps	Digikey.com	$6.50	7	$45.50
Discrete Components	Raelco, Digikey, ECE Stockroom, Radioshack	--	--	~$15

Table 2 - Development Bill of Materials

Part	Source	Unit Cost	Qty.	Total Cost
Additional AM Rod Antennas	http://www.angelfire.com/electronic2/index1/loopstick.html	$2.47	6	$14.82
Additional Variable Capacitors, Op Amps, and Other Basic Components, Prototype Boards	ECE Stockroom, Digikey, RadioShack.	--	--	~$200
Apple iPhone	http://www.verizonwireless.com/b2c/splash/iphone.jsp	$649.99	1	$649.99
Avalanche Beacon	http://www.ebay.com	$120.00	1	$120.00

Appendix C1 - Complete Circuit Schematic



Appendix C2 - Photograph of Final Product