# Acoustic Measurements with a Quadcopter

Embedded System Implementations for
Recording Audio from Above

Jonathan Klapel

Embedded Computing Systems
Submission date: June 2014
Supervisor: Peter Svensson, IET

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

# Problem Description

It is often desirable to make acoustic measurements in a larger number of positions or in positions that are somewhat inaccessible. In these cases, a solution using some form of quadcopter can be attractive.

Three challenges can be addressed for this: (1) Can noise from quadcopters be controlled, with low noise quadcopter models, directional microphones, active noise suppression and/or other methods? (2) Can quadcopters be quiet enough to perform acoustic measurements? There are different requirements for different types of acoustic measurements. (3) How should a lightweight measuring device be designed, with a microphone, signal processing, storage and/or transmission device?

The project should study one or several of these challenges by discussing solutions as well evaluate some of them.

# Abstract

At present, making audio recording requires an investment of time and energy. Microphones must be setup, connected to amplifiers and filters, and then passed into a computer's sound card. If the microphone is not positioned properly, someone must manually move the microphone stand and re-test the audio. If multiple measurements are needed from around a room, a significant portion of time will be spent just moving the microphone and cables. An automated method of moving the microphone around the room is often desired and this thesis lays out the option of using a quadcopter to facilitate this work.

Explored here are two different embedded systems that are capable of making measurements from a hovering quadcopter. The first is a rapidly developed design using an Apple iPod Touch. While the recording quality and the ease-of-use of the iPod are very high, the system does not offer any low-level OS control, resulting in unpredictable delays when synchronizing to the loudspeaker setup. However, this system does allow for making significant measurements of quadcopter noise and stability, which prove valuable for future designs. The second design seeks to improve upon the synchronization issue. Built using two Atmel SAM4L Xplained Pro development boards, the system achieves a sample accurate synchronization between the quadcopter and loudspeaker systems.

The final conclusions combine the results from both systems. Recommendations for future iterations of the SAM4L design are discussed. Suggestions for the SAM4L design include improving audio recording quality, reducing the amount of noise the quadcopter generates, and multiple methods for cancelling noise that is recorded by the microphone.

## Preface

This thesis research is submitted as part of the requirements for the European Master in Embedded Computing Systems (EMECS), a joint program between the Norwegian University of Science and Technology, the University of Kaiserslautern, and the University of Southampton. Work on this thesis has been performed in the Department of Electronics and Telecommunications at NTNU under the supervision of Prof. Peter Svensson.

The original inspiration for this project came from Prof. Svensson who had the idea of mounting a microphone to a quadcopter, with the intent to make the process of recording audio easier. During the fall 2013 semester, I was working with Prof. Svensson on another project when he presented the idea to me. As a student who is studying embedded systems, the project idea was immediately fascinating and plans to develop this into a full master's thesis evolved.

## Acknowledgements

I would like to thank my adviser, Prof. Svensson, for presenting this thesis idea to me, providing much insight and guidance along the way, and giving me numerous chances to expand and apply the knowledge I have learned. Many thanks go to Tim Cato Netland for his assistance in building and testing the acoustical portion of the design. My gratitude also goes to Prof. Lars Imsland and Kenneth Eide Haugen in the Department of Engineering Cybernetics for the use of their quadcopter and the research they have performed. Appreciation also goes to Frode Haukland of SINTEF ICT and Lars Morset of Morset Sound Development for their insights and Gary Klapel for his help in drawing some of the figures used in this report.

In addition, I would also like to express my deepest thanks to the advisers and staff of the EMECS program for accepting me into the program and providing an unbelievable experience learning about embedded systems. I also am incredibly thankful for my friends and loved-ones who have supported me along the way. Finally, I would like to express my love and appreciation for my family members who have always encouraged me to look for the possibilities the future may hold.

*- Jon Klapel*

# Contents

# List of Figures

# List of Tables

# List of Abbreviations

**ABDACB** Atmel’s Audio Bitstream Digital-to-Analog Converter

**ADC** Analog-to-Digital Converter

**AJAX** Asynchronous JavaScript and XML

**API** Application Programming Interface

**ASF** Atmel Software Framework

**ASTRAEA** Autonomous Systems Technology Related Airborne Evaluation & Assessment

**CIC** Cascaded Integrator-Comb

**COM** Serial Communication

**CPU** Central Processing Unit

**DAC** Digital-to-Analog Converter

**DMA** Direct Memory Access

**EMECS** Erasmus Mundus Embedded Computing Systems

**FFT** Fast Fourier Transform

**GPIO** General-Purpose Input/Output

**GPS** Global Positioning System

**GUI** Graphical User Interface

**HDMI** High-Definition Multimedia Interface

**HTML(5)** HyperText Markup Language (fifth revision)

**HTTP** HyperText Transfer Protocol

**Hz** Hertz

**$I^2C$** Inter-Integrated Circuit

**$I^2S$** Inter-Integrated Circuit Sound

**IDE** Integrated Development Environment

**IFFT** Inverse Fast Fourier Transform

**IR** Impulse Response

**JPEG** Joint Photographic Experts Group

**kHz** Kilohertz

**ksps** Kilosamples per second

**LED** Light Emitting Diode

**(m)V** (milli)Volt

| | |
|---|---|
| **NTP** | Network Time Protocol |
| **OS** | Operating System |
| **OSI** | Open Systems Interface |
| **PC** | Personal Computer |
| **PCB** | Printed Circuit Board |
| **PDCA** | Peripheral DMA Controller |
| **PWM** | Pulse Width Modulation |
| **RAM** | Random-Access Memory |
| **RBS** | Reference Broadcast Synchronization |
| **RF** | Radio Frequency |
| **ROM** | Read-Only Memory |
| **RPM** | Rotations Per Minute |
| **RTOS** | Real-Time Operating System |
| **SD** | Secure Digital |
| **SoC** | System on Chip |
| **SPI** | Serial Peripheral Interface |
| **STDIO** | Standard Input/Output |
| **TCP/IP** | Transmission Control Protocol/Internet Protocol |
| **TPSN** | Timing-sync Protocol for Sensor Networks |
| **TWI** | Two Wire Interface |
| **UART** | Universal Asynchronous Receiver/Transmitter |
| **UAV** | Unmanned Aerial Vehicle |
| **UDP** | User Datagram Protocol |
| **URL** | Uniform Resource Locator |
| **USB** | Universal Serial Bus |
| **WAV** | Waveform audio file format |
| **Wi-Fi** | Wireless local area network |
| **XML** | Extensible Markup Language |

# 1 Introduction

The reproduction of audio has long been pursued and the quality of these reproductions has been increasing over time. Still, the same basic principles apply now as they did years ago – audio goes in through a microphone, is processed by any number of analog and digital hardware, and then is sent back out to the loudspeakers. However, the time that it takes to setup and move this equipment can be quite large and any hardware that is able to reduce this time is a welcome investment. To better understand the concept behind the use of a quadcopter in the field of audio, it is necessary to examine how audio measurements are performed and how quadcopters are already making in-roads as tools in many industries.

## 1.1 Current Process for Performing Audio Measurements

In order to properly simulate and improve the acoustics of any space, engineers often rely on measuring the room's impulse response. An impulse response is capable of completely describing the audio characteristics of any space and location [**1**]. This includes measuring the delay time of the direct sound waves proceeding from the audio source to the listener position as well as any indirect waves that may reverberate off the walls and result in the listener hearing an echo. The impulse response also will include any sort of phase shifts and frequency degradation that may result as the wave works its way to the listener.

Generating an impulse response is not difficult. A simple layout is given in Figure 1-1. It normally consists of setting up a loudspeaker at the position of the audio source and then placing a microphone

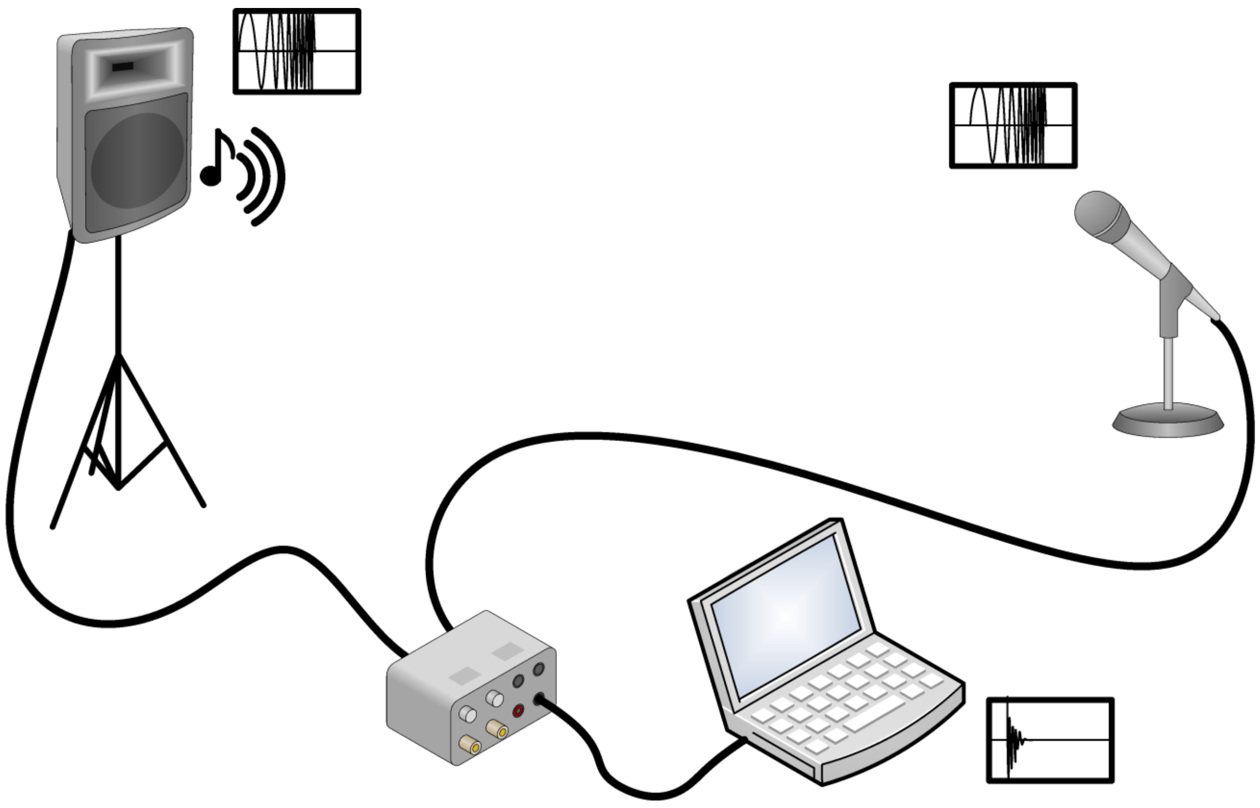


**Figure 1-1: Impulse Response Measurement Setup**

at the position of the listener. The loudspeaker will then play out a sound and the microphone will simultaneously record the audio that arrives at the listener's position, as explained in the "Recording Impulse Responses" section of [**2**]. The audio that the loudspeaker outputs is typically a swept sine signal that starts at a low frequency and exponentially grows to a high frequency signal over a period of several seconds.

By comparing the audio signal that was output by the loudspeakers to the audio signal that was actually recorded by the microphone, the impulse response of the space can be calculated [**3**]. Since the impulse response contains all of the acoustic data about what a listener will hear for a large range of frequencies, it can be used for estimating what any sort of audio played from the loudspeaker would sound like to the listener, simply by convolving the input signal with the impulse response [**1**]. For this reason, impulse responses are a major part of computer generated simulations of performance halls and auditoriums. Impulse responses can be very useful in identifying problems with the acoustics of such halls and providing solutions to improve the audio performance of the room.

Unfortunately, one impulse response is only valid for one specific set of audio source and listener positions. If one wants to simulate the acoustics in a separate part of the auditorium, another impulse response is needed. For a large auditorium, this can result in a need for many impulses response measurements. While each recording only takes a few seconds, it takes significantly longer to move the microphone and/or loudspeaker for each recording, as it means physically picking up the stand, dragging the audio cables to the next position, and then returning to the computer to start the recording once again. In an auditorium that needs hundreds or possibly thousands of impulse response measurements, this can take hours or even days to complete. For this reason, a method of improving the speed with which these measurements can be performed would be a worthwhile development.

## 1.2 Quadcopter as a Tool

Over the past decade, the use of remote controlled vehicles has grown dramatically. Though these devices go by many names, such as Drones, Quadcopters, and Unmanned Aerial Vehicles (UAVs), their features and uses are similar. The overall use of UAVs has been for tasks that are "dull, dirty, and dangerous" [**4**]. UAVs go beyond the normal remote control plane or helicopter and provide a more intelligent flight experience. Differences between quadcopters, drones, and UAVs are normally small and oft debated. The terms drones and UAV are normally used interchangeably, though drone typically has a more negative connotation, while UAV is occasionally considered to be a more bureaucratic title [**5**]. Nonetheless, both refer to a flying device that does not have an on-board crew and is normally remotely programmed. Quadcopters appear similar to a helicopter at first glance; however, they operate using four propellers located in a square pattern. While quadcopters may be big enough to contain a crew for piloting the craft, quadcopters today are often thought of as being small UAVs.

Quadcopters were conceived in the early 20th century and several companies have developed quadcopters capable of carrying humans. However, in the past few decades, engineers recognized the agility of quadcopters for use in UAVs. UAVs have been in use for over a century, including in most of the recent large scale wars. Early designs included balloons, planes, and jets that could all be piloted without someone aboard. In recent decades, the design of UAVs has become much more advanced, just as the tasks that UAVs perform have become increasingly varied.

Today, quadcopters perform myriad tasks, from scientific research to commercial video production. During the 2014 Olympics, quadcopters were used to stream live video of the games [**6**]. Amazon has announced plans to make product deliveries using quadcopters in the U.S. as early as 2015 [**7**] and DHL began testing drone delivery in December 2013 in Germany [**8**]. Meanwhile, the United Arab Emirates is already moving ahead with plans to begin secure delivery of official documents using UAVs to its citizens in the first part of 2015 [**9**].

The UAV momentum is likely just beginning as well. The European Union believes that in 10 years, civil UAVs could make up 10% of the aviation market, equating to approximately 15 billion euro annually [**10**]. ASTRAEA estimates that UAVs could be worth $62 billion globally by 2020, and that the industry serviced by UAVs could be worth $400 billion annually [**11**]. According to the European Union in [**10**], manufacturing of UAVs has the potential to create 150 thousand jobs in Europe by 2050. UAVs are rising in popularity because of their ability to be a versatile tool in so many industries, and the evidence is overwhelming that their growth will continue.

## 1.3 Audio Measurements Using Quadcopters

Given the significant rise in the capabilities of quadcopters, the case is made to investigate the possibility of using them for audio measurements. Their ability to quickly move about with little to no intervention from a human could remove a large amount of the tedium that exists when making the hundreds of audio recordings required for impulse response calculations.

As stated previously, the largest quantity of time is spent moving microphones before each recording. If a microphone could be attached to the quadcopter, the quadcopter could take care of the work required to move the microphone from one to position to the next. Recent research in [**12**] has shown that not only can quadcopters be very agile, but they can even be synchronized and choreographed to act as if they are dancing. A quadcopter could perform measurements in all of the positions that a microphone stand could and also easily position itself in parts of the room that would be difficult or impossible to reach with a microphone stand. Quadcopters could also be used during live performances to record from various positions, moving around as the operator sees fit.

However, quadcopters do not provide an instant solution to improving the ability to make audio measurements. Using a quadcopter introduces a number of hurdles that must be solved before they see widespread use in audio measurements. These include compensating for the noise generated by the quadcopter itself, reducing the potential for noise from small movements of the quadcopter during recordings, and finding a way to synchronize the operation of the audio playback and the recording that happen on the ground and on the quadcopter.

## 1.4 Assignment Breakdown and Contribution

According to the project description, there are three questions that may be addressed by the project:

> ***Can noise from quadcopters be controlled, with low noise quadcopter models, directional microphones, active noise suppression and/or other methods?***

Because of the four propellers and multiple moving parts, quadcopters are inherently loud devices. This question would investigate how noise from the quadcopter could be controlled and reduced in the final recordings.

***Can quadcopters be quiet enough to perform acoustic measurements? There are different requirements for different types of acoustic measurements.***
Depending on the type of acoustic measurement being performed, a quadcopter may need to be quiet enough that the noise from the copter does not interfere with the recording or the performance that may be happening. Answering this question would involve coming to a conclusion on which acoustic measurements, if any, could be reliably performed by a quadcopter-based microphone.

***How should a lightweight measuring device be designed, with a microphone, signal processing, storage and/or transmission device?***
In order to actually perform measurements, a complete system must be designed that is capable of performing the audio measurements satisfactorily. This would include designing a small recording device that the quadcopter is capable of carrying and which can be remotely controlled from a ground station that gives instructions on when and where recordings should be performed.

Work on this project, at a minimum, needs to discuss possible solutions to one of the challenges above. Ideally, some of these solutions would be implemented and tested for feasibility

While the project description states that only one of the challenges needs to be addressed, the work demonstrated in this report touched on solutions to all three challenges of varying degrees. Possible solutions to these questions are discussed. Several of these solutions were also implemented in prototypes and physical designs so that the efficacy of the designs could be evaluated. Finally, further revisions and solutions are offered based off of the evaluations of the implemented designs.

## 1.5 Project Implementation

This project included two different design implementations. The first design utilized an Apple iPod Touch, which allowed for multiple quick implementations and measurement recordings. By utilizing the iPod Touch, a significant amount of information regarding quadcopter noise was generated. This information was useful for then designing a more thorough embedded system that could compensate for multiple sources of noise identified by the iPod Touch. The final design was based on the Atmel SAM4L series microprocessor, along with extra modules and components that were capable of generating a complete system that could be attached to a quadcopter for performing audio measurements.

The report is organized in the following manner:

***Section 1: Introduction*** provides a brief introduction to audio recording and the use of quadcopters in industry. It then introduces the idea of using quadcopters as a tool for performing audio measurements and outlines the ideas researched in this report.

***Section 2: Background*** goes deeper into explaining the design of quadcopters and embedded systems, primarily in regards to audio. The section also discusses specific requirements that acoustical recordings present.

***Section 3: Prototype Implementation on Apple iOS*** describes the first prototype implementation on the Apple iPod Touch, including both the recording application on the iPod Touch and the audio generation program on the PC.

***Section 4: Final Design Using Atmel SAM4L Xplained Pro*** takes the information collected from the previous prototype and implements a final embedded system for performing audio recordings.

***Section 5: Design Evaluation*** presents the measurements taken from the prototype and final design implementations, discusses what these results mean in terms of audio recording, and finally makes recommendations for further improvements on the designs.

***Section 6: Final Design Discussions and Conclusions*** explains overall thoughts on the final design and presents possible future work regarding audio recordings aboard quadcopters.

# 2 Background

Before moving into the full design and implementation of a quadcopter recording system, there is a large quantity of background information that must first be obtained in order to design such a system. This section starts by lightly explaining how quadcopters function and their advantages over other options. Thereafter, the necessary aspects for an embedded system are thoroughly discussed, including processor choice, hardware modules, and communication protocols. The final parts of this section discuss the synchronization of multiple embedded systems and how audio measurements are made. At the very end, a list of challenges and goals to quadcopter recordings is given.

## 2.1 Quadcopter Architecture

As the name implies, a quadcopter consists of four propellers situated in the same horizontal plane located around the body of an aircraft. These four propellers provide lift for the craft to become airborne, stability and directional control to keep the quadcopter pointed in the proper direction, and lateral movement to allow the quadcopter to propel itself forward in any direction.

Control of quadcopters is often performed using either a remote control that an operator can use or automated hardware that will direct the quadcopter to specific locations, normally by way of GPS. More sophisticated quadcopters have the ability to use both modes of operation.

There are approximately 500 manufacturers of drones worldwide with 1,700 different types of drones on the market, according to [**10**]. With this many options available and many more coming out in the next few years, the use of drones to perform a wider variety of tasks will continue to grow.

### 2.1.1 Advantages to Using a Quadcopter

The design of the quadcopter provides for many benefits over other types of aircrafts. These benefits include:

- **Design Simplicity** – With four primary rotors providing control and stability instead of just one as conventional helicopters have, control of the quadcopter is maintained by adjusting the speed of the rotors. As there are no additional control options on a basic quadcopter, there is a reduction in the mechanical parts [**13**].
- **Increased Reliability** – As there are fewer mechanical parts, there is higher reliability and robustness in quadcopters as compared to other aircrafts [**13**].
- **Reduced Cost** – With fewer parts and reduced complexity, the overall cost of quadcopters also drops [**13**,**14**].
- **Reduced Damage Potential** – The size of the blades on a quadcopter are smaller than a normal helicopter, which means there is a reduced amount of kinetic energy in each and impacting another object will result in a reduced energy transfer. In addition, quadcopters often have a protective shield around the propellers which allows the quadcopter to harmlessly bump into other objects [**14**].
- **Indoor and Outdoor Operation** – Since there is less potential for damage, quadcopters can also be flown indoors safely [**14**].

- **Simplified Learning Curve** – With the additional safety features, such as the protective shields around the propellers, quadcopters are more forgiving than other forms of UAVs. Users have an easier time learning how to control and use the quadcopter without injuring themselves, the quadcopter, or the surroundings [**14**].
- **Increased Automation** – Because of the reduced cost and the increased simplicity of the design, quadcopters have become popular research platforms and many now come with the ability to perform pre-programmed autonomous flight [**15**,**16**,**17**].

### 2.1.2 Drawbacks of Quadcopters

While quadcopters have many benefits and look like an ideal tool for improving audio measurements, there are a few downsides to using a quadcopter that represent major hurdles for design implementations. These include:

- **Mechanical Noise** – Having four high speed motors and propellers makes the quadcopter very loud and this noise can easily disturb audio recordings taking place [**18**].
- **Movement and Stability** – A quadcopter's stability is based on adjusting the speed at which the propellers rotate and a microcontroller must calculate these speeds based on external sensors on board. The delay between reading the sensors, performing the calculations, and then altering the propeller speeds causes the quadcopter to inherently be slightly unstable and always moving [**13**]. These quadcopter movements have been noted to affect audio recordings, especially at high frequencies [**18**].
- **Payload Size** – The size of the payload on a quadcopter is very restricted. In order to carry a heavier load, a larger quadcopter must be used. A larger quadcopter often means more noise is generated. To reduce noise, it is often better to use as small a quadcopter as possible and, therefore, as small a payload as possible [**18**].
- **Battery Duration** – Batteries often provide five to fifteen minutes of flight time (or slightly more in special cases), after which the quadcopter must return to a base station to charge or to the user in order to have the battery swapped out. Flight time is often one of the biggest concerns of UAV consumers [**19**].

### 2.1.3 Quadcopter Selection

While there are a lot of quadcopters available on the market, for the purposes of this project, the Parrot AR.Drone 2.0 [**20**] was already pre-selected. This quadcopter is controlled by a mobile app on an Android or iOS phone or tablet. It also features two on-board cameras that are streamed live to the app. Communication is via an ad-hoc 802.11n Wi-Fi network setup by the AR.Drone. The mobile device simply connects to the Wi-Fi network and the app takes care of the low-level communication, leaving the consumer to use the simple on-screen GUI for the flight control. Additionally, the AR.Drone also has a USB port included on the quadcopter itself, though it is primarily intended for Parrot's Flight Recorder pack.

The AR.Drone has been quite popular in the research market with over half a million sold since the quadcopter became available in 2010 [**21**]. Much of the source code for the quadcopter's microcontroller has been made available online [**22**]. In addition, the API for remotely communicating with the

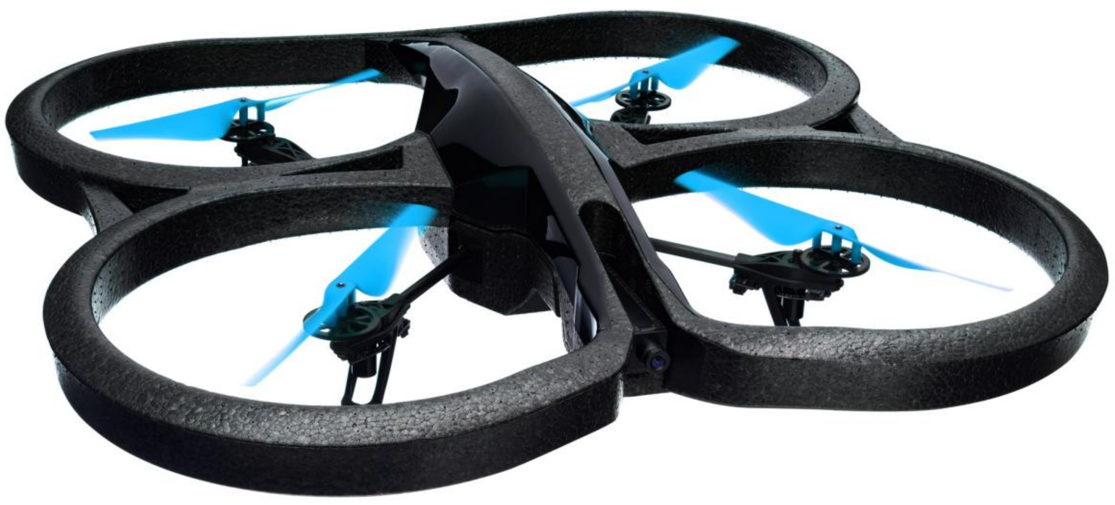


**Figure 2-1: AR.Drone 2.0 Quadcopter**
Photo courtesy of Parrot SA [**20**]

quadcopter is also open source and includes the source code for the iPhone and Android apps [**23**]. Releasing the API has made it very easy for researchers to build their own control programs and algorithms. Not only is the AR.Drone very reasonably priced, the API includes a thorough SDK library that both sends fine-grained commands to the quadcopter and receives detailed in-flight statistics back from the quadcopter.

### 2.1.4 Selection Imposed Restrictions

By using the AR.Drone, there are several additional restrictions that are placed upon the final design implementation. These are described in Table 2-1.

| Restriction | Quadcopter Limit | Design Goal | Description |
|---|---|---|---|
| **Payload** | 130 grams [**18**] | 100 gram device | Heavier payloads make the UAV unstable and reduce the battery life |
| **Flight Time / Battery Life** | 15 minutes [**20**] | (Very) low power microcontroller and devices | Adding a microcontroller with microphone means more battery drain and less flying time. |
| **Noise** | Very loud [**18**] | Low noise in recordings | One or more methods of reducing the noise from the UAV will be needed |
| **Stability** | Limited stability [**18**] | Very steady microphone positioning | Movement of the microphone can lead to reduced accuracy in measurements, especially likely at high frequencies |

**Table 2-1: AR.Drone Specific Design Requirements**

## 2.2 Development of an Embedded System for Signal Generation and Acquisition

Designing an embedded system opens up many different options for which components should be used and how they should be interfaced. However, when designing an embedded system that is intended for audio generation and acquisition, the options become more limited and more specific.

Each embedded system will likely need to generate a high quality audio signal or record the audio signal in a high quality. This means that the systems will need to include an ADC or DAC, respectively. Recorded signals will also need to be saved to some external media and playback audio will either be read from external media or algorithmically generated, all in real-time, meaning the system also must operate in a deterministic way. Finally, since the audio recording will be on-board a quadcopter, there must be some form of wireless communication between the remote recording system and the ground system.

| | iPod Touch [24] | Raspberry Pi [25] | nRF51822-EK [26] | SAM4L Xplained Pro [27] |
|---|---|---|---|---|
| **Processor** | Apple A4 (ARM Cortex A8) | Broadcom BCM2835 (ARM11) | nRF51822 SoC (ARM Cortex M0) | SAM4LC4 (ARM Cortex M4) |
| **Memory** | 512 MB | 512 MB | 256kb Flash ROM & 16kb RAM | 256kb Flash ROM & 32kb RAM |
| **Operating System** | Full OS pre-installed, non-deterministic timing | Linux OS recommended, mostly deterministic timing | No default OS, fully deterministic timing | No default OS, fully deterministic timing |
| **ADC / Audio Recording** | Integrated microphone | No ADC included | 10 bit ADC included | 12 bit ADC included |
| **DAC / Audio Playback** | Integrated speaker and headphone jack | Integrated headphone jack | No DAC included | Audio bitstream DAC |
| **Audio Storage** | Integrated hard-drive | External SD card required | No default storage medium | USB host |
| **Wireless Connectivity** | Wi-Fi and Bluetooth included | External Wi-Fi or Bluetooth module required | Low power Bluetooth included | Atmel Zigbit modules required |
| **Power Managment** | 10 hour battery included and low power processor | USB power and no low power mode | Button battery and very low power processor | USB power or external battery and very low power processor |
| **Software Development** | Proprietary Apple software and hardware required | Very easy implementation on Linux OS | Build software from scratch and pre-built libraries | Build software from scratch and pre-built libraries |

**Table 2-2: Overview of Considered Embedded Systems**

### 2.2.1 Comparison of Embedded Systems

Since the purpose of this project is to lay the foundation for remote audio recordings on board a quadcopter, a full embedded system will not be engineered from scratch. Rather, a partially designed and built platform will be used to start. Additional modules and components will then be added to make up for any features that are not included in the pre-built platform.

Four platforms were considered in the initial design phase. These platforms included Apple's iPod Touch (4th Generation), the Raspberry Pi (Model B), Nordic Semiconductor's nRF51822 Evaluation Kit, and Atmel's SAM4L Xplained Pro. A quick overview of these systems is given in Table 2-2 and detailed information about each system follows in sections 2.2.1.1-4.

#### *2.2.1.1 Apple iPod Touch (4th Generation)*

Apple's iPod Touch (4th Generation) is a complete embedded system bundled together in a very compact package. According to specifications published by Apple [**24**], it has numerous features that are all coupled together through an efficient custom ARM core processor. These features include high quality ADCs and DACs for audio recording and signal generation, respectively. The iPod Touch also includes both Wi-Fi and Bluetooth transponders, allowing multiple communication options.

**Figure 2-2: Apple iPod Touch**

On top of the software sits Apple's iOS operating system. The iOS allows only limited low level control of the operating system and peripheral devices. Any custom code is run as an app on top of the operating system and memory space is restricted. Apps do not have any guarantees in terms of processor time and therefore run times are non-deterministic. Custom code must be compiled on an Apple computer in order to run on an iPod Touch.

A battery is also included in the iPod Touch. Specific battery-life times will depend on how old the battery is and what peripherals are being used on the iPod Touch. Normal usage gives several hours of use before the battery requires recharging. While a battery is included in the design, the overall weight of the iPod Touch is just over 100 grams.

#### *2.2.1.2 Raspberry Pi (Model B)*

The Raspberry Pi was developed to be a low cost but highly functional embedded system. Its original purpose use was as a learning system where students could be introduced to embedded systems. However, it has found use in many areas besides the classroom because of the many features that are included and the low price point [**25**].

Raspberry Pi's Raspian operating system is the primary operating system, but users may also install several other operating systems or even develop their own operating system if they so choose. Since Raspian is based on Debian Linux, users are mostly able to install and run programs just as they would on a normal Linux computer. This allows for many different coding languages to be used and compiled. An altered Linux kernel can also be used when a real-time (deterministic) system is required. There is a caveat though, in that the GPU is not open source and is not real-time guaranteed. Any operations that use the GPU are not guaranteed to be deterministic.

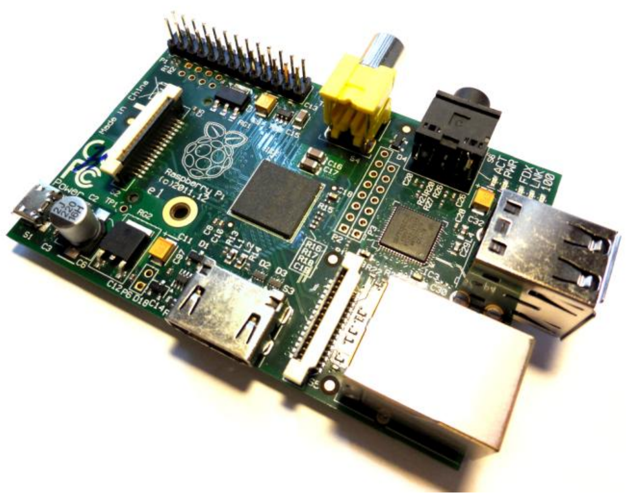


**Figure 2-3: Raspberry Pi**

Included on the Raspberry Pi are two USB headers, an RJ45 Ethernet port, a 3.5 mm headphone jack, an HDMI port, and a composite RCA video out. In addition, there are multiple GPIO headers that can provide UART, SPI, I$^2$C, and I$^2$S. For storage media, the Raspberry Pi requires an SD card in its SD slot, and this SD card must include the operating system. The Raspberry Pi is powered by a microUSB port or a GPIO header and draws 700 mA. The overall weight is 45 grams.

#### *2.2.1.3 Nordic Semiconductor nRF51822-EK*

Nordic's nRF51822 Evaluation Kit [**26**] was designed to showcase the features of the nRF51822 SoC [**28**]. As with most of Nordic's chips, the nRF51822 centers on a built-in wireless transceiver. The nRF51822 specifically works on the Bluetooth low energy protocol as well as a proprietary 2.4GHz protocol. The evaluation kit comes in two parts (as seen in Figure 2-4): the first is a larger development board with GPIO pins, buttons, LEDs, and the nRF51822 SoC; the second is a small USB dongle that removes the GPIO pins, buttons, and LEDs but still has an nRF51822 chip.

This chip features a 32-bit ARM Cortex-M0 CPU. It includes interfaces for SPI, I$^2$C/TWI, and UART. Also included is an 8/9/10 bit variable ADC, depending on sample frequency. Additional GPIO ports are available for other custom modules that may be needed. The evaluation kit includes a button battery holder but can also be run from the USB port

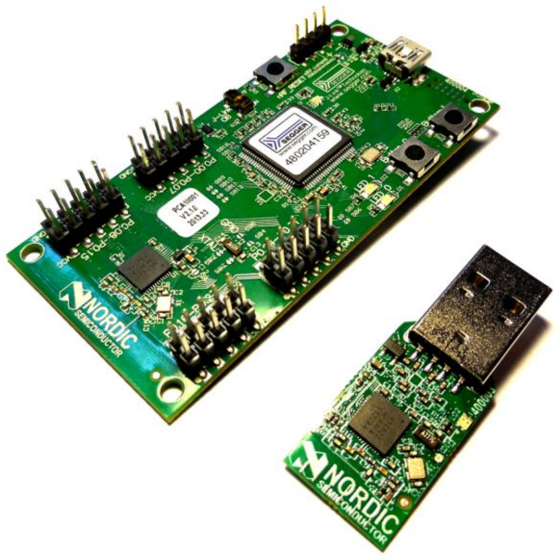


**Figure 2-4: Nordic Semiconductor nRF51822-EK**
The nRF51822 Evaluation Kit comes with both a development board and a USB dongle

or external headers. The board is an ultra low power consumer, using as little as 0.6 µA while asleep, and 13 mA when communicating at high speeds. Overall weight is lower than 100 grams.

Software development is performed using either the Keil MDK-ARM toolchain or the IAR Embedded Workbench toolchain. No base operating system is required for the microprocessor though there have been reports of installing real-time operating systems on the nRF51822 [**29**,**30**] and the Keil MDK-ARM tool does include the Keil RTX real-time operating system [**31**]. There are a number of libraries already built for the nRF51822, including a full Bluetooth stack. The nRF51822 relies on the internal 128kB/256kB flash memory for program storage and the nRF51822 evaluation kit includes an on-board Segger J-Link for programming the chip.

#### *2.2.1.4 Atmel SAM4L Xplained Pro Evaluation Kit*

Atmel's SAM4L series of microcontrollers are relatively recent additions to Atmel's microprocessor line-up. The SAM4L Xplained Pro Evaluation Kit includes the SAM4LC4 microcontroller as well as many peripheral pins. Additional modules are easily connected to the Xplained Pro via the Xplained Pro extension headers [**27**].

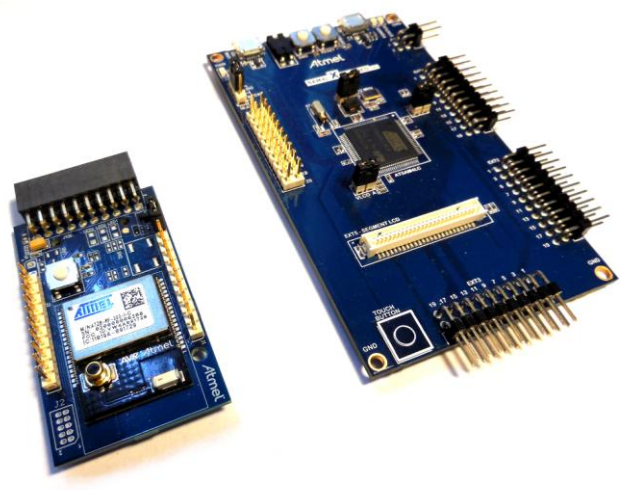


**Figure 2-5: Atmel SAM4L Xplained Pro**
Pictured with the AT86RF233 Amplified ZigBit Xplained Pro Extension

The SAM4LC4 includes an ARM Cortex-M4 processor. The overall weight is less than 100 grams, even with additional modules. It also has a USB port (host or device mode), SPI, $I^2C$/TWI, $I^2S$, and UART. Additionally, it also has multiple 12-bit ADC channels capable of up to 300 ksps, and multiple 10-bit DAC channels that can also output 16 bit audio bitstream data at sample frequencies of 8kHz – 48kHz. Power is provided by either the USB port or from external headers. Atmel's picoPower technology has also been included so that the microcontroller can run on low voltages and the chip has multiple sleep mode levels that significantly reduce current draw [**32**].

Programming the Xplained Pro is done through an on-board J-Link debugger. Software development is performed through Atmel Studio, which is Atmel's free IDE based on Microsoft's Visual Studio. Atmel Studio also includes the Atmel Software Framework (ASF) which has many pre-built libraries that abstract out the hard work of interfacing with the low-level peripherals. Many code examples and projects are also included as part of ASF. Debugging can be performed directly from Atmel Studio and the on-board debugger also sets up a USB virtual COM port on the user's computer for easy interfacing to the board.

### 2.2.2 Operating System

There are numerous operating systems available for embedded systems. Most embedded operating systems are real-time operating systems at the most basic level. A real-time operating system implies that the operating system can complete most tasks in a deterministic amount of time. If all tasks are always completed in a deterministic amount of time, the system is considered a hard real-time operating system; otherwise, an operating system that only generally completes tasks in a set amount of time can be considered a soft real-time system.

For audio signal generation and recording, a hard-real time operating system is normally required for certain tasks. For example, audio must be generated and sent to the DAC at a very specific frequency. The same is true for recording data from the ADC. Not sending data to the DAC at the proper speed or reading the ADC at the wrong time will result in corrupted audio. Other tasks are not as critical and can be completed when the operating system has some idle time. Of the embedded devices introduced in section 2.2.1, only the Nordic and Atmel processors are truly capable of having hard real-time operating systems.

The Apple iPod Touch's iOS is a real-time operating system at a very low level, but, as applications are abstracted several layers above, there are no real-time guarantees. However, peripheral interfaces, such as the ADC and DAC should complete in real-time such that there is no degradation in quality. So long as other tasks in the user app do not have any real-time needs, iOS is a good choice for an operating system.

Multiple real-time operating systems are available for the Raspberry Pi, though most can only provide soft real-time guarantees because the GPU is a closed system and there are no guarantees on how often it will interrupt the processor. Any sort of system that uses the GPU will lose any hard real-time guarantees.

The Nordic and Atmel processors are both capable of using hard real-time operating systems. ASF provides a port of the FreeRTOS operating system for the Atmel SAM4L boards and Keil IDE provides the Keil RTX real-time operating system ported to the nRF51822. Both operating systems are fully functional, providing most of the required basics of an RTOS including multitasking, semaphores and mutexes, message passing, and task prioritization.

### 2.2.3 Hardware Interfacing

Besides the microcontroller, several other components and interfaces are important for this project. As have been mentioned before, the ADC and DAC are critical for the task of recording and producing audio, respectively. A UART interface is needed in order to quickly and easily interact with the system. USB interfaces provide a way to save audio and build a more advanced interaction medium with the system that can be referenced by future PC software libraries. Finally, there is a need for some sort of wireless communication between the ground station and the system on board the quadcopter.

#### *2.2.3.1 ADC*

The Analog-to-Digital Converter (ADC) reads in the analog voltage from the microphone and converts it to a digital value. Using an ADC always represents a slight loss of precision from the analog signal. However, it is necessary in order to work with and store data in a fully digital system. Figure 2-6A demonstrates how an ADC quantizes the analog signal. The number of steps is dependent upon the number of bits available in the ADC. For example, an 8-bit ADC means there are $2^8$ or 256 possible steps. There are two primary types of ADCs. The first is a single-ended ADC which has just one voltage input pin and this is compared to ground. The second type is a differential ADC which has a positive and negative input and computes the voltage difference between the two inputs. A differential ADC also has the benefit of calculating negative voltages [**33**].

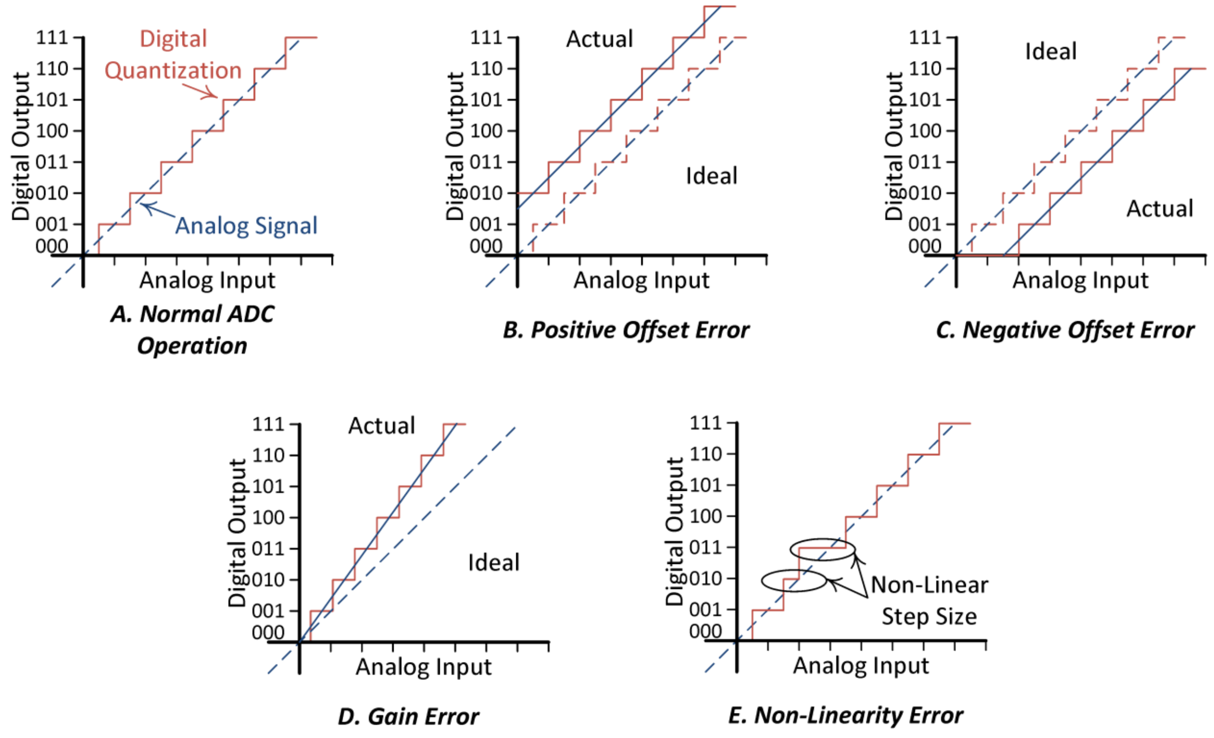


**Figure 2-6, A-E: ADC Operation and Error**

An ADC also includes a reference voltage that the analog voltage is compared to and is the highest possible voltage allowed by the ADC. Using an 8-bit single-ended ADC which has a reference signal of 1 V, each step will be 1/256 or approximately 3.9 mV. A 0.5 V input to the ADC would therefore receive a value of 128 output from the ADC. Any analog signal that is between two ADC voltage steps will result in a digital output value closest to the analog input. This means that there is a quantization error ±1/2 of the step size. With the example 8-bit 1V ADC, this would equate to a maximum quantization error of about ±1.95mv.

Another source of error is in the offset. Sometimes, the transition from digital value 0 to 1 does not take place at ±1/2 of the step size. When this transition takes place at a different location, the step graph can be translated vertically either upwards or downwards, as seen in Figure 2-6B and Figure 2-6C. A third source of error is in the gain. In this case, the step graph increases too quickly or slowly up to reference voltage, as shown in Figure 2-6D. The final source of error is in step graph non-linearity. At some points, the step size may be larger or smaller than other steps. Figure 2-6E illustrates this case. Correcting for quantization error simply means using a higher bit value ADC. Offset, gain, and non-linearity error can all be compensated for to some extent in code, though this can be difficult [**33**].

ADCs used for audio should ideally have as many bits as possible. Low quality audio may use as low as 8 bits, normal audio often has a 16 bit audio depth, and high quality audio often uses 24 bit. However, more bits also means more data to work with and larger audio files to read and write. This can result in significant increases in computation and memory access times. As a reminder, Nordic's nRF51822 chip has an on-board 10 bit ADC while the Atmel SAM4L has a 12 bit ADC.

#### *2.2.3.2 DAC*

Digital-to-Analog converters work in the reverse direction as the ADC, taking a digital value and converting it to an analog signal. All of the errors that affect ADCs also affect DACs, albeit in the reverse direction. Because of the quantization error, a typical DAC will result in a step like output, similar to Figure 2-7. The more bits that are available, the smoother the output will look, but a basic DAC will always result in a step response at the most detailed level. The Raspberry Pi includes a very basic PWM DAC for the headphone jack. This works well for low frequency sounds but the quality degrades in the higher frequencies on the Raspberry Pi.

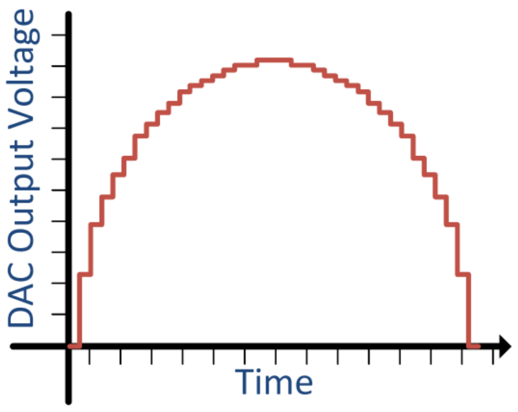


**Figure 2-7: DAC Output Example**

Specialized audio DACs also exist for generating output signals that are akin to analog audio signals and the Atmel SAM4L's ABDACB is one such example. The SAM4L sends the audio data through a CIC Interpolation Filter and then a Sigma Delta Modulator to achieve a better sound than the Raspberry Pi. To further improve the audio, Atmel recommends running this signal through a low pass filter as the filter and modulator create a significant amount of high frequency noise that can be seen on an oscilloscope [**34**].

#### *2.2.3.3 UART*

For systems that have a pre-built operating system, a UART interface is not normally necessary as some higher level of interfacing with the device is often included. This is true of the iPod Touch, which has the screen, and the Raspberry Pi, which has both a display hook up and native remote desktop connections. However, for the SAM4L and nRF51822, a UART interface is essential. Having a UART connection means that the board can be connected to a serial COM port on a PC (usually via an RS-232 cable or a USB virtual COM port). A simple text console on the board is then established and the board can be programmed to both send and receive text from the PC. Both the nRF51822 and SAM4L-EK include UART libraries for easy setup in the development process.

#### *2.2.3.4 USB*

USB is becoming the new UART and many new development boards include USB ports. The USB standard uses only four pins: Data+, Data-, a 5 VDC rail, and a ground pin. The Data pins are a balanced pair so that the external interference is lower and so the length of USB cables may be increased. Normally, development boards act simply as a USB device, allowing some other device (namely, a PC) to act as a host, but some development boards are now implementing the USB-On-The-Go standard or the full USB host mode. This is true of the SAM4L which can act as either a device or a host.

When the development board acts as a host and a USB flash drive is plugged into it, it will then have the capability to read the files that are currently on the flash drive and add files to it.  This is especially important as audio recordings from the quadcopter will be directly stored on the USB flash drive.  The board will also act as USB device when plugged into the PC, allowing the PC to send and receive data in much the same way as the UART.

### 2.2.4 Wireless Communication

Wireless communication is especially important in this project as the system on the ground cannot have a physical connection to the quadcopter.  Therefore, some form of wireless protocol between the two will be need in order for the two nodes to properly indicate what events are taking place.

#### *2.2.4.1 Wi-Fi*

Wi-Fi is one of the most ubiquitous communication protocols available.  Most laptops, tablets, and mobile phones now include it.  The 802.11 standards are very thorough in how they should be setup and there are many integrity constraints along the way to reduce the transmission of corrupted packets. Combined with TCP/IP, Wi-Fi is extremely powerful.  Many libraries exist for TCP/IP communication and there are many Wi-Fi modules available that also come with their own driver libraries.

That said, the Wi-Fi stack is very large and typically requires a large amount of memory.  Further, Wi-Fi has lots of areas where latency can be induced.  This makes it difficult to rely on Wi-Fi for time sensitive tasks.  There are never any time guarantees for wireless communication and the Wi-Fi standard is particularly susceptible to task deadline overruns.

The Apple iPod Touch is the only embedded system examined in this project that includes a Wi-Fi module.  The Raspberry Pi has TCP/IP protocols installed and many wireless module drivers are available, but an additional USB Wi-Fi adapter would be needed.  The Nordic and Atmel boards do not include any Wi-Fi modules.

#### *2.2.4.2 Bluetooth*

Bluetooth was originally meant to serve as a wireless replacement of RS-232, the hardware layer of a UART connection.  It has advanced since then but still operates in a similar principal, allowing the user to open a COM port on the PC to communicate with a UART on the processor.  While Wi-Fi was meant to create a large wireless network over a given space, Bluetooth is more about smaller “personal” networks and working directly with peers.

The size of the Bluetooth stack is typically smaller than the Wi-Fi stack.  Similar to Wi-Fi, there are many libraries and modules that are built around Bluetooth that can be used as add-ons to the development boards.  Since the Bluetooth stack is smaller and not as complex as the Wi-Fi stack, it is easier to track clock usage by the Bluetooth library.  It is still not completely deterministic because of the wireless communication, but it should be possible to have a more deterministic library.  With the introduction of the Bluetooth Core Specification 4.0, there also is a low energy version of Bluetooth. This architecture significantly reduces the power usage over classic Bluetooth and allows Bluetooth low energy devices to run for extended amounts of time on just coin cell batteries.

Only the iPod Touch and the Nordic nRF51822 implement the Bluetooth standard. The iPod Touch uses the classic Bluetooth standard while the Nordic nRF51822 uses the Bluetooth Low Energy standard. Atmel's SAM4L and the Raspberry Pi would require external modules or adaptors in order to implement Bluetooth.

#### *2.2.4.3 ZigBee*

In contrast to Wi-Fi and Bluetooth, ZigBee is largely intended as a mesh networking technology. There is no need for a central controller that can reach all of the nodes. ZigBee is intended for applications that need only a low data throughput and do not need to send data often. This is particularly useful for embedded sensors and automation controls.

ZigBee is also intended to be much simpler and cheaper than both Bluetooth and Wi-Fi. The stack size of ZigBee is significantly smaller than the other two and can be implemented by smaller microprocessors without issues. This simplified structure makes it much easier to track how many cycles each step of the transmission and reception of data will take, apart from the non-deterministic time required for successfully transmitting a message wirelessly.

None of the embedded devices here have a ZigBee module included. However, Atmel has an add-on ZigBee module that is meant to easily attach to the SAM4L-EK. ASF also includes the library for operating the ZigBee module and setting up the ZigBee stack, allowing for quick creation of ZigBee products.

## 2.3 Synchronization

Events occurring between multiple systems often need to be synchronized. This is especially true for systems that pertain to audio. The following sections discuss the requirements for audio synchronization, possible methods for synchronizing systems, and what latencies can be expected in a network.

### 2.3.1 Requirements for Audio Recording

Under normal circumstances, impulse responses are recorded using the same sound card. This allows the recording and playback audio to begin at the exact same sample. Each additional recording and playback sample is continually synchronized as there is only one clock on the sound card that controls the timing of the samples.

When using a quadcopter, utilizing the same sound card for both playback and recording is not an option. Instead, the sampling on the two nodes needs to be synchronized. Audio is typically recorded at sampling frequencies from 8 kHz for low quality wireless microphones to 48 kHz for professional audio equipment. To have sample accurate clocks, the accuracy must therefore be 125 μs for 8 kHz and 20.83 μs for a 48 kHz sample frequency.

A goal for this project therefore should be to maintain a clock accuracy of at least 21 μs in the worst case scenario. If this occurs, it should be possible for the ground node to know exactly when the quadcopter node begins recording, and for the quadcopter node to know exactly when the ground node begins playback.

## 2.3.2 Timing-sync Protocol for Sensor Networks

There are a number of ways of performing time synchronization across networks. The most commonly used is the Network Time Protocol (NTP). It works on the application layer of the OSI model, sending data as UDP packets across the network from a reference computer to the client computers. However, NTP has a limited accuracy. In a local area network with full speed Ethernet connections, the accuracy can be as low as 100 μs. If the speed is reduced, the connection is changed (for example, to wireless), or if the network is enlarged so that the number of nodes between the reference computer and client computers is increased, the accuracy of synchronization can be reduced to as much as 100 ms [**35**].

Unfortunately, 100 μs is not accurate enough for audio synchronization, especially when this accuracy is only plausible under ideal conditions. Instead, another protocol must be used that can obtain a much better accuracy. Another commonly used protocol is Reference Broadcast Synchronization (RBS), however this has the disadvantage of requiring a third node [**36**]. Instead, a simpler and less common option is the Timing-sync Protocol for Sensor Networks (TPSN). TPSN can be implemented across just two nodes and in theory has twice as much accuracy as RBS. Implementations of TPSN by the designers led to network synchronization accuracy of less than 20 μs [**37**].

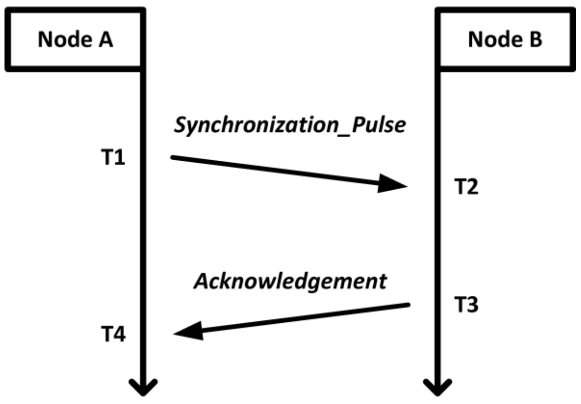


**Figure 2-8: TPSN Messaging Passing Scheme**
Node A sends a synchronization pulse at time T1 that is received at time T2 by Node B. Node B sends an acknowledgement back at time T3 which is received by Node A at time T4.

Figure 2-8 shows the basic TPSN message passing between two nodes, A and B. At time T1, node A sends a *synchronization_pulse* message to node B containing the time T1. Node B receives this at time T2. In turn, node B sends an *acknowledgement* message to node A at time T3 containing T1, T2, and T3. Node A receives this message at time T4. At this point, node A can calculate the clock drift Δ and propagation delay *d* using the formulas:

$$\Delta = \frac{(T2-T1)-(T4-T3)}{2}$$

and

$$d = \frac{(T2-T1)+(T4-T3)}{2}.$$

Now that node A knows the clock drift for node B, it can correct its own clock to match that of node B. In addition, node A can predict the amount of time that a message takes to be transmitted to node B by the propagation delay [**37**].

## 2.3.3 Network Latencies

The reason for such inaccurate time using NTP as compared to RBS or TPSN is due to where the event time-stamping occurs. Communication systems are often described by the OSI model, as seen in Figure 2-9. The OSI model breaks down communication into seven layers of abstraction. The uppermost layer is the application layer where the end user is directly involved with the software beginning the communication process. The lowest layer is the physical layer that defines the electrical and physical components of the communication interface.

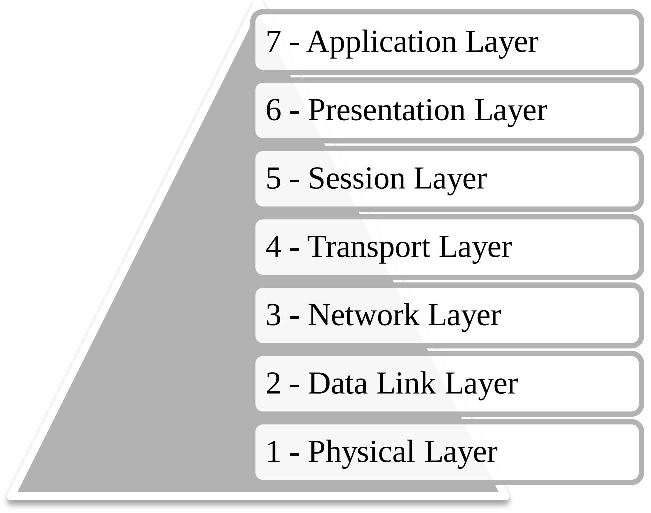


**Figure 2-9: Open Systems Interconnect (OSI) Model**

As mentioned previously, NTP works at the application layer. Being in the highest layer of abstraction, there are six lower levels of abstraction that NTP must work through before the message is even transmitted, and then it needs to work back up seven layers on the receiving end in order to be processed. Each layer adds its own amount of latency to the system. It is also difficult for an upper layer to know exactly how a lower level works and nearly impossible for a lower level to know anything about an upper level. This amount of uncertainty makes it difficult to know exactly how long a process will take, especially when sitting at one of the higher levels, as NTP does.

RBS and TPSN include functionality at a much lower level of abstraction. They recommend performing the time-stamping at the MAC/Data Link Layer, which is layer two. This removes a large amount of the uncertainty that is generated in the upper layers, resulting in significantly more accurate clock synchronization.

## 2.4 Audio Measurements

The Introduction described the current process for performing audio measurements but there are a few more details that are needed in order to complete the process of making audio measurements. First, a discussion about the microphone and loudspeakers is needed, since the quality of these components can affect the final quality of the impulse response. Second, a discussion of the signal that is sent out over the loudspeaker should also be made.

### 2.4.1 Microphone and Loudspeakers

In the ideal circumstance, both the microphone and the loudspeaker have perfectly flat frequency responses across the entire spectrum of frequencies that are of interest. In reality, the frequency response of loudspeakers and microphones vary significantly over different frequencies. This variance is what characterizes microphones and loudspeakers that sound different from one another. Accordingly, these variances should be removed in final impulse response calculations because the desired impulse response is of the room, not of the recording hardware.

The microphone used in the final design implementation is the Knowles Electronics SPM0408LE5H-TB. This is a MEMS microphone that is small enough to fit in a hearing aid but is also suitable in mobile phones, laptops, digital cameras, and other small embedded systems. It uses a low power design while also providing wideband audio performance and RF immunity. It also includes an output amplifier that delivers up 20 dB of gain [**38**].

Multiple sets of loudspeakers are used during testing. Where appropriate, the type of loudspeaker used is indicated in the results. However, the final design implementation is not limited to one type of loudspeaker and can be connected to a variety of amplifiers and speakers using standard audio connectors.

### 2.4.2 Swept Sine Signal Generation

To perform the impulse response measurement, the loudspeakers must first output an exponentially swept sine signal. In its basic form, this is an audio signal that starts at a low frequency and exponentially grows to a higher frequency over time. After the microphone records the swept sine signal, it is possible to compare the output and input signals to see at what frequencies the microphone performs the best and at what frequencies the microphone loses quality.

The swept sine signal is calculated using the following equation:

$$signal = \frac{1}{2} * \sin\left(2 * \pi * \frac{F_{start} * T_{sweep}}{\ln \frac{F_{end}}{F_{start}}} * \left(e^{\frac{t}{T_{sweep}} * \ln \frac{F_{end}}{F_{start}}} - 1\right)\right),$$

where the output *signal* is a function of the starting frequency $F_{start}$ in hertz, the ending frequency $F_{end}$ in hertz, the duration of the signal sweep $T_{sweep}$ in seconds, and the current time $t$ in seconds. Depending on the impulse response desired, different Fast Fourier Transforms (FFTs) and Inverse FFTs (IFFTs) can be taken of the recorded signal and the frequency response of the system can be found from there.

## 2.5 Imposed Project Goals, Challenges, and Restrictions

The project's overall goal is to develop an embedded system that is capable of making audio recordings from a quadcopter. In the project description, several technical challenges are presented that need to be met in order to have a good quality measurement. Several additional challenges and restrictions were identified from the background of quadcopters, embedded systems, and audio measurements. In full, these challenges and restrictions are presented in Table 2-3.

With these goals and restrictions in mind, it is possible to begin generating a prototype recording system for the quadcopter.

| Identification Area | Goal | Description |
|---|---|---|
| **Project description** | Noise cancellation | Quadcopters generate a lot of noise. How can this noise be cancelled? |
| | Noise reduction | Are there ways of reducing the amount of noise generated by a quadcopter? |
| | Lightweight measuring device | How can a system with microphone, signal processing, and recording storage be designed? |
| **Quadcopter imposed challenges** | Maximum payload of 100 grams | Reducing payload will make the quadcopter more stable and increase flight time. |
| | Low power design | Any embedded design on the quadcopter should utilize a low power design in order to extend battery life. |
| | Low noise recordings | Remove any noise generated from quadcopter. |
| | Increased microphone stability | Prevent microphone from moving during recordings as much as possible. |
| **Audio measurements and embedded systems** | Synchronization within recording sample rate | For recordings to really be useful, the exact time that both playback and recording started must be known. |
| | Meet ADC and DAC hard real-time deadlines | Missed ADC or DAC deadlines will result in corrupted audio, so data must be recorded from/fed to the peripherals in time. |
| | Quality audio performance | Hardware must be able to produce good quality sound output and make good quality sound recordings. |

**Table 2-3: System Challenges and Restrictions**

# 3 Prototype Implementation on Apple iOS

The first implementation was a rapid prototype that was meant to collect a large amount of data without needing a lot of setup work. For this, the Apple iPod Touch was chosen for several reasons. First, the iPod Touch represents a complete system that has been pre-constructed. No further work putting together hardware would be necessary in order to start making recordings. The iPod Touch also is a device whose primary purpose is for audio, so the audio quality of recordings should presumably be of decent quality.

Using the iPod Touch also meant that there was a large library of pre-built apps available, further reducing the amount of time required to make recordings. Once an app and a communication protocol were selected, the only development was in generating the sine sweep signal on a PC and outputting it over a loudspeaker for the iPod Touch to record.

## 3.1 Prototype Layout

The basic layout of the system consisted of the iPod Touch as a remote node that could be attached to the quadcopter and then a PC that acted as a ground node, as seen in Figure 3-1. The iPod Touch performed the recordings based on commands sent to it by the PC. The PC was in control of generating the audio signal that was sent to the speakers as well as controlling the synchronization of the recording and the playback of the audio.

Since the remote node was the iPod Touch, both Bluetooth and Wi-Fi were available as wireless communication methods. Choosing between the two ultimately came down to whichever was easier

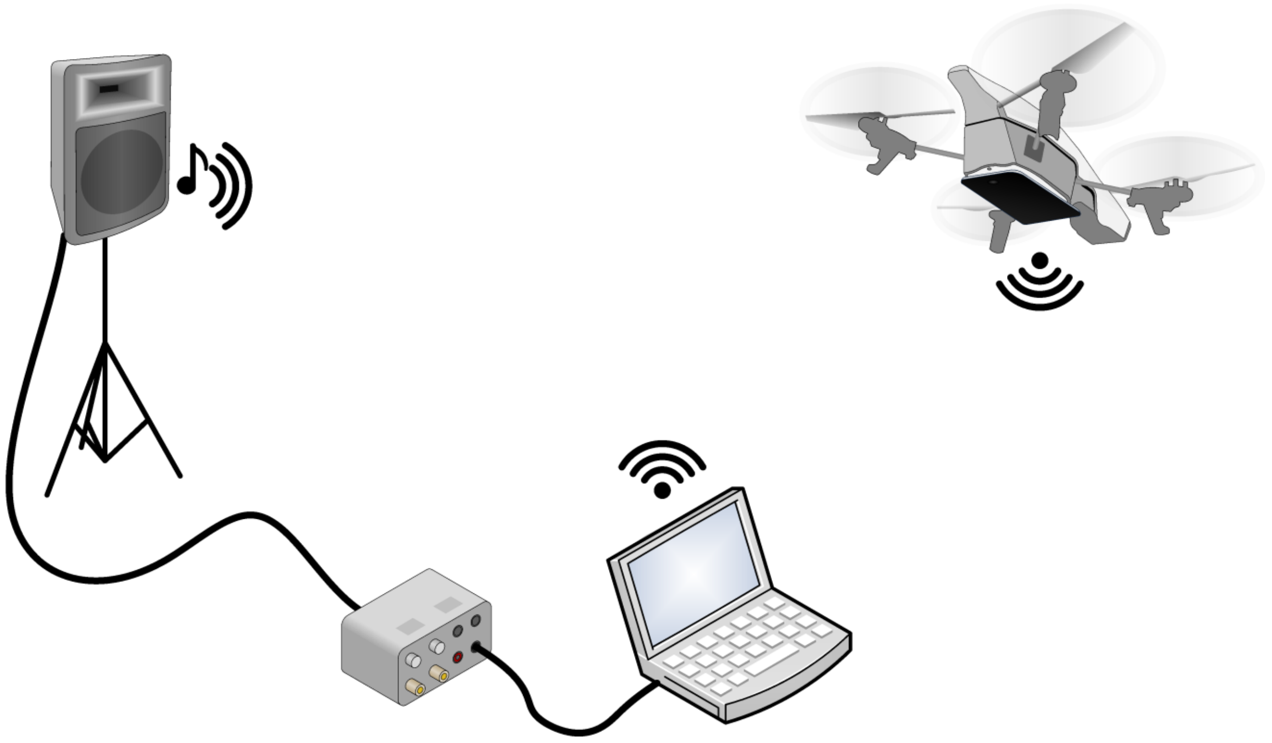


**Figure 3-1: iPod Touch Setup**
The PC generates the sine sweep signal which goes out through an amplifier to the loudspeakers. At the same time, the PC notifies the iPod Touch to begin recording audio.

and likely to whatever the pre-built app supported.

As stated, the app running on the iPod Touch performed the recordings and was a pre-built app available publicly on the Apple iTunes store. This app needed to have the ability to be controlled from a remote PC. Further details on the requirements for the app are found in section 3.2.1. The PC also needed some form of an application running on it that performed two tasks. The first task was to generate the required audio signals that were to be played over the loudspeakers. The second task was to start playing this audio at the same moment that the app on the iPod Touch began recording audio. There were multiple possibilities for how this PC application could be written, depending on how the iPod Touch app's communication protocol functioned, how simplistic or complex the control algorithm needed to be, and how much control over the system was desired.

## 3.2 Embedded Application

Apple allows many developers around the world the ability to write applications for their line of mobile devices all running the iOS operating system. These apps are normally written in a high-level language that then is compiled into native code required for the iPod Touch. Apps written for iOS devices are uploaded to the Apple iTunes store and made available to users around the world for free or for a small fee.

These publicly available apps were the source for the embedded applications used on the iPod Touch in this application. A custom app could have also been written, but this would have added a considerable amount of time to the prototype design phase. Additionally, there was a plethora of varying quality apps already available on the Apple iTunes store that offered the features needed for this project.

### 3.2.1 App Requirements

The iPod touch app had several requirements that had to be met in order for it to be a viable option for recording. These requirements are outlined in Table 3-1.

| Requirement | Details |
|---|---|
| **Real-time operation** | Needed to operate by meeting certain deadlines with only minimal amounts of delay. |
| **Remotely controlled** | Operation needed to be controlled from a device located on the ground. Having to physically interact with the iPod Touch for each recording was not an option. |
| **Bluetooth or Wi-Fi communication** | Communication needed to be via Bluetooth or Wi-Fi as these two standards were the only two supported by both the iPod Touch and the PC. |
| **High audio recording quality** | Audio quality from the recordings ideally needed to be near CD-quality. |
| **Well designed** | Other user reviews of the app should indicate a positive experience. |

**Table 3-1: iPod Touch App Requirements**

### 3.2.2 App Selection

While multiple applications were reviewed, the final app that was selected was Appologics's AirBeam [**39**]. AirBeam managed to meet all of the requirements from section 3.2.1 as well as provide several additional features. The app markets itself as "high definition video monitoring with your iDevices." Control of the app is normally performed via a web browser on the client's PC. Video from the iOS device is streamed directly to the user's browser in near real-time.

The user also has the ability to start and stop recordings directly from the browser, only touching the iPod Touch at the beginning to start the AirBeam app. In addition, recording quality settings can be set inside of the app. Other users have also given this app very good reviews and it is available for download for only $3.99 [**40**].

### 3.2.3 Functional Design

AirBeam functions using a client-server model. When the AirBeam app is started, it also begins a small server instance on the iPod Touch for handling any requests that come from outside clients. Clients may either be other iOS devices running the AirBeam app in "Monitor" mode, or users using their web browser. Since the ground node in this project was a PC not running iOS, the web browser was the most applicable method for accessing the server.

Communication to the iPod Touch server was performed using HTTP requests. Web browsers make GET requests for certain pages and files and the iPod Touch responds by delivering those files. Control of the camera and audio functions is also made by GET HTTP requests. The web browser makes an AJAX request to a specific "page", and the iPod Touch server responds by turning on the camera, starting a recording, stopping a recording, etc.

The actual web page served up by AirBeam is seen in Figure 3-2. In the center is a live "video" stream of what the iPod Touch is currently viewing, although this is actually a JPEG image that is updated continuously. Pressing the "Record" button initiates a recording. The "Stop" button finishes the recording. Pressing the "Recordings" menu tab displays all of the past recordings and allows the user to download them on to their own device.

## 3.3 PC Application

To make audio measurements, the app needed some form of audio to record. This task came down to the PC to complete, along with notifying the AirBeam app that a recording should be started. Two PC applications were developed. The first application was written as a simple add-on to the webpage interface provided by the AirBeam app. The second used a slightly more complex desktop application written in C# in an attempt to remove some of the latencies introduced by the web browser.

### 3.3.1 Web/HTML Interface

Since the AirBeam app provides a webpage interface, it made sense to fully utilize this interface when performing the audio measurements. By creating a "bookmarklet," external Javascript could be run on top of the AirBeam app webpage.

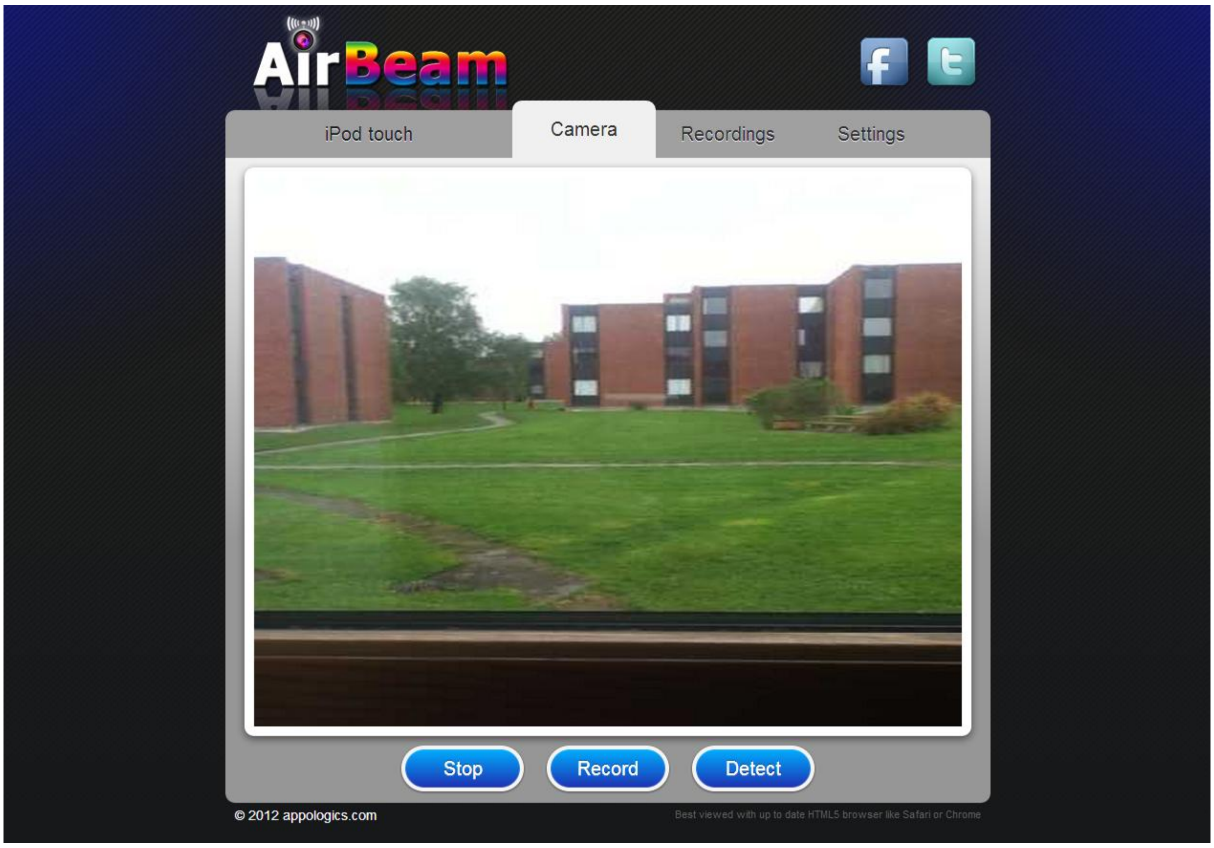


**Figure 3-2: AirBeam Webpage Interface**

#### *3.3.1.1 HTML App*

Visually similar to a browser bookmark, a “bookmarklet” appears as a link in a browser’s “Bookmarks” or “Favorites” bar. However, instead of linking to a webpage, a bookmarklet lists Javascript as its source URL. This allows custom Javascript to be run on a page whenever the user clicks on the bookmarklet. A number of websites utilize bookmarklets to add functionality as the user is browsing the web. For example, Facebook provides a bookmarklet to share a webpage whenever someone clicks on the “Share on Facebook” bookmarklet in their browser toolbar [**41**].

The bookmarklet developed (see Appendix A.2) in this project utilized a new web standard called the Web Audio API [**2**]. As part of the latest HTML standard, the Web Audio API provides advanced ways for developers to generate and work with audio using simple Javascript commands. The swept sine signal was generated using the Web Audio API.

#### *3.3.1.2 Swept Sine Generation*

The Web Audio API provides an oscillator that outputs an audio signal at a set frequency. It can also be set to go from one frequency to another at an exponential rate. This feature was used as an integral part of the HTML bookmarklet app. After requesting start and end frequencies as well as the duration, the Web Audio API generated an oscillator, set the two frequencies, and set the time at which the end frequency should be reached. It then began sending the oscillating swept sine signal to the audio card.

#### *3.3.1.3 Audio Synchronization*

To synchronize the audio playback and the audio recording, the recording was started immediately after starting the audio playback. This was done by simulating a user clicking on the “Record” button on the AirBeam app webpage. In Javascript, such a simulated click is a single line of code and allows a much more automated and timely process of pressing a button, rather than assuming the user will click the button at the correct moment every time.

As an additional feature, the start of the audio playback could be delayed by a certain amount of time. This was useful when making measurements on the accuracy of the system. If the recording consistently did not start on time, it could be calibrated with this delay.

### 3.3.2 C# Desktop Interface

After a bit of testing and measuring of the HTML app, it was apparent that the HTML app had certain timing inconsistencies that would result in poor synchronization between the PC and the iPod Touch app. In an attempt to gain more control over the timing synchronization, a second PC application was developed. This option removed the web browser from the system and instead relied on a lower level library to make the HTTP request and could provide for more accurate timing than Javascript could.

#### *3.3.2.1 Application Design*

The application was written in C# using the .NET Framework and Microsoft Visual Studio. The initial design used a console application that made a request to the AirBeam app to begin recording and then begin outputting audio to the sound card. Using a timer, the C# app was able to measure precisely when the audio had finished playing and when the recording should be finished. At this point, a second request was made to the AirBeam app to stop the recording.

HTTP requests were made using the HttpWebRequest Class provided by the .NET Framework [**42**]. The same page requests that the AirBeam app webpage uses were utilized in the C# application. Appologics has not released any formal documentation on these page requests, but they can easily be determined by reviewing the Javascript included on the AirBeam app webpage. This is available on any web browser that allows viewing page source code.

Under the web browser program, any script that is running on a webpage may be interrupted by the web browser for other tasks that may need execution in the same thread. With the C# application, there was no longer a chance that the program will interrupt itself to execute any other code. Further, the C# application raised its CPU priority level momentarily while the application ran, thus decreasing the number of times that the application would be delayed by other applications requiring processing time. Finally, by utilizing a C# application, the executing code would be closer to the hardware on which it was operating, as opposed to the Javascript that first needed to be interpreted by the web browser.

#### *3.3.2.2 Swept Sign Generation*

As a way to reduce computation time, the sine sweep signal was pre-calculated using MATLAB (as demonstrated in Appendix A.1.1) and saved as a WAV file. This MATLAB script used the formula from section 2.4.2. The C# application was then able to pre-load this file and begin playing at the desired time with reduced latency. The .NET SoundPlayer Class was used to load the file and begin playback [**43**]. All further work for loading the file and outputting the sound was taken care of by the .NET Framework.

#### ***3.3.2.3 Audio Synchronization***

With the same reasoning as the HTML app, the audio playback could be delayed if necessary. Synchronization worked by first pre-loading the audio from the WAV file using the SoundPlayer Class. The AirBeam was then requested to start playing. When the HttpWebRequest Class received an acknowledgement that the request had been received properly, it would wait the requested delay and then inform the SoundPlayer Class to begin audio playback.

Once the end of the recording time was reached according to a timer that was started at the beginning of the audio playback, the second HTTP request was sent to the AirBeam app to stop the recording. The AirBeam app would send an acknowledgement back to the HttpWebRequest Class if it was successful in receiving the message.

# 4 Final Design Using Atmel SAM4L Xplained Pro

After the initial prototype with the iPod Touch, a number of revisions were necessary for the next design. More control over the timing requirements were required than the iPod Touch could provide and any additional methods of noise-cancelling would improve upon the design of the iPod Touch.

The proposed system was built from a much lower level than the iPod Touch, beginning with a development board with a SoC processor. For the second design, it was decided that a real-time operating system capable of meeting hard real-time deadlines was required. An ADC would be required for recording the microphone and a DAC also ended up being required in the final design for generating the audio output signal. Sensors for reading propeller position were also included in the design for potential future active noise cancelation.

The three remaining development boards listed in section 2.2.1 were considered for the second design implementation and the Atmel SAM4L Xplained Pro was chosen. This board provides on-board ADC and DAC at an equivalent or higher bit depth than the other options. The SAM4L processor also provides a newer and higher performance processor than the nRF51822 or Raspberry Pi. Memory is much more limited on the SAM4L than on the Raspberry Pi. However, the SAM4L has a hard real-time operating system that has been thoroughly tested, unlike the Raspberry Pi. ASF also has built-in support for a ZigBee type wireless network, reducing implementation time.

*Note: For development, Atmel Studio 6.2 and ASF3.15 (Feb 2014) were used. As of the submission date of this work, ASF3.16 (Apr 2014) and ASF3.17 (May 2014) have been released. However, these updates remained untested, though no significant SAM4L changes are mentioned in the release notes and the applications should work without issue.*

## 4.1 Design Layout Options

Three primary design layouts were considered. The primary difference between all three designs lay with deciding what role the PC played. In the first layout, the PC played the primary role as the ground node, including maintaining synchronization and playing audio, with the help of a USB dongle. In the second layout, the process of synchronization was moved to a complete ground node attached to the PC, but the PC still had the responsibility of producing the audio output. The third design moved the audio output to the external node as well, leaving the PC to only issue commands for performing audio measurements and receive status information from the ground node about the system.

### 4.1.1 Quadcopter Node with PC Application and USB-ZigBee Adapter

The first layout includes the remote node aboard the quadcopter performing the audio recordings. The PC is the ground node and includes a USB-ZigBee dongle. A general layout of this is seen in Figure 4-1.

The USB-ZigBee dongle would contain a small microcontroller on it to perform the wireless communication and would also handle the synchronization protocols. The PC would issue commands for

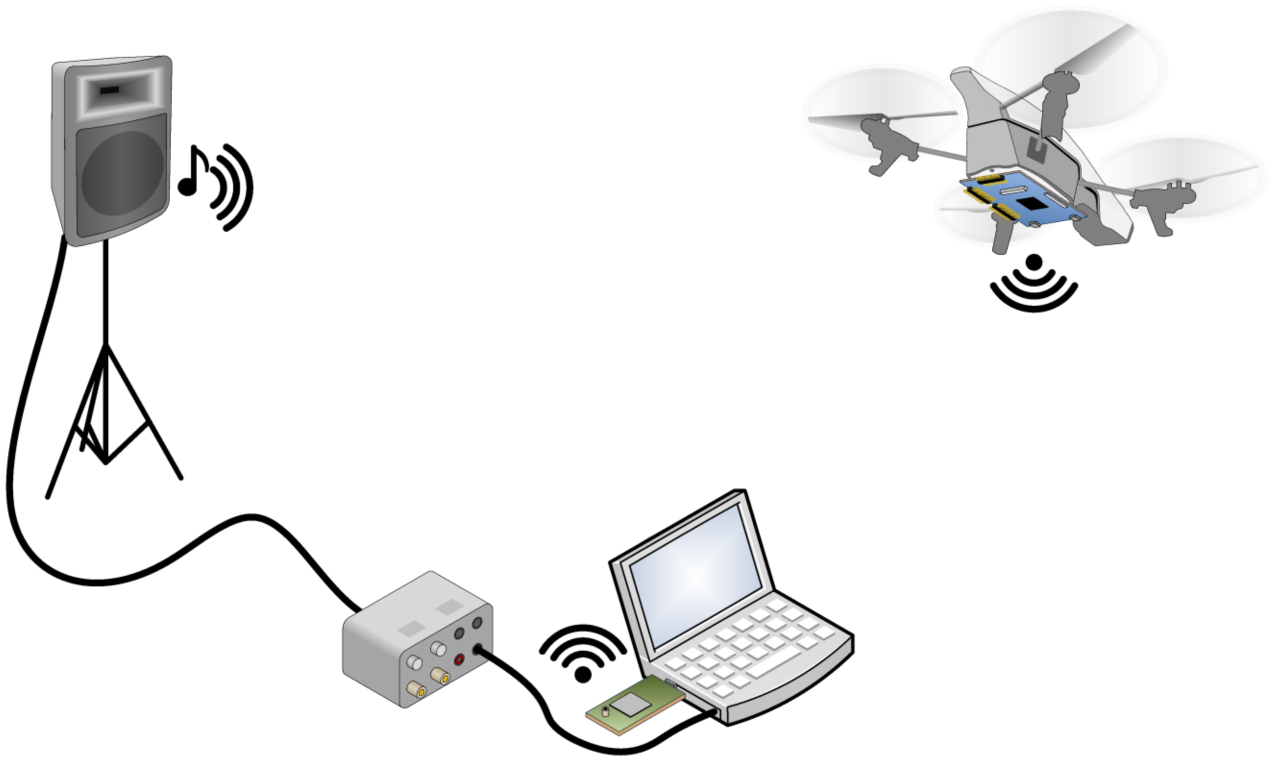


**Figure 4-1: Final Design Layout Option 1**
Remote node located on quadcopter with USB-ZigBee dongle attached to PC. PC creates the sine sweep signal that is sent out to the amplifier and on to the loudspeakers.

telling the dongle to perform a synchronization check as well as to tell the remote node when to start recording. The PC would be in charge of checking its synchronization with the dongle and then making sure it is generating and playing the audio out to the sound card at the correct time.

An advantage to this setup is that it reduces the overall complexity of the system down to the remote node, a USB dongle with only a small amount of code, and the PC as the ground node. The PC can have code written and compiled for it in any number of different languages, thus providing more flexibility for the PC.

The disadvantage of this system is ASF that does not provide the real-time operating system for the USB dongle, which brings a risk of non-determinism into the system. In addition, there are at least four clocks that would need to be tightly synchronized in the entire system: remote node clock, USB dongle clock, PC clock, and audio card clock. All of these clocks are organized in a chain between each other with no single way to interact with and synchronize all clocks at one time.

### 4.1.2 Quadcopter Node, Base Station Node, and PC Interface with Audio

To cover the issue of non-determinism in the USB-ZigBee dongle, the second design replaces the USB-dongle with a second SAM4L Xplained Pro as the ground node. This is depicted in Figure 4-2. The ground node connects to the PC via a full USB connection.

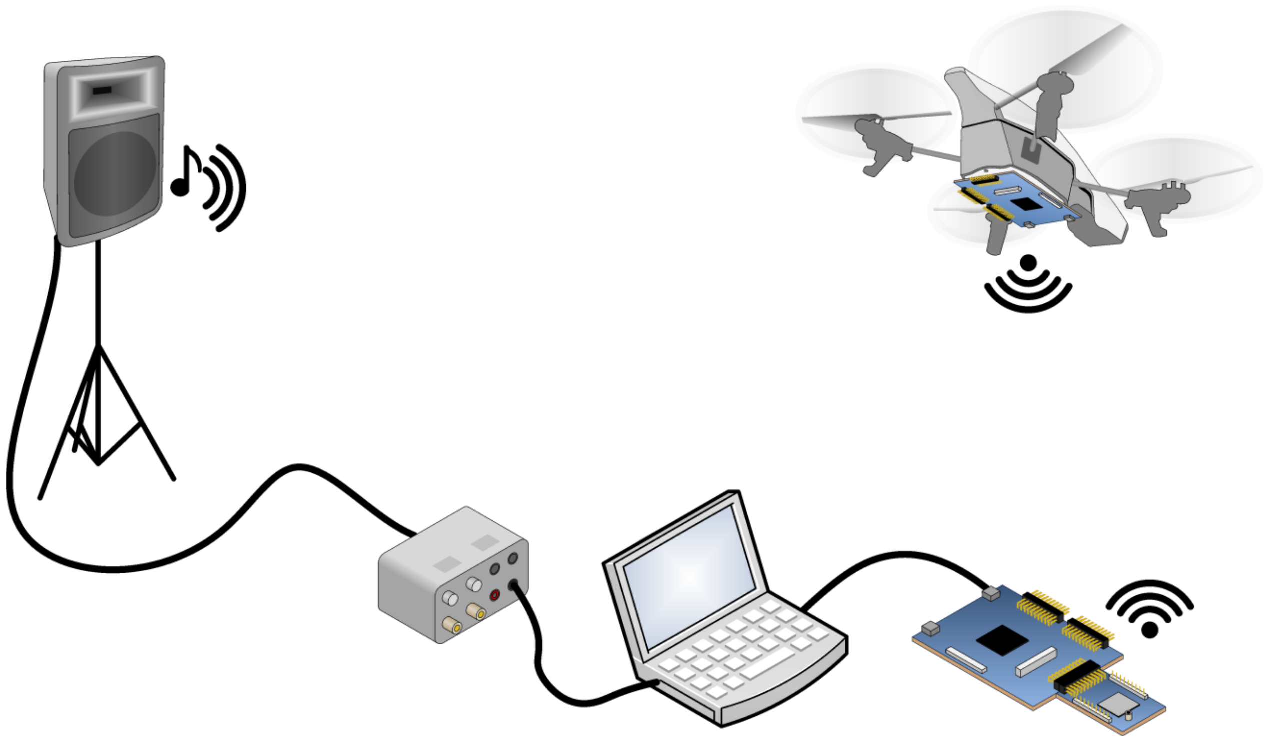


**Figure 4-2: Final Design Layout Option 2**
Remote node located on quadcopter with second ground node attached to PC. PC creates the sine sweep signal that is sent out to the amplifier and on to the loudspeakers. Ground node performs wireless synchronization.

The advantage of this system is that the ground node now runs the same real-time operating system as the remote note on the quadcopter. Additionally, any code synchronization libraries written for the remote node could be written generically to work for the ground node as well, reducing the amount of coding that would need to be performed.

However, this layout still leaves the problem of chained clock synchronization open. There also is a problem with USB. According to the specification [**44**], the USB protocol is strictly a polling protocol. This makes synchronization and issuing of information across USB subject to potentially long and indeterminate delays while waiting for the USB to perform its next polling routine.

### 4.1.3 Quadcopter Node, Base Station Node with Audio, and PC Control

The final design removes the issue of needing four chained clocks to be exactly synchronized. It moves the audio output from the PC to the ground node, as seen in Figure 4-3. Audio produced by the SAM4L ground node is performed by the same physical clock as the clock that performs synchronization with the remote node. This reduces the number of clocks from four to two: one each for the ground and the remote nodes.

As the final layout only contains two clocks, the likelihood of being able to synchronize the audio signal from the loudspeaker to the microphone recording is increased. The layout also simplifies the amount of code needed on the PC, as the PC only needs to issue commands to perform a recording. Finally, the third layout also allows more code between the remote node and ground node to be shared, potentially reducing the amount of implementation work necessary.

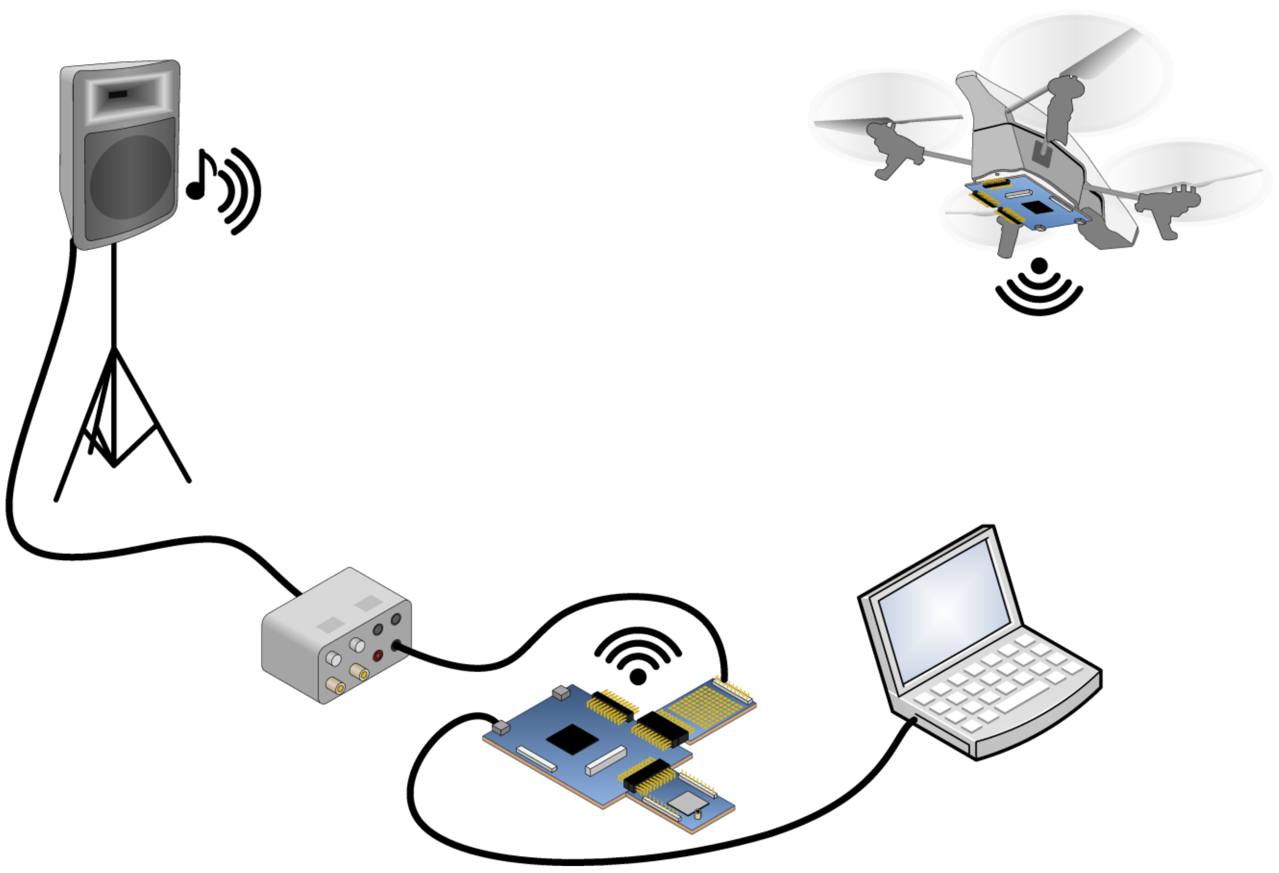


**Figure 4-3: Final Design Layout Option 3**
Remote node located on quadcopter with a second ground node attached to PC via USB. Ground node performs wireless synchronization and generates the sine sweep signal. PC sends and receives data to ground node.

There are several disadvantages of moving the audio to the ground node. The first disadvantage is that the ground node now needs physical hardware to support the interfacing of an amplifier and/or loudspeaker to it. Second, there are far fewer pre-built libraries and functions available on a development board than on a full PC. A couple of lines of C# code on a PC may result in four times as many lines of code in C on a development board, resulting in more work for the implementer.

### 4.1.4 System Design Overview

Despite the disadvantages of the third layout, it was the layout chosen. Increasing the accuracy of the clock synchronization was of most importance and the disadvantages could be overcome with additional time and work.

Based on the decision to have two SAM4L Xplained Pro nodes, each node needed specific hardware and software modules. Figure 4-4 outlines the basic modules that each node needed.

The ground node included two options for interfacing with the computer: UART and USB. UART was meant to be a simple interface that any computer can connect to and instantly start working with the boards. The USB interface was a slightly more advanced interface that included a PC library intended to be used in applications for communicating with the board. The ground node also included

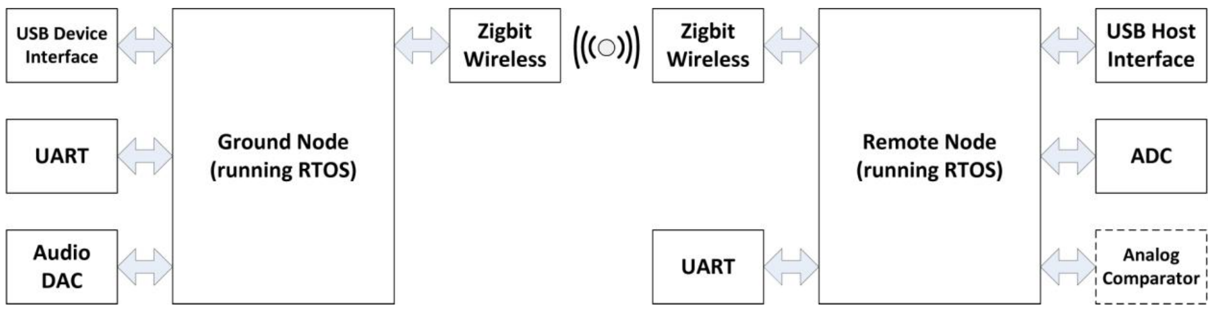


**Figure 4-4: Final Design Module Overview**
Block outline of the modules required in the final design implementation. Ground node is on the left and the remote (quadcopter) node is on the right. Communication is over the ZigBit Wireless modules on each node.

the DAC for sending audio signals to loudspeakers and Atmel’s AT86RF233 ZigBit module [**45**] for communication with the remote node.

The remote node also included the same ZigBit module. In addition, it also had the microphone ADC and an optional set of analog comparators for detecting the propeller motion. Finally, the remote node had a USB host interface for recording audio to the flash drive and a UART interface for logging debug statements to a PC console application.

## 4.2 Operating System

In order to guarantee the completion of tasks by their deadlines, a real-time operating system was employed on both the remote node and the ground node. Each node had a specific set of tasks that needed to be completed and each task had a priority. The tasks for each node are outlined in section 4.2.1. ASF provides an embedded real-time operating system that is a port of the popular and open-source FreeRTOS project [**46**].

### 4.2.1 Real Time Requirements

Each node had separate tasks that needed to be executed periodically. Some events ran periodically on a schedule while others only occurred on certain triggers. The operating system used a prioritized round-robin scheduling algorithm, with 0 being lowest and the user configuring what the highest level was. Table 4-1 and Table 4-2 give descriptions of each task and its priority for the ground node and remote node, respectively. Highest priority tasks always ran before lower priority tasks unless they were suspended or block by an I/O event. Tasks with the same priority ran in a pre-emptive round robin scheme, with the user able to configure how often a context switch occurred. For the purposes of this project, context switches occurred approximately 1000 times per second.

FreeRTOS also provides mutexes and semaphores. Binary semaphores were used primarily in the CLOCK SYNC tasks to keep track of which pulse signals had been sent and received by the two nodes. They were also used to issue audio recording requests from the WPAN task to the AUDIO task on the remote node. In addition, FreeRTOS also has the ability to change task priorities dynamically. This was used primarily in the AUDIO tasks when a status message to the other node needed to be sent. By quickly raising the WPAN task’s priority, it could receive more CPU time than other executing tasks.

| Task | Initial Priority | Description |
|---|---|---|
| **IDLE** | 0 | Automatically begins at startup. Runs when there are no other tasks left to run. Only used to update background operating system values |
| **WPAN** | 1 | Started in system initialization. Performs the sending of messages and any processing necessary for wireless communication |
| **AUDIO** | 1 | Started when a request for an audio recording is made. Sends remote node notice to record, generates audio, and then deletes its own task. |
| **CONSOLE** | 0 | Started in initialization of system. Polls for user input from the UART console and then issues requests or returns status information. |
| **LOW SPEED USB** | 0 | Performs low speed USB interaction. Note: high speed USB is entirely interrupt driven and does not use this task. |
| **CLOCK SYNC** | 0 | Starts on successful wireless connection establishment between two nodes. Synchronizes and then maintains synchronization between the two nodes. Once synchronized, synchronization is checked approximately every five seconds but paused during audio recordings. |

**Table 4-1: Ground Node Tasks**

| Task | Initial Priority | Description |
|---|---|---|
| **IDLE** | 0 | Automatically begins at startup. Runs when there are no other tasks left to run. Only used to update background operating system values |
| **WPAN** | 0 | Started in initialization of system. Performs the sending of messages and any processing necessary for wireless communication. |
| **AUDIO** | 1 | Starts after a wireless connection has been established and then waits (using a semaphore) for a command to start recording. |
| **ADC** | 2 | Kicked off from the AUDIO task after an audio recording request has been received. Performs the recording and then deletes its own task. |
| **CLOCK SYNC** | 0 | Starts on successful wireless connection establishment between two nodes. Synchronizes and then maintains synchronization between the two nodes. Once synchronized, synchronization is checked approximately every five seconds but paused during audio recordings. |

**Table 4-2: Remote Node Tasks**

### 4.2.2 Atmel’s FreeRTOS Port

FreeRTOS is developed as an open source operating system and the developers encourage individuals and companies to make ports of the operating system to different embedded systems and processors. Atmel has done this and provides the operating system through ASF. The most recent version of FreeRTOS available through ASF and used throughout this project was 7.3.0. (At the time of this paper, the latest version of FreeRTOS source available was 8.0.1.)

Atmel’s port of FreeRTOS is similar to what is documented on the FreeRTOS API site [**46**] with a few exceptions: some of the function names have been altered slightly to better match their implementation in ASF and most of the base type and typedef names have been changed. There is significant documentation provided in the ASF FreeRTOS library files in order to determine what ported names are used.

## 4.3 User Interaction Interfaces

In order to control the timing of recordings, a human interface was needed. The first interface developed used a UART connection for debugging and generating a console menu that allowed the user to see the current status of the nodes and to start an audio recording.

The second was a USB interface. On the ground node, the USB interface provided an additional way to communicate with the nodes, primarily intended for larger applications. The remote node used the USB interface to write audio data to a removable USB flash drive.

### 4.3.1 UART Menu Interface

The Atmel SAM4L Xplained Pro includes a debugger port over USB that installs a virtual COM port on the user's PC when plugged in. The SAM4L also provides additional pins for RX and TX UART transmissions if desired, though they were not used here. ASF provides a low level library for communicating over UART. The nodes required a configuration file and then could call the initialization function provided by ASF. Thereafter, any calls made to STDIO functions (such as *printf*) were over this connection.

On the ground node, a menu allowed the user to enter commands for the system. This included requesting the status of the system, which returned whether the two nodes were connected and communicating, if they were synchronized, and what the estimated clock drift was. The menu interface was read using normal STDIO commands in the CONSOLE task.

The remote node also included a UART interface for displaying debug messages. Sending characters to the remote node over UART did not issue any response from the node.

### 4.3.2 USB Interface

USB is capable of operating as either a host or as a device. As a host, it controls the bus, issues commands, and polls the other USB devices for any new information. As a device, it awaits commands from the host and must wait until the host asks for any new information before putting data on the bus. Both modes of operation were used in the project. Host mode was required on the remote node to store audio data on the flash drive and device mode was required on the ground node to interface with the PC.

#### *4.3.2.1 Audio Storage*

Because the SAM4L does not contain enough memory to store more than a few milliseconds of audio, the data needed to be offloaded to an external storage media of some sort. The most useful way to do this was to use a USB flash drive, since flash drives can be read by most computers. The SAM4L also includes a native USB interface that significantly simplifies the process of making a working USB port.

ASF includes a functional USB library for both host and device modes. As the remote node was going to be controlling the USB flash drive, it needed to be the host. Data on a flash drive is often stored in the FAT file system and Atmel makes the process of working with FAT file systems easy by including a port of the FatFS project [**47**]. As seen in Figure 4-5, the FatFS library connects to the USB drive through the ASF USB library.

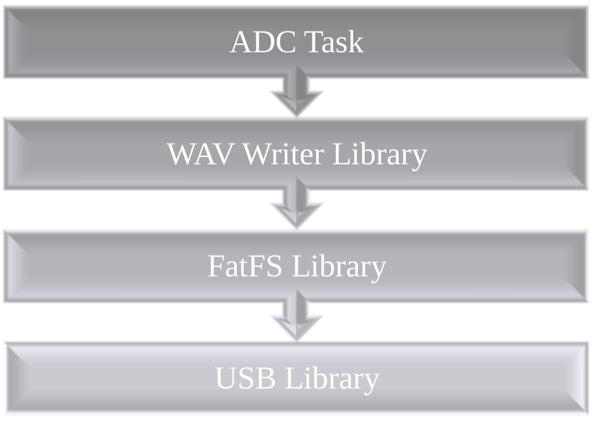


**Figure 4-5: USB WAV File Writing Stack**
To write a WAV file to the USB flash drive, a call from the ADC task is made to the WAV Writer library, which in turn calls to the FatFS library, and that calls to the USB library.

Above FatFS sat the WAV file library. The WAV file library was originally based on the *wavwriter* library written for the CSE202011 class at the University of Notre Dame [**48**]. However, the library was heavily modified in order to work on the SAM4L system and to cover the needs of recording from an ADC (see Appendix A.4.5). The library first created the audio file and pre-cached spaced on the drive in the initial call, *wavfile_open*. The next call, *wavfile_write*, was repeatedly called each time there was data to be written to the file. A final call to *wavfile_close* wrote the final header data required in a WAV file and then closed out the file. Another function called *wavfile_crop* was also written to handle cutting the excess audio data from the beginning and end of audio files.

#### *4.3.2.2 Audio Control Library*

For the ground node, an additional PC library was implemented in order to connect and communicate with the board via USB (see Appendix A.3). This library provided similar functionality as the UART connection, but was intended to be used in conjunction with other libraries for creating larger applications. One example of this would be using it with the AR.Drone SDK library, available at [**23**], to create an application that automatically moved the quadcopter from one position to the next and, once positioned, told the nodes to begin their recording. Upon completion, the embedded USB interface would inform the PC application it could move on to the next location.

The ground node used the same ASF library as the remote node for USB communication. However, the remote node was implemented in USB’s device mode, since the PC would act as the host. This meant that the PC would control the bus and the ground node needed to be ready for any commands that might be issued from the PC.

## 4.4 ADC Audio Recordings

The ADC on board the SAM4L was used to read the microphone values and store them to the USB using the previously mentioned WAV file library functions. In order for this to occur properly, the signal from the microphone needed to be cleaned up and amplified before it reached the ADC. The SAM4L includes seven differential ADC positions. For reference, ADC[2] and ADC[3] pins were used in this implementation for the positive and negative inputs, respectively. During the initialization of the ADC, a further gain was applied to the signal before being quantized. Audio data was stored in buffers and then written to the USB flash drive in larger chunks. Details of each step are in the following sections. (Sample code for reads from the ADC and writes to the USB can be found in Appendix A.4.3)

### 4.4.1 Audio Signal Conditioning

The audio signal voltage range coming from the microphone was very small and in order to be read by the ADC, needed to be amplified. The microphone includes an optional 20 dB gain that was enabled. However, the impedance from the microphone did not match that of the ADC, resulting in large amounts of noise. To compensate for this, a second amplifier was included between the microphone and the ADC. This amplifier only provided a 3 dB gain but matched the impedance of the two circuits much more closely. The full signal conditioning circuit is depicted in Figure 4-6.

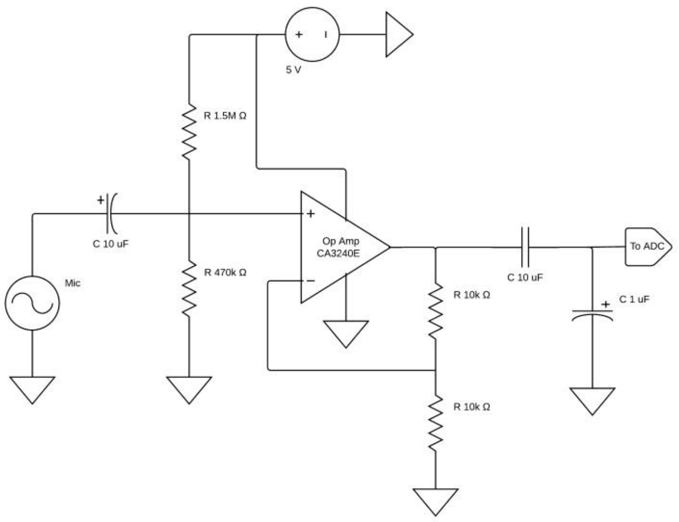


**Figure 4-6: ADC Signal Conditioning Circuit**
(Large scale schematic included in digital attachment)

### 4.4.2 ADC Initialization

The SAM4L includes a 12 bit ADC on-board, which can be set as either a single-ended or differential ADC. It was initialized to be a differential ADC, thus allowing the two pins of the microphone to be plugged into the ADC directly. It could have been simplified to a single-ended ADC with the microphone ground tied to the development board ground. However, this introduced a large amount of noise into the measurements.

The ADC also has a built in signal gain from 0.5 to 64 times the input signal. To get the largest possible voltage swing across the 12 bit ADC, the microphone signal was given a gain of 16. ADC measurements can be activated in multiple ways. The most efficient way is to set a counter in the ADC that automatically counts down to zero. Once it does, the ADC performs a measurement and resets the counter to its initial position. The reading from the ADC was passed to the peripheral DMA controller (PDCA).

### 4.4.3 PDCA Transfers

Rather than interrupting the CPU each time a measurement was performed, control of the ADC was passed to the PDCA. This module allows peripherals, such as the ADC, to directly access memory without the intervention of the processor. This freed up the processor to work on other tasks, such as writing data to the USB.

The ADC required two PDCA channels, TX and RX. The TX channel included data about the operation of the ADC, including which pins to use and what gain to use. It was set up as a ring buffer so that the same data was given to the ADC each time it requested it. The RX channel was where the ADC wrote out the measured audio data. A large buffer was given to the RX channel and only after the ADC had filled up the buffer was an interrupt called to the processor.

Once the processor received this interrupt, it assigned a new buffer to the RX channel. The previously written buffer was then flagged as full for the main loop to write to the USB. Three buffers were rotated through in this way, in an attempt to mitigate the case where the main loop was slow in writing one of the buffers to the USB. If all three buffers were ever marked as full, an overload counter was incremented. Ideally, this overload counter would be zero by the end of the recording process, indicating that no audio data was lost.

### 4.4.4 File Writing Loop

After the ADC and PDCA had begun, the main audio loop was started. The primary purpose of this loop was to write audio data to the USB. When a buffer was flagged as full, it would be written to the USB in the next iteration of the main loop, using the *wavfile_write* function. By writing data in large chunks, the time spent accessing the USB was reduced as less overhead was required to start the transfers as compared to many small transfers.

The raw data coming in from the ADC could not be written directly to a WAV file. The ADC uses the quantization formula:

$$output\ decimal\ code = 2047 + \left(2^n * \frac{V_{in}}{V_{ref}}\right) * 2047,$$

where $n$ is the gain, $V_{in}$ is the voltage coming into the ADC (may be negative), and $V_{ref}$ is the reference voltage. WAV files using 16 bit data however are signed integers [**49**]. This meant that the 12 bit value needed to be shifted four bits left (which could be implemented directly on the ADC) and the uppermost bit inverted.

To compensate for a ground node that may start audio playback early, the recordings also started slightly before the time that the ground node stated the recording should be started. The recording also continued for slightly longer than the required duration in case playback started late and so all of the important data was pulled out from the PDCA buffer before ending. The recording could then be cut down to the right size once the ground node notified the remote node of when the audio playback had actually started. Further details of this process are elaborated in section 4.6.3.2.

## 4.5 DAC Audio Playback

Recording audio was only half of the work required for making audio measurements. The remote node needed a sine sweep signal to record and this task was carried out by the ground node, as decided by the design in section 4.1.3. The SAM4L includes a specialized Audio Bit Stream DAC (ABDACB) that attempts to output sound at a higher quality than using a simpler DAC. (Sample code for writing to the ABDACB can be found in Appendix A.4.4.)

### 4.5.1 Atmel ABDACB Interface

The Atmel SAM4L's ABDACB is a 16 bit digital stereo DAC. Input data can actually be in multiple sizes (8 bit, 16 bit, or 32 bit) in two's complement form. The ABDACB converts it to 16 bit data before beginning the conversion. During the conversion, digital data is fed into a CIC Interpolation Filter and then a Sigma Delta Modulator. Because the CIC Interpolation Filter creates a large amount of high frequency noise and because the ABDACB outputs digital square waves, Atmel recommends running this through a low pass filter before passing it on to a loudspeaker.

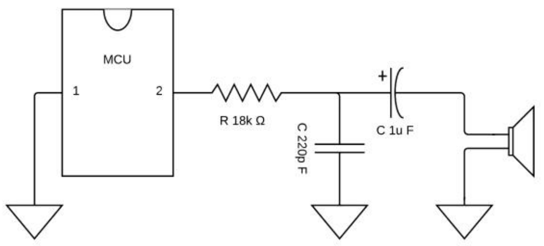


**Figure 4-7: ABDACB Output Low Pass Filter** The ABDACB has one output (pin 2 above) that goes into the low pass filter. The low pass filter also needs to be connected to the board's ground (pin 1 above). (Large scale schematic included in digital attachment)

This ABDACB is capable of using a sampling frequency between 8 kHz and 48 kHz. As it is a stereo DAC, it outputs on two user-defined channels simultaneously. It also can output the inverse signals on separate channels for those applications that require such a signal. The ABDACB has two input registers, one for each channel, though this can be reduced to one channel if the output format is mono instead of stereo. The ABDACB can be fed data directly, however, it is also connected to the PDCA for DMA operations, freeing up the processor for other calculations.

The output from the remote node only needed to be a mono signal as there is only one sine sweep signal playing at any time. The PDCA was also utilized so as to give the processor more time for calculating the next samples of the sine swept signal. As suggested by Atmel, the final output signal was also run through a low pass filter. The low pass filter design is shown in Figure 4-7.

### 4.5.2 Signal Calculation Requirements

As the SAM4L has limited memory space, the sine swept signal could not be generated prior to beginning a recording. Generating the signal at compile time and placing it in ROM was also not possible because of space requirements. Storing the audio on USB or other external media was not viable since the USB interface was already being used for the PC-USB interface and other external media would have required additional hardware. Additionally, pre-generated audio likely would have restricted the design to only pre-generated sine swept signals and not allowed changes to sine sweep frequencies or duration at run-time.

This left only the possibility of generating the sine swept signal in real-time. Since the output DAC was running at 44.1 kHz, each signal needed to be generated in less than 22.7 microseconds or 272 cycles of the 12 MHz microprocessor. This included time taken by other tasks and processes running simultaneously on the processor, receiving the notice from the DAC that more samples were needed, and sending new samples to the DAC. Utilizing the PDCA reduced the number of times the DAC interrupted the processor and allowed the processor to send a large block of samples to the DAC instead of sample by sample. The audio generation task was also given exclusive use of the processor to guarantee that no other processes would steal vital processing time.

Despite these improvements, initial tests quickly demonstrated that using the formula in section 2.4.2 was too time consuming. Though the *ln* function value could be pre-calculated, the *e* and *sin* functions needed to be calculated on a sample by sample basis and this proved to be difficult for the real-time OS. An alternate method of calculating the signal was needed.

## 4.5.3 Iteratively Calculated Swept Sine Signal

The alternate method proved to be an iteratively calculated sine swept signal. Based on experimental work from Matthew van Erde posted at [**50**], it was possible to calculate the next sample based on the phase and frequency of the previous sample. The frequency changed by a constant multiplier for each sample:

$$F_n = F_{n-1} * \Delta_f,$$

where $F_n$ is the current sample frequency, $F_{n-1}$ is the previous sample frequency, and $\Delta_f$ is the frequency multiplier. The multiplier was calculated using:

$$\Delta_f = \frac{f_{end}^{\frac{1}{F_S * D}}}{f_{start}^{\frac{1}{F_S * D}}},$$

where $f_{end}$ and $f_{start}$ are the ending and starting frequencies, respectively; $F_S$ is the sampling frequency; and, $D$ is the duration of the sine sweep. The multiplier does not change over the span of the sine sweep signal and therefore could be pre-calculated before starting the sine sweep signal generation process.

The phase of the new sample, $\theta_n$, could be calculated by adding the previous phase, $\theta_{n-1}$, to the new frequency times the inverse of the sampling frequency:

$$\theta_n = \theta_{n-1} + F_n * \frac{1}{F_S}.$$

The final step was to take the *sin* of the new phase, which gave the desired output decimal value:

$$DAC_{OUT} = \sin(\theta_n).$$

The only additional required step was to change the decimal value to the two’s complement format required by the ABDACB.

As a further speed increase, several specialized functions were used. The ARM processor in the SAM4L includes special DSP functions, including an optimized *sin* function for values in Q15 format and a second function for converting decimal values to Q15 format. As the resulting value out of the optimized ARM *sin* function was a two’s complement integer value already, it could be fed directly to the ABDACB with minimal additional processing.

## 4.6 Wireless Communication Protocols

The previous prototype in section 3 used Wi-Fi for the wireless communication. While Wi-Fi is useful for its ubiquitous nature and flexibility, it also can be a very large stack that is hard to include on smaller designs and is difficult to modify. For the final design implementation, the wireless communication was switched to the ZigBee standard.

Using ZigBee meant that a smaller stack needed to fit on the SAM4L. Since the ZigBee stack is also less complex than Wi-Fi, it was easier to alter for the purposes of synchronization. Atmel has developed several ZigBee wireless solutions, which they refer to as “ZigBit” modules. ASF also includes several libraries for interfacing with the modules, reducing the amount of time required to get wireless communication functioning.

### 4.6.1 Atmel’s Zigbit Wireless Protocol

The final design implementation used a pair of AT86RF233 Amplified ZigBit Xplained Pro Extensions. These modules connect directly to EXT1 on the SAM4L Xplained Pro and require no further hardware configuration. ASF includes multiple communication stacks and for this design, the IEEE 802.15.4 MAC stack was chosen. This stack provides a basic and generic data transport API for controlling communication across the network.

Use of the IEEE 802.15.4 MAC stack abstracted out the lower layers of the communication protocol, allowing the application to make and receive function calls to and from the lower levels. The user may define with which specific layer they would like to interact. This stack can be used with both beacon and non-beacon networks. Using the highest level abstraction layer, calls to *wpan_init* and *wpan_task* initialize the WPAN network and then perform routine tasks regarding sending and receiving messages on the WPAN network.

### 4.6.2 Zigbit Implementation

The final design utilized the IEEE 802.15.4 MAC stack at the highest abstraction, allowing the lower level libraries to take care of the most critical functions. The API makes calls back to the user application during the setup of the network. These calls allow the user to define if the node will be setting up the network or attaching to a pre-existing network, as well as to define if a beacon or non-beacon network should be used. Once established, the API makes calls to specific application functions when a message has been successfully sent or received.

A single generic user application library was developed that was shared between both the remote node and ground node (see Appendix A.4.2). Figure 4-8 shows the full wireless stack used on both nodes. The remote node was set to create the network and then the ground node searched for this network until it could successfully connect to the remote node. The user application automatically made calls to the upper level application for properly sent and received messages and the upper level application can make a call to the library for sending a message.

Messages across the WPAN network followed a general structure. The first data byte identified the type of message that was being sent. Table 4-3 shows the types of messages that were expected to be traversing the network. Data after the first byte depended on the type of message being sent and the node that was sending.

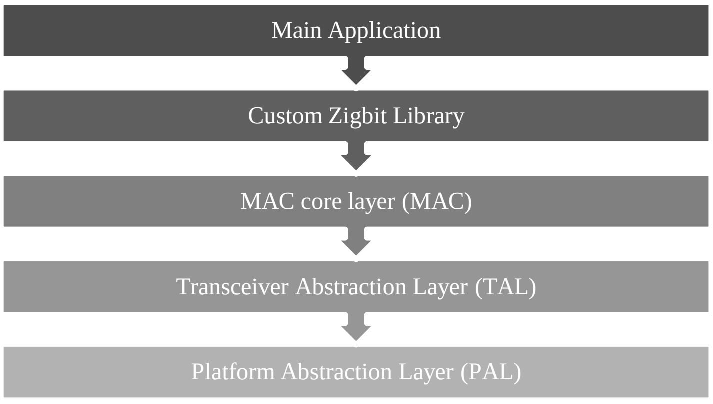

Main Application
Custom Zigbit Library
MAC core layer (MAC)
Transceiver Abstraction Layer (TAL)
Platform Abstraction Layer (PAL)

**Figure 4-8: Atmel Wireless Communication Stack**
Atmel provides the PAL, TAL, and MAC layers of the IEEE 802.15.4 MAC implementation in ASF. Another custom abstraction layer was built on top of that, and then the main application code interfaced with that custom library. (Based on Figure 2-2 of [**56**])

| Byte Value | Message Type | Description |
|---|---|---|
| **0** | UNKNOWN | A message that does not match any of the other types. |
| **1** | SYNC_PULSE | First pulse of the TPSN algorithm. |
| **2** | ACK_PULSE | Second pulse of the TPSN algorithm. |
| **3** | TIMESTAMP_PULSE | Includes the timestamps of T2 and T3 of the TPSN algorithm. |
| **4** | CONFIRM_SYNC | Message confirming the synchronization of the two nodes. |
| **5** | START_RECORDING | Request to start recording on the nodes. |
| **6** | RECORDING_STATUS | Contains information about a recording that just finished. |
| **7** | ALTER_WAV_FILE | Request to alter a WAV file that has been previously recorded. |

**Table 4-3: Wireless Network Message Types**

## 4.6.3 Synchronization

As per the requirements outlined in section 2.5, synchronization between the remote node and ground node was very important. Synchronization between the two nodes was implemented using the Timing Sync Protocol for Sensor Networks (TPSN). However, because of the way that the time-stamping occurred in ASF's IEEE 802.15.4 MAC API, the TPSN implementation needed to be altered slightly.

To further increase the synchronization of audio between the two nodes, a couple of additional steps were taken. Prior to recording, the start and end times of the recordings were altered slightly. After the recording, the audio was trimmed to the proper start and end times.

### *4.6.3.1 TPSN Implementation*

The IEEE 802.15.4 MAC API has the ability to timestamp successful data transmission and data reception events at the MAC level, as is recommended by the original TPSN implementation [**37**]. However, in the ASF library, the timestamp is not available until messages have been successfully sent and not before. This means that time T3 from the original TPSN implementation (see section 2.3.2) is not generated until after the message has been successfully acknowledged by the receiving node.

Because of this time-stamping issue, the TPSN implementation used in the final design is a bit more complex, as seen in Figure 4-9. (See Appendix A.4.1 for sample synchronization code.) A second message was sent from node B after the first had been received successfully. This second message contained the timestamps T2 and T3, while the first message from node B contained no additional data in it. The initial message from node A to node B also deviated from the original TPSN model, as it did not contain time T1, since

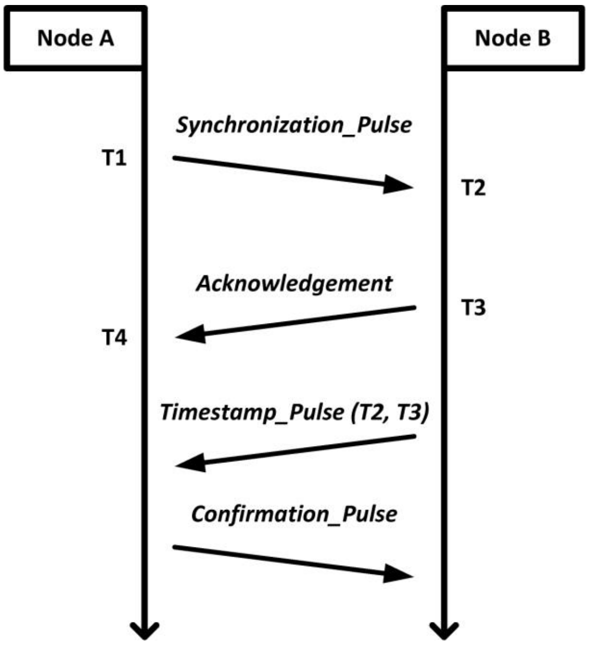


**Figure 4-9: TPSN Implementation Timing**
Node A sends a synchronization pulse at time T1 that is received at time T2 by Node B. Node B sends an acknowledgement back at time T3 which is received by Node A at time T4. The T2 and T3 data is then transferred in the timestamp pulse. Final confirmation of the synchronization is given in the last pulse.

node B does not need to know about time T1. Finally, a fourth message was sent from node A to node B on the successful completion of the TPSN synchronization as a way for node B to know that it had been synchronized with node A.

An addition was also made to the original TPSN in order to handle the common case of clock frequency differences. A difference in the clock frequencies is noticed in a change in the Δ between each synchronization process. To handle this, a separate $\Delta_{change}$ was calculated as:

$$\Delta_{change} = \frac{\Delta_n - \Delta_{n-1}}{T_{4_n} - T_{4_{n-1}}},$$

where $\Delta_n$ and $\Delta_{n-1}$ are the current and previous delta values, and $T_{4_n}$ and $T_{4_{n-1}}$ are the current and previous $T_4$ times, respectively. This value could then be used to calculate how much the $\Delta$ has changed since the last synchronization by multiplying $\Delta_{change}$ by the amount of time that has passed since the most recent $T_4$.

Since $\Delta_{change}$ relied on two synchronization events, a method was needed to keep track of when node A (ground node) was fully synchronized with node B (remote node). The final implementation considered the two nodes synchronized when the following statement was evaluated true twice in a row on synchronization attempts:

$$|\Delta_{n-1} - \Delta_n| < \left|\Delta_{change} * (T_{4_n} - T_{4_{n-1}})\right| * 1.5.$$

By adding in the 1.5 multiplicand, a small amount of leniency was given for clocks that may vary over time. Continually evaluating this formula as each synchronization event occurred allowed the implementation to catch any badly transmitted timestamps. Whenever the system was considered out of synchronization, a new synchronization event was requested within 1.5 seconds. Once the system was synchronized, synchronization events were reduced to every five seconds so as to allow the processor time to perform other tasks.

#### *4.6.3.2 Recording Synchronization*

As a way to prevent missed start times on playback and recording because of wireless miscommunications, the ground node issued a request to start recording with an absolute time on the remote node approximately one half second in the future. This allowed the ground node to have multiple chances to send the message to the remote node and then time for both the ground and remote nodes to do preprocessing work prior to the actual start time. Once the recording started, the actual start time of the first sample was recorded on both nodes for future comparison.

Even with this feature, the actual start times of both nodes had the possibility of fluctuating slightly. To compensate for this, the recording start time on the remote node was adjusted to be slightly earlier. This way, if the playback occurred early or if the recording started slightly late, the audio playback start would still be covered. A similar method was used on the end time of the recording: it was pushed back slightly so that if the recording ended early, it would still cover the entire recording time desired.

The excess audio at the start and end of recordings was removed after the recording had been finished. The recording task would pause on the remote node while it waited for information from the ground node that said the exact time that the playback actually started. Once this information was received by the remote node and the task was resumed, the start times of the recording and the playback were compared and the appropriate number of samples was removed from the beginning of the audio. The end of the audio could then be interpreted as the sampling rate multiplied by the requested duration and any samples after this point could be removed in the final piece of the audio recording.

## 4.7 Propeller Position Measurements

As an optional addition, the propeller position could also be tracked. Since a significant amount of noise was generated from the quadcopter, it is quite possible that some of this noise was generated in such a way that it produced tonal sounds that could be attributed to motor position and/or speed. Similar occurrences appearoften in commercial propeller planes and helicopters and they require active noise cancellation that depends on propeller speed and position [**51**].

While the final submitted implementation did not include any formal inclusion of measuring propeller positioning, a method of measuring propeller positioning was tested part way through. This method is included here for completeness and for future potential use.

### 4.7.1 Hardware Design

To determine the position of the propeller, a photo interrupter was used, similar to that pictured in Figure 4-10. A photo interrupter consists of an infrared emitter on one side and an infrared detector on the other. When the infrared beam is broken by an object passing through the photo interrupter, the output of the infrared detector will change. This worked significantly well on the black propellers of the AR.Drone, which block almost all infrared light from reaching the detector.

The photo interrupter could easily be mounted to the foam body of the AR.Drone. There is approximately 1.3 cm of clearance between the edge of the propeller and the main housing of the drone, enough for the photo interrupter to be mounted without interference. The propeller stands only about 3 mm tall, however, during flight a much larger amount of deflection in the wing was observed, so the opening of the photo interrupter needs to be at least several centimeters.

Since the output of the infrared switched between 0 VDC and the source voltage without much variance, no intermediate signal processing was necessary as was required for the microphone. The output could be wired directly to the ADC or to the analog comparator pins on the SAM4L Xplained Pro.

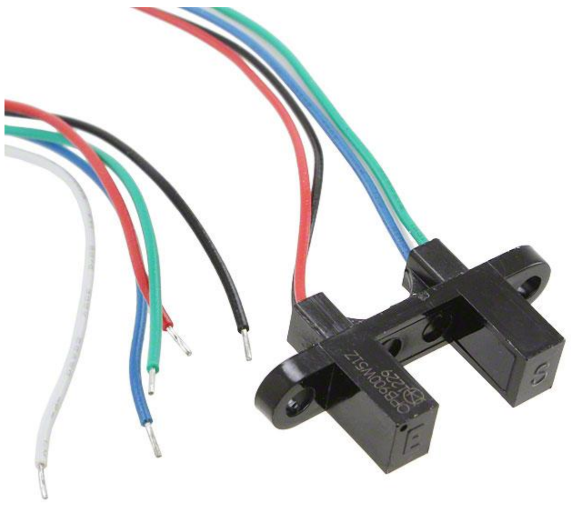


**Figure 4-10: Photo Interrupter**
Photo from [**57**]

### 4.7.2 Software Implementation

Once the signal was wired to the SAM4L, it could be detected by either the ADC or the analog comparator. The ADC would provide fine grain readings of the interference while the analog comparator would simply state whether the propeller was blocking the beam or not. All that was really required was the analog comparator. However, since the ADC was already being used by the microphone, the ADC could be used for this as well with less work.

The data from the ADC or analog comparator could easily be pushed on to the USB just as the audio data was. An additional WAV file could be generated for easy synchronization with the audio, or a simple log file of timestamps matching each time passing of the propeller could also be used.

A photo interrupter would be required for each propeller and each would need their own pins on the SAM4L, which the SAM4L Xplained Pro could accommodate. A WAV or log file for each propeller would then need to be produced so as to distinguish the timing of each propeller individually, since the propellers would each be rotating at their own variable speeds over time.

# 5 Design Evaluation

A design is only worthwhile if it works in reality as well as it was theorized to work. In this section, different measurements taken from the iPod Touch implementation and the SAM4L implementation are presented. After the data is presented, an analysis of the data is made and, as requested by the problem description, possible improvements to the design are given.

## 5.1 Measurement overview

Three primary sources of measurement error were hypothesized and tested. These sources were:

- quadcopter movement,
- quadcopter noise, and
- node synchronization.

Each source was predicted to have a different effect on the frequency response of the measurement.

The effect of quadcopter movement is discussed in section 5.2. For these measurements, the Apple iPod Touch was used to record sine swept signals that were then processed and compared in MATLAB. Quadcopter noise was also measured using the Apple iPod Touch in section 5.3. In these measurements, only the sound of the quadcopter was recorded with all other noises in the room kept to a minimum. Section 5.4 compares the audio synchronization between Wi-Fi and ZigBee and between the Apple iPod Touch and the Atmel SAM4L Xplained Pro implementation. Finally, section 5.5 looks at the audio quality produced with the final design.

## 5.2 Effect of Quadcopter Stability and Movement

As described in the background, quadcopters are inherently unstable because of their design. This instability allows them to do complex movements, but it also can be a potential source of error. During impulse response measurements, the microphone should be held as still as possible so that there is no question about when pressure waves reach the microphone. As there are continual movements of the quadcopter, the variability may affect the impulse response.

Two possible errors are hypothesized to occur due to the quadcopter's instability. The first is interference with high frequency noise. A 10 kHz frequency sound has a wavelength of approximately 3.4 cm at 20° C through air. If a quadcopter is constantly moving by even a few centimeters, it is believed that this could potentially cause interference. The second possible error resulting from quadcopter movement could come from indirect sounds. Indirect noises must travel a greater distance and therefore take longer to reach the quadcopter. If the quadcopter drifts during this time, the indirect sound will arrive at the quadcopter at a different time than if the quadcopter had remained motionless.

### 5.2.1 Measurement Types

To test these possible sources of error, several measurements using the iPod Touch were proposed and carried out. The iPod Touch was located approximately 20 cm away from the loudspeaker that was playing a sine sweep signal. After the loudspeakers finished playing, the iPod Touch would continue to record for slightly longer to capture any indirect reverberations that may still be approaching the device.

In order to test how the quadcopter motion may affect the impulse response, five variations of the iPod Touch movement test were performed. The movements included:

1) Stable recording (no microphone movement)
2) Hand-held steady (slight microphone movement)
3) Small and slow movements (microphone moves only a few centimeters over a period of several seconds)
4) Small and fast movements (microphone moves back and forth only a few centimeters over a period of less than a second)
5) Large and slow movements (microphone moves 10 to 20 centimeters over a period of several seconds)

Movements 2-5 were intended to simulate different ways in which quadcopters may move over time. Movement 1 was used as a benchmark comparison point, simulating the microphone connected to a stand and the measurement performed in the conventional manner without the quadcopter.

### 5.2.2 Impulse Response Results

The signal sweep playback took place over a period of 10 seconds and ranged in frequency from 500 Hz to 10 kHz, produced using the HTML method described in section 3.3.1.2. The recording occurred on the iPod Touch and lasted for approximately 15 seconds (10 seconds of direct sound plus 5 seconds for indirect reverberation).

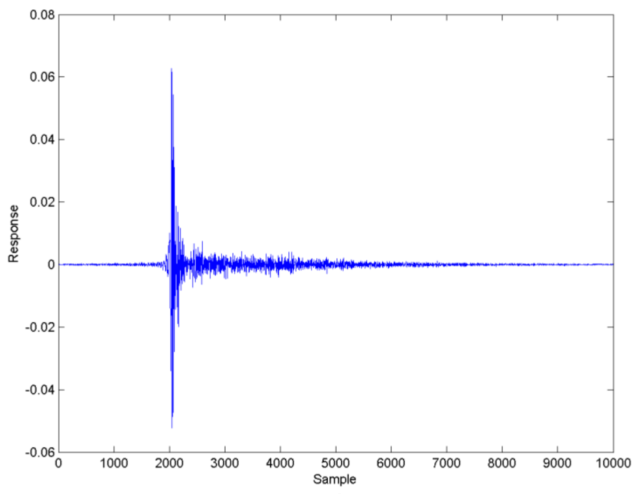
The impulse response was calculated using MATLAB (see Appendix A.1.2). An FFT of the microphone recording was divided by the FFT of the ideal signal sent to the loudspeaker. Thereafter, the real values from the IFFT were used in the final impulse response, which were also windowed to 10,000 samples so as to provide consistent comparisons for the frequency response across all measurements. To obtain the frequency response, a second FFT was taken of the impulse response.

Multiple measurements were taken from each of the five movements in order to test consistency between measurements. A sample of the impulse response from each of the movements is shown in Figure 5-1, A-E. The resulting frequency response of each impulse response is then shown in Figure 5-2, A-E. Finally, a few frequency response comparisons between measurements of the same movements and of different movements are shown in Figure 5-3, A-F.

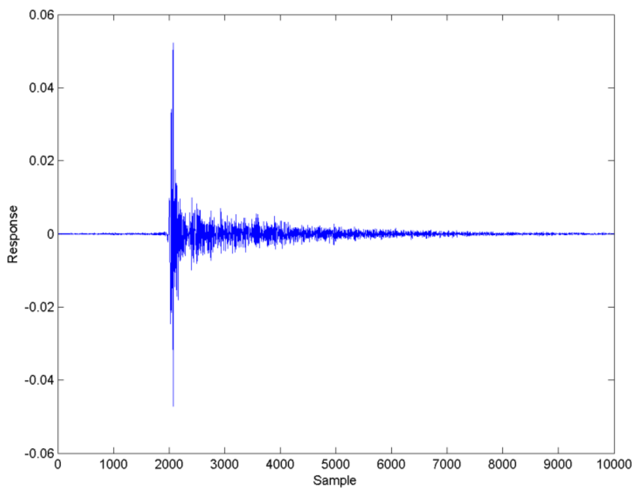
### 5.2.3 Frequency Response Analysis

Looking at the impulse responses from Figure 5-1, it is possible to see that with more movement came more noise. While the stable impulse response of Figure 5-1A was close to the ideal response, the large and slow impulse response had quite a bit more noise surrounding the impulse response in Figure 5-1E.

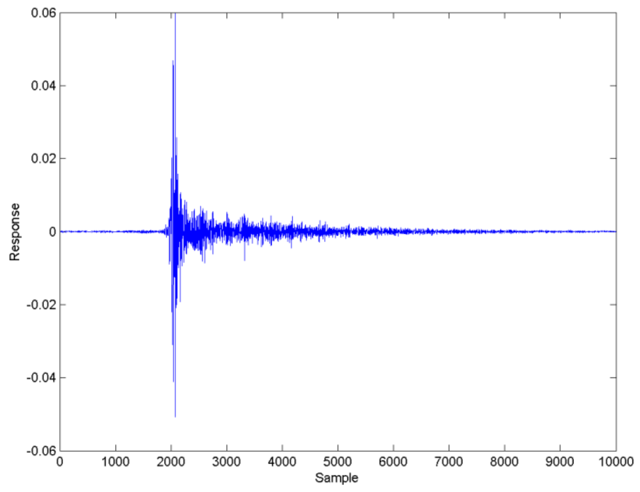
The frequency response of the measurements was fairly similar across most of the measurements. As could be expected, the largest distortions came with the largest movements. Consistency between



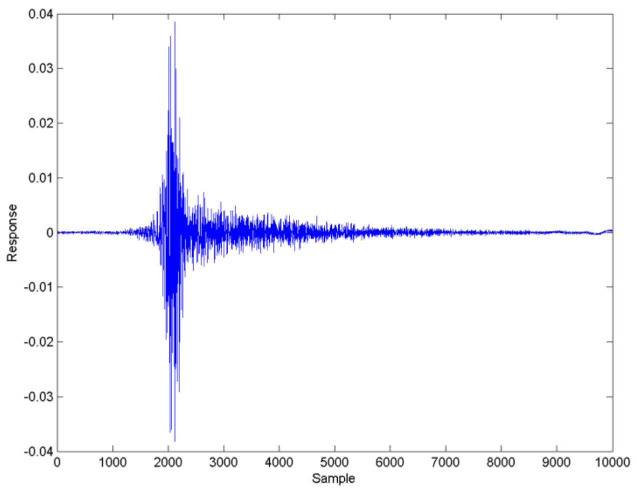
A – Stable



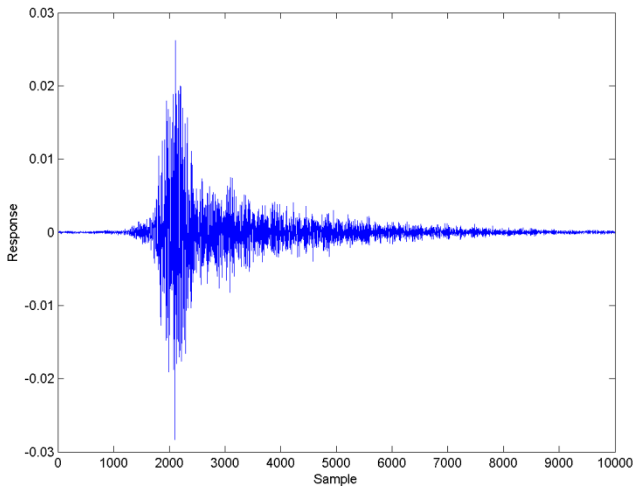
B – Steady hand movement



C – Small and slow movement

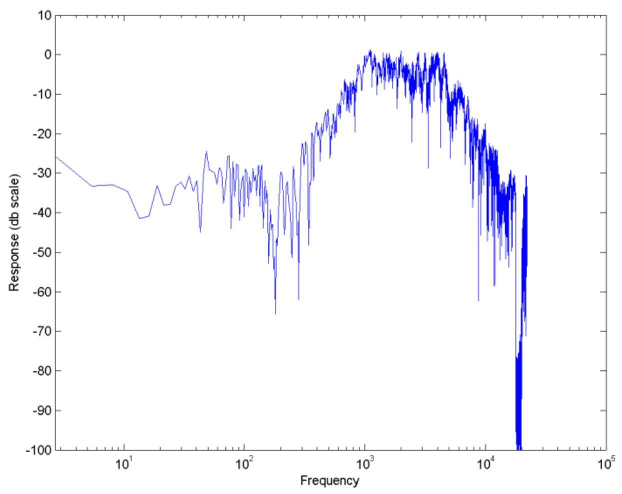


D – Small and fast movement

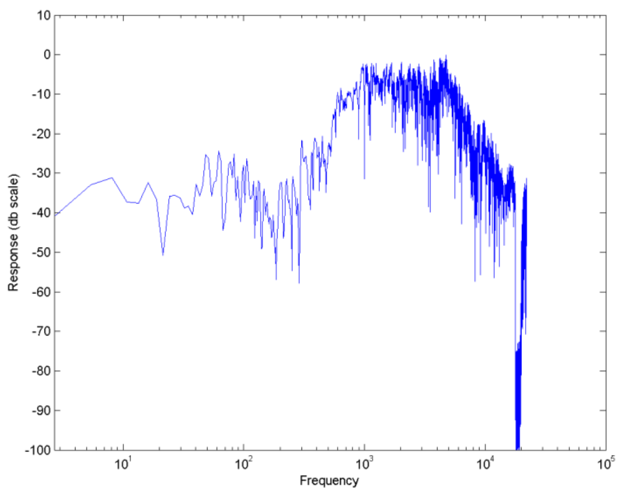


E – Large and slow movement

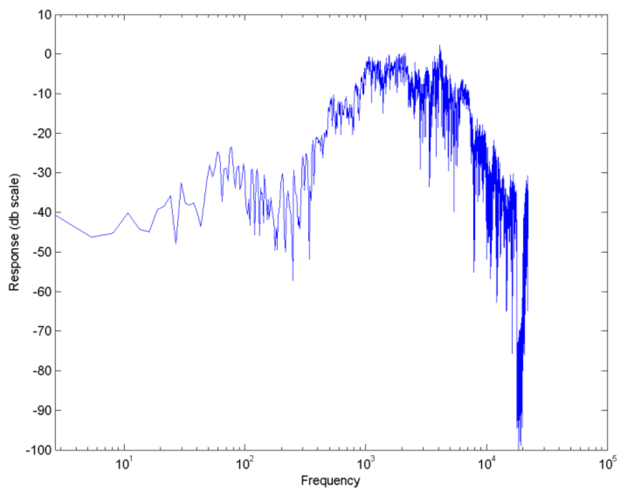
**Figure 5-1, A-E: Quadcopter Movement Impulse Response Measurements**



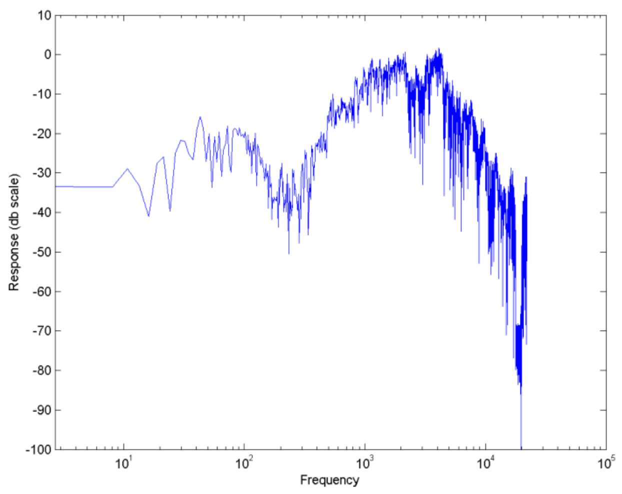
A – Stable



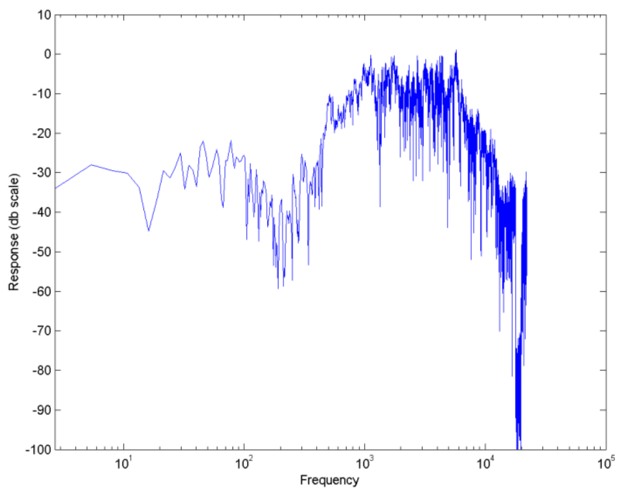
B – Steady hand movement



C – Small and slow movement

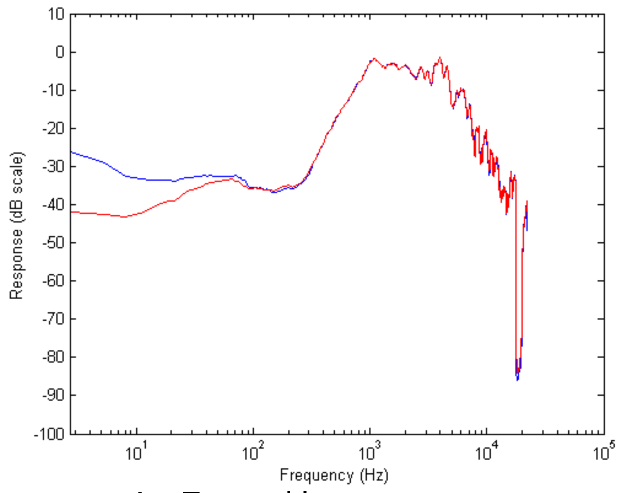


D – Small and fast movement

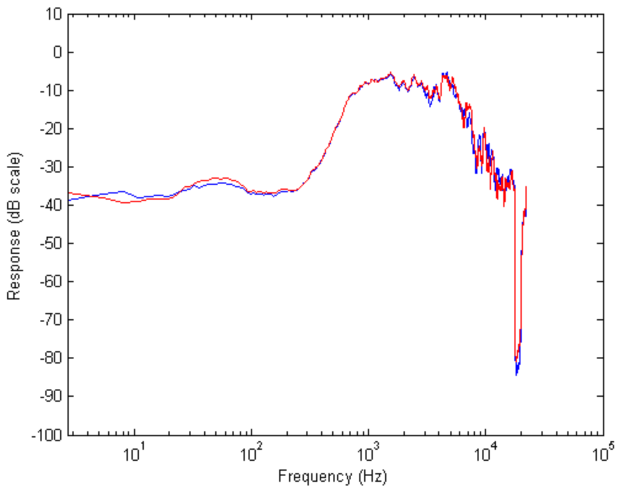


E – Large and slow movement

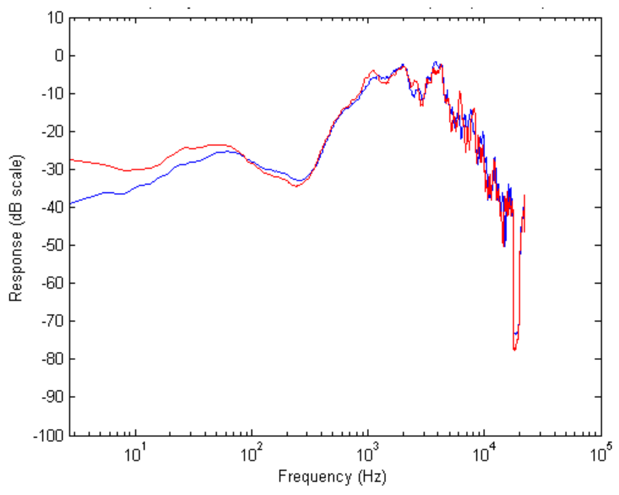
**Figure 5-2, A-E: Quadcopter Movement Frequency Response Measurements**



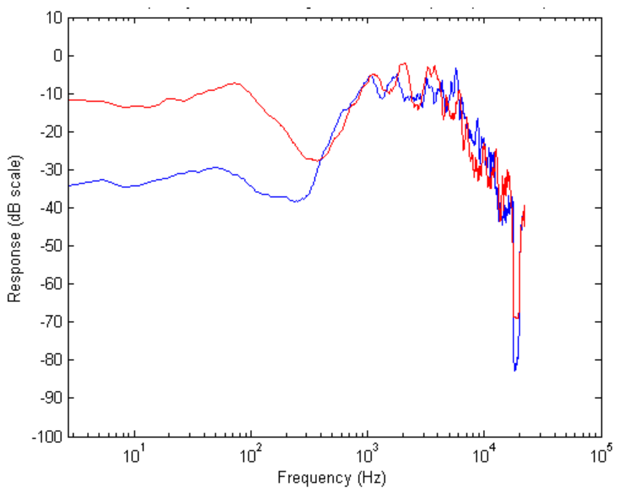
A – Two stable movements
(Smoothed for clarity)



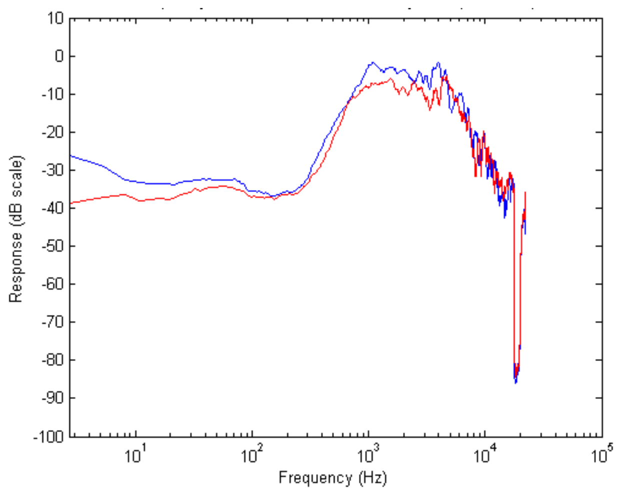
B – Two steady hand movements
(Smoothed for clarity)



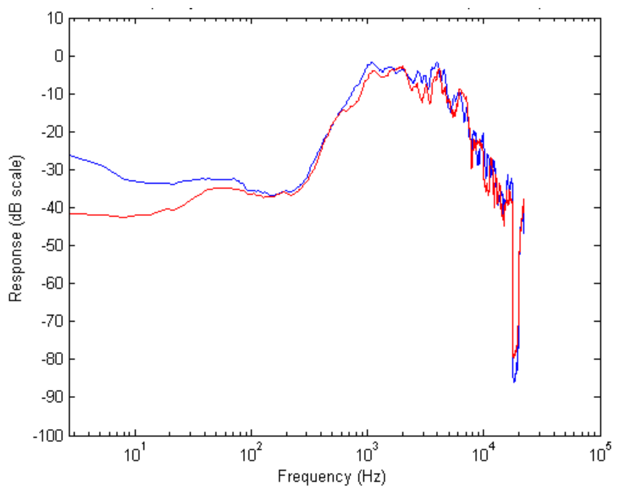
C – Two small and fast movements
(Smoothed for clarity)



D – Two large and slow movements
(Smoothed for clarity)



E – Stable (blue) vs. steady hand (red) movements
(Smoothed for clarity)



F-Stable (blue) vs. small and slow (red) movements
(Smoothed for clarity)

**Figure 5-3, A-F: Quadcopter Movement Frequency Response Comparison**

measurements also was fairly good for small movements. In the ideal situation, measurements on the quadcopter would be similar to the baseline measurement consistency, as shown in Figure 5-3A, where the two graphs are nearly identical. The steady hand measurements showed slight deviations between measurements as seen in Figure 5-3B. However, with larger movements, the consistency began to decrease, as seen in the large and slow movements of Figure 5-3D.

When comparisons between movement types were made, the differences became a bit more apparent. Figure 5-3E shows a comparison between the stable measurement and the steady hand movement and there is a bit of variation between the two, even in the lower frequencies. Interestingly, the stable and small and slow movements of Figure 5-3F looked to be a bit more consistent with each other than the previous graph did, even in the higher frequencies.

The results demonstrated that a steady microphone is needed on board the quadcopter in order to have consistent results. Larger movements result in increased variations in the impulse response. While it was hypothesized that it would be the higher frequencies that would be most affected by the movements, the results demonstrated that frequencies across the entire spectrum were affected by the movements.

### 5.2.4 Possibilities for Movement Reduction

There are several possibilities for reducing movement from the quadcopter. The first option that should always be taken is to choose a quadcopter that is as stable as possible. Multiple parts of the quadcopter can affect the stability. The configuration and size of the quadcopter can result in more or less stable systems, as can the choice of components used on the quadcopter. Bigger or smaller propellers, faster processors, and faster and more accurate sensors all can add to the stability.

Just as important are the software algorithms that are used to increase stability. If the processor can perform smaller and more frequent updates to the propeller speeds, the quadcopter can operate more smoothly. Smarter algorithms can take into account more variables that affect stability as well. Research into quadcopter stability is a large field and multiple institutes are performing experiments and releasing updates in this area.

It is unlikely in the near future that any quadcopter will be natively stable enough to have small enough movement for a microphone performing measurements. An additional method of reducing microphone movement is to introduce a gimbal type system for mounting the microphone. Such systems are often used when performing professional video recordings on-board quadcopters. Video recordings attached directly to a quadcopter show large amounts of vibration and jitter. By using a gimbal system, these small movements are removed during flight.

The gimbal system used may not need to be complex. Attaching the microphone to elastic or rubber below the quadcopter may be enough to dampen many of the higher frequency movements. Using a higher quality or commercial system would provide protection from larger or slower motions. Many commercial gimbal systems remove excess motion in all three axes, allowing the quadcopter to move suddenly in any direction without the microphone moving.

## 5.3 Effects of Quadcopter Noise

The noise from a quadcopter is perhaps the most prevalent notion of introducing error into audio measurements. When a quadcopter is turned on, it is quite easy to hear how that is possible as quadcopters are very noisy. However, removing this noise is only one part of the challenge of recording from a quadcopter.

In this paper, the topic of noise cancellation is only covered modestly since noise cancellation is part of a much bigger area of study and deserves quite a bit of research to be covered properly. The measurements here only begin to provide insight into the possibility of removing noise in a variety of ways. Most of the ideas tested here come from the evolving principles of noise cancellation in helicopters and propeller airplanes.

It is hoped that the noise coming from a quadcopter is primarily tonal in nature. That is, the noise that the quadcopter produces comes from specific parts of the quadcopter (e.g. motor or gears) and that it is restricted to a very narrow frequency band. The measurements performed test this idea by recording the quadcopter's noise and analyzing the resulting frequency spectrum.

### 5.3.1 Noise Measurement Types and Locations

The measurements performed in this section were similar to the previous section's measurements. Measurements were once again carried out using the iPod Touch. However, this time the position of the microphone was varied to see if there was a position that gave the greatest advantage for noise reduction. The microphone positions included were:

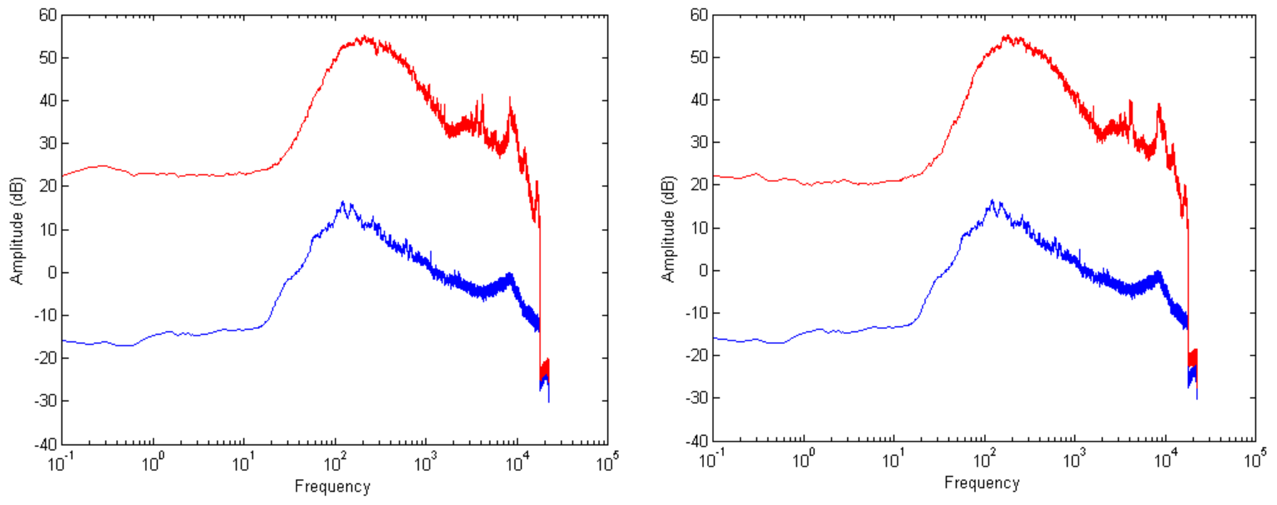
1) On ground, facing upwards toward quadcopter (approximately 80 cm away)
2) Below quadcopter, facing away from quadcopter (approximately 20 cm away)
3) Above quadcopter, facing towards quadcopter (approximately 20 cm away)
4) Above quadcopter, facing away from quadcopter (approximately 20 cm away)
5) Side of quadcopter, facing towards quadcopter (approximately 20 cm away)

Unlike the last measurements, no sine sweep signal was played during recordings as only the noise of the quadcopter was desired. Multiple measurements were taken at each location for comparison. As a benchmark, a recording of the room without the noise of the quadcopter was also made.

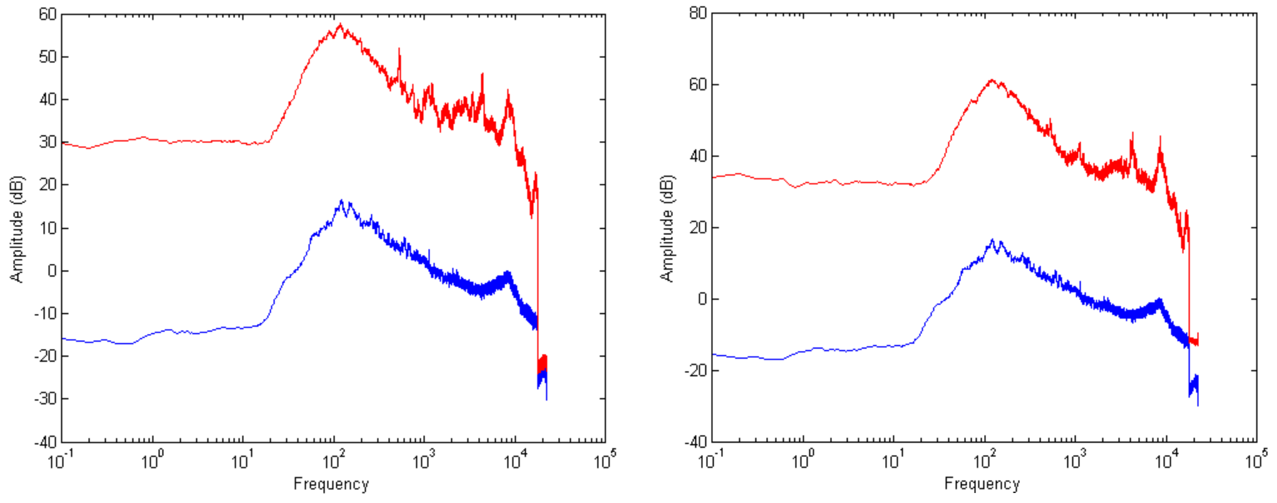
### 5.3.2 Impulse Response Results

Measurements were made with the iPod Touch using the HTML interface for starting and stopping recordings. Recordings were made with minimal background noise. Recordings lasted between 12 and 15 seconds, and then were cut down to the most consistent 10 seconds afterwards (i.e. 10 seconds with the least drone movement and least propeller speed changes).

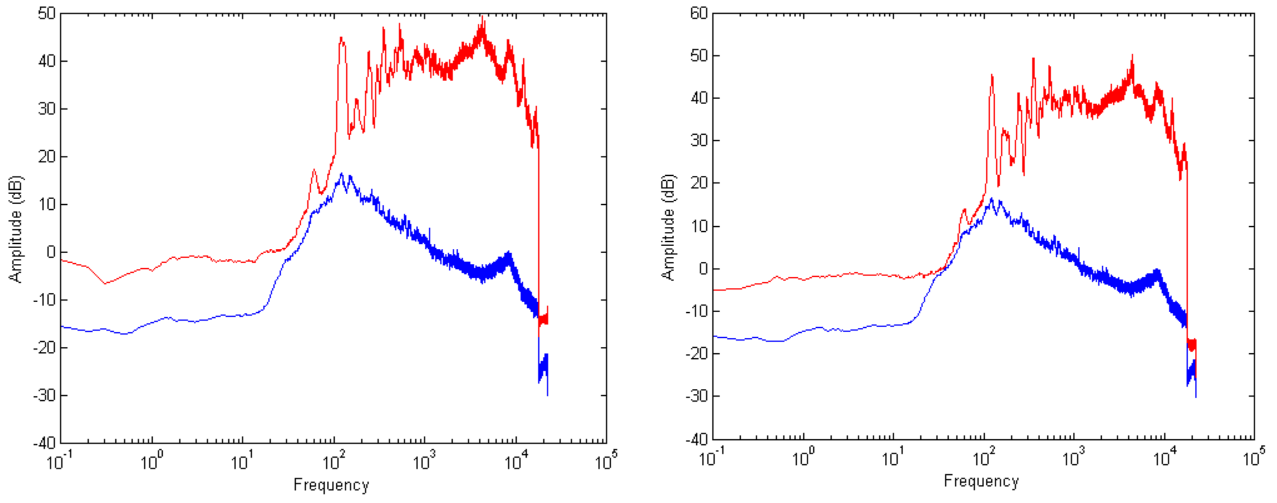
Two frequency responses are included from each position in Figure 5-4. Frequency response calculations were performed in MATLAB by taking the FFT of the recorded signal and then plotting the results (see Appendix A.1.2).



A – Below quadcopter, facing towards quadcopter, approx. 80 cm away. (Blue is quiet room, for comparison.)

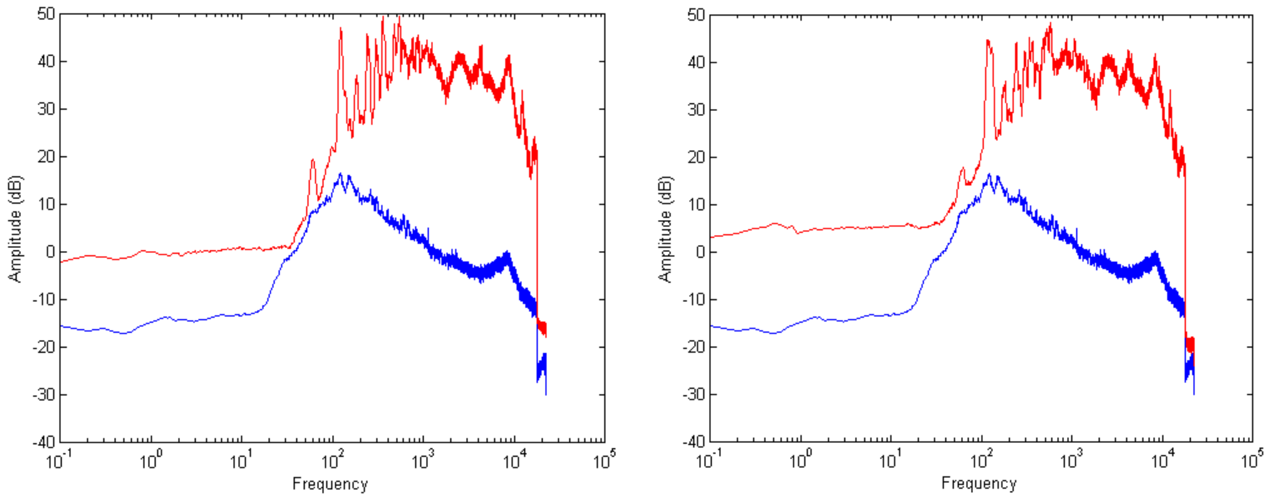


B – Below quadcopter, facing away from quadcopter, approx 20 cm away. (Blue is quiet room, for comparison.)

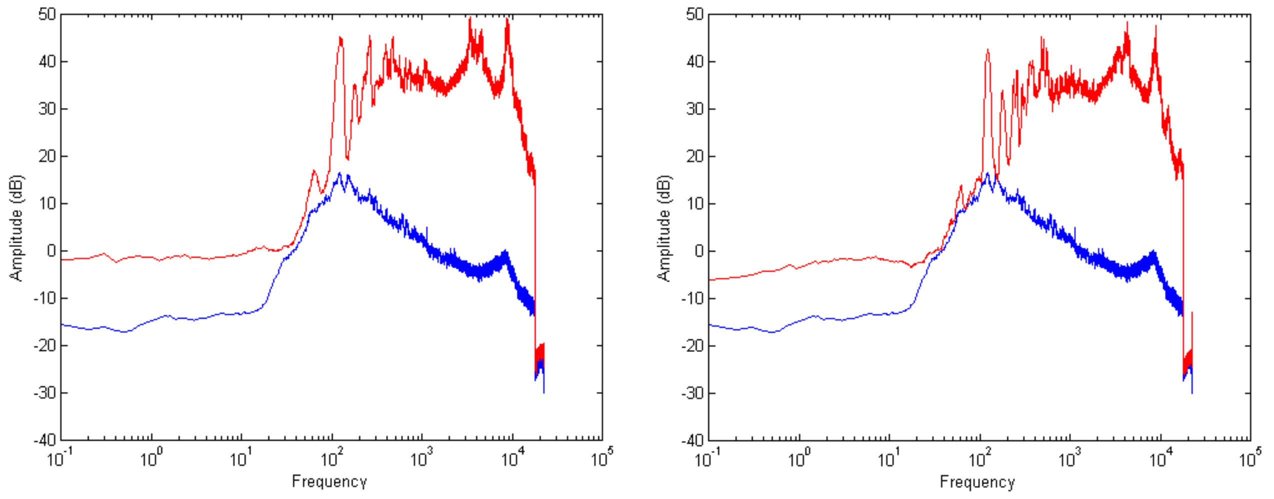


C – Above quadcopter, facing towards quadcopter, approx 20 cm away. (Blue is quiet room, for comparison.)

**Figure 5-4, A-E: Quadcopter Noise Frequency Response Measurements**
Two measurements performed per position



D – Above quadcopter, facing away from quadcopter, approx. 20 cm away. (Blue is quiet room, for comparison.)



E – Beside quadcopter, facing towards quadcopter, approx 20 cm away. (Blue is quiet room, for comparison.)

**Figure 5-4, A-E: Quadcopter Noise Frequency Response Measurements**
Two measurements performed per position

### 5.3.3 Frequency Response Analysis

The recordings provided some valuable insight into noise cancellation and microphone positioning. One observation that could clearly be noticed in the original recordings is that the positions below the quadcopter receive a lot of wind noise, much more than the positions above and beside the quadcopter received. This wind noise would need to be removed in some way if future iterations included the microphone position below the quadcopter.

The effect of the wind noise is noticeable in the frequency response graphs as well. Figure 5-4, A and B show a wide spectrum of level noise with no peaks at any point. However, Figure 5-4, C-E, show lower levels of noise overall and multiple peaks. This was especially true in the lower frequencies up to about 1000 Hz. After 1000 Hz, there was a significant amount of noise over the base noise level of the microphone in the quiet room, but even in this region there occurred a few peaks.

These results support the idea that it may be possible to cancel tonal noises with active cancellation. That is, it may be possible to reduce the noise of the peaks in the frequency response graphs for the positions above and beside the quadcopter. However, there is still a significant amount of wideband noise outside of the peaks which may prove more difficult to remove.

### 5.3.4 Propeller Position Measurements

Many applications of advanced noise cancellation require an external stimulus of some sort in order to know what noises should be cancelled. For propeller applications, this often comes in the form of propeller positioning, hence why the final design discussed the possibility of including propeller position sensors.

An often posed question about propeller positioning is if the photo interrupter can react fast enough to capture the propeller passing through. The photo interrupter tested has a maximum “low to high” propagation delay time of 9 μs and a maximum “high to low” propagation delay time of 15 μs [**52**]. The AR.Drone has a propeller speed of 1,200 to 4,800 rpm, with 3,250 rpm at the hover speed [**53**]. Alternatively, this can be stated as 12.5 ms per rotation at full speed. The photo interrupter would be interrupted twice for each complete rotation of the propeller, or once every 6.25 ms. This speed provides plenty of time for the photo-interrupter to both rise and fall before the next propeller blade passes through.

### 5.3.5 Possibilities for Noise Reduction

While the problem of quadcopter noise appears to be a very large one, there are also lots of options regarding how to reduce the noise. Noise reduction should start with the source of the noise itself, the quadcopter components. From there, it is possible to reduce the amount of noise that reaches the microphone through passive noise reduction elements. As a final option, it is possible to reduce the amount of noise after the microphone has recorded it, using active noise cancellation techniques.

### 5.3.5.1 Quadcopter Improvements

Because quadcopters are not overly complex from a design point of view, there are only a limited number of locations that the noise may come from. Most of the noise comes from the motors and the gear housing and much of the remaining noise comes from the propellers.

Most motors used in quadcopters are cheaply built and noise was not a primary concern during production. Replacing these motors with high quality motors that feature smoother components and quieter operation would be a good way to start noise reduction. The next step would be to change out the gears running from the motors to the propellers. On the AR.Drone, these gears were constructed primarily of plastic. These gears are likely the largest source of noise on the quadcopter, so replacing them with a more advanced gearing that generates less noise would likely provide the largest benefit.

A final potential option for reducing noise would be to alter propellers and propeller operation speed. If the propeller could be switched for a different propeller that generates the same amount of lift at a lower speed, the operating speed of the propellers could be reduced. Reducing the propeller speed would decrease the speed at which the gears and motor rotate, returning a further reduction in noise.

### 5.3.5.2 Passive Noise Cancellation

Once all options for reducing noise generated by the quadcopter have been exhausted, the next option is to perform passive noise cancellation. Passive noise cancellation is normally considered to be any noise cancellation techniques that do not use electronic circuits. For a quadcopter, this comes in the form of insulating the sources of noise.

Commercial airplanes include passive noise cancellation in their frame using dampening materials to attempt to block any noise coming from the outside into the cabin. On a quadcopter, insulation around the gears and the motors could provide one element of noise reduction. The AR.Drone's motor is currently enclosed in a metal case which passes most sounds running through it. Adding some sort of dampening material around this may reduce some of the noise reaching the microphone. The gears on the AR.Drone are currently open with no protection around them. Wrapping these inside of a gear housing that also provides noise dampening would also reduce the amount of noise.

Housing around the propellers can also provide some protection from propeller noise. The AR.Drone actually comes with an indoor propeller shield that provides a bit of protection in some directions. As seen in Figure 5-4E, the noise from beside the quadcopter was a few decibels lower than similar measurements performed above the quadcopter. More advanced versions of this shield might provide further noise reduction, so long as they do not interfere with propeller lift and do not provide too much of a reflection surface for sound.

### 5.3.5.3 Active Noise Cancellation

In contrast to passive noise cancellation, active noise cancellation uses electronic circuits to cancel noise. Commercial propeller planes and helicopters often use active noise cancellation inside of their cabins and in their pilots' headphones [**51**]. Propellers generate a large amount of air pressure periodically that "beats" against the hull of the aircraft. This beating sound can be cancelled using active noise cancellation, normally using an implementation of a multiple error filtered-x LMS algorithm, which uses the propeller position as part of the error filtration.

Such a filtering algorithm may be possible for quadcopters as well. Little research has been done in this area as of yet, but research from commercial airplanes may be an important start in future research. The measurements performed here indicate that there are certain peaks that naturally occur from the system, and cancelling these peaks could likely be possible using LMS and similar algorithms, with the information from the propeller position playing an important role in the error filtration.

## 5.4 Control of Audio Synchronization

A large portion of the final design implementation focused on the synchronization of audio. Audio synchronization is important when multiple impulse responses need to be compared. By having the start time of the recordings synchronized with the playback time, it is possible to see at which location the direct sound arrived first and then to compare the indirect sounds that each heard thereafter. Without the synchronization of audio, it is much more difficult to acquire these timings.

Two different audio synchronization techniques were implemented. The first used a Wi-Fi stack implementation on the iPod Touch while the second used a ZigBee stack on the final design implementation with the Atmel SAM4L. Utilizing the Wi-Fi stack was fairly straightforward and allowed for a couple different design implementations. The ZigBee stack was a bit harder to implement, but allowed for more control over the stack usage.

### 5.4.1 Wi-Fi Stack Synchronization

In the end, two different techniques were used to perform recordings over Wi-Fi using the iPod Touch. The first method used the native HTML5 interface of the iPod Touch app and was very quickly implemented. However, as the results in section 5.4.1.1 demonstrate, the performance was far from the requirements. As a second iteration, a C# implementation was designed that was supposed to give more control over the TCP/IP stack and processor utilization, the results of which are in section 5.4.1.2.

The actual measurements made with each implementation were performed in similar formats. The recording on the iPod Touch was started and a short time later, the playback of the audio would start. The implementation would then inform the iPod Touch to stop recording a short time later. Looking at the audio recording file, it was possible to see the exact moment that the playback started and the overall length of the recording. To limit wireless transmission interferences, an ad-hoc network was created on the PC that the iPod Touch could connect to. Under this configuration, very little data should have been on the network besides that related to the audio synchronization test.

#### *5.4.1.1 HTML5 App Synchronization Results*

The results of the synchronization test with the HTML5 app are shown in Table 5-1. The test was performed nine times. The length of the recording and the start time of sine sweep signal were obtained by reviewing each audio file, from which the “post playback start time” could be found.

From the results, the arithmetic mean and standard deviation of the different values can be found. Overall, the sine sweep signal started on average 3.534 seconds after the recording did, and the recordings lasted 18.640 seconds. The standard deviation was 1.794 seconds for the start time of the sine sweep signal and 1.775 seconds for the recording length. This deviation was five orders of magnitude greater than the value specified in the requirements and improving the functionality of the HTML5 app would likely not result in much better functionality. A better approach was needed and was the reason for the C# implementation.

| Test # | Sine Sweep Start Time (s) | Recording Length (s) | Post Playback Start Time (s) |
|---|---|---|---|
| **1** | 5.909 | 21.060 | 15.151 |
| **2** | 2.364 | 17.438 | 15.074 |
| **3** | 2.355 | 17.485 | 15.130 |
| **4** | 5.871 | 20.944 | 15.073 |
| **5** | 2.317 | 17.345 | 15.028 |
| **6** | 2.333 | 17.485 | 15.152 |
| **7** | 2.349 | 17.531 | 15.182 |
| **8** | 5.997 | 21.014 | 15.017 |
| **9** | 2.311 | 17.461 | 15.150 |

**Table 5-1: HTML5 App Synchronization Measurements**

#### *5.4.1.2 C# App Synchronization Results*

Testing of the C# app was similar to the HTML5 app. The test was carried out eight times and the results were analyzed in the same way. Results are shown in Table 5-2.

The arithmetic mean of the sine sweep start time was 0.326 seconds with a standard deviation of 62.3 milliseconds. The recording length had an average value of 5.137 seconds and a standard deviation of 64.6 milliseconds. This implementation reduced the deviation to within about three orders of magnitude of the requirements – better than the HTML5 app, but still far from the desired goal.

| Test # | Sine Sweep Start Time (s) | Recording Length (s) | Post Playback Start Time (s) |
|---|---|---|---|
| **1** | 0.345 | 5.140 | 4.795 |
| **2** | 0.294 | 5.108 | 4.814 |
| **3** | 0.292 | 5.119 | 4.827 |
| **4** | 0.286 | 5.085 | 4.799 |
| **5** | 0.268 | 5.085 | 4.817 |
| **6** | 0.415 | 5.224 | 4.809 |
| **7** | 0.424 | 5.248 | 4.824 |
| **8** | 0.280 | 5.085 | 4.805 |

**Table 5-2: C# App Synchronization Measurements**

#### *5.4.1.3 TCP/IP Stack Timing Results*

After the results of the C# app were observed, the question of whether the Wi-Fi stack could be used for audio synchronization came into question and another set of measurements was suggested. In this test, traces of all wireless messages sent and received from the PC application were tracked using WireShark [**54**]. Ten audio tests were performed in the same way as those in the previous section.

A total of eight messages are normally sent during each audio recording process (ignoring retransmissions and similar messages) as seen in Figure 5-5. There is an initial start request from the PC to the iPod which returns an ACK (every message must receive an ACK confirmation back). This is "Start –

Request to First ACK". After a slight pause, referred to as "Start – ACK to Response", the iPod then sends back a response message that the PC must return an ACK for, referred to as "Start – Response to Final ACK". Once the recording is finished, a stop request is sent from the PC to the iPod and an ACK is returned, referred to as "Stop – Request to First ACK". After the "Stop – ACK to Response" break, the iPod finally sends back a stop response message that the PC must ACK, referred to as "Stop – Response to Final ACK".

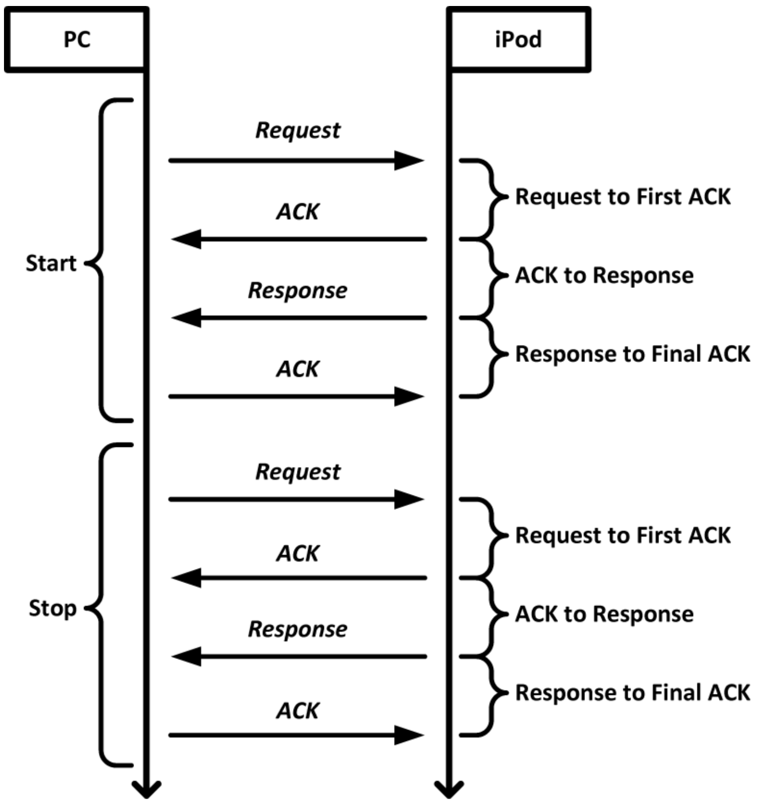


**Figure 5-5: TCP/IP Transactions**

The results of the tests are shown in Figure 5-6. As can be seen, the "Start – Response to Final ACK" and "Stop – Response to Final ACK" dominated the amount of time taken, followed by "Start – ACK to Response" and "Stop – ACK to Response". In comparison, the "Start – Request to First ACK" and "Stop – Request to First ACK" took very little time. What is perhaps more important, however, is the variability between the samples. The "Start – Response to Final ACK" varies from 217 ms to 300 ms and the "Stop –Response to Final ACK" varies from 194 ms to 294 ms. This type of variability is likely what caused the large standard deviation in the C# app results.

Finding the source of this variability can be difficult. The "Response to Final ACK" variability comes from the PC, as it was the PC that must send the final ACK. This likely means that the PC's Wi-Fi stack does not have priority and must wait on other processes to complete first before the ACK is sent. However, the "ACK to Response" variability is caused by the iPod. During this time, the iPod is starting up the recording and then forming the response message. If another task on the iPod's processor takes priority, then it may take a while for the response message to be sent.

### 5.4.2 ZigBee Stack Synchronization

The SAM4L implementation used a ZigBee stack for communication rather than Wi-Fi. The ZigBee stack is significantly smaller and less complex than the Wi-Fi stack, which allowed for easy implementation of TPSN. TPSN was reported to have had very good time synchronization and similar results were desired in this implementation.

To test the synchronization from end to end, the ground node was setup to output a low to high transition on a specific pin at the exact same time as beginning the audio playback. This pin was then wired to the positive input of the differential ADC on the remote node, while the negative input was wired to the ground pin of the ground node. When a recording was started on the nodes, the resulting recorded "audio" file would show a transition from zero to max amplitude. In order to guarantee that the transi-

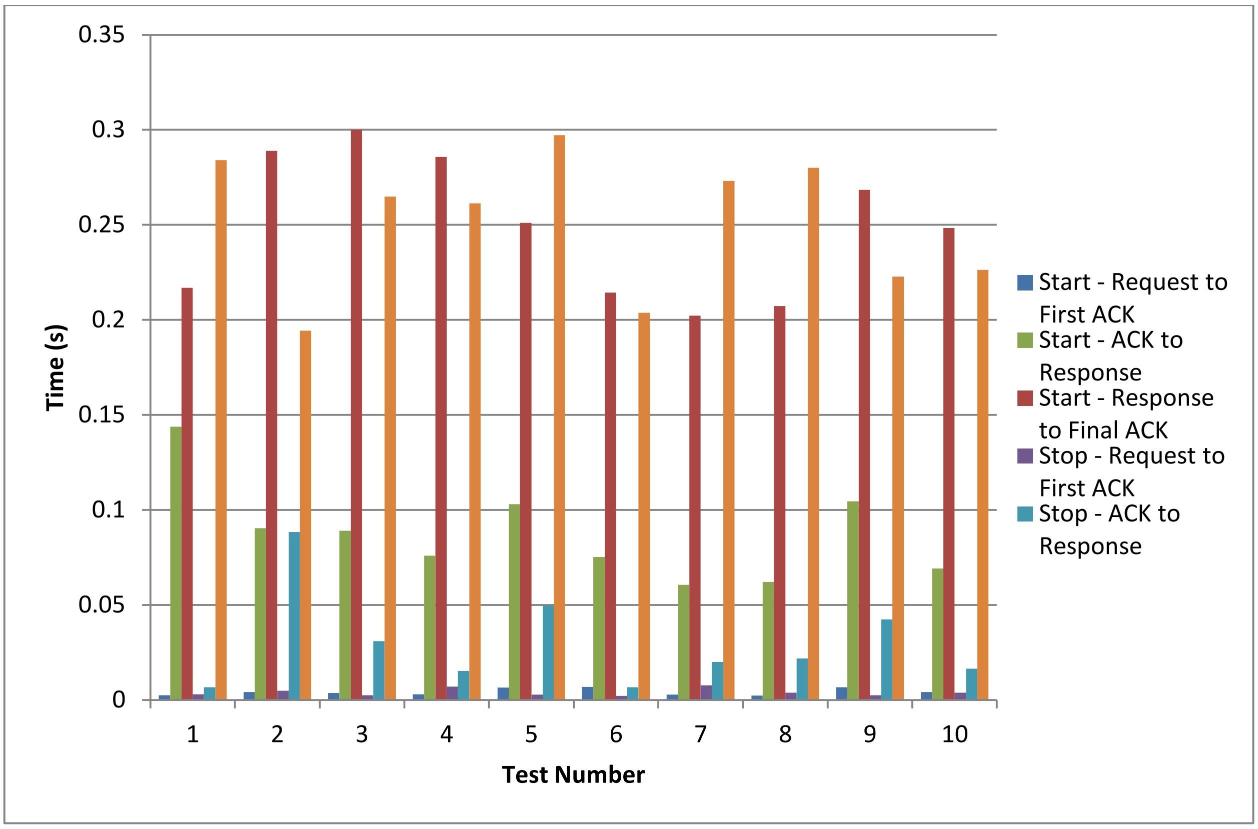


**Figure 5-6: TCP/IP Transaction Results**

tion from zero to max would occur while the recording was taking place, the playback was artificially delayed by one millisecond after the start of the recording.

Without the post-recording processing that was outlined in section 4.6.3.2, the audio started four samples too early when recording at 22050 Hz, approximately 180 milliseconds. Adding in the post-processing cleared up a few of the outlying recordings that were not four samples too early. Since the tests showed the playback started four samples early very consistently, a hard-coded correction value was added into the post-recording processing that took care of this deviation, resulting in a perfectly synched audio recording.

Since the results now showed that the audio was synched down to the sample frequency, the original requirement for audio synchronization had been met. This was three orders of magnitude better than audio synchronization using the PC and the Wi-Fi stack were able to acquire. Use of Wi-Fi should not be ruled out for future use as the communication protocol, however, Wi-Fi includes several challenges that ZigBee does not have.

## 5.5 Audio Quality

In addition to canceling noise and synchronizing audio, the SAM4L's audio quality that is actually output by the DAC and that is recorded by the ADC is also important. To test this, the nodes were brought into an anechoic chamber and measured.

### 5.5.1 Quality of Generated Audio

The DAC was tested by routing the output from the ground node to the input of a sound card. The sound card included an amplifier that boosted the signal to values necessary for recording on the PC.

Then the ground node was instructed to output a 500 to 10,000 Hz sine sweep signal over two seconds. This signal was recorded on the PC and analyzed in MATLAB.

Results from the analysis are shown in Figure 5-7. The red line is the calculated output by the ground node and the blue line is the PC recording of the output. An additional comparison to the ideal 500 to 10,000 Hz sine sweep signal was also made, but is not shown as the ideal and the calculated outputs were nearly identical. The largest difference between the ideal and the calculated outputs was only 0.1 dB.

As can be seen in the figure, there appears to be a rise in the lower frequencies. This is outside of the test frequencies of 500 to 10,000 Hz but should still be evaluated in future designs. (This was seen on two different test setups.) Inside of the test frequency range, there also is a slight drop in the highest frequencies. However, for the majority of the test frequencies, the two outputs are very similar.

## 5.5.2 Quality of Recorded Audio

Recording quality is just as important as the generated sound quality. Two tests were carried out to test the microphone and the ADC recording quality. In the first test, the same Knowles microphone used on the remote node was attached to a PC to record a sine sweep signal and was compared to a measurement microphone in the same setup. The second test attached the Knowles to the remote node and the sine sweep signal was again recorded and compared to the measurement microphone.

For the first test, the setup for the measurement microphone consisted of a condenser measurement microphone (cartridge type 4149) fed into a Norsonic 1201 microphone preamplifier. This then led to a Norsonic Front End Type 336 amplifier and then in to the computer. For the Knowles microphone,

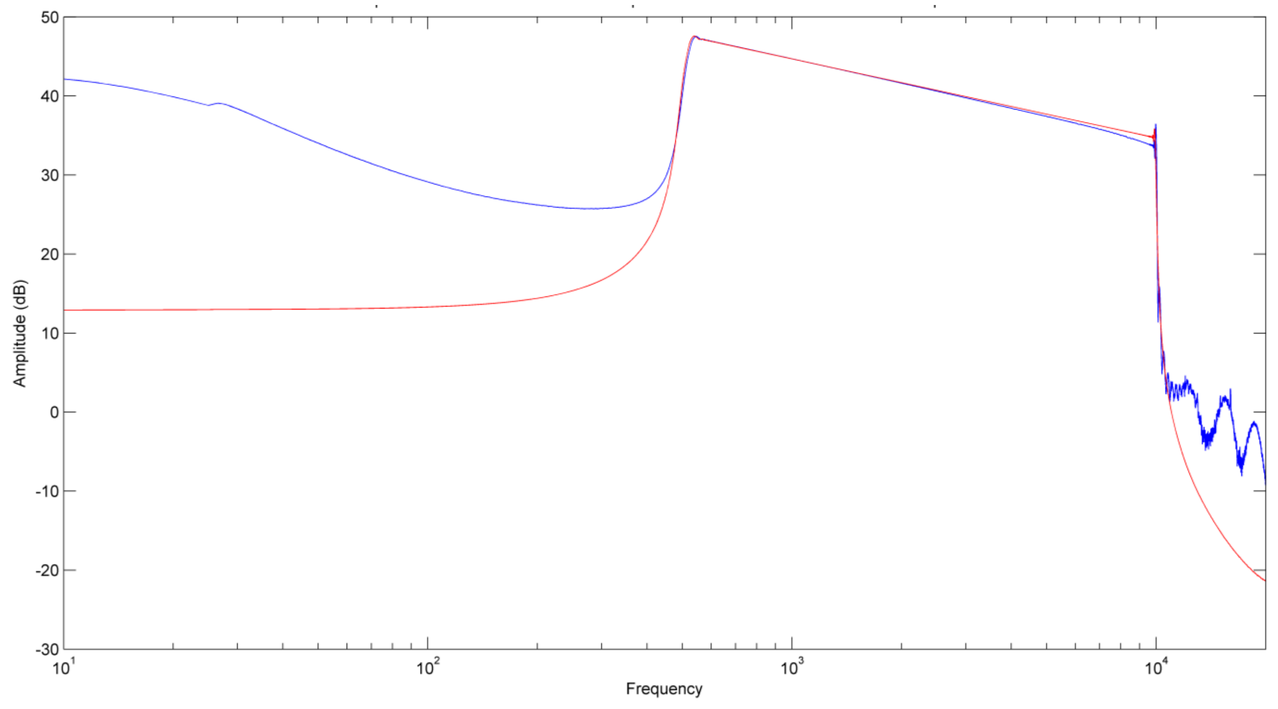


**Figure 5-7: DAC Audio Quality Measurement**
Red is the calculated output by the ground node. Blue is the actual recorded signal output by the ground node.

a Shure FP23 preamplifier was used to boost the signal into the recording PC. The microphones were placed 1.47 meters from the loudspeaker and 1.05 meters above the metal grating surface of the anechoic chamber. A 500 to 10,000 Hz, two second sine sweep signal was played out of the loudspeaker and recorded by the microphones. In addition, a loopback from the sound card output channel to a second input channel on the sound card was used to get the actual signal that was output to the loudspeaker.

The recorded signals were post-processed in MATLAB using a version of the script in Appendix A.1.2. This script divides the FFT of the microphone signal by the FFT of the sound card output. An IFFT of the result is then taken to get the impulse response. The impulse response is windowed to the critical area and another FFT is performed to get the final frequency spectrum. The results of this process for both microphones are shown in Figure 5-8. The figure shows that the frequency responses of the two microphones are fairly similar, with the Knowles microphone boosting the higher frequencies a few decibels.

For the second test, the Knowles microphone was attached to the remote node. The setup positioning was otherwise the same as the previous test. The same 500 to 10,000 Hz signal was recorded by the remote node. Analysis of the results was performed in MATLAB using the same process as the previous test.

The results for this test are shown in Figure 5-9. The frequency response of the board's recording is compared to the frequency response of the measurement microphone from the previous test. This time, the frequency response of the Knowles microphone is not quite as close to the measurement microphone. Additionally, low frequencies have a large gain that the ADC circuit appears to be including in the final recording.

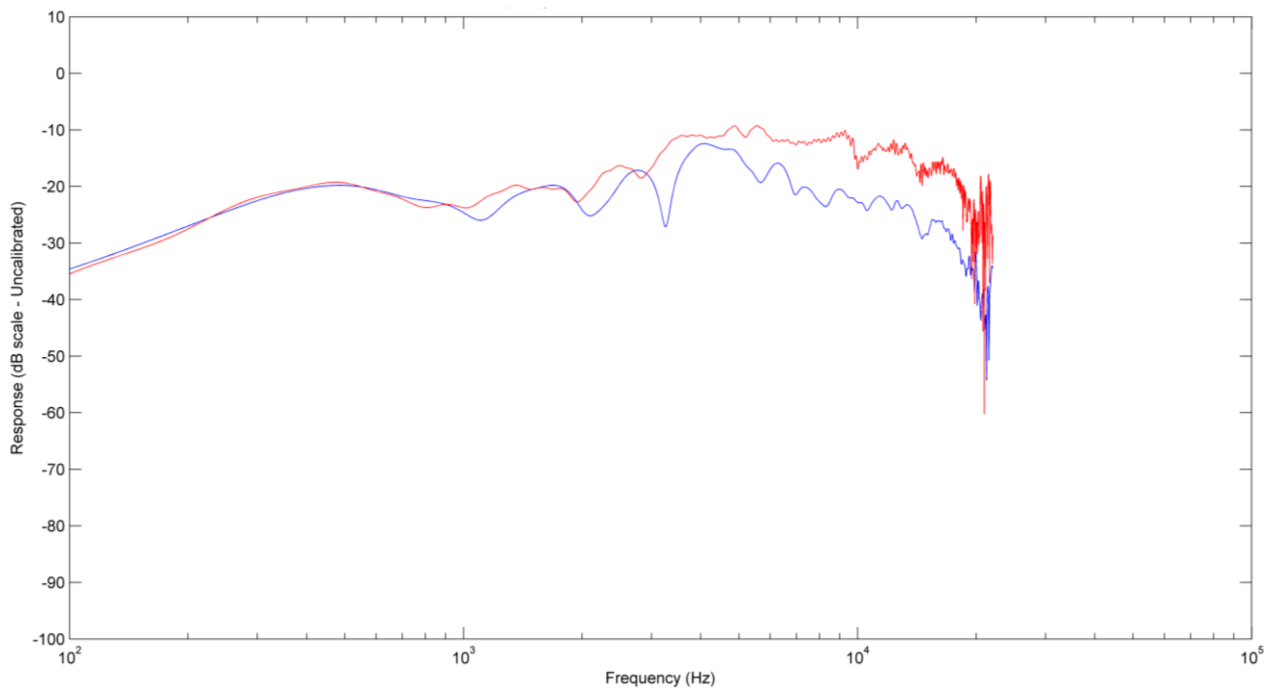


**Figure 5-8: Measurement Microphone vs. Knowles Microphone Frequency Response**
Blue line is the measurement microphone and red line is the Knowles microphone.

Qualitatively, there is still a low level of noise in the recording despite the low pass filter and very high frequency sounds include a few artifacts. There also is a slight DC offset that was corrected prior to being analyzed in MATLAB. (Keeping it in results in a larger gain at the lowest frequencies.)

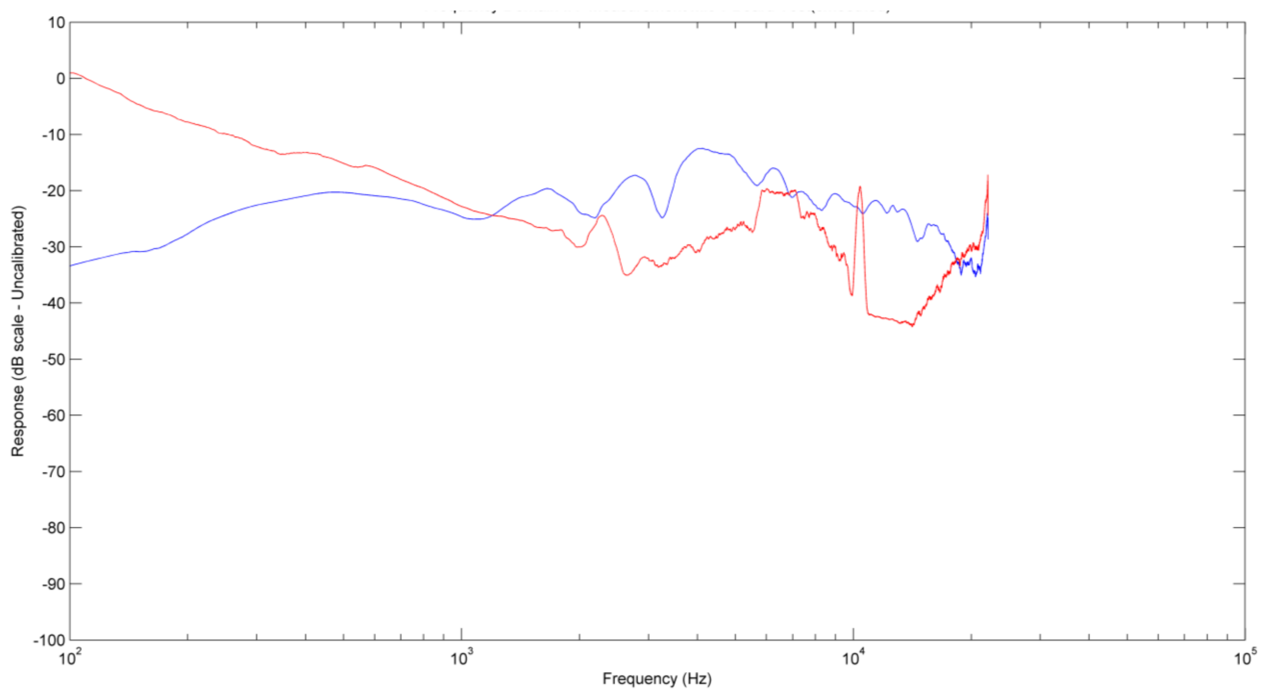


**Figure 5-9: Measurement Microphone vs. Board Microphone Frequency Response**
Blue line is the measurement microphone and red line is the Knowles microphone.

# 6 Final Design Discussions and Conclusions

Through this research, two proposed designs have been implemented and tested. The first was a quick prototype using the Apple iPod Touch that allowed for a large quantity of measurements and a basis for the second design. Atmel's SAM4L Xplained Pro was the base for the final design. In this section, a final discussion on the quality of the audio recordings and synchronization techniques is given. Then, the original goals and challenges are evaluated based on the final design and final conclusions on the design are put forth. The section concludes with possible design improvements and future work that could build off this research.

## 6.1 Quality of Audio Recordings

The results from section 5.5 showed Atmel's ABDACB was capable of outputting a good audio signal, even with a low pass filter that did not meet Atmel's recommendations. There appeared to be interference from some low frequency sounds, but cleaning this should be possible with a small bandpass filter or more advanced filtering methods.

The ADC's results were not quite as good as the ABDACB's, but were still functional, especially given the system that they were built in. The small size of the microphone made it more difficult to design around, and the limited abilities of the ADC showed in the recordings final quality. Further improvements to the ADC are certainly possible with better signal conditioning and possibly more sophisticated recording hardware.

## 6.2 Synchronization Techniques

As the results in section 5.4.1 showed, Wi-Fi demonstrated significant problems for audio synchronization. The final Wi-Fi implementation was able to have an accuracy of approximately 60 milliseconds, far away from the desired goal. Further, Wi-Fi has a demonstrated variability in its transmission and reception timings on many systems, hindering adoption as a good audio synchronization medium.

In contrast, ZigBee showed a very good ability to perform synchronization down to the levels required for audio measurements. Because of the reduced complexity of the ZigBee stack, it was possible to implement TPSN on top of the stack while utilizing low level time-stamping in the stack. TPSN combined with additional post-processing led to audio recordings that began sampling at the same time as the playback began.

The results demonstrated that the ZigBee/TPSN implementation had a sample accuracy of at least 22,050 Hz or 45 microseconds. However, the clocks used on board the nodes have a theoretical accuracy down to one microsecond. New methods of testing would need to be developed in order to test if the boards are able to achieve this level of accuracy without further modifications. If one microsecond accuracy were indeed achievable, sampling frequencies in excess of 192 kHz would be possible, at least from a timing standpoint.

The use of Wi-Fi for audio synchronization has not been completely ruled out however. To use Wi-Fi, a protocol such as TPSN would need to be implemented across this communication medium. In order to implement TPSN, the data packets would need to be time-stamped at the MAC layer, which is significantly lower than most modules allow the program code to reach. Specialized hardware or Wi-Fi stacks would be needed to gain this ability. Even with this specialized implementation, the results from section 5.4.1.3 indicate that Wi-Fi would need to be given more time to complete operations as compared to ZigBee, and that the synchronization accuracy acquired using ZigBee may not be possible to reliably achieve over Wi-Fi. The large presence of Wi-Fi hardware makes it a very attractive alternative.

## 6.3 Ability to Meet System Goals, Challenges, and Restrictions

In section 2.5, a selection of obstacles was laid out that would need to be met if a quadcopter could successfully perform high quality audio measurements. The following reviews the “Quadcopter imposed challenges” and “Audio measurements and embedded systems” goals in regards to the final design implementation on the SAM4L Xplained Pro and the prototype implementation on the Apple iPod Touch. (“Project Description” goals are evaluated in section 6.4.)

Quadcopter imposed challenges:

1) ***Maximum payload of 100 grams*** – The Atmel SAM4L Xplained Pro, ZigBit module, microphone circuitry, and microphone batteries totaled slightly more than 100 grams. Removing the batteries and using the battery on board the AR.Drone provided a significant savings of 40-50 grams alone, thus easily meeting the 100 gram maximum.
2) ***Low power design*** – The SAM4L is intended to be a low power system and as such draws a low current. Additional savings can be gained by further using the Atmel power saver modes that are included in the chip’s design.
3) ***Low noise recordings*** – The final design did not try to reduce any noise on the quadcopter itself, nor did it implement any sort of noise cancellation. However, tests regarding noise were performed with the iPod Touch and several possible design implementations were discussed in section 5.3.5.
4) ***Increased microphone stability*** – Similar to the previous goal, the final design did not focus on implementing any way to stabilize the microphone. However, tests regarding microphone stability were performed. These results indicated that for a quadcopter with slow and small movements, audio quality would not be greatly affected. To further increase quality though, several stabilization techniques were discussed in section 5.2.4.

Audio measurement and embedded systems:

1) ***Synchronization within recording sample rate*** –The final design implementation had a consistent recording that started within the first sample of the audio playback, fully meeting the goal.
2) ***Meet ADC and DAC hard real-time deadlines*** – A full RTOS was used in the final design, allowing the deadlines of both ADC and DAC to be met in a controlled way, along with meeting the soft deadlines of other tasks.
3) ***Quality audio performance*** – The final sound quality output by the DAC showed very good results consistent with what would normally be desired from an embedded system. The ADC

was a bit limited from the beginning as the bit depth was low, however, the recordings turned out decent results. Better performance could be obtained through a more complex design (see section 6.5), though, the results were good for an embedded system not originally intended to record audio.

## 6.4 Final Design Conclusions

Overall, two of the four quadcopter imposed challenges were fully met and the other two challenges were evaluated and possible designs for solving those challenges were conceived. Of the three audio measurements and embedded systems challenges, two were fully met, and the third was satisfactorily met. Further improvements are always possible, but for a first full design implementation, it obtained decent results.

From the original project description, there were three possible challenges that could be covered. The research only needed to cover one of those challenges; however, the research presented here touches on all three challenges and goes into depth on solving the third challenge. The challenges and presented solutions in this paper were:

***Quadcopters generate a lot of noise. How can this noise be cancelled?***
The noise from quadcopters was analyzed in sections 5.3.3. Possibilities for noise cancellation were discussed in sections 5.3.5.2 and 5.3.5.3. Two types of noise cancellation exist: passive and active. For passive noise cancellation, several methods of cancelling sound from the noisy components were discussed including wrapping insulating materials around the loud components and putting up a protective housing around the propellers. For active noise cancellation, the noise from the quadcopter is cancelled by electronics. The beginning of a propeller positioning system was implemented in section 4.7, which is the basis for an active noise cancellation filter. Further details and possible implementations of active noise cancellation were discussed in section 5.3.5.3.

***Are there ways of reducing the amount of noise generated by a quadcopter?***
The best way to cancel noise is to stop the noise from being generated in the first place. The design of a quadcopter can reduce the amount of noise that needs to be cancelled. Section 5.3.5.1 discussed options for how to reduce the mechanical noise coming from the quadcopter. These suggestions included changing out the motors for higher quality motors that generate less noise, changing to a gearing system that is internally housed, and switching propellers that give greater lift at lower speeds.

***How can a system with microphone, signal processing, and recording storage be designed?***
The large majority of this thesis focused on the implementation of an embedded recording system. This system started from a fast prototype version that allowed for performing significant measurements and resulted in a final design that was almost entirely self-contained, requiring very little interaction from the user. It was able to wirelessly synchronize two nodes, one on the ground connected to the PC and loudspeaker, and the other in the air attached to the quadcopter. On command from the PC, the ground node would alert the remote node to begin recording at the

same time as it began playing a sine swept signal in real time. The remote node recorded this audio to a USB flash drive and after the recording was finished, would finalize the recording by trimming excess sound from the beginning and end of the recording.

While quadcopters are an attractive option for performing audio recordings, are they capable of performing quality measurements? This was the primary question formulated from the project description. Based on the designs, implementations, and results presented in this thesis, quadcopters do appear to be a potentially good option for performing measurements. The final design implementation functioned satisfactorily within the specifications that were given. Further research and work regarding quadcopters and audio measurements is needed to make them into viable devices for making quality recordings. However, the work presented here should provide a stable foundation for that future work.

## 6.5 Potential Areas for Design Improvement

Future implementations of what was the final design implementation for this thesis should focus on several areas of improvement. The final design presented here was based on meeting certain requirements while trying to keep the complexity low. Future iterations will need to increase the design complexity slightly in order to gain better results.

From the implementation and results of the final design, several areas of improvement were identified. These include: audio generation (DAC design), audio recording (ADC design), PC interface improvements, and low power design. Details for each area are described below.

### 6.5.1 Audio Generation (DAC Design)

While the audio that results from the ABDAC on the Atmel SAM4L is of good quality, several possible improvements to the hardware that follows it could improve the overall quality. In the final design, a first order low pass filter was used, even though, the ABDAC documentation recommends a fourth order or larger low pass filter [**34**]. A fourth order would allow a steeper cutoff next to the sampling frequency of the system and remove any of the noise generated by the Sigma Delta Modulator.

Following up the low pass filter with an integrated amplifier or voltage follower circuit would allow the circuit to drive higher loads. With the implementation of both the low pass filter and the amplifier, the DC offset would also likely need to be adjusted. The DC offset of the final design was quite sensitive and future implementation could make this slightly more stable.

Upon enabling or disabling the ABDAC, a large DC swing is currently sent through the system. While this did not affect the output of the sine swept signal, it did show up on recordings of the output signal. Adjusting this either in code or in hardware would improve the resulting output, especially for sensitive devices that may be attached to the output system.

### 6.5.2 Audio Recording (ADC Design)

The ADC included on the SAM4L has only a 12 bit depth. Future implementations would likely want to increase this bit depth to 16 or 24 bits by using an external ADC module. In addition, the signal conditioning coming into the ADC could be improved. Using the operational amplifier to create a larger gain would reduce the need to apply a gain at the ADC and would likely reduce some of the noise. Along with this would come a better filter. The current filter is a first order system but could be increased to a higher order to provide more dampening of high frequency noise.

As the SAM4L's ADC was not designed for audio, the clock on it does not exactly match the sample rates at which audio is normally recorded. A better ADC may include audio sampling frequencies. Another option is to use an audio encoder, such as those by VLSI [**55**]. These encoders are specially designed to take in audio from a microphone and then generate audio files in a variety of formats. Encoding ICs can be connected to the SAM4L via a number of interfaces, such as $I^2C$.

Another improvement that would help increase audio quality would be to increase write speed to the USB. The final design was limited to a sample rate of 22,050 Hz because the USB could not handle a higher throughput while the processor was also receiving data from the ADC. If an alternative method of writing data to the USB that increased the speed with which audio could be written, it would allow for higher sampling frequencies.

Finally, the clocks on the remote node and ground node of the SAM4L system are not exactly the same and drift from each other slightly. While the first sample of both nodes occurs at about the same time, synchronization between the nodes does not occur until after the recording is finished, allowing the nodes to be sampling at slightly different instants as time goes on. A good impulse response measurement system would have the samples synchronized at all moments and future iterations will need to have an ADC that can correct for the clock drift.

### 6.5.3 PC Interface Improvements

The UART and USB interfaces that the PC connected to in the final design were fairly simplistic in the functions that they provided the user with. The amount of status data was also limited. Expanding the functionality of these interfaces to include such functionality as repeating the previous test or auto repeating tests in certain circumstances would be helpful for increasing the speed of the overall system. System messages about such things as failed recordings or overrun/underruns on the nodes would also be helpful for the user.

### 6.5.4 Low Power Design

While the SAM4L is already a low power consumer, it has the ability to go into sleep modes that cut the power consumption much further. Several areas of the final design could benefit from the use of these sleep modes. These include the idle task, which could sleep after it has finished anything important, and the AUDIO tasks on both nodes when they are waiting on the peripherals to perform certain functions.

## 6.6 Future Work

In addition to the improvements that could be implemented in the design, there are also several potential areas for further research and design work. The two largest of these are in noise cancellation and quadcopter improvement. These two tasks currently stand as the biggest barriers to full implementation of sound recording systems using quadcopters. In addition to these, development work to build a fully automated measurement system and design work on building a custom PCB and hardware enclosure are two other areas that need time investments.

### 6.6.1 Noise Cancellation

As has been suggested several times throughout this research, there are several possibilities for noise cancellation. A strong possibility is in active noise control using the propeller position. The multiple error filtered-x LMS algorithm has had strong usage in other areas of noise cancellation, such as is suggested in [**51**]. Its use in filtering quadcopter noise may also be possible, but further research into this and other algorithms is needed.

### 6.6.2 Quadcopter Improvements

There are multiple parts of the quadcopter itself that could use improvement for several reasons. The first is in the use of components that generate less noise. The motors, gears, and propellers are the primary contributors to this noise, so research into alternatives may yield less noise that needs to be cancelled. The addition of passive noise cancellation devices (e.g. insulation, padding, and protective shields) would also be useful in reducing the amount of noise that would need to be removed by an active noise filter.

The addition of a stabilized platform for the microphone would help reduce the noise and distortion that occurs due to vibration. Systems may be as simple as rubber or elastic that do not pass on high frequency movements, or they may be more complex, such as the gimbal systems often used on the professional video quadcopters.

### 6.6.3 Software Development

The intention of the USB API built in the final design was for use in a PC application that automates the process of moving around a room and performing measurements. The API could be combined with the AR.Drone's SDK to provide such a system. This would remove much of the need for a user to be present and working during the duration of the measurement. Setting the quadcopter to take off, locate itself in the room, begin recording, and then moving to the next location automatically would be the ideal situation.

### 6.6.4 PCB and Enclosure Design

A final possibility for future work exists in the design of a PCB and enclosure for the PCB. A custom designed PCB would remove many of the extraneous features on SAM4L Xplained Pro board, integrate the wireless module, and could include the microphone signal conditioning circuitry as well. This would reduce the overall size, and likely reduce the overall weight of the design as well. A weight reduction on the remote node is a good chance to increase the flight time of the quadcopter.

An enclosure for the PCB and peripherals would also be a good way to protect the components from damage. It is not unreasonable to expect a quadcopter to crash or make a hard landing at some point. Protecting the components in a case would extend their lifetime and prevent unneeded downtime while a new PCB is built. This design could also be integrated into a gimbal system to provide microphone stability.

## References

[1] V. Verfaille, M. Holters, and U. Zölzer, "Fundamentals of Digital Signal Processing," in *Digital Audio Effects*, 2nd ed., U. Zölzer, Ed. Chichester: John Wiley & Sons Ltd, 2011, p. 34.

[2] P. Adenot, C. Wilson, and C. Rogers, "Web Audio API," W3C, October 10, 2013. [Online]. Available: http://www.w3.org/TR/webaudio

[3] A. Farina, "Advancements in impulse response measurements by sine sweeps," in *Audio Engineering Society*, Vienna, 2007. [Online]. Available: http://pcfarina.eng.unipr.it/Public/Papers/226-AES122.pdf

[4] B. Tice, "Unmanned Aerial Vehicles: The Force Multiplier of the 1990s," *Airpower Journal*, Spring 1991. [Online]. Available: http://www.au.af.mil/au/cadre/aspj/airchronicles/apj/apj91/spr91/4spr91.htm

[5] J. Stanley. (2013, May) *"Drones" vs "UAVs" -- What's Behind A Name?* [Online]. Available: https://www.aclu.org/blog/technology-and-liberty-national-security/should-we-call-them-drones-or-uavs

[6] N. Kelley, "Drones Buzz Sochi: UAVs are changing the way we watch sports," *OutsideOnline*, February 2014. [Online]. Available: http://www.outsideonline.com/news-from-the-field/Drones-Buzz-Sochi.html

[7] Amazon.com, Inc. (2014) *Amazon Prime Air*. [Online]. Available: http://smile.amazon.com/b?node=8037720011

[8] Deutsche Post DHL. (2013, December) *DHL Paketkopter im Anflug*. [Online]. Available: http://www.dpdhl.com/de/presse/mediathek/tv-footage/tv-footage_dhl_paketkopter.html

[9] T. Mogg, "Drone deliveries set to start in Dubai," *Digital Trends*, February 2014. [Online]. Available: http://www.digitaltrends.com/cool-tech/drone-deliveries-set-to-start-in-dubai

[10] European Commission. (2014, April) *Remotely Piloted Aviation Systems (RPAS) - Frequently Asked Questions*. [Online]. Available: http://europa.eu/rapid/press-release_MEMO-14-259_en.htm

[11] C. Wickham, "Military drones zero in on $400 billion civilian market," *Reuters*, November 2012. [Online]. Available: http://www.reuters.com/article/2012/11/14/us-science-drones-civilian-idUSBRE8AD1HR20121114

[12] A. Schoellig, R. D'Andrea, and F. Augugliaro, "Dance of the Flying Machines," *IEEE Robotics & Automation Magazine*, vol. 20, no. 4, pp. 96-104, December 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6655919 doi: 10.1109/MRA.2013.2275693

[13] P Pounds, R Mahony, and P Corke, "Modelling and control of a large quadrotor robot," *Control Engineering Practice*, vol. 18, no. 7, pp. 691-699, July 2010. [Online]. Available: http://dx.doi.org/10.1016/j.conengprac.2010.02.008 doi: 10.1016/j.conengprac.2010.02.008

[14] G. Hoffmann, H. Huang, S. Wasl, and E. Tomlin, "Quadrotor helicopter flight dynamics and

control: Theory and experiment," in *Proc. of the AIAA Guidance, Navigation, and Control Conference*, 2007. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.9015&rep=rep1&type=pdf

[15] UPenn Engineering. (2014, April) *GRASP Laboratory*. [Online]. Available: https://www.grasp.upenn.edu/

[16] Institute for Dynamic Systems and Control - ETH Zurich. (2014) *Flying Machine Arena*. [Online]. Available: http://flyingmachinearena.org/

[17] Massachusetts Inst. Technol. (2014, April) *Aerospace Controls Laboratory*. [Online]. Available: http://acl.mit.edu/

[18] K. Okutani, T. Yoshida, K. Nakamura, and K. Nakadai, "Outdoor Auditory Scene Analysis Using a Moving Microphone Array," in *IEEE/RSJ International Conference on*, Vilamoura, 2012, pp. 3288-3293.

[19] O. Liang. (2014, February) *How to choose battery for Quadcopter, Tricopter and Hexacopter*. [Online]. Available: http://blog.oscarliang.net/how-to-choose-battery-for-quadcopter-multicopter/

[20] Parrot SA. (2013) *AR.Drone 2.0*. [Online]. Available: http://ardrone2.parrot.com/

[21] D. Cornish. (2013, March) *ESA launches drone app to crowdsource flight data*. [Online]. Available: http://www.wired.co.uk/news/archive/2013-03/15/esa-crowdsource-drones

[22] Parrot SA. (2009) *Open Source Software used in 'ARDrone'*. [Online]. Available: https://devzone.parrot.com/projects/show/oss-ardrone2

[23] Parrot SA. (2009, December) *AR.Drone Open API Platform*. [Online]. Available: https://projects.ardrone.org/

[24] Apple Inc. (2014) *Refurbished iPod Touch*. [Online]. Available: http://store.apple.com/us/product/FC540LL/A/refurbished-ipod-touch-8gb-black

[25] *Raspberry Pi*. [Online]. Available: http://www.raspberrypi.org/

[26] Nordic Semiconductor. *nRF51822 Evaluation Kit*. [Online]. Available: http://www.nordicsemi.com/eng/Products/Bluetooth-R-low-energy/nRF51822-Evaluation-Kit

[27] Atmel Corp. (2014) *SAM4L Xplained Pro Evaluation Kit*. [Online]. Available: http://www.atmel.com/tools/ATSAM4L-XPRO.aspx

[28] Nordic Semiconductor. *nRF51822*. [Online]. Available: http://www.nordicsemi.com/eng/Products/Bluetooth-R-low-energy/nRF51822

[29] Nordic Developer Zone. (2014, February) *CMSIS RTOS with NRF51822 and S110*. [Online]. Available: https://devzone.nordicsemi.com/index.php/discussions/cmsis-rtos-with-nrf51822-and-s110

[30] Real Time Engineers Ltd. (2012, October) *nRF51822 - FreeRTOS Support Archive*. [Online]. Available: http://www.freertos.org/FreeRTOS_Support_Forum_Archive/October_2012/freertos_nRF51822_5943467.html

[31] Keil Software, Inc. (2013) *RTX Real-Time Operating System*. [Online]. Available:

http://www.keil.com/arm/rl-arm/kernel.asp

[32] Atmel Corp. (2014) *picoPower - Low Power Technology*. [Online]. Available: http://www.atmel.com/technologies/lowpower/default.aspx

[33] Atmel Corp. (2006, February) *AVR120: Characterization and Calibration of the ADC on an AVR*. [Online]. Available: http://www.atmel.com/images/doc2559.pdf

[34] Atmel Corp. (2014, March) *ATSAM ARM-based Flash MCU - SAM4L Series*. [Online]. Available: http://www.atmel.com/images/atmel-42023-arm-microcontroller-atsam4l-low-power-lcd_datasheet.pdf

[35] D. Mills. (2012, May) *Executive Summary: Computer Network Time Synchronization*. [Online]. Available: http://www.eecis.udel.edu/~mills/exec.html

[36] J. Elson, L. Girod, and D. Estrin, "Fine-Grained Network Time Synchronization using Reference," in *Proceedings of the Fifth Symposium on Operating Systems Design*, Boston, 2002. [Online]. Available: http://www.stanford.edu/class/cs344e/papers/broadcast-osdi.pdf

[37] "Timing-sync Protocol for Sensor Networks," in *SenSys '03*, New York, November 2003, pp. 123-149, Proceedings of the 1st international conference on Embedded networked sensor systems. [Online]. Available: http://www.cens.ucla.edu/sensys03/proceedings/p138-ganeriwal.pdf doi: 10.1145/958491.958508

[38] Knowles Corp. (2013, March) *SPM0408LE5H-TB Datasheet*. [Online]. Available: http://www.knowles.com/eng/content/download/3901/49448/version/4/file/SPM0408LE5H.pdf

[39] Appologics. (2013) *AirBeam*. [Online]. Available: http://appologics.com/airbeam

[40] Appologics UG. (2013, November) *iTunes Preview: AirBeam - Live HD video surveillance and motion detection*. [Online]. Available: https://itunes.apple.com/en/app/airbeam/id428767956

[41] Facebook. (2014) *Share Bookmarklet*. [Online]. Available: https://www.facebook.com/share_options.php

[42] Microsoft Corp. (2014) *HttpWebRequest Class*. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.net.httpwebrequest.aspx

[43] Micrsoft Corp. (2014) *SoundPlayer Class*. [Online]. Available: http://msdn.microsoft.com/en-us/library/system.media.soundplayer.aspx

[44] Compaq Comput. Corp., Hewlett-Packard Company, Intel Corp., Lucent Technol. Inc., Microsoft Corp., NEC Corp., Koninklijke Philips Electron. N.V. (2000, April) *Universal Serial Bus Specification Revision 2.0*. [Online]. Available: http://www.usb.org/developers/docs/usb20_docs/usb_20_042814.zip

[45] Atmel Corp. (2014) *AT86RF233 Amplified ZigBit Xplained Pro Extension*. [Online]. Available: http://www.atmel.com/tools/ATZB-A-233-XPRO.aspx

[46] Real Time Engineers Ltd. (2013) *FreeRTOS*. [Online]. Available: http://www.freertos.org/

[47] ChaN. *FatFs - Generic FAT File System Module*. [Online]. Available: http://elm-chan.org/fsw/ff/00index_e.html

[48] D. Thain. (2013) *wavfile: A Simple Sound Library*. [Online]. Available: http://www.cse.nd.edu/~dthain/courses/cse20211/fall2013/wavfile/

[49] V. Lazzarini, "Soundfiles, Soundfile Formats, and libsndfile," in *The Audio Programming Book*, Richard Boulanger and Victor Lazzarini, Eds. Cambridge, United States of America: The MIT Press, 2011, ch. Appendix C, pp. 739-769.

[50] M. van Eerde. (2009, August) *How to calculate a sine sweep - the right way*. [Online]. Available: http://blogs.msdn.com/b/matthew_van_eerde/archive/2009/08/07/how-to-calculate-a-sine-sweep-the-right-way.aspx

[51] S. Johansson, "Active Control of Propeller-Induced Noise in Aircraft: Algorithms & Methods," Dept. Telecommun. Signal Process. Blekinge Inst. Technol., Ronneby, Sweden, Ph.D. Dissertation 2000.

[52] SHARP Corporation. (2005, October) *GP1A57HRJ00F Photointerrupter Datasheet*. [Online]. Available: https://www.sparkfun.com/datasheets/Components/GP1A57HRJ00F.pdf

[53] AR Drone Flyers. (2012, January) *AR.Drone Motor*. [Online]. Available: http://www.ardrone-flyers.com/wiki/Motor

[54] Wireshark Found. (2014, April) *Wireshark*. [Online]. Available: http://www.wireshark.org/

[55] VLSI Solution. (2014, March) *Products*. [Online]. Available: http://www.vlsi.fi/en/products.html

[56] Atmel Corp. (2013, February) *Atmel AVR2025: IEEE 802.15.4 MAC Software Package - User Guide*. [Online]. Available: http://www.atmel.com/images/doc8412.pdf

[57] Digi-Key Corp. (2014) *OPB9-W Photo Interrupter*. [Online]. Available: http://media.digikey.com/Photos/TT%20Electronics-Optek%20Photos/OPB9-W.jpg

# A Appendices

Included in the appendices are short pieces of the code included in the full digital attachment submitted with this thesis. The code that is included here is meant to illustrate and supplement the information that is given in the previous sections. However, it is not meant to be complete code, since the complete code is many times the size of these sections.

There are four sections of appendices included:

- MATLAB Code – This includes the code used to generate reference sine sweep signals as well as some of the base code used to analyze the frequency and impulse responses.
- HTML Code – This code demonstrates the bookmarklet used with the iPod Touch app.
- C++ Code – A sample program is included that demonstrates how to read from and write to the SAM4L USB port.
- C Code - A very small sample of the code from the SAM4L ground and remote nodes is included. The samples focus on those parts most important for audio, such as synchronization, ADC recording, and DAC playback.

The digital attachment includes the full versions of all the code, along with pre-compiled binaries/executables that can be used immediately. For any questions about code that is not included in these appendices, please see the digital attachment. There is a *ReadMe.txt* file included in the root folder of the submission for further clarification on file locations.

## A.1 MATLAB Code

The first MATLAB script is the code used to generate the sine sweep signals. It is setup for a 500-10,000 Hz sweep over two seconds at a sample rate of 44,100 Hz, but these values can be altered. The second script is the function used to generate frequency plots for an input WAV file. The final script is a function used to generate the impulse response for a recorded WAV file, given a reference signal.

The digital attachment also includes two additional scripts based on the second and third scripts shown here. These additional scripts plot two frequency plots or impulse responses simultaneously, for comparison purposes.

### A.1.1 Signal Generation

```
% Sine Sweep Signal Generation
%
% Based on a script from Prof. Svensson
%
% Paramters:
%  fstart - Starting frequency (Hz)
%  fend - Ending frequency (Hz)
%  Tsweep - Duration (seconds)
%  fs - Sample rate (samples/second)

fstart = 500;
fend = 10000;
Tsweep = 2;
fs = 44100;

tvec = [0:Tsweep*fs-1]/fs;
b = log(fend/fstart);
sig = 0.5*sin(2*pi*fstart*Tsweep/b*(exp(tvec/Tsweep*b)-1));

filename = ['Sweep_',int2str(fstart),'_',int2str(fend),'Hz.wav'];

wavwrite(sig,fs,filename);
```

### A.1.2 Analysis Code

#### *Frequency Plot*

```
function [ output_args ] = fplot( file )
%fplot Displays an FFT spectrum plot of the file
%
% Paramaters:
%  file - Input file to display the spectrum of

[y fs]=wavread(file);

nfft_plot = length(y);
TF_useful = fft(y,nfft_plot);
fvec = fs/nfft_plot*[0:nfft_plot/2-1];

semilogx(fvec,smooth(20*log10(abs(TF_useful(1:nfft_plot/2))),101))

title(file);
xlabel('Frequency');
ylabel('Amplitude (dB)');
```

```
end
```

### *IR Response Plot*

```
function [ output_args ] =
ir_response(lsp_file,mic_file,window_size,plot_title,smoothing_constant)
%ir_response Computes and displays the impulse response of a recording
%   Computes and displays the impulse response of a recording
%
% Parameters:
%  lsp_file: Loudspeaker Output File (WAV)
%  mic_file: Microphone Recording File (WAV)
%  window_size: IR window size to use
%  plot_title: Title of the plot
%  smoothing_constant: Constant to use when smoothing plot

%Read input files
[sig,fs] = wavread(lsp_file);
[micrec, fs_rec] = wavread(mic_file);

%Check that Sampling Frequencies are equal
if fs ~= fs_rec
    error('Sample Frequency Mismatch');
end

%Make sure we are only using mono files
sig = sig(:,1);
micrec = micrec(:,1);

%Add default values
if nargin < 5
    smoothing_constant = 1;
end
if nargin < 4
    plot_title = mic_file;
end
if nargin < 3
    window_size = size(micrec,1);
end

%Begin analysis
nfft_analyze = 2^20;
TF_system = fft(micrec,nfft_analyze)./fft(sig,nfft_analyze);
ir_system = real(ifft(TF_system));
ivdirectsoundwindow = [1:window_size];
ir_useful = ir_system(ivdirectsoundwindow);

%write the IR out
wavwrite(ir_useful,fs,plot_title);

%Plot Time Domain IR
figure(1)
plot(ir_useful)
title(strcat('Time Domain IR - ',plot_title));
xlabel('Sample');
ylabel('Response');
```

```
%Plot Frequency Domain IR
nfft_plot = 16384;
TF_useful = fft(ir_useful,nfft_plot);
fvec = fs/nfft_plot*[0:nfft_plot/2-1];
figure(2)
semilogx(fvec,smooth(20*log10(abs(TF_useful(1:nfft_plot/2))),1))
title(strcat('Frequency Domain IR - ',plot_title));
xlabel('Frequency');
ylabel('Response (db scale)');
axis([0 100000 -100 20]);

figure(3)
semilogx(fvec,smooth(20*log10(abs(TF_useful(1:nfft_plot/2))),smoothing_con
stant))
title(strcat('Frequency Domain IR - ',plot_title,'(smoothed)'));
xlabel('Frequency');
ylabel('Response (dB scale)');
axis([0 100000 -100 20]);

%Save images
%saveas(1,strcat('Time Domain IR - ',plot_title,'.png'));
%saveas(2,strcat('Frequency Domain IR - ',plot_title,'.png'));
%saveas(3,strcat('Frequency Domain IR - ',plot_title,'(smoothed).png'));

end
```

## A.2 HTML Code

The following HTML5/Javascript code uses the Web Audio API to generate a sine sweep signal and simultaneously control the AirBeam app used on the Apple iPod Touch.

```
/* AirBeam Synchronization Bookmarklet */
/* Usage: Click on the bookmarklet when AirBeam page is open */
/* Setup: Remove all comments and bookmarks then save as a bookmarklet */
javascript:(
  function(){
    /* Request sweep information from user */
    var start = parseInt( prompt( "Enter Start Delay" , "0" ) );
    var startfreq = parseInt( prompt( "Enter Start Frequency" , "500" ) );
    var endfreq = parseInt( prompt( "Enter End Frequency" , "10000" ) );
    var duration = parseInt( prompt( "Enter Play Duration" , "10" ) );
    var reverb = parseInt( prompt( "Enter Reverb Record Time" , "5" ) );

    /* Setup Web Audio API */
    var context = new webkitAudioContext();
    var oscillator = context.createOscillator();

    /* Configure Oscillator Node */
    oscillator.type = 0;
    oscillator.frequency.value = startfreq;
    oscillator.connect( context.destination );
    oscillator.frequency.setValueAtTime( startfreq , start );
    oscillator.frequency.exponentialRampToValueAtTime( endfreq ,
        start + duration );
    oscillator.start( start );
    oscillator.stop( start + duration );

    /* Everything is ready, click the record button */
    document.getElementById( "button_record" ).click();

    /* Setup timers to press the AirBeam stop button */
    var myStopVar = setInterval( function(){ myStopTimer() } , 100 );

    function myStopTimer() {
      if( context.currentTime >= ( start + duration ) ) {
        myStopStopFunction();
        setTimeout( function() {
          document.getElementById("button_record").click();
        }, reverb * 1000);
      }
    }

    function myStopStopFunction() {
      clearInterval( myStopVar );
    }
  }
)();
```

## A.3 C++ Code

The following sample code demonstrates how to communicate with the SAM4L USB port from a PC using the AtUsbHid driver provided by Atmel.  This sample can read status codes and inform the ground node to begin an audio recording.

```
/**
* File: main.cpp
* Creator: Jonathan Klapel
* Date: 08 June 2014
* Description: USB sample application
*
* APPLICATION USE
* Use:
* - Using the USB cable, connect the SAM4L USB port to the PC
* - The PC should automatically install a generic HID driver
* - Run the application and follow the console menu
*
* Notes:
*  Simple code examples are given here to demonstrate how to use the USB
*  library.  For more details on the Atmel AtUsbHid driver, see
*  http://www.atmel.com/dyn/resources/prod_documents/doc7645.pdf
*  and http://www.atmel.com/dyn/resources/prod_documents/AVR153.zip
*/

#include <iostream>
#include <stdio.h>
#include <windows.h>
#include <thread>
#include "AtUsbHid.h"

using namespace std;

// USB HID device Vendor ID and Product ID.
#define VID 0x03EB
#define PID_1 0x2402
#define PID_2 0x2013

// USB message types
#define MSG_HELO 0
#define MSG_AUDIO_REQUEST 1
#define MSG_AUDIO_CONFIRM 2
#define MSG_STATUS 3

bool running = false;

/**
 * USB Reads Function
 *
 * Reads from the USB and reacts to different message types
 */
void usb_reads(void) {
  unsigned char sbuffer[256];
  // Continue reading until application exit
  while (running) {
    // Read from the USB
    if (DYNCALL(readData(sbuffer)) != 0) {
      // React to message
      switch (sbuffer[0]) {
      case MSG_HELO:
        cout << "Received HELO Request" << endl;
        break;
      case MSG_AUDIO_REQUEST:
        cout << "Received Audio Request" << endl;
        break;
      case MSG_AUDIO_CONFIRM:
        cout << "Received Audio Confirm" << endl;
        break;
      case MSG_STATUS:
        cout << "Received Connection Status" << endl;
        if (sbuffer[1] == 2 || sbuffer[1] == 3)
          cout << "Wireless Connection Established" << endl;
        else
          cout << "Wireless Connection Not Established" << endl;
        if (sbuffer[2] == 1)
          cout << "Nodes Synced" << endl;
        else
          cout << "Nodes Not Synced" << endl;
        break;
      }
    }
  }
}

int main(void);
/**
 * Main Function
 *
 * Starting function for the application.  Sets up and carries out USB
calls
 */
int main(void) {
  // Write Buufer used to set data_out
  unsigned char data_out[9];

  // Handle to AtUsbHid.dll
  HINSTANCE hLib = NULL;  // Handle to our USB HID DLL.

  printf("Atmel USB HID Library Test Program\n\n");

  // Explicitely load the AtUsbHid library.
  printf("Loading USB HID Dll.\n");
  hLib = LoadLibraryA(AT_USB_HID_DLL);
  if (hLib != NULL) {
    printf("USB HID Dll loaded\n");
  } else {
    if (GetLastError() == ERROR_MOD_NOT_FOUND) {
      printf("Error: Could not find the Atmel USB HID Dll.\n");
      printf("       Please update the PATH variable.\n");
    } else {
      printf("Error: While opening Dll.\n");
    }
    return 1;
  }

  // Get USB HID library functions addresses.
  printf("Loading all Dll functions.\n");
```

```
if (loadFuncPointers(hLib)) {
  printf("All function of the Dll has been loaded\n");
}
else {
  printf("Error: Could not find load function of the Dll.\n");
  return 1;
}

// Open our USB HID device.
printf("Opening USB HID device with Vendor ID= 0x%04X and Product
   ID=0x%04X\n", VID, PID_1);
if (DYNCALL(findHidDevice)(VID, PID_1)) {
  printf("USB HID device  VID=0x%04X, PID=0x%04X opened.\n", VID,
    PID_1);
}
else {
  switch (GetLastError()) {
  case ERROR_USB_DEVICE_NOT_FOUND:
    printf("Error: Could not open the device.\n");
    break;
  case ERROR_USB_DEVICE_NO_CAPABILITIES:
    printf("Error: Could not get USB device capabilities.\n");
    break;
  default:
    printf("Error: While opening device.\n");
    break;
  }
}

// Print Some information about the connected device

printf("USB HID Input   Buffer size is %dByte.\n",
  DYNCALL(getInputReportLength()));
printf("USB HID Output  Buffer size is %dByte.\n",
  DYNCALL(getOutputReportLength()));
printf("USB HID Feature Buffer size is %dByte.\n",
  DYNCALL(getFeatureReportLength()));

// Start USB Read thread
running = true;

thread t1(usb_reads);
int menu_choice = 0;

// Display menu and react to choices.  Exit on "4"
while (menu_choice != 4) {

  // Output Menu
  cout << endl << "Menu options" << endl;
  cout << "0: HELO Request" << endl;
  cout << "1: Audio Request" << endl;
  cout << "2: Audio Confirm" << endl;
  cout << "3: Connection Status Request" << endl;
  cout << "4: Exit" << endl;
  cout << "Enter menu number choice: ";

  // Read in menu selection choice
  cin >> menu_choice;
```

```
    LARGE_INTEGER lg;
    SYSTEMTIME st;

    // React to choice
    switch (menu_choice) {
      // 0: HELO Request - Send current system time
      case MSG_HELO:
        data_out[0] = 0;
        GetSystemTime(&st);
        lg.QuadPart = ((st.wHour * 60 + st.wMinute) * 60 + st.wSecond) *
          1000 + st.wMilliseconds;
        data_out[1] = (lg.QuadPart >> 0) & 0xff;
        data_out[2] = (lg.QuadPart >> 8) & 0xff;
        data_out[3] = (lg.QuadPart >> 16) & 0xff;
        data_out[4] = (lg.QuadPart >> 24) & 0xff;
        data_out[5] = (lg.QuadPart >> 32) & 0xff;
        data_out[6] = (lg.QuadPart >> 40) & 0xff;
        data_out[7] = (lg.QuadPart >> 48) & 0xff;
        DYNCALL(writeData)(data_out);
        break;
      // 1: Audio Request - Request an audio measurement
      case MSG_AUDIO_REQUEST:
        data_out[0] = 1;
        data_out[1] = (500 >> 0) & 0xff;
        data_out[2] = (500 >> 8) & 0xff;
        data_out[3] = (10000 >> 0) & 0xff;
        data_out[4] = (10000 >> 8) & 0xff;
        data_out[5] = 5;
        data_out[6] = 5;
        DYNCALL(writeData)(data_out);
        break;
      // 2: Audio Confirm - Request confirmation of audio
      case MSG_AUDIO_CONFIRM:
        data_out[0] = 2;
        DYNCALL(writeData)(data_out);
        break;
      // 3: Connection Status Request - Request current node status
      case MSG_STATUS:
        data_out[0] = 3;
        DYNCALL(writeData)(data_out);
        break;
    }
  }

  // Finished application, tell read thread to stop
  running = false;
  t1.join();

  // Close our USB HID device.
  DYNCALL(closeDevice)();
  printf(">>> USB HID device VID=0x%04X closed.\n", VID);

  // Wait thread to be close before freeing DLL
  Sleep(1000);
  FreeLibrary(hLib);
  printf(">>> Please press a key to exit");
  getchar();
  return 1;
}
```

## A.4 C Code

The following code is excerpted from the ground and remote nodes. These are not the complete c files and only include information that is useful for understanding how the library functions. Additional functions and libraries are used in the full files, which can be found in the digital attachment.

### A.4.1 Time

```
/**
 * File: time.c
 * Creator: Jonathan Klapel
 * Date: 08 June 2014
 * Description: Performs the synchronization and time-keeping
 */

#include "time.h"

/* Time variables */
static uint32_t sync_start = 0;
static uint32_t sync_end = 0;
static uint32_t ack_start = 0;
static uint32_t ack_end = 0;
static uint32_t prev_ack_end = 0;

/* Delta variables */
static int32_t delta = 0;
static int32_t transmit = 0;
static double delta_change = 0;

/* Synchronization boolean */
static uint8_t synced = 0;

/**
 * Calculate Offsets Function
 *
 * Performs TPSN synchronization calculations and sets synchronization
variables
 */
void calculate_offsets(void) {
  static uint8_t sync_count = 0;
  int32_t old_delta = delta;
  int32_t temp_delta = ((int32_t)(sync_end-sync_start)-(int32_t)(ack_end-
    ack_start))/2;

  if (old_delta == 0) {
    delta = temp_delta;
    delta_change = (float)(delta -old_delta)/(float)(ack_end-
      prev_ack_end);
    transmit = ((sync_end-sync_start)+(ack_end-ack_start))/2;
  } else if(abs(old_delta-temp_delta) < abs(delta_change*(ack_end-
      prev_ack_end))*1.5) {
    sync_count++;
    if (sync_count >= 2) {
      synced = 1;
      sync_count = 2;
    }
    delta = temp_delta;
    delta_change = (float)(delta -old_delta)/(float)(ack_end-
      prev_ack_end);
    transmit = ((sync_end-sync_start)+(ack_end-ack_start))/2;
  } else {
    printf("Missed delta max\r\n");
    synced = 0;
    ack_end = prev_ack_end;
    delta_change = delta_change*1.5;
  }
}

/**
 * Get Global Time Function
 *
 * Gets the current time on the coordinator node
 */
uint32_t get_global_time(void) {
  return (uint32_t)((int32_t)sw_timer_get_time() + delta +
    get_estimated_delta_change(sw_timer_get_time()));
}

/**
 * Get Estimated Delta Change Function
 *
 * Returns an estimate of how much the delta has changed since the last
 *  ACK timestamp
 */
int32_t get_estimated_delta_change(uint32_t timestamp) {
  return (int32_t)(delta_change*(float)(timestamp-ack_end));
}
```

### A.4.2 Zigbit

```
/**
 * File: zigbit.c
 * Creator: Jonathan Klapel
 * Date: 08 June 2014
 * Description: Communicates over the Atmel Zigbit Module
 */

#include "zigbit.h"

/**
 * Send Message Function
 *
 * Sends a message to the other node via Zigbit
 *
 * Takes the Message Type, the length of the message, and a pointer to the
 * data
 */
uint8_t send_message(msg_type msgtype, uint8_t length, uint8_t * data) {
  node_status = TRANSMITTING;
  uint8_t src_addr_mode;
  wpan_addr_spec_t dst_addr;
  uint8_t payload[length+1];
  static uint8_t msduHandle = 0;

  src_addr_mode = WPAN_ADDRMODE_SHORT;
  dst_addr.AddrMode = WPAN_ADDRMODE_SHORT;
  dst_addr.PANId = DEFAULT_PAN_ID;

  payload[0] = msgtype;
  for (uint8_t i = 0; i < length; i++) {
    payload[i+1] = data[i];
  }

  /* Perform different tasks depending on who this is */
#if (DEVICE == 1)
  ADDR_COPY_DST_SRC_16(dst_addr.Addr.short_address,coord_addr.short_addr);
#endif
#if (COORDINATOR == 1)
  ADDR_COPY_DST_SRC_16(dst_addr.Addr.short_address,device_list[0].short_ad
    dr);
#endif

  msduHandle++;

  /* Perform the actual send without interruption */
  taskENTER_CRITICAL();
  bool success = wpan_mcps_data_req( src_addr_mode, &dst_addr, length+1,
    &payload, msduHandle, WPAN_TXOPT_ACK);
  taskEXIT_CRITICAL();

  return msduHandle;
}

/**
 * User MCPS Data Confirm Function
 *
 * Called by Zigbit library on successful send
 */
#if defined(ENABLE_TSTAMP)
void usr_mcps_data_conf(uint8_t msduHandle,
    uint8_t status,
    uint32_t Timestamp)
#else
void usr_mcps_data_conf(uint8_t msduHandle,
    uint8_t status)
#endif
{
  node_status = IDLE;

  if (status == MAC_SUCCESS) {
    data_transmitted(msduHandle, status, Timestamp);
    LED_Off(LED_DATA);
  }

  /* Keep compiler happy. */
  msduHandle = msduHandle;
}

/**
 * User MCPS Data Indicate Function
 *
 * Called by Zigbit library on data reception
 */
void usr_mcps_data_ind(wpan_addr_spec_t *SrcAddrSpec,
    wpan_addr_spec_t *DstAddrSpec,
    uint8_t msduLength,
    uint8_t *msdu,
    uint8_t mpduLinkQuality,
#ifdef ENABLE_TSTAMP
    uint8_t DSN,
    uint32_t Timestamp)
#else
    uint8_t DSN)
#endif  /* ENABLE_TSTAMP */
{
  data_received(msduLength,msdu,Timestamp);

  LED_On(LED_DATA);

  /* Start a timer switching off the LED */
  sw_timer_start(TIMER_LED_OFF, 500000, SW_TIMEOUT_RELATIVE,
    (FUNC_PTR)data_exchange_led_off_cb, NULL);

  /* Keep compiler happy. */
  SrcAddrSpec = SrcAddrSpec;
  DstAddrSpec = DstAddrSpec;
  msduLength = msduLength;
  msdu = msdu;
  mpduLinkQuality = mpduLinkQuality;
  DSN = DSN;
}

/**
 * User MLME Associate Confirm Function
 *
 * Confirmation of correct association with another node
 */
#if (MAC_ASSOCIATION_REQUEST_CONFIRM == 1)
void usr_mlme_associate_conf(uint16_t AssocShortAddress,
    uint8_t status)
{
  if (status == MAC_SUCCESS) {
    sw_timer_stop(TIMER_LED_OFF);
    LED_On(LED_NWK_SETUP);
    node_status = IDLE;
    wireless_ready();
  } else {
    /* Something went wrong; restart */
    wpan_mlme_reset_req(true);
  }

  /* Keep compiler happy. */
  AssocShortAddress = AssocShortAddress;
}

#endif  /* (MAC_ASSOCIATION_REQUEST_CONFIRM == 1) */

/**
 * User MLME Communication Status Indicator Function
 *
 * Gives the status after successful communication between nodes
 */
#if ((MAC_ORPHAN_INDICATION_RESPONSE == 1) || \
  (MAC_ASSOCIATION_INDICATION_RESPONSE == 1))
void usr_mlme_comm_status_ind(wpan_addr_spec_t *SrcAddrSpec,
    wpan_addr_spec_t *DstAddrSpec,
    uint8_t status)
{
  if (status == MAC_SUCCESS) {
    node_status = IDLE;
    wireless_ready();
  }

  /* Keep compiler happy. */
  SrcAddrSpec = SrcAddrSpec;
  DstAddrSpec = DstAddrSpec;
}

#endif

/**
 * User MLME Scan Configuration
 *
 * Scans for a broadcast network
 */
#if ((MAC_SCAN_ED_REQUEST_CONFIRM == 1)      ||  \
  (MAC_SCAN_ACTIVE_REQUEST_CONFIRM == 1)  || \
  (MAC_SCAN_PASSIVE_REQUEST_CONFIRM == 1) || \
  (MAC_SCAN_ORPHAN_REQUEST_CONFIRM == 1))
void usr_mlme_scan_conf(uint8_t status,
    uint8_t ScanType,
    uint8_t ChannelPage,
    uint32_t UnscannedChannels,
    uint8_t ResultListSize,
    void *ResultList) {
  if (status == MAC_SUCCESS) {
/* Different functions for each device */
#if (DEVICE == 1)
printf("Scanning for coordinator\r\n");
    wpan_pandescriptor_t *coordinator;
    uint8_t i;

    /*
     * Analyze the Result List
     * Assume that the first entry of the result list is our coodinator.
     */
    coordinator = (wpan_pandescriptor_t *)ResultList;
    for (i = 0; i < ResultListSize; i++) {
      /*
       * Check if the PAN descriptor belongs to our coordinator.
       * Check if coordinator allows association.
       */
      if ((coordinator->LogicalChannel == current_channel) &&
          (coordinator->ChannelPage == current_channel_page) &&
          (coordinator->CoordAddrSpec.PANId == DEFAULT_PAN_ID) &&
          ((coordinator->SuperframeSpec &  ((uint16_t)1 <<
            ASSOC_PERMIT_BIT_POS)) == ((uint16_t)1 <<
          ASSOC_PERMIT_BIT_POS))) {

        /* Store the coordinator's address */
        if (coordinator->CoordAddrSpec.AddrMode == WPAN_ADDRMODE_SHORT) {
          ADDR_COPY_DST_SRC_16(coord_addr.short_addr,
            coordinator->CoordAddrSpec.Addr.short_address);
        } else if (coordinator->CoordAddrSpec.AddrMode ==
          WPAN_ADDRMODE_LONG) {
          ADDR_COPY_DST_SRC_64(coord_addr.ieee_addr,
            coordinator->CoordAddrSpec.Addr.long_address);
        } else {
          /* Something went wrong; restart */
          wpan_mlme_reset_req(true);
          return;
        }

        /* Associate to our coordinator */
        wpan_mlme_associate_req(coordinator->LogicalChannel,
          coordinator->ChannelPage, &(coordinator->CoordAddrSpec),
          WPAN_CAP_ALLOCADDRESS);
        return;
      }

      /* Get the next PAN descriptor */
      coordinator++;
    }

    /*
     * If here, the result list does not contain our expected coordinator.
     * Let's scan again.
     */
    wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE, SCAN_CHANNEL, SCAN_DURATION,
      current_channel_page);
#endif
#if (COORDINATOR == 1)
      /*
```

```
        * Set the short address of this node.
        */
       uint8_t short_addr[2];
       short_addr[0] = (uint8_t)COORD_SHORT_ADDR; /* low byte */
       short_addr[1] = (uint8_t)(COORD_SHORT_ADDR >> 8); /* high byte */
       wpan_mlme_set_req(macShortAddress, short_addr);
#endif
  } else if (status == MAC_NO_BEACON) {
#if (DEVICE == 1)
     /*
      * No beacon is received; no coordiantor is located.
      * Scan again, but used longer scan duration.
      */
     wpan_mlme_scan_req(MLME_SCAN_TYPE_ACTIVE,SCAN_CHANNEL,SCAN_DURATION,
       current_channel_page);
#endif
#if (COORDINATOR == 1)
       uint8_t short_addr[2];
       short_addr[0] = (uint8_t)COORD_SHORT_ADDR; /* low byte */
       short_addr[1] = (uint8_t)(COORD_SHORT_ADDR >> 8); /* high byte */
       wpan_mlme_set_req(macShortAddress, short_addr);
#endif
  } else {
    /* Something went wrong; restart */
    wpan_mlme_reset_req(true);
  }

  /* Keep compiler happy. */
  ScanType = ScanType;
  UnscannedChannels = UnscannedChannels;
  ChannelPage = ChannelPage;
}

#endif
```

### A.4.3 ADC

```
/**
 * File: adc.c
 * Creator: Jonathan Klapel
 * Date: 29 May 2014
 * Description: ADC readings and writing to USB
 */

#include "adc.h"

/** Size of buffers from ADC */
#define BUFFER_LENGTH 512

/** Conversion values */
uint16_t g_adc_sample_data[10];
uint16_t audio_buf1[BUFFER_LENGTH];
uint16_t audio_buf2[BUFFER_LENGTH];
uint16_t audio_buf3[BUFFER_LENGTH];
uint8_t buffer1_full = 0;
uint8_t buffer2_full = 0;
uint8_t buffer3_full = 0;
uint32_t overload_count = 0;
uint8_t first_loop = 1;
uint32_t position = 0;

/** ADC instance */
struct adc_dev_inst g_adc_inst;

/* Time variables */
static uint32_t start_time;
static uint32_t end_time;
static uint32_t recording_duration;
static uint32_t playback_start;
static uint32_t time;
uint32_t sample_count = 0;
uint32_t actual_start_time = 0;
uint32_t actual_end_time = 0;
uint32_t end_time_offset = 0;

/* Audio file name */
char file_name[22];

/** ADC CDMA config value */
struct adc_cdma_first_config g_adc_cdma_first_cfg[1] = {
  {
    /* Select Vref for shift cycle */
    .zoomrange = ADC_ZOOMRANGE_0,
    /* Neg Input Channel 3 */
    .muxneg = ADC_MUXNEG_3,
    /* Pos Input Channel 2 */
    .muxpos = ADC_MUXPOS_2,
    /* Enables the internal voltage sources */
    .internal = ADC_INTERNAL_0,
    /* Disables the ADC gain error reduction */
    .gcomp = ADC_GCOMP_DIS,
    /* Disables the HWLA mode */
    .hwla = ADC_HWLA_EN,
    /* 12-bits resolution */
    .res = ADC_RES_12_BIT,
    /* Enables the differential mode */
    .bipolar = ADC_BIPOLAR_DIFFERENTIAL,
    /* Gain factor is 16x */
    .gain = ADC_GAIN_16X,
    .strig = 1,
    .tss = 1,
    .enstup = 1,
    .dw = 0
  }
};
/* PDCA Config values */
struct adc_pdca_config g_adc_pdca_cfg = {
  /* ADC Window Mode */
  .wm = false,
  /* ADC Nb channel */
  .nb_channels = 1,
  /* ADC Buffer */
  .buffer = g_adc_sample_data,
  /* ADC PDC Rx Channel */
  .pdc_rx_channel = CONFIG_ADC_PDCA_RX_CHANNEL,
  /* ADC PDC Tx Channel */
  .pdc_tx_channel = CONFIG_ADC_PDCA_TX_CHANNEL,
  /* ADC CDMA Configuration Structure */
  .cdma_cfg = (void *)&g_adc_cdma_first_cfg[0]
};

/**
 * Callback function for PDCA interrupt
 */
void pdca_transfer_done(enum pdca_channel_status status)
{
  /* Get PDCA RX channel status and check if PDCA transfer complete */
  if (status == PDCA_CH_TRANSFER_COMPLETED) {
    /* Time is before required start time, so pause for one moment */
    if (time < start_time) {
      actual_start_time = sw_timer_get_time();
      pdca_channel_write_load(CONFIG_ADC_PDCA_RX_CHANNEL,
        g_adc_sample_data, 1);
      first_loop = 1;
    /* Time is between start and end time, so data should be saved */
    } else if (start_time < time && (end_time+end_time_offset) > time) {
      /* Loop through three buffers filling up each */
      if (position == 0) {
        pdca_channel_write_load(CONFIG_ADC_PDCA_RX_CHANNEL, audio_buf2,
          BUFFER_LENGTH);
        buffer1_full = 1;
        position = 1;
      } else if (position == 1) {
        pdca_channel_write_load(CONFIG_ADC_PDCA_RX_CHANNEL, audio_buf3,
          BUFFER_LENGTH);
        buffer2_full = 1;
        position = 2;
      } else if (position == 2) {
        pdca_channel_write_load(CONFIG_ADC_PDCA_RX_CHANNEL, audio_buf1,
          BUFFER_LENGTH);
        buffer3_full = 1;
        position = 0;
        }

        /* Keep track of how many samples are supposed to be saved */
        sample_count += BUFFER_LENGTH;

        /* On first loop through, there are no useful values in buffers */
        if (first_loop == 1) {
          buffer1_full = 0;
          buffer2_full = 0;
          buffer3_full = 0;
          first_loop = 0;
          sample_count -= BUFFER_LENGTH;
        }

        /* Check if we are not writing fast enough, */
        /* resulting in all buffers being filled */
        if(buffer1_full == 1 && buffer2_full == 1 && buffer3_full == 1) {
          overload_count++;
        }
      /* Time is past required end time so end ADC */
      } else if (actual_end_time == 0) {
        pdca_channel_write_load(CONFIG_ADC_PDCA_RX_CHANNEL,
          g_adc_sample_data, 10);

        /* Note actual end time */
        actual_end_time = sw_timer_get_time();

        /* Disable PDCA */
        pdca_channel_disable_interrupt( CONFIG_ADC_PDCA_RX_CHANNEL,
          PDCA_IER_TRC);
        pdca_channel_disable(CONFIG_ADC_PDCA_TX_CHANNEL);
        pdca_channel_disable(CONFIG_ADC_PDCA_RX_CHANNEL);

        /* Disable ADC */
        adc_stop_itimer(&g_adc_inst);
      }
    }
}

/**
 * ADC PDCA Set Config
 *
 * A rewrite of the base ADC PDCA setup in adcife.h.  This version
 * allows the ADC to record multiple values to the buffer before
 * interrupting the CPU.
 */
void adc_pdca_set_config2(struct adc_pdca_config *cfg)
{
  /* Enable PDCA module clock */
  pdca_enable(PDCA);

  /* PDCA channel options */
  pdca_channel_config_t PDCA_TX_CONFIGS = {
    /* memory address */
    .addr = (void *)cfg->cdma_cfg,
    /* select peripheral */
    .pid = ADCIFE_PDCA_ID_TX,
    /* transfer counter */
    .size = cfg->wm == true ? (cfg->nb_channels)*2 : cfg->nb_channels,
    /* next memory address */
    .r_addr = NULL,
    /* next transfer counter */
    .r_size = 0,
    /* select size of the transfer */
    .transfer_size = PDCA_MR_SIZE_WORD,
    /* set ring buffer */
    .ring = true
  };
  pdca_channel_config_t PDCA_RX_CONFIGS = {
    /* memory address */
    .addr = (void *)cfg->buffer,
    /* select peripheral */
    .pid = ADCIFE_PDCA_ID_RX,
    /* transfer counter */
    .size = 1,
    /* next memory address */
    .r_addr = NULL,
    /* next transfer counter */
    .r_size = 0,
    /* select size of the transfer */
    .transfer_size = PDCA_MR_SIZE_HALF_WORD
  };
  /* Init PDCA channel with the pdca_options. */
  pdca_channel_set_config(cfg->pdc_tx_channel, &PDCA_TX_CONFIGS);
  pdca_channel_set_config(cfg->pdc_rx_channel, &PDCA_RX_CONFIGS);
  /* Enable PDCA channel */
  pdca_channel_enable(cfg->pdc_tx_channel);
  pdca_channel_enable(cfg->pdc_rx_channel);
}

/**
 * Start ADC function
 *
 * Initialize ADC, set clock and timing, and set ADC to given mode.
 */
void start_adc(void)
{
  struct adc_config adc_cfg = {
    .prescal = ADC_PRESCAL_DIV4,
    .clksel = ADC_CLKSEL_APBCLK,
    .speed = ADC_SPEED_300K,
    /* ADC Reference voltage is 0.625*VCC */
    .refsel = ADC_REFSEL_1,
    .start_up = CONFIG_ADC_STARTUP
  };

  /* Calculate ADC internal timer value */
  uint32_t internal_timer_max_count = (sysclk_get_pba_hz())/((uint32_t)(4
    << adc_cfg.prescal)*(uint32_t)WAVFILE_SAMPLES_PER_SECOND)-40;

  /* Initialize and enable ADC */
  if(adc_init(&g_adc_inst, ADCIFE, &adc_cfg) != STATUS_OK) {
    puts("-F- ADC Init Fail!\n\r");
    while(1);
  }
  if(adc_enable(&g_adc_inst) != STATUS_OK) {
    puts("-F- ADC Enable Fail!\n\r");
    while(1);
  }

  /* Configure PDCA */
  adc_disable_interrupt(&g_adc_inst, ADC_SEQ_SEOC);
  adc_pdca_set_config2(&g_adc_pdca_cfg);

  pdca_channel_set_callback(CONFIG_ADC_PDCA_RX_CHANNEL,
    pdca_transfer_done, PDCA_0_IRQn, 1, PDCA_IER_TRC);
  pdca_channel_write_load(CONFIG_ADC_PDCA_TX_CHANNEL,
    g_adc_cdma_first_cfg, 1);
  pdca_channel_write_reload(CONFIG_ADC_PDCA_TX_CHANNEL,
    g_adc_cdma_first_cfg, 1);

  adc_configure_trigger(&g_adc_inst, ADC_TRIG_INTL_TIMER);
  adc_configure_itimer_period(&g_adc_inst,internal_timer_max_count);
  adc_start_itimer(&g_adc_inst);

  adc_configure_gain(&g_adc_inst, ADC_GAIN_16X);
}

/**
 * ADC Task function
 *
 * OS task used for starting and running audio recordings.
 */
void adc_task(void) {
  printf("ADC Task Started\r\n");

  static FIL audio_file_object;
  actual_start_time = 0;
  actual_end_time = 0;
  char str_time[10];

  time = sw_timer_get_time();

  strcpy(file_name, "0:");
  strcat(file_name, "audio_");
  strncat(file_name,u32_itoa(time,str_time),10);
  strcat(file_name, ".wav");

  /* Open initial WAV file */
  wavfile_open(&audio_file_object, &file_name,
    WAVFILE_SAMPLES_PER_SECOND*5+BUFFER_LENGTH*3);

  /* Add buffer time to end of recording */
  end_time_offset = 1000*1000*BUFFER_LENGTH*3/WAVFILE_SAMPLES_PER_SECOND;

  /* Check to make sure we aren't late starting recording */
  if(time > start_time) {
    printf("Missed recording start time by %u ms\r\n",(time-start_time));
  }

  /* Make sure no one else interrupts us */
  taskENTER_CRITICAL();

  /* Start the ADC processing data from mic */
  start_adc();

  uint32_t bytes_written = 0;
  /* Write data to the USB while there is potentially data to be written
*/
  while ((end_time+end_time_offset) > time || buffer1_full == 1 ||
      buffer2_full == 1 || buffer3_full == 1) {
    if (buffer1_full == 1) {
      uint16_t i = 0;
      while (i < BUFFER_LENGTH) {
        audio_buf1[i] ^= 0x8000;
        i++;
      }
      bytes_written +=  wavfile_write( &audio_file_object, audio_buf1,
        BUFFER_LENGTH);
      buffer1_full = 0;
    }
    if (buffer2_full == 1) {
      uint16_t i = 0;
      while (i < BUFFER_LENGTH) {
        audio_buf2[i] ^= 0x8000;
        i++;
      }
      bytes_written += wavfile_write( &audio_file_object, audio_buf2,
        BUFFER_LENGTH);
      buffer2_full = 0;
    }
    if (buffer3_full == 1) {
      uint16_t i = 0;
      while (i < BUFFER_LENGTH) {
        audio_buf3[i] ^= 0x8000;
        i++;
      }
      bytes_written += wavfile_write( &audio_file_object, audio_buf3,
        BUFFER_LENGTH);
      buffer3_full = 0;
    }
    time = sw_timer_get_time();
  }

  /* Finished with everything.  Close up and leave. */
  taskEXIT_CRITICAL();

  printf("Finished.  Overloaded %u times. Samples: %u\tBytes Written:
    %u\r\n",overload_count,sample_count,bytes_written);

  wavfile_close(&audio_file_object);

  vTaskSuspend(NULL);
  adc_recording_crop();

  printf("Exiting audio recording\r\n");
  vTaskDelete(NULL);
}

/**
 * ADC Recording Crop
 *
 * Removes excess audio at the beginning and ending of files
 */
void adc_recording_crop(void) {
  char file_name2[22];
  char str_time[10];

  strcpy(file_name2, "0:");
  strcat(file_name2, "edaud_");
  strncat(file_name2,u32_itoa((start_time+1),str_time),10);
  strcat(file_name2, ".wav");

  /* wavwriter library takes care of the actual cropping */
  wavfile_crop(&file_name, &file_name2, actual_start_time, playback_start,
    recording_duration);
}
```

### A.4.4 DAC

```
/**
 * File: dac.c
 * Creator: Jonathan Klapel
 * Date: 08 June 2014
 * Description: DAC Audio output functions
 */

#include "dac.h"

/* Defines */
#define PDCA_ABDAC_CHANNEL0  0
#define PDCA_ABDAC_CHANNEL1  1
#define SAMPLE_RATE    44100
#define RECORD_INDICTATOR    PIN_PC08
#define SOUND_SAMPLES 0x100

/* Output buffer */
int16_t samples_buffer[SOUND_SAMPLES];

/* PDCA channel options */
static const pdca_channel_config_t pdca_abdac_config0 = {
  .addr = 0,   /* memory address */
  .pid = ABDACB_PDCA_ID_TX_CH0,     /* select peripheral */
  .size = 0,   /* transfer counter */
  .r_addr = 0,                  /* next memory address */
  .r_size = 0,                  /* next transfer counter */
  .ring = false,                /* disable ring buffer mode */
  .transfer_size = PDCA_MR_SIZE_HALF_WORD /* select size of the transfer
*/
};
static const pdca_channel_config_t pdca_abdac_config1 = {
  .addr = 0,   /* memory address */
  .pid = ABDACB_PDCA_ID_TX_CH1,     /* select peripheral */
  .size = 0,   /* transfer counter */
  .r_addr = 0,                  /* next memory address */
  .r_size = 0,                  /* next transfer counter */
  .ring = false,                /* disable ring buffer mode */
  .transfer_size = PDCA_MR_SIZE_HALF_WORD /* select size of the transfer
*/
};

/** ABDAC instance */
struct abdac_dev_inst g_abdac_inst;

/** ABDAC configuration */
struct abdac_config   g_abdac_cfg;

/* Output variables */
double fstart;
double fend;
double duration;
double record_duration;

uint32_t underrun = 0;

uint8_t audio_start_handle = 0;
```

```
/**
 * Audio Init Function
 *
 * Initializes the ABDAC and IO pin
 */
void audio_init(void) {
  /* Setup ABDAC */
  status_code_t status;

  /* Setup ADC */
  abdac_get_config_defaults(&g_abdac_cfg);
  g_abdac_cfg.sample_rate_hz = ABDAC_SAMPLE_RATE_44100;
  g_abdac_cfg.data_word_format = ABDAC_DATE_16BIT;
  g_abdac_cfg.mono = true;
  g_abdac_cfg.cmoc = false;
  status = abdac_init(&g_abdac_inst, ABDACB, &g_abdac_cfg);
  if (status != STATUS_OK) {
    printf("-- Initialization fails! --\r\n");
    while (1) {
    }
  }

  /* Setup IO Testing Pin */
  ioport_set_pin_dir(RECORD_INDICTATOR,IOPORT_DIR_OUTPUT);
  ioport_set_pin_level(RECORD_INDICTATOR,false);
}

/**
 * Audio Underrun Function
 *
 * Called when the DAC does not receive data fast enough
 */
void audio_underrun(void) {
  underrun++;
  abdac_clear_interrupt_flag(&g_abdac_inst, ABDAC_INTERRUPT_TXUR);
}

/**
 * Audio Task Function
 *
 * Primary function for performing audio measurements
 */
void audio_task(void) {
  printf("Audio Recording Task Started\r\n");
  uint8_t key;
  uint32_t i, count;
  uint8_t payload[8];

  printf("\tGenerated Signal Sweep: %u-%u over %u seconds\r\n",
    (uint32_t)fstart,(uint32_t)fend,(uint32_t)duration);
  printf("\tAudio recording duration: %u\r\n",(uint32_t)record_duration);

  /* Prep ABDAC */
  abdac_enable(&g_abdac_inst);
  abdac_clear_interrupt_flag(&g_abdac_inst, ABDAC_INTERRUPT_TXRDY);
  abdac_clear_interrupt_flag(&g_abdac_inst, ABDAC_INTERRUPT_TXUR);
  abdac_set_callback(&g_abdac_inst,ABDAC_INTERRUPT_TXUR,audio_underrun,1);
  /* Config PDCA module */
  pdca_enable(PDCA);
  pdca_channel_set_config(PDCA_ABDAC_CHANNEL0, &pdca_abdac_config0);
  pdca_channel_enable(PDCA_ABDAC_CHANNEL0);

  /* Set Volumes */
  abdac_set_volume0(&g_abdac_inst, false, 0x7FFF);
  abdac_set_volume1(&g_abdac_inst, false, 0x7FFF);

  /* Initial calculations */
  float_t inverse_sample_rate = 1/(float_t)SAMPLE_RATE;
  float32_t pow_val = (1/((float32_t)SAMPLE_RATE*duration));
  float32_t freq_change = (pow(fend,pow_val) -
    pow(fstart,pow_val))/pow(fstart,pow_val);

  /* Calculate recording times and send remote node notice to record */
  uint32_t start_time = get_global_time() + RECORD_DELAY_SECONDS * 1000 *
    1000;
  uint32_t record_time = record_duration * 1000 * 1000;
  for (i = 0; i < 8; i++) {
    payload[7-i] = record_time % 256;
    record_time = record_time / 256;
    if (i == 3) {
      record_time = start_time;
    }
  }
  record_time = record_duration * 1000 * 1000;
  start_time += 1000;

  audio_start_handle = send_message(START_RECORDING,8,&payload);
  printf("Audio Handle: %u\r\n",audio_start_handle);

  /* Wait until message has been properly sent */
  vTaskSuspend(NULL);
  printf("Message Sent\r\n");

  i = 0;
  q15_t q15_sin_arg;
  float32_t frequency = fstart;
  float32_t phase = 0;
  uint32_t loop_max = SAMPLE_RATE*duration;

  taskENTER_CRITICAL();
  underrun = 0;
  count = 0;

  /* Prepare first samples */
  while (count < (SOUND_SAMPLES))     {
    frequency = frequency + frequency*freq_change;
    phase = modf(phase + frequency*inverse_sample_rate,NULL);
    arm_float_to_q15(&phase, &q15_sin_arg, 1);
    samples_buffer[count] = (int16_t)(arm_sin_q15(q15_sin_arg)^0x8000);
    count++;
  }

  /* Wait until it is time to actually start */
  uint32_t system_time = get_global_time();
  if (system_time > start_time) {
    printf("Missed start deadline\r\n");
  }
  while (system_time < start_time) {
    system_time = get_global_time();
  }

  pdca_channel_write_reload(PDCA_ABDAC_CHANNEL0,(void *)samples_buffer,
    SOUND_SAMPLES);
  ioport_set_pin_level(RECORD_INDICTATOR,true);
  uint32_t playback_start_time = get_global_time();
  underrun = 0;
  i += count;
  while(!(pdca_get_channel_status(PDCA_ABDAC_CHANNEL0)==
    PDCA_CH_COUNTER_RELOAD_IS_ZERO));

  /* Keep feeding samples to the PDCA/DAC */
  while(i < loop_max) {
    count = 0;
    while (count < (SOUND_SAMPLES))   {
      frequency = frequency + frequency*freq_change;
      phase = modf(phase + frequency*inverse_sample_rate,NULL);
      arm_float_to_q15(&phase, &q15_sin_arg, 1);
      samples_buffer[count] = (int16_t)(arm_sin_q15(q15_sin_arg)^0x8000);
      count++;
    }
    pdca_channel_write_reload(PDCA_ABDAC_CHANNEL0,(void *)samples_buffer,
      SOUND_SAMPLES);
    i += count;
    while(!(pdca_get_channel_status(PDCA_ABDAC_CHANNEL0)==
      PDCA_CH_COUNTER_RELOAD_IS_ZERO));
  }

  /* Once finished, close everything out */
  ioport_set_pin_level(RECORD_INDICTATOR,false);
  printf("Exiting the audio output with %u underruns\r\n\r\n",underrun);
  abdac_set_volume0(&g_abdac_inst, true, 0x7FFF);
  abdac_set_volume1(&g_abdac_inst, true, 0x7FFF);
  taskEXIT_CRITICAL();

  system_time = get_global_time();
  while (system_time < (start_time+record_time)) {
    system_time = get_global_time();
  }

  /* Send notice to remote node of audio start time */
  printf("Requesting Status\r\n");
  uint8_t msg[4];
  for (i = 0; i < 4; i++) {
    msg[3-i] = playback_start_time % 256;
    playback_start_time = playback_start_time / 256;
  }
  send_message(RECORDING_STATUS,4,&msg);
  vTaskSuspend(NULL);

  /* Delete task */
  printf("Recording Status found\r\n");
  vTaskDelete(NULL);
}
```

### A.4.5 Wavwriter

```
/**
 * File: wavwriter.c
 * Creator: Jonathan Klapel
 * Date: 08 June 2014
 * Description: Writes WAV files
 */

#include "wavwriter.h"
#include <string.h>

/* WAV file header */
struct wavfile_header {
  char riff_tag[4];
  int    riff_length;
  char wave_tag[4];
  char fmt_tag[4];
  int    fmt_length;
  short  audio_format;
  short  num_channels;
  int    sample_rate;
  int    byte_rate;
  short  block_align;
  short  bits_per_sample;
  char data_tag[4];
  int    data_length;
};

/**
 * WAV File Open function
 *
 * Begins a WAV file that will have audio data streamed to it.
 */
void wavfile_open(FIL *file_object,char const *file_name,uint32_t length)
{
  /* Format header of WAV file*/
  struct wavfile_header header;

  int samples_per_second = WAVFILE_SAMPLES_PER_SECOND;
  int bits_per_sample = 16;

  strncpy(header.riff_tag,"RIFF",4);
  strncpy(header.wave_tag,"WAVE",4);
  strncpy(header.fmt_tag,"fmt ",4);
  strncpy(header.data_tag,"data",4);

  header.riff_length = 0;
  header.fmt_length = 16;
  header.audio_format = 1;
  header.num_channels = 1;
  header.sample_rate = samples_per_second;
  header.byte_rate = samples_per_second*(bits_per_sample/8);
  header.block_align = bits_per_sample/8;
  header.bits_per_sample = bits_per_sample;
  header.data_length = 0;

  FRESULT res;
  /* Create and open file for editing */
  res = f_open(file_object,(char const *)file_name,FA_CREATE_ALWAYS |
    FA_WRITE);
  if (res != FR_OK) {
    printf("Could not open file object\r\n");
    f_close(file_object);
    return;
  }

  /* Write header to file */
  uint32_t bytes_written;
  while( f_write(file_object,&header,sizeof(header),&bytes_written) !=
    FR_OK ) {;}

  /* Pre-allocate space for remained of file */
  f_lseek(file_object,length*2);
  f_lseek(file_object,sizeof(struct wavfile_header));
}

/**
 * WAV File Write function
 *
 * Writes an array of bytes to the file
 */
uint32_t wavfile_write(FIL *file_object, short data[], int length )
{
  /* Write bytes to the file */
  uint32_t bytes_written;
  while( f_write(file_object,data,sizeof(short)*length,&bytes_written) !=
    FR_OK) {;}

  return bytes_written;
}

/**
 * WAV File Close function
 *
 * Closes the WAV file once recording is finished
 */
void wavfile_close(FIL *file_object )
{
  uint32_t bytes_written;

  /* Remove any additional pre-allocated space that may exist */
  f_truncate(file_object);

  /* Get length of data sections to write into WAV file header chunks */
  int file_length = f_tell(file_object);
  int data_length = file_length - sizeof(struct wavfile_header);
  f_lseek(file_object,sizeof(struct wavfile_header) - sizeof(int));
  while( f_write(file_object, &data_length, sizeof(data_length),
    &bytes_written) != FR_OK ) {;}

  /* Get file length to write in WAV file header */
  int riff_length = file_length - 8;
  f_lseek(file_object,4);
  while( f_write(file_object, &riff_length, sizeof(riff_length),
    &bytes_written) != FR_OK ) {;}
  /* Close the file.  We're done with it. */
  f_close(file_object);
}

/**
 * WAV File Crop function
 *
 * Cuts the excess audio data off the front and back of the audio record-
ing
 */
void wavfile_crop(char const *file1, char const *file2, uint32_t record-
  ing_start_time, uint32_t playback_start_time, uint32_t duration) {
  FRESULT res;
  FIL f_file1, f_file2;
  BYTE buffer[128];
  uint32_t br, bw;

  /* Open file1 for reading from */
  res = f_open(&f_file1, (char const *)file1,FA_OPEN_EXISTING | FA_READ);
  if (res) return (int)res;

  /* Open file2 for writing to */
  res = f_open(&f_file2, (char const *)file2,FA_CREATE_ALWAYS | FA_WRITE);
  if (res) return (int)res;

  /* Copy header of old file into new file */
  f_read(&f_file1, buffer, sizeof(struct wavfile_header), &br);
  if (res || br == 0) return; /* error or eof */
  res = f_write(&f_file2, buffer, br, &bw);  /* Write to the destination*/
  if (res || bw < br) return; /* error or disk full */

  /* If there is excess audio on the front, skip over it.  If there is not
    enough audio, insert blank audio */
  if (recording_start_time < playback_start_time) {
    uint32_t remove_samples = (playback_start_time - recording_start_time)
      * WAVFILE_SAMPLES_PER_SECOND / (1000 * 1000) - CODE_OFFSET;
    remove_samples *= 2;
    res = f_lseek(&f_file1,f_tell(&f_file1)+remove_samples);
    if (res) return;
  } else {
    uint32_t add_samples = (recording_start_time - playback_start_time) *
      WAVFILE_SAMPLES_PER_SECOND / (1000 * 1000);
    while (add_samples > 0) {
      f_putc((TCHAR)0,&f_file2);
      f_putc((TCHAR)0,&f_file2);
      add_samples--;
    }
  }

  uint32_t i = 0;
  uint32_t max_samples = duration / (1000*1000) *
    (WAVFILE_SAMPLES_PER_SECOND*2);
  /* Copy source file to destination file, up to max audio duration*/
  while (i < max_samples) {
    if ((max_samples - i) < sizeof buffer) {
      res = f_read(&f_file1, buffer, (max_samples - i), &br);
      /* Read a chunk of source file */
    } else {
      res = f_read(&f_file1, buffer, sizeof buffer, &br);
      /* Read a chunk of source file */
    }
    if (res || br == 0) break; /* error or eof */
    res = f_write(&f_file2, buffer, br, &bw);
    /* Write it to the destination file */
    if (res || bw < br) break; /* error or disk full */
    i += br;
  }

  /* Close open files */
  f_close(&f_file1);
  wavfile_close(&f_file2);
}
```