UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

Underwater Communications System for an Acoustic Release

Daniel Brennan Wait

20887507

Report submitted in partial fulfilment of the requirements of the module Project (E) 448 for the degree Baccalaureus in Engineering in the Department of Electrical and Electronic Engineering at Stellenbosch University.

Supervisor: Prof. D.J.J. Versfeld

November 2020

Acknowledgements

UNIVERSITEIT•STELLENBOSCH•UNIVERSITY
jou kennisvennoot • your knowledge partner

Plagiaatverklaring / *Plagiarism Declaration*

1. Plagiaat is die oorneem en gebruik van die idees, materiaal en ander intellektuele eiendom van ander persone asof dit jou eie werk is.
   Plagiarism is the use of ideas, material and other intellectual property of another's work and to present is as my own.

2. Ek erken dat die pleeg van plagiaat 'n strafbare oortreding is aangesien dit 'n vorm van diefstal is.
   I agree that plagiarism is a punishable offence because it constitutes theft.

3. Ek verstaan ook dat direkte vertalings plagiaat is.
   I also understand that direct translations are plagiarism.

4. Dienooreenkomstig is alle aanhalings en bydraes vanuit enige bron (ingesluit die internet) volledig verwys (erken). Ek erken dat die woordelikse aanhaal van teks sonder aanhalingstekens (selfs al word die bron volledig erken) plagiaat is.
   Accordingly all quotations and contributions from any source whatsoever (including the internet) have been cited fully. I understand that the reproduction of text without quotation marks (even when the source is cited) is plagiarism

5. Ek verklaar dat die werk in hierdie skryfstuk vervat, behalwe waar anders aangedui, my eie oorspronklike werk is en dat ek dit nie vantevore in die geheel of gedeeltelik ingehandig het vir bepunting in hierdie module/werkstuk of 'n ander module/werkstuk nie.
   I declare that the work contained in this assignment, except where otherwise stated, is my original work and that I have not previously (in its entirety or in part) submitted it for grading in this module/assignment or another module/assignment.

Studentenommer / *Student number*	Handtekening / *Signature*
Voorletters en van / *Initials and surname*	Datum / *Date*

Abstract

English

Octopus traps have lines which extend from the seafloor to a buoy on the surface for retrieval. There have been numerous whale fatalities due to entanglement in these lines. It is mandated that traps must have a submerged buoy which is operated with a triggered release mechanism. An acoustic communications system has been designed to interrogate individual buoys.

Investigations into underwater communications were conducted. The decision was made to modulate 12-bit identification codes with chirped Frequency Shift Keying and detect the message with matched-filter cross-correlation. It is required to interrogate a specific receiver with low risk of another receiver accidentally being triggered. Therefore, a frequency hopping pattern is implemented to reduce the probability of a symbol/bit corresponding to the expected bit of another receiver.

Unit tests are conducted to confirm that the receiver (which is implemented on a Teensy microcontroller) correctly detects the expected message. Finally, simulations determine the detection and false alarm rates to determine the requisite signal characteristics at the receiver.

Afrikaans

Die Afrikaanse uittreksel.

Contents

List of Figures

List of Tables

List of Abbreviations

AWGN Additive white Gaussian noise

CDMA Code Division Multiple Access

DSP digital signal processing

DSSS direct sequence spread spectrum

FHSS frequency hopping spread spectrum

FSK Frequency Shift Keying

ISI intersymbol interference

PSK Phase Shift Keying

QAM Quadrature Amplitude Modulation

SNR Signal-to-noise ratio

UWA Underwater Acoustic

Chapter 1

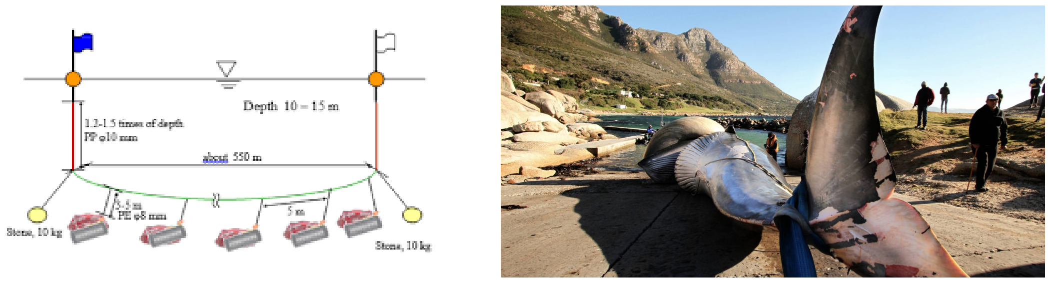
Introduction

1.1. Problem Statement

Fishing plays a significant role in the economy of the Western Cape and supports a large workforce. Octopi are a much desired for food and has generated an exploratory fishery.

Octopus traps are laid in their natural habitat, close to the shores of the False Bay area. Commonly, a jar is submerged to remain on the seabed and a chain or rope is connected between the jar and a buoy. Often, multiple jars are similarly interconnected. Any octopi which have taken refuge in the jars will be trapped and the traps are hoisted to the surface.

However, marine life has been collateral to these rudimentary traps. On several occasions, whales have been caught in the buoy-lines. The whales may not free themselves and sustain severe lacerations from struggling against the ropes. The South African Whale Disentanglement Network prevented many fatalities, but some were inescapable; among the deaths were several Brydes whales – which are classified as a vulnerable species [1].



(a)

(b)

Figure 1.1: Dangers of an octopus trap: (a) A high-level schematic of an octopus trap. (b) A juvenile Brydes whale which perished in the lines of an octopus trap in False Bay.

In June 2019, the Department of Environment, Forestry and Fisheries placed a moratorium on octopus fishing. The moratorium was lifted in November 2019; but the department issued a set of standards and conditions to ensure whale conservation. The new rules require the trap's buoy to be mounted on the bottom line with a release mechanism to allow the buoy to surface for retrieval and all components of the system must be retrieved [2].

1.2. Project Solution

The release mechanism requires a suitable communications system and receiver design.

The given user-requirements are that an acoustic communication system is implemented which can operate at a 40m depth. Furthermore, each device will have a identity number which acts as the 'release' command.

The ultimate goal of this project is to facilitate sustainable fishing. But, release mechanisms have a variety of applications which can be commercialized. A successful design can be adapted for other 'bottom line' fishing industries as well as to deploy equipment for scientific applications.

1.3. Project Scope

There are several challenges presented by underwater communication systems which need to be investigated. Any receiver which is submerged for lengthy periods must be intolerant of accidental triggers whilst reliably detecting its identification code. Thus a robust transmission protocol and receiver algorithm needs to be established.

Users will require multiple releases to operate in the same area. A set of identification codes needs to be determined which have a low probability of false alarms.

The design of a functional acoustic release requires multidisciplinary design with respect to electronics and mechanical challenges. The mechanical design of the release is excluded from the scope of work detailed herewith.

1.4. Summary of Work

The following work activities were executed to meet the project objectives:

- Study characteristics of UWA channels
- Develop a modulation scheme suited for UWA channels
- Design a receiver algorithm
- Acquire proficiency in the use of Teensy 4.1 MCU and the associated libraries
- Learn to use ARM's CMSIS DSP libraries
- Implement the receiver algorithm on a Teensy MC
- Determine a set of codes to individually interrogate releases
- Optimize code for low-power applications

1.5. Project Overview

Concepts and Literature Review

Underwater channels are investigated to characterize the operating conditions of the required system. With this in mind, telecommunications techniques which are appropriate for the scope of the project are determined and further information related to chosen techniques are presented.

With a telecommunications Digital Signal Processing background is discussed. Much of this section is in the context of hardware limitations such that an algorithm can be put into practice.

Design

The algorithmic design of the

Measurements and Results

Conclusion

Chapter 2

Literature Review and Related Concepts

2.1. Underwater Telecommunications

2.1.1. Underwater Channel Acoustics

Successful wireless communication requires a wave to propagate through the medium with minimal attenuation. Radio and optic waves do not propagate well in an underwater channel – thus, they have been excluded from use in this project [3, p.1]. Acoustic waves have become the standard in underwater channel communication because sound waves exhibit low attenuation. There is an inversely proportional relationship between the transmitting frequency and range of the wave; thus frequencies in the range of 1kHz to 100kHz are typically used [3, p.1]. Yet, there are several characteristics of underwater channels that present challenges to reliable acoustic communications.

Multipath Fading

Signals naturally spread and take scattered paths which are reflected off surfaces towards the receiver. This echo is combined with the signal in the direct path such that the received signal can be simulated as a signal mixed with an attenuated and time delayed version of itself. This is commonly referred to as 'multipath fading'. Fortunately, vertical paths experience shorter multipath effects compared to horizontal paths, in which the effect can occur for hundreds of milliseconds [3, p.11].

Path Loss

Acoustic waves experience path loss which can be described as a function of frequency, range, and a selected spreading coefficient [4, p.2]. The loss can be attributed to spreading of the pressure wave and absorption by the medium. Thorp's formula models an 'absorption coefficient' based on the frequency (in kHz) of the transmitted wave. The spreading coefficient (k) describes the directivity of the wave's propagation; k = 2 is used for an unguided medium with spherical spreading [5, p.6]. For l km and f kHz: Equation 2.1a gives Thorp's absorption constant in dB/km and Equation 2.1b shows the total path loss in dB.

$$10 \log a(f) = 0.11 \frac{f^2}{1+f^2} + 44 \frac{f^2}{4100+f} + 2.75 \times 10^{-4} f^2 + 0.003 \tag{2.1a}$$

$$10 \log A(l,f) = k.10 \log l + l.10 \log a(f) \tag{2.1b}$$

Noise

The ocean is quite alive, and sound propagates well. Shipping, thermal noise, surface winds, and turbulence all generate ambient noise which contends with the received signal. These ambient noises can be estimated with the following formulae [4, p.2].

Ambient Noise Equations:

$$10 \log N_t(f) = 17 - 30 \log f \tag{2.2a}$$

$$10 \log N_s(f) = 40 + 20(s - 0.5) + 26 \log f - 60 \log f + 0.03 \tag{2.2b}$$

$$10 \log N_w(f) = 50 + 7.5 w^{frac12} + 20 \log f - 40 \log f + 0.4 \tag{2.2c}$$

$$10 \log N_{th}(f) = -15 + 20 \log f \tag{2.2d}$$

Although there may be site-specific noise, the ambient noise is a sufficient predictor to estimate a Signal-to-noise ratio (SNR). The noise is typically simulated with Additive white Gaussian noise (AWGN), although the zero-mean (white) assumption is not necessarily true [3, p.4]. Equation 2.3 shows the estimated SNR dependent on: the projector's signal level ($SL_{projector}$), path loss, noise, and the bandwidth of the receiver (Δf).

$$SNR(l,f) = \frac{SL_{projector}/A(l,f)}{N(f)\Delta f} \tag{2.3}$$

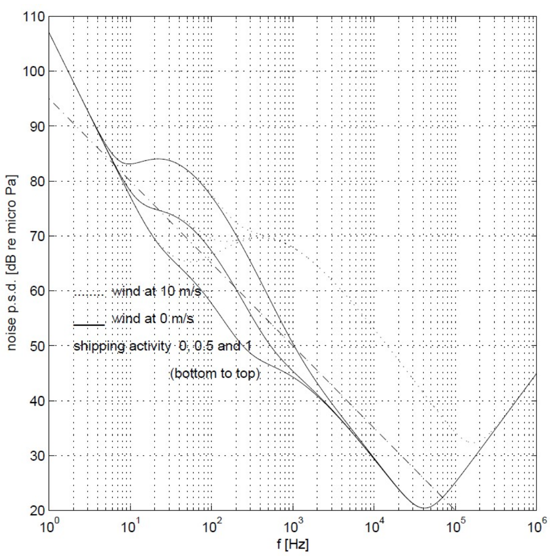


Figure 2.1: SNR versus frequency for UWA channel

2.1.2. Acoustic Transducers

An acoustic transducer is a device which can be used as a hydrophone, to receive Underwater Acoustic (UWA) signals, or as a projector, to transmit UWA signals.

As a receiver, the metrics of interest are:

- The sensitivity in dB re 1V/Pa - i.e. the voltage induced by a 1 µPa pressure wave.
- The useful range of the receiver, in kHz.
- And, the directionality of the receiver.

As a projector, the metrics of interest are:

- The Transmitting Voltage Response (TVR) in dB re 1 µPa / V @ 1m dB - i.e. the pressure measured at 1m from the source given a 1V input of a particular frequency
- The driving RMS voltage (V_{RMS}).
- And, the source signal level (SL) which is calculated as shown in Equation 2.4 [6, p.11].

$$\text{SL} = \text{TVR} + 20 \log V_{RMS} \tag{2.4}$$

2.1.3. Digital Message Detection

This project only requires simplex communication and the receiver must detect a single binary code. Digital telecommunications modulate a binary message for transmission and the receiver must interpret the modulated signal by some means.

Digital Demodulaion

When the receiver is required to demodulate arbitrary messages, a system to demodulate individual bits must be employed. These schemes are broadly categorized as being coherent or non-coherent based on whether the demodulation algorithm requires timing synchronization with the transmitted signal.

Matched Filter Detection

A matched filter is an alternative method to detect the presence of an individual bit or message. With prior knowledge of the modulation scheme and the expected binary message, the matched filter seeks that particular modulated message and quantifies the input's likeness to the expected message. A threshold value is set to make a binary decision as to whether the signal is absent or present. Salahdine's investigations into detector methods compares the performance of matched filters, energy detection, and autocorrelation detection. The results show that the matched filter has the best performance for low sample number and low SNR [7, p.5].

A typical digital demodulation system would be excessive for the application. Rather, a matched filter will be used. Furthermore, matched filters are considered to be the optimum receiver for AWGN channels [8, p.544] which would make it an appropriate choice for underwater channels. The implementation of a matched filter is discussed further in Section 2.2.4.

2.1.4. Digital Modulation Methods

The digital modulation techniques which are commonly applied in UWA comms are: Frequency Shift Keying (FSK), Phase Shift Keying (PSK), and Quadrature Amplitude Modulation (QAM). PSK and QAM were excluded because they rely on phase information so coherent detection is required to compensate for phase distortion [3, p.1].

FSK is a method of digital modulation which encodes the bit values as frequencies. FSK will be the suitable for modulation as it is resistant against multipath effects [9, p.3]. Furthermore, FSK signals can be detected with a simple matched filter.

Frequency Shift Keying

The most basic case of FSK is Binary FSK (BFSK or 2-FSK). '1's and '0's are modulated by sinusoidal carrier frequencies - called a 'mark' and 'space' respectively [8, p.373]. When symbols constituted of N bits, are represented by M frequencies (where $M = 2^N$), the signal is referred to as M-ary FSK.

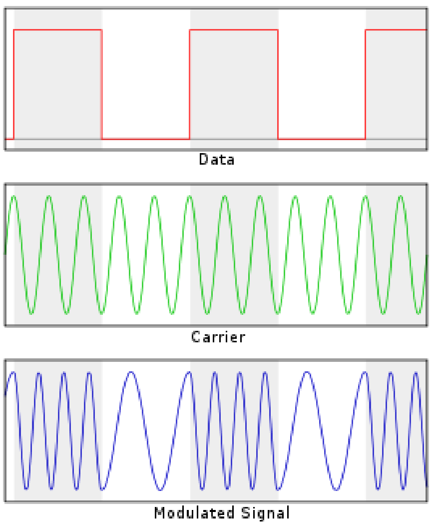


Figure 2.2: 2-FSK modulation of a binary signal

Orthogonal Carriers

Due to multipath fading intersymbol interference (ISI) will occur, so it is desirable for symbols to be orthogonal during a symbol period. The carrier frequencies can be chosen such that the signals are orthogonal (or uncorrelated). The orthogonality condition states that if the inner product of two time-domain signals equal zero, the signals are orthogonal [8, p.31].

$$< g(t), x(t) >= \int_{t_1}^{t_2} g(t).x(t).dt = 0$$

To guarantee that the carrier frequencies are orthogonal for the duration of a bit period (T_b), a minimum frequency separation ($\Delta f = f_1 - f_0$) must be determined which satisfies the orthogonality condition.

$$\int_0^{T_b} Acos(2\pi f_0 t).Acos(2\pi f_1 t).dt = \frac{A^2}{2}.\frac{sin(2\pi(f_1 + f_0))T_b}{2\pi(f_1 + f_0)} + \frac{A^2}{2}.\frac{sin(2\pi(f_1 - f_0))T_b}{2\pi(f_1 - f_0)} = 0$$

The inner product produces two terms, the first of which can be neglected given that the frequencies are typically in the order of kHz. The orthogonality condition simplifies so the minimum frequency separation which fulfills the condition is:

$$\Delta f = \frac{1}{2T_b} \tag{2.5}$$

With the orthogonality condition and knowledge of the bit period (T_b), a minimum frequency separation (Δf) can be determined such that the modulation frequencies will be orthogonal. Any integer multiple of the minimum separation satisfies orthogonality as well [8, p.381].

2.1.5. Chirps

A ‘chirp’ typically refers to a method of spread spectrum communication which linearly increases or decreases the transmitting frequency within the duration of a bit period. A linearly increasing sweep is called an ‘up-chirp’ whereas a linearly decreasing sweep is called a ‘down-chirp’.

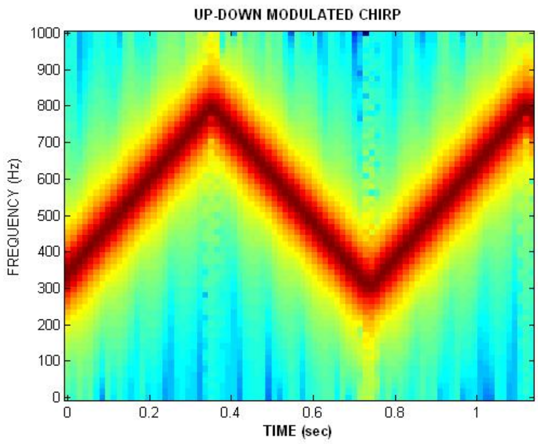


Figure 2.3: Spectrogram of up-chirps and down-chirps at the the same centre frequency

Investigations into chirped FSK indicates that it sufficiently mitigates the multipath effect [9]. The characteristics of the multipath effect were stochastically modelled and the performance was compared to a standard FSK message. In this case, up-chirps were applied to marks and down-chirps were applied to spaces.

2.1.6. Frequency Hopping Spread Spectrum

When multiple receivers are using the same channel and bandwidth for communication, it becomes difficult to target communications towards a single device. Code Division Multiple Access (CDMA) is a class of modulation algorithms which may circumvent this challenge. direct sequence spread spectrum (DSSS) and frequency hopping spread spectrum (FHSS) are the most well-known CDMA techniques. DSSS requires timing synchronization which is not relevant to this project; thus, it will not be discussed further.

FHSS is a spread spectrum technique which increases the bandwidth of the communication by choosing multiple modulating frequencies to represent a single symbol. A hopping pattern is known to both the transmitter and receiver, the modulation frequency is offset based on the pattern; and the targeted receiver successfully demodulates the signal with knowledge of the expected frequency offset. FHSS can be classified as 'slow' if the frequency offset is applied slower than or equal to the symbol-rate or 'fast' if the offsets are applied multiple times within a symbol period. Military communications systems in particular implement FHSS to avoid signal jamming. However, UWA networks have found this to be particularly useful both for multiple access communications and to reduce ISI consequential of the multipath effect [3, p.15].

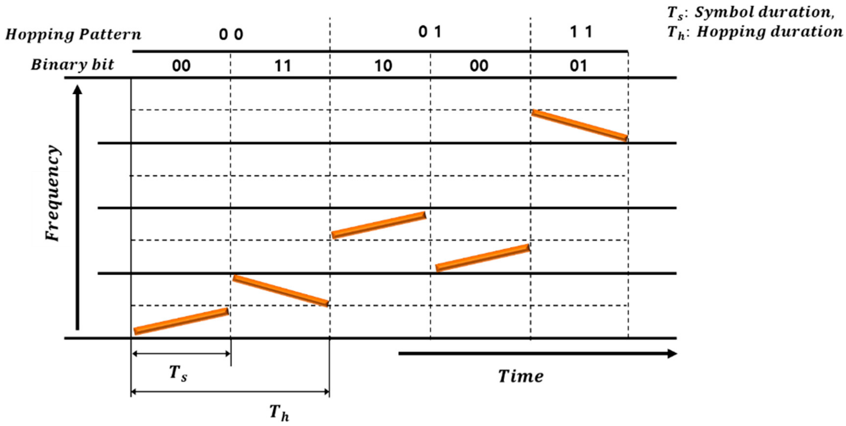


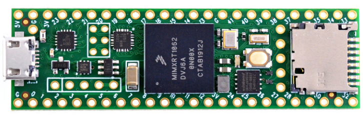
Figure 2.4: Spectrogram of frequency hopping with chirps

2.2. Digital Signal Processing

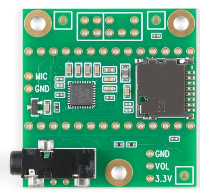
2.2.1. Hardware

The receiver is required to be implemented on a Teensy 4.1 Developement Board with a Teensy Audio Adapter Board. These devices are ideal for digital signal processing (DSP) with audio projects. The development board makes makes use of a powerful ARM Cortex-M7 based microcontroller and is programmed in the Arduino IDE. The Audio Adapter encodes this data into a digital I2S format which fetched by the microcontroller.

A significant benefit to the Teensy devices is the Audio Library and Audio System Design Tool which are used with it. The Teensy Audio Library defines many classes which offer a range of functions from hardware control to synthesizers to filters. The Audio System Design Tool is a graphical interface to drag and drop Audio Objects and connect them for a project setup. C code is generated from the schematic and is exported to the Arduino IDE for initialization.



(a) Teensy 4.1



(b) Audio Adapter Rev. D

Queue Object

The ‘queue’ object is provided for users to fetch samples from the Audio Adaptor. The microphone input accepts a maximum voltage input of 1.7 V_{pp} which is followed by a gain stage and an ADC. An interrupt is triggered each time a packet of 128 samples is available. The ADC sampling frequency is 44.1kHz, thus 128 samples arrive every 2.9ms. A queue of 208 packets may be stored - but the queue must be cleared or the program will malfunction.

2.2.2. Fourier Transforms

Discrete Fourier Transforms

It is often neccessary to do a frequency analytsis on received signals. A transform can be applied to finite-length sampled time-domain signals which represent the frequency content of that signal. This is called a Discrete Fourier Transform (DFT) and is calculated as shown in Equation 2.6a. The Inverse Discrete Fourier Transform (IDFT) similarly transforms frequency domain signals to the time-domain.

$$H[k] = \text{ DFT\{ h[n] \} } = \sum_{n=0}^{N-1} h[n] \cdot e^{-j2\pi kn/N} \tag{2.6a}$$

$$h[n] = \text{ IDFT\{ H[k] \} } = \frac{1}{N} \sum_{n=0}^{N-1} H[k] \cdot e^{j2\pi kn/N} \tag{2.6b}$$

For the DFT/IDFT, there is a quadratic relationship between the number of complex multiplications and the length of the transformed signal.

$$\text{\# Real Multiplications DFT/IDFT} = N^2 \tag{2.7}$$

The Fast Fourier Transform

If the complex input length (N) is a power of two, the Fast Fourier Transform can be applied. The algorithm recursively breaks the calculations into smaller sections and reuses precalculated values. The number of calculations does not escalate as dramatically for an increased length of the input data.

$$\text{\# Real Multiplications FFT/IFFT} = 2N \log_2 (N) \tag{2.8}$$

Double Length Algorithm

FFTs generally expect complex data input and complex data output; yet in most cases the data which is available purely real. For a real input signal, one could assign zero magnitude to each imaginary value of the array and proceed with the FFT; however, the Double-Length Algorithm exploits the properties symmetrical property of real input FFTs to reduce calculations.

The input array, of N real values, is copied into an array with N/2 complex values – even data is assigned to the real elements and odd data is assigned to the imaginary elements. Thus, the Fast Fourier Transform of half the input length is calculated and multiplications are required to construct the output array of N complex values [10, p.20]. The number of multiplications to complete this adjusted Fourier Transform is shown in Equation 2.13.

$$\#\text{ Real Multiplications RFFT/RIFFT} = N\log_2(N/2) + 4(\frac{N}{2} - 1) \tag{2.9}$$

2.2.3. Filtering

Discrete Convolution

The impulse response (h) of a Linear Time-Invariant system allows one to determine the output (y) given an arbitrary input (x) with the discrete convolution of h and x [11, p.73]. Discrete convolution is mathematically described in Equation 2.10.

$$y[n] = x[n] * h[n] = \sum_{i=-\infty}^{\infty} x[i].h[n-i] \tag{2.10}$$

FIR Filters

The output of a Linear Time-Invariant systems can often be described as a simple arithmetic operation. A certain number of past and present input samples are scaled by a coefficient and summed to produce the current output sample.

$$y[n] = \sum_{k=0}^{M} b_k.x[n-k] \tag{2.11}$$

Graphically, this can be represented in Figure 2.6. "z^{-1}" represents a delay by the sample period.

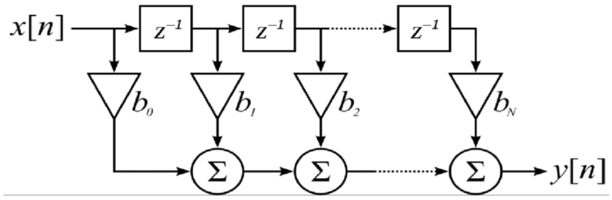


Figure 2.6: FIR Block Diagram

In DSP, filters are generally implemented as a FIR filters. This is a continuous digital processing technique – meaning that the algorithm is executed for a single input sample to generate a single output sample. The number of multiplications required for each sampled output equals the number of filter coefficients. Therefore, a long filter requires many calculations which introduces latency.

Overlap-Add

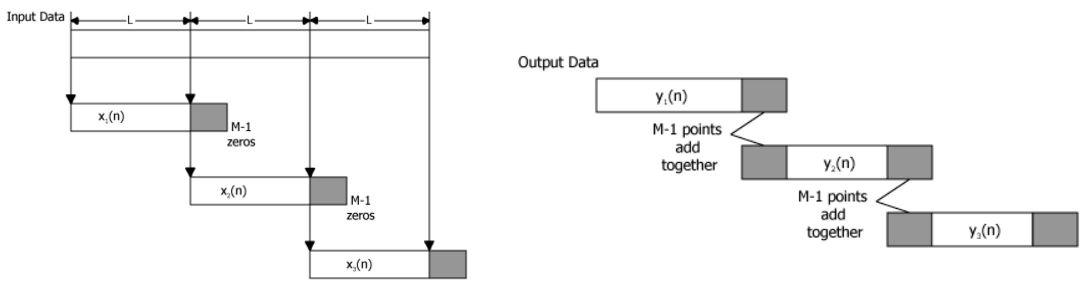
An alternative to a convolution filtering is the overlap-add method. An overlap-add method exploits the equivalent relationship of convolution in the time-domain and multiplication in the frequency-domain. The DFT of the filter coefficients is multiplied with the DFT of the input, and the IDFT is applied to the result.

$$X_m[k] = DFT\{x_m[n]\} \tag{2.12a}$$

$$Y_m[k] = H[k]X_m[k], \text{ and } k = 0..(N-1) \tag{2.12b}$$

$$y_m[n] = IDFT\{Y_m[k]\} = h[n] * x_m[n] \tag{2.12c}$$

If an M length filter is to be used with an L length block of data, the filter and input are zero-padded to a length of $N = L + M - 1$ (N may be longer, but this is the typical choice). The resultant time domain data is of length N. The first M-1 samples of the new output block are added with the previous block's samples and the remaining L samples are appended to the total output [].



(a) (b)

Figure 2.7: Overlap-Add Data Blocks: (a) Zero-padded input data (b) Overlap management of output data

The overlap-add filter is a batch processing technique, and it is generally assumed that the input data is complex. The computational complexity per batch can be calculated based on the multiplications for each DFT. If N is chosen to be a power of 2, the FFT can be used such that the number of calculations is reduced. An optimal choice of L and N can greatly reduce the number of calculations per input sample compared to an FIR implementation.

$$\# \text{ computations per input sample} = \frac{4N\log_2 2N}{L} \tag{2.13}$$

Uniformly Partitioned Overlap-Add

The traditional overlap-add algorithm presupposes that the input block is much longer than the filter. Many MCUs' APIs facilitate optimized FFTs; however, the transform's input data is limited in length. Thus, any filter of significant length may violate the condition of a filter response shorter than data available from the input buffer and exceeds the supported lengths of the FFTs.

The Uniformly Partitioned Overlap-Add (UPOLA) method, depicted in Figure 2.8, is a partitioned frequency convolution technique which extends upon the traditional overlap-add filter to solve the above conflicts. With a partitioned overlap-add, filters of a much longer length can be used. The filter is subdivided into K blocks of length M. The input blocks are comprised of the most recent K blocks, each of length L. The condition is imposed that $N = 2 \cdot L$. A filter block length $M = L$ is most calculationally efficient choice. The FFT of each filter block is multiplied by the FFT of a related input block. The IFFT of the resultant blocks are summed and the output overlap is managed similarly to the traditional implementation.

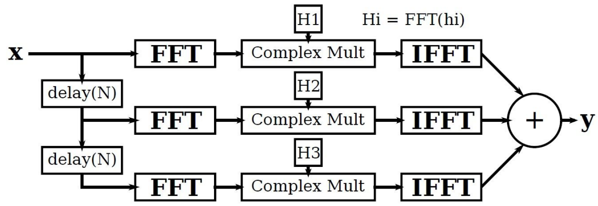


Figure 2.8: Uniformly Partitioned Overlap-Add

To remove redundant transforms, the frequency delay line can be applied (Figure 2.9). With this method, the FFT of each input block is stored and the distributive property is applied to the summation of IFFTs. The algorithm is computed with an FFT, $K \cdot N$ complex multiplications of frequency domain data, and an IFFT is applied to the summation.

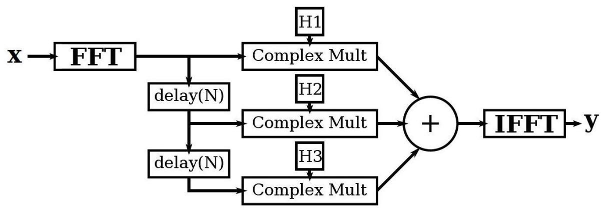


Figure 2.9: Partitioned Overlap-Add with a Frequency Delay Line

2.2.4. Matched Filters

Discrete Autocorrelation

Correlation can be mathematically described as the integral of the multiplication of two signals. The magnitude of the correlator output is a measure of the likeness between two signals. Autocorrelation is the correlation of a signal with itself. The peak of the output quantifies the maximum likeness of a signal to itself. Equation 2.14 shows the autocorrelation of a discrete signal [11, p.119].

$$r_{xx}[n] = \sum_{i=-\infty}^{\infty} x[i].x[i-n] \tag{2.14}$$

As shown in the formula, one signal is right-shifted by n samples relative to the other. The number of shifts is referred to as **'lags'**.Where the lag value is non-zero the number of samples being compared decreases by n. At zero lags the signals are compared to each other for all samples within the finite sequence and will produce a peak of the autocorrelation output.

Discrete Crosscorrelation

Crosscorrelation is the correlation of two signals which may not be related. Equation 2.15a shows the discrete crosscorrelation of signals x and y [11, p.118].

If a signal (x) is transmitted, it will be attenuated, time-delayed, and polluted with noise (w) such that the received signal (y) can be represented by Equation 2.15b. The crosscorrelation of x and y can be described in terms the autocorrelation of the transmitted signal and the crosscorrrelation of the transmitted signal with the noise as shown in Equation 2.15c.

$$r_{yx}[n] = \sum_{i=-\infty}^{\infty} y[i].x[i-n] \tag{2.15a}$$

$$y[n] = \alpha.x[n-D] + w[n] \tag{2.15b}$$

$$r_{yx}[n] = \alpha.r_{xx}[n-D] + r_{wx}[n] \tag{2.15c}$$

Therefore, if a cross-correlation is performed continuously performed on incoming data, the magnitude of the output can be compared to the autocorrelation peak to decide whether the expected signal is present.

Matched Filter Correlation

Discrete crosscorrelation may also be described as the discrete convolution of a signal with the time-reversed version of another. This is the basis for a matched filter: an input signal can be convolved with a time reversed known signal (called the template) to produce the crosscorrelation of the signals. Thus, the template signal acts as the filter impulse response [].

$$r_{yx}[n] = y[n] * x[-n] \tag{2.16}$$

2.2.5. Downsampling

Downsampling is process which effictively reduces the sampling rate. The sampled data is preprocessed so that every Jth sample is saved whereas other samples are discarded. Intuitively, the perceived frequency of the downsampled signal becomes a factor of J higher than the original signal; accordingly, the DFT of the downsampled data shows that the frequency spectrum becomes wider by a factor of J.

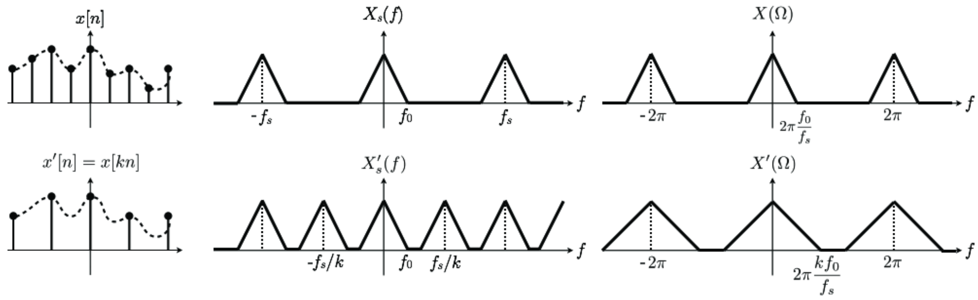


Figure 2.10: Caption

It is important to note that downsampling poses a risk of aliasing. 'Decimation' is often used interchangeably with 'downsampling', but may also be used to describe the downsampling with an anti-alias filter stage to compensate for this.

2.2.6. CMSIS DSP Library

"The Cortex Microcontroller Software Interface Standard (CMSIS) is a vendor-independent hardware abstraction layer for microcontrollers that are based on Arm Cortex processors" - ARM Holdings [13].

ARM-based microcontrollers are designed for DSP applications and facilitate floating-point arithmetic with inclusion of an FPU. The CMSIS libraries include an extensive DSP library for complex math, filters, transforms, and much more. These libraries are far more computationally efficient than user-defined functions based on first principles due to the use of look-up tables and other methods to reduce computations. Aspects particularly relevant to this project are discussed below.

q15_t datatype

The q15_t data type is a type definition of the int16_t datatype which is stored on ARM Cortex M processors in 2's complement format. The difference is in the interpretation of the data; the q15_t datatype is a fixed-point number in the format of a single integer bit followed by fifteen fractional bits [14]. The conversion of q15_t data to its floating-point equivalent is division by 32768 ($= 2^{15}$). Given that most sensors and analog data are represented in an integer format, the q15_t datatype allows for a standard interpretation of input data which must be represented in integer format.

Fourier Transforms

The CMSIS library offers FFTs and IFFTs which can be applied to fixed-point or floating point arrays. The transforms generally support arrays with 64 to 4096 elements - where the length is a power of two.

The transforms are further classified by complex or real input data. The Complex FFT/IFFT expects interleaved real and imaginary values of the same datatype within the same array. Therefore, an input of P complex values is stored in an array of 2P elements and the output is stored in an array of the same length [15].

Alternatively, the Real FFT/IFFT expects real valued time-domain data of length P and the frequency domain data is stored in an array of 2P elements - the Double-Length Algorithm is applied with these transforms [16]. Benchmarking tests show that Real FFTs are considerably faster than Complex FFTs [17, p.10-11]; thus, the RFFT is favoured for efficiency.

2.3. Related Works

2.3.1. EdgeTech 8242XS Release

This commercial release is intended for harsh environments and has been thoroughly documented. The device employs duplex communications with a digital demodulation system. Commands are comprised of 16-bit, B-FSK modulated messages with a symbol period of 22 milliseconds. Carriers in the range of 9.5 to 10.7 kHz were used.

2.3.2. A Practical Guide to Chirp Spread Spectrum for Acoustic Underwater Communication in ShallowWaters

This paper produced by the Hamburg University of Technology explores matched filter detection of chirped FSK modulated synchronization codes for Autonomous Underwater Vehicles. Multipath fading is identified as a major source of ISI and investigations due to the horizontal nature of the communications system. Optimization for the symbol rate and chirp bandwidth produced some useful observations.

- Longer symbol periods, in the range of 1ms to 10ms, were used. Increasing the symbol period non-linearly improved the matched filter detection.
- Equation 2.17: choose a chirp bandwidth (B) which based on the difference in time of arrival of the (Line of Sight) and the strongest reflection (Non Line of Sight).

$$B >= \frac{1}{\Delta t_{TOA}} = (\frac{d_{LOS} - d_{NLOS}}{v_{uw}})^{-1} \tag{2.17}$$

2.3.3. Development of a set of optimum synchronization codes for a unique decoder mechanization

Often a matched filter is implemented in communications systems which require timing synchronization of data-frames. Typically a binary sequence indicates the beginning of such frames. This means that this code must have a low chance of being randomly transmitted and, given that timing must be accurately synchronized, the autocorrelation peak must be distinguishable.

This paper investigates binary sequences for synchronization, namely: Barker, Legendre, Goode-Phillips, and Maury-Styles. These sequences are ideal for digital correlation because they produce an output with low Peak-to-Sidelobe ratios and they have a low probability of being generated by noise.

Patterns with a low Peak-to-Sidelobe Ratio are desirable because the peak of the correlation is more distinguishable so that the lobes are not incorrectly assumed to be peaks - naturally, this is important for timing synchronization. A low sidelobe ratio implies that the code has low similarity with itself for overlapped portions of the signal.

The fundamental use of this research to the project is stated best by the author:
"A designer, implementing a particular detector ... will evolve a singular criterion of code optimality. Usually, once this criterion has been defined, the binary pattern best fulfilling said standards is subsequently generated. Consequently, there exist sets of 'optimum' codes corresponding to the various investigations."

Therefore, the modulation scheme and detector algorithm must be established before the optimal set of codes can be determined.

Chapter 3

Design

The goal of this section is to define a transmission protocol, design a receiver, and determine a set of identification codes such that individual buoys can be released with low probability of false triggers.

A preliminary transmission protocol is defined so that a matched filter algorithm can be implemented on the given hardware. To determine a set of optimal codes, the modulation scheme will be optimized and the template generation adjusted accordingly.

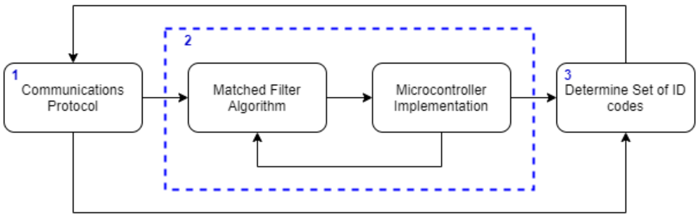


Figure 3.1: Iterative design process

In this chapter, the design is presented in the following order:

1. Matched Filter Algorithm Design
2. Microcontroller Implementation
3. Optimization for Multiple Access

3.1. Matched Filter Algorithm Design

A matched filter is to be used to detect if the release signal is present. With prior knowledge of the binary ID code and the modulation scheme, a template of the expected signal is locally generated. Thence, the matched filter continuously performs a crosscorrelation of the incoming audio data with the template.

This section's objective is to find a computationally cheap algorithm which can be implemented with the CMSIS library to reproduce the cross-correlation output. The algorithms were simulated in an iPython Notebook which is included in Appendix ??.

3.1.1. Preliminary Transmission Scheme

Message Structure

Each receiver is to be individually addressed by a binary message which represents its ID, thus a standard message length and symbol rate need to be specified.

Longer symbol duration and a longer transmission duration improves detection; thus, the release code is defined as a 12-bit message which is transmitted with a symbol period of 20 ms i.e. 50 bps (bits per second).The 12-bit message transmitted at 50bps is sampled by the Teensy at 44.1kHz; therefore, each bit will be sampled 882 times. Thus, the 12-bit message will be represented by a matched filter template of $12 \times 882 = 10584$ samples. Each bit is modulated as defined in Section 3.1.1.

FSK Carriers

The selection of the carrier frequencies depends on 4 criteria: Nyquist Sampling Theorem, the minimum frequency shift, the TVR of the hydrophone, and the SNR of the received signal.

The Nyquist Sampling Theorem states that, to capture a particular frequency, the sampling frequency must be twice as large. This is a minimum requirement and not a sufficient condition. Therefore, carrier frequencies must be selected which are sufficiently sampled - frequencies less than $\frac{44100}{5} = 8820$ Hz were preferred.

Given a symbol period of 20ms, the minimum frequency shift required for orthogonal FSK modulation is 25Hz as shown in Equation 2.5. Therefore carrier frequencies will be a multiple of 25Hz. This project is not severely bandwidth constrained, so the frequency separation can be significantly larger than the minimum distance.

The TVR of the hydrophone which shall be used in the final system is constrained for frequencices in the low kilohertz; therefore, carrier frequencies must be selected within the local maxima of the TVR. Examination of the TVR plot indicates that frequencies in the range of 1.2kHz to 3.5kHz will have a TVR of at least 110 dB and a peak of 113dB at 2.3kHz.

A MATLAB script was written to evaluate the SNR of various frequencies with the equations given in Section 2.1.1. The final SNR depends on four other variables which have been fixed: communication distance (r), spreading factor (k), shipping activity (s), wind speed (w), and receiver bandwidth (BW). The average wind speed in False Bay is 22 km/h, the shipping activity is assumed to be the mean, the spreading factor is assumed to be spherical, and the receiver bandwidth is chosen to be 600 Hz. An SNR of at least 10 dB was suggested by an experienced maritime engineer and the minimum frequency which satisfies that constraint is 1.57 kHz. Therefore, the space and mark frequencies are selected as 1.6 kHz and 2.2 kHz respectively. Appendix B contains the calculations which support this choice and Figure B.1 plots the SNR versus frequency.

3.1.2. Expected Input and Output

The binary message ’101011010111’ was randomly selected for testing. The message is FSK modulated as defined in Section 3.1.1. The correlation function in Python’s Numpy library was used to produce the expected auto-correlation as shown in Figure 3.2.

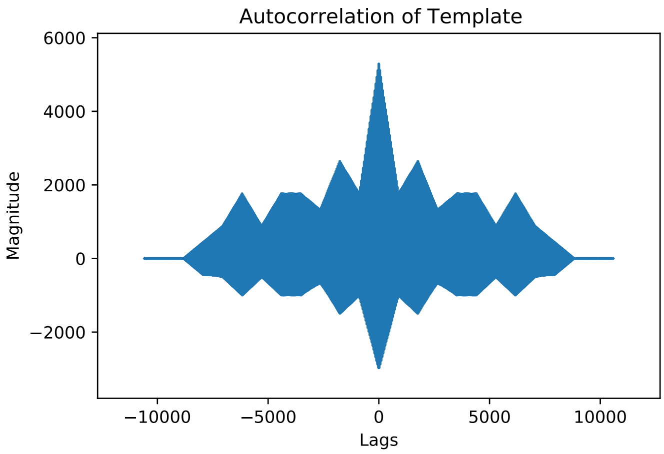


Figure 3.2

3.1.3. Algorithm Selection

FIR Filter:

The FIR filter was automatically excluded. The implementation would require 10584 real multiplications per input sample; this is too inefficient and the calculation time may exceed the sampling period. Also, a continuous processing technique would not be suitable for the Teensy which buffers audio input data in batches.

Overlap-Add Algorithm:

A straightforward Overlap-Add implementation of the matched filter was considered. If 173 packets of 128 samples were used for the input and zero-padded to a length of 32768 and the requisite multiplications would reduce to 300.86 multiplications per sample. However, the CMSIS DSP library supports a maximum FFT input length of 4096; therefore, it can not be implemented on the microcontroller in practice because the FFT of the 10584 sample template can not be calculated with a single transform.

Partitioned Overlap Add:

Because the straightforward Overlap-Add filter can not be implemented, a Partitioned Overlap-Add is required to segment the filter and input data into lengths which are managable by the CMSIS library. The Frequency Delay Line method is used so each new block of data will only require a single FFT and IFFT to reduce calculations.

3.1.4. Computations

A uniform segmentation is used with a Frequency Delay Line to reduce the computations. The segmentation length (L) and number of blocks (K) must be chosen to require the fewest real multiplications. Equation 2.8 is applied for the FFT/IFFT multiplications because the double-length algorithm is applied to real input data.

The table below shows the calculational complexity for the range of possible input lengths which the CMSIS DSP library supports. Based on the table, 6 blocks of 2048 real input samples would result in the fewest multiplications per sample. Therefore, the template is zero-padded to be split into 6 blocks.

K	**L**	**N**	**#Mult. FFT/IFFT** (Nlog(N/2)+4(N/2-1))	**#Mult. Filter × Input** (4KN)	**#Mult./sample**
83	128	256	2300	84992	699.94
42	256	512	5116	86016	375.97
21	512	1024	11260	86016	211.98
11	1024	2048	24572	90112	135.99
6	2048	4096	53244	98304	99.99

Table 3.1: Number of real multiplications required for a template of 10584 samples

3.2. Microcontroller Implementation

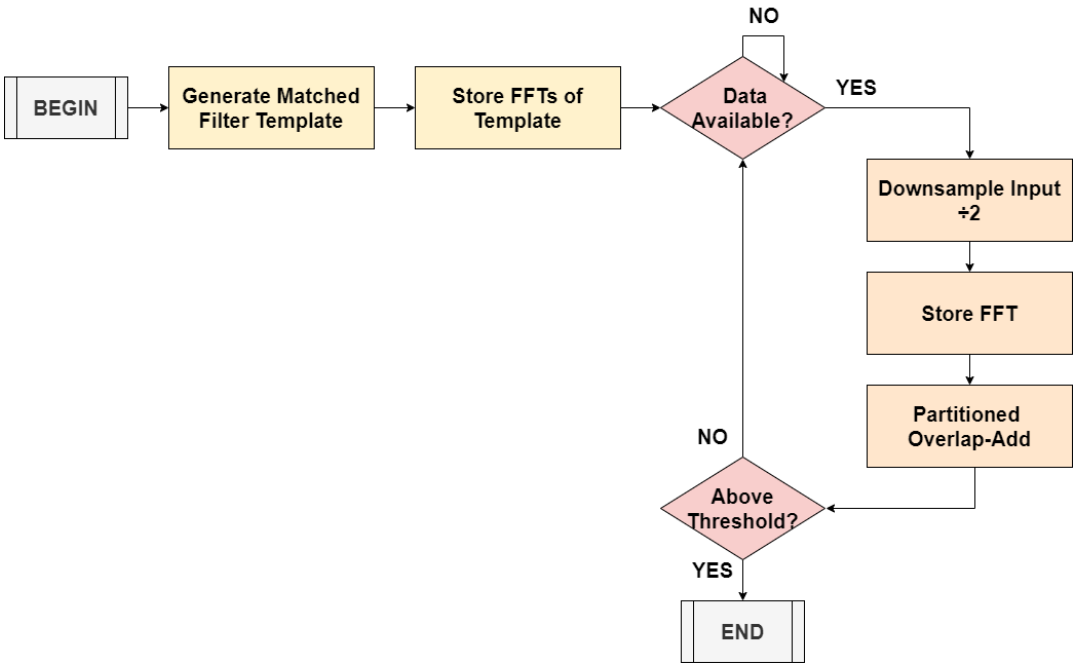
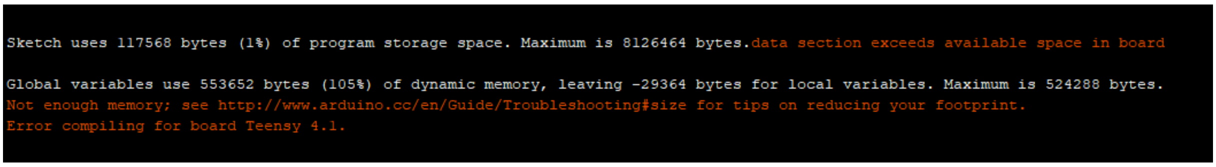


Figure 3.3: High-Level Flow Diagram of Software Design

3.2.1. Memory Limitations

An unexpected error occurred during the development of the code. The stored FFT arrays consumed too much memory allocated to variables and the following error was thrown:



```
Sketch uses 117568 bytes (1%) of program storage space. Maximum is 8126464 bytes.data section exceeds available space in board

Global variables use 553652 bytes (105%) of dynamic memory, leaving -29364 bytes for local variables. Maximum is 524288 bytes.
Not enough memory; see http://www.arduino.cc/en/Guide/Troubleshooting#size for tips on reducing your footprint.
Error compiling for board Teensy 4.1.
```

Figure 3.4: Teensy Memory Error Message

The possible workarounds were to: solder additional RAM memory to the development board, increase the bitrate, or downsample the data. Downsampling was favoured because the 44.1kHz sampling frequency exceeds the requirement of the application. By downsampling, the perceived bit-rate is increased with the convenient relaxation of processing deadlines.

A downsampling factor of 2 was applied such that the dynamic memory was not over-consumed whilst retaining a low symbol-rate of 10ms (441 samples). The perceived message length is halved to 120ms 5292 samples and the perceived frequency of incoming signals is doubled - the matched filter template is adjusted accordingly. Therefore, Table 3.1 was recalculated; L remains 2048 and K is changed from 6 blocks to 3 blocks.

3.2.2. Initialization

FIFO Circular Buffers

Two circular buffers are used to handle input and output data. Both of these buffers are First In First Out and only require a tail counter to insert data. The first buffer is a 3 x 8192, two-dimensional array which handles the Frequency Delay Line. The tail loops through the first dimension such that the FFTs of input data can be stored. The second buffer stores the output of the UPOLA algorithm. It works in conjunction with the function 'write_ola_buffer' to handle the overlap-add of output data with the previous output.

Matched Filter Template

A binary code is defined in the setup. 'generateTemplate' is used to generate the partitions of the time-domain signal defined by the modulation scheme. To ensure a zero-padded signal of length 4096, a two-dimensional array of 3 blocks by 4096 samples is initialized with zeroes and the first 2048 samples are written to with the template values. Next, the FFT of each partition is calculated and stored in the global variable 'mf_fft'.

3.2.3. Main Loop

Data Availability

The audio data is fetched by the queue object. Each time a block of 128 samples are available, an interrupt is generated. It is more efficient to exploit the interrupt than to redundantly poll for data for 2.9 ms. A wait for interrupt (WFI) assembly instruction is inserted into the main loop to put the microcontroller into a low-power sleep mode until the interrupt flags a new packet of data.

Input Data Preprocessing

The program waits for 4096 samples (32 packets) of the q15_t data to arrive, and coverts it to float32_t datatype. The data is then passed to the 'downsample' function which copies every 2^{nd} sample to a buffer of length 2048. The downsampled input data is then transformed with an FFT and written to the tail of the circular Frequency Delay Line buffer.

Processing

The UPOLA algorithm is implemented here. The most recent input FFT is multiplied with the 1st segment of the filter FFT, and so forth until the oldest block of the input FFT is multiplied with the 3rd segment of the filter FFT.

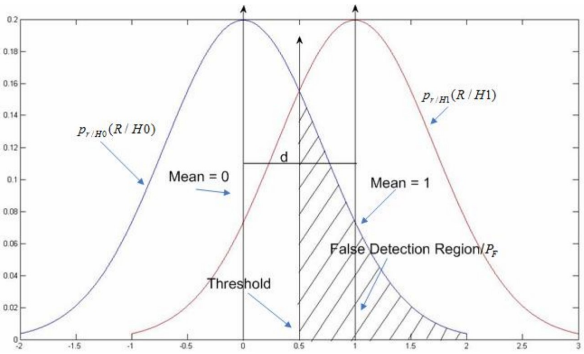


Figure 3.5: Detection Threshold

If the data is retained as q15_t/uint16_t, multiplication of magnitudes in the order of 10^3 will saturate or overflow and the IFFT will not be viable. Thus, it is neccessary to convert the q15_t data to float32_t values in the range ± 1 so the multiplication will not overflow.

Finally the IFFT of the summations are contained and passed to the 'write_ola_buffer' function to write data.

Threshold Detection

If the detection and probability of false alarms are considered to be Guassian distributions of the same variance but different means, the threshold value which maximizes detection whilst minimizing false alarms is considered to be the magnitude halfway between the means [18, p.4-6].

A threshold is set as the mid value of the auto-correlation peak and the tolerated cross-correlation peak between the selected ID codes (which is defined in Section **??**). 'write_ola_buffer' has code embedded within its loops to track to find the peak of the output. Post-processing, the peak of the input data is compared to the threshold magnitude to make a binary decision on the presence of the signal. Several such receptions will have to be triggered.

3.3. Optimization for Multiple Access

The problem of communicating with individual receivers is alluded to in Section **??**. This project is required to operate in a highly distortive channel, yet must be intolerant of false alarms. Furthermore, the intended recipient may not receive the strongest signal.

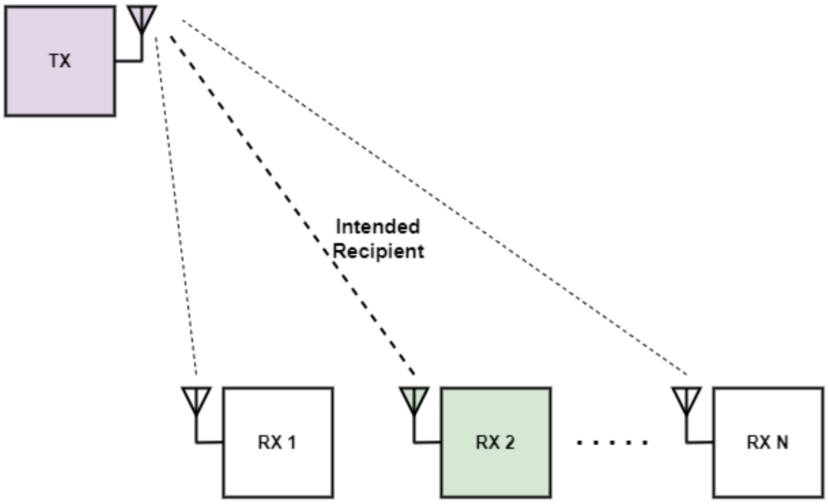


Figure 3.6: Caption

It is therefore optimal to determine a set of codes which have the least similarity when modulated so that false alarms are minimized. The receiver algorithm implemented on the microcontroller simply produces the cross-correlation of a signal. Thus, the opitmal set of messages those which, when modulated, produce the lowest cross-correlation peaks with each other.

Note, a Hamming Distance between codes (i.e. the number of different symbols) will not sufficiently decrease cross-correlation peaks of different messages because, as the lags vary, portions of the binary messages may be the same.

The iPython Notebook simulations related to this section are attached in Appendix C.

3.3.1. Predicting the Shape of a Modulated Bit Sequence

Initially, the plan was to iterate through the 4096 possible 12-bit sequences and perform a cross-correlation of each BFSK modulated message with the others for analysis. Unfortunately, the computational strain stalled the simulations; so a simplification was required.

Discrete Correlation of the Binary Code

Figure 3 demonstrates that the autocorrelation of binary messages cannot adequately predict the shape of the autocorrelation of the modulated message. Therefore, the discrete cross-correlation cannot be applied to binary messages. A customized function has been created to do exactly this.

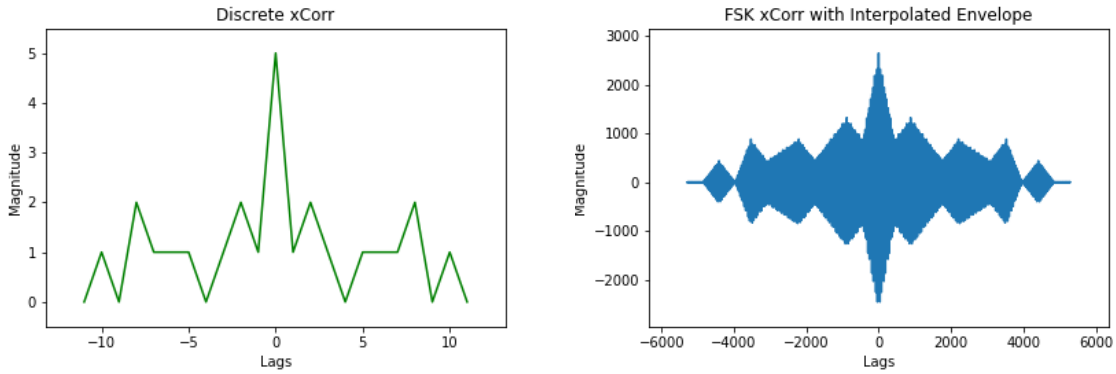


Figure 3.7: Code 101100010100 : Discrete Autocorrelation v.s. Autocorrelation of the modulated message

Discrete Correlation of FSK Modulated Bits

Beginning with the discrete auto-correlation of a modulated bit with itself, it can be shown that the maximum magnitude approximately evaluates to half the magnitude of the samples contained within the bit period interval.

$$
\begin{aligned}
r_{xx}[n] &= \sum_{i=0}^{N} cos(2\pi f T_s i) \cdot cos(2\pi f T_s i) \\
&\approx \int_0^N cos^2(2\pi f T_s i).di \\
&= \frac{sin(4\pi f T_s i)}{8\pi f T_s} + \frac{N}{2} \approx \frac{N}{2}
\end{aligned}
$$

However, the cross-correlation of a modulated '1' and '0' is approximately zero because the carrier frequencies are chosen to be orthogonal.

XNOR Correlation of Bits

A 12-bit BFSK modulated message can be described as a piecewise function of sinusoids of the same period. Therefore, the magnitude of the cross-correlation can be estimated by the number of bits which match during the correlation.

This observation implies that an envelope of the cross-correlation of various BFSK modulated signals can be predicted with just the binary symbols of the message. Intuitively, the number of matching bits is proportional to the cross-correlation of the FSK modulated message for each lag value. A customized discrete cross-correlation function has been written which applies XNOR logic rather than a multiplication to quantify the number of matching symbols per lag value.

Symbol A	Symbol B	XNOR
0	0	1
0	1	0
1	0	0
1	1	1

Table 3.2: XNOR Truth Table

The XNOR cross-correlation can be scaled and linearly interpolated to demonstrate its efficacy for FSK modulation.

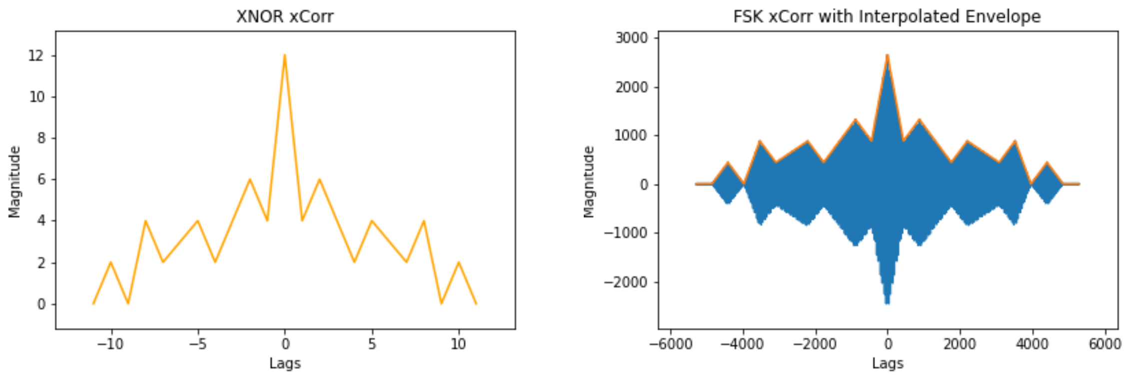


Figure 3.8: Code 101100010100 : XNOR Autocorrelation and the Interpolated Envelope

3.3.2. Optimized Modulation Scheme

Linear Chirps

Investigations in the use of linear chirps with FSK proved that this is an appealing method to mitigate ISI. An upchirp is applied to marks and a downchirp is applied to spaces. Equation 2.17 is used to determine the requisite chirp bandwidth from the difference in arrival of the LOS and strongest NLOS paths. The direct path from the transmitter to receiver receiver would be 40m and the strongest NLOS path is estimated to be no less than 60m. Therefore, the chirp bandwidth must be larger than 75 Hz and is chosen as 100Hz.

$$B > \frac{1}{\Delta t_{TOA}} = (\frac{d_{LOS} - d_{NLOS}}{v_{uw}})^{-1} = \frac{20m}{1500m/s}^{-1} = 75\,\mathrm{Hz}$$

Frequency Hopping

An additional carrier for each bit is selected; these carriers are centered at 1800 Hz and 2000 Hz to obey the orthogonality condition. The bandwidth of the receiver remains 600Hz. The system makes use of a simple hopping pattern: bits are modulated in an alternating pattern with one of two carriers associated with its value.

Binary number	Even or Odd	Carrier Frequency	Up or Down Chirp
0	even	$f_{00} = 1600$ Hz	down
0	odd	$f_{01} = 1800$ Hz	down
1	even	$f_{10} = 2000$ Hz	up
1	odd	$f_{11} = 2200$Hz	up

Table 3.3: Caption

For example, the binary message ”1, 0, 1, 0, 0, 1, 1” would be modulated as show in Figure 3.9. The arrow indicates the direction of the chirp.

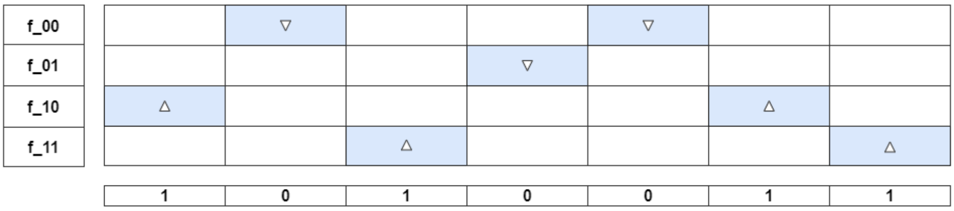


Figure 3.9: Modulation of a binary message

Now, bits of the same value may not be represented by carrier of the same centre-frequency, thus the binary message can not be passed to the XNOR crosscorrelation function to examine the cross-correlation pattern. Therefore, the digital data is first passed to a function which converts each bit to one of 4 symbols which represents its carrier frequency.

Influences on the Correlation Characteristics

Refer to Figure 3.10: Chirps vary the frequency during the symbol period, so the autocorrelation magnitude of the similarity of a signal with a time-shifted version of itself is greatly reduced; thus, spikes appear at lag values where symbols are aligned.

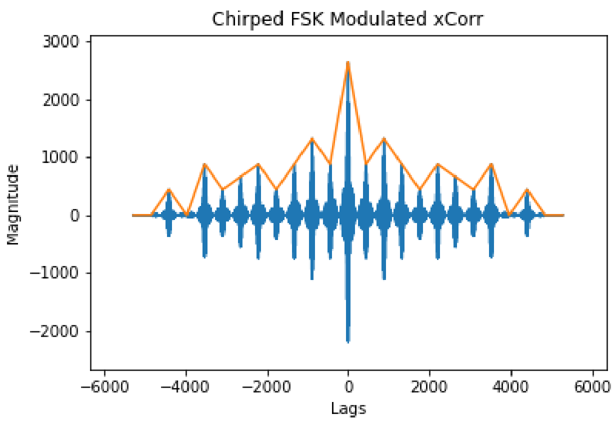


Figure 3.10: Code 101100010100: FSK with Linear Chirps

Refer to Figure 3.11: In the same way that the chirp reduces the similarity of a symbol with itself, frequency-hopping reduces the similarity of the overall message with itself. All 4 carriers are orthogonal, therefore, the sidelobes of the pattern are suppressed.

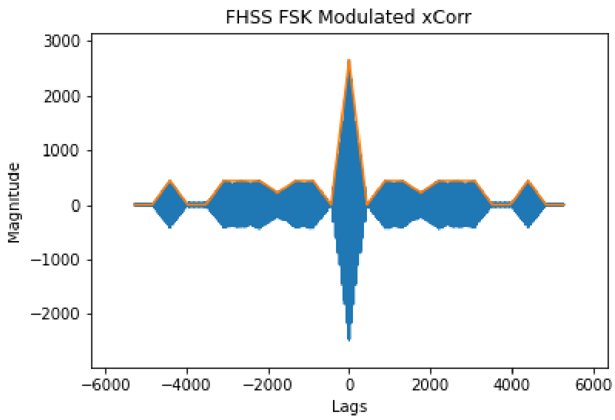


Figure 3.11: Code 101100010100: FSK with Frequency-Hopping

Finally, autocorrelation of a chirped, frequency-hopped FSK message observes the combined effects of both characteristics.

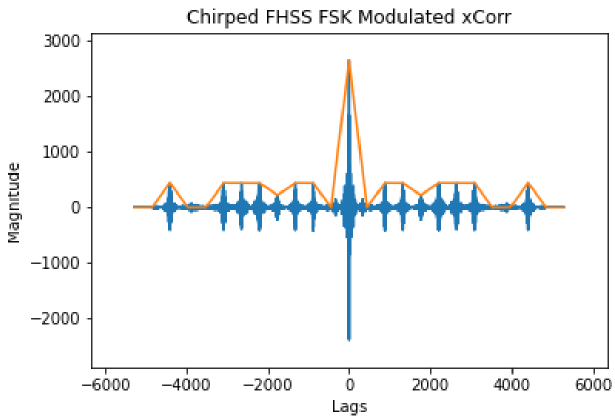


Figure 3.12: Code 101100010100: Chirped FSK with Frequency-Hopping

3.3.3. Obtaining a Set of Synchronization Codes

The objectives are to find a set of codes which have low sidelobe ratios and to discard sequences which have a cross-correlation peak above a tolerated value.The Python script associated with this can be found in Appendix ??.

Firstly, a two-dimensional matrix is generated. Each row of the matrix corresponds to an integer in the range 0 to 4095. Each integer is converted to a binary sequence and stored in the matrix.

```
BITS = 12
```

```
MAX = 2**BITS
vals = np.array(range(0, MAX))

num = 0
slr = num+1
bit_begin = slr+1
bit_end = bit_begin+BITS
rxx_begin = bit_end+1
rxx_end = rxx_begin + 2*BITS - 1

A = np.zeros( shape = (MAX, rxx_end))
A[:,num] = np.array(range(0, MAX))
A[:, bit_begin:bit_end] = vec_bin_array(arr = vals, m = BITS)
```

Next, a loop runs through each code. The binary sequences are converted to the FHSS symbol sequence and passed to the XNOR autocorrelation function and stored to the matrix. The result of the autocorrelation is used to calculate the sidelobe ratio of each code; then, the matrix is sorted according to sidelobe ratios. Now, the matrix is in an order which is biased to favour the codes with the lowest sidelobe ratio.

```
for i in vals:
  bin_seq = A[i, bit_begin:bit_end]
  cdma_seq = cdma_symbols_conv(bin_seq, fz, fi)
  cdma_rxx = xnor_autocorr(cdma_seq)
  A[i, rxx_begin:rxx_end] = cdma_rxx
  max_lobe = cdma_rxx[0]
  for k in range(1, BITS-1):
      if cdma_rxx[k] > max_lobe:
        max_lobe = cdma_rxx[k]
  A[i, slr] =  max_lobe/cdma_rxx[BITS-1]

A = A[A[:,slr].argsort()]
```

Beginning with the code with the lowest sidelobe ratio, each code is iteratively compared with the successive codes using the XNOR cross-correlation. If more than 5 of 12 symbols match, the code which is being compared is flagged for deletion and removed from the list of iterated codes.

```
xcor_thresh = 5
dels = []
bdel = np.zeros( shape = (A.shape[0],) )

for i in range(0, A.shape[0]):
  if bdel[i] == 0:

    bitseq1 = A[i, bit_begin:bit_end]
    fhss_seq1 = cdma_symbols_conv( bitseq1, fi, fz )

```

```
    for k in range(i+1, A.shape[0]):
      if bdel[k] == 0:

        bitseq2 = A[k, bit_begin:bit_end]
        fhss_seq2 = cdma_symbols_conv( bitseq2, fi, fz )

        rik = xnor_xcorr(fhss_seq1, fhss_seq2)
        rik_max = np.max(rik)
        if ( rik_max > xcor_thresh):
          bdel[k] = 1
          dels.append(k)

A_xcor5 = np.delete(A, obj = np.array(dels), axis = 0)
```

The final result is a set of 13 codes which do not produce a cross-correlation peak of more than $5 \times \frac{441}{2} = 1102.5$, when FHSS FSK modulation is applied. For comparison, the same process was repeated with BFSK modulation and only 2 codes were produced which fulfill the same requirements.

	Integer Number	Binary Sequence
1	1413	010110000101
2	2757	101011000101
3	3018	101111001010
4	1575	011000100111
5	1290	010100001010
6	889	001101111001
7	3724	111010001100
8	1082	010000111010
9	2481	100110110001
10	2676	101001110100
11	1239	010011010111
12	2399	100101011111
13	994	001111100010

Table 3.4: Low Cross-correlation Codes

Chapter 4

Measurements and Results

4.1. Characterizing the Anti-Alias Filter

An Anti Alias Filter is important for digital signal processing because frequencies above the Nyquist frequency alias. However, there is some dispute as to whether or not the Audio Adapter's ADC input is preceded by an anti-alias filter due to inadequate documentation. Therefore, a test was required to determine its existence.

4.1.1. Test Method

Sinusoidal waves of increasing frequency were input directly to the MIC_IN pin. A simple Arduino script exploits the 'peak' object from the Teensy Audio Library to quantify the frequency response. If an anti-alias filter is present, there will be rapid attenuation of frequencies higher than the Nyquist frequency ($\frac{44100}{2} = 22\,050\,\mathrm{Hz}$).

4.1.2. Generating Sine Waves

For lack of a function-generator, an STM32F334R8 developement board was used to generate sine waves. The board is equipped with a DAC peripheral which can be triggered to output samples in the range of 0 to 5V with a 12-bit resolution. An array is initialized with a set values corresponding to the voltages of one cycle of the sine wave; and, on each timer interrupt, the next value of the array is output on the pin. The interrupt timer period determines the frequency of the sinusoid.

Note that DAC has use the DMA controller so that waves above 15kHz can be generated. The DMA is set to circular buffer mode to avoid missed samples - so the last sine value is succeeded by the first value.

4.1.3. Results

A table of the peak input versus sine wave frequencies was recorded (Appendix **??**). It became apparent that an Anti-Alias filter is indeed present. The filter cuts in at 20 kHz and after 25.6 kHz input sine waves could not be detected.

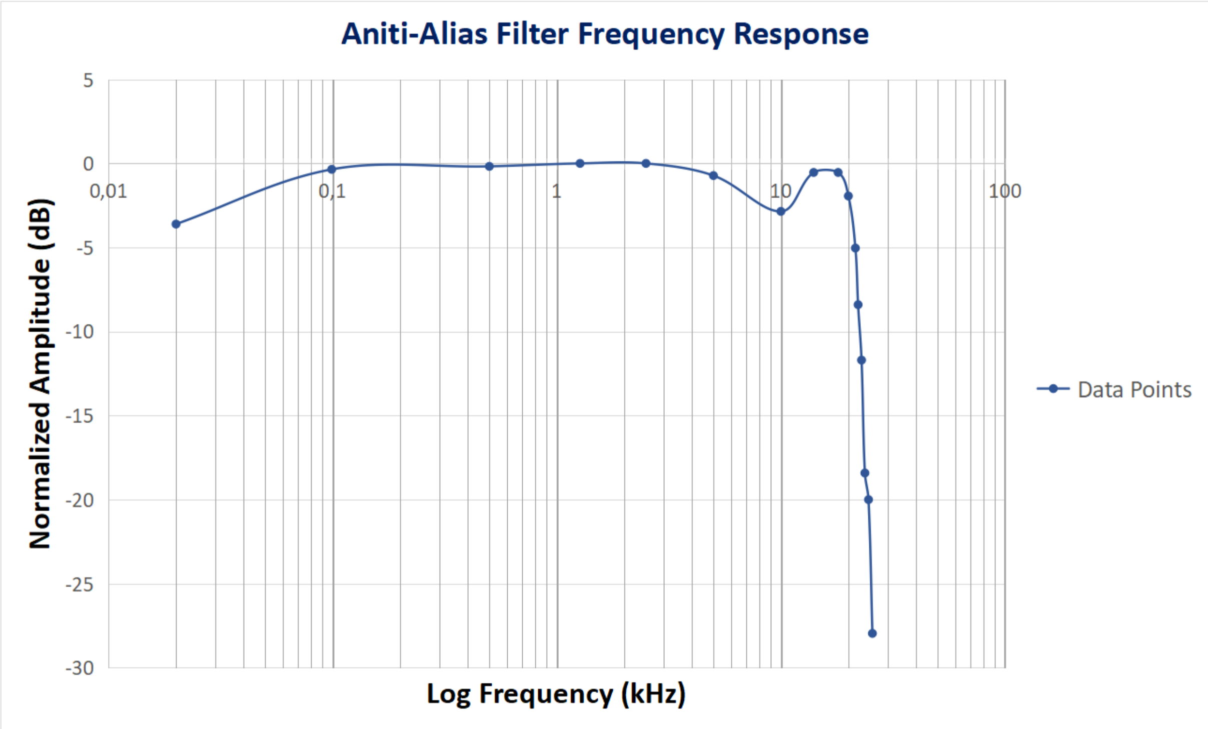


Figure 4.1: Anit Alias Filter Bode Plot

For further investigation, the ripples in the frequency response followed by a sharp transition band bears a strong resemblance to a 4th order Elliptic Filter.

4.2. Microcontroller Matched Filter Unit Test

The receiver needs to be tested to ensure that the major functional blocks are performing correctly. In particular: the data preprocessing, UPOLA algorithm, and output circular buffer management.

4.2.1. Method

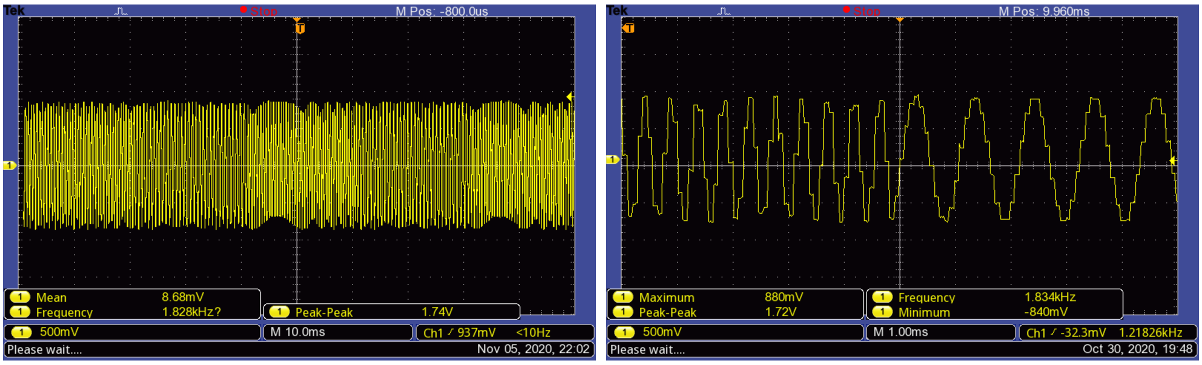
Modulated messages will be input direct to MIC_IN. If the matched filter output is greater than the threshold value, the received q15_t data, downsampled float32_t data, and overlap-add buffer contents are printed to serial data and saved to a text-file with Putty.

The text file is imported to MATLAB and split into the relevant data arrays. A MATLAB script (Appendix E) is used to test the functional blocks. First, a template is initialized and the autocorrelation output is displayed. Then, the downsampled data is passed to the MATLAB 'xcorr' function to compare with the UPOLA algorithm and microcontroller output buffer. The UPOLA outputs should be nearly identical, except for the final blocks which have not been overlapped.

4.2.2. Generating Modulated Messages

Once again the STM32F344R8 is used to generate modulated codes. An array of values is generated with code similar to that of the 'generateTemplate' method which was implemented on the receiver.

The DAC with DMA will repeatedly transmit the expected message; however, there will be some distortion given the voltage steps. Figure ?? depicts a portion of the modulated message and the resolution of the voltage steps.

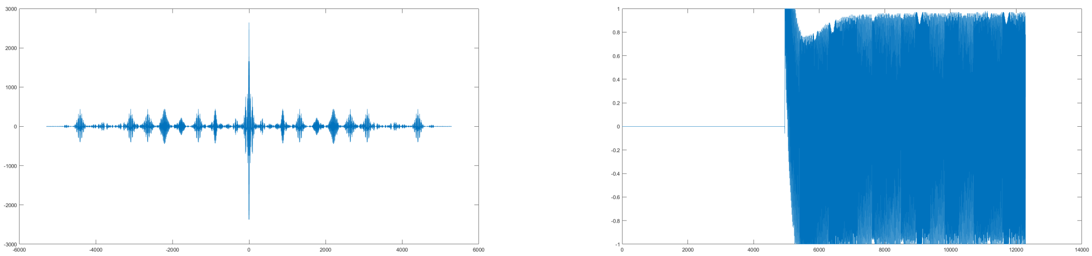


(a) **(b)**

Figure 4.2: Code 010110000101 : (a) Modulated code (b) Close-up of wave output

4.2.3. Results

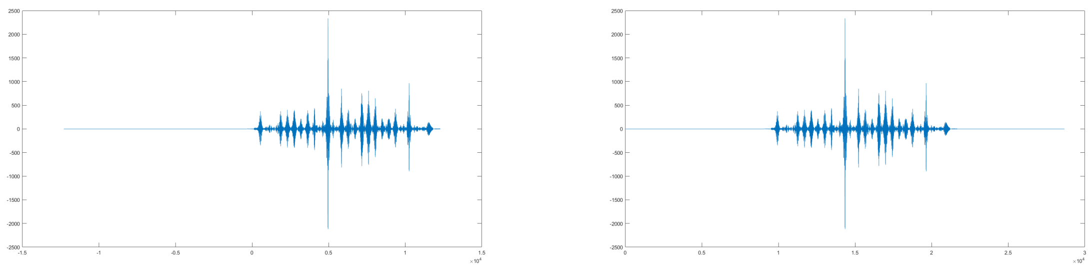
Firstly, the input signal, as shown in Figure ??, is sampled by the ADC and represented in q15_t format as a range of 32678 to -32678; then, the data is downsampled. The Audio Adapter's ADC preamplifier seems to have a negative gain, thus the template message must be multiplied by a factor of negative one.



(a) **(b)**

Figure 4.3: Code 010110000101 : (a) Autocorrelation of template (b) Received signal

The distortion on the generated message causes minor differences between the template's autocorrelation and the crosscorelation. However, the UPOLA algorithm clearly reproduces the cross-correlation for the blocks which have been processed and overlapped.



(a) **(b)**

Figure 4.4: Code 010110000101 : (a) MATLAB 'xcorr' function (b) MATLAB UPOLA algorithm

Finally, the Teensy output buffer meets the same conditions as the MATLAB UPOLA algorithm.

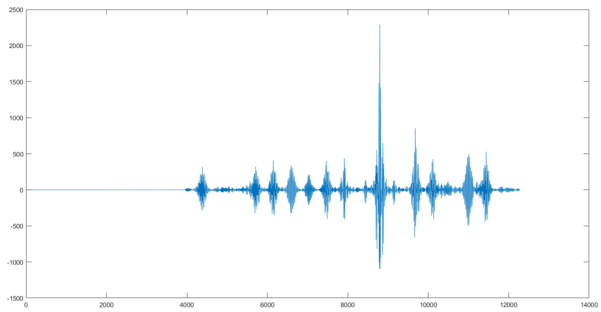


Figure 4.5: Teensy UPOLA output

4.3. Multiple Access Tests

The objective of this section is to characterize the matched filter detector's performance in harsh environments for the modulation scheme. Firstly, the intended receiver should have a fair probability of detecting a polluted message. Secondly, a polluted message must not induce false alarms in other receivers.

4.3.1. Method

This MATLAB simulation was conducted by performing the cross-correlation of modulated codes. One code is selected as the receiver and a template is generated. The other code is also modulated; but it is also polluted by the channel and Additive White Gaussian Noise - the AWGN is selected to have an SNR of 10 dB in relation to the message. Finally, the polluted message is limited to a range of ± 1.0 to mimic the clipping of the ADC. Each receiver is compared to each polluted code only 2000 times due to computational limitations; however, the statistical trends of detections and false-alarms is apparent.

Two scenarios were simulated which produce the least favourable conditions for reliable detection: high attenuation and severe multipath effects.

The high attenuation scenario distorts the message by attenuating it to a range of ± 0.76. The threshold of detection is set to 1874.25, which is the equivalent to perfectly detecting 8.5 of 12 bits; **or**, if a received signal is attenuated such that the floating point representation of the microphone input is in the range ± 0.71. For comparison, the preliminary BFSK modulation is simulated as well.

The multipath effects cause a large distortion which poses a risk of false alarms. In Section 3.3.2 the bandwidth of the chirp was decided according to the difference of arrival time between the LOS path and the strongest NLOS path; a difference of 13.3ms = 558 samples was assumed and is similarly used for this simulation. The LOS signal is chosen to have a gain of 0.7 and the reflection has a gain of 0.4 such that the signal will saturate at the receiver.

4.3.2. Results

The full output for each simulation of the optimal codes is contained in Appendix F.2. Table 4.1 contrasts each scenario for a single receiver to indicate the effectiveness of the modulation techniques, resistance to multipath, and overall detection rate. For this code, column ”1” indicates the number of times that the receiver correctly detected a message intended for it whereas columns ”2” to ”13” indicate the number of times a polluted message incorrectly passed the detection threshold.

Modulation	Scenario	1	2	3	4	5	6	7	8	9	10	11	12	13
BFSK	Attenuated	896	0	0	0	0	0	0	0	0	0	0	0	0
Optimized	Attenuated	1161	0	0	0	0	0	0	0	0	0	0	0	0
Optimized	Multipath	2000	0	0	0	0	0	0	0	0	0	0	0	0

Table 4.1: Code # 1 : Detection and False Alarms

This shows that the optimized modulation scheme (Chirped FHSS FSK) outperforms BFSK for high attenuation. The false alarm rate for the optimized scheme is demonstrably negligible - or at least of an order smaller than 10^{-4}.

In conclusion, the optimized modulation scheme is likely to be adequate for the severe environment in which the matched filter detector must operate. It displays resistance against false alarms and missed detections for the challenges of high SNR, severely attenuated, and multipath channels.

Chapter 5

Summary and Conclusion

5.1. Project Achievements

The communication system has been designed and tested to fulfill the primary objectives of the project. The product of the investigations and designs is summarized by four outcomes: an optimal modulation scheme, a matched filter detection algorithm, the microcontroller implementation of the detector, and a set of codes which can be used for identification.

The modulation scheme is heavily dependent on factors which must be chosen both at the discretion of the designer and according to the hardware limitations. Distortion by the underwater was estimated to be severe and compensated with thoroughly researched adjustments - linear chirps and frequnecy-hopping.

The matched filter detector is proven to produce the expected outputs. Based on simulations of signal distortion, the matched filter should cope in underwater environments.

The set of 13 codes can be used to address multiple buoys with low likelihood of producing false-alarms. The number of buoys can be expanded by tolerating a higher cross-correlation peak between codes and/or by employing pseudo-random hopping patterns for each receiver.

By my own initiative, I maximized computation efficiency at each stage of the design so as to ensure that future development can progress with minimal limitations by my contribution. This includes the MATLAB scripts and iPython Notebooks which require many iterations to obtain results; redundancy is removed, except where intended, and the user-defined functions can be easily adjusted for future designers.

5.2. Further Development

There are three activities which should occur before a release system can be deployed: testing in an underwater environment, electromechanical system design, and power consumption optimization.

Firstly, there was not an opportunity to integrate a acoustic transducers into the project for testing in an underwater environment. Because this project was largely an algorithmic design, the parameters were selected according to which hardware is intended to be used. The system was designed with the H1c acoustic transducers in mind; however the Signal Level and Sensitivity of these transducers is limiting. A projector device which can project 150 dB signals should be sourced. Similarly, a hydrophone device with a sensitivity of at least -180 dB would have sufficient gain for the intended use case scenario - if the Teensy's microphone input gain stage is set to 63 dB. A redundancy safety protocol can be included; the number of detections required to trigger the release may be increased and may even be required to occur within a fixed time period.

The development of an electromechanical release system must be completed with multidisciplinary input. The microcontroller code includes a demarcated "Release Control" section so the release's control system can be included after the detector threshold is satisfied.

For the purpose of this project, optimization of the power consumption should be considered to prolong battery life. Plenty of steps were taken to reduce the calculation complexity and exploit interrupts for efficiency purposes. The execution times per CPU speed are already tabulated, all that needs to be done is to power the board with a battery and use a current transducer (e.g. ZXCT1008) to compare the power consumption per CPU speed.

For further improvement of the system, I recommend investigating: spectrogram matched filter detection, real-time operating systems such as FreeRTOS [19], and adaptive signal processing [20].

Bibliography

[1] Swati Thiyagarajan . (2019, June) Octopus fishing in false bay is killing bryde's whales, and with them, a magical and unseen kingdom. [Online]. Available: https://www.dailymaverick.co.za/article/2019-06-17-octopus-fishing-in-false-bay-is-killing-brydes-whales-and-with-them-a-magical-and-unse

[2] Jenni Evans . (2019, November) New octopus fishing rules imposed to save cape town's whales. [Online]. Available: https://www.news24.com/news24/SouthAfrica/News/new-octopus-fishing-rules-imposed-to-save-cape-towns-whales-20191108

[3] M. Stojanovic et al. , *Ocean Engineering*. Springer, 2016, ch. 4.

[4] D. E. Lucani, M. Stojanovic, and M. Medard, "On the relationship between transmission power and capacity of an underwater acoustic communication channel," in *OCEANS 2008 - MTS/IEEE Kobe Techno-Ocean*, 2008, pp. 1–6.

[5] Gunilla Burrowes and Jamil Y. Khan, "Short-range underwater acoustic communication networks," in *Autonomous Underwater Vehicles*, N. A. Cruz, Ed. Rijeka: IntechOpen, 2011, ch. 8. [Online]. Available: https://doi.org/10.5772/24098

[6] Stephen C. Butler , "Properties of transducers: Underwater sound sources and receivers," Tech. Rep.

[7] F. Salahdine, H. E. Ghazi, N. Kaabouch, and W. F. Fihri, "Matched filter detection with dynamic threshold for cognitive radio networks," in *2015 International Conference on Wireless Networks and Mobile Communications (WINCOM)*, 2015, pp. 1–6.

[8] Lathi and Ding , *Modern Digital and Analog Communication Systems 4E*. Oxford University Press, 2010.

[9] E. Kaminsky and L. Simanjuntak, "Chirp slope keying for underwater communications," in *Proc. SPIE Sensors, and Command, Control, Communications, and Intelligence*, vol. 5778, 2005, pp. 894–905.

[10] W. Smith and J. Smith , *Handbook of Real-Time Fast Fourier Transforms*. THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, 1995.

[11] J. Proakis and D. Manolakis , *Digital Signal Processing 4E*. Pearson, 2014.

[12] .

[13] ARM Holdings . (2020) Cmsis. [Online]. Available: https://developer.arm.com/tools-and-software/embedded/cmsis

[14] ——. Arm developer suite axd and armsd debuggers guide. [Online]. Available: https://developer.arm.com/documentation/dui0066/d/axd/axd-facilities/data-formatting/q-format

[15] ——. Complex fft functions. [Online]. Available: https://www.keil.com/pack/doc/CMSIS/DSP/html/group__ComplexFFT.html

[16] ——. Real fft functions. [Online]. Available: https://www.keil.com/pack/doc/CMSIS/DSP/html/group__RealFFT.html

[17] Thomas Lorenser , "The dsp capabilities of arm® cortex®-m4 and cortex-m7 processors," Tech. Rep., November 2016. [Online]. Available: https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/white-paper-dsp-capabilities-of-cortex-m4-and-cortex-m7

[18] N. Ravirala, "Device signal detection methods and time frequency analysis," 2007.

[19] J. Desvignes . Freertos - teensy 4. [Online]. Available: https://libraries.io/platformio/FreeRTOS-Teensy4

[20] E. Demirors, G. Sklivanitis, G. E. Santagati, T. Melodia, and S. N. Batalama, "Design of a software-defined underwater acoustic modem with real-time physical layer adaptation capabilities," in *Proceedings of the International Conference on Underwater Networks Systems*, ser. WUWNET '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2671490.2674473

Appendix A

Project Planning Schedule

Week	Date	Project progression
1	27 Jul	Project definition and analysis of similar systems
2	03 Aug	Research into underwater acoustic channels and Telecommunications theory.
3	10 Aug	Research into modulation and matched filter detection algorithms - also with respect to multiple access communcations.
4	17 Aug	Familiarization with microcontroller and CMSIS DSP libraries
5	24 Aug	Simulation of detector algorithms
6	31 Aug	Detector algorithm optimization
Tests	7 Sep	- -
Recess	14 Sep	Microcontroller Implementation
7	21 Sep	Microcontroller Implementation
8	28 Sep	Investigation into known synchronization codes.
9	5 Oct	Obtain a set of unique ID codes.
10	12 Oct	Report - Introduction and Literature Review
11	19 Oct	Report - Design
12	26 Nov	Report - Measurements
13	2 Nov	Report - Conclusion and Appendices. Consultation with supervisor and KENAKO Writing Lab

Table A.1: Project Planning Schedule

Appendix B

SNR Estimation

$$
\begin{aligned}
&f = 1.6\text{kHz}\\
&r = 40\text{m}\\
&k = 2\\
&s = 0.5\\
&w = 22\text{km/h} = 6.11\text{m/s}\\
&BW = 600\text{Hz}
\end{aligned}
$$

$$
\begin{aligned}
&TVR = 110\text{dB}\\
&V_{in} = \frac{V_{pk}}{\sqrt{2}} = \frac{15}{\sqrt{2}} = 10.61V_{RMS}\\
&SL = TVR + 20\log V_{in} = 130.51\text{dB}
\end{aligned}
$$

$$
PL = 10\log(PathLoss(k, r)) = k \cdot 10log(r) = 20\log(40) = 32.04
$$

$$
\begin{aligned}
&10\log N_t = 10.88\text{dB}\\
&10\log N_s = 32.58\text{dB}\\
&10\log N_w = 60.58\text{dB}\\
&10\log N_{th} = -10.92\text{dB}\\
&NL_0 = 10\log(\sum Noise(f, w, s)) = 60.59\text{dB}\\
&NL = NL_0 + 10\log BW = 60.59 + 27.78 = 88.37
\end{aligned}
$$

$$
SNR = SL - PL - NL = 130.51 - 32.04 - 88.37 = 10.10\text{dB}
$$

Similarly, for $f = 2.2$ kHz, SNR = 11.89 dB.

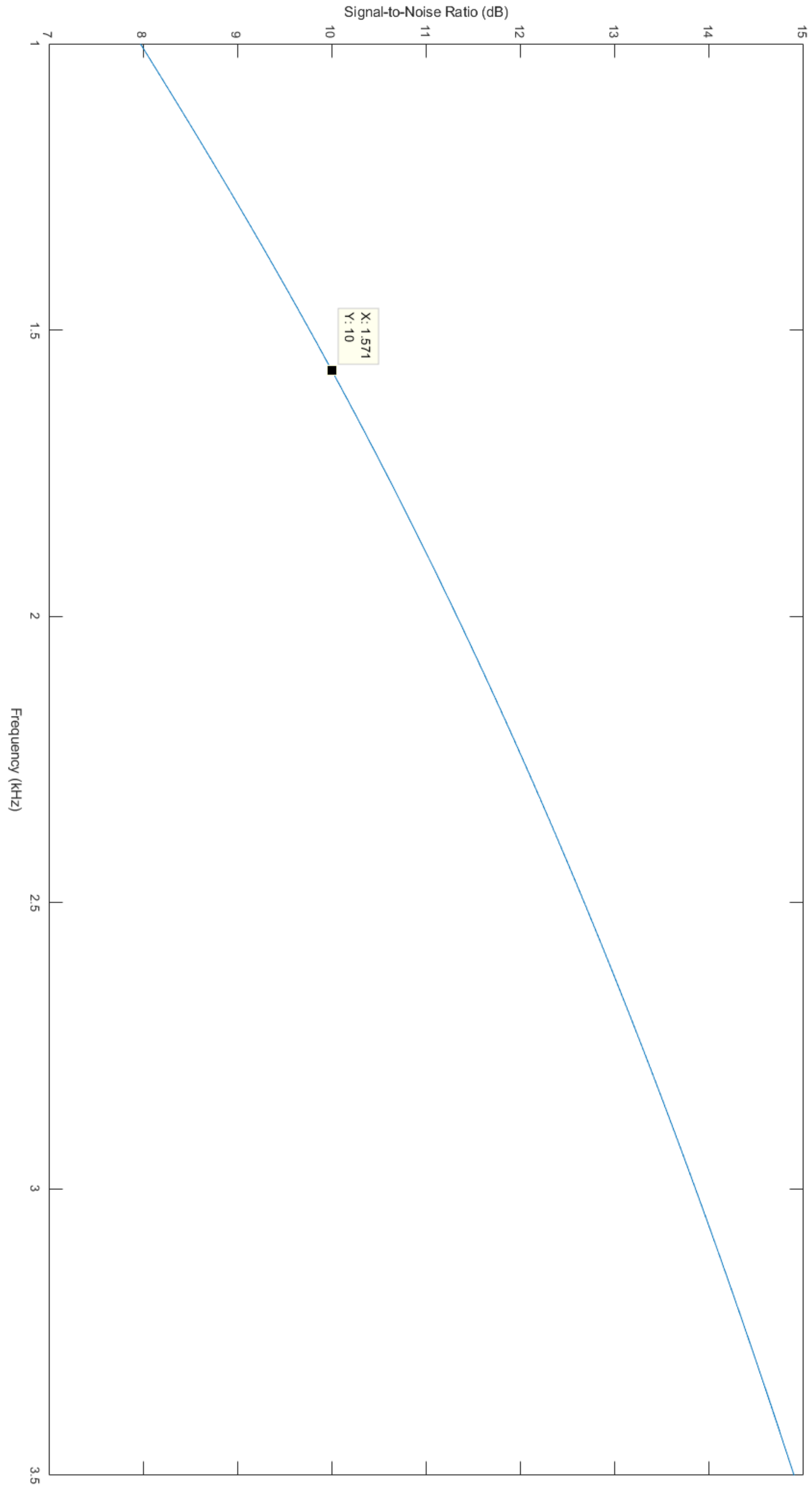


Figure B.1: SNR vs Frequency

Appendix C

Microcontroller Implementation

```
#include <stdint.h>

#include "arm_math.h"
#include "arm_const_structs.h"

#define DOWN_SAMP 2
#define BIT_LEN 441 // SAMPLES PER SYMBOL    DOWN_SAMP = 44.1kHz/20ms    2 = 441

#define L_LEN 2048
#define N_LEN 4096
#define B_BLOCKS 3
#define PACKETS 16 // 2048 SAMPLES    128 SAMPLES per PACKET

#include <Audio.h>
#include <Wire.h>
#include <SPI.h>
#include <SD.h>
#include <SerialFlash.h>

// GUItool: begin automatically generated code
AudioInputI2S            i2s2;
AudioMixer4              mixer1;
AudioOutputI2S           i2s1;
AudioRecordQueue         queue1;
AudioConnection          patchCord1(i2s2, 0, mixer1, 0);
AudioConnection          patchCord2(i2s2, 1, mixer1, 1);
AudioConnection          patchCord3(mixer1, 0, i2s1, 0);
AudioConnection          patchCord4(mixer1, 0, i2s1, 1);
AudioConnection          patchCord5(mixer1, queue1);
AudioControlSGTL5000     sgtl5000_1;
// GUItool: end automatically generated code

uint8_t bstart = 1;

uint8_t flag = 1;
```

```
int16_t fdata_head = 0;

int8_t detected_thresh = 0;
float corr_thresh = ((12+7)/2)*BIT_LEN/2;

const float f1 = 2000;
const float f0 = 1000;

float32_t mf_fft[B_BLOCKS][2*N_LEN] = {{0}};
float32_t rx_fft[B_BLOCKS][2*N_LEN] = {{0}};

const int ola_max = 3*N_LEN;
float32_t ola_buffer[ola_max] = {0};

int ola_head = N_LEN-L_LEN;
const int ola_overlap = N_LEN-L_LEN;

float32_t ola_peak = 0;
int ola_peak_index = 0;


/*SETUP*/
void setup() {
  // put your setup code here, to run once:
  float32_t mf_msg[B_BLOCKS][N_LEN] = {{0}};

  Serial.begin(300);

  AudioMemory(100);
  sgtl5000_1.enable();
  sgtl5000_1.inputSelect(AUDIO_INPUT_MIC);
  sgtl5000_1.micGain(0);
  sgtl5000_1.unmuteHeadphone();
  sgtl5000_1.audioPreProcessorEnable();
  sgtl5000_1.audioPostProcessorEnable();
  sgtl5000_1.volume(0.3);

  while (!Serial) {}

  const int bit_num = 12;

  int bitseq[bit_num] = {0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1};

  float space = f0*DOWN_SAMP;
  float mark = f1*DOWN_SAMP;
  generateTempate(mf_msg, bitseq, bit_num, 44100, BIT_LEN , space, mark)
```

```
  ;

  for(int p = 0; p < B_BLOCKS; p++)
  {
    f32tRealFFT(mf_fft[p], mf_msg[p], N_LEN);
  }

  queue1.begin();
}


/*LOOP*/
void loop()
{
  int t0, t1;

  asm(" WFI");

  if(bstart == 1)
  {
    if(detected_thresh < 1)
    {
      q15_t q_rx_msg[L_LEN*DOWN_SAMP] = {0};

      if (flag == 1)
      {
        if (queue1.available() >= PACKETS*DOWN_SAMP)
        {
          t0 = micros();
          for (int k = 0; k < PACKETS*DOWN_SAMP; k++)
          {
            memcpy(q_rx_msg + 128*k, queue1.readBuffer(), 256);
            queue1.freeBuffer();
          }

          flag = 0;
        }
      }


      if (flag == 0)
      {
        flag = 1;
        UPOLA(q_rx_msg);
        check_detect();
        t1 = micros();
        Serial.println(t1-t0);
```

```
      }


    }
    else
    {
      queue1.clear();
      queue1.end();
      bstart = 0;

      /*PRINT*/
      for(int m = 0; m < ola_max; m++)
      {
        int indx = ola_head + m;
        if(indx >= ola_max)
        {
          indx -= ola_max;
        }
        Serial.println( ola_buffer[indx] );
      }
    }
  }
}


/*DETECTION*/
void check_detect()
{
  if(ola_peak >= corr_thresh)
  {
    detected_thresh += 1;

    //int dist_LOW = BIT_LEN;
    float LOW_max_val = 0;

    for(int q = -10; q < 10; q++)
    {
      int index = (ola_peak_index - LOW_max_val) + q;
      if( index < 0 )
      {
        index += ola_max;
      }
      if ( abs( ola_buffer[index] ) > LOW_max_val )
      {
        LOW_max_val = abs( ola_buffer[index] );
      }
    }
```

```
    if( LOW_max_val < (ola_peak/10) )
    {
      detected_thresh += 1;
    }

  }

}


/*MATCHED FILTER*/
void generateTempate(float32_t out_fsk_modulated[][N_LEN], int*
    in_bit_sequence, int num_bits, float fs, int samples_per_bit, float
    f0, float f1)
{
  float f_mux;
  float f_chirp = 100*DOWN_SAMP;
  float f_add = 250*DOWN_SAMP;

  const int fhss_num = 2;
  float fhss_0[ fhss_num ] = {0};
  float fhss_1[ fhss_num ] = {0};

  int tally_0 = 0;
  int tally_1 = 0;

  for(int m = fhss_num; m >= 0; m--)
  {
    fhss_0[m] = f0 + m*f_add;
    fhss_1[m] = f1 - m*f_add;
  }


  int modu_len = samples_per_bit*num_bits;
  float32_t pre_modulated[ modu_len ] = {0};
  float32_t reversed_modu[ modu_len ] = {0};

  for (int i = 0; i < num_bits; i++)
  {
    int index = (int)(i * samples_per_bit);

    if (in_bit_sequence[i] == 0)
    {
      for (int cnt = 0; cnt < samples_per_bit; cnt++)
      {
        f_mux = fhss_0[ tally_0 % fhss_num ] - f_chirp*cnt/
```

```
samples_per_bit - f_chirp/2;
        pre_modulated[index + cnt] = -1*arm_sin_f32( (float32_t)(2 * PI * f_mux * cnt ) / fs );
      }
      tally_0++;
    }
    else
    {
      for (int cnt = 0; cnt < samples_per_bit; cnt++)
      {
        f_mux = fhss_1[ tally_1 % fhss_num ] + f_chirp*cnt/samples_per_bit - f_chirp/2;
        pre_modulated[index + cnt] = -1*arm_sin_f32( (float32_t)(2 * PI * f_mux * cnt ) / fs );
      }
      tally_1++;
    }

  }

  for(int k = 0; k < modu_len; k++)
  {
    reversed_modu[k] = pre_modulated[modu_len - k - 1];
  }

  for (int p = 0; p < B_BLOCKS; p++)
  {
    for (int j = 0; j < L_LEN; j++)
    {
      int indx = p*L_LEN;

      if(indx+j < modu_len)
      {
        out_fsk_modulated[p][j] = reversed_modu[indx + j];
      }
      else
      {
        out_fsk_modulated[p][j] = 0.0;
      }

    }
  }
}


/*BLOCK CONVOLVER*/
void UPOLA(q15_t* q15_data)
```

```
{
  float32_t y_fft[2*N_LEN] = {0};
  float32_t rx_msg[L_LEN*DOWN_SAMP] = {0};
  float32_t rx_down[N_LEN] = {0};

  arm_q15_to_float(q15_data, rx_msg, L_LEN*DOWN_SAMP);
  downsample(rx_down, rx_msg, L_LEN);

  /*for(int k = 0; k < DOWN_SAMP*L_LEN; k++)
  {
    Serial.println( q15_data[k] );
  }

  for(int k = 0; k < L_LEN; k++)
  {
    Serial.println( rx_down[k] );
  }*/

  f32tRealFFT(rx_fft[fdata_head], rx_down, N_LEN);


  float32_t sum_msg[N_LEN] = {0};
  float32_t sum_fft[2*N_LEN] = {0};


  int index;
  for(int k = 0; k < B_BLOCKS; k++)
  {
    index = fdata_head-k;

    if(index < 0)
    {
      index += B_BLOCKS;
    }

    arm_cmplx_mult_cmplx_f32(rx_fft[index], mf_fft[k], y_fft, N_LEN);

    for(int j = 0; j < 2*N_LEN; j++)
    {
      sum_fft[j] += y_fft[j];
    }
  }
  f32tRealIFFT(sum_msg, sum_fft, N_LEN);



  if(fdata_head >= B_BLOCKS-1)
```

```
  {
    fdata_head = 0;
  }
  else
  {
    fdata_head++;
  }

  write_ola_buffer(sum_msg);
}


/*DOWN-SAMPLING*/
void downsample(float32_t* out_arr, float32_t* in_arr, int len)
{
  int cnt = 0;
  for(int k = 0; k < len; k++)
  {
    out_arr[k] = in_arr[cnt];
    cnt += DOWN_SAMP;
  }
}


/*WRITE TO OLA_BUFFER*/
void write_ola_buffer(float32_t* input)
{
  ola_head -= ola_overlap;

  if(ola_head < 0)
  {
    ola_head += ola_max;
  }

  for(int i = 0; i < ola_overlap; i++)
  {
    ola_buffer[ola_head] +=  input[i];

    if ( (abs(ola_buffer[ola_head]) > ola_peak) )
    {
      ola_peak_index = ola_head;
      ola_peak = abs(ola_buffer[ola_head]);
    }

    ola_head++;
    if(ola_head >= ola_max)
    {
```

```
      ola_head = 0;
    }
  }

  for(int i = ola_overlap; i < N_LEN; i++)
  {
    ola_buffer[ola_head++] = input[i];

    if(ola_head >= ola_max)
    {
      ola_head = 0;
    }
  }

}


/*FFT f32_t*/
void f32tRealFFT(float32_t* out_fsk_fft, float32_t* in_fsk_msg, int
    in_len)
{
  arm_rfft_fast_instance_f32 fastfft32;

  arm_rfft_fast_init_f32(&fastfft32, in_len);

  arm_rfft_fast_f32(&fastfft32, in_fsk_msg, out_fsk_fft, 0);
}


/*IFFT f32_t*/
void f32tRealIFFT(float32_t* out_fsk_msg, float32_t* in_fsk_fft, int
    out_len)
{
  arm_rfft_fast_instance_f32 fastfft32;

  arm_rfft_fast_init_f32(&fastfft32, out_len);

  arm_rfft_fast_f32(&fastfft32, in_fsk_fft, out_fsk_msg, 1);
}
```

Appendix D

STM32 Function Generator

```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "math.h"
```

```c
#include <stdbool.h>
/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

#define BIT_LEN 221
#define BIT_NUM 12

#define DSAMP 4

#define L_LEN 1042
#define N_LEN 2048

#define PI 3.141592
#define FS 44100

#define Ns 100

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
DAC_HandleTypeDef hdac1;
DMA_HandleTypeDef hdma_dac1_ch1;

TIM_HandleTypeDef htim2;

/* USER CODE BEGIN PV */

uint32_t sine_val[ Ns ];

/* USER CODE END PV */

```

```c
/* Private function prototypes
    -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_DAC1_Init(void);
static void MX_TIM2_Init(void);
/* USER CODE BEGIN PFP */

void get_sine_val(void);
void generateFSK2D(uint32_t* out_fsk_modu, int* in_bit_sequence, int
    num_bits, float fs, int samples_per_bit, float f0, float f1);

/* USER CODE END PFP */

/* Private user code
    ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

void get_sine_val(void)
{
  for(int i = 0; i < Ns; i++)
  {
    sine_val[i] = ( (sin(2*PI*5*i/Ns) + 1.1) )*(4096/8);
  }
}

void generateFSK2D(uint32_t* out_fsk_modu, int* in_bit_sequence, int
    num_bits, float fs, int samples_per_bit, float f0, float f1)
{
  float f_mux;

  float f_chirp = 100*DSAMP;
  float f_add = 250*DSAMP;

  const int fhss_num = 2;
  float fhss_0[ 2 ] = {0};
  float fhss_1[ 2 ] = {0};

  int tally_0 = 0;
  int tally_1 = 0;

  for(int m = fhss_num; m >= 0; m--)
  {
    fhss_0[m] = f0 + m*f_add;
    fhss_1[m] = f1 - m*f_add;
  }
```

```

  for (int i = 0; i < num_bits; i++)
  {
    int index = (int)(i * samples_per_bit);

    if (in_bit_sequence[i] == 0)
    {
    for (int cnt = 0; cnt < samples_per_bit; cnt++)
    {
      f_mux = fhss_0[ tally_0 % fhss_num ] - f_chirp*cnt/samples_per_bit - f_chirp/2;
      //f_mux = fhss_0[ 0 ] - DSAMP*50*cnt/samples_per_bit;
      out_fsk_modu[index + cnt] =  (uint32_t)( ( 1.1 + sin( (float)(2 * PI * f_mux * cnt) / fs ) )*(4096/4) );
    }
    tally_0++;
    }
    else
    {
      for (int cnt = 0; cnt < samples_per_bit; cnt++)
      {
        f_mux = fhss_1[ tally_1 % fhss_num ] + f_chirp*cnt/samples_per_bit - f_chirp/2;
        //f_mux = fhss_1[ 0 ] + DSAMP*50*cnt/samples_per_bit;
        out_fsk_modu[index + cnt] =  (uint32_t)( ( 1.1 + sin( (float)(2 * PI * f_mux * cnt) / fs ) )*(4096/4) );
      }
      tally_1++;
    }
  }

}

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration --------------------------------------------------------*/
```

```
  /* Reset of all peripherals, Initializes the Flash interface and the
   Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_DAC1_Init();
  MX_TIM2_Init();
  /* USER CODE BEGIN 2 */
     //[1,0,1,1,0,1,1,0,1,1,0,1]
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  //int bit_seq[BIT_NUM] = {1,0};
  //int bit_seq[BIT_NUM] = {1,0,1,0,1,0};

  //int bit_seq[BIT_NUM] = {1,0,1,0,1,0,1,0,1,0,1,0};
  //int bit_seq[BIT_NUM] = {1,1,0,1,0,0,1,1,1,1,0,0};

  //int bit_seq[BIT_NUM] = {1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0};
  //int bit_seq[BIT_NUM] = {1, 1, 0, 0, 1, 0, 1, 0, 1};

  int bit_seq[BIT_NUM] = {0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1};
  //int bit_seq[BIT_NUM] = {0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1};

  uint32_t mf_msg[BIT_NUM*BIT_LEN];

  generateFSK2D(mf_msg, bit_seq, BIT_NUM, FS, BIT_LEN, 1000*DSAMP, 2000*
   DSAMP);

  //get_sine_val();

  HAL_TIM_Base_Start(&htim2);
```

```
  HAL_DAC_Start_DMA(&hdac1 , DAC_CHANNEL_1, mf_msg, BIT_NUM*BIT_LEN,
   DAC_ALIGN_12B_R);

  //HAL_DAC_Start_DMA(&hdac1 , DAC_CHANNEL_1, sine_val, Ns,
   DAC_ALIGN_12B_R);

  while (1)
  {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL16;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK
```

```
  )
  {
    Error_Handler();
  }
}

/**
  * @brief DAC1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_DAC1_Init(void)
{

  /* USER CODE BEGIN DAC1_Init 0 */

  /* USER CODE END DAC1_Init 0 */

  DAC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN DAC1_Init 1 */

  /* USER CODE END DAC1_Init 1 */
  /** DAC Initialization
  */
  hdac1.Instance = DAC1;
  if (HAL_DAC_Init(&hdac1) != HAL_OK)
  {
    Error_Handler();
  }
  /** DAC channel OUT1 config
  */
  sConfig.DAC_Trigger = DAC_TRIGGER_T2_TRGO;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
  if (HAL_DAC_ConfigChannel(&hdac1, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN DAC1_Init 2 */

  /* USER CODE END DAC1_Init 2 */

}

/**
  * @brief TIM2 Initialization Function
  * @param None
```

```c
  * @retval None
  */
static void MX_TIM2_Init(void)
{

  /* USER CODE BEGIN TIM2_Init 0 */

  /* USER CODE END TIM2_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM2_Init 1 */

  /* USER CODE END TIM2_Init 1 */
  htim2.Instance = TIM2;
  htim2.Init.Prescaler = 0;
  htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 1451*DSAMP; //1451*DSAMP
  htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
    HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM2_Init 2 */

  /* USER CODE END TIM2_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
```

```
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel3_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel3_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel3_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOA_CLK_ENABLE();

}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return
     state */

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line
     number
  *         where the assert_param error has occurred.
```

```
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and
    line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file,
     line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF
    FILE****/
```

Appendix E

Microcontroller Unit Test

```
%--------------------------------%
%             Setup              %
%--------------------------------%

fs = 44100;
Ts = 1/fs;

dsamp = 2;
Tb = 20e-3;
samps = Tb*fs;

n = 1;
delta_f = n/(2*Tb);

f1 = 2000;
f0 = 1000;
f1 = [f1 f1-250];
f0 = [f0 f0+250];

L = 2048
N = 4096
K = 3

bits = [0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1];

%Import Data from Table
T = table2array(MHztest);
T_len = size(T, 1)

%Display Teensy UPOLA Buffer Output
corr_out = T(T_len - K*N + 1 : T_len);
figure('Name', 'Teensy UPOLA Output')
plot(corr_out)

%Generate Template
tx = -1*cdma_chirp_fsk(bits, samps/2, f0*dsamp, f1*dsamp, fs, dsamp*100)
```

```
    ;
figure('Name', 'MATLAB TX Message')
plot(tx)

%Separate Downsampled Float Data and Display
rx = T(T_len - 2*K*N + 1 : T_len - K*N);
rx = reshape(rx, 1, K*N);
figure('Name', 'Teensy RX Downsampled F32 Message')
plot(rx)

%Display Auto-Corr of Template
[rtt,lags] = xcorr(tx, tx);
figure('Name', 'MATLAB xcorr self')
plot(lags, rtt)

%Display Expected xCorr of Template & Downsampled Float Data
[rtr,lags] = xcorr(rx, tx);
figure('Name', 'MATLAB xcorr output')
plot(lags, rtr)

%--------------------------------%
%MATLAB script of UPOLA Algorithm%
%--------------------------------%

%Partitioned Filter FFTS
mf_coef= flip(tx);
mf_coef_n = zeros(K,N);
mf_fft_n = zeros(K,N);

for j  = 1:K
    start = (j-1)*L;
    stops = start+L;
    if (stops > size(mf_coef,2))
        stops = size(mf_coef,2);
    end
    mf_coef_n(j,:) = [mf_coef((start+1):stops) zeros(1, N-(stops-start))
   ];
    mf_fft_n(j,:) = fft( mf_coef_n(j,:) );
end


%Zeropad & Read RX Data into arrays
xf = zeros(K*4,N);
xn = zeros(K*4,N);
yf = zeros(K*3,N);
yn = zeros(K*3,N);
ola = zeros(1,N+12*L);
```

```
rx = [rx zeros(1, L*3);]

for i = 1:K*3
  cnt = (i-1)*L+1;
  xn(i+K,:) = [rx(cnt:(cnt+L-1)) zeros(1, N-L)];
  xf(i+K,:) = fft( xn(i+K,:) );
end

%Freq. Domain Convolution
for i = K+1:K*4
    for j = 1:K
        yf(i-K,:) = yf(i-K,:) + xf(i-j,:).*mf_fft_n(j,:);
    end
    yn(i-K,:) = ifft( yf(i-K,:) );
end


%Overlap Management
ola(1:L) = yn(1, 1:L);
ola(L+1:N) = yn(1, L+1:N);

for i = 1:K*3
  ola( i*L+1 : N+(i-1)*L ) = ola( i*L+1 : N+(i-1)*L ) + yn(i, 1:N-L);
  ola( N+(i-1)*L+1 : (i+1)*L ) = yn(i, N-L+1:L);
  ola( (i+1)*L+1 : N+i*L) = yn(i, L+1:N);
end

figure('Name', 'MATLAB Block Convolver Output')
plot(ola)


%--------------------------------%
%      Modultaion Function       %
%--------------------------------%

function mn = cdma_chirp_fsk(bitseq, spb, f_0, f_1, F, df)
    T = 1/F;
    nb = size(bitseq, 2);
    num = 1:spb;
    mn = zeros(1, spb*nb);
    mn_len = 1:size(mn,2);
    chirp = (df/spb).*(0:(spb-1));
    cnt0 = 0;
    cnt1 = 0;

    for i = 1:nb
```

```
        if (bitseq(i) == 1)
            fi = f_1( mod(cnt1, size(f_1, 2))+1 );
            fi = fi + chirp - df/2;
            bit = sin(2*pi*T*(fi.*num));
            cnt1 = cnt1 + 1;
        end
        if (bitseq(i) == 0)
            fz = f_0( mod(cnt0,  size(f_0, 2))+1 );
            fz = fz - chirp - df/2;
            bit = sin(2*pi*T*(fz.*num));
            cnt0 = cnt0 + 1;
        end
        mn((i-1)*spb+1:i*spb) = bit;
    end
end
```

Appendix F

False Alarms Tests

F.1. MATLAB Code

```
fs = 44100;
Ts = 1/fs;

Tb = 20e-3;
samps = Tb*fs;

delta_f = 1/(2*Tb);


f1 = delta_f*80;
f0 = delta_f*40;


fi = [f1-250, f1];
fz = [f0, f0+250];

dsamp = 2;
blocks = 3;
L = 2048;
N = L*2;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

fhss_codes12 = [
 [0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1,],
 [1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1,],
 [1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 0,],
 [0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1,],
 [0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 0,],
 [0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1,],
 [1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0,],
 [0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0,],
 [1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1,],
 [1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0,],
```

```
 [0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1,],
 [1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1,],
 [0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0,],
 ]

BITS = 12;
NUM = size(fhss_codes12, 1);

bit_begin = 1;
bit_end = bit_begin + BITS -1;
trgcnt_begin = bit_end + 1;
trgcnt_end = trgcnt_begin + NUM -1;
xcorrmax_begin = trgcnt_end + 1;
xcorrmax_end = xcorrmax_begin + NUM -1;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
sat = @(x, delta) min(max(x/delta, -1), 1);

B = zeros(NUM, xcorrmax_end);

B(:, bit_begin:bit_end) = fhss_codes12;

SNRdb = 3;
SNR = 10^(SNRdb/10);

thresh = ((BITS+5)/2);
thresh = 5;

noise_stddev = 1/SNR;
xcorr_thresh = thresh*(samps/(2*dsamp));

rv = cdma_chirp_fsk( B( 2, bit_begin : bit_end ), samps, fz, fi, fs,
   delta_f*4);
rv = downsample(rv, dsamp);
rv_polluted = sat(awgn(rv, SNRdb),1);
figure('Name', 'RX Message')
plot(rv_polluted)

for j = 1:NUM
    j
    template = cdma_chirp_fsk(B( j, bit_begin : bit_end ), samps/dsamp,
   dsamp*fz, dsamp*fi, fs, dsamp*delta_f*4 );

    for i = 1:NUM
        rx = cdma_chirp_fsk( B( i, bit_begin : bit_end ), samps, fz, fi,
    fs, delta_f*4);
```

```
        rx = downsample(rx, dsamp);

        for n = 1:100
            rx_polluted = awgn(rx, SNRdb);
            rx_polluted = sat(rx_polluted,1);
            rtr = xcorr( rx_polluted, template );

            if ( max(rtr) >= xcorr_thresh )
                B(j, trgcnt_begin + i-1) = B(j, trgcnt_begin + i-1) + 1;
            end

            if ( max(rtr) > B(j, xcorrmax_begin+ i-1) )
                B(j, xcorrmax_begin+ i-1) = max(rtr);
            end

        end
    end
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



function mn = cdma_chirp_fsk(bitseq, spb, f_0, f_1, F, df)
    T = 1/F;
    nb = size(bitseq, 2);
    num = 1:spb;
    mn = zeros(1, spb*nb);
    mn_len = 1:size(mn,2);
    chirp = (df/spb).*(0:(spb-1));
    cnt0 = 0;
    cnt1 = 0;

    for i = 1:nb
        if (bitseq(i) == 1)
            fi = f_1( mod(cnt1, size(f_1, 2))+1 );
            fi = fi + chirp - df/2;
            bit = sin(2*pi*T*(fi.*num));
            cnt1 = cnt1 + 1;
        end
        if (bitseq(i) == 0)
            fz = f_0( mod(cnt0,  size(f_0, 2))+1 );
            fz = fz - chirp - df/2;
            bit = sin(2*pi*T*(fz.*num));
            cnt0 = cnt0 + 1;
        end
        mn((i-1)*spb+1:i*spb) = bit;
    end
```

```
end
```

F.2. Results

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	896	0	0	0	0	0	0	0	0	0	0	0	0
2	0	742	0	0	0	0	0	0	0	0	0	0	0
3	0	0	523	0	0	0	0	0	0	0	0	0	0
4	0	0	0	757	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1129	0	0	0	0	0	0	0	0
6	0	0	0	0	0	523	0	0	0	0	0	0	0
7	0	0	0	0	0	0	720	0	0	0	0	0	0
8	0	0	0	0	0	0	0	898	0	0	0	0	0
9	0	0	0	0	0	0	0	0	739	0	0	0	0
10	0	0	0	0	0	0	0	0	0	696	0	0	0
11	0	0	0	0	0	0	0	0	0	0	558	0	0
12	0	0	0	0	0	0	0	0	0	0	0	370	0
13	0	0	0	0	0	0	0	0	0	0	0	0	715

Figure F.1: BFSK - High Attenuation

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	1161	0	0	0	0	0	0	0	0	0	0	0	0
2	0	892	0	0	0	0	0	0	0	0	0	0	0
3	0	0	761	0	0	0	0	0	0	0	0	0	0
4	0	0	0	903	0	0	0	0	0	0	0	0	0
5	0	0	0	0	1272	0	0	0	0	0	0	0	0
6	0	0	0	0	0	766	0	0	0	0	0	0	0
7	0	0	0	0	0	0	893	0	0	0	0	0	0
8	0	0	0	0	0	0	0	1153	0	0	0	0	0
9	0	0	0	0	0	0	0	0	871	0	0	0	0
10	0	0	0	0	0	0	0	0	0	878	0	0	0
11	0	0	0	0	0	0	0	0	0	0	814	0	0
12	0	0	0	0	0	0	0	0	0	0	0	521	0
13	0	0	0	0	0	0	0	0	0	0	0	0	873

Figure F.2: Optimized Modulation Scheme - High Attenuation

	1	2	3	4	5	6	7	8	9	10	11	12	13
1	2000	0	0	0	0	0	0	0	0	0	0	0	0
2	0	2000	0	0	0	0	0	0	0	0	0	0	0
3	0	0	2000	0	0	0	0	0	0	0	0	0	0
4	0	0	0	2000	0	0	0	0	0	0	0	0	0
5	0	0	0	0	2000	0	0	0	0	0	0	0	0
6	0	0	0	0	0	2000	0	0	0	0	0	0	0
7	0	0	0	0	0	0	2000	0	0	0	0	0	0
8	0	0	0	0	0	0	0	2000	0	0	0	0	0
9	0	0	0	0	0	0	0	0	2000	0	0	0	0
10	0	0	0	0	0	0	0	0	0	2000	0	0	0
11	0	0	0	0	0	0	0	0	0	0	2000	0	0
12	0	0	0	0	0	0	0	0	0	0	0	2000	0
13	0	0	0	0	0	0	0	0	0	0	0	0	2000

Figure F.3: Optimized Modulation Scheme - Strong Reflection