DEVELOPMENT OF A VERSATILE GROUND STATION FOR SATELLITE COMMUNICATIONS

By

James Miller, B.S.

A Project Submitted in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in

Electrical Engineering

University of Alaska Fairbanks

August 2023

APPROVED:

Denise Thorsen, Committee Chair
Michael Hatfield, Committee Member
Dejan Raskovic, Committee Member

Abstract:

The purpose of this project is to create a communication link from a Student Ground Station at the NOAA facility in Fox to the Cubesat Communication Platform (CCP) Cubesat . The Student Ground Station will send Commands (Uplink) and receive Telemetry (Downlink). The radio onboard of the CCP Cubesat is the GOMSpace AX100, which is a Commercial off the shelf system. The GOMSpace AX100 uses Cubesat Space Protocol and AX.25 standards for their packet structure. The GOMspace AX100 uses Frequency Shift Keying (FSK) for modulation.

My project will use a Software Defined Radio (SDR) that was coded in Matlab to communicate with the GOMSpace AX100. For Uplink, I will receive bits from COSMOS and conform to Cubesat Space Protocol and AX.25 standards. After I have Cubesat Space Protocol and AX.25 encoded bits, I perform FSK modulation on the bits and send the voltages to the SDR. In Downlink, I will receive samples from the SDR and perform FSK demodulation. After I have bits from demodulation, I will perform AX.25 decoding, then Cubesat Space Protocol decoding. The payload is then extracted from the decoded bits, then the payload data is sent to COSMOS for interpretation. I was able to successfully send commands to the GOMSpace AX100 and I was able to decode Telemetry packets sent by the AX100, which means I was are able to establish a communication link to the AX100.

CCP-GSR-DSN-UHF_SOFTWARE-R02

Native File: N/A

Cubesat Communications Protocol

Ground Station (GS) UHF Transceiver Software Design

James Miller Subsystem Lead	James Miller Signature of Approval	3/29/23 Date
Dr. Denise Thorsen Director	Signature of Approval	Date

Alaska Space Grant Program
University of Alaska Fairbanks
Duckering Hall 207
306 Tanana Loop Fairbanks, Alaska 99775

Revision History

Rev Level	Description	Author	Date	Approval
1	First rough draft of the Design Document.	JM	3/9/23	
2	Added more sections and fixed errors	JM	6/16/23	
3				
4				

TBD/TBR

Item No.	Page	Summary	Author	Due Date

Table of Contents

Contents

1. Introduction

1.1 Purpose

The purpose of the UHF transceiver is to establish a communication link between the Student Ground Station (SGS) and the Cubesat Communications Platform (CCP) nanosatellite after it has been launched. We will use a Software Defined Radio (SDR) that will be able to communicate to the radio onboard the CCP nanosatellite.

1.2 Scope

Included in this document is the list of Ground Segment (GS) UHF transceiver requirements, their relationship to the CCP mission requirements, and preliminary design choices made to meet those requirements.

1.3 Document Organization

Section 1 Specifies purpose and content of this document.

Section 2 Lists reference documents.

Section 3 Provides an overview of subsystem purpose with respect to satisfying specific mission requirements.

Section 4 Specifies the packet structure that will be used, the type of encoding that will be used on the packet and modulation that will be used on the bits.

Section 5 Specifies the process for demodulating the received signal and decoding the received bits.

Section 6 Details the testing procedure and testing results between the Ground Stations UHF transceiver software and the COMM subsystem.

Section 7 List of the web resources and acronyms used for writing the Design Document.

2. Reference Documents

The following documents listed here were used as a reference for this document. Please refer to them for detailed information not included herein:

2.1 Internal Documents

1	File Name	Document Description
2	CCP-SYS-REQ-Requirements_Verification_Matrix-R08.xlsx	List of system requires for each subsystem

2.2 Industrial Documents

3		

2.3 Government Documents

3		

3. Overview

The UHF Transceiver needs to match the same protocols as the CCP COMM subsystem for Command and Telemetry. The CCP COMM subsystem transmits binary FSK using Ham standard AX.25 protocol.

Below is described the high-level Ground Segment Requirements that the UHF Transceiver will satisfy. The Ground Segment (GS) will operate in the Amateur satellite frequency band (GSR-8.1). The Amateur-Satellite band that will be used will be UHF, which ranges from 435-438 MHz.

3.1 Ground Segment Requirements

GSR-8.1 The Ground Segment shall be capable of downlink/uplink at the 437.25 MHz communication frequency.

GSR-8.2 The Ground Segment shall provide downlink/uplink capability of at least 9600 bits per second.

GSR-8.3 Ground Segment shall send an RF Off Command to the spacecraft upon request.

3.2 Concept of Operation

Figure 1 shows the general process for uplink and downlink from CCP to the GS. The GS is set up for continuous downlink and only switches to uplink mode when the satellite is in sight of the GS and when a command needs to be sent.

In downlink mode, the *UHF_Transceiver.m* script will be in receive mode, waiting to receive telemetry packets sent by the COMM subsystem [5] aboard the CCP cube satellite. The In-phase and Quadrature (I/Q) voltages, sampled by a Universal Software Radio Peripheral (USRP) are sent to a Matlab script, *UHF_Transceiver_R1.m*, for processing. *Fsk_demod.m* function demodulates I/Q samples into bits. After which, *ax25_decode.m* performs AX.25 decoding. Then, the *csp_decode.m* function decodes the AX.25 payload using the Cubesat Space Protocol (CSP). After the CSP packet is decoded, it is sent to COSMOS [6], which will handle all of the Command and Telemetry data.

In uplink mode, COSMOS sends Command packets to the *UHF_Transceiver_R1.m* script, which will perform CSP encoding using the *csp_encode.m* function and then AX.25 encoding using the *ax25_encode.m* function. Then, the *fsk_mod.m* function will perform FSK modulation, turning the encoded bits into an array of I/Q samples, which is then sent to the B210. The B210 will take the array of I/Q samples, perform Digital to Analog Conversion (DAC) and then mix the analog signal with the carrier frequency.

UHF Transceiver Software Structure

Uplink **Command**	Downlink **Receiver**
1. Receive a Command packet from COSMOS	1. Receive Samples from the B210
2. CSP Encode	2. FSK demodulation
3. AX.25 Encode	3. AX.25 Decode
4. FSK modulation	4. CSP Decode
5. Send samples to the B210	5. Send Telemetry packet to COSMOS

Figure 1: General process for Command and Telemetry.

3.3 System Components

The *UHF_Transceiver_R1.m* will be using the Ettus USRP B210, which can transmit and receiving at the carrier frequency of 437.25 MHz (GSR 8.1). The B210 is also able to meet the required 9600 baud rate (GSR 8.2). The B210 will be attached to a computer through a USB cable, then *UHF_Transceiver_R1.m* script will handle all of the signal processing.

3.4 Items of Concern

This system needs to be tested at low power to check the demodulation parameters and Coarse Frequency Compensation algorithm. The Received Signal Strength Indicator (RSSI) threshold needs to be set low enough to detect the signal and not trigger due to ambient noise. The Transmitter and receiver scripts are in the process of being combined and tested.

4. Command Uplink

4.1 B210 Transmitter setup

The first step is to set up the parameters for the B210, as shown in Figure 2.

```
radioTx = SDRuTransmitter('Platform','B210','SerialNum','321E863');
radioTx.CenterFrequency = 437.25e6;   %437.25 MHz carrier frequency
radioTx.Gain = 40;   %Amount of gain applied before going out the Tx port
radioTx.MasterClockRate = 9.6e6;    %9.6 MHz Master clock rate
radioTx.InterpolationFactor = 200;  %sets the baseband sample rate to 9.6 MHz / 200 = 48 kHz
radioTx.EnableBurstMode = 1;    %Enables burst mode for transmitter
radioTx.NumFramesInBurst = 3;   %Number of frames to be sent in a burst
```

Figure 2: *UHF_Transceiver.m script* setup for the B210.

The first line creates the object **radioTX** using the Matlab function *comm.SDRuTransmitter()* which defines the platform as a B2xx USRP with a specific serial number and model number. The serial number and model number can be found by typing **findsdru** into the Matlab command prompt window.

The **radioTx.CenterFrequency** parameter is to set the carrier frequency at 437.25 MHz.

The **radioTx.Gain** parameter is the gain applied to the signal before reaching the TX port of the USRP.

The **radioTx.MasterClockRate** parameter is for establishing the base clock of the B210. The master clock rate parameter was chosen to be a multiple of the bit rate, 9600, for matched filtering. The master clock rate was chosen to be 9.6 MHz and an interpolation factor of 200. The baseband sample rate is derived from taking the master clock rate and dividing by the interpolation factor, which is shown in equation 1. Dividing the master clock rate of 9.6 MHz by an interpolation factor of 200 yields a baseband sample rate of 48 kHz, which equates to 5 samples per symbol.

$$\begin{aligned} Baseband\ Sample\ Rate &= \frac{Master\ Clock\ Rate}{Interpolation\ Factor} \\ Baseband\ Sample\ Rate &= \frac{9.6\ MHz}{200} \\ Baseband\ Sample\ Rate &= 48\ kHz \end{aligned} \tag{1}$$

The **radioTx.EnableBurstMode** parameter enables burst mode. Burst mode allows contiguous samples without overruns or underruns between *UHF_Transceiver.m* script and the B210, which means it will not underflow the TX buffers on the USRP. Burst mode had to be enabled because samples were being dropped when I started to transmit pulses. Burst mode stopped the samples from being dropped and the transmitter can send pulse transmissions.

The **radio.Tx.NumFramesInBurst** parameter is the number of frames the *UHF_Transceiver_R1.m* script will collect before the B210 transmits.

Frames are just an array of samples. The *UHF_Transceiver_R1.m* will collect samples for a given period, then output those samples into a frame. The total time of a single frame is shown in equation 2, where T_{Frame} is the total time of the frame, and T_s is the baseband sample rate.

$$T_{Frame} = Number\ of\ samples * T_s \tag{2}$$

The number of samples per frame is not a parameter for the transmitter. But, once you send a frame, the B210 will expect subsequent frames to be the same size. Because AX.25 is a variable sized format, it does not mesh very well with the expectation of a constant frame size. To send frames of different sizes, you would need to rerun the entire script, which can take up to a minute to rerun. A minute to rerun the script during a pass would be unacceptable, because telemetry is expected to occur every 10 seconds.

To get around the problem of variable sized frames from AX.25, I setup a constant frame size, based from the max number of bits in a packet. By always sending a constant frame size, you do not have to rerun the script to send packets of different frame sizes. I append 'dummy samples' so that the total will match the constant frame size. Figure 3 shows how I stuff samples until reaching a fixed packet size. The Packet (samples) portion is the actual signal of interest. The dummy values are random I/Q samples that are essentially noise. By appending dummy samples, the script does not have to be rerun each time the packet changes size.

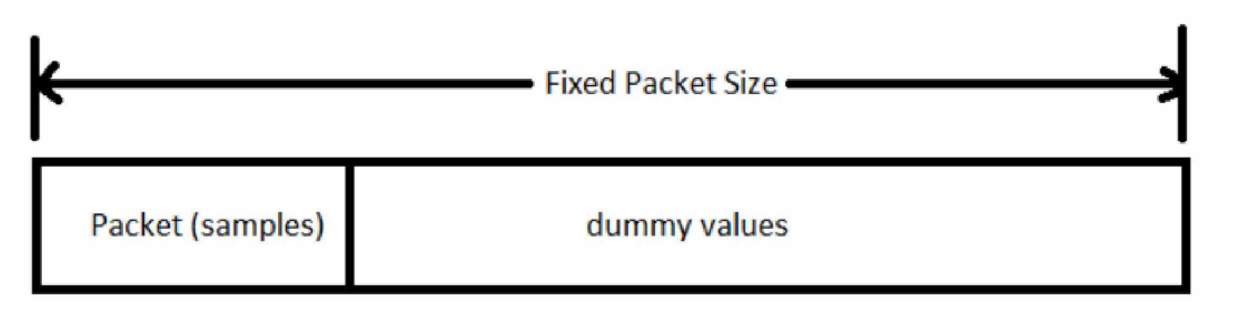

Figure 3: Sample stuffing to reach a fixed packet size.

4.2 Packet Structure

Before encoding can take place, it is important to note the structure of packets. The payload is encoded using CSP, with the *csp_encode.m* function and then encoded using AX.25 using the *ax25_encode.m* function. Figure 4 shows the complete packet structure. A CSP packet is the inner structure, which will be the payload inside of the AX.25 packet. The reason why CSP is used is because COMM uses CSP in all of their packet structures. Figure 4 shows the general structure of the packet that will be used.

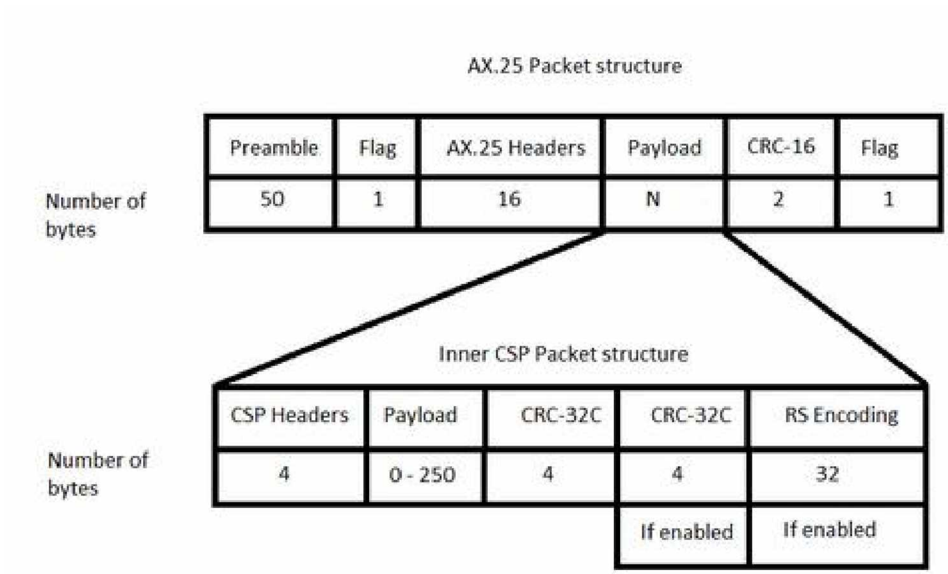

Figure 4: General packet structure of the transceiver.

CSP can either have 1 or 2 CRC-32s attached. Reed Solomon (RS) encoding can be enabled on the AX100. If enabled, the RS encoding takes 32 bytes and can correct up to 16 bytes.

4.3 CSP packet structure

The CSP packet uses Most Significant Bit (MSBit) first for packets. Figure 5 shows the CSP header information, which is a CSP Header 1.x style [3]. There are a total of 4 bytes of headers.

CSP Header 1.x

Bit offset	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	Priority		Source					Destination					Destination Port						Source Port						Reserved				HMAC	XTEA	RDP	CRC
32	Data (0 – 65,535 bytes)																															

Figure 5: CSP header information [3].

Header Information:

- Priority – 0 - Critical
  1 - High
  2 - Normal

3 - Low

- Source – 5 bit number
- Destination - 5 bit number
- Destination Port – 6 bit number
- Source Port – 6 bit number
- HMAC - Hash-Based Message Authentication Code, it uses authentication keys for secure transmission and reception.
- XTEA – encryption (can be used for uplinking for amateur radio users)
- RDP – enacts RDP headers (never used?)
- CRC – Adds an additional CRC32-C to the packet
- Payload – The payload can range from 0 to 250 bytes, MSbit first.

The algorithm that is used for the CRC32 calculation is the CRC32-C, which uses the Castignoli polynomial 0x82F63B78. The bits are always fed MSBit first into the *crc.m* function. The CSP has two different options for CRC32(s). If the CSP CRC header bit is zero, there will only be one CRC32 and the headers and payload are input into the *crc.m* function. If the CSP CRC header bit is high, there will be two CRC32s. The first CRC32 is calculated with only the payload. The second CRC uses the headers, payload and inner CRC. Figure 6 shows how the CRC32(s) are calculated, depending on the CSP header bit. The arrows signify what part of the packet is input into the *crc.m* function for each CRC32.

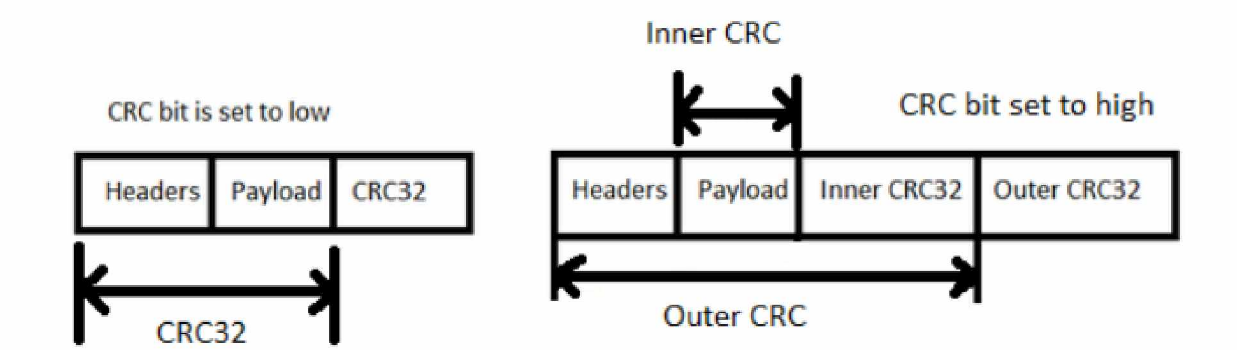

Figure 6: CRC32 calculations, depending on CSP CRC bit in the headers.

4.4 CSP encoding

Figure 7 shows the encoding process for the CSP packet. The payload has 4 bytes of headers attached. The CRC(s) is calculated, and appended. Then, the packet is CCSDS scrambled using the *CCSDS_Scramble.m* function, where each byte is input MSBit first. After the packet is scrambled, each byte is swapped from MSBit first to LSbit first and then sent to be AX.25 encoding.

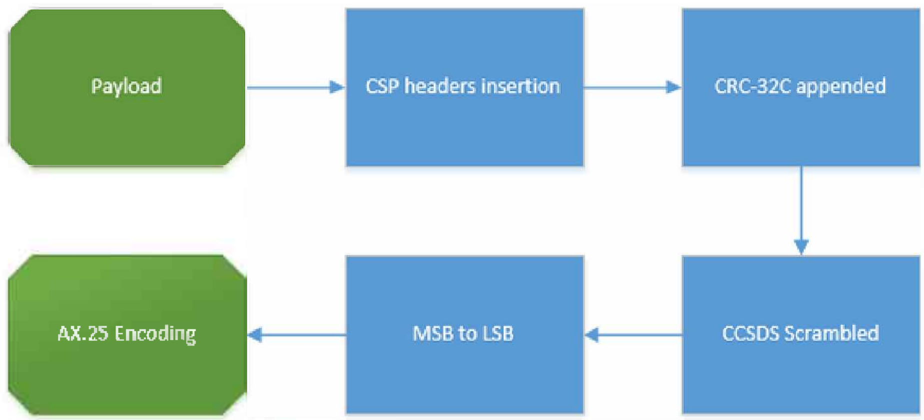

Figure 7: CSP encoding process for the *csp_encode.m* function.

4.4.1 CCSDS Scrambler

The scrambling process is shown in Figure 8 [4]. The scrambler polynomial that is used is $x^8 + x^7 + x^5 + x^3 + 1$. The shift register is initialized to all ones. The *CCSDS_Scramble.m* function is used to scramble the CSP packet.

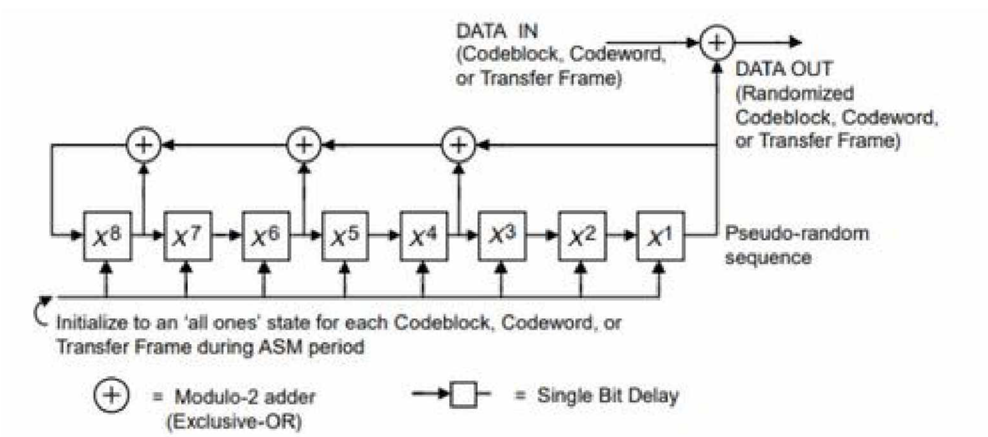

Figure 8: CCSDS scrambling process for the *CCSDS_Scrambler.m* function.

4.4.2 MSB to LSB

The reason why it swaps from MSbit to LSbit is because the AX.25 protocol demands each byte be in the LSbit format. After each byte is in LSbit format, it is then AX.25 encoded, which is continued in section 5.2.3.

4.5 AX.25 Packet structure

The AX100 uses an Unnumbered Frame packet structure or simply, U frame. U frames typically do not have a PID field, but there are exceptions for U frames to include a PID field. Figure 9 shows the U frame structure [1].

	Flag	Adress	Control	PID	Payload	FCS	Flag
Number of Bytes	1	14	1	1	N	2	1

Figure 9: U frame packet structure for AX.25.

4.5.1 Flag

The purpose of the flag is to mark the beginning and the end of a packet. The flag is denoted as 0x7E or 01111110 in binary.

4.5.2 Address Field

There are 14 total bytes in the Address field. The first 7 bytes are for the destination and the 2nd set of 7 bytes is for the source. Figure 10 shows how the address bytes are arranged. The SSID is a single byte at the end of the destination and source address fields. The purpose of the SSID field is to allow the radio station operator to operate more than one station. The Destination SSID is set to 0x60 and the Source SSID is set to 0x61.

Destination		Source	
Destination Callsign	SSID	Source Callsign	SSID

Figure 10: Address structure for the AX.25 headers.

A callsign is a six-character sequence of alphanumeric characters in ASCII. The alphabet characters are all caps. If the callsign is not six characters long, it will be appended with the space character. An example of a callsign is "KL3JP ", note the space after the letter P. The SSID bytes clarify command and response frames. All of the bytes in the address field (except the SSID) are bit shifted, meaning you are essentially multiplying the hex value of the byte by 2.

4.5.3 Control Field

The control field specifies the type of packet that is being sent, like the send or receive sequence number and sending, receiving or Acknowledging State Variables [1]. The Control field byte is set to 0x03.

4.5.4 PID Field

Protocol Identifier (PID) field identifies the type of protocol used [1]. It is typically only used for I and UI type of frames, but there can be exceptions for U frames. The PID field is set to 0xF0.

4.5.5 Payload

The payload would be an N number of bytes, max number of bytes for our system would be 250 bytes excluding overheard like CRC(s) and headers.

4.5.6 Frame Check Sequence (FCS)

The FCS is 2 bytes long and uses the CRC16 CCITT polynomial, $x^{16} + x^{12} + x^5 + 1$. The headers and payload are fed into the CRC16 algorithm, LSBit first. After the FCS has been calculated, the bytes are changed from LSbit to MSbit. These two bytes are the only two bytes that are MSbit first in the entire packet.

4.6 AX.25 Encoding

Figure 11 shows the encoding process for the *ax25_encode.m* function. The headers are attached to the packet and then the CRC-16 is appended to the end. The packet is then bit stuffed, flags are appended at the start and end of a packet and the preamble is attached to the start. The packet is then Non Return to Zero Inverted (NRZI) encoded, after that, the packet is then scrambled using the G3RUH polynomial.

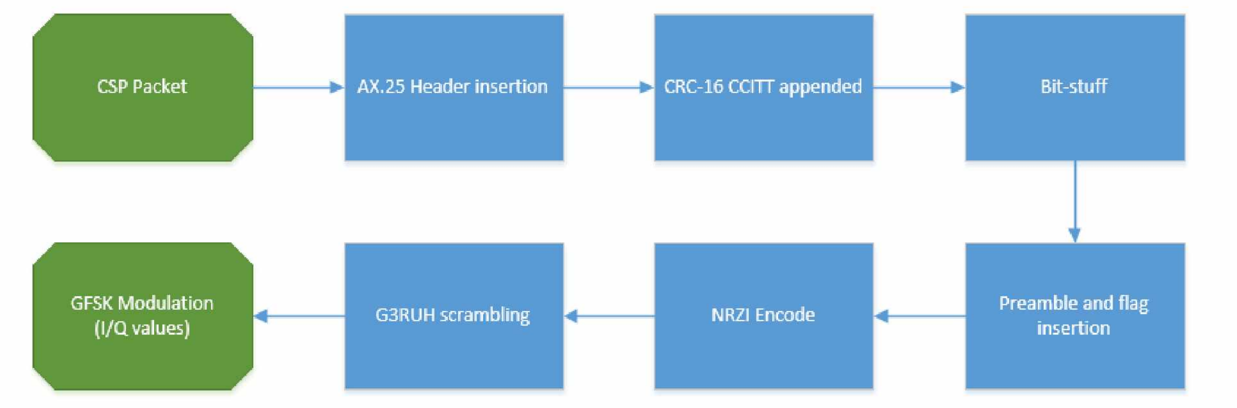

Figure 11: Encoding process for *ax25_encode.m* function.

4.6.1 Bit stuffing

Bit stuffing inserts a zero after five consecutive ones, so that the flag cannot appear anywhere in the headers, payload or CRC, but only at the start and end of a packet. Figure 12 shows the bit stuffing process. The bit stuffed zero is highlighted in red.

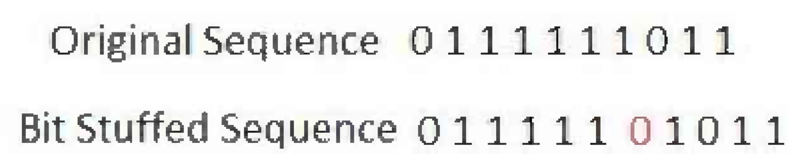

Figure 12: Bit stuffing process.

4.6.2 Preamble and flag Insertion

The preamble byte is the same as the flag byte, 0x7E. So, 51 bytes of the flag are attached at the start of the AX.25 packet and 1 flag byte at the end at the end of the packet.

4.6.3 NRZI Encoding

The purpose of Non-Return to Zero Inverted (NRZI) encoding is to force transitions when a zero appears. Figure 13 illustrates the NRZI encoding process, which can be shown as a Finite State Machine (FSM).

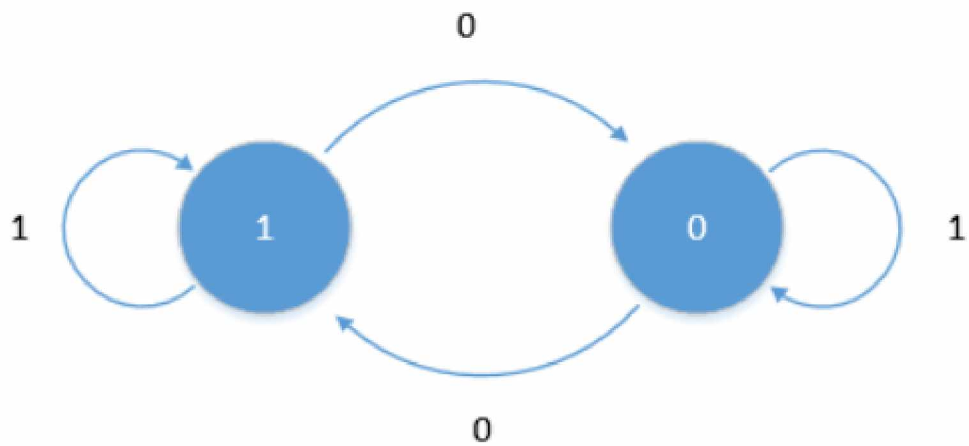

Figure 13: NRZI encoding process for the *ax25_encode.m* function.

The blue circles represent the output, and the lines represent the input. The initial state is set to zero. If the input is a 0, the output will invert. If the input is a 1, the output will keep its current state. Figure 14 shows an example of a NRZI encoded bit sequence.

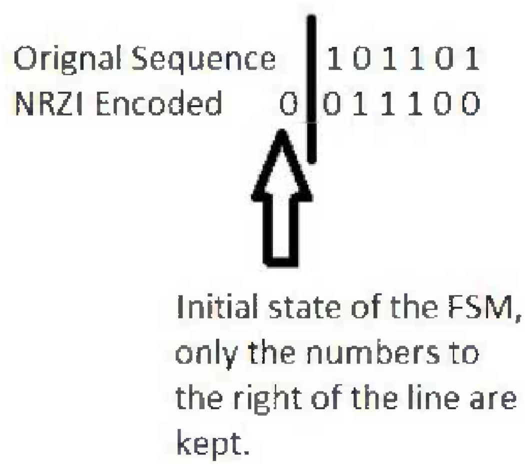

Figure 14: Example of NRZI encoding of the *ax25_encode.m* function.

4.6.4 Scrambler

The scrambler polynomial that is used is the G3RUH or $x^{17} + x^{12} + 1$. Figure 15 shows the process for scrambling. The shift register is initialized to all zeros. The 17^{th} and 12^{th} spot in the register are XORd with each other and the result is then XORd with the input. The output is then fed into the shift register.

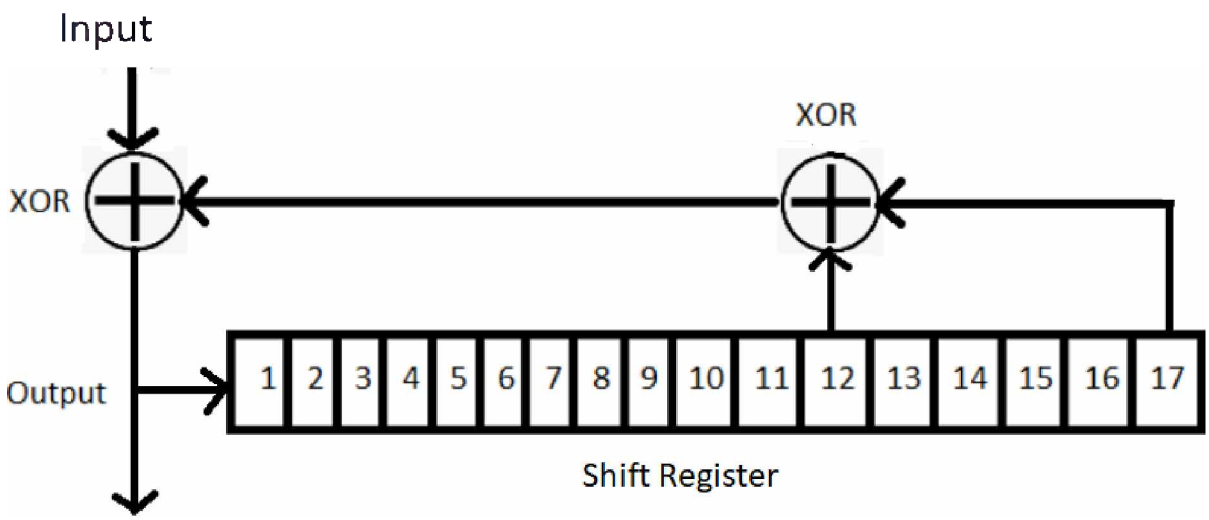

Figure 15: Scrambler process for the *ax25_encode.m* function.

4.7 FSK modulation

Frequency Shift Keying (FSK) is modulation or turning bits into I/Q samples. In binary FSK, a digital one is represented using a sinusoid called the mark frequency. A binary 0 in FSK format is called the space frequency. The signal will switch between the mark and space frequencies to represent the digital signal switching between ones and zeros.

FSK signals are defined by their carrier frequencies and symbol rate. The transmitter will multiply a baseband signal (FSK) with the carrier frequency. The other way FSK is defined is by the symbol rate. The symbol rate for non-coherent (non-coherent means you do not know the phase of the carrier) is the frequency separation between the mark and space frequencies. The separation between for non-coherent signals will be the symbol rate, which will be 9600 Hz. The reason why the separation is 9600 Hz is so that both frequencies will be orthogonal to each other. Figure 16 shows the baseband signals frequency spectrum for FSK. Figure 17 shows the frequency spectrum for a carrier wave modulated FSK signal. The difference between Figure 16 and Figure 17 is that the baseband signal in Figure 17 is just translated up to the carrier waves frequency.

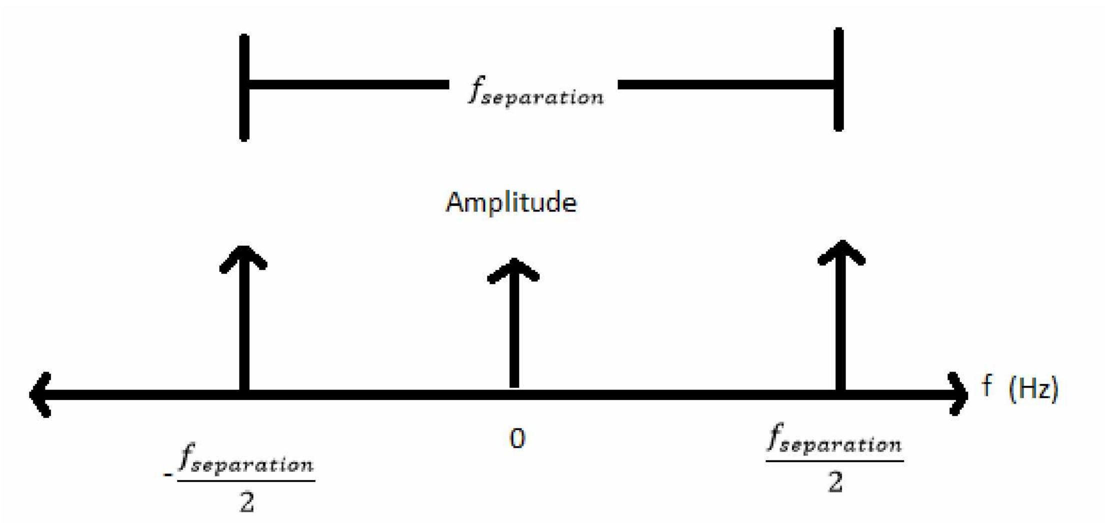

Figure 16: Frequency spectrum for a baseband FSK signal.

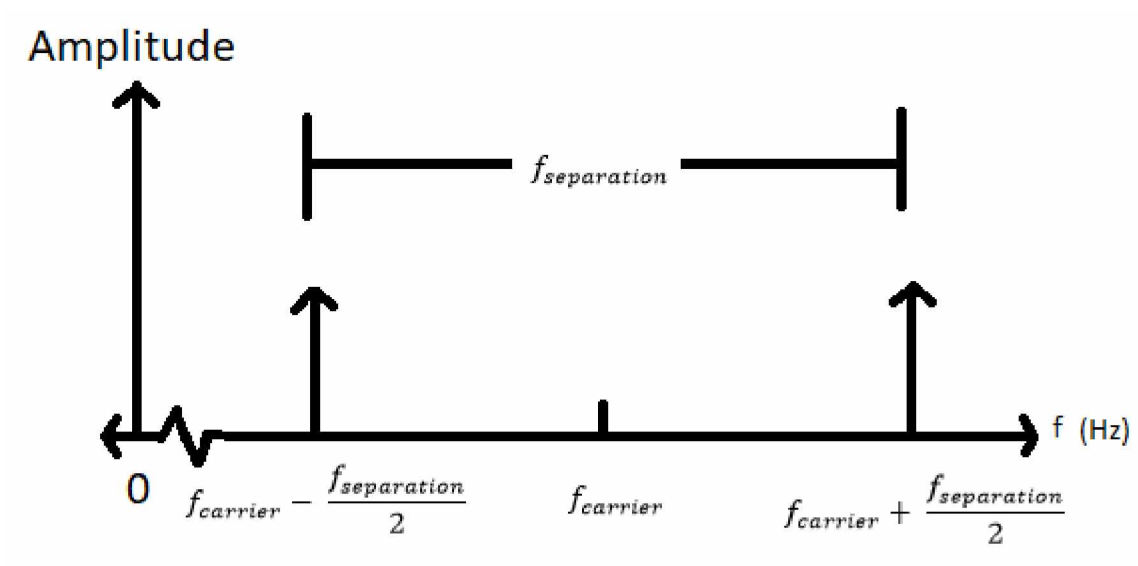

Figure 17: Frequency spectrum for a carrier wave modulated FSK signal.

The transceiver will use a form of FSK called Continuous Phase FSK (CPFSK). CPFSK means there will be no phase discontinuities during transitions from the mark to space or space to mark frequencies. The modulation index is set to one, which means a symbol will go from state A to state B or state B to state A, which shown in Figure 18. The Mark frequency will have a positive rotation across the complex unit circle. The Mark symbol will go from state A to state B or state

B to state A in a positive rotation. The Space frequency will have a negative rotation across the complex unit circle. The Space symbol will go from state A to state B or from state B to state A in a negative rotation across the complex unit circle.

Your initial bit will start at state A. The second bit will start at state B. To summarize, even numbered bits state at state A and go to state B. Odd numbered bits will start at state B and rotate until reaching state A.

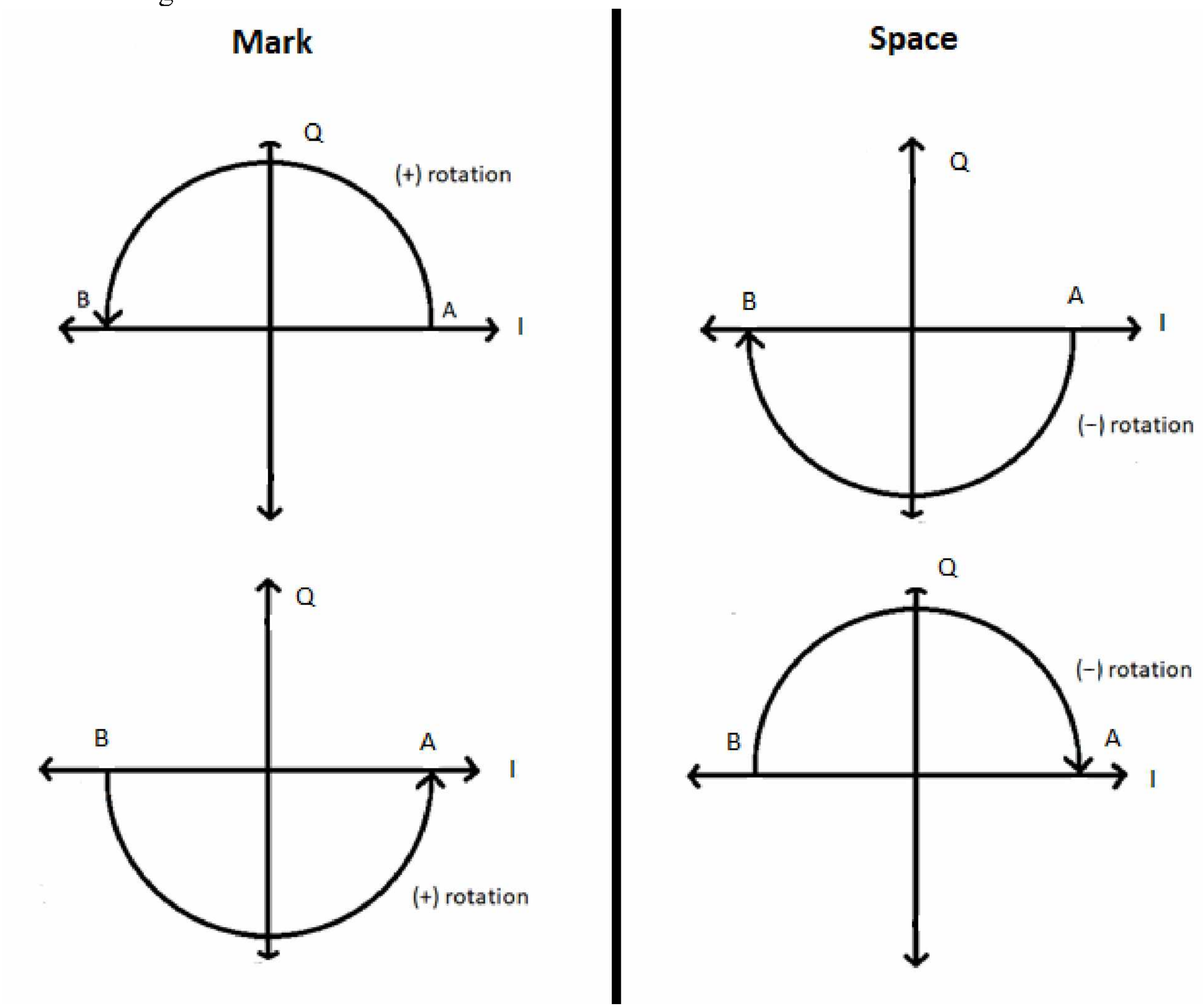

Figure 18: In-phase and Quadrature representation for Mark and Space symbols.

4.7.1 Sampling

To be able to properly capture the signal on the receiver, a suitable sample rate has to be chosen. A starting point would be the Nyquist criterion, shown below in equation 3.

$$Nyquist\ Criterion = 2 * frequency\ of\ Interest \tag{3}$$

If there were no frequency offsets, the sample rate would be 2 * symbol rate, which would be 19.6 kHz. But there are two sources of frequency offsets, which makes a 19.6 kHz sample rate impractical. To choose a suitable sample rate, I would have to observe the causes of frequency offsets and how they will affect the receiver. The reason for the greatest cause of frequency offsets is Doppler shifts. The other source of frequency offsets is due to crystal impurities. The impurities of the crystal will create an offset when generating the carrier frequency. Calculating the maximum frequency offsets due to Doppler shifts and crystal impurities will influence the decision of choosing a sampling frequency.

Doppler shifts are frequency shifts caused in the observed carrier frequency from the ground stations point of view. When the satellite is approaching the ground station, the carrier frequency will appear to be much faster than what is being transmitted from the satellite. The Doppler shift will go to zero when it has reached a maximum elevation, or when the satellite is closest to the ground station. As the satellite recedes (moving away from the GS), the carrier frequency appear much slower than the transmitted from the satellite.

For calculating the Doppler shift, some assumptions were made about the orbital speed and the carrier frequency. The orbital speed of the LEO satellite was set to 7500 m/s. The carrier frequency was set to 437 MHz (tentative). To calculate the Doppler shift, equation 4 was employed [2]. The Doppler shift will be rounded down to 11 kHz for simplicity

$$Doppler\ Shift = v_r \times \frac{f_{carrier}}{c}$$

$$v_r - \ velocity\ of\ the\ satellite$$
$$f_{carrer} - frequency\ of\ the\ carrier$$
$$c - Speed\ of\ light\ (vacuum)$$

$$Doppler\ shift = 7500\ \times \frac{437_E6}{3_E8}$$
$$Doppler\ shift \approx 11.3\ kHz \tag{4}$$

According to the ETTUS website, the B210's frequency accuracy is $\pm 2\ ppm$, meaning for every MHz, the frequency can be off between plus or minus 2 Hz [4]. Applying 437 MHz to the $\pm 2\ Hz\ per\ ppm,$ the frequency offset due the crystal impurities can be $\pm$ roughly 880 Hz. The frequency offset due to the crystal will be rounded up to 1 kHz for simplicity. I am unaware of the frequency accuracy of the radio for the COMM subsystem, I will be assuming it will be ± 3 kHz.

I will be observing two extreme cases for the carrier frequency. The first extreme case will be when the Doppler and crystal offsets add to the carrier frequency. The second case will be when the Doppler and crystal offsets both subtract from the carrier frequency. Figure 20 shows the effect of both constructive offsets on the mark and space frequencies. Figure 21 shows the effect of both offsets adding destructively on the mark and space frequencies. The frequency separation of the radios is assumed to be 4 kHz, when added with the +11 kHz Doppler shift, which will cause the carrier frequency to be shifted by 15 kHz. The 15 kHz carrier will cause the space frequency to appear at 10 kHz and the mark frequency to appear at 20 kHz.

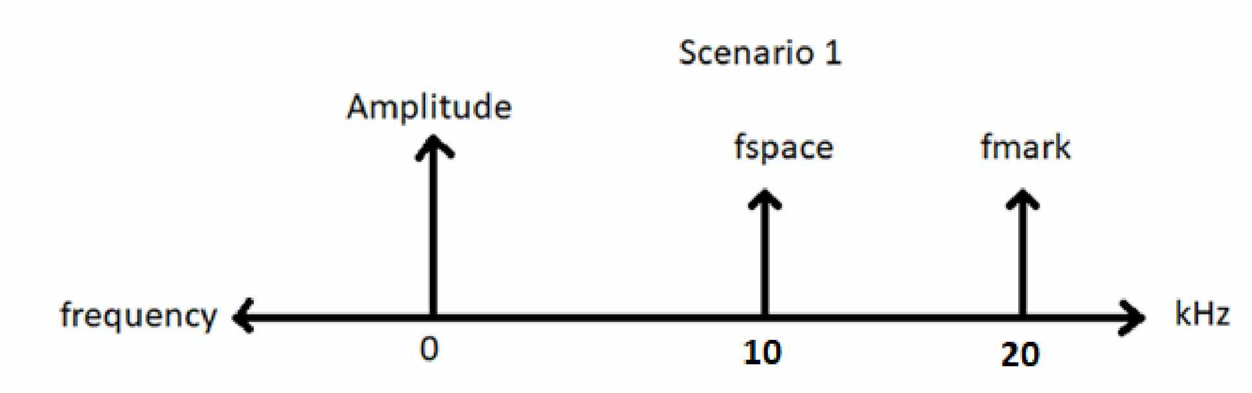

Figure 19: The Doppler and crystal frequency offsets adding constructively.

For scenario 2, the carrier frequency will be observed at -15 kHz, placing the mark frequency at -10 kHz and the space frequency at -20 kHz.

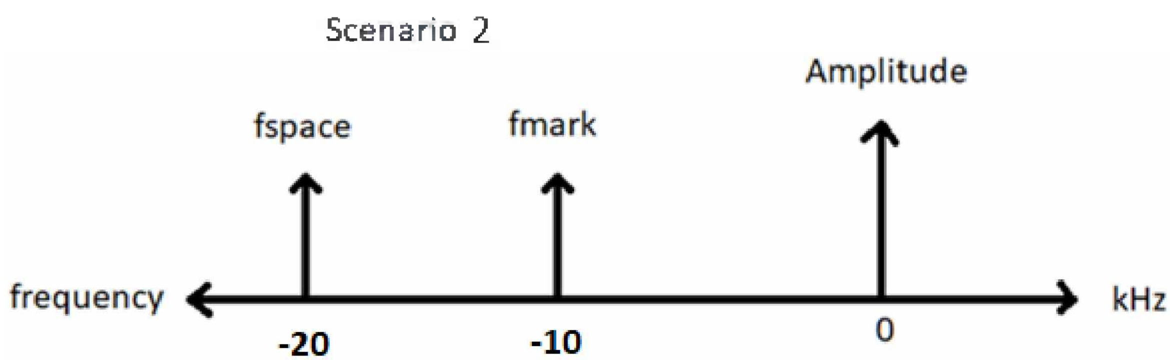

Figure 20: The Doppler and crystal frequency offsets adding destructively.

After observing both worst case scenarios, I need to be able to capture the mark frequency in scenario 1 and the space frequency in Scenario 2. Figure 21 shows the required bandwidth to cover both worst-case scenarios.

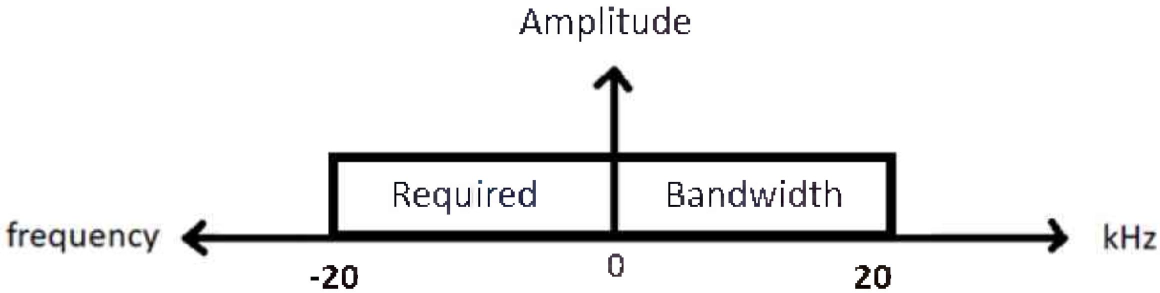

Figure 21: Required bandwidth for both worst case scenarios.

The minimum samples per symbol to be able to meet the required bandwidth in Figure 21 is 5, or 48 kHz. A 48 kHz sample rate leaves roughly a 4 kHz guard bandwidth before aliasing occurs during the extreme cases. Using 5 samples per symbol will ensure I am able to capture the signal, even with frequency offsets due to Doppler shifts and crystal impurities.

5. <u>Downlink</u>

5.1 USRP Receiver Setup

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%Receiver setup B210
radio = comm.SDRuReceiver('Platform','B210','SerialNum','321E863');
%%%-3016 is the offset from the b205 mini xmitter, 850 is the offset
%%%The offset needs to be determined from comm and the B210
radio.CenterFrequency = 437.25e6;   %center frequency at 437.25 Mhz with a 850 frequency offset
radio.Gain = 20;   %sets up the radio gain
radio.MasterClockRate = clockRate;   %Master clock rate at 9.6 Mhz
radio.DecimationFactor = 200;  %Sets the baseband sample rate to 9.6 MHz / 200 = 48 kHz
radio.ChannelMapping = 1;  %Uses the first channel
radio.SamplesPerFrame = samplesPerFrame;   %assumes 2500 bit size per packet
radio.OutputDataType = 'double';
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Figure 22: B210 receiver setup in the *UHF_Transceiver.m* script.

Figure 22 shows how to setup the B210 in the *UHF_Transceiver.m* script. The main parameters of interest are the type and ID number of the USRP, center frequency, gain, master clock rate, decimation factor, channel mapping, samples per frame and the data type of the output.

USRP setup

The first parameter is identifying the type of USRP and the identification number for the B210. The serial number for this particular B210 is 321E863.

Center frequency
The center frequency is simply the carrier frequency. The B210 will take the carrier frequency and perform carrier wave demodulation. After carrier wave demodulation, the B210 will output the signal at baseband. The carrier wave or center frequency will be at 437.25 MHz (tentative).

Gain
The gain will be amplified by 20 amount in dB. The gain will be increased to around 60-80 once low power testing starts.

Master clock rate
This parameter is for setting up the baseband sample rate. The baseband sample rate is calculated by taking the master clock rate and dividing it by the decimation factor, which is shown in equation 5. I chose the master clock rate to be a multiple of the bit rate, which was set to 9.6 MHz.

$$Baseband\ Sample\ Rate = \frac{Master\ Clock\ Rate}{Decimation\ Factor}$$
$$Baseband\ Sample\ Rate = \frac{9.6\ MHz}{200}$$
$$Baseband\ Sample\ Rate = 48\ kHz \tag{5}$$

Decimation Factor
The decimation factor is used for the baseband sample rate. The master clock rate is divided by the decimation factor to get the baseband sample rate. The decimation factor was chosen to be 200, so the baseband sample rate would be 48 kHz, which equates to 5 samples per symbol.

Channel mapping
The B210 will use Rx port 1.

Samples per frame
This parameter controls the number of samples that are collected in a single time period. The number that was calculated by multiplying the max number of bits in a frame and multiplying by the samples per symbol. Equation 6 shows how to calculate the Samples per frame parameter.

$$Samples\ Per\ Frame = (\max number\ of\ bits) \times \frac{N\ samples}{symbol}$$
$$Samples\ Per\ Frame = 2500\ bits\ \times \frac{5\ samples}{bit}$$
$$Samples\ Per\ Frame = 12500\ samples \tag{6}$$

Output data type

The output data type will be double type, which is a 64 bit representation of the sample.

5.2 FSK Demodulation

Figure 24 shows the process for turning samples into bits using the *fsk_demod.m* function.

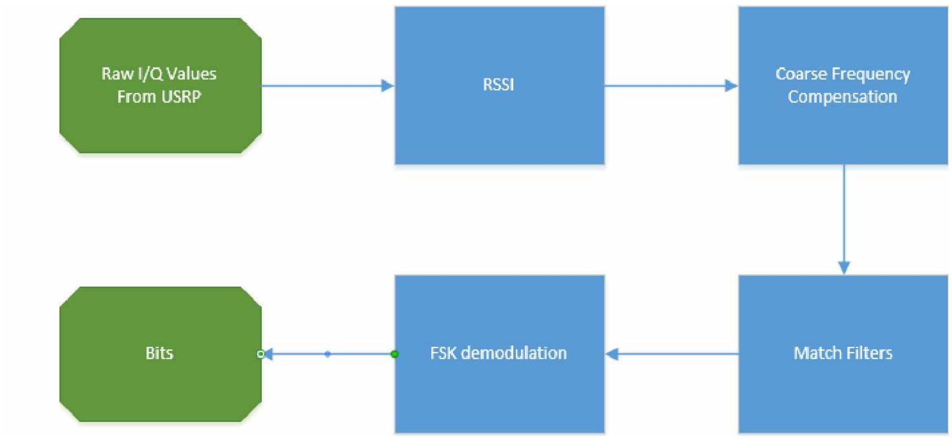

Figure 23: Overview for FSK demodulation process for the *fsk_demod.m* function.

5.2.1 Received Signal Strength Indicator (RSSI)

RSSI is for determining if a signal is present within a frame. To see if there is a signal, you need to convert I/Q samples into power by using equation 7. Equation 7 assumes a 50 Ω load. Each sample is converted to dBm and stored into an array. If the maximum power of power array is greater than -40 dBm (tentative), it will proceed into Coarse Frequency Compensation.

$$P_{dBm} = 10\log_{10}(10(I^2 + Q^2)) \tag{7}$$

5.2.2 Buffering

There are two scenarios, one where you do not buffer and the other one is where you need to buffer. For the first case, shown in Figure 25, you do not need to buffer because the signal is contained into a single frame, and you can continue on to Coarse Frequency Compensation. In the second case where you need to buffer, you save the current frame, and wait for the second frame to arrive, which is shown in Figure 26. Once you have both frames in the buffer, you can begin Coarse Frequency Compensation.

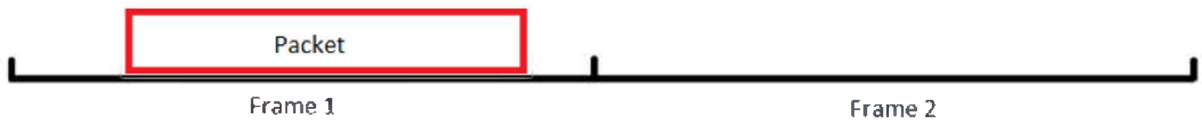

Figure 24: Scenario 1, packet is contained within a single frame.

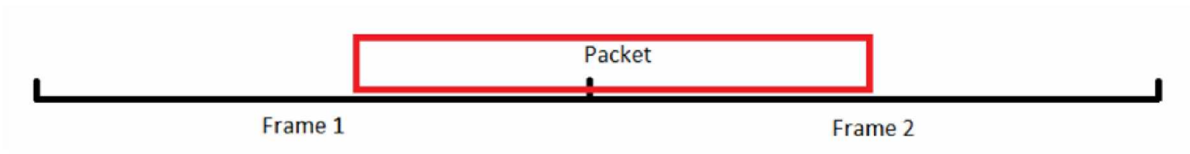

Figure 25: Scenario 2, a packet is overlapping between two frames.

So how do I determine if I need to buffer? I simply use the RSSI information and look at the last index in the frame. If the last sample within the frame's power is less than the power threshold, buffering will not occur. If the last sample's power is greater than the power threshold, it most likely means the packet is spilling over into the next frame and buffering will occur. The initial frame is saved into the buffer, after the second frame has arrived, it is saved to the buffer and continues onto Coarse Frequency Compensation.

5.2.3 Coarse Frequency Compensation

Coarse Frequency Compensation is adjusting a demodulated carrier wave signal so that it centered at 0 Hz. Having the signal centered at 0 Hz is critical for digital communications.

Figure 27 shows my process for Coarse Frequency Compensation. After the signal is present through RSSI, it performs a FFT of the signal. The *fsk_demod.m* function finds the two maximum peaks and estimates the average frequency between said maximum peaks. The average frequency is used to create a complex sinusoid and multiplied with the original signal.

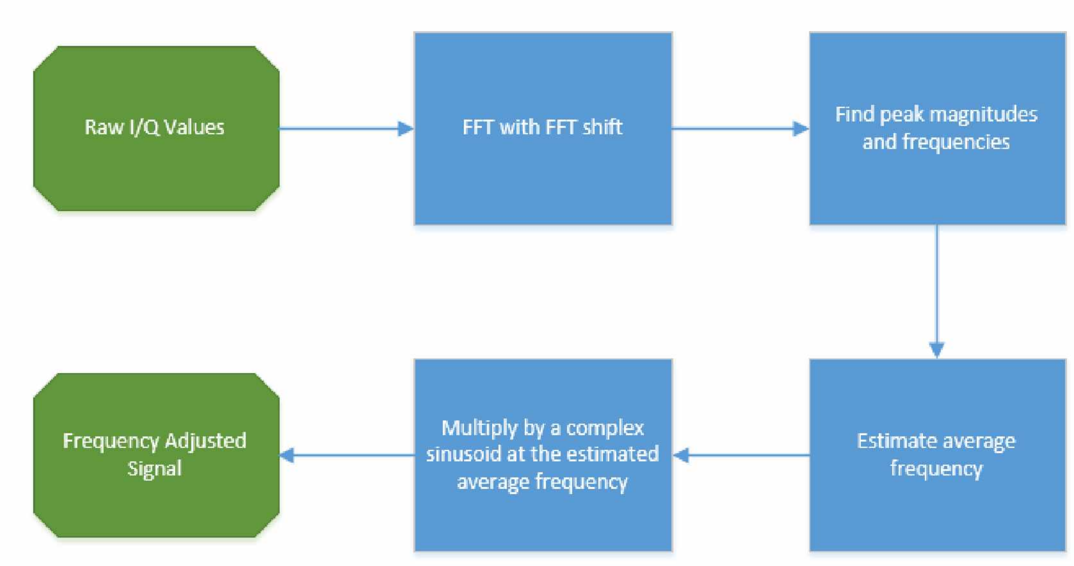

Figure 26: Coarse Frequency Compensation process for the *fsk_demod.m* function.

Figure 28 shows a received signal without Coarse Frequency Compensation, which is centered around -10 kHz. The signal should be centered on 0 Hz. How the *fsk_demod.m* function's frequency compensation works is a FFT is taken of the signal. The *fsk_demod.m* function stores the frequency peaks highlighted in the red circles. The *fsk_demod.m* function computes the average frequency between said maximum peaks. The average frequency is then turned into a complex sinusoid and multiplied by the original signal to provide Coarse Frequency Compensation.

Figure 28 shows an example of an uncompensated signal, the *fsk_demod.m* function calculated the peaks to be around -5 and -15 kHz. The average frequency was calculated to be -10 kHz. The average frequency was turned into a complex sinusoid using equation 8, where frequencyOffset is the average frequency and t is an array going from 0 to number of samples per frame, which is then multiplied by the sample period, 20.833 μs.

$$Complex\ sinusoid = e^{j2\pi(frequencyOffset)t} \tag{8}$$

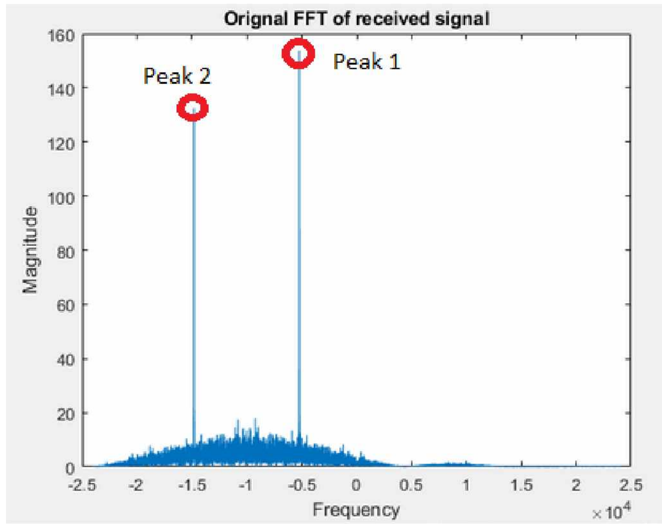

Figure 27: Example of an uncompensated frequency signal.

Figure 28 shows the compensated signal after the complex sinusoid multiplication of the original signal. As you can see, the signal is now centered on 0 Hz, which is crucial for FSK demodulation. A signal centered at 0 Hz allows the matched filters to work properly.

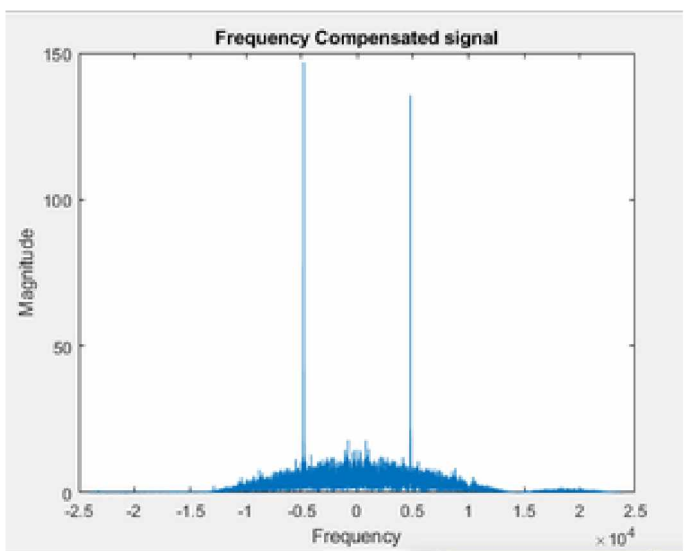

Figure 28: Example of a Frequency compensated signal.

5.2.4 Automatic Gain Control (AGC)

AGC is a method for increasing the power output of a signal. It is a built in Matlab function inside of the communications toolbox. The AGC will boost the power of a received signal, up to roughly the power at which the signal was sent. But, the AGC does come with some drawbacks. For instance, it has a massive overshoot, as shown in Figure 29. The max amplitude of the matched filter output should only be 5, but the AGC output reaches an amplitude of almost 1600.

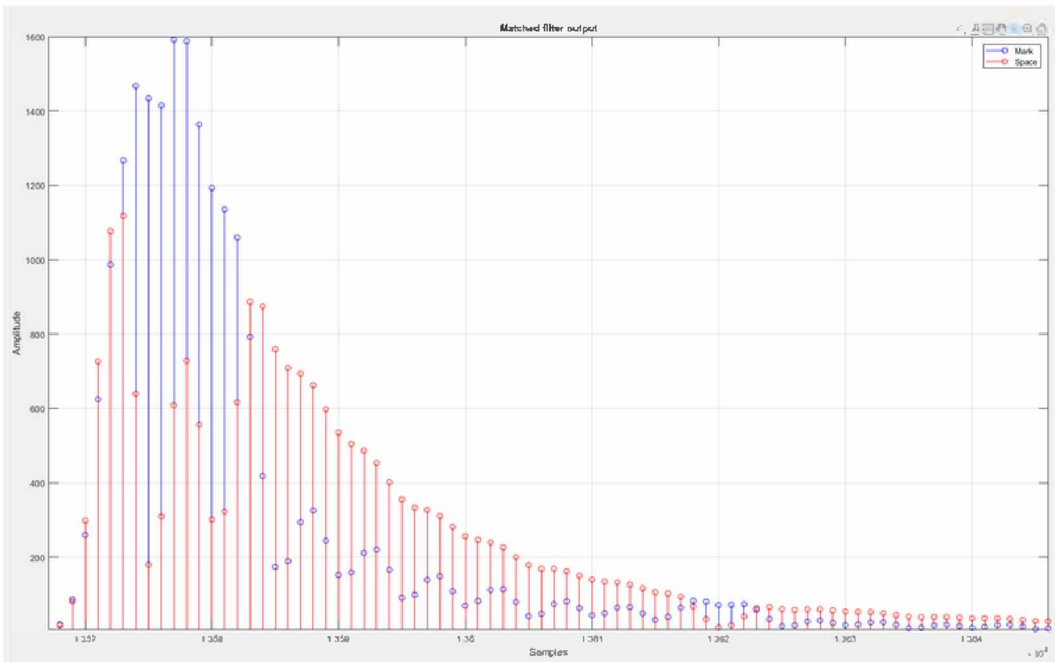

Figure 29: AGC transient response.

The other transient effect of the AGC is the settling time, which is roughly 50 samples before the matched filter output reaches an amplitude of 6 or less. The 50 samples lost equates to 10 bits, which is information lost from the preamble. The AGC can be altered so that the transient response is shorter, but that comes at the cost of stability. As the signal progresses, it may become unstable near the end of the signal. I would rather have information loss occur during the preamble than sacrifice bits in the payload. I would recommend not to change the AGC.

5.2.5 Matched filters

A matched filter is taking a series of samples for a symbol and performing a time reversal of the series of samples. Figure 30 shows the example of creating a matched filter. Essentially, you are just simply performing a left to right reflection of the samples of the Mark symbol. The same process for created a matched filter is repeated for the Space symbol.

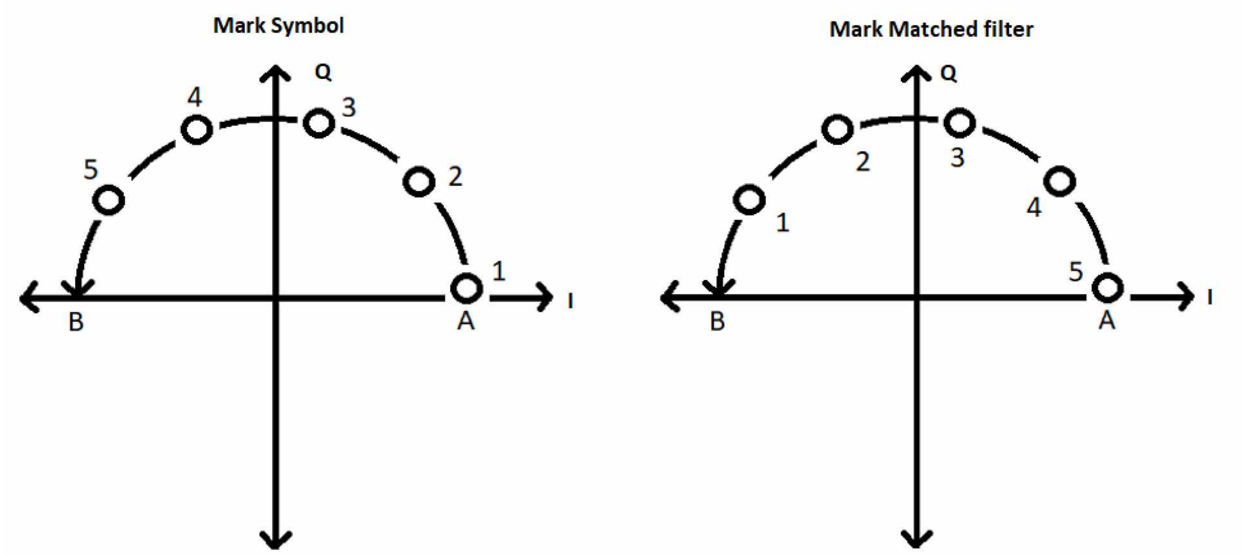

Figure 30: Process of creating a matched filter.

Figure 18 shows four different symbols, which would mean you would need four different matched filters. But, you only need to create two matched filters. One matched filter for the Mark symbol and one matched filter for the Space symbol. The reason why you can use two matched filters is because if you take the magnitude for both symbols for the Mark matched filters, you will have the same output. The same output means you will only need one matched filter for the Mark and one matched filter for the Space symbol.

The next step after creating the matched filters is to perform matched filtering. You perform convolution with the FSK signal with both matched filters. Then, you take the magnitude for both matched filter outputs. After you have the matched filter output, you need to choose a point for demodulation. Choosing a point for demodulation is choosing when a matched filter peak is a maximum and the other matched filter is at a minimum. If the Mark matched filter is greater than the space matched filter, it is a binary one. If the Space matched filter at the point is greater than the mark matched filter, it is a zero. After the point is chosen, the next spot will be 5 (number of samples per symbol) samples to the right and do perform a comparison. This process repeats until reaching the final sample. Figure 30 shows the ideal matched filter output of a simulation.

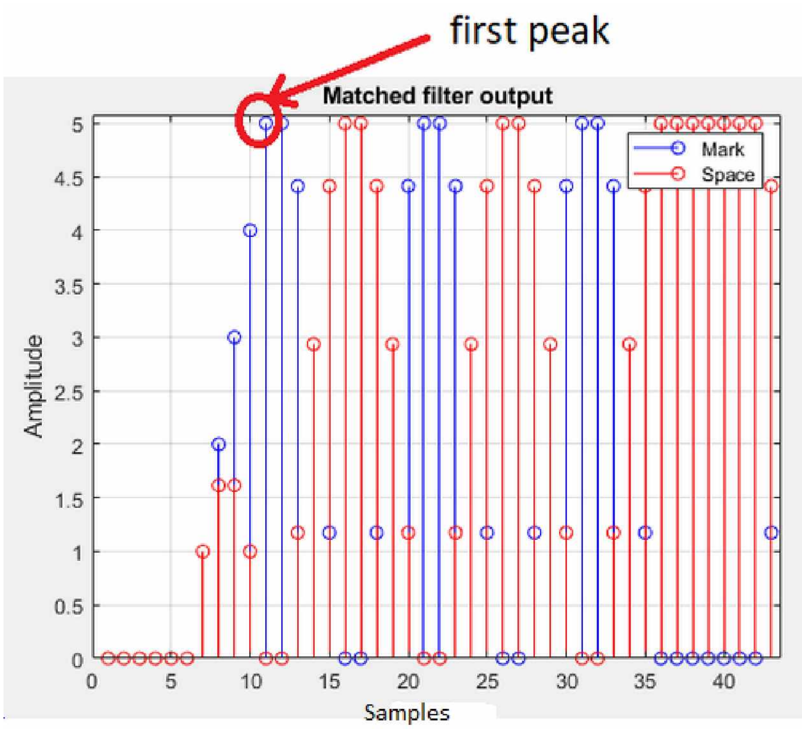

Figure 31: Matched filter output of an ideal signal (no noise).

To demodulate the ideal signal shown in Figure 30, you start at the first peak (located at sample 11). Since the Mark output is greater than the Space output, it will generate a binary one. Then you move to the next peak, which occurs at around sample 16. The next peak shows the space matched filter output is greater than the mark matched filters output, so it will be a binary 0. This process repeats and you will have a binary output of 1 0 1 0 1 0 0. Also, notice how the Space matched filter is at a minimum when the Mark matched filter output is at a maximum and vice versa? The reason why that happens is because the two frequencies are orthogonal to each other. When a Mark Symbol is convoluted with a Space symbol, the values cancel each other out, resulting in a zero output.

An actual signal will never be as easy to demodulate. The original signal will have been attenuated upon reaching the receiver, which is also contaminated by noise. The receiver signal and noise are then boosted, the effects of noise can be shown as uneven peaks in Figure 31. The boosted noise will increase the challenge of deciding between a mark and a space.

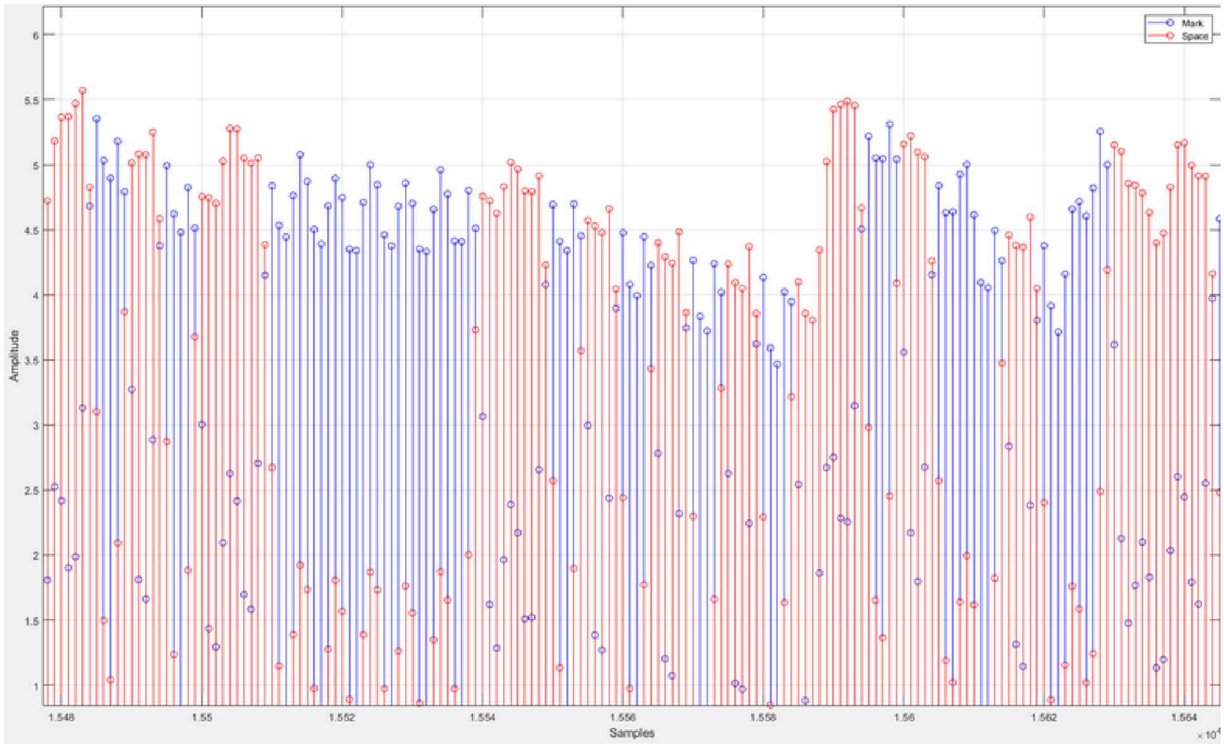

Figure 32: Matched filter output after the initial offshoot and both matched filters have settled.

Figure 32Figure 1 shows the matched filter output after the offshoot and settling time has settled. I couldn't use the same process of finding a peak of a matched filter for FSK demodulation when compared to the ideal matched filter. I created some criteria for looking for a starting point for demodulation. The criteria for choosing a starting point are listed below.

1. Start looking for a point after the largest peak.
2. After both matched filter outputs are below an amplitude of 6.
3. Look for a point for maximum difference between matched filters.

Point one refers to looking for samples after the initial offshoot, like the example shown in Figure 29. Point two is just referring to the signal after it has settled, meaning both matched filter outputs amplitude are below an amplitude of 6. For point three, it is looking for a point for when the difference is very noticeable. You cannot just pick a max amplitude like the figure shown in Figure 31, so I figured it would be easier to use the maximum difference between two filters for symbol timing. Once the difference is greater than say, 4.2, that means one filter is likely a maximum and the other matched filter is at a minimum, meaning it is the best time to decode the bit. After a point has been chosen for demodulation, it will start comparing the matched filter outputs to determine if a Mark or Space has been sent. After a bit is decoded, it will move 5 samples and do another comparison to output a bit. The process repeats until there are no more samples to process.

5.3 AX.25 decoding

Figure 33 shows the process for decoding AX.25 in the *ax25_decode.m* function.

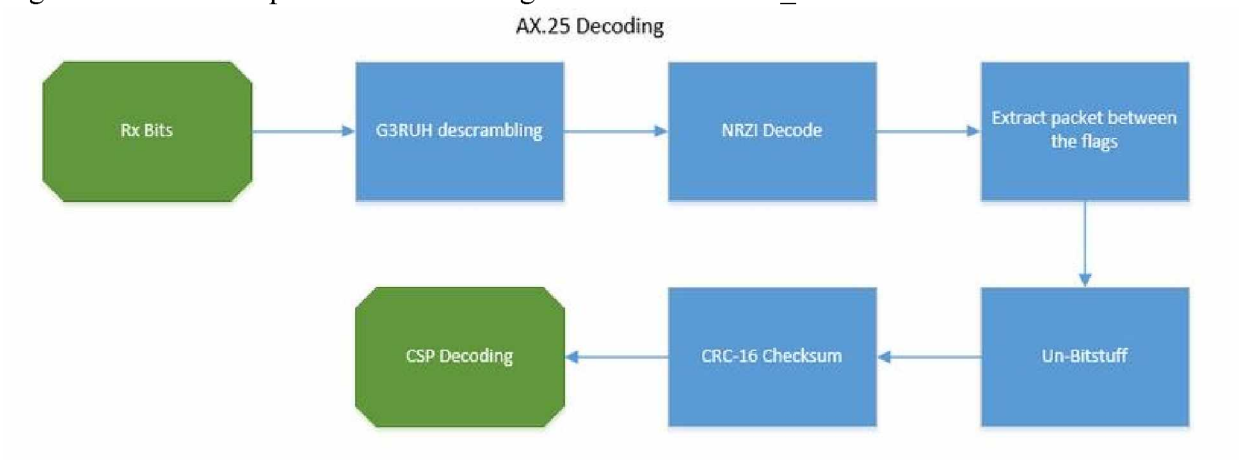

Figure 33: Signal flow process for AX.25 decoding in the *ax25_decode.m* function.

5.3.1 G3RUH descrambling

The descrambling polynomial is $x^{17} + x^{12} + 1$ and process is shown in Figure 34. The shift register is initialized to all zeros. The output is just the initial input bit XORd with the 12^{th} spot in the shift register XORd with the 17^{th} spot in the shift register.

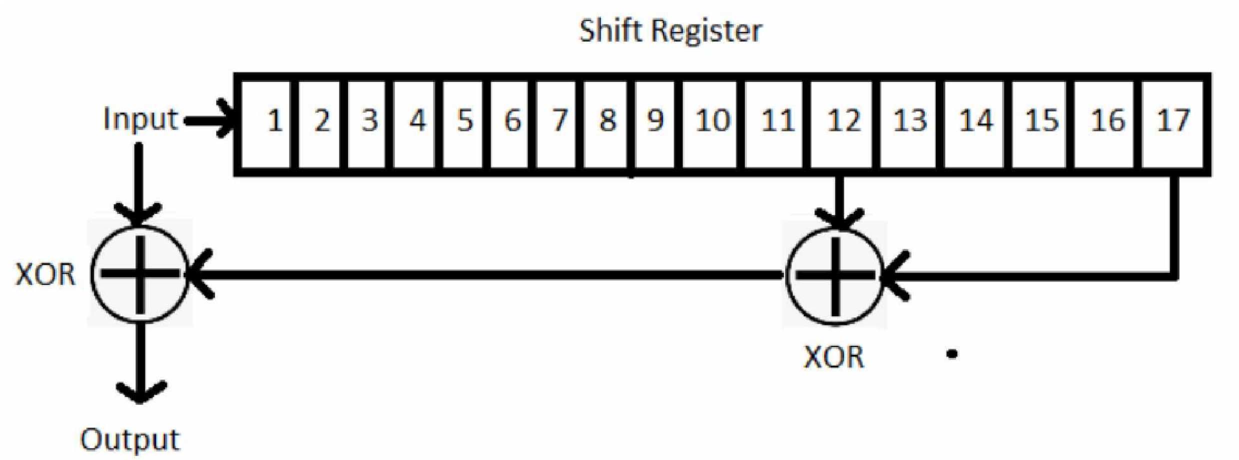

Figure 34: Descrambler process in the *ax25_decode.m* function.

5.3.2 NRZI Decoding

After the bits have been descrambled, the NRZI decoding process begins. NRZI decoding is just taking the signal, delaying it by one sample and looking for transitions. If the two bits are different, a binary 0 was encoded. If the two bits are the same, then a binary 1 was encoded. In summary, you are performing an XNOR with the bit stream with a delayed version of itself. Figure 35 shows the NRZI decoding process in the *ax25_decode.m* function. After NRZI decoding, the packet can be extracted.

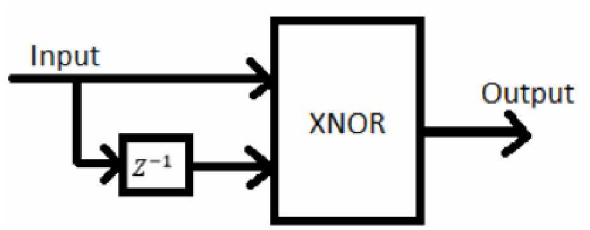

Figure 35: NRZI decoding process in the *ax25_decode.m* function.

5.3.3 Extract the packet between the flags

This process is just extracting all of the bits within the initial and ending flag. If you refer back to Figure 4, it will be extracting the AX.25 headers, payload and CRC16. The *ax25_decode.m* function operates by finding all of the flags present within the signal and searches for two flags that contain a minimum of 26 bytes space between them (16 bytes of AX.25 header, 4 bytes of CSP headers, 4 bytes of CRC32 and 2 bytes of CRC16). After the AX.25 headers, payload and CRC16 have been extracted, the un-bit stuffing process can begin.

5.3.4 Un-bit stuffing

Un-bit stuffing is just removing 0s after five consecutive ones, which is a part of the *ax25_decode.m* function. Un-bit stuffing is just removing all of the extra 0s that were stuffed in the original bit stream so the flag doesn't appear anywhere within the AX.25 headers, payload or CRC16. Figure 36 shows an example of the un-bit stuffing process. As you can see, it just removed the bit stuffed zero high-lighted in red from the bit stuffed sequence. A bit stuffed zero is defined as the zero after five consecutive ones.

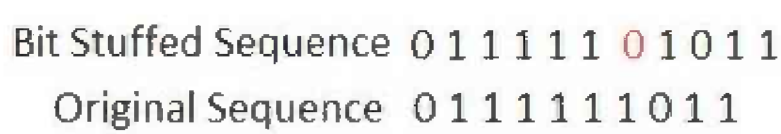

Figure 36: Example of the un-bit stuffing process.

5.3.5 CRC16 Checksum

The received CRC16 check sum is saved. The header and payload bits are fed into the *ax25_crc.m* function. The calculated CRC16 is then compared to the received CRC16. If there are no errors, the calculated and received CRC16 should match each other. If the calculated and received CRC16(s) do not match each other, then there were errors with the received packet. The CRC16 is capable of detecting up to 16 errors within the packet. After the CRC16 checksum has been made, the AX.25 headers and CRC16 are removed so that only the payload remains. The payload is comprised of an encoded CSP packet, which goes onto CSP decoding in section 5.4.

5.4 CSP Decoding

Figure 37 shows the decoding process for the CSP packet in the *csp_decode.m* function.

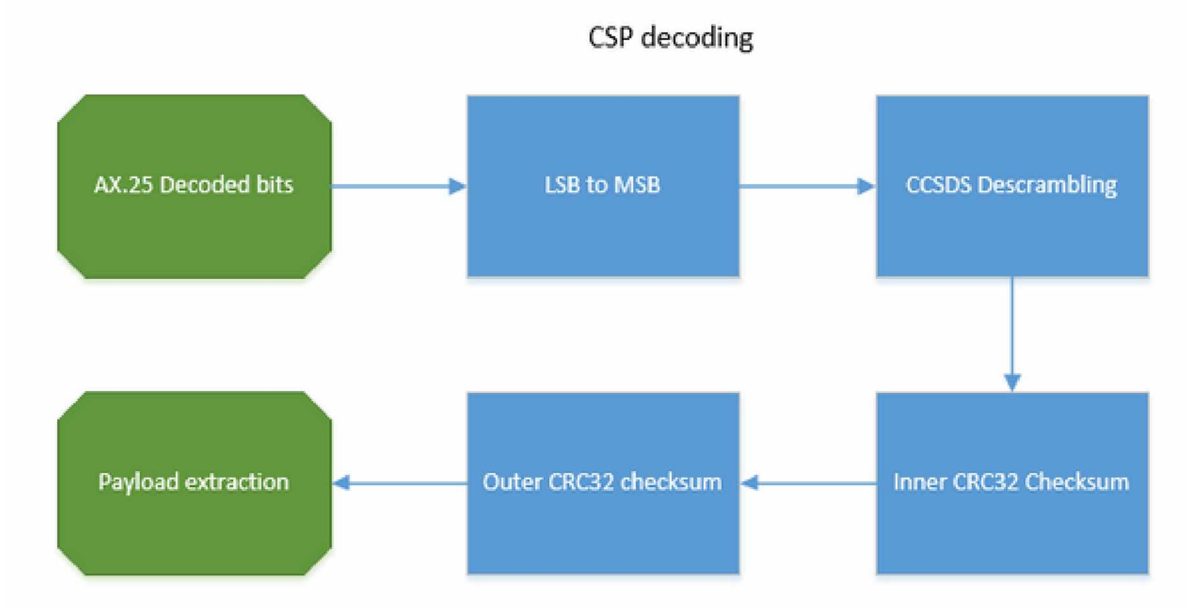

Figure 37: CSP decoding process in the *csp_decode.m* function.

5.4.1 LSB to MSB

This process takes each byte and changes it from LSBit to MSBit. This is because the format and encoding for CSP is MSBit. Changing from LSBit to MSBit is just a bit reversal for each byte.

5.4.2 CCSDS Descrambling

After each byte is in MSBit format, the descrambling process can begin. The bit stream is fed into Figure 8, which is the same process as scrambling. The function *CCSDS_Scrambler.m* function is the script used to descramble the CSP packet

5.4.3 CRC Check sums

There are two different process that are not elaborated in Figure 37. If the CSP header bit is 0, then there is going to be only 1 checksum. When the CSP CRC header bit is low, the CRC32 is saved, which is the last 32 bits of the CSP packet. The CRC checksum is calculated by using *crc.m* function, which the headers and payload, MSBit first, are fed into the *crc.m* function. After the CRC32 has been calculated, it is bitwise compared to the received CRC32 checksum. If the received and calculated checksums are the same, then are no errors present.

The second option is when the CSP CRC bit is high in the CSP headers. The last 4 bytes are saved as the 'Outer' CRC32 in the received packet. The last 8 through 5 bytes are saved as the 'Inner' CRC32. There will be two different CRC32s present within the CSP packet. The first or 'Inner' CRC32 is calculated using inputting the payload into the *crc.m* function. The second CRC32 or 'Outer' CRC32 is calculated by placing the headers, payload and 'Inner' CRC32 into the *crc.m* function. The calculated 'Inner' CRC32 is then bitwise compared to the received CRC32 to see if there was any errors present within the payload. The received 'Outer' CRC32 is then bitwise compared to the calculated 'Outer' CRC32 to see if there were any errors present within headers, payload and 'Inner' CRC32.

5.4.4 Payload Extraction

After the CRC checksum(s) has been performed, no errors were detected, the headers and payload are sent to COSMOS using a TCP/IP connection (Not confirmed).

6. Testing

6.1 Setup

I tested the receiver part of *UHF_Transceiver.m* script by attaching a 30 dB attenuator to COMM and hooking up a SMA cable from COMM to the RX port of the B210.

The next test was to test the transmitter of the *UHF_Transceiver.m* script. This was done by attaching a 30 dB attenuator to the TX port of the B210 and attaching a SMA cable from the B210 to COMM.

6.2 Results

Two tests were performed with COMM and the *UHF_Transceiver.m* script. The first one was a ping, sent from COMM to *UHF_Transceiver.m* script. A ping counts from 0 to a number (max is 250). Figure 38 shows the AX.25 headers, CRC16 checksum, CSP headers, CSP payload and CRC32 checksum.

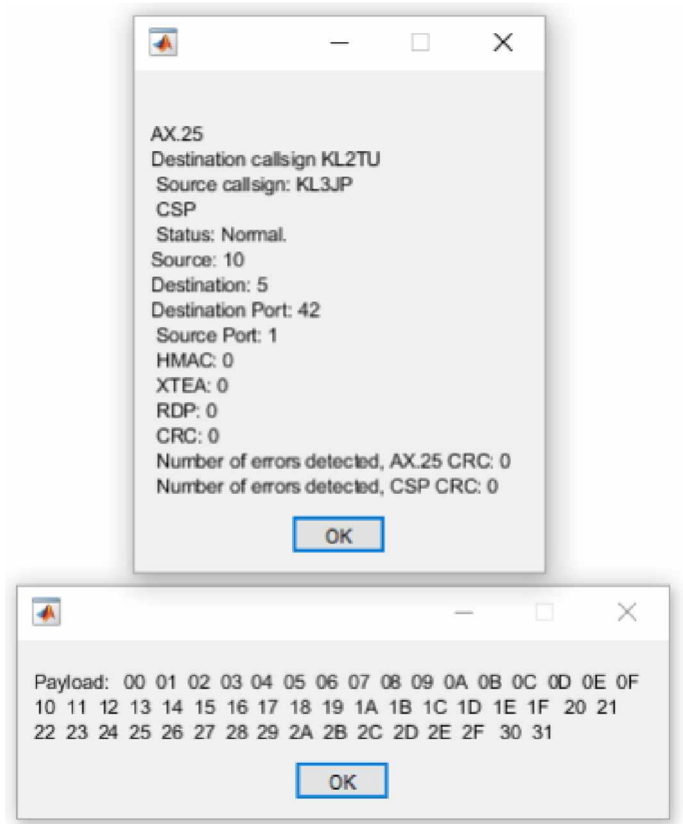

Figure 38: *UHF_Transceiver.m* script output on the receiver from a COMM subsystem ping.

To test the 'Inner' and 'Outer' CRCs on the receiver, a telemetry packet was sent. Figure 39 shows the AX.25 headers, CSP headers, AX.25 CRC16 and CSP CRC(s). There were no errors detected from the AX.25 CRC16 or 'Outer' CRC32, but there were 19 errors detected on the 'Inner' CRC32. The reason for the errors on the 'Inner' CRC32 is because I assumed it required header and payload bytes. After changing the 'Inner' CRC32 to be calculated from the payload, that is when the received and calculated 'Inner' CRC32 matched.

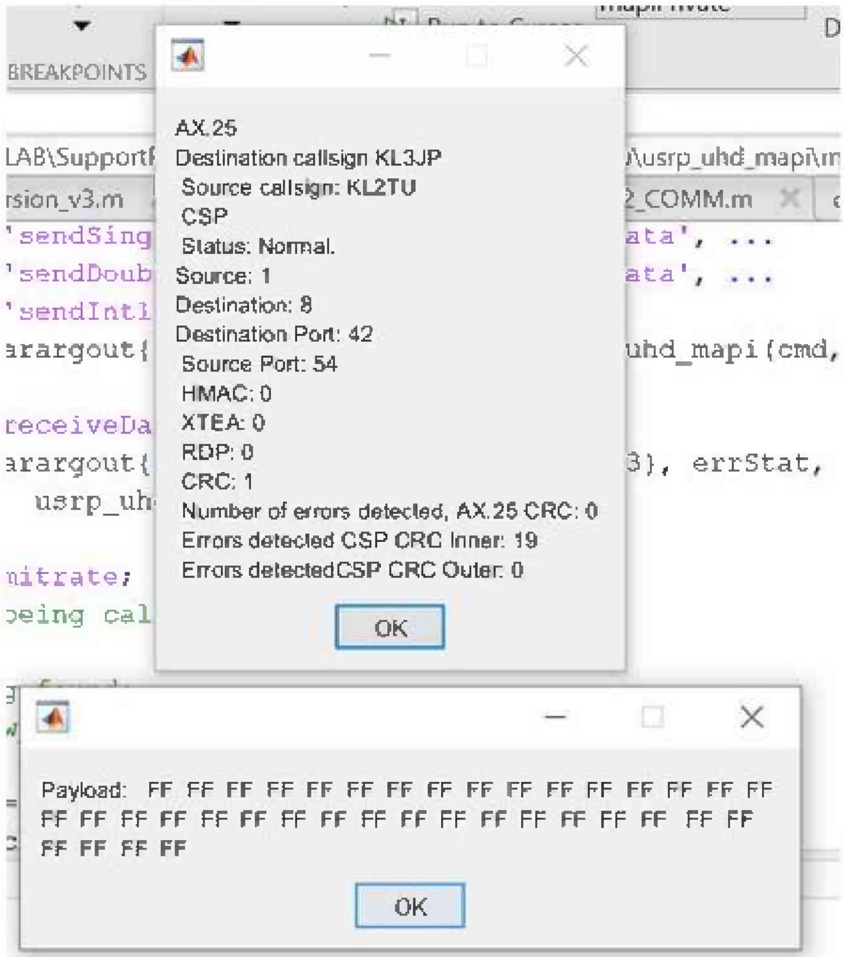

Figure 39: Decoded telemetry packet on the *UHF_Transceiver.m* script.

For testing my transmitter part of *UHF_Transceiver.m* script. I constructed a CSP packet with a payload with the ASCII string ‘To the Moon!”. Figure 40 shows the hex output on the CDH after COMM decoded my packet. The top part of CSP header information along with the payload. The bottom hex characters is what I sent, which matches what was received on the CDH. Figure 40 shows the transmitter being able to properly encode a message and COMM being able to properly decode a packet generated by the *UHF_Transceiver.m* script.

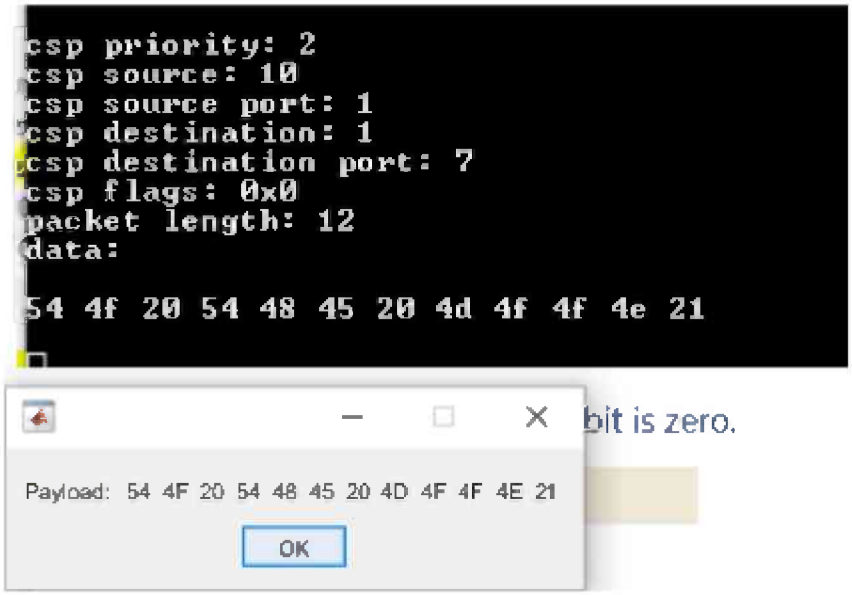

Figure 40: Comparison of the COMM output versus the payload I sent.

6.3 Conclusion

I was able to create a Matlab script that uses the B210 to communicate with CCP COMM. To communicate to the COMM, I had to generate a CSP packet and CSP encode the packet. After the packet was CSP encoded, I transform it into an AX.25 packet by adding AX.25 headers and CRC16. After the AX.25 packet is generated, that is when AX.25 encoding happens. After the packet is AX.25 encoded, that is when the encoded bits are converted into FSK samples. The samples are then sent to the B210 for transmission.

I also developed script that will be able to receive a packet from COMM. I had to develop a script that decodes FSK samples from the B210 into bits. After the bits are extracted, the bits are then AX.25 decoded. The AX.25 packet is then extracted, then checked for errors using the CRC16. If no errors are present, the AX.25 payload is then CSP decoded. The CSP packet will then be checked for errors using the CRC(s) built into the CSP packet.

Future work that needs to happen is testing the *UHF_Transceiver.m* script. Also, there needs to be a function that integrates COSMOS with *UHF_Transceiver.m* script, so the script can either send telemetry packets to COSMOS or receive Command packets from COSMOS. The RSSI threshold needs to be determined after low power testing. Also, I believe Coarse Frequency Compensation can be replaced by a Phase Locked Loop (PLL).

7. Web Resources

[1] https://www.tapr.org/pdf/AX25.2.2.pdf	Description AX.25 Link Access Protocol for Amateur Packet Radio.
[2] https://en.wikipedia.org/wiki/Doppler_effect	Website that provides the Doppler shift formula.
[3] https://en.wikipedia.org/wiki/Cubesat_Space_Protocol	Website that provides the CSP header structure.
[4] https://ntrs.nasa.gov/api/citations/20180002430/ downloads/20180002430.pdf	Nasa document detailing how to perform CCSDS scrambling.
[5] https://gomspace.com/UserFiles/Subsystems/datasheet/gs-ds-nanocom-ax100-33.pdf	GOMSpace AX100 data sheet, the COMM subsytstem onboard of CCP.
[6] https://ballaerospace.github.io/cosmos-website/	Website for the user interface for Command and Telemetry for Cosmos.

Acronyms:
AGC – Automatic Gain Control
CCP – Cubesat Communications Platform
CPFSK - Continuous Phase Frequency Shift Keying
CRC – Cyclic Redundancy Check
CSP – Cubesat Space Protocol
DAC – Digital to Analog Conversion
FCS – Frame Check Sequence
FSK – Frequency Shift Keying
FSM – Finite State Machine
GS – Ground Segment
I/Q – In-phase and Quadrature
LSBit – Least Significant Bit
MSBit – Most Significant Bit
NRZI – Non-Return to Zero Inverted
PID – Protocol Identifier
PLL – Phase Locked Loop
RS – Reed Solomon
RSSI – Received Signal Strength Indicator
USRP – Universal Software Radio Peripheral