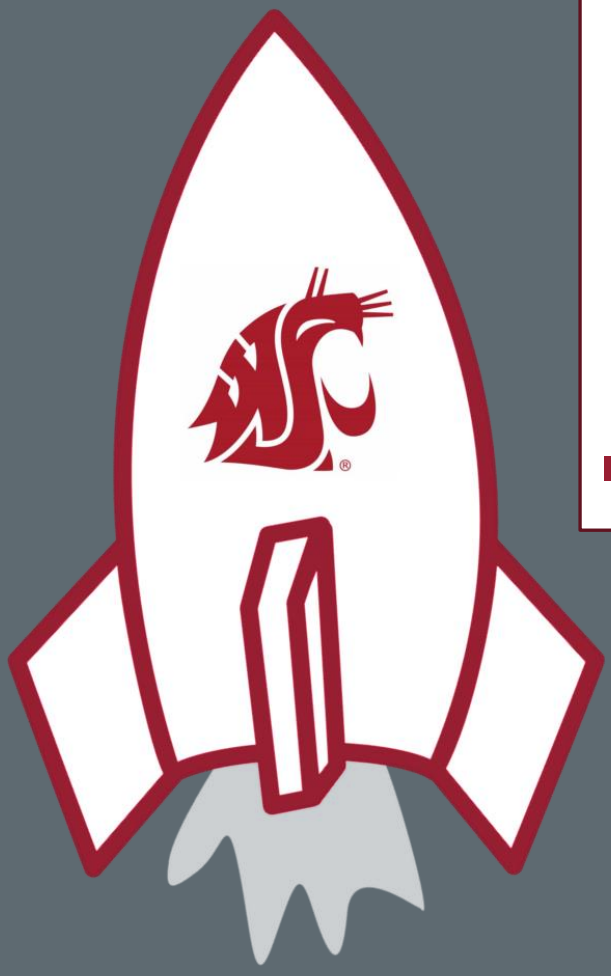
The communication to/from the satellite with the ground is organized into OSI layers. This document explains each layer in detail.

CougSatNet

CougSat and Ground Communication Protocol

Revision: 1.0.1



Bradley Davis

Table of Contents

1 Introduction

These are topics that should be fully comprehended to understand how CougSatNet works.

1.1 Open Systems Interconnection (OSI) Model

The OSI model[1] is a conceptual model that can be applied to any communication system. It has eight layers, see Table 1; each layer serves the layer above it and is served by the layer below it. This document will progress through each layer explaining its contents and how the layers interface with their neighbors.

Table 1: OSI Model Layers

Layer			Protocol Data Unit	Function
Host layers	7	Application	Data	High-level APIs
	6	Presentation		Translation of data between a networking service and an application
	5	Session		Managing communication sessions
	4	Transport	Segment	Reliable transmission of data segments between points on a network
Media layers	3	Network	Packet	Structuring and managing a multi-node network
	2	Data link	Frame	Reliable transmission of data frames between two nodes connected by a physical layer
	1	Physical	Symbol	Transmission and reception of raw bit streams over a physical medium
	0	Medium	Electrons, Photons	The physical medium: copper, fiber, wireless

1.2 Radio Frequency Wireless Transmission

The modulation of electromagnetic waves is used to carry information. When the carrier frequency is between $30Hz$ and $300GHz$, it is classified as radio frequency[2].

[1] For more information, read Wikipedia's article on the OSI model
[2] For more information, read Wikipedia's article on radio

1.2.1 Radio Frequency Components

1.2.1.1 Antenna

An AC current applied to a length of conductors will radiate energy outwards as electromagnetic waves with the same frequency and phase as the current. The output pattern depends on the shape of the conductors. This pattern is described generally by three parameters: bandwidth, gain, and polarization. There are many more parameters to characterize an antenna[3] but these are the important ones to understand a radio system.

The bandwidth specifies the range of frequencies that the antenna can effectively radiate. When a carrier signal is modulated to encode information, the frequency can spread out from the center and the antenna needs to be able to operate in this spectrum.

The gain specifies the directivity of the radiation pattern. Thanks to the conservation of energy, a passive antenna cannot amplify the signal so this parameter is better known as directivity. An antenna with an isotropic radiation pattern has a gain of $0dBi$[4]. Any angle outward from an isotropic antenna will have the same output energy. A higher gain antenna will have a higher directivity, that is more of its radiated energy is emitted in the same direction. This does mean that other off angles to a high gain antenna will not transmit as much energy therefore it needs to be aimed at the target better. The same property works in the reverse direction: a low gain antenna does not need to aimed very well but is not sensitive, a high gain antenna is more sensitive but needs to be aimed well.

The polarization specifies the direction of the electromagnetic wave that is transmitted or received by the antenna. A linear polarization will radiate waves in a plane. This plane needs to be aligned between the transmitter and receiver to reduce the cross-polarization loss. A circular polarization, will radiate waves that spin around the velocity of propagation. The angle between the transmitter and receiver does not matter but ensuring that both antennae are polarized in the same direction, left or right-handed, to reduce cross-polarization loss. A linear and circular-polarization antenna can communicate to each other but will incur a fixed cross-polarization loss regardless of angle. Note that reflections, refractions, and other physics, will bend the polarization from these ideal shapes into a general elliptical polarization.

1.2.1.2 Amplifier

An amplifier multiplies a small signal to increase the amplitude to a large signal. An ideal amplifier will multiply the small signal by a fixed gain. A real amplifier has a maximum output power so larger input signals cannot by

[3] For more information, see Wikipedia's article on radio antennae
[4] Units are decibels with a reference value of an isotropic antenna

amplifier by the full gain. This is called compression or clipping depending on the behavior.

1.2.1.3 Filter

A filter removes unwanted frequency components from a signal. There are many implementations of filters in hardware and software which are not explained here. A filter is named by the frequency response: a low-pass filter passes low frequencies and attenuates (removes) high frequencies. There are also high-pass, band-pass, band-stop, or any combination. A simple low pass filter is taking the average of a range of data: the outliers are removed (high frequency components) and average follows the value of the middle of the group (low frequency components)

1.2.1.4 Combiner

A combiner adds two signals together in the time domain, see equation 1. In the frequency domain, a combiner superpositions the two input frequencies. Note how the output in equation 1 has both input frequencies.

$$C[A(t), B(t)] = C[\sin(at), \sin(bt)] = \sin(at) + \sin(bt) \quad (1)$$

1.2.1.5 Mixer

A mixer multiplies two signals together in the time domain, see equation 1. In the frequency domain, a mixer adds and subtracts the two input frequencies, see equation 2 and Figure 1, yielding an lower and upper sideband. Most often only one output frequency is desired, so the unwanted output frequency is attenuated (removed) with a Filter. The $\frac{1}{2}$ scaler represents how the output power is shared between the two output frequencies, referred to as the conversion loss.

$$M[A(t), B(t)] = M[\sin(at), \sin(bt)] = \sin(at) * \sin(bt) = \frac{\sin[(a+b)t] + \sin[(a-b)t]}{2} \quad (1)$$

$$f_{out} = f_a \pm f_b \quad (2)$$

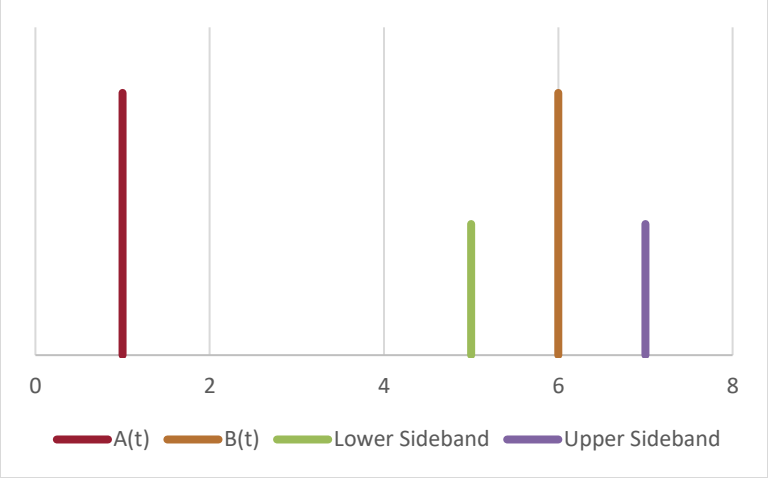


Figure 1: Mixer in the frequency domain

The derivation of the mixer's equation is as follows. Note how the result has a cos function not sin as the mixer is described. This is because the phase shift is ignored.

$$\sin(\theta + \phi) = \sin(\theta)\cos(\phi) + \cos(\theta)\sin(\phi)$$

$$\sin(\theta - \phi) = \sin(\theta)\cos(\phi) - \cos(\theta)\sin(\phi)$$

$$\sin(\theta + \phi) + \sin(\theta - \phi) = 2[\sin(\theta)\cos(\phi)] + 0[\cos(\theta)\sin(\phi)]$$

$$\frac{\sin(\theta + \phi) + \sin(\theta - \phi)}{2} = \sin(\theta) * \cos(\phi)$$

If the mixer results in a negative frequency, it is reflected about the origin and becomes positive, referred to as frequency folding. There is no real mechanism that does this, just a mathematical convenience.

1.2.2 Modulation

An electromagnetic wave can be described by its frequency, amplitude, and phase. Each of these parameters can be varied with time, or modified, to encode information which is called modulation[5]. The modulation schemes used in CougSatNet are described here.

1.2.2.1 In-Phase and Quadrature (I/Q) Components

A wave of time varying amplitude, frequency, and/or phase can be described by the time varying amplitude of its in-phase and quadrature components, equation 1. These components are two waves that are shifted by 90° to each other, hence in-phase and quadrature. Note how the components, are the only time dependent signals, the carrier frequency, ω, is constant. Since the

[5] For more information, see Wikipedia's article on modulation

components are sin and cos complements, they are plotted on a phasor diagram called a constellation diagram.

$$A(t) * \sin[\omega(t)t + \phi(t)] = I(t) * \cos(\omega t) + Q(t) * \sin(\omega t) \quad (1)$$

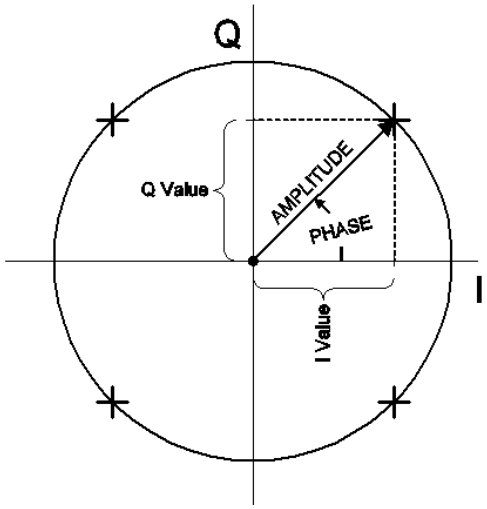


Figure 2: I/Q Constellation Diagram[6]

To amplitude modulate the wave, simply the amplitude of both I/Q components are varied together. On the constellation diagram, the amplitude of the phasor varies but the phase remains constant.

To phase modulate the wave, change the amplitude of I/Q separately. With $I = 1$ & $Q = 0$, the resultant wave is a cos wave, from equation 1. With $I = 0$ & $Q = 1$, the wave is a sin wave (a cos wave with 90° phase offset). Any combination between will be a different phase. On the constellation diagram, the phase of the phasor varies but the amplitude remains constant.

To frequency modulate the wave, continuously vary the phase of the wave around the circle. A $1Hz$ modulation will rotate the phasor around the circle once every second. A frequency modulation is just a continuous phase modulation. This is described by how multiplying two waves in the time domain is the same as adding in the frequency domain, see Mixer.

1.2.2.2 Quadrature Phase-Shift Keying (QPSK)

This modulation scheme shifts the phase of the wave to indicate one of four symbols. The shifts are 90° apart and usually aligned to 45°. The phase is a step function which is represented by stepping the baseband in-phase and quadrature phase components between ± 1. This is convenient as electronics are very good at generating square waves rather than synthesizing a sine wave with phase shifts.

[6] Source: https://www.tek.com/blog/what%E2%80%99s-your-iq-%E2%80%93-about-quadrature-signals%E2%80%A6

A constellation diagram, a plot of $I(t)$ versus $Q(t)$ taken at the sampling points, reveals four dots at each of the angles described above. These points are $(\pm 1, \pm 1)$, see Figure 3. A window around these points is used to identify the symbols during demodulation.

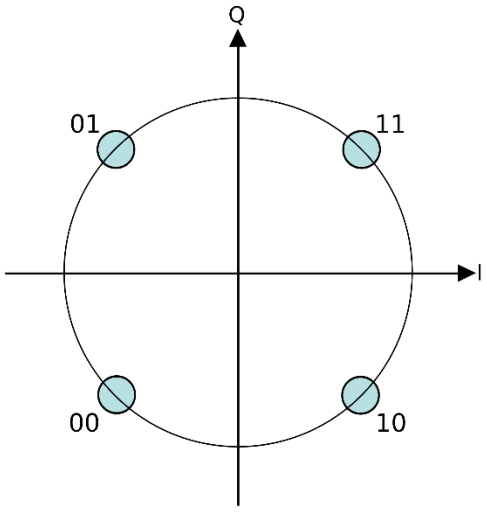


Figure 3: QPSK Modulation Constellation Diagram[7]

1.2.2.3 Data Over Voice (DOV)

This modulation technique allows both voice (spoken words) and data to be simultaneously transmitted. The baseband signals are modulated with the sum of amplitude modulation for voice and frequency modulation for data.

Amplitude modulation takes both the in-phase and quadrature components and varies their amplitudes together. For voice, this is done according to the sound being encoded. On the constellation diagram, the dot moves along the $45°$ diagonal, including negative amplitude.

Frequency modulation could be viewed as a continuous phase shift modulation. On the constellation diagram, the dot moves around in a circle with a fixed amplitude. The faster it moves around, the farther from the carrier frequency the result is. The baseband in-phase component gets a sine wave with a frequency of the modulation and the quadrature gets that wave shifted by $90°$. Most often the symbols are distinct tones.

Human speech operates in the $30Hz - 3.4kHz$ range. For the data, the symbols should be outside of this range to not interfere. Furthermore, human hearing is $20Hz - 20kHz$ which could result in an audible hum if the symbols are located within this range. Some radios will band-pass only the speech range to reduce noise from a $10kHz$ tone for example.

[7] Source: https://commons.wikimedia.org/wiki/File:QPSK_Gray_Coded.svg

Together data over voice will look on the constellation diagram as a weird mess. Think the voice is moving randomly along its line then the frequency modulation is a circle around that point. Luckily, a band-pass filter will separate these signals.

1.2.3 Demodulation

Taking a wave and extracting its information is demodulation. This is commonly done by down converting the wave to remove the carrier frequency then extracting its In-Phase and Quadrature (I/Q) Components which can be sampled to yield information.

1.2.3.1 Down Converting

Just as the wave was synthesized by multiplying the baseband signals by a carrier wave using a mixer, the received wave is multiplied by the same carrier wave resulting in the baseband signals. Multiplying by two carriers, separated by $90°$ results in the in-phase and quadrature baseband components. Ideally, without any offsets of any kind, these received I/Q components are the same as the transmitted pair. Since there will be offsets introduced and noise added, this process is trickier.

1.2.3.2 Frequency Offset

A frequency offset, which looks the same as the signal having a carrier frequency, depicted as the constellation diagram rotating around, is encountered due to doppler shift, poor carrier frequency tuning, or any other reasons. Based on the orbital velocity, the satellite will encounter at most $25ppm$ of shift due to doppler. The hardware clocks can synthesize frequencies within $5ppm$.

1.2.3.3 Phase Offset

A phase offset, depicted as the constellation diagram being at a fixed rotation, is encountered due to phase offset in the synthesized carrier frequency when down converting.

1.2.3.4 Symbol Timing Offset

Just as the doppler shifts the carrier, it also shifts the baseband signal containing the symbols. The receiver needs to know when to sample the baseband signals to avoid sampling on a bit transition. It will adjust sample timing to measure in the middle of each bit.

1.2.3.5 Costas Loop

A Costas Loop[8] is a Phase-Locked Loop (PLL), its output phase is continuously adjusted to match a reference, that works well with QPSK modulated signals. The input signal is fed into a mixer whose carrier signal is the output phase of the next stage. The mixed signal's phase, once locked only contains the demodulated signal, is measured from the ideal phase angle, this

[8] For more information, see Wikipedia's article on Costas Loop

error is fed into the aforementioned mixer to remove the offset on the next sample. See Appendix A: QPSK MATLAB Code for an implementation of the loop. The demodulated output is lastly filtered to reduce the noise.

The first sample received has some random phase offset, either due to a constant phase offset or a frequency offset, for sake of example $117° = \langle -0.45, 0.89 \rangle$. This sample is decided to be one of the four symbols based on which quadrant it lies, $135° = 01$. The difference in angle is the error, which can be found from taking the cross product of the two vectors. Thanks to the small angle approximation, $\sin(x) \approx x$, the angle between the vectors is $\theta \approx \frac{\hat{I}Q - \hat{Q}I}{|\hat{I}\hat{Q}||IQ|}$, where $\hat{I}$ & $\hat{Q}$ are the ideal vector found by taking the sign of the real vector. In the example, $\frac{\langle -0.45, 0.89 \rangle \times \langle -1, 1 \rangle}{1 * \sqrt{2}} = \frac{-0.89 - (-0.45)}{\sqrt{2}} = -0.31 rad = -17.8°$. This is fed into the PLL and the baseband signal is rotated by $25°$. With a frequency offset, each sample will be rotate by the same amount and each time this phase is adjusted for.

This type of feedback cannot detect if a symbol is more than $45°$ from where it should be. Past that phase shift, it thinks the symbol is the next one. This is corrected with a fixed preamble such that the software can read the decoded bit stream and rotate the symbols until a known pattern matches up. If a frequency offset exceeds $\frac{45°}{sample} = \frac{f_{sample}}{8}$, the shift aliases and begins to look like a slower shift. For example, if the symbols were rotating by $90°$ every sample, the feedback would think there is no offset and the data is actually changing that quickly.

1.3 Cyclic-Redundancy Check

A CRC is a type of hash function that is used to detect bit errors in a bit stream by dividing the entire file by an *Algorithm*

The CRC algorithm is similar to long division as the entire file, a number with billions of bits, is divided starting with the most significant bit and systematically shifting to the least significant bit.

1.3.1.1 Modulo Division Error Checking

Take a bit stream, for example "CIS" represented in ASCII, add an empty byte to the end. Turn it into a single number, in this case it has $32b$ and is $1{,}128{,}878{,}848$. Then take that number and find the remainder after dividing by 49, see equation 1. Our example has a check value of 41.

$$1128878848 \; mod \; 49 = 41 \tag{1}$$

After computing the check value, we can add the difference between the divisor and the check value to the number. Since our divisor is $8b$ and an empty byte was appended to the bit stream, the difference is easily moved into that

location. With that difference added, the bit stream is now evenly divisibly by the divisor, see equation 2.

$$(1128878848 + (49 - 41)) \; mod \; 49 = 0 \; \#2$$

When the destination received the adjusted bit stream, it can perform the same calculation and hopefully yield zero which means no errors were introduced. Since the math is dividing by the divisor, if bit errors were introduced that changed the bit stream by a factor of the divisor, add $49 \; or \; 98$ in the example, that would not be detectable by the CRC.

1.3.1.2 Number System

When an error is introduced into the bit stream, it flips the bits but mathematically could look like adding a very large number that spans multiple bits in the case that the error causes a long chain of carrying. To avoid this, the bit stream is represented as a polynomial, for example $x^3 + x^1 + 1$, and mathematical operations are restricted to a single bit using finite fields.

A byte, $'C' = 0100\ 0011b$, is equivalent to each bit multiplied by its weight, $0(2^7) + 1(2^6) + 0(2^5) + 0(2^4) + 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0)$. This is the same for base-10 numbers, the base is just ten instead of two. With CRC, the base used is x which doesn't really mean anything it just helps with the math, $'C' = 0100\ 0011b$=1x6+1x1+1x0. The example "CIS" would be represented with the polynomial in equation 3.

$$1x^{22} + 1x^{17} + 1x^{16} + 1x^{14} + 1x^{11} + 1x^8 + 1x^6 + 1x^4 + 1x^1 + 1x^0 \quad (3)$$

Dividing this bit stream polynomial with the CRC polynomial using normal math results in terms with any constants. This is a problem as binary numbers only have ones and zeros. The solution is a finite field that has only ones and zeros, the operations are explained in Table 2.

Table 2: CRC Finite Field

Addition			Subtraction	
$0 + 0 = 0$	$0 + 1 = 1$		$0 - 0$=0	$0 - 1$=1
$1 + 0 = 1$	$1 + 1 = 0$		$1 - 0$=1	$1 - 1$=0
Multiplication			Division	
$0 * 0 = 0$	$0 * 1 = 0$		$\frac{0}{0} = undef.$	$\frac{0}{1} = 0$
$1 * 0 = 0$	$1 * 1 = 1$		$\frac{1}{0} = undef.$	$\frac{1}{1} = 1$

1.3.1.3 Calculation

The bitstream, equation 1, is appended with enough bits to store the remainder, equation 2. Then it is divided by the polynomial, equation 3. The remainder, equation 4, is then subtracted from the bit stream, equation 5, same

as moving it into the reserved section. This makes the adjusted bitstream evenly divisible by the polynomial.

$$"CIS" = \ 01000011\ 01001001\ 01010011b \tag{1}$$

$$"CIS"\ lsh\ 8 = 01000011\ 01001001\ 01010011\ 00000000b \tag{2}$$

$$(x^{30} + x^{25} + x^{24} + x^{22} + x^{19} + x^{16} + x^{14} + x^{12} + x^9 + x^8)\ mod\ (x^8 + x^2 + x^1 + 1) \tag{3}$$

$$remainder = x^6 + x^5 + x^1 + 1 = 0110\ 0011b \tag{4}$$

$("CIS"\ lsh\ 8) - remainder = 01000011\ 01001001\ 01010011\ 01100011b$ #5□□

This calculation in hardware translates to: process the bitstream the length of the CRC at a time; if the most significant bit is a one, bitwise XOR the polynomial; shift the bitstream left by one and repeat. For an example, see the calculation below.

```
 01000011 01001001 01010011 00000000
- 1000001 11
=00000010 10001001 01010011 00000000
-       10 0000111
=00000000 10000111 01010011 00000000
-          10000011 1
=00000000 00000100 11010011 00000000
-                100 000111
=00000000 00000000 11001111 00000000
-                   10000011 1
=00000000 00000000 01001100 10000000
-                    1000001 11
=00000000 00000000 00001101 01000000
-                       1000 00111
=00000000 00000000 00000101 01111000
-                        100 000111
=00000000 00000000 00000001 01100100
-                          1 00000111
=00000000 00000000 00000000 01100011
```

The polynomial division cannot detect leading zeros as $0001 = 1$ nor trailing zeros as 22 and 220 are both divisible by 11. To detect leading zeros, the remainder is initialized with a known value, usually all ones. This is mathematically the same as inverting the first n bits. It does not affect the calculation and a CRC over the bit stream and the remainder will still be zero. To detect trailing zeros, the remainder is inverted before adding to the bit stream. When performing the calculation over the bit stream and the inverted remainder, the result is no longer zero but instead a fixed non-zero value.

1.3.1.4 Lookup Table

The hardware calculation explained above is a bit-by-bit algorithm. To increase efficiency, the algorithm is moved to a byte-by-byte operation albeit more obfuscated. A table of 256 codes is initialized by performing the bit-by-bit algorithm on each index. This is done either statically or once when the hardware boots up. Equation 1 is performed on each byte of the bitstream using the generated table.

$$remainder = (remainder\ lsh\ 8)\ xor\ table[byte\ xor\ (8\ MSB\ of\ remainder)] \quad (1)$$

This algorithm expressed in pseudocode is as follows:

```
rem = -1; // All ones
for (b in byteArray) {
  rem = (rem << 8) ^ table[b ^ (rem >> (n-8))];
}
rem = ~rem; // Invert all the bits
```

Polynomial. The remainder is subtracted from the bit stream which makes the bit stream even divisible by the polynomial. When the destination takes the entire bit stream and performs the polynomial division, the remainder is zero when there are no errors[9] and non-zero is there were errors.

1.3.2 Algorithm

The CRC algorithm is similar to long division[10] as the entire file, a number with billions of bits, is divided starting with the most significant bit and systematically shifting to the least significant bit.

1.3.2.1 Modulo Division Error Checking

Take a bit stream, for example "CIS" represented in ASCII, add an empty byte to the end. Turn it into a single number, in this case it has $32b$ and is $1{,}128{,}878{,}848$. Then take that number and find the remainder after dividing by 49, see equation 1. Our example has a check value of 41.

$$1128878848\ mod\ 49 = 41 \quad (1)$$

After computing the check value, we can add the difference between the divisor and the check value to the number. Since our divisor is $8b$ and an empty byte was appended to the bit stream, the difference is easily moved into that location. With that difference added, the bit stream is now evenly divisibly by the divisor, see equation 2.

$$(1128878848\ + (49 - 41))\ mod\ 49 = 0 \quad (2)$$

[9] The errors introduced could be divisible by the polynomial and thus not detectable. This makes the choice of the polynomial very important and difficult.
[10] For an excellent video explaining how CRC works, check out Ben Eater's, this explanation is heavily influenced by his work

When the destination received the adjusted bit stream, it can perform the same calculation and hopefully yield zero which means no errors were introduced. Since the math is dividing by the divisor, if bit errors were introduced that changed the bit stream by a factor of the divisor, add $49\ or\ 98$ in the example, that would not be detectable by the CRC.

1.3.2.2 Number System

When an error is introduced into the bit stream, it flips the bits but mathematically could look like adding a very large number that spans multiple bits in the case that the error causes a long chain of carrying. To avoid this, the bit stream is represented as a polynomial, for example $x^3 + x^1 + 1$, and mathematical operations are restricted to a single bit using finite fields.

A byte, $'C' = 0100\ 0011b$, is equivalent to each bit multiplied by its weight, $0(2^7) + 1(2^6) + 0(2^5) + 0(2^4) + 0(2^3) + 0(2^2) + 1(2^1) + 1(2^0)$. This is the same for base-10 numbers, the base is just ten instead of two. With CRC, the base used is x which doesn't really mean anything it just helps with the math, $'C' = 0100\ 0011b = 1x^6 + 1x^1 + 1x^0$. The example "CIS" would be represented with the polynomial in equation 3.

$$1x^{22} + 1x^{17} + 1x^{16} + 1x^{14} + 1x^{11} + 1x^8 + 1x^6 + 1x^4 + 1x^1 + 1x^0 \quad (3)$$

Dividing this bit stream polynomial with the CRC polynomial using normal math results in terms with any constants. This is a problem as binary numbers only have ones and zeros. The solution is a finite field that has only ones and zeros, the operations are explained in Table 2.

Table 2: CRC Finite Field

Addition			Subtraction	
$0 + 0 = 0$	$0 + 1 = 1$		$0 - 0 = 0$	$0 - 1 = 1$
$1 + 0 = 1$	$1 + 1 = 0$		$1 - 0 = 1$	$1 - 1 = 0$
Multiplication			Division	
$0 * 0 = 0$	$0 * 1 = 0$		$\frac{0}{0} = undef.$	$\frac{0}{1} = 0$
$1 * 0 = 0$	$1 * 1 = 1$		$\frac{1}{0} = undef.$	$\frac{1}{1} = 1$

1.3.2.3 Calculation

The bitstream, equation 1, is appended with enough bits to store the remainder, equation 2. Then it is divided by the polynomial, equation 3. The remainder, equation 4, is then subtracted from the bit stream, equation 5, same as moving it into the reserved section. This makes the adjusted bitstream evenly divisible by the polynomial.

$$"CIS" = \ 01000011\ 01001001\ 01010011b \quad (1)$$

$$"CIS"\ lsh\ 8 = 01000011\ 01001001\ 01010011\ 00000000b \quad (2)$$

$$(x^{30} + x^{25} + x^{24} + x^{22} + x^{19} + x^{16} + x^{14} + x^{12} + x^9 + x^8)\ mod\ (x^8 + x^2 + x^1 + 1) \quad (3)$$

$$remainder = x^6 + x^5 + x^1 + 1 = 0110\ 0011b \quad (4)$$

$$("CIS"\ lsh\ 8) - remainder = 01000011\ 01001001\ 01010011\ 01100011b \quad (5)$$

This calculation in hardware translates to: process the bitstream the length of the CRC at a time; if the most significant bit is a one, bitwise XOR the polynomial; shift the bitstream left by one and repeat. For an example, see the calculation below.

```
 01000011 01001001 01010011 00000000
- 1000001 11
=00000010 10001001 01010011 00000000
-      10 0000111
=00000000 10000111 01010011 00000000
-         10000011 1
=00000000 00000100 11010011 00000000
-              100 000111
=00000000 00000000 11001111 00000000
-                  10000011 1
=00000000 00000000 01001100 10000000
-                   1000001 11
=00000000 00000000 00001101 01000000
-                       1000 00111
=00000000 00000000 00000101 01111000
-                        100 000111
=00000000 00000000 00000001 01100100
-                          1 00000111
=00000000 00000000 00000000 01100011
```

The polynomial division cannot detect leading zeros as $0001 = 1$ nor trailing zeros as 22 and 220 are both divisible by 11. To detect leading zeros, the remainder is initialized with a known value, usually all ones. This is mathematically the same as inverting the first n bits. It does not affect the calculation and a CRC over the bit stream and the remainder will still be zero. To detect trailing zeros, the remainder is inverted before adding to the bit stream. When performing the calculation over the bit stream and the inverted remainder, the result is no longer zero but instead a fixed non-zero value.

1.3.2.4 Lookup Table

The hardware calculation explained above is a bit-by-bit algorithm. To increase efficiency, the algorithm is moved to a byte-by-byte operation albeit more obfuscated. A table of 256 codes is initialized by performing the bit-by-bit algorithm on each index. This is done either statically or once when the hardware boots up. Equation 1 is performed on each byte of the bitstream using the generated table.

$$remainder = (remainder\ lsh\ 8)\ xor\ table[byte\ xor\ (8\ MSB\ of\ remainder)] \quad (1)$$

This algorithm expressed in pseudocode is as follows:

```
rem = -1; // All ones
for (b in byteArray) {
  rem = (rem << 8) ^ table[b ^ (rem >> (n-8))];
}
rem = ~rem; // Invert all the bits
```

1.3.3 Polynomial

Selection of a polynomial is influential on the types of errors the CRC can detect. If one knows the types of errors possibly introduced onto the bit stream, a polynomial can be chosen. Or one can pick a polynomial from a catalog based on its performance with the length of expected bit streams. Phillip Koopman maintains a catalog of best polynomials given a constant random independent bit error rate. He lists polynomials based on Hamming Distance for a maximum bit stream length. Hamming distance is the distance between symbols that the CRC is guaranteed to detect. For a hamming distance of four, the CRC is guaranteed to detect up to three-bit errors.

1.3.3.1 16 bit

`0xD175` – This polynomial has a Hamming distance of four for bit streams up to $32kb = 4kB$.

1.3.3.2 32 bit

`0xC9D204F5` – This polynomial has a Hamming distance of four for bit streams up to $2.2Gb = 268MB$.

1.4 8b/10b Encoding

This encoding scheme maps $8b$ words, read bytes, into $10b$ codes to achieve DC balance and sufficient data transitions for clock recovery from the transmitted bit stream. The maximum number of consecutive symbols, zeros and ones, is five and the maximum disparity, difference in count of zeros and ones, is two.

1.4.1 Encoding Mapping

A $8b$ word is split into two codes, each one gaining a bit: a $5b/6b$ code and a $3b/4b$ code. The resultant bit stream is bounded to a disparity of $-1\ or\ 1$. Each $8b/10b$ code has a disparity of $-2, 0, or\ 2$. A code with a non-zero disparity has both positive and negative variants. The appropriate version is selected to keep the running disparity within the bounds. The $5b/6b$ code is sent first, then the $3b/4b$ code.

The $8b$ word's bits are assigned letters A-H, A being the most significant bit and H the least significant. The $5b/6b$ code uses bits $ABCDE$, adds a x bit and

become $abcdex$, see Table 3. The $4b/3b$ code uses bits FGH, adds a y bit and become $fghy$, see Table 4. Therefore the $8b$ word $ABCDEFGH$ is mapped to $abcdexfghy$.

Table 3: 5b/6b Encoding Mapping

Input		$RD = -1$	$RD = +1$		Input		$RD = -1$	$RD = +1$
Name	$ABCDE$	$abcdex$			Name	$ABCDE$	$abcdex$	
D.00	00000b	100111b	011000b		D.16	10000b	011011b	100100b
D.01	00001b	011101b	100010b		D.17	10001b	100011b	
D.02	00010b	101101b	010010b		D.18	10010b	010011b	
D.03	00011b	110001b			D.19	10011b	110010b	
D.04	00100b	110101b	001010b		D.20	10100b	001011b	
D.05	00101b	101001b			D.21	10101b	101010b	
D.06	00110b	011001b			D.22	10110b	011010b	
D.07	00111b	111000b	000111b		D.23	10111b	111010b	000101b
D.08	01000b	111001b	000110b		D.24	11000b	110011b	001100b
D.09	01001b	100101b			D.25	11001b	100110b	
D.10	01010b	010101b			D.26	11010b	010110b	
D.11	01011b	110100b			D.27	11011b	110110b	001001b
D.12	01100b	001101b			D.28	11100b	001110b	
D.13	01101b	101100b			D.29	11101b	101110b	010001b
D.14	01110b	011100b			D.30	11110b	011110b	100001b
D.15	01111b	010111b	101000b		D.31	11111b	101011b	010100b
					K.28	11100b	001111b	110000b

Table 4: 3b/4b Encoding Mapping

Input		$RD = -1$	$RD = +1$		Input		$RD = -1$	$RD = +1$
Name	FGH	$fghy$			Name	FGH	$fghy$	
D.x.0	000b	1011b	0100b		K.x.0	000b	1011b	0100b
D.x.1	001b	1001b			K.x.1	001b	0110b	1001b
D.x.2	010b	0101b			K.x.2	010b	1010b	0101b
D.x.3	011b	1100b	0011b		K.x.3	011b	1100b	0011b
D.x.4	100b	1101b	0010b		K.x.4	100b	1101b	0010b
D.x.5	101b	1010b			K.x.5	101b	0101b	1010b
D.x.6	110b	0110b			K.x.6	110b	1001b	0110b
D.x.P7[11]	111b	1110b	0001b		K.x.7	111b	1000b	0111b
D.x.A7[11]	111b	0111b	1000b					

1.4.1.1 Example Encoding

Table 5 has a four-byte stream that is encoded with $8b/10b$. Running disparity starts at $RD = -1$ and each byte updates that. The encoded bit stream has $19 - 0s$ and $21 - 1s$; a max of five consecutive ones and three consecutive zeros; and a maximum running disparity at any point of the stream of ± 3. Note

[11] Either the primary or alternate code is chosen to avoid more than five consecutive symbols. The alternate is used for only D.17, D.18, and D.20 for $RD = -1$ and for D.11, D.13, and D.14 for $RD = +1$

that encoding the same byte may result in an alternating output code, consecutive D.29.2.

Table 5: Example 8b/10b Encoding

Mode	Byte 1	Byte 2	Byte 3	Byte 4
Hex	0x97	0x19	0xEA	0xEA
Binary	10010111b	00011001b	11101010b	11101010b
8b/10b	0100110111b	1100011001b	0100010101b	1011100101b
Code	D.18.A7	D.03.1	D.29.2	D.29.2
RD	-1	$+1$	$+1$	-1	$+1$

1.4.2 Control Codes

Since not all codes that meet the requirements for running disparity and maximum consecutive symbols, some can be mapped to control codes. These are used for low level control such as synchronization, start and end delimiters, etc. See Table 6 for a list of codes. K.28.1, K.28.5, and K.28.7 are comma codes that can be used for synchronization, if K.28.7 is not used, the bit sequence `0011111b` nor `1100000b` cannot be found with any combination of normal codes. A sequence of multiple K.28.7 is not allowed as this would result in undetectable misaligned comma codes. K.28.7 is the only comma code that cannot be the result of a single bit error in the bit stream.

Table 6: 8b/10b Control Codes

Input			$RD = -1$	$RD = +1$
Name	Hex	$ABCDE\ FGH$	$abcdex\ fghy$	
K.28.0	0xE0	11100 000b	001111 0100b	110000 1011b
K.28.1	0xE1	11100 001b	001111 1001b	110000 0110b
K.28.2	0xE2	11100 010b	001111 0101b	110000 1010b
K.28.3	0xE3	11100 011b	001111 0011b	110000 1100b
K.28.4	0xE4	11100 100b	001111 0010b	110000 1101b
K.28.5	0xE5	11100 101b	001111 1010b	110000 0101b
K.28.6	0xE6	11100 110b	001111 0110b	110000 1001b
K.28.7	0xE7	11100 111b	001111 1000b	110000 0111b
K.23.7	0xBF	10111 111b	111010 1000b	000101 0111b
K.27.7	0xDF	11011 111b	110110 1000b	001001 0111b
K.29.7	0xEF	11101 111b	101110 1000b	010001 0111b
K.30.7	0xF7	11110 111b	011110 1000b	100001 0111b

1.5 Data Formatting

This document uses extensive use of byte and bit tables to present data. An example byte table can be found in Table 7. To read a byte table, the cell's offset from the beginning of the data set is given by summing the byte offsets in its row and column. In the example, *A* is found at `0x0003`, *B* is at `0x0009`, *C* is at `0x000D`, and *D* is at `0x0016`. Note that some values may span more than one byte with which its cells will be merged together, see *D.* Note that when a bit table is presented, the first cell's offset is zero and it is the most significant bit. This is reverse of the bit's value (the most significant bit has the highest position and value).

Table 7: Byte Table Example

Byte Offset	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x0000				*A*				
0x0008		*B*				*C*		
0x0010							*D*	

2 Layer Description

This describes how the OSI model applies to CougSatNet. As a club we aim to develop more than CougSat-1 ergo this interface is designed for future expandability.

2.1 Layer 7: Application

This layer is the closest to the user. It is the thing shown to the end user. The Ground's end user is a ground station operator. The end user on the satellite side are the sensors and end effectors in the various subsystems.

2.1.1 Ground Side

The Ground software shall present the information in a graphical user interface (GUI). With relative ease and speed, a ground station operator will understand the status of the satellite on the other end of the CougSatNet.

The real value numbers (degrees Celsius, amperes, etc.) will be visible as plain text. Graphical elements will enhance the experience with graphs, maps, etc. that include a color scale or acceptable ranges. These ranges are unique to each parameter being measured.

2.1.2 Satellite Side

Each subsystem has a plethora of sensors and end effectors that either measure or control real life parameters of the satellite. This includes, but not limited to:

- Temperature sensors
- Gyroscopes
- Voltmeters
- Ammeters
- Magnetometer
- Illuminance sensors
- Radionavigation-satellite service receivers
- Cameras
- Gas sensors
- Thermal heaters and knifes
- LEDs
- Magnetorquers
- Radio transmitters and receivers

2.2 Layer 6: Presentation

This layer is the data syntax layer. It transforms raw bits into presentable information. File types, such as PNG, are found on this layer. Files exchanged between the Ground and the satellite include, but not limited to:

- Telemetry containing files, see CougSatTelemetry (CST)

- Images, see CougSatGraphics (CSG)
- Binary images for processors, see CougSatBinaries (CSB)
- Command instructions, see CougSatCommand (CSC)
- Payload configuration files
- Log files

Both the Ground and Satellite, upon receiving one of these files, knows what GUI element or end effector to update. They are also knowledgeable on how to construct these files.

2.3 Layer 5: Session

This layer handles organizing a communication link. This layer also includes methods for securing a communication link to prevent unauthorized access. On the satellite, the C&DH transfers session messages to and from the Comms.

2.3.1 Session Message Format

Each session message begins with a header, then a file, and finished with a CRC, see Table 8.

Table 8: Session Message Format

Byte Offset	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x0000	Session Flags		Password [63:16]					
0x0008	Password [15:0]		File Length			File Name...		
0x0010	File...							
...								
End-4	CRC							

2.3.1.1 Session Flags

The most significant bit is the secure flag, a password is included when secure flag is set. When the secure flag is unset, the password is all zeros. The remaining $15b$ is the session ID number that is incremented each time a new session is created.

2.3.1.2 Password

Note this is only included if the secure flag is set in the Session Flags. If the flag is unset, this field is all zeros. The password is a $64b$ number that is generated using the SHA-256 algorithm[12]. A passphrase is hashed using SHA-256 then the lowest $64b$ are used as the password to validate the session. If the

[12] For more information, see Wikipedia's article on SHA-2

received password does not match the stored password, the session is aborted. If there is no password when the attached file requires authentication to process, the session is aborted.

Only the first file transferred in a session may require a password. Once a session is authenticated, all messages belonging to that session are authenticated, this is to reduce the number of transmissions of the secret password. The authentication for that session ID is lost once the session ends.

2.3.1.3 File Length

A $24b$ number represent the number of bytes the attached file contains. This leads to a maximum file size of $(2^{24} - 1)B = 16MiB$. This is around the limit for a $15min$ link window running at $500kbps$. If a larger file needs to transferred, then it should be broken into $16MiB$ chunks.

2.3.1.4 File Name

The name of the attached file is a null-terminated string with UTF-8 encoding[13]. The file name is limited to $256B$; if this limit is reached before a null character, the file name is invalid, and the session is aborted.

2.3.1.5 File

The attached file is appended to the session header without any modifications or transformations.

2.3.1.6 CRC

A 32 bit Cyclic-Redundancy Check is performed on the session header and data. Its checksum is appended to the session data. If the given checksum does not match the computed checksum, the session is aborted.

2.3.2 Control Files

There are certain file names that are used to control the session. The attached file has details on the control

2.3.2.1 REQUEST

This requests a file (or files) to be sent from the other end. The attached file contains a list of file paths, separated by newlines, to be sent if present. The example below requests for the newest payload file then the EPS's status report.

```
payload/newest
eps/status
```

2.3.2.2 CLOSE

This closes a session by invalidating the session ID's password, if provided, and incrementing the session ID counter. An explicit close should be given but

[13] For more information, see Wikipedia's article on UTF-8

the session will automatically close if a new one is started or after a reasonable timeout.

2.4 Layer 4: Transport

This layer coordinates data transfer between the Ground and the satellite with quality of service functions.

2.4.1 Segmentation

Large session messages need to be broken into smaller segments to reduce the segment error rate. A bit error rate of $10ppm$, the target rate of the hardware, results in out of $1kiB$ chunks, $(1-10ppm)^{1kiB*\frac{8b}{B}} = 92\%$ will have zero errors. $8kiB$ chucks, just 8 times longer, are 52% error free. Smaller chucks reduce the number of chunks that need to be retransmitted. The drawback is more byte overhead from the more headers which also requires more time to process.

CougSatNet's transport protocol segments session messages into $1kiB$ chunks. Each segment is given an ID that is used to reassembly the segments in the correct order.

2.4.2 Acknowledgement

When the sender wants to validate the receiver read a valid segment, it will send a segment with A set. The receiver then returns a list of segment IDs it successfully received as a Receipt Segment.

2.4.2.1 Acknowledge Timing

It is impractical to ask for acknowledgement on every segment as there is delay in switching from transmit to receive in the hardware. The throughput would be severely inhibited. Instead, CougSatNet utilizes selective acknowledgement on a block of segments.

The destination keeps a list of IDs of the most recent 256 segments successfully received. Every so often, up to 128 segments, the source will set the ACK flag in the next segment being transmitted. The destination, upon reading this flag, replies with a Receipt Segment containing that list of IDs. The source receives that receipt and figures which segments it needs to resend during Retransmission.

If the source does not receive a receipt within $100ms$, it transmits the same segment with acknowledge flag set again. After ten failed attempts at receiving a receipt, the source indicates the link has been lost.

Note that the source may decide to change its chunk length during transmission up to the limit of 128 segments to test the link quality before operating at full-length. A session message may also only be a single or couple segments long and not require a full-length chuck at all.

2.4.2.2 Receipt Segment

A Receipt is a segment with the Receipt flag set whose Session Data Payload contains a list of segment IDs that have been successfully received. The list is a concatenation of the 256 most recent Segment IDs. Since the IDs are $16b$, each $16b$ is a new ID.

2.4.3 Retransmission

When retransmitting the lost segments, the source will set the acknowledge flag in the last one and repeat the Acknowledgement process until all segments in the block have successfully been received. It then begins sending the next block.

2.4.4 Segment Format

Table 9: Segment Format

Bit Offset	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x0000	Segment ID [15:8]							
0x0008	Segment ID [7:0]							
0x0010	Session Message ID				Keep Alive	Last	ACK	Receipt
0x0018	Session Data Payload…							

2.4.4.1 Segment ID

This is a $14b$ number indicating which segment it is in the segmented session message. The first segment has an ID of $0x0000$. Each following segment increments this number for a total of $2^{16} = 65536$ segments per session message.

2.4.4.2 Session Message ID

This is a $4b$ indicating which session message it belongs to. It begins at zero and increments each session message. If a session message is not fully transferred before the link is lost and Keep Alive is set, both sides will reserve the current session message ID to resume later and skip over it when incrementing for session messages transferred in between.

2.4.4.3 Keep Alive

This flag indicates, when set, that both sides should reserve the session message ID in case the link terminates before the session message is fully transferred. When link is re-established, the sender can resume sending with the same reserved session message ID to resume transfer. The session message ID can be reserved, and thus resumed, for up to $24\ hours$ or until the satellite reboots for some reason.

2.4.4.4 Last

This flag indicates, when set, that the segment is the last segment for the session message ID being assembled. Upon processing this segment, the session message is completely transferred and assembled.

2.4.4.5 ACK

This flag indicates, when set, that the destination needs to acknowledge its received segments as described in Acknowledgement.

2.4.4.6 Receipt

This flag indicates, when set, that the segment data payload contains a receipt as described in Receipt Segment.

2.4.4.7 Session Data Payload

This is the data of the segment, appended to the header, without any modifications or transformations.

2.5 Layer 3: Network

This layer adds addressing to the segment such that the receiver of a packet knows who it came from and where it needs to go. For CougSat-1 this functionality will rarely be used. It is in place for future expansion for inter-satellite communication and for the Ground to know which CougSat it should process the packet for.

Table 10: Packet Addressing

Address	Description
0x00	Broadcast – All intended to receive
0x01	Ground
0x02	CougSat-1
0x03	Reserved
0x04	Reserved
0x05	Reserved
0x06	Reserved
0x07	Reserved

2.5.1 Packet Format

Table 11: Packet Format

Bit Offset	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x0000	Sender			Recipient			Length [9:8]	
0x0008	Length [7:0]							
0x0010	Packet Payload Data…							

2.5.1.1 Sender

This is a $3b$ number indicating the source of the packet, a list of addresses can be found in Table 10: Packet Addressing.

2.5.1.2 Recipient

This is a $3b$ number indicating the intended destination of the packet, a list of addresses can be found in Table 10: Packet Addressing.

2.5.1.3 Length

This is a $10b$ number representing the length of the packet payload data in bytes. This limits packet payloads to a length of $2^{10} = 1024B$.

2.5.1.4 Packet Payload Data

This is the data of the packet, appended to the header, without any modifications or transformations.

2.6 Layer 2: Data Link

This layer adds error avoidance and detection such that a corrupted frame is unlikely to occur and should it occur, be detected and disposed of.

2.6.1 Synchronization

A preamble, or special set of symbols, is sent before the frame's payload to allow the destination's receiver to synchronize with the incoming symbol stream. The first set of symbols is a fixed pattern of sufficient time that the receiver measures the transition rate and adjusts its algorithms to synchronize with the stream. The last symbol of the preamble is unique from the others to indicate the start of frame. The receiver cannot count how many symbols of the preamble it got as it may have missed some while tuning itself so a start frame symbol allows a synchronized receiver to know when the symbols belong the body of the frame.

2.6.2 Encoding

A frame is encoded using 8b/10b Encoding. This provides sufficient data transmissions for the destination's receiver to recover a clock from the data stream. It also maintains long term DC bias of zero, that is there is the same amount of $0s$ and $1s$ in a data stream. Most RF hardware is designed to block DC

between stages So having a non-zero DC bias in the signal could result in bit errors.

2.6.3 End of Frame

An end of frame marker, a special set of symbols, indicates that the end of the frame has been reached an no more symbols belong to that frame.

2.6.4 Frame Format

Table 12: Frame Format

Code Offset	0x00	0x01	0x02	0x03	0x04	0x05	0x06	0x07
0x0000	Preamble							Start of Frame
0x0008	Frame Payload Data							
…								
EOF-5	CRC		End of Frame	Inter-Frame Delay [11:7]				
EOF+3	Inter-Frame Delay [6:0]							

2.6.4.1 Preamble

The preamble consists of 7 *codes* of Control Codes K.28.5 beginning with running disparity of −1.

2.6.4.2 Start of Frame

The start of the frame is indicated by the Control Codes K.23.7. Immediately following this code is the payload data.

2.6.4.3 Frame Payload Data

The payload data is encoded into the appropriate codes then appended to the start of frame code. This data can be up to $1026B$ long.

2.6.4.4 CRC

A 16 bit CRC is performed on the payload data (before encoding). The resultant checksum is encoded into the appropriate codes then appended to the frame payload data.

2.6.4.5 End of Frame

The end of the frame is indicated by the Control Codes K.27.7. Immediately preceding this code is the last code of the CRC.

2.6.4.6 Inter-Frame Delay

The minimum dead time between frames is the time required to send $12\ codes$. This delay allows either side to finish processing the frame and prepare for the next one.

2.7 Layer 1: Physical

The symbols differ depending on the mode of transmission. The high gain antenna will always use Quadrature Phase-Shift Keying (QPSK) modulation. The low gain antenna will use Data Over Voice (DOV) for beacon transmission, with integrated telemetry, and QPSK for command and control messages.

The QPSK modulation uses the in-phase component as the first bit of the symbol and the quadrature component as the second as pictured in Figure 3: QPSK Modulation Constellation Diagram. Each symbol has two bits.

The DOV modulate uses a $20kHz$ tone as a zero symbol and a $30kHz$ tone as a one signal. Each symbol has one bit. The phase is matched when switching between the ones to eliminate discontinuities and thus reduce sideband power.

2.8 Layer 0: Medium

Since the satellite is $400km$ into space and moving very fast, copper wires don't work well. The information is a radio frequency electromagnetic wave. The carrier frequency is $\approx 435MHz$ for the low gain antenna and $\approx 1.25GHz$ for the high gain antenna. The low gain antenna is linearly polarized for simplicity and thus reliability. The high gain antenna is circularly polarized for low cross-polarization loss.

Appendix A: QPSK MATLAB Code

This software modulates a QPSK signal then demodulates it. It incorporates clock jitter on the transmitter and receiver, adds random gaussian noise, and demodulator carrier offset (frequency and phase). Its function can be illustrated from the constellation diagram, Figure 4, the connected line plot is the entire signal whilst the scatter plot is just the signal at the sampling period (middle of each bit). Note how the errors introduced propagate to the signal from the Software Defined Radio (SDR) as a circle due to the frequency shift continuously rotating the signal. After the Costas Loop, the errors are removed, and the diagram becomes a square.

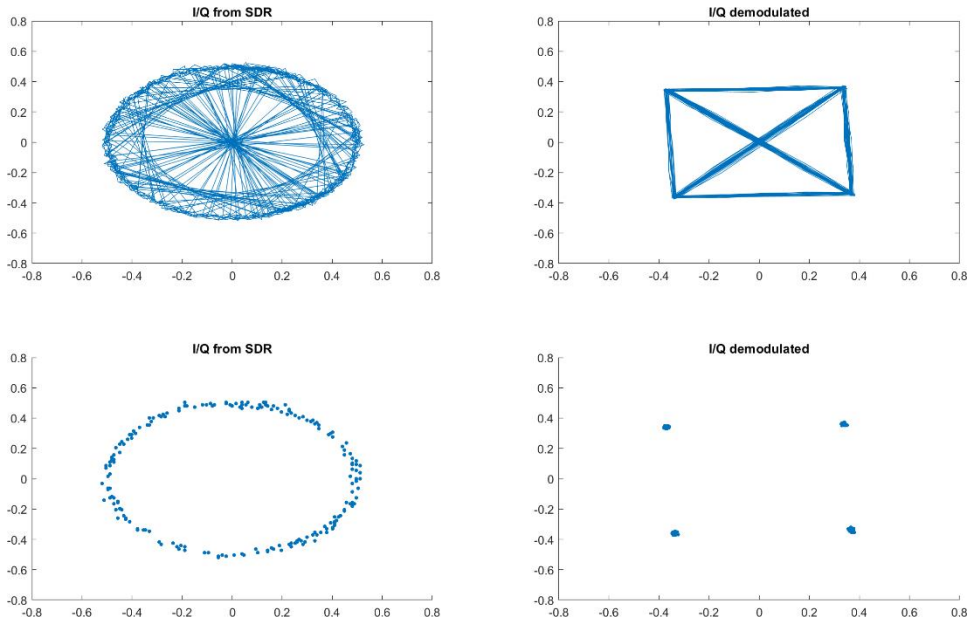


Figure 4: QPSK MATLAB Modulator and Demodulator

```
%% Reset the current setup
clc;
clear all;
close all;

%%%%% Generate RF signal from random data %%%%%
%% Generate random data to transmit
% Data is -1 or 1
numSym = 200;
dataI  = randi([0 1], numSym, 1) * 2 - 1;
dataQ  = randi([0 1], numSym, 1) * 2 - 1;

%% Modulation defines
baud      = 250000;          % 250kHz
carrierF  = 435000000;       % 435MHz
RFSampleF = carrierF * 100; % 43.5GHz
duration  = numSym / baud;
RFSamples = round(duration * RFSampleF);

%% Generate baseband signals and filter
```

```
% Output is a trapezoidal wave to reduce high frequency harmonics
baseBandI     = zeros(1, RFSamples);
baseBandQ     = zeros(1, RFSamples);
trapezoidF    = 2000000; % 2MHz
trapezoidRate = 2 * trapezoidF / RFSampleF;
for i = 2 : RFSamples
  t = i / RFSampleF;

  baseBandI(i) = baseBandI(i - 1) + trapezoidRate * dataI(ceil(t * baud));
  baseBandQ(i) = baseBandQ(i - 1) + trapezoidRate * dataQ(ceil(t * baud));

  baseBandI(i) = max(-1, min(1, baseBandI(i)));
  baseBandQ(i) = max(-1, min(1, baseBandQ(i)));
end

%% Mix baseband with carrier
modPhaseJitter = 0.01; % jitters 1% of 2pi
modPhaseOffset = 0.3;
signalTX       = zeros(1, RFSamples);
for i = 1 : RFSamples
  t = i / RFSampleF;

  phi = modPhaseOffset + modPhaseJitter * randn() * 2 * pi;

  yI = baseBandI(i) * cos(2 * pi * carrierF * t + phi) * sqrt(2) / 2;
  yQ = baseBandQ(i) * sin(2 * pi * carrierF * t + phi) * sqrt(2) / 2;

  signalTX(i) = yI + yQ;
end

%%%%% Data is transmitted %%%%%
%% Add noise
% randn is a gaussian distribution with std. dev. of 1
snr = 10; % link budget calculated >10dB snr
for i = 1 : RFSamples
    signalTX(i) = signalTX(i) + 10 ^ (-snr / 10) * randn();
end

%% SDR demodulates into I/Q %%%%%
% doppler effects at 7.6km/s = 25ppm, clock stability = 5ppm
demodCarrierF    = carrierF * (1 + 30e-6); % 30ppm off
demodPhaseOffset = 0.4;
demodPhaseJitter = 0.01; % jitters 1% of 2pi
disp("Carrier offset: " + (demodCarrierF - carrierF));

signalRXI = zeros(1, RFSamples);
signalRXQ = zeros(1, RFSamples);
for i = 1 : RFSamples
  t = i / RFSampleF;

  phi = demodPhaseOffset + demodPhaseJitter * randn() * 2 * pi;

  signalRXI(i) = signalTX(i) * cos(2 * pi * demodCarrierF * t + phi);
  signalRXQ(i) = signalTX(i) * sin(2 * pi * demodCarrierF * t + phi);
end

%% Low pass received signal to remove the USB frequencies
[filterB, filterA] = butter(6, carrierF / (RFSampleF / 2), "low");
signalRXI = filter(filterB, filterA, signalRXI);
signalRXQ = filter(filterB, filterA, signalRXQ);
```

```
%% Sample mixer output
% Round the samples to the nearest bit value, adds quantization noise
SDRSampleF = 2e6; % 2MHz
samples    = duration * SDRSampleF;
adcBits    = 7;
adcMax     = (2 ^ adcBits) - 1;
sdrRXI     = zeros(1, samples);
sdrRXQ     = zeros(1, samples);
for i = 1 : samples
  t = i * RFSampleF / SDRSampleF;
  sdrRXI(i) = round(signalRXI(t) * adcMax) / adcMax;
  sdrRXQ(i) = round(signalRXQ(t) * adcMax) / adcMax;
end

% Plot I/Q from the SDR, time domain
figure;
subplot(5, 1, 1);
plot(sdrRXI);
title("I from SDR")
ylim([-0.5, 0.5]);
subplot(5, 1, 2);
plot(sdrRXQ);
title("Q from SDR");
ylim([-0.5, 0.5]);

%% Remove carrier (freq and phase) offset using a costas loop
% Symbols may be rotated by 0, 90, 180, or 270. Removed with a known preamble
% Output is integrated (rolling average) to low pass the signal
freqOffset = demodCarrierF - carrierF;
phiOffset  = demodPhaseOffset;
costasI    = zeros(1, samples);
costasQ    = zeros(1, samples);
sdrRXIOut  = zeros(1, samples);
sdrRXQOut  = zeros(1, samples);

integrateSamples = round(SDRSampleF / baud / 2);
disp("Integrating " + integrateSamples + " samples");

phi   = zeros(1, samples);
error = zeros(1, samples);
for i = 1 : samples
  t = i / SDRSampleF;
  if i > 1
    % Take the cross product between symbol vector and ideal symbol vector
    error(i) = sign(costasI(i - 1)) * costasQ(i - 1) - sign(costasQ(i - 1)) * costasI(i - 1);
    error(i) = error(i) / sqrt(2) / sqrt(costasI(i - 1) ^ 2 + costasQ(i - 1) ^ 2);
    phi(i) = phi(i - 1) + error(i);
  end
  % Remove the offset by applying a rotation transformation
  costasI(i) = sdrRXI(i) * cos(phi(i)) + sdrRXQ(i) * sin(phi(i));
  costasQ(i) = sdrRXQ(i) * cos(phi(i)) - sdrRXI(i) * sin(phi(i));

  % Take the rolling average to low pass filter the output signal
  start = max(1, i - integrateSamples + 1);
  for j = start : i
    sdrRXIOut(i) = sdrRXIOut(i) + costasI(j);
    sdrRXQOut(i) = sdrRXQOut(i) + costasQ(j);
  end
end
```

```
sdrRXIOut = sdrRXIOut./ integrateSamples;
sdrRXQOut = sdrRXQOut./ integrateSamples;

% Plot the I/Q out, time domain
subplot(5, 1, 3);
plot(sdrRXIOut);
title("Demodulated I");
ylim([-0.5 0.5]);
subplot(5, 1, 4);
plot(sdrRXQOut);
title("Demodulated Q");
ylim([-0.5 0.5]);
subplot(5, 1, 5);
error = error./ (2 * pi / SDRSampleF);
plot(error);
title("Frequency Offset");

%% Sample I/Q every bit period
dataRXISDR = zeros(1, numSym);
dataRXQSDR = zeros(1, numSym);
dataRXI    = zeros(1, numSym);
dataRXQ    = zeros(1, numSym);
for i = 1 : numSym
  t = (i - 0.5) * SDRSampleF / baud;

  dataRXISDR(i) = sdrRXI(t);
  dataRXQSDR(i) = sdrRXQ(t);
  dataRXI(i)    = sdrRXIOut(t);
  dataRXQ(i)    = sdrRXQOut(t);
end

% Plot the I/Q in and out, phase domain
figure;
subplot(2,2,1);
plot(sdrRXI, sdrRXQ);
title("I/Q from SDR");
axis([-0.8 0.8 -0.8 0.8]);
subplot(2,2,3);
scatter(dataRXISDR, dataRXQSDR, 10, "filled");
axis([-0.8 0.8 -0.8 0.8]);
title("I/Q from SDR");
subplot(2,2,2);
plot(sdrRXIOut, sdrRXQOut);
title("I/Q demodulated");
axis([-0.8 0.8 -0.8 0.8]);
subplot(2,2,4);
scatter(dataRXI, dataRXQ, 10, "filled");
axis([-0.8 0.8 -0.8 0.8]);
title("I/Q demodulated");
```