CubeSat Ground Station: Development of a Simplex Link, and Power Budget Analysis

Jay Williams | KE2DZX
Cooper Union Satellite Initiative
The Cooper Union
New York, NY, USA
jay.williams@cooper.edu

***Abstract*—This paper summarizes the ongoing semester research and development of a satellite ground station in New York City. The accomplishments in the Fall 2024 semester are successful digital transmissions using an N200 software defined radio in the UHF Amateur band, to mimic a signal that an Arduino expects to hear. Further research was done to establish transmission power requirements, involving analysis of an antenna to determine gain and reflections. Further work is needed to complete the simplex communication link between the Arduino and the SDR.**

Keywords—Software Defined Radio (SDR), Ultra-High Frequency (UHF), Amateur Radio, Amateur Satellites, Simplex Links, Antennas, Power.

I. Introduction

The CubeSat program is becoming increasingly popular among colleges and universities as it gives the opportunity to do experimentation in space at a low cost. The Cooper Union's CubeSat team aims for the completion of one such satellite in the next couple of years. The ground station subgroup is responsible for researching and designing a communication link between Earth and a satellite, to transfer experimental data. This paper details the progress of the ground station subgroup, highlighting efforts for a simplex communication link, and understanding the power requirements to reach a satellite in low earth orbit (LEO).

The foundation of the ground station is an Ettus USRP N200 SDR for RF front-end signal processing, where further digital signal processing (DSP) was done with an accompanying laptop running the open-source software GnuRadio. It is also typical to include RF filters and amplifiers in the signal chain, to prevent the loss of very quiet signals to the noise floor.

Some digital transmission experiments were performed, using small Arduino UNO CC1101 tranceivers. While it is nice that they work for short distances, they must be modified to handle much more sensitive and quiet signals. For this reason, in the Spring 2024 semester, their transmission protocol was reverse engineered to work on the SDR, so more versatility and precision is possible, along with longer ranges. A premade driver was used to get started with text transmission, and to get reception working on the SDR, its protocol was reverse engineered to match the same techniques.

The transmitter uses a binary frequency shift keying modulation scheme, along with a clock synchronization and symbol synchronization word, followed by a byte detailing the length of the transmission, and then the actual data. To analyze the signal, filtering, demodulation, and synchronization were performed in GNU Radio, and bits were piped to a C program in real time to pack bits into bytes.

While time consuming, this reverse-engineering process proved to be extremely insightful on the way that SDRs and transmitters function, as well as the intricacies of wireless data transfer.

Fully understanding the signal then allowed us to configure the SDR to transmit that same signal. This was justified because the transceivers were originally designed to each other, and when we were "intercepting" its signal, we were hearing exactly what another transceiver would expect. Thus we mimicked the signal's format, sent it over the air, and the Arduino processed it and printed the expected text to the screen.

On the subject of transmitting, one must obtain a ham radio license to legally transmit over the air. We are restricted to certain frequencies, and must have proper etiquette when interference is detected. These licenses can be obtained through the ARRL (Amateur Radio Relay League) and require the passing of an exam. Several of the team members passed this exam, and callsigns were obtained to legally transmit. However, in the setting of the Cooper Union, any tests made indoors are very unlikely to reach the outside world, because the school's architecture makes it a giant Faraday cage.

II. Background

A. *Software Defined Radios (SDRs)*

SDRs are extremely useful devices, making entry into the world of RF extremely easy. They can be programmed to receive and transmit over a wide range of frequencies, depending on how much you are willing to spend. The benefit of having a radio be software-defined, is that all further signal processing can be done entirely digitally, leading to endless applications not limited by hardware. Without an SDR, one would have to design entire signal chain circuits depending on how the incoming signal is modulated. For example, cellphones have extremely advanced circuitry within them to be able to switch the type of demodulation depending on signal strength and bit error rate.

An SDR is basically a Field Programmable Gate Array (FPGA) with RF front-end components to amplify and filter a signal, down-convert it to baseband, and sample it to a digital form so that it can be digitally processed on an accompanying computer. Analog to digital conversion at a high sampling rate is an ongoing field of research, and as such makes top-of-the-line SDRs extremely expensive. The associated RF hardware is not cheap either, and having high-quality oscillators with such a large variation in frequency drives up the cost as well.

The SDR used is an Ettus USRP N200 SDR, modified with a UBX10 daughterboard to expand its bandwidth to 40 MHz with a 10 MHz to 6 GHz frequency range, arguably overkill for this application. However, such a large bandwidth can prove fruitful if spread-spectrum modulation techniques are employed in the future to overcome the noise floor. This large bandwidth was used to generate our binary frequency shift keying signal, because a sufficiently high sampling rate is necessary to cleanly modulate our data.

B. *Modulation and Demodulation*

A common household device for internet access is known as a modem, which stands for "modulator/demodulator". This device converts packets of data into a form suitable for transmission through a cable or over the air. This is an example of digital modulation, but there also exist analog modulation schemes, such as broadband FM radio that you may listen to while driving.

FM, or "frequency-modulated" signals are very simple. For the example of audio transmission, we can consider the bandwidth of an audible signal to be 20 Hz to 20 kHz. Thus we can frequency modulate audio to a carrier frequency, such as 100 MHz, so on the RF spectrum, we see a spike at 100 MHz surrounded by a blob that is 40 kHz wide (real signals are symmetric over 0 Hz, in this way we can say a 'negative' frequency is also shifted to 100 MHz). To shift the signal back down to baseband, we simply have to use the same 100 MHz carrier frequency to recover the original signal.

Of more interest to us, is digital modulation. Most schemes use symbol constellations, which are a set of points plotted on the complex plane to symbolize what bits convert to what symbols. Using the previous modem example, a constellation often used for wired internet connections and really strong wireless connections, is 256-QAM. This stands for "Quadrature Amplitude Modulation" and has 256 points densely packed together. Because each point, or 'symbol' is a complex number, it carries with it both amplitude and phase information of the carrier frequency, which can be used to represent bits. For 256-QAM, each symbol thus contains 8 bits of information, or 8 bits per symbol.

The transmitter-receiver designed as a proof-of-concept for the ground station uses 2-FSK modulation, standing for 2-frequency-shift-keying or binary FSK (BFSK). This scheme doesn't use a constellation, instead using a set of pre-determined frequencies centered around a carrier, similar to FM, however each frequency represents a symbol. For 2-FSK, there are only two frequencies, one for a 1, and another for a 0, meaning that this scheme has 1 bit per symbol.

C. *Noise and Bit Error Rate (BER)*

Noise is a random process that is a fact of life. It interferes with our signal, and when our signal is too weak, noise can overpower it to the point of undetectability. Analog modulation schemes are sensitive to noise, as there are few tricks to use besides proper amplification and filtering to extract the signal, all of which add noise into the signal. However digital modulation schemes can do much, much better. We can use coding techniques to detect and correct bit errors, and the type of modulation can also affect our bit error rate. The previous example of the 256-QAM constellation is incredibly fast, producing 8 bits per symbol, at the cost of a higher bit error rate due to symbols being packed close together. However 2-FSK, while slower, has symbols spread much further apart and are easily differentiated. For small amounts of data, 2-FSK is suitable for weak signals, such as those propagating from satellites.

Many CubeSats use FSK-CW, CW meaning continuous-wave, which is essentially Morse code. The NOAA weather satellites for example, use FM for picture transmission, causing many issues in noisy environments.

III. Experimentation

A. *Simulating in GnuRadio*

Before any transmissions were made, an entirely self-contained simulation of the modulation and demodulation process was created within GnuRadio. The underlying hypothesis for this simulation was to use the same demodulation process as was used for the Arduino to SDR communication link developed previously, with the assumption that the Arduino uses the same process.

To explain the simulation, a more in-depth explanation of frequency-shift-keying is warranted. Looking at the time-domain waveform is the best starting point for understanding the modulation scheme. Below is a time-domain representation of FSK, where one can see a sinusoid that changes frequency over time. The distinction is arbitrary, but for our purposes, a low frequency represents a '0' bit, and a high frequency represents a '1' bit.

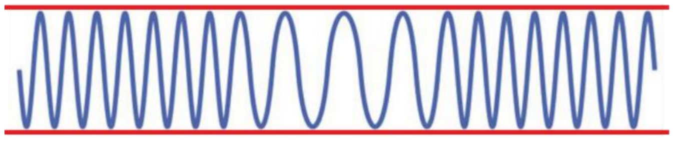


Figure 1: Time-domain representation of FSK.

The red lines on the top and bottom are to show that this scheme's amplitude does not change over time, and illustrates that this scheme only relies on deviations in frequency. To receive an FSK signal and demodulate it, one must know the carrier frequency f_c, which bits correspond to which frequency (from this, one can extrapolate the frequency deviation Δf), and the bandwidth of the signal (can be estimated as $2\Delta f$ or $4\Delta f$).

The most basic elements are the frequency deviation and carrier frequency, but bandwidth is a little more nuanced. For a digital signal, bandwidth is technically infinite. A digital signal can typically be thought of as a square wave, which when analyzed in the frequency domain, produces a spectrum over all frequencies. It is a question then, how could we receive a digital signal when hardware has a limited bandwidth and a digital signal is infinite! Thankfully for us, there is a simple theorem called the Nyquist-Shannon Sampling Theorem that tells us that with a bandwidth of only twice the data rate, we can perfectly reconstruct the original signal. In practice, however, it is preferable to use a larger bandwidth for cleaner data whenever possible.

So, using GnuRadio, we must convert our data into a frequency-shift-keyed signal. To do this, we first feed in bits from our file one by one. Most programs, including GnuRadio, will feed files byte by byte, but we want the individual bits, so a block is necessary to convert a byte to a bit. The next step is to feed these bits into mock voltage controlled oscillator (VCO). It is a software implementation of the real-life circuit, that produces a variable frequency depending on what voltage is applied to it. They way GnuRadio implements it, is that one must supply it with a floating-point value, and it will produce a sinusoid with a frequency of that value times the normalized frequency, i.e. the frequency it should produce when given 1 as the input. In GnuRadio, this is denoted as the 'sensitivity' and given in angular frequency. So, we would set the VCO's sensitivity to Δf, so that when a bit is 0, we get a sinusoid at 0 Hz, and when a bit is 1, we get a sinusoid at Δf Hz. At this point, we can say we have produced a frequency-shift-keyed signal, but we are far from done. In fact, keeping the approach this simple will not work in real life, and not even in simulation. Several other factors must be considered.

Firstly, it is entirely possible (and is true in our application) that our bitrate is higher than our frequency deviation. Thus if the bits are switching faster than our VCO can complete a full cycle, our output waveform will look crooked and jagged and nothing like what is seen in figure 1. Thus, we must drastically increase the 'sensitivity' of the VCO, while also keeping the shift in frequency the same as our original Δf. A quick solution is to set the sensitivity to a multiple N of Δf, but determining what multiple to use is somewhat arbitrary and was determined through trial and error. To keep the frequency deviation the same, we can no longer feed in 0s and 1s. These were converted to 1-1/N and 1, respectively. Thus we can get frequencies at $(N-1)\Delta f$ and $N\Delta f$, maintaining a separation of Δf.

We must remember, however, that we are working in a digital space and sampling rates must be considered. For GnuRadio to be able to produce these frequencies, our sampling rate must also be much higher. Currently, the way we feed in bits is the exact same as the sampling rate, meaning that raising the sampling rate will also increase our bitrate—something we don't want[1]. To prevent this, we *interpolate* our input bits by N—the multiplication factor. This means that instead of the previous 1 sample per bit, we get N samples per bit. At this point, the modulation simulation is working, and when passed to the same demodulator, the output bits are the exact same as the input bits.

B. *Verification of Working Hardware*

With the modulation simulation working, testing began to transmit over the air and attempt to receive it with the Arduino. Before we could transmit, however, an up-conversion frequency must be chosen. The drivers for the CC1101 expect a certain carrier frequency where the bits should appear. Because our VCO is set to $N\Delta f$, we tell the SDR to transmit at f_c-$N\Delta f$, so that our bits appear exactly at f_c. As expected, the first attempt was unsuccessful. The difficulty in fixing any errors is that there is not much feedback to be gained from the Arduino, and there were no more SDRs to test with. Last semester, we were able to directly analyze waveforms in the time domain, and see exactly what was going wrong. This time, with just the Arduino with premade drivers, there was nothing to look at. We expect to see text at the output of the Arduino, but if the signal is not transmitted properly, nothing will show up.

We verified that the CC1101 was working by transmitting to it using another Arduino, but verifying the SDR was transmitting was done differently. To do this, we used a handheld UV-5R radio tuned to our carrier frequency. Most radios have a built in "squelch" feature, which serves the purpose of muting noise when no signal is present. The UV-5R has a green light that tells the user when squelch is turned off, indicating that a signal is present. When tuned to our up-conversion frequency, nothing was audible, but the green light was on, showing that our SDR was indeed transmitting. However, while the SDR was transmitting our up-conversion frequency, there was nothing found at the actual carrier, where we expected data to be.

If the signal was being transmitted correctly, we would expect a signal at f_c-$N\Delta f$ and f_c. Because only one was heard, one problem could be a bandwidth limitation; the signal was getting cut off. This was not the only problem, but the others were discovered almost accidentally.

C. *Debugging*

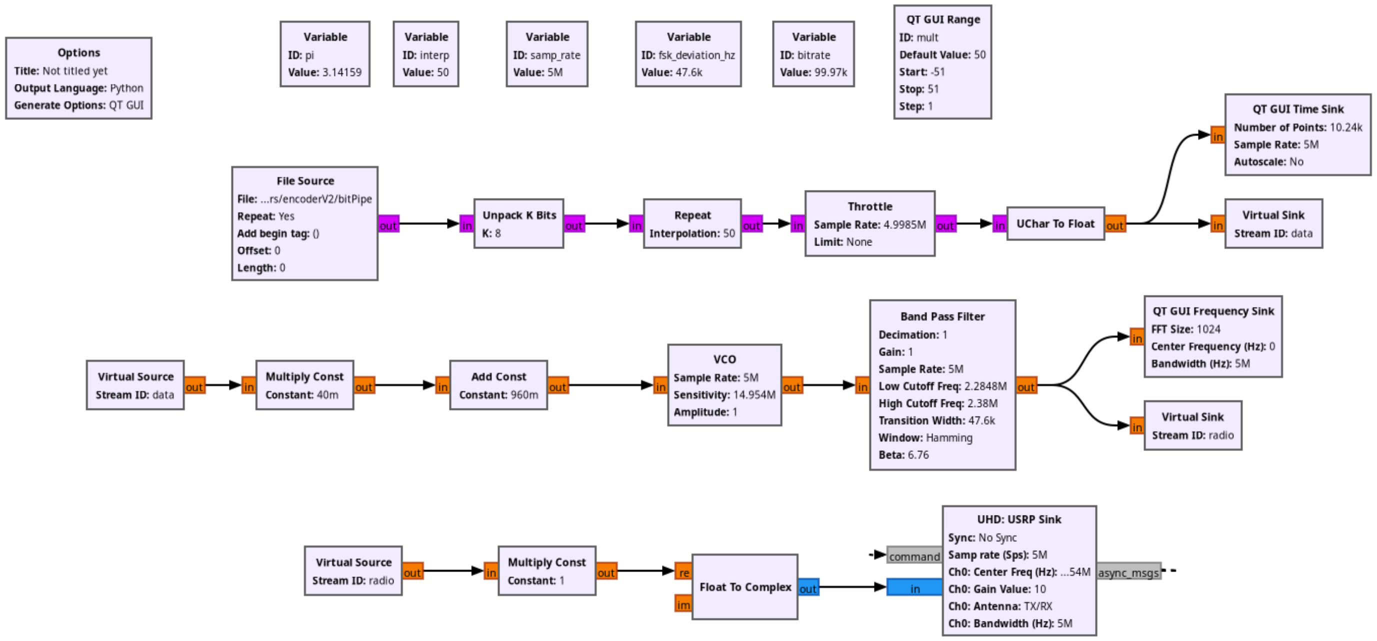
As expected from the handheld radio test, the bandwidth of the SDR's transmitter was set incorrectly. This was discovered during a demonstration for a prospective student, where while explaining the system, the numbers seemed wrong. It was much lower than necessary, hence the signal getting cut off. For proper transmission, we needed at least $2N\Delta f$, which is based off of the previously discussed $N\Delta f$, and doubled. The bandwidth being set so low was a complete accident, as it seemed like such an unlikely mistake to be set incorrectly. But once this was fixed, our data signal was observable through the handheld radio. However, still nothing from the Arduino.

During another demonstration, a student was shown the actual transmitted FSK signal from the CC1101, using GQRX. GQRX is a software that comes with a spectrum visualizer, so two peaks could be observed for the FSK signal. To show the meaning of frequency deviation, a measurement was made to show the distance between the two peaks. And it was observed to be exactly $2\Delta f$. During design, we expected frequency deviation to mean just Δf, but through direct observation we could see that it was not true. Implementing this into the system was not difficult. It merely means a change to what we feed into the VCO. To fix this, we feed in 1-2/N and 1 instead of 0 and 1. Thus yielding peaks at $(N-2)\Delta f$ and $N\Delta f$; a deviation of $2\Delta f$.

[1] Of course everyone wants a higher data rate, but the device (CC1101) we want to talk to <u>expects</u> a certain bitrate, and that must not be deviated from.

These two changes were enough for the CC1101 to pick up our signal and demodulate it into readable text. The complete GnuRadio flowgraph is visible below in figure 2.

then fed to the VCO, which has its sensitivity set to 2πNΔf, because this block wants angular frequency, which comes out to 2π*50*47.6 kHz = 14.954 Mrad/s, corresponding to 2.38 MHz.


Options
Title: Not titled yet
Output Language: Python
Generate Options: QT GUI

Variable
ID: pi
Value: 3.14159

Variable
ID: interp
Value: 50

Variable
ID: samp_rate
Value: 5M

Variable
ID: fsk_deviation_hz
Value: 47.6k

Variable
ID: bitrate
Value: 99.97k

QT GUI Range
ID: mult
Default Value: 50
Start: -51
Stop: 51
Step: 1

File Source
File: ...rs/encoderV2/bitPipe
Repeat: Yes
Add begin tag: ()
Offset: 0
Length: 0

Unpack K Bits
K: 8

Repeat
Interpolation: 50

Throttle
Sample Rate: 4.9985M
Limit: None

UChar To Float

QT GUI Time Sink
Number of Points: 10.24k
Sample Rate: 5M
Autoscale: No

Virtual Sink
Stream ID: data

Virtual Source
Stream ID: data

Multiply Const
Constant: 40m

Add Const
Constant: 960m

VCO
Sample Rate: 5M
Sensitivity: 14.954M
Amplitude: 1

Band Pass Filter
Decimation: 1
Gain: 1
Sample Rate: 5M
Low Cutoff Freq: 2.2848M
High Cutoff Freq: 2.38M
Transition Width: 47.6k
Window: Hamming
Beta: 6.76

QT GUI Frequency Sink
FFT Size: 1024
Center Frequency (Hz): 0
Bandwidth (Hz): 5M

Virtual Sink
Stream ID: radio

Virtual Source
Stream ID: radio

Multiply Const
Constant: 1

Float To Complex

UHD: USRP Sink
Sync: No Sync
Samp rate (Sps): 5M
Ch0: Center Freq (Hz): ...54M
Ch0: Gain Value: 10
Ch0: Antenna: TX/RX
Ch0: Bandwidth (Hz): 5M

Figure 2: Complete GnuRadio 2FSK Modulator Flowgraph

IV. Design

The description of the simulation was purely abstract, with no numbers attached for the purpose of being able to apply it to other situations. There is some redundancy in the above flowgraph, such as the variables 'interp' and 'mult' which are both set to 50 and serve the same purpose; the multiplication factor. The QT GUI blocks simply serve as visualization tools, and do not affect the signal processing chain. The virtual sinks and virtual sources are just so that the flowgraph can be read from left to right and top to bottom.

Starting at the file source, this is where data is fed in from a file byte by byte. It then passes through a byte unpacking block, so that the bytes are converted to individual bits. The bits are then 'interpolated' by our multiplication factor of 50. The next block, the 'Throttle' is unexpectedly important. Without this block, GnuRadio will attempt to feed in bits from the source file as fast as the computer can, driving up CPU use, and with an indeterminate sampling rate. We want to preserve the specified bitrate, so this is set to NR_b = 50*99.97kHz = 4.9985MHz. The bitrate of 99.97 kHz is specified by the CC1101 driver documentation, so that is what we followed.

Currently, 0s and 1s are stored in the 'unsigned character' datatype, and must be converted to floats so that they can be changed to their proper values for the VCO. The conversion to 1-2/N and 1 happens through the next two blocks, where the values are multiplied by 2/N and then 1-2/N is added. These are

The value of frequency deviation was also obtained from the CC1101 driver documentation. The bandpass filter simply serves to clean up our signal, so that not too much spectral content is transmitted, potentially annoying our neighbors. The signal is then fed to the USRP sink block, where the signal is transmitted through the SDR. The sink block is set to have its up-conversion frequency as f_c-NΔf, which for us means 433.92MHz-50*47.6kHz = 431.54 MHz. The SDR then translates our signal up to that frequency, meaning that our data will appear exactly at 433.92 MHz.

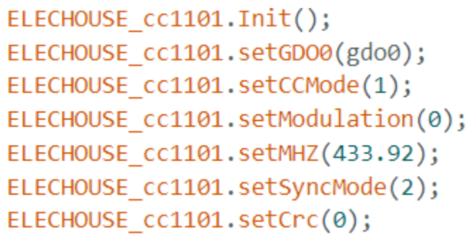
At several points in this flowgraph, one may notice that the sampling rate of some blocks is set to 5 MHz, and this was also chosen as the bandwidth of the SDR. This decision is purely to be larger than any other rate in the processing chain[2], and could be wider. The multiplication factor of 50 was chosen through experimentation in simulation. During testing, 25 seemed to be the minimum that the bits could be perfectly reconstructed, and 50 produced very clean results. By 'clean' we mean negligible timing jitter, a form of phase noise that causes bits to appear out of sync.

A. *Arduino UNO and CC1101 Driver*

The reception portion of the driver had several settings to change, like modulation type, sync word, and CRC (Cyclic Redundancy Check). Modulation was set to 2-FSK , as it is the simplest digital modulation scheme to deal with. The Sync Mode is important for proper sampling, as its purpose is to have a computer's clock period synchronize with the incoming signal's bit period, so that bits are sampled as best as possible.

[2] The up-conversion process is an analog system and is not limited by sampling rates.

The bit processing portion also uses this sync word to determine whether a bit change was noise, or an actual signal coming through. CRC (Cyclic Redundancy Check) was disabled, as the added complexity of bit error detection and correction can be dealt with at a later time.



```
ELECHOUSE_cc1101.Init();
ELECHOUSE_cc1101.setGDO0(gdo0);
ELECHOUSE_cc1101.setCCMode(1);
ELECHOUSE_cc1101.setModulation(0);
ELECHOUSE_cc1101.setMHZ(433.92);
ELECHOUSE_cc1101.setSyncMode(2);
ELECHOUSE_cc1101.setCrc(0);
```

Figure 3: CC1101 Receiver Settings

B. *Putting Together The Data*

As shown in figure 3, the sync mode is set to 2, which means that the receiver expects two synchronizing characters at the beginning of a transmission, as well as a brief clock signal. Using the empirically determined sync word characters from previous work, we were able to write some C code to serially feed bits into GnuRadio. The program had the sync words hard coded, and at the beginning of a transmission, would send characters that have the same ASCII code as an on/off clock cycle, the two synchronization characters, then a character indicating the number of characters in the string, followed by the data. This same figure was used in last semester's report, but it still applies here.

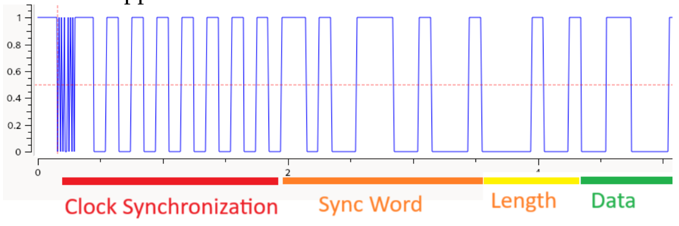


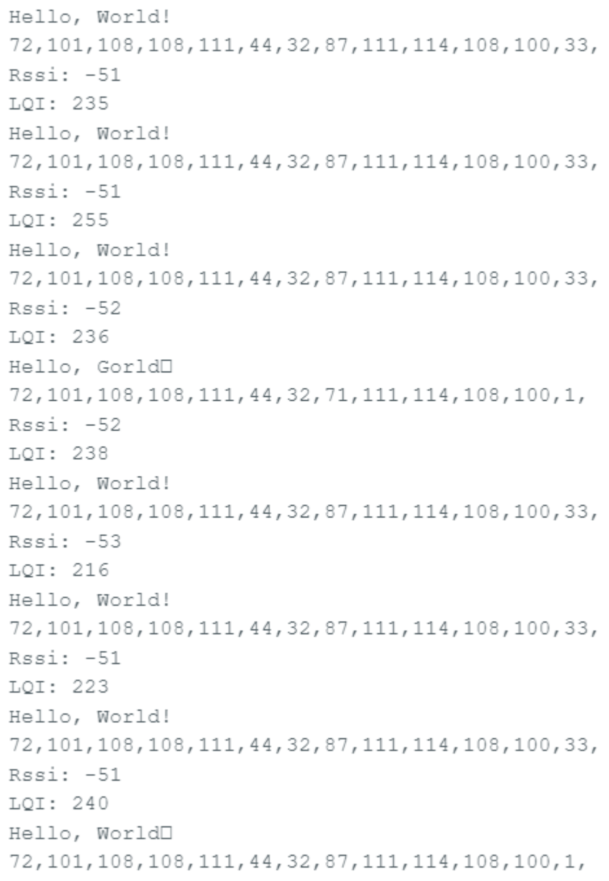
Figure 4: Dissected Signal

The actual characters used for this are not alphanumeric, and also extend past the signed character range (unsigned characters add an extra bit of range), so they cannot be directly displayed here. In integer form, the character values are: 170, 170, 211, 145, 13. We have not yet looked to see if the sync words are changeable, but if they are, perhaps using the ham radio callsign serves both the purposes of synchronization and identification of the control operator.

C. *Results*

As far as how the CC1101 demodulates the data, that is a black box. We analyzed its output and gave it what we think it expects, and through the design and debugging process, our text was displayed at the output of the Arduino.

The data input was just the "Hello, World!" string. The ASCII code for each character can be seen, as well as some other indicators such as Rssi and LQI. Rssi stands for received signal strength indicator, measured in dBm, a measurement of how much signal power the device received. LQI stands for link quality indicator, but that metric is of no use to us. LQI is used



```
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -51
LQI: 235
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -51
LQI: 255
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -52
LQI: 236
Hello, Gorld□
72,101,108,108,111,44,32,71,111,114,108,100,1,
Rssi: -52
LQI: 238
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -53
LQI: 216
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -51
LQI: 223
Hello, World!
72,101,108,108,111,44,32,87,111,114,108,100,33,
Rssi: -51
LQI: 240
Hello, World□
72,101,108,108,111,44,32,87,111,114,108,100,1,
```

Figure 5: Arduino Serial Monitor Output

for large networks with routing algorithms for the best quality path.

Despite the short link distance, there is considerable signal degradation. Looking at strings that were transmitted incorrectly, and comparing the individual characters to a perfect transmission, we can see that the characters differ by a single bit error. Single bit errors can be drastic in some cases, because for number representation in computers, a bit swap in the MSB will have much more impact than a bit swap in the LSB, hence the naming convention.

This could be caused by several factors, but an immediate discovery was that the receiving device's antenna touching the benchtop severely degraded signal quality. This was also observed when developing Arduino to SDR communication, as the signal looked much more noisy and jagged. The quick solution was to crumple up a paper towel and place the antenna on that, which greatly helped with bit error rate.

The other possibility is that the SDR's antenna is not sufficiently resonant at our transmit frequency of 433.92 MHz, and hence has a lot of reflections back into the port of the SDR meaning less power is transmitted. Perhaps it is worth proving this with the VNA, but these small antennas are very cheap and a more optimal one could be purchased.

V. Antenna Power Analysis

The large antenna we are using for satellite reception is the CLP5130-2N. It is a log-periodic antenna, with an alleged bandwidth of 105-1300 MHz, and a forward gain of 11-13 dBi with a maximum VSWR of 2:1. Its maximum power rating is 0.5 PEP/kW, i.e. PEPmax = 500W. Of these specifications, we will accept the datasheet's value of power rating and antenna gain, but the bandwidth and VSWR will be experimentally determined with a VNA. This was done last semester, but the experimental setup was poor (cables were too long) and the data was collected over too large of a bandwidth and thus precision was lost. The next test will use only the 420-450 MHz bandwidth for amateur satellites.

The antenna gain is the direction of maximum power concentration. The antenna is a passive device, meaning that no power is added to the system. The actual gain comes from the physical structure of the antenna taking advantage of constructive and destructive interference of radio waves to concentrate power in a single direction, while reducing power in others, so conservation of energy is preserved.

Peak envelope power (PEP) is the peak power of the envelope of the signal. For our application, where we use FSK, the PEP is very easily calculated because RMS power is independent of frequency, and FSK is not amplitude-encoded, so the envelope is constant. Looking back at figure 1, the red line shows the constant envelope.

Using the datasheet for the daughterboard of our SDR, we find that in our frequency range, the radio can transmit a maximum power of 20 dBm. We will label this as P_t, our transmit power. The wavelength of our signal is a simple calculation, simply done with c/f_c, yielding roughly 0.7m. We will also somewhat arbitrarily say that the receiver would like a minimum signal power P_r = -90 dBm. This can later be more accurately determined using some background noise temperature analysis, and a minimum SNR depending on the receiver architecture. The distance of the signal will be taken as the lower end of LEO altitudes, using R = 500 km. These values have set us up to use Friis Radio Link Formula:

$$P_r = \frac{G_t G_r \lambda^2}{(4\pi R)^2} P_t$$

Plugging our values in and solving for G_tG_r, the gain of the signal chain, and factoring out the 13dBi antenna gain, yields a gain requirement of 16.18 dB. To play it safe, we can transmit much less power from the SDR, suppose 10dBm, yielding a gain requirement of 26.18 dBm. The net change in power, however, stays the same at 36.18 dBm = 4.15 W. These calculations were made using the help of Pozar's Microwaves [3].

This calculation is very ideal. It does not factor in antenna reflections, indicated by VSWR, which can severely reduce how much power is sent over the air versus what was sent to the antenna. If the VSWR is bad enough, power can be reflected back into our signal chain and damage components. For this, a device called a circulator may be necessary that can redirect reflected power into a load mounted on a heatsink.

Power amplifiers on the market can achieve this level of gain and power output, and it is also worth nothing that the satellite's receiver will also have gain blocks, so not everything has to be done on the transmitter side. In fact, it is better that not everything is done there, because too much power output in one stage can push amplifiers intermodulation distortion very high, which is bad for signal quality.

VI. Conclusion and Future Work

A. *Data Transmission and "Simplex" Links*

The C program discussed previously for feeding data into GnuRadio is very inefficient. It repeats every 1 second, but in between it transmits a series of null characters, instead of nothing. Transmitting a null character is an extreme waste of power and bandwidth. To avoid this, it is possible to create an out-of-tree (OOT) block within GnuRadio, and there is official documentation on how to do this [4]. The design in mind is to simply copy the source code of the File Source block and add a 'sleep' parameter with a defined time, so that the block turns off periodically.

This is also important for establishing a simplex communication link. A simplex link is one in which transmission and reception is done on the same frequency, but the radios switch between the two to prevent interference. The hardware of the SDR automatically performs this switch. It does this by declaring that transmission has priority, so the SDR can be left in receiving mode, and whenever a transmission needs to be made, it will switch.

B. *RF Signal Chains and Microwave PCBs*

Just as important as software is the hardware. Due to the high frequencies, some microwaves theory may have to be implemented to properly impedance match antennas and other components on a PCB for the satellite. For the ground, larger components and more power consumption is allowed because available space is effectively infinite. However, for a satellite, size, weight, and power must be very strictly considered.

The goal is to work on a whole RF signal chain for the satellite, based on minimum SNR that can be experimentally determined from CC1101. Unless we would like to work with a better transceiver, it would be possible to directly attach gain and filter elements to the antenna port of the CC1101 and design around that. A PCB is required due to the small size of the CubeSat, which will be a large learning curve but very useful for the team in their future careers.

C. *Other Conclusions and Future Work*

Getting pictures from NOAA satellites would be nice as a proof of working equipment, but more desirable would be getting callsigns from digitally transmitting CubeSats, as it fits better with the scope of the project. However this is more difficult, as they are not as well documented in terms of modulation parameters, such as frequency deviation for an FSK modulated signal. In the short timeframe that a satellite pass occurs, such decisions cannot be made on the fly.

More experimentation should be done with hardware. There is plenty to be researched in the field of communication electronics to be applied for proper RF front-end circuitry to get the cleanest signals possible. However the expensive nature of these products and their often long time to deliver is cause for concern, as well as the electrostatic sensitivity of such devices in an outside environment, also leading to handling causing devices to break.

The noise floor of NYC is also concerning. If satellite reception proves to be exceptionally difficult because of this, there are techniques to overcome this. An example is GPS. GPS satellites manage to hide signals below the noise floor, and spread the bandwidth of the information extremely wide, to prevent jamming. This technique is known spread spectrum modulation, which can be done multiple ways. Such a technique is extremely involved both electrically and mathematically so it can be done as a last resort.

Having a working mock communications setup is a great place to continue working from. It is a short leap to integrate this with the e-paper part of the experiment, where a wireless picture transmission is possible to display. Getting started on the transmitter architecture of the ground station is also important, as there are a whole other set of design considerations that must be made, as well as legal with regards to licensing. Many members of the team will be working to get ham radio certified, ensuring a comfortable future of the project.

References

[1] M. Bernardi. NOAA-APT. https://noaa-apt.mbernardi.com.ar/ (April 2nd 2024)

[2] W. Shen. SmartRC CC1101 Driver Lib. https://github.com/LSatan/SmartRC-CC1101-Driver-Lib (April 5th 2024)

[3] D. M. Pozar, "Chapter 14: Noise and Active RF Components," in Microwave Engineering, 4th ed. Hoboken, NJ: Wiley, 2011, pp. 658-708.

[4] "Creating C++ OOT with gr-modtool," GNU Radio Wiki. [Online]. Available: https://wiki.gnuradio.org/index.php?title=Creating_C%2B%2B_OOT_with_gr-modtool. [Accessed: Dec. 19, 2024].