Project Belka

Authors: John Paul Harriman, Mia Han, Samuel Adams
Electrical and Computer Engineering, Carnegie Mellon University

***Abstract*—The initial intention behind Project Belka was to create a system capable of verifying data integrity on both serial and WiFi data transmission protocols that would be used on the Cube Rover. However, because of the current circumstances, Project Belka has become a study of various error correcting and detecting algorithms. Various protocols are simulated using a GUI to evaluate which protocol yields the best performance while also meeting the requirements and constraints set by the current design of the Cube Rover.**

***Index Terms*—Bit Erasures, Bit Flips, Carnegie Mellon University, Computer Architecture, Convolutional Encoding, CubeRover, Data Correction, Data Error Detection, Data Integrity, Data Manipulation, Data Packet, Embedded Programming, Field Programmable Gate Array, Finite State Machine, General Purpose In-Out, Graphical User Interface, Hamming Encoding, Inter-Integrated Circuit, Microcontroller, Moon Radiation, Moon Rover, Protocol, Protocol Wrapper,Quad Serial Peripheral Interface, Transmission Control Protocol, Turbo Coding, Universal Asynchronous Receiver-Transmitter, User Datagram Protocol, Viterbi, Wifi**

1 INTRODUCTION

Project Belka is a collaboration with Carnegie Mellon's Cube Rover team to determine error correcting and detecting protocols that best fit Project Belka's constraints and achieves the highest recovery rate. Currently, users of CubeRover have no way of checking the correctness of the collected data being sent to earth from the rover, besides a shallow checksum being sent at the tail of packets; the data may be corrupted by external extremities or harsh moon conditions. Because the data may get corrupted, creating and implementing error correcting and detecting protocols is necessary to ensure the data is transferred successfully without any corruption.

To solve this issue, we implemented various verification protocols for UART/I2C, UDP, Wifi and QSPI and tested each protocol using random data and erroneous data generation. Each implementation adheres to the requirements set by the current design of the Project Belka and the hardware requirements determined by the capacity of various Cube Rover components (TI Hercules, MSP 430, STM32 and FPGA). By exploring various error correcting methods, we have determined the best fit protocols that CubeRover should use that meet their requirements and achieve the highest recovery rate.

2 DESIGN REQUIREMENTS

The specifications for each component we are working with is listed under Table 1 in Appendix A. These are our motivational components from the CubeRover team and serve as our main constraints. The pervasive idea is that our solution approach be lightweight under conditions where timing and power need to minimized as much as possible.

Component Name	Function	Number of Instructions
Ti Hercules	Main Microcontroller	180,000,000/Second
MSP430	Motor Control/Monitoring	8,000,000/Second
Cyclone 10 LP FPGA	Camera Connection	100,000,000/Second

Figure 1: Number of instruction granted by clock frequency per component

Based on the internal clocks for our components, for each protocol we now have an upper limit on the number of instruction we can perform within this second. Luckily, none of the data sent over serial communication will be a larger size than the transfer speed of the protocol. The MSP430 has a significantly small number of instructions that we are able to execute for error correction/detection. This has limited the number of approaches that we can take, this will be explain more in the rationale part of this document.Cyclone 10 LP FPGA has a maximum of 100 MHz clock speed, and CubeRover will be using SPI and GPIO to transfer a stream of bits with a minimum length of 8 bytes; the maximum number of bits is 2^{31} since the maximum file size that will be transferred is 2^{31} bits. Thus, given these restrictions, we are limited to the range of values to set for code rate and constraint length due to the ranging number of instructions that will vary depending on the variable values. To stay within these constraints, our convolutional codes and viterbi algorithm must use no more than 100,000,000 instructions, which will dictate the potential combinations of constraint lengths and code rates to consider due to the need of wanting to stay lightweight and low-power.

The WiFi design requires that we create a protocol with 100% packet recovery using UDP packets as specified by CubeRover. The problem with the current UDP solution is that there is no way to confirm the receipt of packets. This is the essential requirement that will determine the success of the protocol. We also only need to be communicating between two nodes as CubeRover only requires a WiFi connection between the rover and the lander. Another requirement was the transmission of pictures relatively quickly. Since CubeRover is using a MT9P031 camera sensor the resolution is around 5.04 Mega Pixels with 12 bits of Analog Data Conversion (ADC). Doing some computation with the 100% JPEG compression at 12 bits per

pixel, each image is around 501KB [3]. Knowing this we settled on 20Mbps as it is a very reasonable throughput and will be more than enough for the data transfer that CubeRover needs for compressed JPEG pictures.

The serial cases of UART and I2C only have to deal with small data packets, but we wish for these instructions to be as fast as possible as to not disrupt the flow of data being sent on either communication call. Since we are only interacting with at max 1 byte per data frame for both protocols, we can use a lightweight encoding algorithm and still maintaining a high recovery rate. We are shooting for 75% correction and 100% recovery for these two protocols while not breaking the underlying protocol.

SPI and GPIO are mostly either images between the FPGA and the primary MCU or doing checks on the primary MCU to make sure that everything is functioning properly. Because both are transmitting variable amounts of data (discussed in section 4), the percentage of data recovered should be the highest value achievable/the maximum value. It's difficult to set a hard value for the percentage of data recovered, since higher recovery rates may be associated with smaller/larger data streams. Additionally, since each of the protocols that are implemented vary with the number and types of parameters that are set, it's difficult to pinpoint an accuracy rate because it's difficult to hold all of the protocols to the same standard when they aren't inherently the same. Therefore, the goal for the optimal protocol is to achieve the highest percentage of data recovery and correction.

3 ARCHITECTURE OVERVIEW

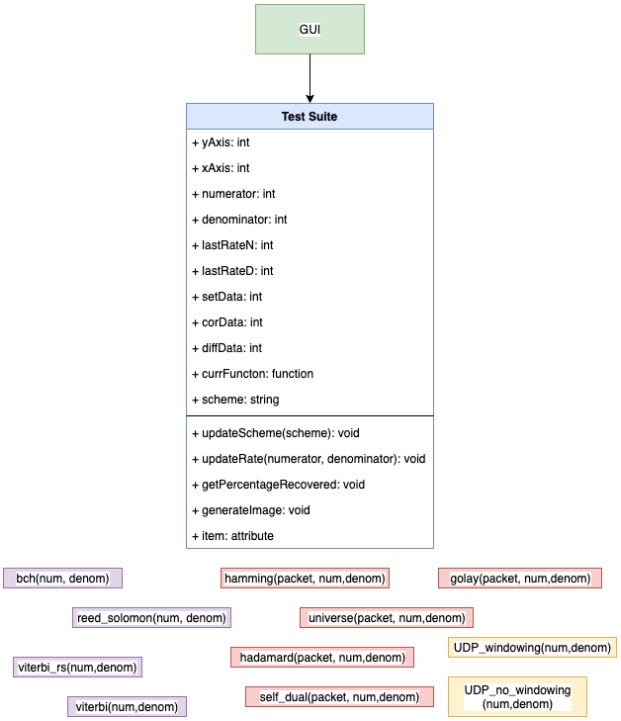


Figure 2: A diagram of the data structure that controls protocol changes, stores the results, and updates the changes in the fraction of packet that is fuzzed. The GUI contains a TestSuite object, and the test suite calls the functions in purple/red/yellow to simulate each different protocol.

3.1 Design Modifications

Our overall architecture remained relatively the same, but had to be fully integrated into a software solution. Our original architecture looked like the system in figure 3.

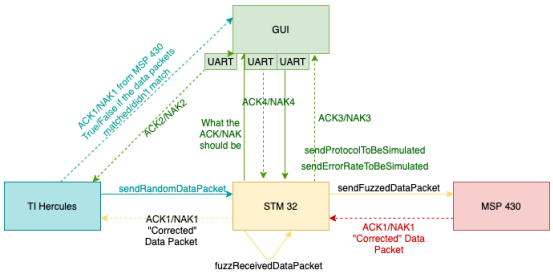


Figure 3: A diagram of the initial architecture of our project during the design phase

There was no problem with the architecture, but we decided on software due to the unforeseen circumstances. We moved the simulation of of the TI Hercules, intermediate micro-controller for fuzzing and verification, and MSP 430 into the GUI and maintained the same commands that would have been sent between the different hardware components.

3.2 Current Implementation

Our GUI manipulates a TestSuite object which serves as the emulation between of the hardware components and intermediate data manipulation. The pervasive idea was to keep the same functionality as originally intended, which behaved in four sequential steps.

First was defining the packet generation for each specific protocol implementation, e.g. UART would need an 8-bit randomly generated packet. Then we would pass the generated packet into our encoding function based off of which the protocol implemented. The encoded packet would then go through the "fuzzing" process which would flip the bits based off the user-defined fuzzing rate. After we got back the "fuzzed" packet, we would then pass it into the decoding function to see if it would be able to return a corrected data packet that matched our originally generated one. In this step we would also generate our data visualization. For UART as an example this visualization would be based on a grey-coded pixel line based off of the 0-255 values (for 8 bits). If our data was able to correct itself fully, then the line would display as white, and then the packet and corrected packet would be represented on two more lines. The percentage of correct corrected packets generated would be served as a percentage correct that we would pass back to the GUI to update the graph.

The GUI is later described in section 5.4, but in order to keep everything updating concurrently we needed to use threading. Our main thread serves as the handler to interactions that the user inputs. This includes the rate updates, scheme changes, and starting and stopping simulation. The other thread serves as the updating thread. While simulation is occurring, the thread will generate a percentage based off the TestSuite and update the graph. The generating thread also emits a signal to update the picture that's generated within the TestSuite object to the main thread. The main thread will catch the signal and repaint the image with the new image. The pass back and forth must be done under the constraint of the library as only the main thread is able to make these updates. Once the user has decided to stop simulation, the main thread emits a signal to the data generating signal to end it's execution and return to be joined.

4 PROTOCOL DESCRIPTIONS

4.1 UART and I2C

For UART and I2C we mainly covered 5 different algorithms. However, three of these algorithms were able to emulated through the use of another algorithm, Reed-Muller. By tuning parameters for this algorithm, we are able to successfully emulate others such as Universe, Self-Dual, and Punctured Hadamard Codes.

A couple of definitions to understand the terminology better is that we will call the block length n (the encoded message size), message length k (the size of data we want to send), and distance d (a measure of how far away the error can be).

4.1.1 Hamming

The version that we implemented for Hamming Code was an 8,4 model, meaning that we used regular Hamming(7,4), but added an extra parity bit for extra error detection of 1 bit. This gives us n = 8, k = 4, d = 3.

The main idea of Hamming codes is that it employs a generator matrix, a syndrome matrix, and a decoding matrix.

The 4 bits that get passed into the encoding function are multiplied by the generator matrix to produce an 8 bit encoded packet. When the data is fuzzed and passed into the decoding function, it first is multiplied by a syndrome matrix to see if it can correct the bit. It's also double checked with an added parity bit to see if the data has been corrupted. The syndrome will give a value that points to the exact error to fix.

4.1.2 Golay

Golay had values of n = 24, k = 12, d = 8.

We were able to accomplish this by generating three packets at a time and sending it to get a whole number sent. So the total k value would be 24.

The way that we managed to implement this is by relying on the math behind the code itself. We generated the matrix that allowed for encoding and decoding and kept it as a static variable that we could access anytime we wanted to encode or decode. This helped reduce the computation time. This is a sizable matrix though and could be dramatically sped up with custom hardware to do matrix operations.

4.1.3 Universe

Universe codes have values of n = 8, k = 8, and d = 1

Universe codes are binary linear code that is the set of all binary n-tuples. This algorithm was achieved by tuning the Reed-Muller with R,M values = 3,3.

4.1.4 Self-Dual

Self-Dual codes have values of n = 32, k = 16, and d = 8

An encoding algorithm is called self-dual when it is both a repetition code and an SPC code. This algorithm was achieved by tuning the Reed-Muller with R,M values = 2,5.

Since the k value for Self-Dual was 16 we needed to generate two different packets to be able to maintain the consistency.

4.1.5 Hadamard

Hadamard codes have values of n = 128, k = 8, and d = 64

Hadamard codes were used to transmit photos of Mars and the results will later show why it was used. There are multiple ways to implement Hadamard codes, but we used the Reed-Muller for this one as well to maintain the consistency. We were able to achieve this by tuning the R,M values = 1,7

4.2 WIFI

In order to perform the error correction and data verification that we needed on the UDP protocol we implemented 2 different types of TCP protocol using the UDP interface. The first protocol that we implemented was a simple TCP protocol that didn't use windowing. This protocol was very simple but ensured packet delivery by using acknowledgement (ACK) packets in order to confirm successful delivery of packets to their destination.

The packets were defined using the standard UDP packet structure FIGURE, source port, destination port, length and checksum. The rest of the fields were stored in the "data body" of the UDP packet. The TCP headers that were included were the sequence number, the acknowledgement number, the window size, and the flags (DATA, ACK). The FIGURE shows the breakdown of UDP (blue) and TCP (red) headers.

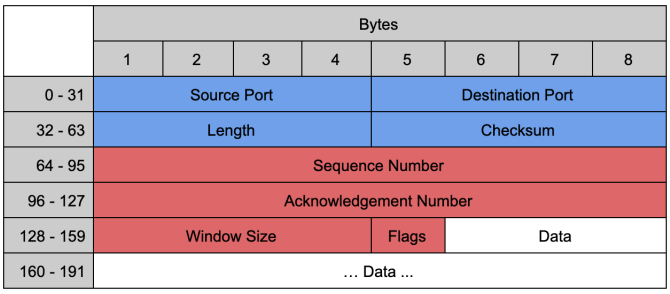


Figure 4: Packet structure for UDP

As for the protocol, the sender of any packet would first assemble the data packet by calculating the length of the data to be sent, assign sequence/acknowledgement numbers, set packet timeout (time spent waiting for an acknowledgement), assign proper flags and finally calculate checksum. This part of the protocol is defined as encoding" and it is what is being timed during our performance calculations. The packet will then be queued and sent to the destination. Once the packet is sent the sender will wait for a response from the receiver before sending another packet.

When received at the destination the packet is "decoded". The checksum is checked in order to ensure the data hasn't been corrupted and the data from the packet is extracted. This is called the "decode" stage. Once the packet is decoded, the receiver will send an acknowledgement (ACK) packet using the sequence number received to calculate the acknowledgement number of the packet. This packet also generates a checksum via the standard UDP protocol. This ACK packet is then queued and sent back to the original sender.

If the ACK packet reaches the original sender before the timeout is up the packet will be verified using the checksum field and will trigger the send of subsequent data. This protocol is good, it ensures packet delivery but it can only send one packet at a time as it waits for an acknowledgement for each packet before sending subsequent packets.

In order to try and improve the performance and test for alternative solutions we also implemented a TCP pseudo protocol that uses windowing. This protocol works very similarly to the first protocol, the only difference is that instead of sending and verifying one packet at a time, we send and verify a series of packets at a time. For example, if the window size was 10 the sender would send 10 packets in a row to the receiver and then would wait for an ACK to those 10 packets. The goal with windowing is to increase throughput by reducing the amount of time spent waiting for ACK packets.

4.3 QSPI and GPIO

For QSPI and GPIO, four different algorithms were explored and implemented to determine which one that best meets the constraints set by CubeRover's design while also best achieves the highest percentage of data bits recovered. By implementing BCH, Convolutional Codes and Viterbi, Reed Solomon, and Convolutional Codes and Viterbi + Reed Solomon, a wide variety of types of error correcting (Cyclic Error Correction, Forward Error Correction and Binary Convolutional Codes), an algorithm/type of algorithm was determine to be the best fit algorithm for CubeRover's QSPI and GPIO usage.

4.3.1 Convolutional Codes and Viterbi

The stream of bits is inputted into the convolutional encoder. The convolutional encoder provides knowledge of the possible data when moving onto the next stage; it will pass the stream of bits through a series of polynomials (XOR's) and shift registers. The polynomials reflect the convolutional encoder behavior and define the number of states in the convolutional encoder. The polynomials and shift registers combined create a state machine for input data, current state, output bit, and next state; the total number of states is defined as 2*(k-1), where k = the constraint length. Each message sequence is encoded into a code sequence. The next stage is the Viterbi stage, a graph that represents the dependencies between the current state and the next states of the encoder. At each node of the graph, transitions to the next node are used to show the change from one set of bits from the current node to another set of bits at the next node, essentially representing the state diagram. The transitions are determined by the input bits from the convolutional encoder. During these transitions, path metrics are calculated using a procedure called Add-Compare-Select, a calculation repeated at every encoder state. At each state, the previous two states of hamming distances are added to the current hamming distance at that state to create the new calculation. Once all the states for that transition are calculated, the calculations are compared, and the smallest calculations are selected while the rest of the paths are dropped.

To decode, an algorithm called maximum likelihood decoding is used. We will be implementing hard decisioning with viterbi decoding; hard decisioning at nodes only consider the hamming distances and selects the smallest hamming distance at each node. Soft decisioning processes the

stream of bits as voltage samples before digitizing them. Since we aren't using custom hardware to create the protocols with and are instead wrapping the protocols around SPI and GPIO implementations, hard decisioning is best fit for our design.

Once the state transition graph is completed, the algorithm will back trace through the graph to determine the nodes which have the smallest saved hamming distance. Moving through each chosen transition will output a sequence of bits. Using the equation r = c xor e, where r is the received stream of bits, c, the original inputted stream of bits, can be determined by tracing back through all of the chosen transitions and matching them to the appropriate outputted bits in the state diagram from the encoding part. Thus, by retrieving the original sequence of bits and comparing those bits to the outputted sequence of bits, the point(s) of error will be determined.

Because we didn't integrate the algorithms with the hardware, I used hard-decisioning for the Viterbi decoding. Soft-decisioning uses the voltage samples in addition to the data to determine the recover the sequence; hard-decisioning decodes the message only using the bit sequence and no other infromation from the receiver's sampling and demapper [**viterbi˙hard˙soft**].

4.3.2 Reed Solomon

Reed Solomon is a type of block-based error correcting codes and a type of forward error correcting code, an error correction technique to detect and correct a limited number of errors in transmitted data without the need for re-transmission [**rs˙bch˙overview**] and are best for identifying bursts of errors in the data. The data is first encoded using a geneartor polynomial composed of various shift registers $p(x)) * x^(n-k)modg(x)$. To decode the data bits, the syndromes, error locator polynomial, error polynomial coefficients[**reed˙solomon˙overview**].

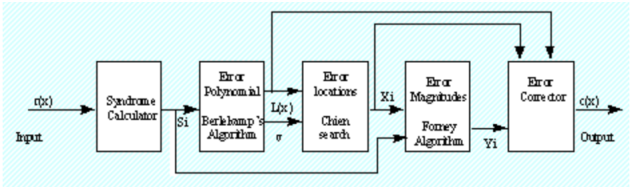


Figure 5: Reed Solomon Decoding Architecture

4.3.3 BCH

BCH codes (Bose–Chaudhuri–Hocquenghem codes) is a type of cyclic-error correcting codes (error correction codes) and optimal for detecting random bits of errors. Error correction codes add additional bits of redundant information to the original message in the form of an ECC; the redundancy allows the receiver to detect a limited number of bits that occur anywhere in the message and corrects the bits of data without re-transmission. BCH utilizes an encoding polynomial to encode the message and add additional redundant bits of information. Decoding consists of four steps: calculating the syndrome, another polynomial, determining the error location polynomial, determining the errors location polynomial, and the correcting the received pattern [**rs˙bch˙overview**].

4.3.4 Reed Solomon + Viterbi

Reed Solomon and Viterbi encoding and decoding follows the same implementation that is already described in the previous parts. The data is first encoded using Reed Solomon encoding; then, the data is encoded again using Viterbi. Once the data is encoded, the data is decoded using Viterbi and then decoded using Reed Solomon. By using both algorithms, bit error bursts (bytes) and random bit fuzzing can be detected.

5 DESIGN TRADE STUDIES

5.1 I2C and UART

The packet format for I2C and UART are very similar in structure with both requiring a small amount of bits (8-10) per ACK/NAK packet received. This is helpful because we can use the same general approach for both instances.

Motivation for I2C Packet Format - CubeRover

Function	Fault and Reg ID	Data Size	Data	CRC8
1 Byte	1 Byte	1 Byte	n Bytes	1 Byte

Standard for I2C Transfer

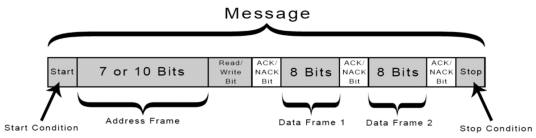


Figure 6: General Packet Format for I2C and CubeRover's Example Packet

Because the data size bit is 1 byte – we will limit our cases to 255 size of bytes for the data n. In total, we will have 259 Bytes transferred over I2C max for each data packet. Checksum will be helpful for error detection, but not for error correction. Our Cube Rover packet has the constraints with each I2C reply, MCU includes a fault bit CubeRover has a limit of 1 sec for a timeout Will at max try 3 times to initiate the same command. If this process fails, Safe Mode is enabled, and the fault register is set. CubeRover limits the total number of retries in any case to 3 tries and after that it will go into safe mode.

Considering what we know, there is already a system in place in case of failure. Checksum is not a guaranteed mechanism and can only do error detection at max. It does not take into consideration that the checksum itself can be compromised by radiation. We must work within the timeframe specified to error correct the entire packet sent or ask for a retry.

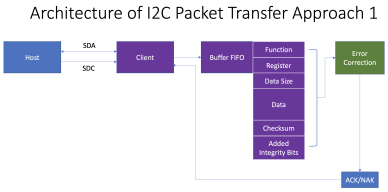


Figure 7: First Approach to I2C

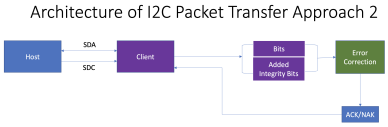


Figure 8: Second Approach to I2C

The two main approaches have two different metrics into consideration. With the first approach, we first accept all packets into a FIFO buffer and run our error correction on our entire data packet, then sending requests for missed packets. This helps avoid the extra computation time running the code on each individual packet, but can potentially increase the computation time exponentially. We decided to avoid this approach in the end because we would need to break the protocol by sending in ACK's after every packet received when it should normally be a NAK. To avoid confusion between protocols, we decided to go with approach 2. In approach 2, we are splitting the packet in half for each packet sent. With this we can employ a simpler algorithm than we could with the approach where we accept all packets at once. This however, cuts our code rate into 1/2 due to adding the extra integrity bits to the end of each half packet. This relies on an implementation of Hamming Encoding, we will explain the code rates for the other four as well.

Memory usage seemed minor for these implementations since they were operating on a smaller scale of packet sizes. At most, the matrices needed for some of the encoding and decoding algorithms reached a size of around 50Kb total for the program. This is static and won't change with different inputs to the encoding function since there is little dynamically allocated memory.

Our main focus was placed on the timing and performance of the algorithms used. Here are the results of our program with differing fuzzing rates for each protocol.

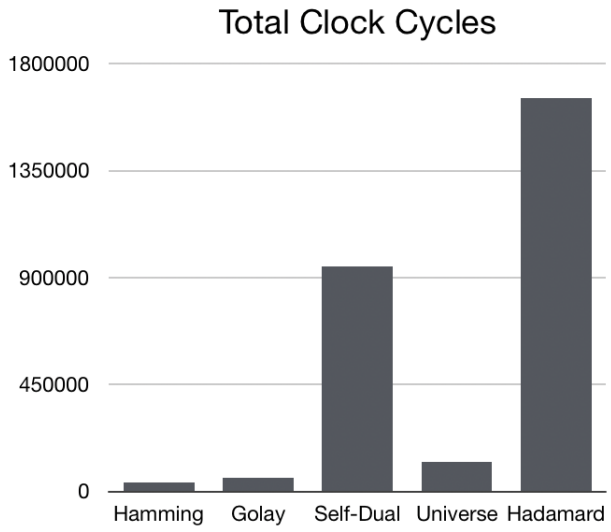
5.1.1 Performance

Figure 9: Performance on fixed size packages with $5\%, 10\%, 20\%, and 50\%$ fuzzing rates

From figure 7, we can see how well each is performing as we scale up the error rate. Hadamard consistently performs the best while the others drop significantly once we increase the fuzzing rate from 10 to 20%. Even once we had increased the fuzzing rate to 80+ percent, Hadamard still was able to correctly fix 50% of the packets being fuzzed. The performance wouldn't mean much without the timing, however.

5.1.2 Timing

We calculated a rough time and cycle approximation that ran on one core to try to simulate the results of the algorithms running on the micro-controllers.



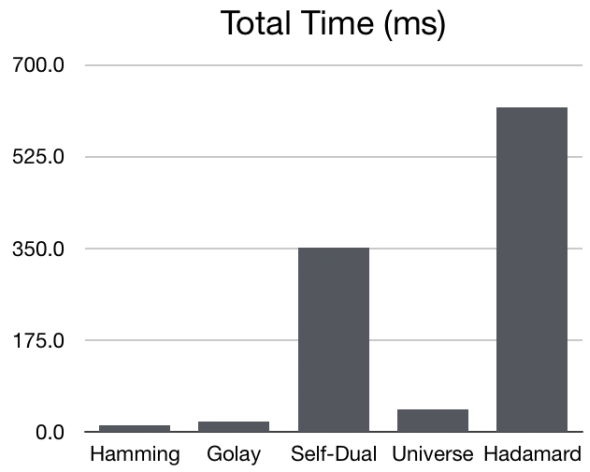
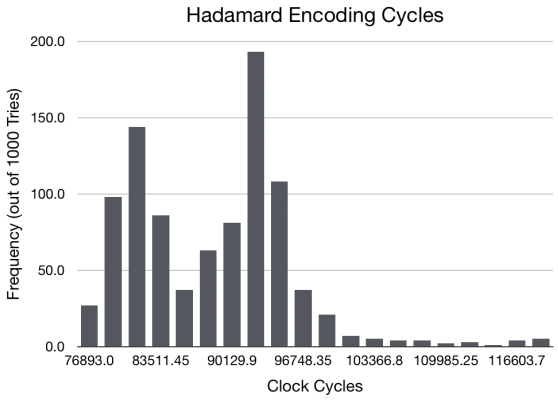
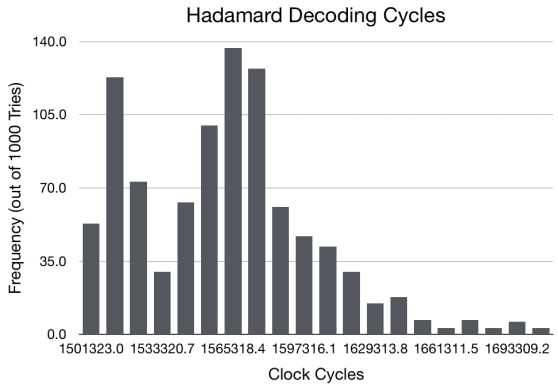
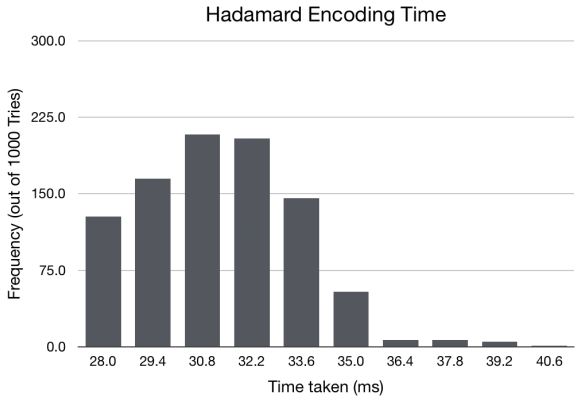


Figure 10:







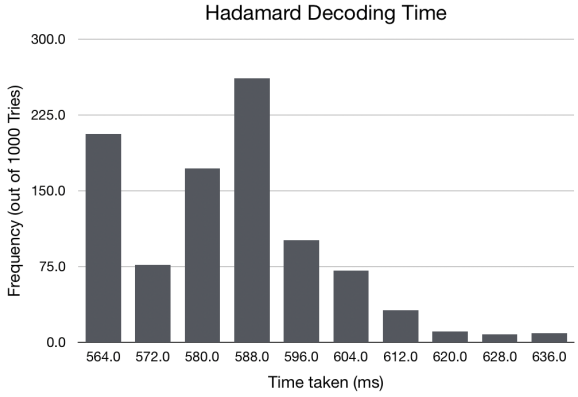
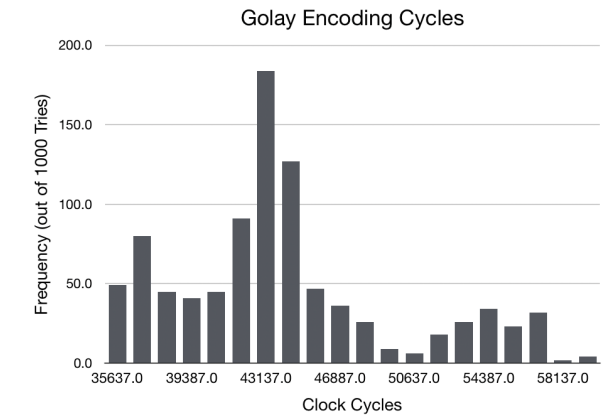
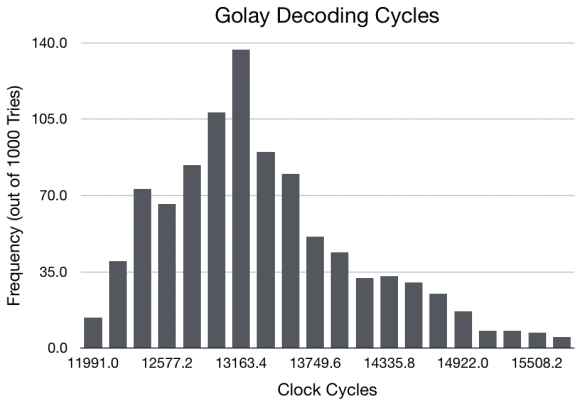
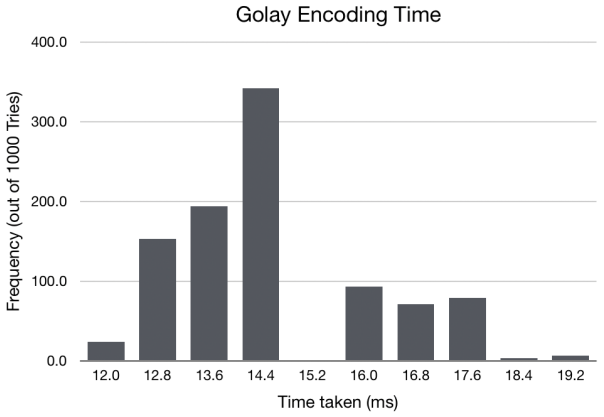


Figure 11:







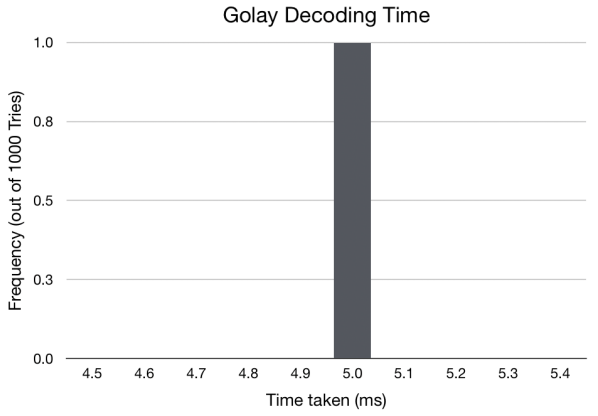
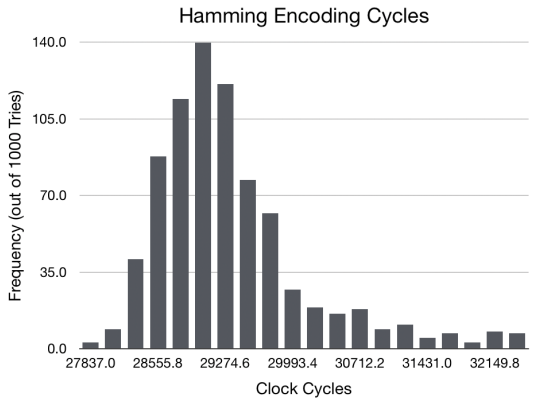
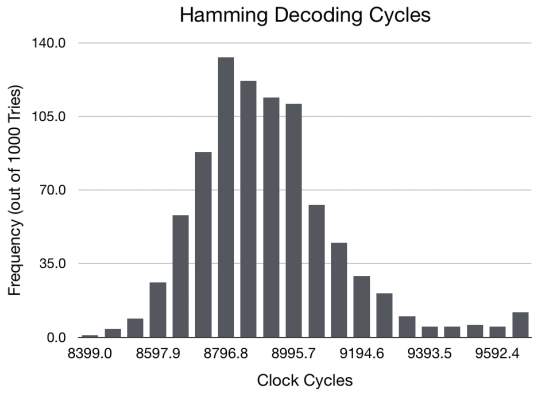
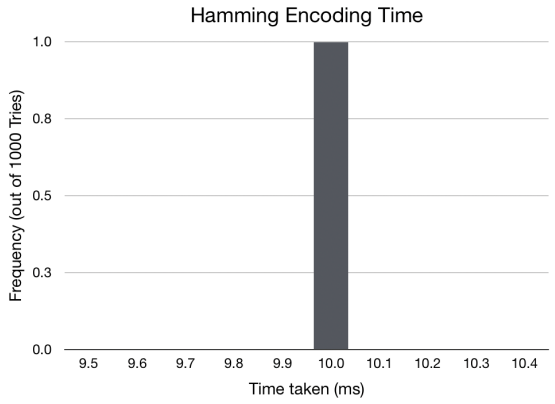


Figure 12:







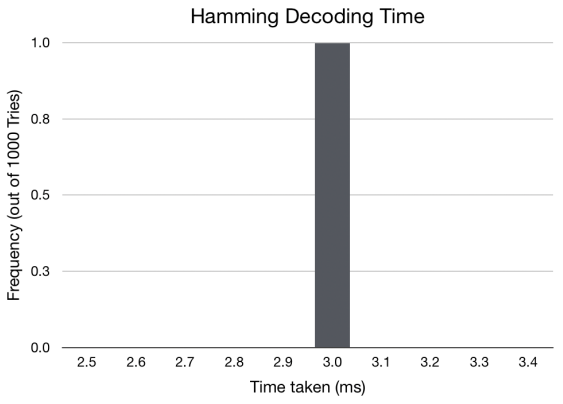
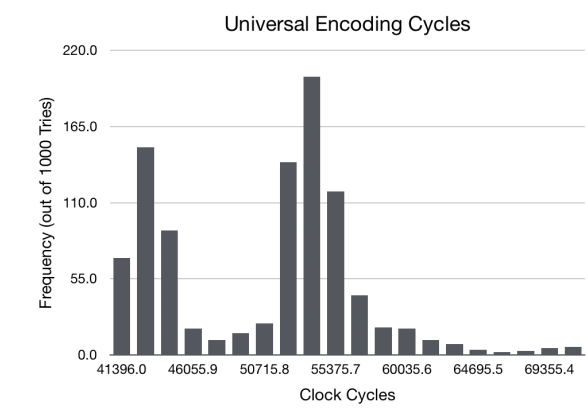
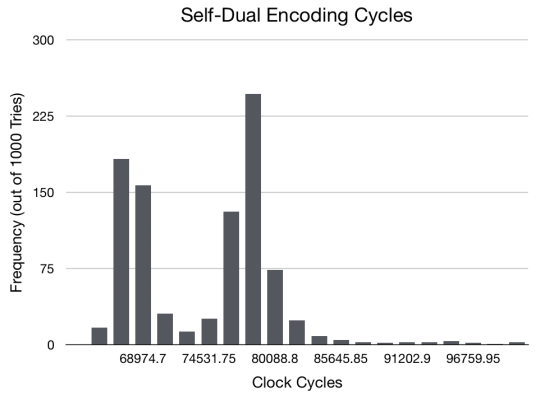
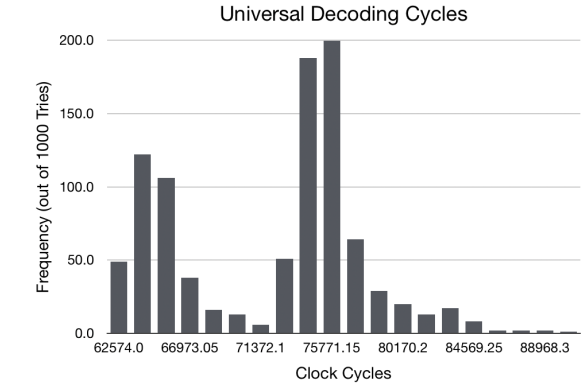
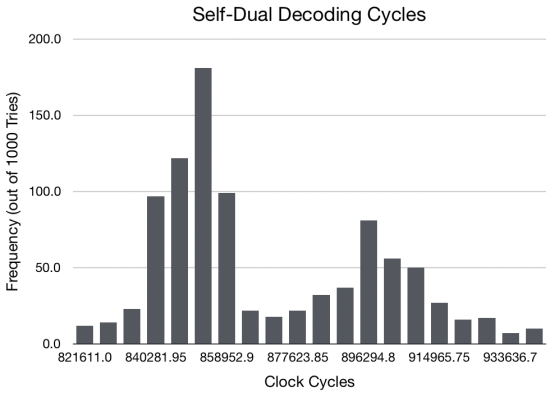


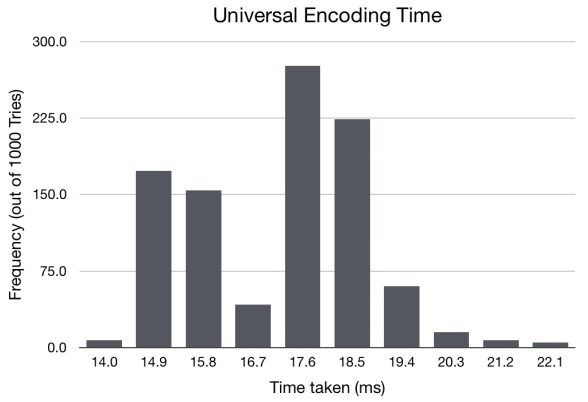
Figure 13:

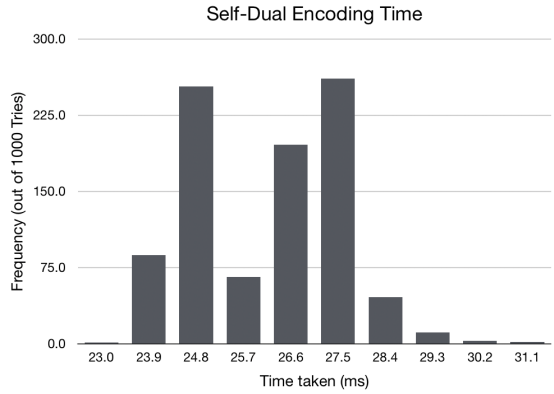


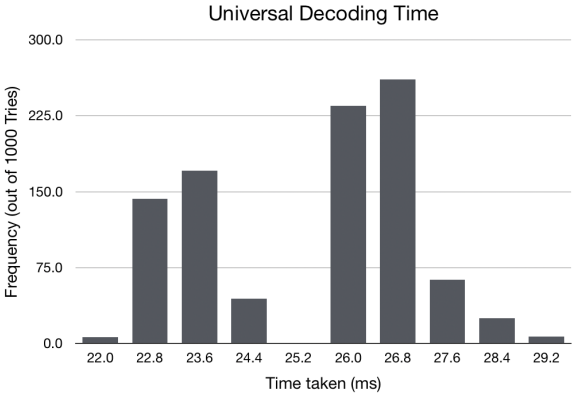












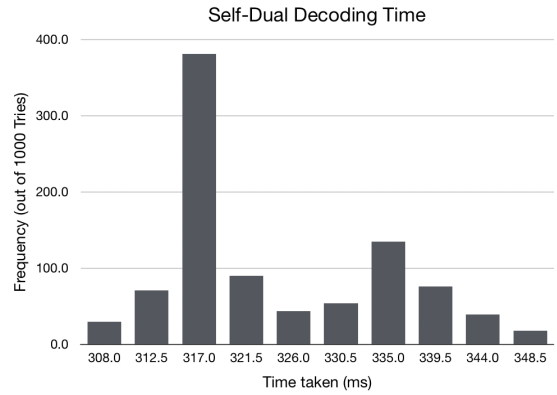
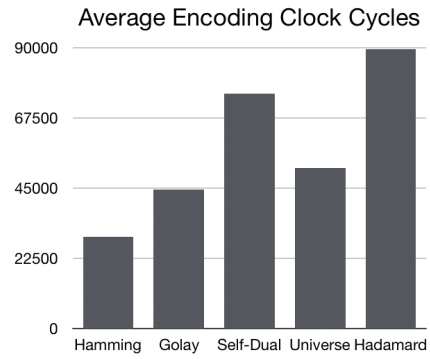
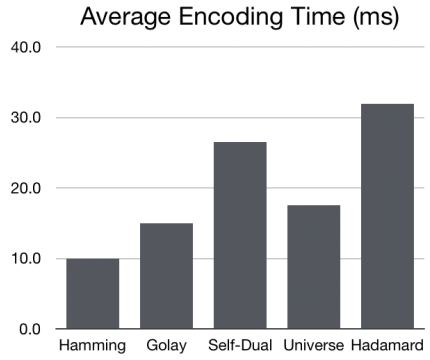
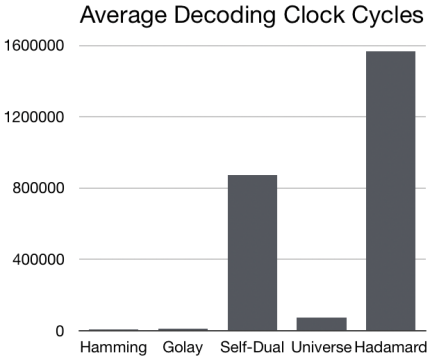


Figure 14:

Figure 15:







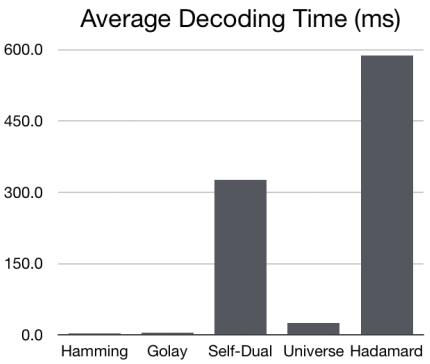


Figure 16:

When comparing the five different protocols in terms of total clock cycles (encoding + decoding) and total time (encoding + decoding). We can see that Hadamard is the only close to approaching our time constraint, but is still well below the threshold for sending one packet. You can see how many clock cycles and time it took for all the other protocols as a histogram with the frequency being how many times that value occurred over 1000 trials of running the encoding and decoding portions. The results can be filtered out if they are not statistically significant, i.e. removing outliers, so when there is only 1 value for timing, then that means the trials almost always were that value. Some analysis of how these algorithms would fair with our upper bound of sent packets is below.

	1 Packet Sent (Average) Cycles	16 Packets Sent Cycles	256 Packets Sent Cycles
Hamming	76,802	1,228,832	19,661,312
Golay	38,683	618,933	9,902,933
Self-Dual	473,857	7,581,704	121,307,264
Universe	124,307	1,988,912	31,822,592
Hadamard	1,656,500	26,504,000	424,064,000

	1 Packet Sent (Average) Time Sec	16 Packets Sent Time Sec	256 Packets Sent Time Sec
Hamming	0.03	0.42	6.66
Golay	0.01	0.21	3.40
Self-Dual	0.18	2.82	45.15
Universe	0.04	0.69	11.05
Hadamard	0.62	9.91	158.63

Figure 17: Average amount of cycles and time taken to run the algorithm on 8 bits for each algorithm adjusted based on per packet basis

A quick analysis of this table show how the times grow as we scale up our total size. Hadamard vs Hamming is the most obvious comparison to draw, which is that Hamming only takes 6.7 seconds total to send 256 packets while Hadamard takes more than 2.5 minutes to do the same. This can be a significant slowdown if most of the processing power is being taken to run these encoding and decoding functions even if they are under the 1 second constraint.

5.1.3 Code Rates

	Code Rate
Hamming	1/2
Golay	2/3
Self-Dual	1/2
Universe	1
Hadamard	1/16

Figure 18: Code rates of each algorithm

These code rates are based on how many of the useful bits are eventually encoded. For example, Hamming has 8-bits that it wants to send, but need 16 bits to fully encode the packet. This means that it has a code rate of 8/16 = 1/2. Hadamard has a very small code rate, but very

high accuracy. This is due to the redundancy built into the algorithm that allows for more faults.

5.2 SPI

SPI communication interface is most significantly used between the camera FPGA and the flight controller.

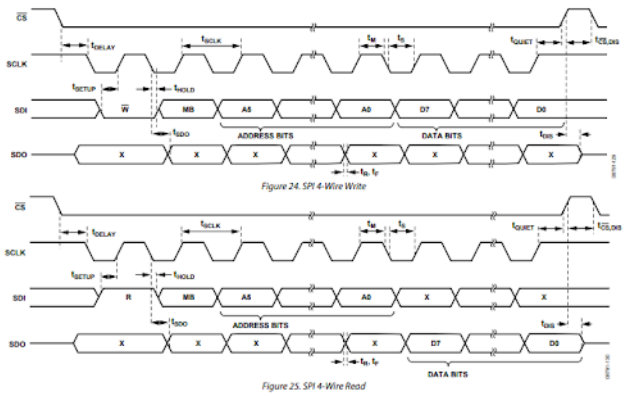


Figure 19: A visual of how the SPI packets are built for data transmission between the Primary Flight MCU and the camera FPGA.

In the timing diagram, the number of data bits is variables; once the address bits are transmitted, the receiving device does not know how many bits will proceed after the address bits. Additionally, CubeRover did not provide any data size limits in the draft of their documentation for the packet; the number of data bits could be of any size. Moreover, SPI doesn't use parity bits unlike UART/I2C, making error detection even more difficult. The implementations take an average amount of data that is greater than eight bits; BCH processes 1023 bits and Reed Solomon process 256 bits per iteration. Because of the unknown data size, I wanted to ensure that the protocols can handle large amounts of data; if they have the capacity to achieve high percentage of recovered data with large amounts of data, the protocols should be able to detect and correct errors in smaller streams of data.

5.2.1 Reed Solomon and BCH

BCH is a type of error correcting codes that is a generalized form of Hamming encoding; I chose implementing BCH over other encoding and decoding schemes because of the control over the precise number of symbol errors corrected by the code and how unlike Hamming encoding and decoding, it can correct multiple bit errors. Additionally, because BCH utilizes a simple algebraic method called syndrome decoding to decode data sequences, it's optimal for lower-power devices like the Cube Rover. The BCH Code configuration:

$$n = 1023 \tag{1}$$

$$k \text{ (dimension)} = 983 \tag{2}$$

After researching current BCH usages and most optimaly combintations of values for those variables, I decided to implement BCH with a code length of 1023. I chose to use 1023 because most implementations of BCH don't use code lengths greater than 100; common implementations like the ones that are used in control channels in cellular TDMA use code lengths of 48 or 31. Additionally, when the code length increases, the overhead will increase; for instance, the overhead increases about 0.4% when the code length increases from 511 to 1023.

Next, I decided to implement Reed Solomon Encoding and Decoding. Reed Solomon is a subset of BCH codes; unlike BCH codes, Reed Solomon is block-based error detecting codes. Block-based error detecting codes are optimal for detecting large error bursts, unlike BCH. Because the potential effects of moon radiation on the Cube Rover are unknown, including an algorithm that detects random bursts of errors diversifies the types of algorithms that I apply. By diversifying the algorithms, I hoped to see how each unique algorithm would perform given the randomly selected indexes that are chosed to be fuzzed. Moreover, Reed Solomon is widely used algorithm, as it's still currently used in storage devices like DVDs, CDs, and hard drives.

$$n = 1016 \tag{3}$$

$$k \text{ (dimension)} = 984 \tag{4}$$

When I was determining the parameters, I wanted to use a code length that was close to the value of the BCH parameters for consistency; if both algorithms used drastically different parameters, it may lead to incorrect comparisons due to different overheads and different distances between errors caused by the different ranges that the errors could possibly occur in.

5.2.2 Viterbi and Reed Solomon + Viterbi

I chose to research and implement convolutional encoding and Viterbi decoding because it's a popular and frequently used error correcting protocol for bit streaming; currently, it still being used in CDMA and GSM digital cellular, dial-up modems, satellite, deep-space communications, and 802.11 wireless LANs. Convolutional codes like Viterbi are also not based on blocks of bits but instead a stream of bits, more closely aligning with how SPI functions.

$$r = 1/2 \tag{5}$$

$$k \text{ (dimension)} = 7 \tag{6}$$

The convolutional code used by the Viterbi algorithm is defined by two parameters: code rate and constraint length. Code rate (R = k/n) is the ratio between the number of input bits into the convolutional encoder (k) to the number of channel symbols outputted by the convolutional encoder (n), ranging from to . After researching optimal code rates and factors that affect the code rate, we determined that the code rate will be decided when the implementation is completed. Since code rate is not a hard specification and other components don't depend on the code rate, various code rates will be tested to determine which one has the lowest latency and smallest overhead.

Constraint length (K) is the number of input frames held in the k-bit shift register. Like our approach for determining the code rate that will be used, we will determine the most optimal constraint length for our implementation by trying various constraint lengths; the most optimal constraint length will yield the lowest latency and smallest overhead.

After thorough research into determining the most optimal code rates, out of the following code rate and constraint combinations ((R = , K = 7), (R = , K = 8), (R = , K = 9), (R = , K = 3), (R = , K=9), (R = 5/7, K = 3) and (R = 13/14, K =4)), a code rate of 1/2 and constraint length of 7 best fit the constraints of the project while being able to recover the most fuzzed data. Prior to implementing the algorithm, after thorough research of frequently used constraint lengths and code rates, I found that the Voyager, an American scientific program that successfully deployed the first two man-made robots to the moon[**nasa˙viterbi**], and (1/2, 7) is the standard for current deep space applications. Because of new technological developments since that project, another popular configuration is (1/2, 10). However, after running both configurations with Viterbi, (1/2, 7) achieve a much higher percentage of data recovered than (1/2, 10).

Combining both Viterbi and Reed Solomon in my implementation combines the two standout capabilities of both algorithms: being able to detect error bursts (byte errors), Reed Solomon, and being able to detect individual bit errors, Viterbi. Despite the significant costs and overhead of implementing both algorithms on the rover, combining both could have yielded a significantly higher to almost perfect data recovery rate. I use both of the implementations of the protocols that are already described above; to implement both, I encoded the data stream using Reed Solomon and then Viterbi and to decode the data stream, I decoded using Viterbi first and then Reed Solomon. Encoding the function using Reed Solomon first would detect the large chunks of fuzzed data; once the sections of fuzzed data are identified, Viterbi would be able to identify the remaining bits that are left. When I reversed the encoding/decoding order, the percentage of fuzzed data recovered was identical to the amount of fuzzed data that was recovered only when I implemented Viterbi, defeating the purpose of using both algorithms.

5.2.3 Performance

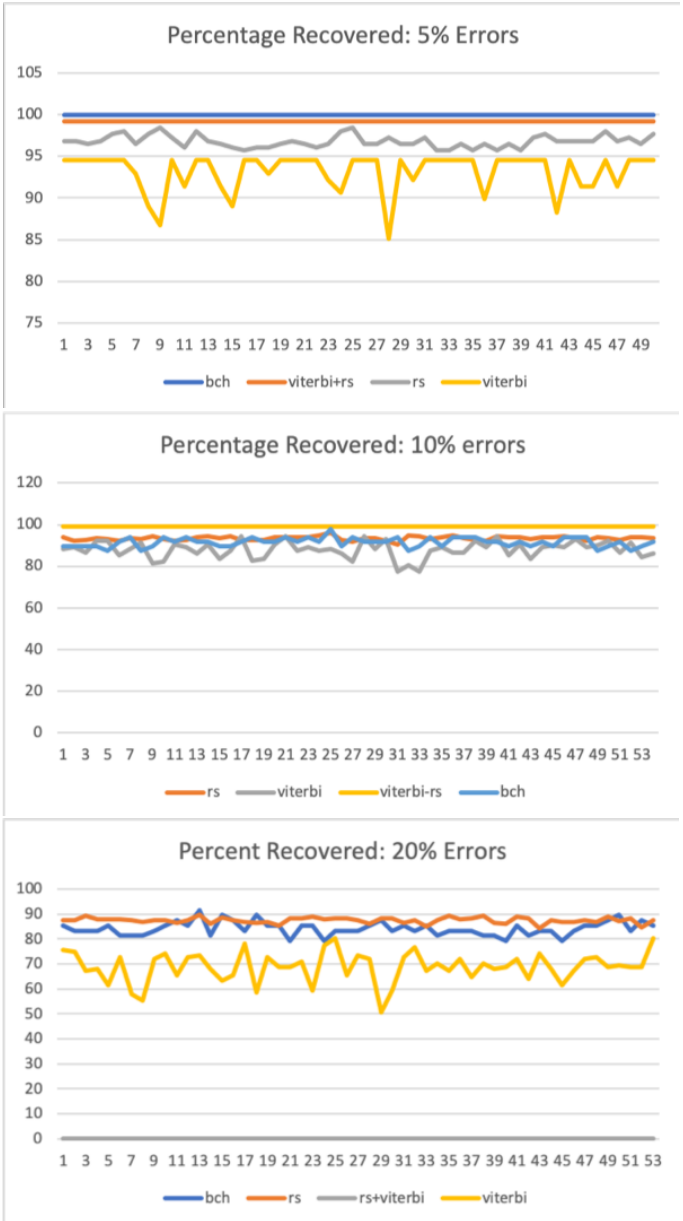


Figure 20: Line graphs that reflect the percentage of data recovered when each protocol fifty times at different amounts of the data being fuzzed. The percentage of data recovered is the fraction of data that is recovered after one iteration.

5.2.4 Timing

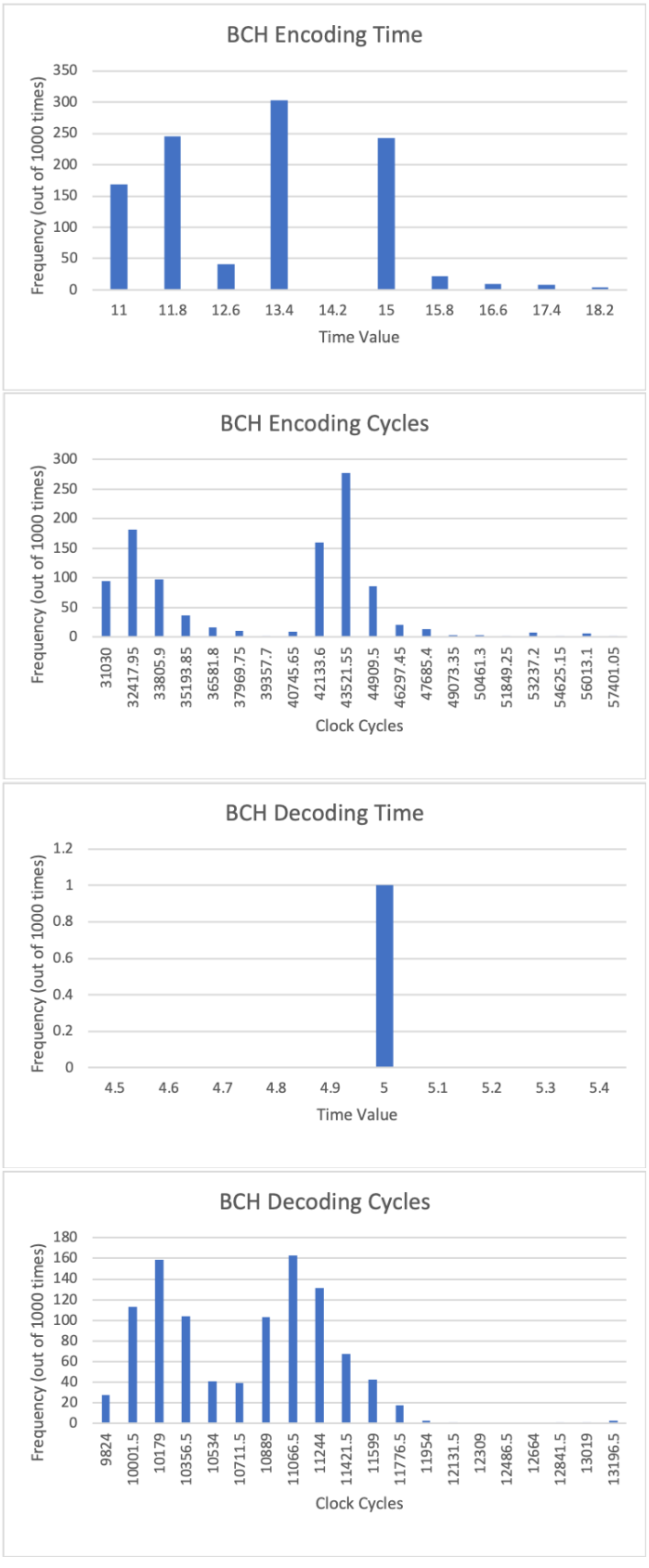


Figure 21: BCH Encoding and Decoding Time and Cycles

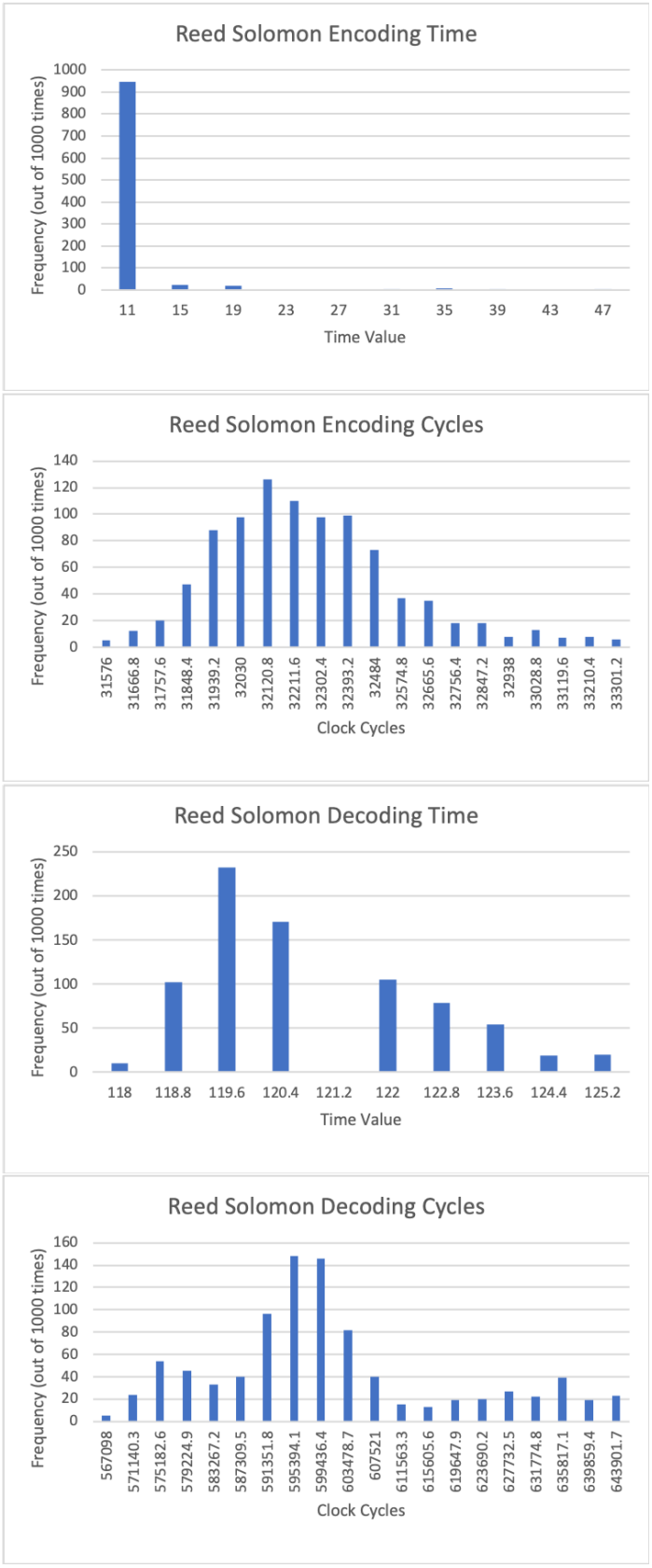
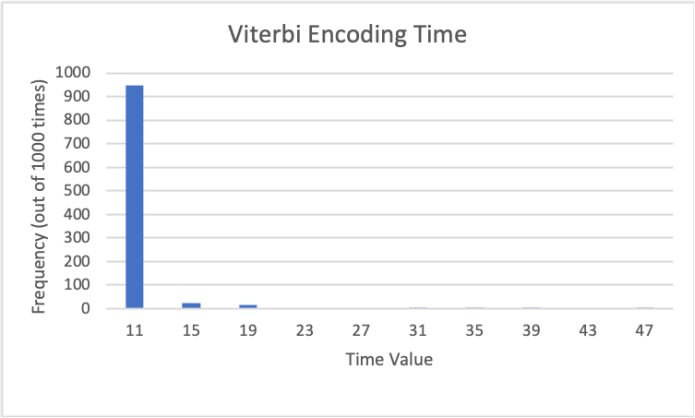
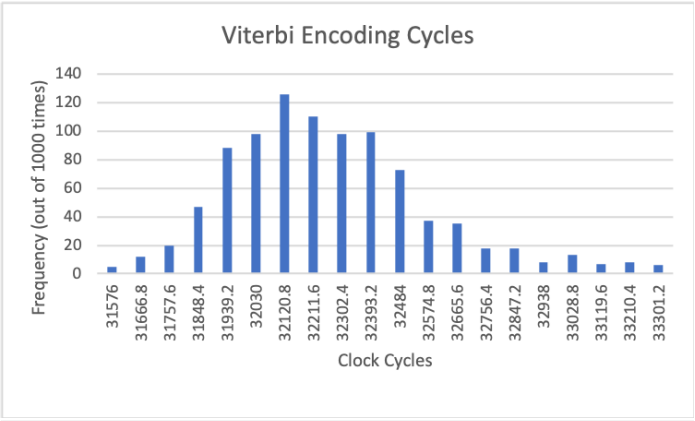
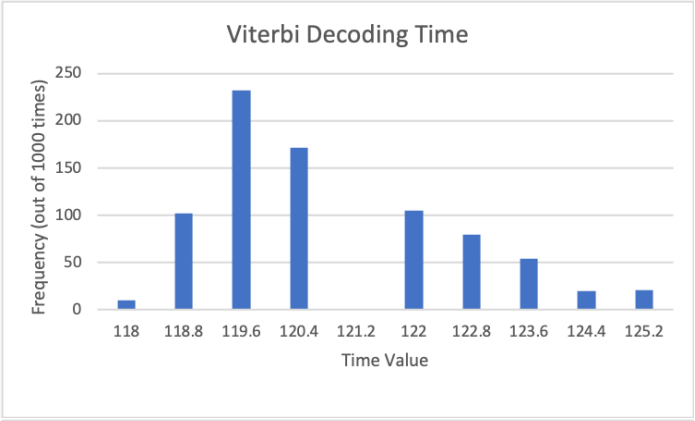


Figure 22: Reed Solomon Encoding and Decoding Time and Cycles







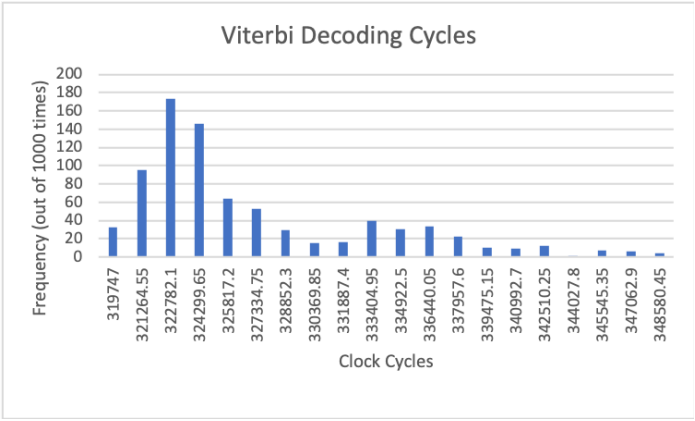


Figure 23: Reed Solomon Encoding and Decoding Time and Cycles

5.3 WiFi

For the UDP protocol the goal was to create a protocol for UDP that ensures packet delivery. Looking at the existing IP protocols that are out there TCP is one of those protocols that ensures packet delivery by standard. The existing TCP stack however is much larger than the UDP stack that is running on the rover. The goal was to create a TCP wrapper around the UDP interface in order to create a lightweight protocol that ensures packet delivery. Within that TCP wrapper we also performed a trade study to determine the effects of a windowing vs a non-windowing protocol when dealing with large numbers of packet drops. As opposed to the other protocols that we worked with UDP focused mainly on restoring dropped packets over data fuzzing.

A couple of things to consider when looking at the design of the UDP protocol. First is the absence of the standard 3-Way TCP handshake. There were a couple of reasons we decided to leave this out of the protocol and the simulator. The first was that we are only doing communication between 2 hosts. In our product application (CubeRover) the rover only needs to talk to the lander and vice versa. This would mean that ideally the rover would have one port dedicated to sending data and one port dedicated to receiving data, and the same on the lander. This would mean that the lander would already be expecting data from the rover and vice versa, thus the 3 way handshake wouldn't be needed to establish a connection. However with this approach one would need to build in a defined structure in order to identify what data is being passed over WIFI. On CubeRover this was not a problem as they had already defined their message structure for a standard UDP protocol. However if other users were to use this design they would have to make those considerations themselves as processing larger blocks of data could add some overhead to the design.

Second is the verification accuracy of checksum when it comes to data fuzzing. When we first began this project we felt that checksum would be enough to detect the aggressive data fuzzing that we would be conducting on the protocols. We quickly found out that this was not the case, although checksum is decently good at checking for 1-2 bit errors per word it isn't very good at anything else. This led to verifying some packets that were not correct at higher fuzzing rates. Upon exploring other options to mitigate this we found that cyclic redundancy codes (CRCs) would be better and just as cheap as doing a checksum [**5**]. As for our simulator we stuck with checksum in order to follow the bounds set for us by CubeRover which was to design a protocol that worked over UDP, and as a standard UDP uses checksum. However, if we were to do this project again more consideration would've gone into CRC and similar verification codes.

5.3.1 Performance

When we first began this project we thought that the windowing approach would be the best one in order to en-

sure validated packed delivery as fast as possible. But after we ran the simulation it became clear that this was not the case. Both of the figures below show the number of packets transmitted with drops vs number of packets transmitted without drops (max number of packets delivered). I performed the tests this way in order to ensure that the speed of my simulation was not affected positively or negatively by gathering this data on my personal computer. In order to gather macro data on the performance of the WIFI protocols incremented probabilities 0 to 0.5 stepping every 0.0001 and for each probability I ran the simulator 100 times in order to generate the data. After the simulation data was generated I used MatLibPlot [**6**] in python to plot the data.

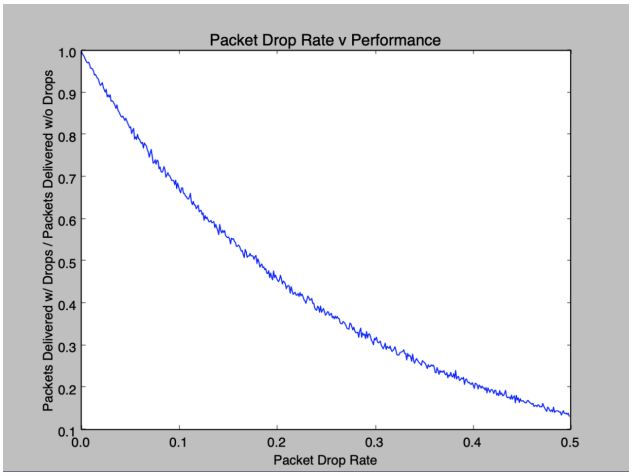


Figure 25: Line graph that reflects the performance of the non-windowing implementation

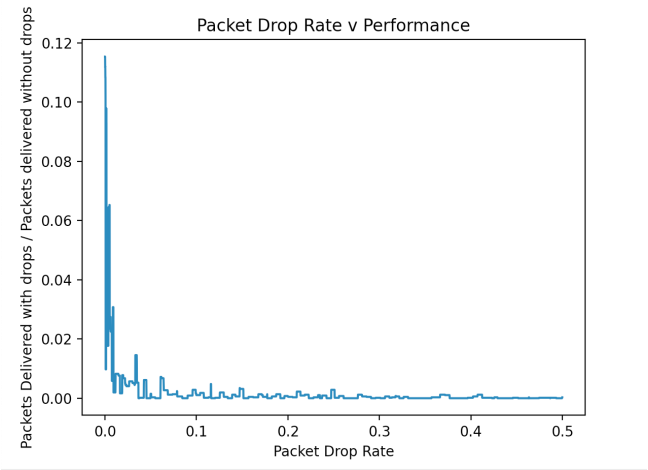


Figure 24: Line graph that reflects the performance of the windowing implementation

Figure 21 shows the performance statistics for the windowing implementation. As you can see The windowing performance does a really good job when the packet drop rate is around 1-2% operating around 20% above the ideal (no drops) non-windowing implementation. Quickly after we can see that there is a steep fall off. This is most likely due to the fact that windowing requires a series of successful packet deliveries, thus if one packet in the series is dropped, the entire window will have to be re-transmitted. We also noticed that the graph was quite choppy when compared to the non-windowing implementation, we suspect this to be random outliers where a window can actually be sent.

We contrast the windowing implementation with the statistics gathered from the protocol without windowing. We can see that even though there indeed is a fall off in performance as you increase the packet drop rate it is nowhere near as steep of a drop off as the windowing implementation. Even at aggressively high packet drop rates such as 0.5 we still are transmitting packets at 13.5% of our potential.

We can conclude that based only on the data displayed above that the non-windowing implementation performs way better in low drop rate scenarios (<2%). However, once the drop rate goes above that threshold it is far better to use the non-windowing solution.

5.4 GUI

Library	Pro's	Con's
PyForms	The library was initially easier to use; creating a form was straightforward and easy to figure out. Additionally, provided a quick to create the GUi and not take time away from the main focus of the project.	The pyForms library was not well-documented at all; it provides a basic form that was good enough when we started. Because we only had a console log and a line graph at first, pyForms worked well for the minimally functioning GUI. However, when we wanted to add in a graph like the data visualization, pyForms barely had any documentation regarding how to integrate and upload photos to the GUI.
QTPy	The library is significantly more documented and a lot more resources existed for the GUI online. Additionally, a designer application called QTPy designer helped with easily creating the GUI, removing the hassle of worrying about the GUI appearance or the syntax for the GUI implementation.	QTPy is more complicated than pyForms; everytime we wanted to change the GUI, we had to use the QTPy designer application and regenerate the Python code.

Figure 26: GUI Implementation Trade-Offs Between PyForms and QTPy

5.5 C-Extension Interface

Library	Pro's	Con's
cTypes[4]	cTypes extends Python and wraps C functions so that they are callable from Python. Using cTypes allows most of work to be completed in C rather than Python, since any C function can be called from Python. Additionally, it's a lot quicker to use, since the C functions are wrapped in Python rather than re-compiling an entire file.	cTypes is difficult to debug, since mistakes that causes segfaulting and bus errors will occur in Python.
Cython [3]	A superset of the Python language, Cython can translate Python files into valid C programs. Additionally, Cython can optimize Python programs by using explicit type declarations in Python code, allowing it to convert the code into pure C. It's also a lot easier to use, because the program simply converts Python to C without having to worry about importing libraries in your functions.	Using Cython doesn't speed up conventional Python code that much; if Cython encounters Python code that it can't translate completely, it will instead transform that code into a series of C calls to Python's internals. Also, Python data structures are slower in pure C. Additionally, if we used Cython, the GUI would have been implemented in C as well, making it significantly more complicated than it needed to be.

Figure 27: C-Extension Interface Trade-Offs Between Using C-Types and Using Cython.

6 DESIGN SPECIFICATION

6.1 I2C/UART Implementation Details

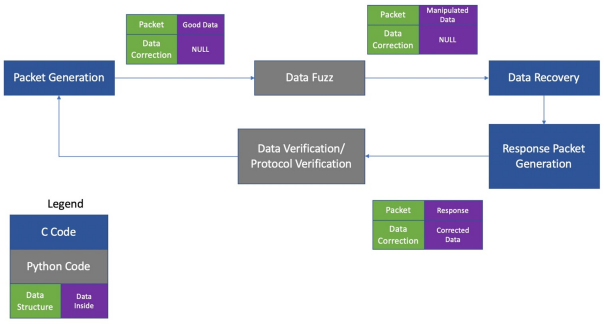


Figure 28: A visualization of the breakdown of the implementation of and relationship between packet generation, data fuzzing, data recovery and response packet generation for I2C/UART.

To best simulate and resemble how packets will actually be passed through the protocols on the CubeRover, we decided to separate the data fuzzing and verification from the packet generation, data recovery and response packet. The data is first randomly generated in C; and the locations and values of the errors are first randomly generated in Python; the data will then be passed to a function as a data structure to the encoding functions set by the GUI. The fuzzed packet goes through the Data Recovery step which is essentially the decoding function and passes it into the Response packet generation, which would signify either the ACK or NAK value, meaning that the caller would have to resend the same packet. When passed back to the Python part of the code, we check if the protocol met what we defined as the acceptable response and see if the corrected packet was actually what we intended to send. Lastly, we recall the generation function to begin the cycle again.

6.2 QSPI Implementation Details

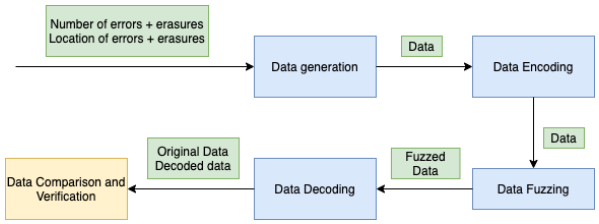


Figure 29: A visualization of the breakdown of the implementation of and relationship between packet generation, data fuzzing, data recovery and response packet generation for the BCH and Reed Solomon Implementations. The parts in yellow are implemented in Python, the parts in blue are implemented in C, and the parts in green are the inputs.

Based off the library [**1**] that is used, most of the work is done in Python; Python is only used to call the function and pass parameters from the GUI and to determine how much data was recovered. Five parameters are passed to the data generation function in C: the number of erasures, the location of erasures, the number of errors, the locations of errors, and the number of initial zeros (only for Reed Solomon). The number of erasures and errors is determined by the inputted percentage of fuzzed data; with this calculation, the unique locations (indexes in the bit array) of the errors and erasures are determine. These five values are then passed to a function in C that initializes the protocol; the data is randomly generated and then encoded using either BCH or Reed Solomon encoding. The data is then fuzzed; the data is fuzzed inbetween encoding/decoding to simulate how the data may be affected by channel noise/external conditions. Afterward, the fuzzed data is passed to the decoding function; once the fuzzed data is decoded, the original data and the recovered data are returned back to initial Python function, where the data is compared to determine how much was accurately recovered.

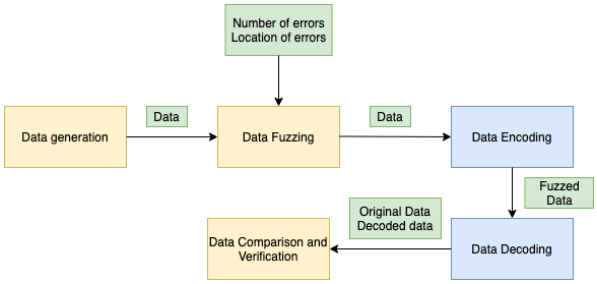


Figure 30: A visualization of the breakdown of the implementation of and relationship between packet generation, data fuzzing, data recovery and response packet generation for the Viterbi Implementation. The parts in yellow are implemented in Python, the parts in blue are implemented in C, and the parts in green are the inputs.

Based off the library [**2**] that is used, the data is randomly generated in Python; unlike the previous protocol, the data is fuzzed in Python. The data is fuzzed by selecting a certain amount of data (the total length times the percentage of bits flipped) and then selecting random indexes in the string to randomly change. The fuzzed data is then passed to the encoder; the encoder will return the fuzzed data. The fuzzed data is then passed to the decoder which will decode the message. Afterwards, both the original data and the recovered data are compared in Python to determine the amount of data that was accurately recovered.

6.3 UDP/Wifi Implementation

The WIFI simulator acts as an extension of the existing UDP interface. It works by providing a channel through which messages can be encoded and decoded as well as dropped. The simulator was written in C without the help of any outside libraries.

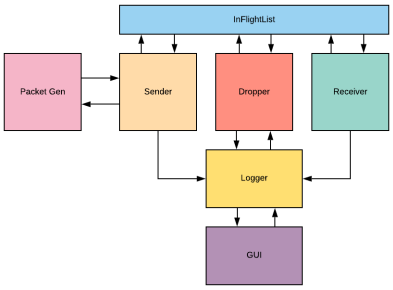


Figure 31: Block diagram showing how WIFI simulator works.

The simulator has 7 distinct parts (as shown above). The first t is the packet generator, we did not end up using the packet generator that all of the other protocols were using because they were not exactly the type of packets that we wanted to produce for the WIFI simulator. This generator is responsible for producing the data that is to be transmitted. The next module was the sender, the sender deals with the protocol on the sending side of the interaction, on the receiving side the receiver deals with the protocols on the receiving side. The inflight list was our way of simulating packets that are "in the air" (going from the sender to the receiver). We opted to make this a list type object in order to make it easier to keep track of the packets in flight. The packet dropper acts as a middleman between the sender and the receiver. The packet dropper handles the dropping of packets by traversing the inflight list and deleting nodes based on the random probability given. The logger is the outside wrapper for the whole simulator. It keeps track of all of the sent, dropped, received packets and relays that information to the GUI in order for it to be displayed.

As a side note, the simulator also contains parts that are not super necessary to its function. Fields like source port and destination port contained in the packet structure were not used in our simulation for routing but rather were put there to make the protocol appear to be a little more realistic. If we were to have had an opportunity to implement this on the micro controllers like we had planned these fields would have come into use.

6.4 GUI

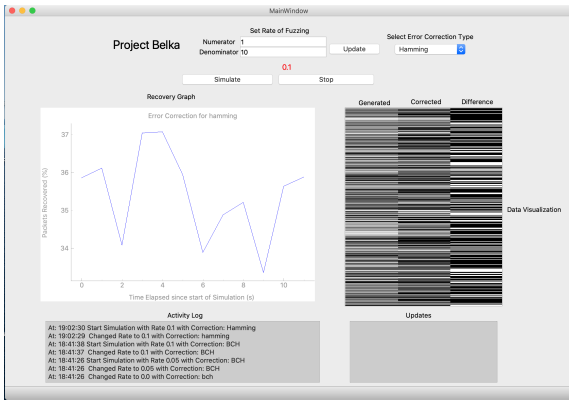


Figure 32: The GUI screen for a I2C/UART

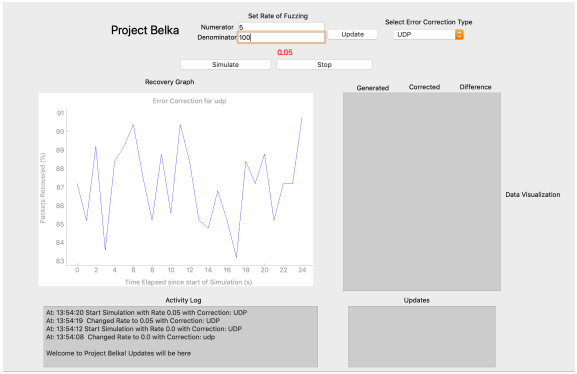


Figure 33: The GUI screen for WIFI Protocols

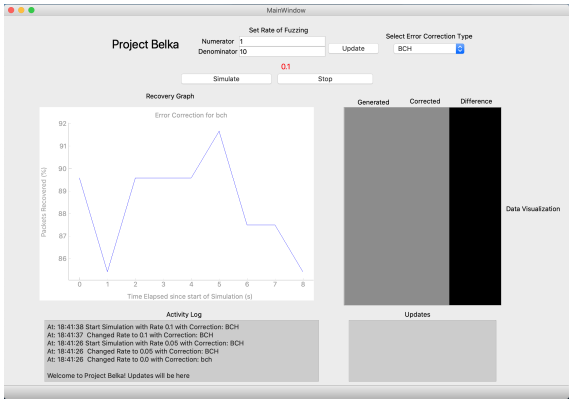


Figure 34: The GUI screen for QSPI Protocols

The GUI allows users to configure and simulate various protocols; users will pick a protocol using the drop down menu in the right corner and then input the fraction amount of data that will be fuzzed. The GUI will update with their inputted configuration and will continuously simulate the protocol on the screen by constantly updating the screen with the results of each iteration.

Real Time Line Graph: The real time line graph displays a time vs. percentage of data packets corrected for each protocol. Time is in seconds on the x-axis, and the percentage of data accurately recovered is on the right hand axis. For the protocols intended for I2C/UART usage, the percentage of data recovered is the number of packets accurately recovered in that second; for example, in that second, the algorithm ran 10 times and was able to recover 90% (nine out of ten packets). Because the size of the data is significantly larger for the protocols intended for QSPI usage, each second, the protocol is ran one time, and the percentage of data that is recovered is the number of bits that are recovered for that iteration; for instance, if the data is fully recovered for that iteration, the percentage of data recovered is 100%. Otherwise, if the data was not fully recovered, it will be reflected as a value ¡100%. Every time a new protocol or change in percentage of data being fuzzed is simulated, the line graph will reset.

Data Visualization: The data visualization reflects the generated data, the corrected data, and the difference of the generate and corrected data.

Real Time Log: The real time log displays the changes that occur on screen and the time stamp at which they occurred. The time log allows the user to keep track of the protocols and configurations that they have simulated.

Updates: Updates originally used to show any major changes within the system including errors. To better utilize this space for data collection, after simulation stops, the results are piped into the update box that included the rate of fuzzing and the data points within the graph. This helped because the program could run its course and then the user would be able to copy and paste with little hassle.

7 IMPLEMENTATION

The GUI communicates the protocol and the error rate that the user would like to simulate to the STM 32; the STM 32 will send back to the GUI whether the protocol was successful at error detection and correction. To implement these interactions, microPython best fits our needs because of the libraries that it offers for the UART ports that the design requires and its supported by the STM 32 devices. The table in part 4.4 outlines why microPython is best for our implementation.

The GUI will be implemented in Python, since that is the easiest language to use with microPython. The standards for I2C, UART, SPI, GPIO and Wifi transmission and protocols that wrap around them will be implemented in C. For the programs on the STM 32, the STM32 Cube Library with HAL will be used since it provides all the drivers necessary for sending and receiving data to and from the I2C, UART, SPI and GPIO ports. For the programs on the MSP 430's, DriverLib will be used since it contains drivers for GPIO, UART and I2C standard implementations. For the programs on the TI Hercules, HALCOGEN (Hardware Abstraction Layer Code Generator) will be used since it provides libraries and drivers to send/receive data from SPI, I2C, GPIO, and UART communication.

The WiFi Protocol will be written in C using the socket library. This is very bare bones but will ensure that the code is very small. A queue will be used to store packets while waiting for ACKs coming from the destination. The entire protocol will run under a HTTP interface.

8 VALIDATION

8.1 Verifying Latency Requirements

PAPI: We used the Performance Application Programming Interface (PAPI) which allows the use of low-level performance counters on hardware to gather statistics for users. We did this by gathering both the time and cycle counts and the beginning and end of the encoding and decoding functions. Using our CMU ECE cluster computers to run the software, we had to be wary of the performance that the cluster machine would provide. By running on one core only and setting a high priority, we tried our best to accurately simulate what the micro-controller would do in isolation. At the very least, we were able to relatively compare the different algorithms running using the same standard. We also ran the same function 1000 times for each protocol to help negate some of the inconsistencies from run to run.

8.2 Verifying Power Requirements

We originally wanted to test the values of power being dissipated by each Micro-controller running our function by using oscilloscopes to find the differences. However, due to the unforeseen circumstances we were unable to access the necessary equipment to test such a case and also did not have the actual components to use their system settings.

9 CONCLUSION

9.1 I2C/UART Recommendation

Out of the five algorithms that we tested for our fixed packet size model, there was a clear winner in terms of performance which was Hadamard. Hadamard survived in the most extreme conditions of fuzzing, but it always comes at a cost. We could clearly see the cost scale up in terms of time and cycle count when reaching the upper bound of our total packets size. Another drawback of Hadamard was due to the code rate. With 1/16 code rate this could very easily bottleneck the wires when sending over the mass amounts of data. The other algorithms would probably also fair better with an added level of redundancy that Hadamard has, but has not been formally verified.

If we are concerned with pure performance with most correction, then the recommendation would be for Universe encoding. If we are concerned with absolute data integrity and are only considering sending small numbers of packets, then the clear winner would be Hadamard due to the high rates that it can endure by nature.

9.2 QSPI Recommendation

After thorough testing and analysis of the four protocols that were implemented, Reed Solomon and BCH tied for being the best protocols; both protocols are intended for different kinds of error detection, burst errors and and random bit fuzzing. Both protocols also best met the constraints set forth by the CubeRover design; however, BCH uses the least amount of clock cycles and time to detect the errors, therefore I recommend BCH. However, I feel that this is contingent on how moon radiation affects the data; if the moon radiation is more likely to cause error bursts rather than random bit fuzzing, finding the capcity to implement Reed Solomon should be considered. Contrarily, if the moon radiation is more likely to cause random bit flipping, BCH should be used.

9.3 WIFI Recommendation

With regard to the WIFI recommendation we can confidently say that a TCP "wrapper" around a user datagram protocol (UDP) works in order to verify the arrival of packets at their destination. With the two protocols we tested both worked to some degree but the non-windowing approach was far superior when dealing with large amounts of packet drops. However if you can confirm that your packet drop ratio is less than 2% a windowing approach would yield a higher transfer rate. In terms of making a recommendation for the CubeRover project the ideal protocol would depend on what the actual conditions of the moon are. If the conditions are such that there would be more than a 2% packet drop rate over WIFI it would be a much better idea to go with the non-windowing solution, but if the drop rate is less than 2% and you needed an extra boost in throughput performance windowing would be the solution. All of this being said there is still a lot to be done in order to fully port this onto a micro controller that would work on devices such as the CubeRover, our claims come merely from simulation.

10 PROJECT MANAGEMENT

10.1 Schedule

Looking at the Gantt chart (Figure 14), we modified the original Gantt Chart due to the current circumstances; unlike the previous Gantt Chart, we spent the second half of the semester researching, creating and integrating more protocols. After we initially implemented the original algorithms, we realized that it took about a week to research the most optimal protocol to implement, create and integrate the protocol, and see if the protocol actually met our expectations for performance and percentage of data recovered. Like the previous Gantt Chart, we still created time to integrate each of our different protocol implementations with the GUI and built in Slack time. Additionally, because we are no longer using hardware, there is no time built in for testing, since testing each algorithm became part of the week long development of the protocol; testing the protocol with the GUI became part of the integration of the protocol with the GUI.

10.2 Budget

From 12.1 a lot of the things we had purchased were not able to be used. We used the Microcontroller for some initial testing, but trying to integrate a low level project like this would have been less cost efficient than not using the parts at all because we would have had to order two more sets to be fully functional across the team.

10.3 Team Member Responsibilities

The team member responsibilities were defined and clear; John Paul was responsible for I2C/UART protocols (protocols that transmitted small packets of data), Mia was responsible for QSPI protocols (protocols that transmitted data that was 256¡ bits), and Sam was responsible for all WIFI protocols. Regarding the GUI, John Paul and Mia were primarily responsible for creating the GUI and making sure that it was easily to integrate with our protocols. Everyone worked on integrating their own protocols into the GUI.

10.4 Risk Management

10.4.1 Design

The biggest risk was initially working around the current circumstances; because our project was initially more focused on integrating the protocols with the actual hardware, we were concerned with how to redesign the project to best achieve our original goal and proposal. Additionally, because we were using unaccounted time to figure out how to move forward with virtual classes, we became slightly behind in our schedule because of the lost time but we wanted to make sure that we were fully confident in the new statement of work that we proposed. To mitigate these risks, we ensured that the new proposed statement of work was thorough and thoughtfully planned out to ensure that we wouldn't have to back track and rethink our new de-

sign. We did decide on a minimal project (Hamming algorithm, Viterbi algorthm, UDP algorithm and GUI) should we be put too far behind our schedule; that's why, on the Gantt Chart, we integrated our initial protocols first to ensure that we had a completed GUI with minimal protocols. Once the GUI was finished and the protocols were implemented, we continued researching protocols with the remaining time.

10.4.2 Schedule

Because of the re-design setback, we lost some time on our schedule to complete our project. By spending time to re-design our project and to rewrite some parts that were specific to the previous design, we found ourselves slightly behind, something that we had not expected. To make up for lost time, we called each other more frequently to speed up the integration process and prevent each other from potentially being stuck. Moreover, we arranged the schedule so that we had a minimum product if we were unable to complete everything that we had planned.

11 SUMMARY

The WIFI goals were able to be met and meet the system's performance goals. Using the conservative non-windowing protocol we can see that if we were operating at a 50% packet drop rating we would be able to recover all the packets at 13.5% of the throughput of a protocol without packet drops. Using the requirements that we outlined in the beginning of the project with a goal of 20Mbps this would be achievable on the CubeRover as their WIFI chip can operate up to 20MBps making our 13.5% throughput at a 50% packet drop rate 20.8Mbps thus satisfying our requirement.

The UART and I2C goals were met by being well under 8 million clock cycles for one packet. It seems like a hard cycle count to reach, but other preliminary encoding algorithms could go well above that for the level of redundancy. Things we wish we could do for this would be getting accurate timings on the system components. We also really wish that we could testing on the data transfer speeds because that could also take up a significant amount of time for that one second delay, but we would never know without the hardware.

11.1 Future Work

We do not intend to work on this beyond the semester, but are planning on handing over our findings to the CubeRover team so that they can use the knowledge we went through to hopefully have a successful project.

As a whole it would have been nice if we could have implemented our solution on hardware. If we were to move forward that would be the next step in making our project better. Moreover, if we had more time, we could have explored other kinds of types of algorithms or variations of the current algorithms; for instance, BCH could also be implemented with Berlekamp-Massey decoding algorithm or the Euclidean algorithm.

11.2 Lessons Learned

We would like to acknowledge the difficulties that can arise when working with a startup for the capstone project. While it gives a way to get started on a project quickly and has a good end product, a lot of the things that you wish were defined aren't exactly designed. There's also some miscommunication that is bound to occur that wouldn't happen if it was just your team. We appreciated the opportunity, but had to veer slightly away from the original intention of the project.

Another when working with these coding algorithms is that there is a very steep learning curve. The recommendation would be is to have more than preliminary knowledge before going into such a project. A majority of the semester was trying to understand the complex topics presented and not a lot of time or resources to do so.

A large lesson learned when gathering data would be to backup scripts and data for future use. We used a lot of python scripting in order to gather and plot data and some of it was lost amongst all of the other things we were doing. In the future it would be nice to have a well defined location to dump all of our data so we could find it if we needed it.

12 THANK YOU

We wanted to thank Professor Tze Meng Low and Mobolaji Bankole for the incredible support that they provided for our project throughout the term as well as Professor Bill Nace for serving as the Course Director and providing guidance through the hard times. Thank you as well to Raewyn Duvall for letting us work on the CubeRover project.

Table 1: Component Metrics

Component Name	Master Clock Frequency	Protocol	Transfer Speeds	Flash Memory Size	Processor	RAM Size
TI Hercules	180 Mhz	SPI	11-bit baud clock	1.25 Mb Integrated	ARM Cortex R4f	192 Kb
		UART	3.125 Mbps			
		I2C	10 - 400 Kbps			
Hercules Wifi Module	33 Mhz	UART	1 - 20 Mbps			
		WIFI	20 Mbps			
MSP430	8 Mhz Low Power - 25 Mhz	I2C	0 - 400 Kbps	10 Kb	MSP430	128 Kb
STM32	168 Mhz	SPI	42 Mbps	1 Mb	ARM Cortex R4	192 Kb
		UART	5.25 - 10.5 Mbps			
		I2C	100 - 400 Kbps			
STM32 Wifi Module	16 Mhz	WIFI	20 Mbps		ARM Cortex-M3	
Cyclone 10 LP FPGA	100MHz	SPI	50 Mbps		Nios II	270 Kb

Appendix A

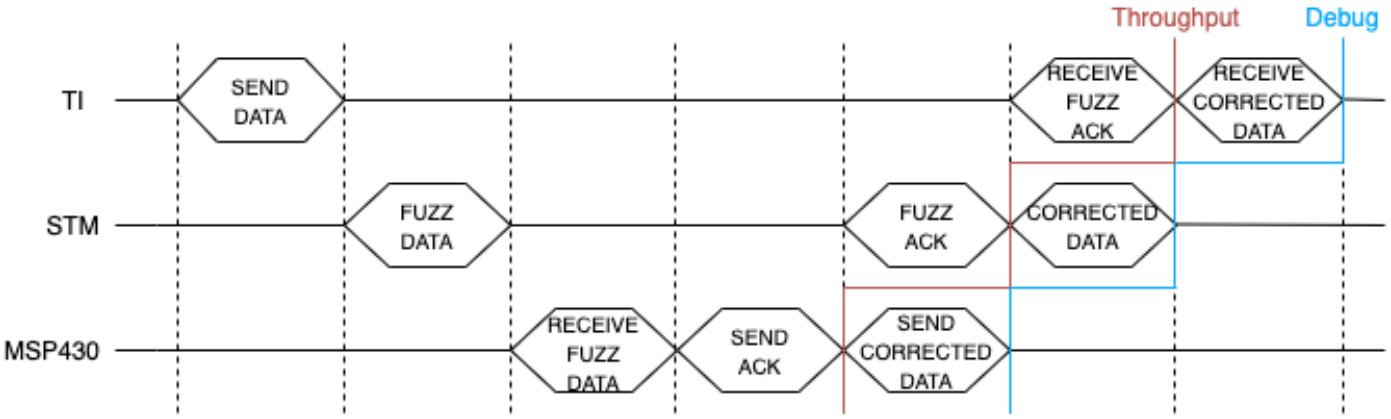


Figure 35: Verification Test Timing Diagram

12.1 Budget

Part	Function	Price
Microcontroller Discovery Board - STM32F407G-DISC1 X2	Microcontroller	$48.14
STM32F4DIS-WIFI - ADD ON BOARD, WIFI, STM32F4-DISCOVERY	Wifi Module- microcontroller	$30.60
MSP-EXP430F5529LP	MSP-430 - Watchdog	$13.49
MSP-EXP430F5529LP	MSP-430 - Motor	$13.49
LAUNCHXL2-RM46	Ti Hercules - Main Board	$51.76
AmazonBasics USB 2.0 Cable - A-Male to Mini-B Cord - 6 Feet (1.8 Meters)	usb to laptop	$5.99
cyclone 10 LP fpga	FPGA	$100.00
DKWF121 (Dev Kit)	Wifi Evaulation Board	$150.00
USB TO TTL SERIAL CABLE - DEBUG	USB to UART converter	$10.00

13 Sources

References

[1] http://the-art-of-ecc.com/

[2] https://github.com/satlab/bbctl

[3] https://www.infoworld.com/article/3250299/what-is-cython-python-at-the-speed-of-c.html

[4] https://docs.python.org/3/library/ctypes.html

[5] Maxino, T., Koopman, P. "The Effectiveness of Checksums for Embedded Control Networks," IEEE Trans. on Dependable and Secure Computing, JanMar 2009, pp. 59-72.

[6] John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science Engineering, 9, 90-95 (2007)

[7] https://ipnpr.jpl.nasa.gov/progress$_r eport/42-63/63H.PDF$

[8] https://www.cs.cmu.edu/ guyb/realworld/reedsolomon/reed$_s olomon_c odes.html$

[9] https://www.cs.cmu.edu/ venkatg/teaching/codingtheory/notes/notes6.pdf

[10] https://www.itu.int/wftp3/av-arch/video-site/h261/H261$_S pecialists_G roup/Contributions/476.pdf$

[11] http://web.mit.edu/6.02/www/f2010/handouts/lectures/L9.pdf

[12] https://ipnpr.jpl.nasa.gov/progress$_r eport/42-63/63H.PDF$

[13] BCM43362 Datasheet: https://www.cypress.com/file/297991/download

[14] https://bitbucket.org/icl/papi/wiki/Home

[15] https://doc.qt.io/qtforpython/api.html

[16] https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-451-principles-of-digital-communication-ii-spring-

[17] https://www.sciencedirect.com/topics/engineering/hamming-distance

[18] https://en.wikipedia.org/wiki/Hamming(7,4)

[19] https://www.isiweb.ee.ethz.ch/archive/massey_pub/pdf/BI321.pdf

[20] https://cse.buffalo.edu/faculty/atri/courses/coding-theory/lectures/lect4.pdf

[21] https://docs.python.org/3/library/threading.html

Image Size Equation: http://hyperloop.net/content/guides/how_big_is_an_image.html

Datasheets:
http://www.ti.com/lit/ug/spnu514c/spnu514c.pdf
https://www.silabs.com/documents/public/data-sheets/WF121-DataSheet.pdf
http://www.ti.com/lit/ds/symlink/msp430f5529.pdf
https://www.st.com/resource/en/datasheet/dm00037051.pdf

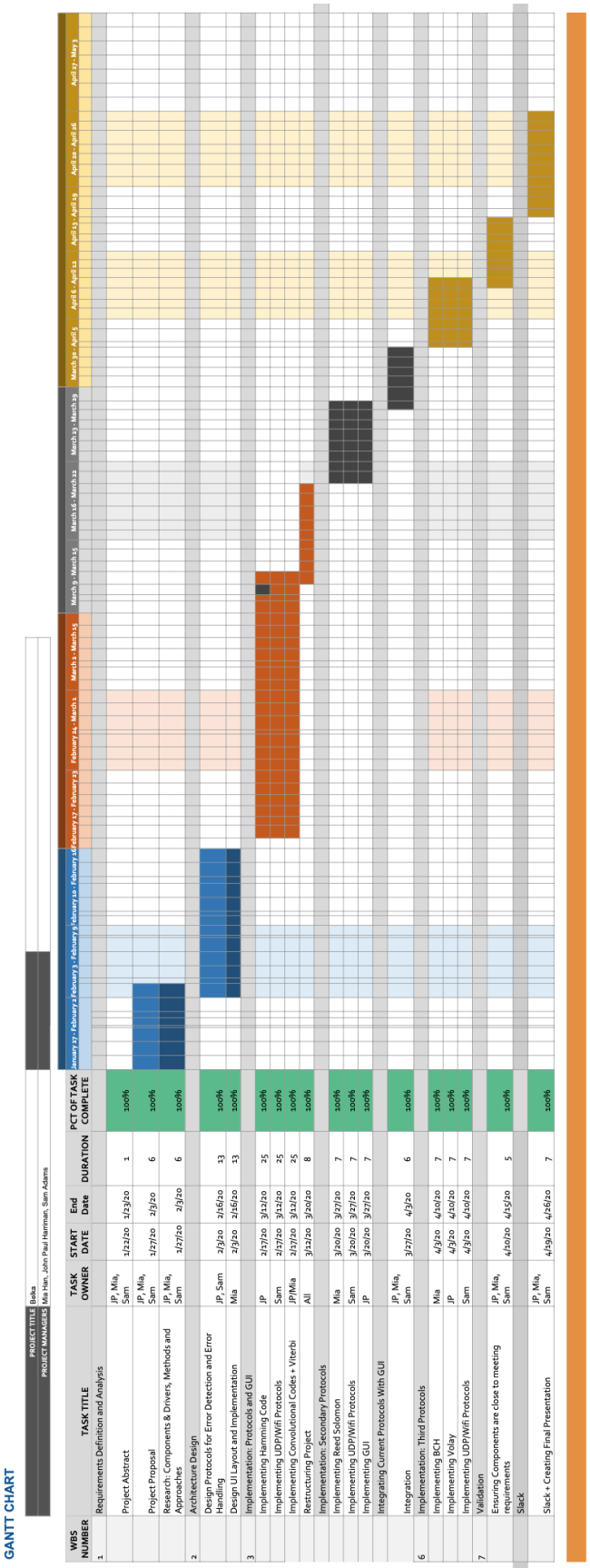


Figure 36: Gantt Chart