Project Belka

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

***Abstract*—A system capable of verifying data integrity on both serial and WiFi data transmission protocols. We want to be able to implement various communication protocols that can ensure complete data integrity, but we also want to create a system to make these protocols verifiable. We did this by designing a system that simulates errors that might happen during data transmission and we use those errors to verify that our protocols are working correctly.**

***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

Collaborating with Carnegie Mellon's CubeRover team, we are verifying the integrity of the data collected by the CubeRover in the domain of software and hardware. Currently, users of CubeRover have no way of checking the correctness of the collected data being sent to earth from the rover; the data may be corrupted by external extremities or harsh moon conditions. To solve this issue, we plan to design a verification protocol that determines how much data is corrupted; additionally, to test the protocols, we plan to simulate the effects of moon radiation on the currently running computers to handle any data loss with protocols including UART, UDP, Wifi, and QSPI.

We have more specifications listed below, but are general metrics are as follows. We must send all serial packets with error correction/detection within the limit of a timeout, 1 second. In terms of data recovery, we would like 100% recovery on all packets being sent serially and over Wifi. This includes both the error correction and detection portions of the protocols. We are aiming for a code rate of 5/7, but have a hard minimum of 1/4. For protocols with defined data packet sizes, we want an error correction of up to 75%. These are done through convolutional encoding and hamming encoding schemes.

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.

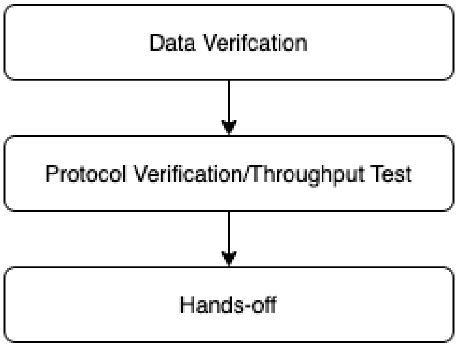


Figure 1: Qualitative approach to our requirements

Our main approach is a three step process where we can increase the level of trust that we have in our system to the point that it needs no manual intervention to run the protocols.

2.1 Data Verification

The data verification section is where we are able to run our error correction coding while testing that these encoding/decoding schemes are doing as expected. In figure 2, we will explain this verification method in more detail. The main idea comes in where we are passing back our corrected data through the same protocol, essentially creating two different modes of communication/computation. These modes are for now deemed - Debug mode and Throughput mode. During debug, we will be sending the corrected data and verifying on our intermediate microcontroller. Throughput mode is for the next stage.

2.2 Protocol Verification and Throughput Test

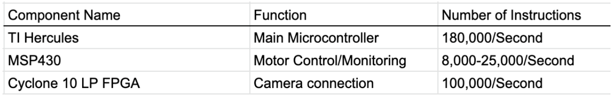
Now that we are sure that our error correction is working, we need to affirm that our protocol is sending what we are expecting. We can now stress test our protocol by continuously send packets through our intermediate micro-controller and limiting ourselves to the constraint of the timeout time. During this stage, we are able to verify that our protocols fit our new standard, while still being able to maintain the necessary constraints.

2.3 Hands-off

During the final part of our design, we would like to be able to remove the intermediate micro-controller and have the design only interact with the Graphical User Interface. We are calling this approach hands-off because now we will no longer be able to manipulate the data. This is the end-goal because this is the deliverable that we will be giving to CubeRover, and should run without any unnecessary hardware.

2.4 Hard Requirements

We have some defined hard requirements that we would like to meet based on our metrics from each hardware component. We are limiting ourselves to a **1 second cut-off** to send back an Acknowledged or Not Acknowledged packet after each serial protocol due to our motivation of our client.



Component Name	Function	Number of Instructions
TI Hercules	Main Microcontroller	180,000/Second
MSP430	Motor Control/Monitoring	8,000-25,000/Second
Cyclone 10 LP FPGA	Camera connection	100,000/Second

Figure 2: 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 $2\hat{3}1$ since the maximum file size that will be transferred is $2\hat{3}1$ 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 instructions, which will dictate the potential combinations of constraint lengths and code rates to consider.

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.

3 ARCHITECTURE OVERVIEW

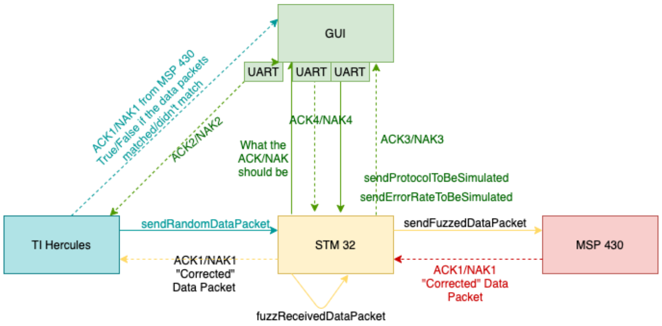

GUI
UART
UART
UART
ACK1/NAK1 from MSP 430 True/False if the data packets matched/didn't match
ACK2/NAK2
What the ACK/NAK should be
ACK4/NAK4
ACK3/NAK3
sendProtocolToBeSimulated
sendErrorRateToBeSimulated
TI Hercules
sendRandomDataPacket
STM 32
sendFuzzedDataPacket
MSP 430
ACK1/NAK1 "Corrected" Data Packet
ACK1/NAK1 "Corrected" Data Packet
fuzzReceivedDataPacket

Figure 3: An example transaction between our main host, the TI Hercules, the component, MSP 430, the data manipulator, STM32, and the Graphical User Interface

The GUI will send inputted data to the STM32 via functions sendProtocolToBeSimulated and sendErrorRateToBeSimulated; since this data is transmitted over UART, the STM 32 will send the GUI ACK/NAK (ACK3/NAK3 on the diagram) once it receives the configuration from the GUI. The TI Hercules will send a randomly generated data packet to the STM 32; The STM32 will then ”fuzz” the data (simulate the moon effects by flipping random bits/erasures) and send the modified to do the component (in this example, MSP 430) to test the protocol. The component will then send pack the ”corrected” packet of data and whether or not it deems it as an ACK/NAK

(ACK1/NAK1). The STM 32 will receive the "corrected" data packet and send it to the TI to compare it to the original packet of data that it generated. The STM32 will compare the ACK/NAK (ACK1/NAK1) to what ACK/NAK it expected to receive; the results of this comparison will be sent to the GUI (What the ACK/NAK should be) and since this information is transmitted over UART, the GUI will send an ACK/NAK (ACK4/NAK4) back to the STM 32. The TI Hercules will compare the original packet to the "corrected" data packet to verify if both packets match; it will send the result of this comparison to the GUI. Since this data is transmitted to the GUI over UART, the GUI will send an ACK/NAK (ACK2/NAK2) back to the TI Hercules.

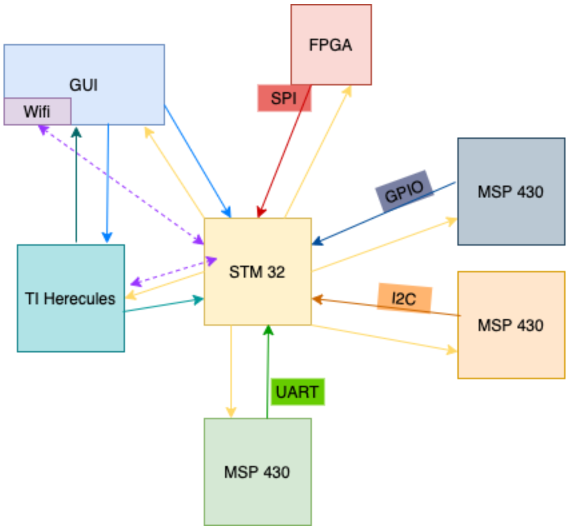
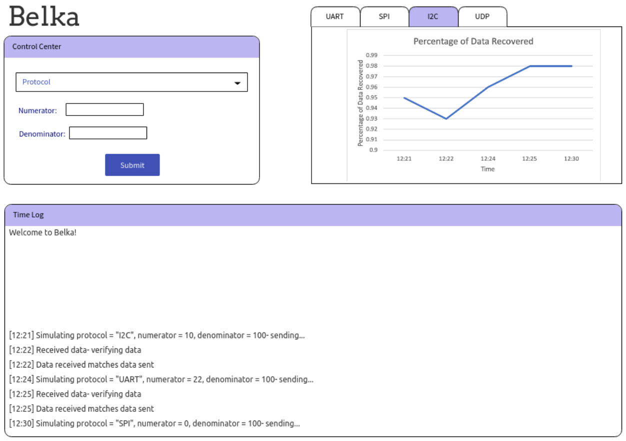


Figure 4: Overview of the Interactions Between All Devices

We decided to take this decentralized approach over a more centralized approach because it allows for better data validation and it allows us to take out the STM while the protocols still work. With the centralized approach we weren't able to obtain the corrected data packets in order to compare them with the packets that were corrupted. With this approach we were able to separate the tests into data validation and throughput testing in order to have a more complete verification suite. In order to do the data validation using this architecture we followed the timing diagram shown in Figure 13 for "Debug". During the "Debug" testing we send one packet at a time in order to send the corrected data after to confirm on the TI that the data is the same as when it was sent. On the centralized model we had before this would not have been possible as we had no device that was able to do that comparison. Once we are able to complete the debug testing without any errors we can move on to the "Throughput" testing also shown in Figure 13 which is the exact same as the debug tests except that the fixed packet is no longer sent back, to verify only the ACKs will be checked. With both the TI and the MCU/FPGA running the protocol without any spectial changes this design also allows us to pull out the STM in order to test the protocols without any fuzzing.

3.1 GUI



```
Welcome to Belka!

[12:21] Simulating protocol = "I2C", numerator = 10, denominator = 100- sending...
[12:22] Received data- verifying data
[12:22] Data received matches data sent
[12:24] Simulating protocol = "UART", numerator = 22, denominator = 100- sending...
[12:25] Received data- verifying data
[12:25] Data received matches data sent
[12:30] Simulating protocol = "SPI", numerator = 0, denominator = 100- sending...
```

Figure 5: Sample GUI screen

GUI Features

Control Center: The control center allows the user to configure the settings for protocol and error rate that they want to simulate; the configuration consists of selecting protocol (I2C, UART, SPI, and GPIO) and setting a numerator and denominator (must be greater than 0) for the error rate. Once the user submits the configuration, the GUI will send the configuration (sendProtocolToBeSimulated and sendErrorRateToBeSimulated) to the STM 32.

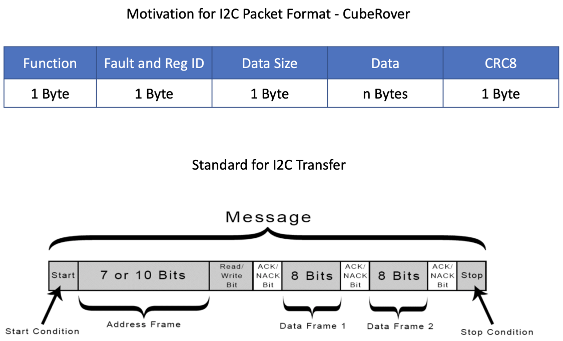
Real Time Log: The real time log displays and notifies the user of all data received and sent from the STM 32. The real time log will confirm that the configuration is sent to the STM 32, whether or not the "corrected" data packet matched the original data packet, and whether or not the ACK/NAK sent by the protocol is correct/expected.

Real Time Line Graph: The real time line graph displays a time vs. percentage of data packets corrected for each protocol. The tabs on the top of the display allow the users to switch between different protocols to view the percentage of data recovered per protocol; the real time line graph shows and verifies the packet recovery accuracy rate of our protocols.

4 DESIGN TRADE STUDIES

4.1 I2C and UART

The packet format for I2C and UART are very similar in structure with both requiring a small amount of bits (8) 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.

Consider 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.

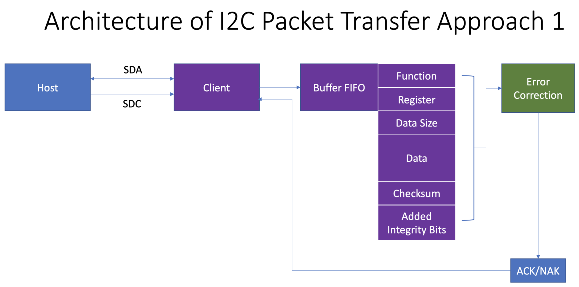


Figure 7: First Approach to I2C

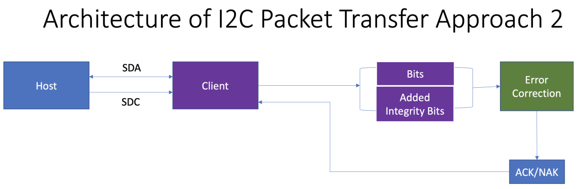


Figure 8: Second Approach to I2C

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.

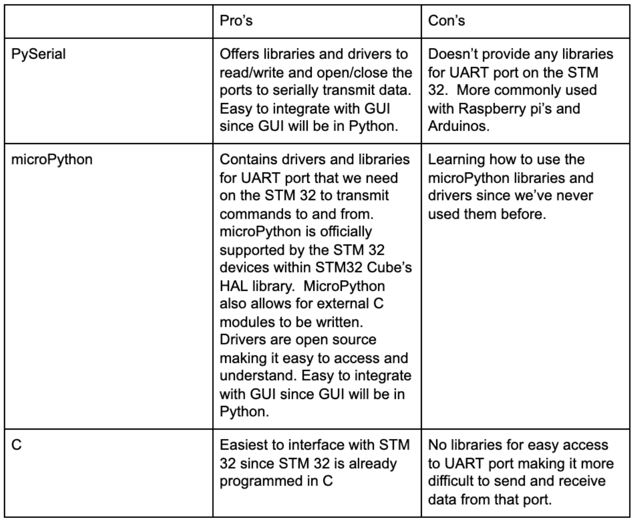
4.2 SPI and GPIO

The implementation will ultimately use convolutional codes and Viterbi primarily because of the amount of memory that convolutional codes and Viterbi utilize. Since the CubeRover has at most 2GB of memory allocated for data storage, minimizing the amount of data that the algorithm will utilize is imperative. Turbo codes use four times the amount of convolutional codes compared to convolutional codes + viterbi, and Reed-Solomon stores parity symbols that could take up to the same amount of bits as the chunk of bits, doubling the amount of memory used. The amount of memory convolutional codes and Viterbi will take up is at most the same amount of memory as the input data in addition to the amount of memory for the input data; however, it uses the fewest number of processes/instructions to detect errors in the stream of bits. Turbo codes will pass the data through four times the number of convolutional codes as convolutional codes + viterbi, and Reed-Solomon four processes (source encoder, encryption, channel encoder, and modulator) for each iteration of encoding; contrarily, convolutional codes + viterbi only use two processes to encode the data. Thus, because it uses one of the least amount of memory and least amount of processes, convolutional codes + viterbi best fit our solution approach.

4.3 WiFi

When designing the WiFi protocol there were a couple of trade offs that we had to decide on. One option was to implement the entire TCP protocol over UDP, including the 3-way handshake. This turned out not to be the best option as the handshake could be eliminated as we are always only talking between 2 nodes. This decreases the time needed to setup the connection by a lot. This also decreases the header overhead in each packet as we were able to make them fairly small.

4.4 GUI



	Pro's	Con's
PySerial	Offers libraries and drivers to read/write and open/close the ports to serially transmit data. Easy to integrate with GUI since GUI will be in Python.	Doesn't provide any libraries for UART port on the STM 32. More commonly used with Raspberry pi's and Arduinos.
microPython	Contains drivers and libraries for UART port that we need on the STM 32 to transmit commands to and from. microPython is officially supported by the STM 32 devices within STM32 Cube's HAL library. MicroPython also allows for external C modules to be written. Drivers are open source making it easy to access and understand. Easy to integrate with GUI since GUI will be in Python.	Learning how to use the microPython libraries and drivers since we've never used them before.
C	Easiest to interface with STM 32 since STM 32 is already programmed in C	No libraries for easy access to UART port making it more difficult to send and receive data from that port.

Figure 9: GUI tradeoffs between Pyserial, microPython, and C

5 SYSTEM DESCRIPTION

5.1 I2C and UART Protocol

Our verification method of our data bits, we will be using Hamming Encoding.

This is our starting approach to the protocols. With four added parity bits we should be able to implement a 2-bit error correction method and 3-bit error detection.

The justification for this method is that the algorithm to run on it on is lightweight and constant in computation time. This will eventually be replaced by a more complex algorithm, but for now we can use it as proof of concept while we do more research.

We can't use Turbo Encoding or Low-Density Parity Coding because both of these require a byte count of over 1000 to become efficient computationally, this is the reason why we must stick to either Hamming Encoding or Reed-Solomon. The initial problem with Reed-Solomon for our main approach is that we need to allocate a significant portion of memory to get it running at all. We will later test to see if this is true.

However, for now we can calculate the code rate for hamming encoding which should meet our specification.

(Added integrity) $r = 4$
(Block Length) $n = 2^4 - 1 = 15$
(Message Length) $k = 15 - 4 - 1 = 10$
(Code Rate) $R = k/n = 10/15 = 2/3$

This puts our code rate at 2/3 which is well about our means.

5.2 SPI and GPIO Protocol

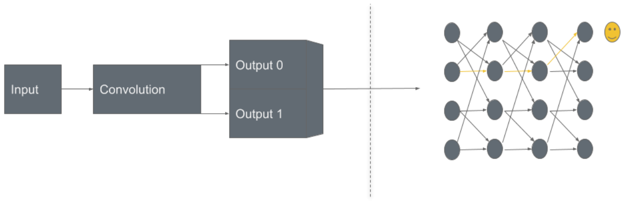


Figure 10: Overview of Viterbi Algorithm when code rate = 1/2 and constraint length = 3

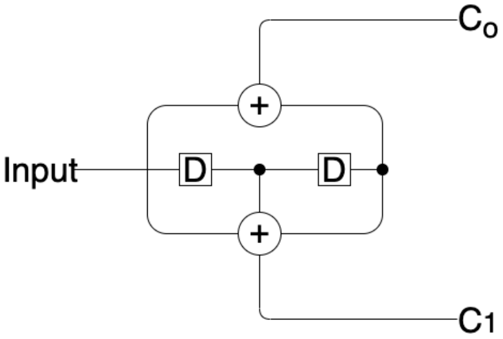


Figure 11: Diagram of Convolution Registers when code rate = 1/2

To verify whether or not streams of bits are altered or affected by noise during transmission, we plan on implementing convolutional codes and the viterbi algorithm. The algorithm is comprised of two parts: a convolutional encoder composed of shift registers with code generator polynomials and the trellis, part of the viterbi stage.

Convolutional Codes and Viterbi Encoding
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

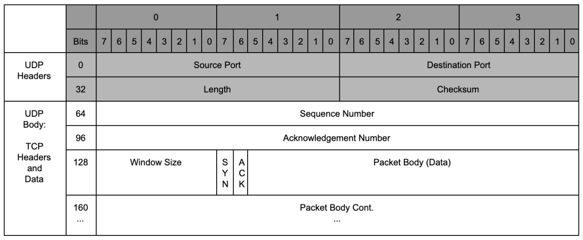
pass the stream of bits through a series of polynomials (XOR's) and shift registers. The polynomials reflect the convolutional encoder behavior and define the number of states in the convolutional encoder. The polynomials and shift registers combined create a state machine for input data, current state, 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.

Convolutional Codes and Viterbi Decoding
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.

Convolutional Code + Viterbi Algorithm Variables
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, we've decided to test the following code rate and constraint length 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). Once n (the denominator in the code rate, 1/n) exceeds 4, research has shown that the gain with code rates with values greater than 4 does not yield much higher gains than the values from 2-4; thus, any code rate smaller than is not worth pursuing . Additionally, with these combinations of code rates and constraint lengths, after calculating the number of instructions for each combination, we concluded that each combination is achievable under the maximum number of instructions from the Cyclone 10 LP FPGA.

5.3 WiFi Protocol

Since UDP does not guarantee packet delivery we needed to find a way to ensure packet delivery while also maintaining the UDP structure that CubeRover requires. The solution for this is to build a pseudo TCP protocol inside the UDP packet. Essentially this means that we will be putting the TCP headers that are needed for the protocol in the body of the UDP packet. This will allow us to acknowledge receipt of packets using ACK packets. In order to improve transmission speed we also decided to implement windowing so we can acknowledge multiple packets at a time increasing our throughput.



		0								1								2								3							
	Bits	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
UDP Headers	0	Source Port																Destination Port															
	32	Length																Checksum															
UDP Body: TCP Headers and Data	64	Sequence Number																															
	96	Acknowledgement Number																															
	128	Window Size								SYN	ACK	Packet Body (Data)																					
	160 ...	Packet Body Cont. ...																															

Above is the packet structure. In addition to the standard UDP headers it includes sequence numbers, acknowledgement numbers, Window size and SYN/ACK flags.

In order to keep packets in order each SYN packet will be assigned an ascending sequence number that will be put in the ”TCP” header of the packet. This ensures that even if the packets arrive to the destination out of order they will be able to be placed back in order. Now to ensure receipt of the packets on the destination side every **n** packets (where n = window size), the destination node will send an ACK packet with an acknowledgement number the same as the last packet received. This ACK represents a confirmed receipt of the packets. An example of a window size of 4 is shown in figure 12.

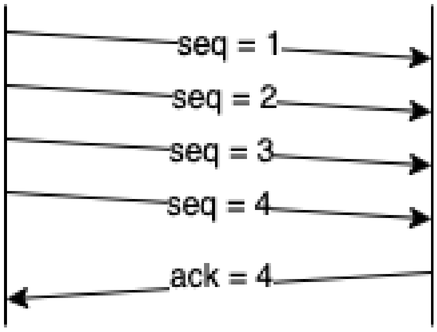


Figure 12: Example TCP Window

As we can see after the first 4 packets are sent a response ACK is sent back to the source to notify it of it's receipt. If the ACK is not recieved the entire window will have to sent again until the ACK is recieved.

6 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.

7 VALIDATION

7.1 Verifying Latency Requirements

TI, MSP 430, and STM 32: The general approach for determining the latency of these devices is to analyze the assembly dump of the protocols and using the specifications of each device, count the number of clock cycles each instruction uses. The total number of clock cycles times the core frequency will give the amount of time that it will take to run the protocol.

STM32: To verify that we are meeting the timing requirements, we plan to also use two other methods:

Oscilloscope via an unused pin: At the beginning of the task, an unused pin would be set to high. Once the task is completed, the pin would be set to low right after; this method is least prone to errors.

Hardware timers: Hal_tick/sys_tick, TIMx HAL library provides sys_tick_handler functions as a base time reference to measure the latency. Using these timers, we can count the number of system clock cycles.

7.2 Verifying Power Requirements

The driver libraries that will be used for each component offer libraries that can verify the power requirements of the components.

8 PROJECT MANAGEMENT

8.1 Schedule

Looking at the Gantt chart (Figure 14), we are currently working on implementing the protocols and standards. We've allocated three and half weeks to implementing the algorithms because first, we wanted to be confident in the algorithms that we are implementing by thoroughly researching them and second, since we've never implemented or used these algorithms, we are currently undergoing a learning curve of figuring out these algorithms.

As we described in the beginning of the document, we will verify that the standards are working. Before we begin putting the components together, the individual components must be tested individually to ensure that we can correctly send data in-between components. Once we verify that the standards are working and correct, we will verify that the protocols that are wrapped around the standards are working correctly. Once we can prove that protocols are working properly, we will remove the micro-controller and let the packet generation run by itself and send updates up to the GUI through the TI micro-controller

8.2 Team Member Responsibilities

We were able to split our work fairly easily. Mia will be responsible for the GPIO and SPI protocols and with a secondary responsibility to the GUI. John Paul will be responsible for the I2C and UART protocols as well as working on the verification tests. Sam will focus on the WiFi protocol as well as the testing routines that will be validating all of the protocols.

8.3 Risk Management

Our current risks are determining the trade-offs between custom hardware and running our code on the specific software components. In industry, many of these coding algorithms are optimized to be run on custom hardware where only the necessary critical paths are used. With these microcontrollers, we might not meet the timing needs because we have to deal with cases such as software interrupts, scheduling, etc.

Other risks that we are currently trying to mitigate is how best to define GPIO. Because there is no defined protocol - as it is literally general purpose. We need to either define what it means or try to use CubeRover's code as our use-cases and write wrappers around those.

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

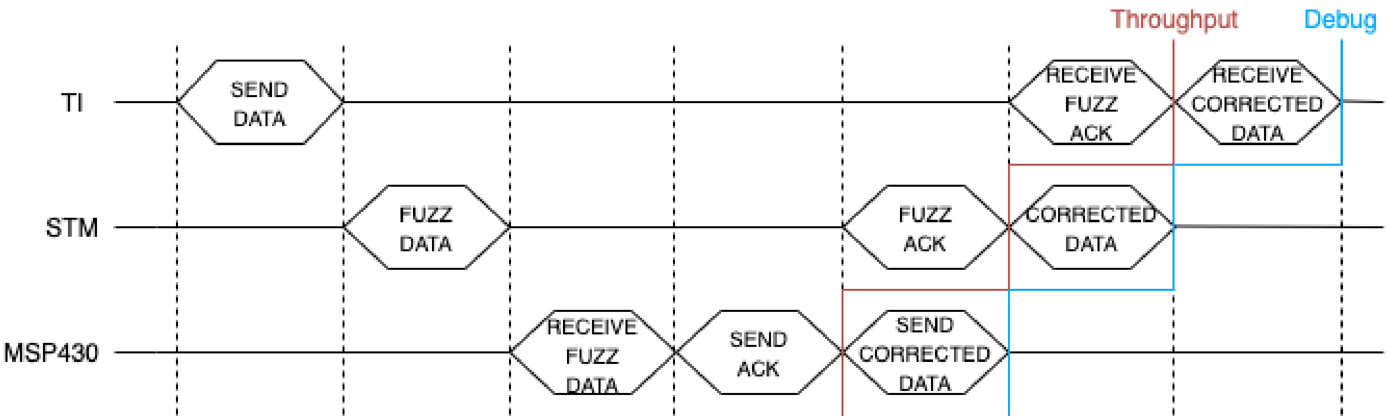
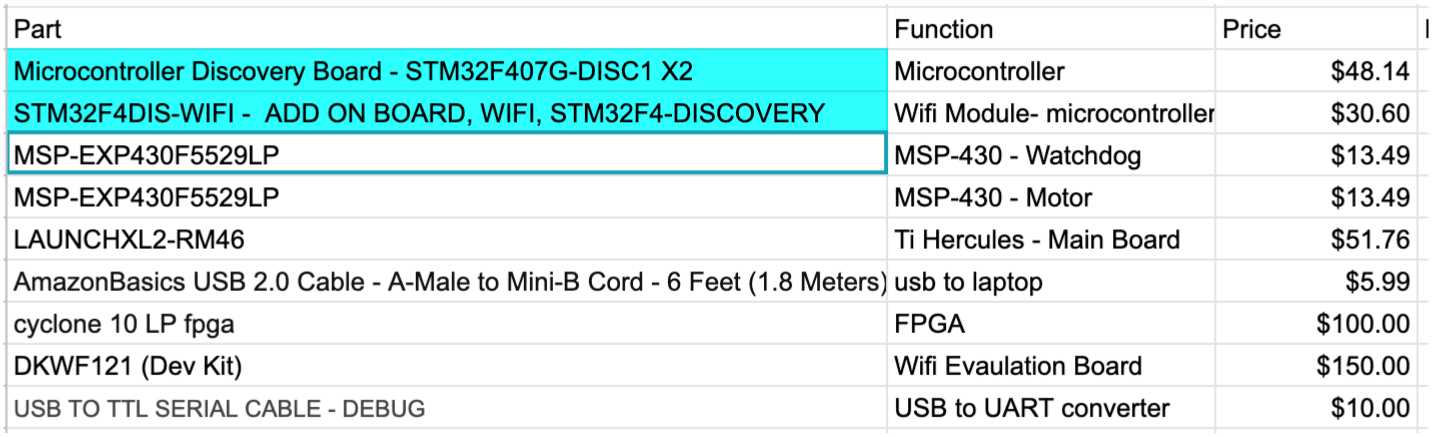


Figure 13: Verification Test Timing Diagram

8.4 Budget



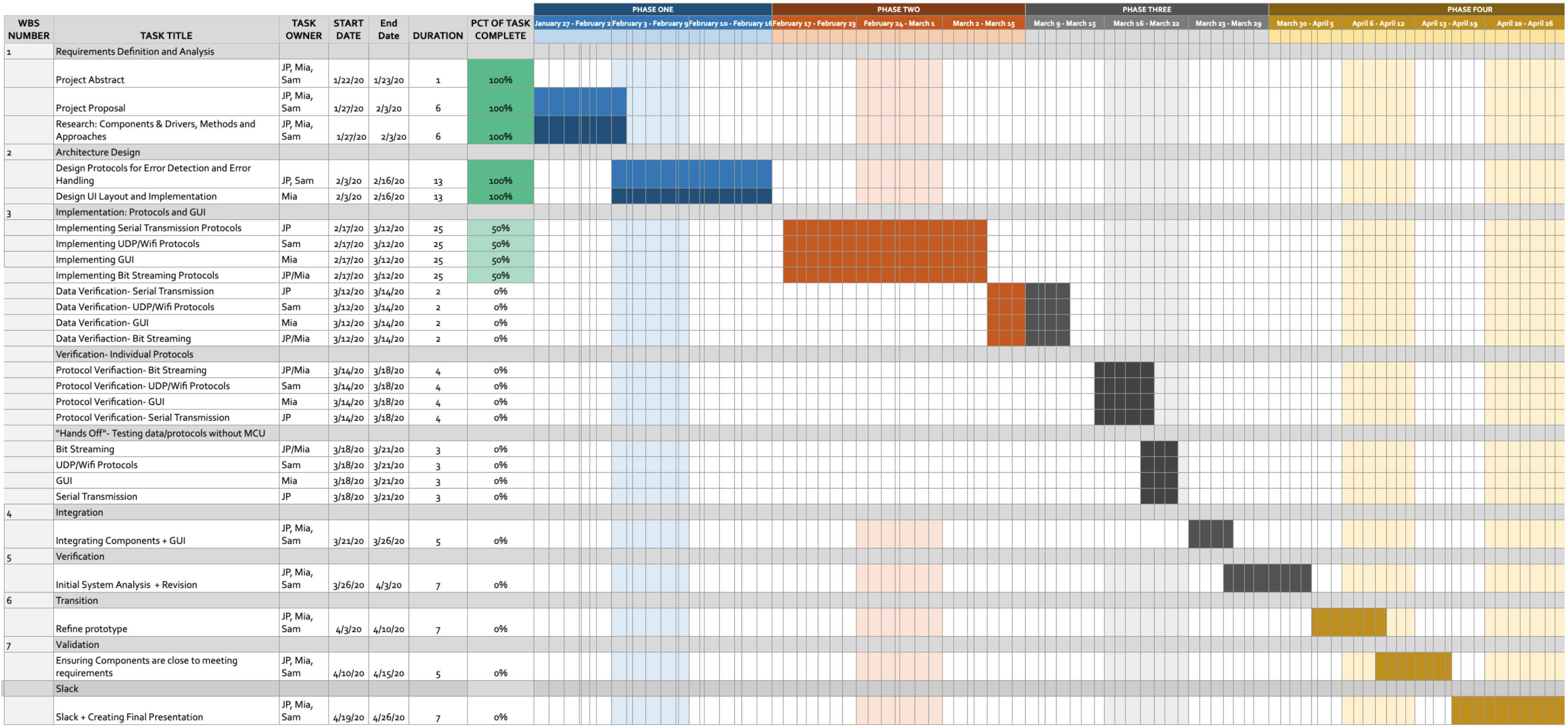
Part	Function	Price
Microcontroller Discovery Board - STM32F407G-DISC1 X2	Microcontroller	$48.14
STM32F4DIS-WIFI - ADD ON BOARD, WIFI, STM32F4-DISCOVERY	Wifi Module- microcontroller	$30.60
MSP-EXP430F5529LP	MSP-430 - Watchdog	$13.49
MSP-EXP430F5529LP	MSP-430 - Motor	$13.49
LAUNCHXL2-RM46	Ti Hercules - 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

9 Sources

Viterbi + Convolutional Code Sources:
http://web.mit.edu/6.02/www/s2012/handouts/8.pdf
https://www.design-reuse.com/articles/21107/viterbi-algorithm.html
https://ieeexplore-ieee-org.proxy.library.cmu.edu/stamp/stamp.jsp?tp=arnumber=255163
https://www.cs.cmu.edu/ guyb/realworld/reedsolomon/reed_solomon_codes.html
http://kom.aau.dk/group/05gr943/literature/print/turbo_tutorial.pdf
https://www.mathworks.com/help/comm/ref/turboencoder.html
http://www.micromouseonline.com/2016/02/02/systick-configuration-made-easy-on-the-stm32/

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



WBS NUMBER	TASK TITLE	TASK OWNER	START DATE	End Date	DURATION	PCT OF TASK COMPLETE
1	Requirements Definition and Analysis					
	Project Abstract	JP, Mia, Sam	1/22/20	1/23/20	1	100%
	Project Proposal	JP, Mia, Sam	1/27/20	2/3/20	6	100%
	Research: Components & Drivers, Methods and Approaches	JP, Mia, Sam	1/27/20	2/3/20	6	100%
2	Architecture Design					
	Design Protocols for Error Detection and Error Handling	JP, Sam	2/3/20	2/16/20	13	100%
	Design UI Layout and Implementation	Mia	2/3/20	2/16/20	13	100%
3	Implementation: Protocols and GUI					
	Implementing Serial Transmission Protocols	JP	2/17/20	3/12/20	25	50%
	Implementing UDP/Wifi Protocols	Sam	2/17/20	3/12/20	25	50%
	Implementing GUI	Mia	2/17/20	3/12/20	25	50%
	Implementing Bit Streaming Protocols	JP/Mia	2/17/20	3/12/20	25	50%
	Data Verification- Serial Transmission	JP	3/12/20	3/14/20	2	0%
	Data Verification- UDP/Wifi Protocols	Sam	3/12/20	3/14/20	2	0%
	Data Verification- GUI	Mia	3/12/20	3/14/20	2	0%
	Data Verifiaction- Bit Streaming	JP/Mia	3/12/20	3/14/20	2	0%
	Verification- Individual Protocols					
	Protocol Verifiaction- Bit Streaming	JP/Mia	3/14/20	3/18/20	4	0%
	Protocol Verification- UDP/Wifi Protocols	Sam	3/14/20	3/18/20	4	0%
	Protocol Verification- GUI	Mia	3/14/20	3/18/20	4	0%
	Protocol Verification- Serial Transmission	JP	3/14/20	3/18/20	4	0%
	"Hands Off"- Testing data/protocols without MCU					
	Bit Streaming	JP/Mia	3/18/20	3/21/20	3	0%
	UDP/Wifi Protocols	Sam	3/18/20	3/21/20	3	0%
	GUI	Mia	3/18/20	3/21/20	3	0%
	Serial Transmission	JP	3/18/20	3/21/20	3	0%
4	Integration					
	Integrating Components + GUI	JP, Mia, Sam	3/21/20	3/26/20	5	0%
5	Verification					
	Initial System Analysis + Revision	JP, Mia, Sam	3/26/20	4/3/20	7	0%
6	Transition					
	Refine prototype	JP, Mia, Sam	4/3/20	4/10/20	7	0%
7	Validation					
	Ensuring Components are close to meeting requirements	JP, Mia, Sam	4/10/20	4/15/20	5	0%
	Slack					
	Slack + Creating Final Presentation	JP, Mia, Sam	4/19/20	4/26/20	7	0%

Figure 14: Gantt Chart