***Wireless Rover Navigation***

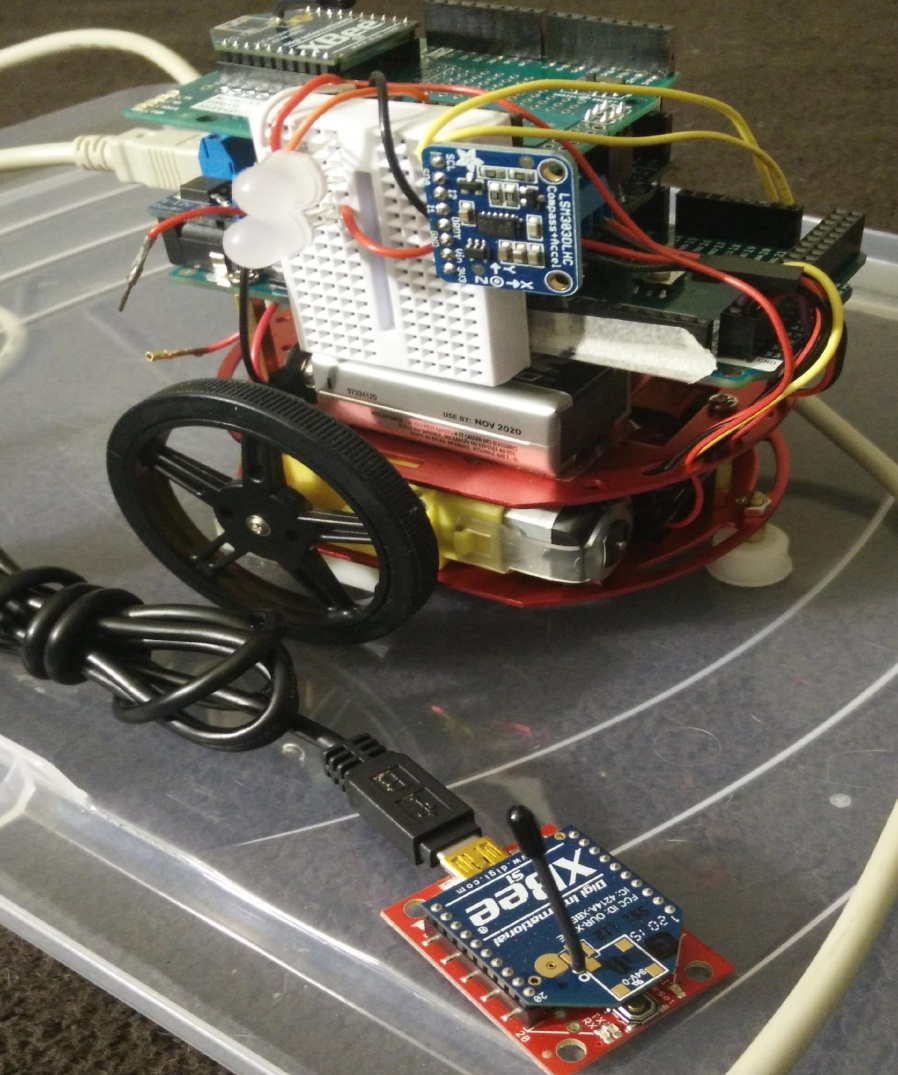
***An Embedded Systems Project***

*CSS 427 - Fall*

December 10th, 2016  
*Team: Embedded\_****RR***

Members:

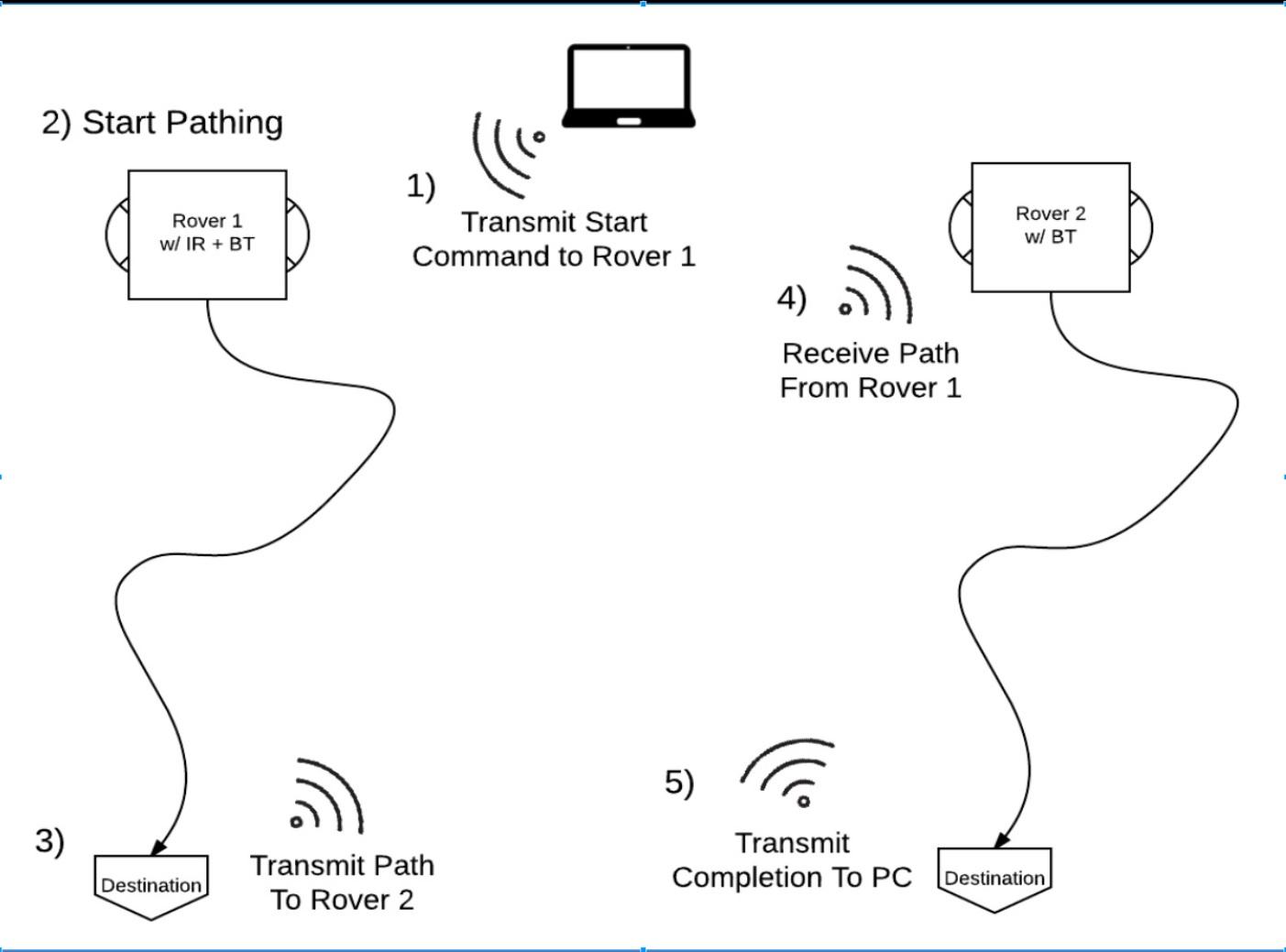
***R***yu Muthui and ***R***obert Griswold



***Program Description:***

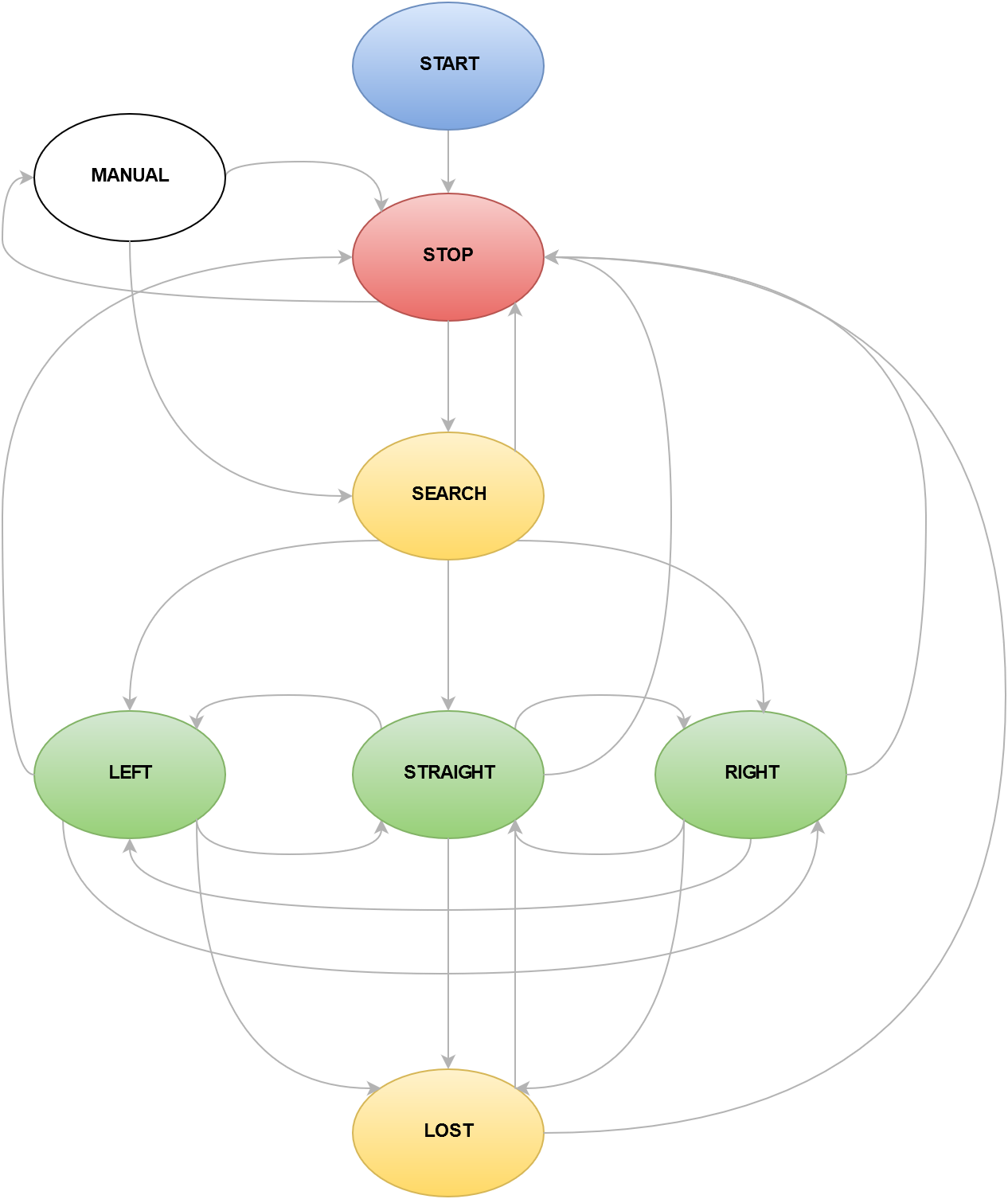
Our design philosophy was to take the simple approach. The overall project was to have two controllable rovers. Although line following robots are a common project that has been done many times, the requirement of the project of having wireless communication made this common project a much more fun yet challenging program. The idea was to implement our own line following rover (master) and collect data on traversal and navigation information. Once this data is captured, it can send it to the second rover (slave) and have it mimic the master rover’s movement.

The overview of the program is as follows: The master rover is positioned near the course. The PC will then send a command for it to start. It will then navigate forward until it detects a line and begin to follow the line. While traversing, it will collect navigation data, encode it as 7 byte navigation packet, load it into a 100 byte payload for xbee transmission. When the payload is at least half full, it will attempt to send the payload to the Slave rover, only clearing its buffer on a successful transmission. The slave rover will also be positioned on an identical course next to the Master Rover. It will accept as many navigation packets as it can, waiting for the PC to send a command to start. After the signal, the Slave rover will synchronize its clock with the first navigation packet, and then execute them in order to mimic the timing and speed of the master rover. The master rover will continue to follow the line until it is lost or it receives a signal from the PC to stop.

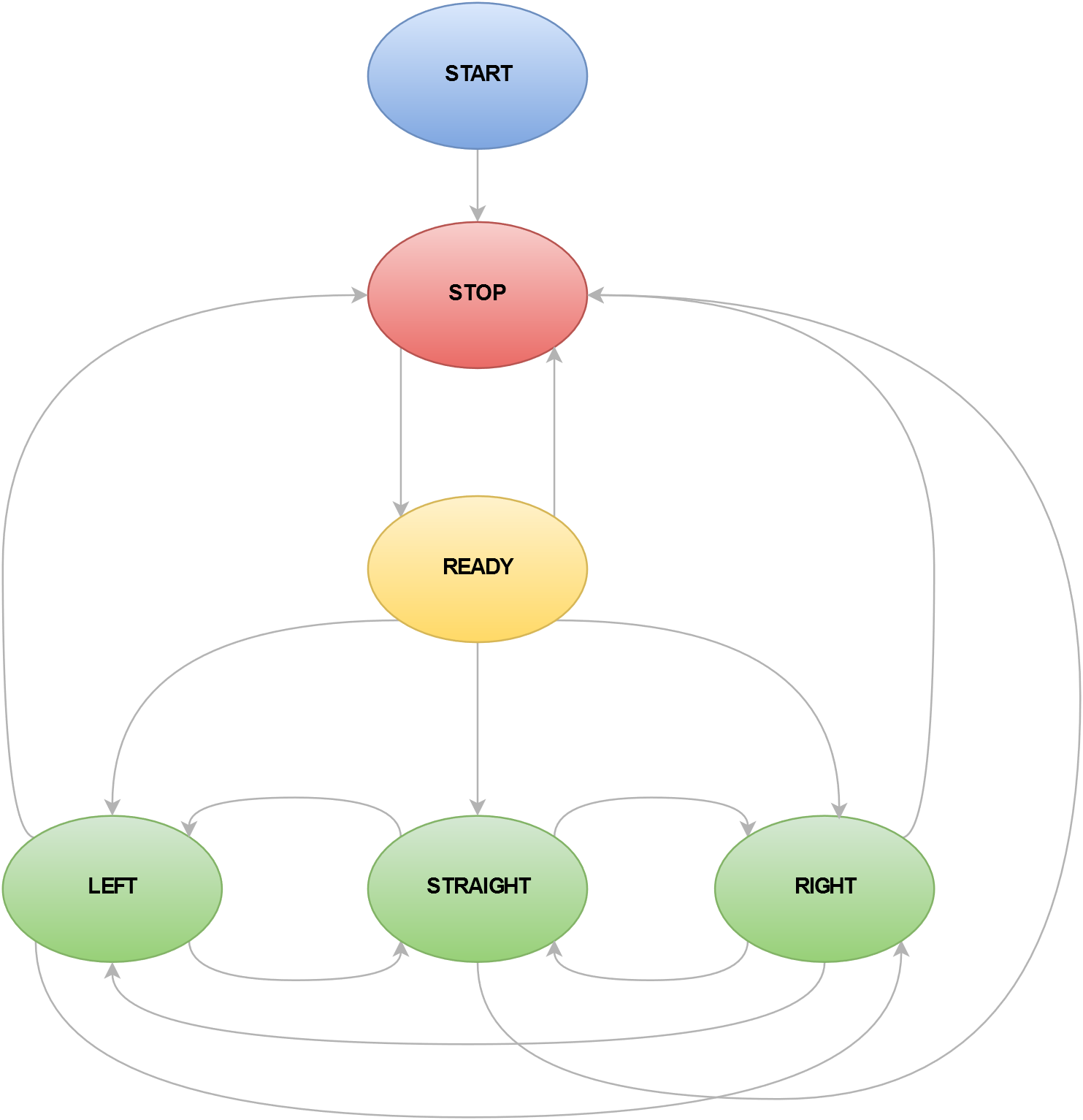
The figure above is an overview of how the system works. The Master rover’s logic is modeled after a finite state machine and is designed so that it can continuously traverse the line course autonomously until commanded to stop or has lost the line.

The following are flowcharts and diagrams for the project:

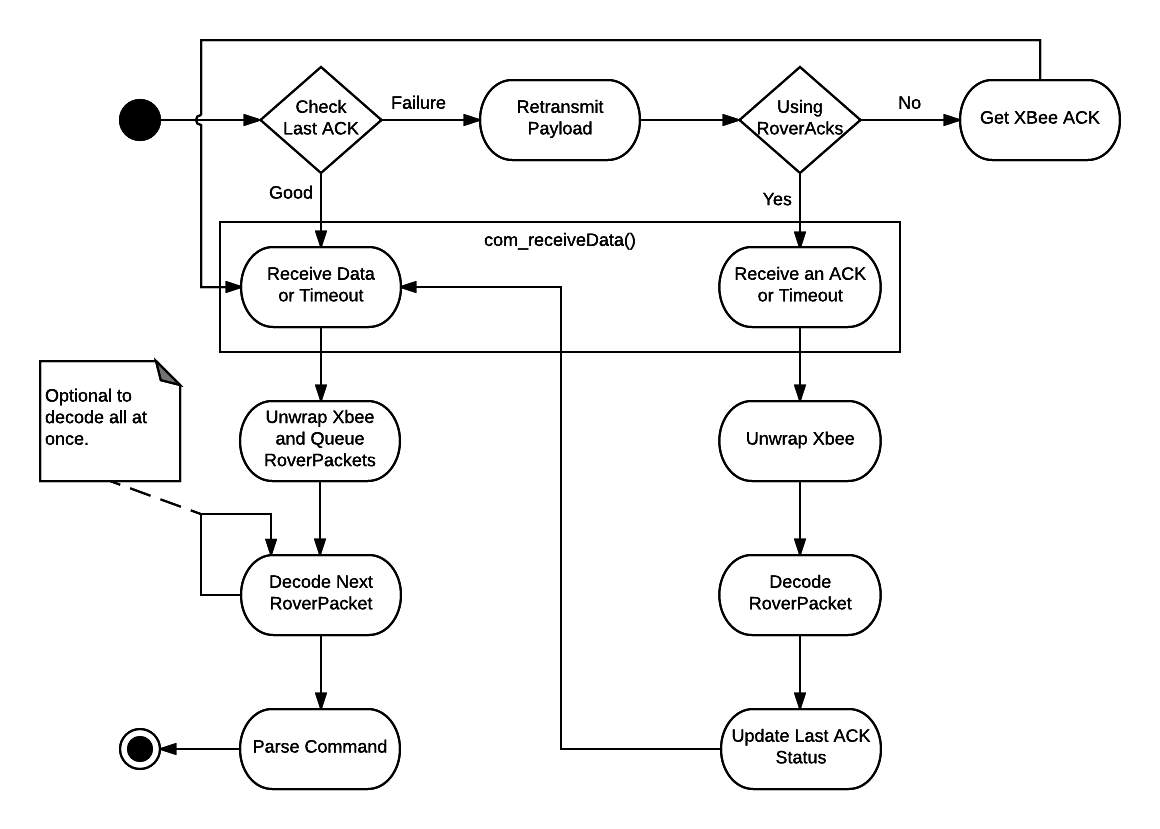
Finite State Machine for the Master Rover



Finite State Machine for the Slave Rover



Sequence diagram of how the XBee communication occurs between the rovers and the PC:

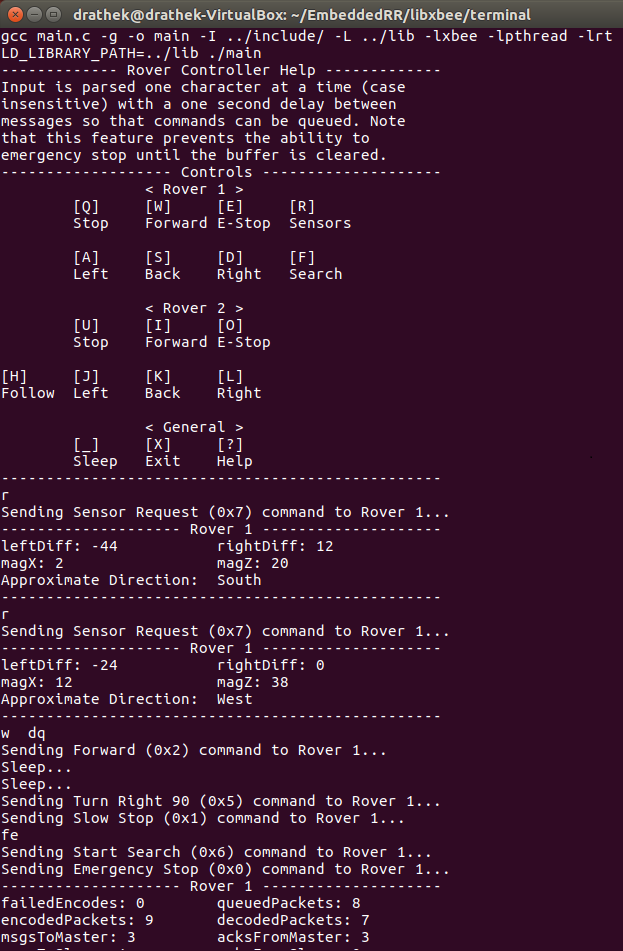


***Specification:***

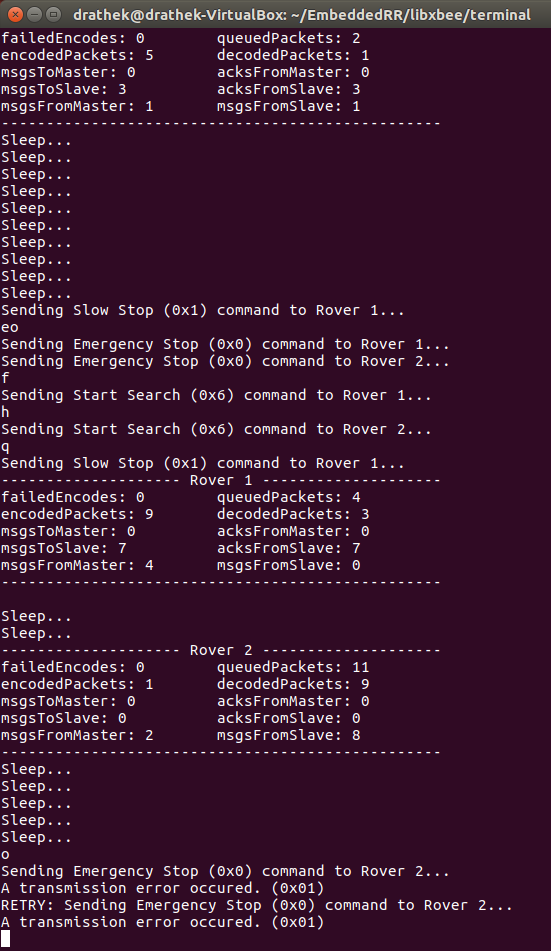
RoverPacket commands:

|  |  |  |  |
| --- | --- | --- | --- |
| 0000 (0x0) | Emergency Stop | 1000 (0x8) | Mag Sensor Data |
| 0001 (0x1) | Slow Stop | 1001 (0x9) |  |
| 0010 (0x2) | Forward | 1010 (0xA) | Navigation Data/ACK |
| 0011 (0x3) | Backward | 1011 (0xB) | Packets Encoded/Decoded |
| 0100 (0x4) | Turn left 90 | 1100 (0xC) | Msgs from Master/Slave |
| 0101 (0x5) | Turn right 90 | 1101 (0xD) | Msgs to Slave/Acks from Slave |
| 0110 (0x6) | Start Search/Follow | 1110 (0xE) | Msgs to Master/Acks from Master |
| 0111 (0x7) | IR Sensor Data/Request | 1111 (0xF) | Failed Encodes/Packets Queued |

Example command inputs from the terminal program and rover responses:



Example of the statistical information displayed at the end of a navigation session:



***Exception Report:***

We set out with the goal of getting a second rover, without any sensors, to be able to mimic another rover. Many hardware factors cause the Slave rover to steer off course quickly, and since it has no sensors, it has no ability to detect and compensate for error. Some factors include inconsistent speed on each of the motors, weight differences, and battery voltages. Small errors, like a different turning speed, quickly accumulate to get a blind rover far off course.

Low speeds, generally between 0-20 (out of 255) usually will cause a stall, thus limiting our ability to slow the rover. Although we avoided the range, it would still create inconsistencies when reversing the direction of a motor. Another factor contributing to errors is the weight difference between the Slave and Master. The Slave rover is significantly lighter as it doesn’t have the extra wiring, breadboard, and sensors resulting in a lighter, and faster rover.

The final major limitation we faced in the project was the available memory. After the code is uploaded, both rovers had about 7,000 bytes of memory available for local variables. We created a RoverPacket as small as possible to save space. However, with a size of only 7 bytes per packet, a rover would stack overflow at approximately 512 packets requiring a manual reset to continue.

***Team Assignments and Report:***

Building the Rovers was relatively easy and simple and hardware work was split evenly. The rovers required soldering and wiring which sometimes took two pairs of hands to construct. Additionally, a large portion of the code generation was performed using pair programming.

The workload was split up as follows:

Robert Griswold

* State Machine (12.5%)
* Communication Library (37.5%)
* Libxbee Integration (12.5%)

Ryu Muthui

* Sensor Library (12.5%)
* Motor Library (12.5%)
* Lights Library (12.5%)

***Public Libraries Used:***

* Arduino XBee Library by *Andrew Rapp*
* Arduino QueueArray Library by *Efstathios Chatzikyriakidis*
* Arduino Adafruit\_MotorShield Library
* Arduino Adafruit\_Sensor Library
* Libxbee - C/C++ Library by *Attie Grand*