RSL autonomous rover

Cris Madrigal  
Santa Clara University

Jiachi Zhang  
Santa Clara University

Nicholas Peacock  
Santa Clara University

Brogan O’Hara  
Santa Clara University

Follow this and additional works at: https://scholarcommons.scu.edu/idp_senior

Part of the Computer Engineering Commons, Electrical and Computer Engineering Commons, and the Mechanical Engineering Commons

Recommended Citation

Madrigal, Cris; Zhang, Jiachi; Peacock, Nicholas; and O’Hara, Brogan, "RSL autonomous rover" (2015). Interdisciplinary Design Senior Theses. 13.  
https://scholarcommons.scu.edu/idp_senior/13

This Thesis is brought to you for free and open access by the Engineering Senior Theses at Scholar Commons. It has been accepted for inclusion in Interdisciplinary Design Senior Theses by an authorized administrator of Scholar Commons. For more information, please contact rsrogin@scu.edu.
I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Cris Madrigal
Jiachi Zhang
Nicholas Peacock
Brogan O'Hara

ENTITLED

RSL Autonomous Rover

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREES OF

BACHELOR OF SCIENCE IN ELECTRICAL ENGINEERING
BACHELOR OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING
BACHELOR OF SCIENCE IN MECHANICAL ENGINEERING

Thesis Advisor

Thesis Advisor

Department Chair

Department Chair

Department Chair
RSL Autonomous Rover

by

Cris Madrigal
Jiachi Zhang
Nicholas Peacock
Brogan O’Hara

Submitted in partial fulfillment of the requirements
for the degrees of
Bachelor of Science in Electrical Engineering
Bachelor of Science in Computer Science and Engineering
Bachelor of Science in Mechanical Engineering
School of Engineering
Santa Clara University

Santa Clara, California
June 9, 2015
ABSTRACT

Autonomous vehicles are useful for a variety of applications such as military, urban, and agricultural environments. This paper discusses adding an autonomous navigation system to an all-terrain vehicle by implementing controllers that interface with its current system, installing sensors on the vehicle for obstacle detection, and developing effective safety mechanisms to prevent injury to others. The result is a vehicle capable of waypoint navigation and obstacle avoidance. Testing the vehicle showed that the LIDAR and the autonomous navigation system were integrated seamlessly, and that the sensor output signals were successfully translated into vehicle commands the existing system uses. This system could be improved with further tuning of the PID controller to prevent a large deviation from the defined path. The LIDAR could also be programmed to allow the vehicle to navigate around the obstacle instead of stopping in front of it.
# Table of Contents

1 Introduction ........................................ 1  
   1.1 Literature Review ................................... 3  
   1.2 Problem Statement .................................. 5  

2 System-Level Design ................................ 6  
   2.1 Vehicle Background ................................ 6  
   2.2 Overview of the Project ............................ 7  
   2.3 Customer Definition and Needs ..................... 9  
   2.4 System Level Requirements ....................... 10  
   2.5 Functional Analysis ................................ 10  
   2.6 System Level Issues and Trade-offs ............... 11  
   2.7 Team and Project Management ..................... 11  
      2.7.1 Initial Team Goals ............................. 11  
      2.7.2 Mitigating Project Challenges ................. 12  
      2.7.3 Timeline ....................................... 12  
      2.7.4 Risks and Mitigations ......................... 13  

3 Sensors .............................................. 14  
   3.1 Role and Requirements ............................. 14  
   3.2 Options and Trade-off Analysis ................... 14  
      3.2.1 SICK - LMS 111 ................................ 14  
      3.2.2 Velodyne VLP-16 ............................. 15  
      3.2.3 Camera ......................................... 15  
      3.2.4 Trade-offs and Decision ...................... 16  
   3.3 Detailed Design Description ....................... 17  
   3.4 Testing and Verification ......................... 17  

4 Autopilot ............................................ 19  
   4.1 Overall Implementation ............................ 19  
   4.2 APM ................................................ 20  
      4.2.1 Overview ...................................... 20  
      4.2.2 Mavlink Protocol ............................. 20  
      4.2.3 Pre-built versus Making a Custom System .... 22  
   4.3 Auto Mega ......................................... 22  
      4.3.1 Overview ...................................... 22  
      4.3.2 Microcontroller Options ...................... 22  
      4.3.3 Detailed Design .............................. 23  
   4.4 Vehicle Mega ....................................... 24  
      4.4.1 Overview ...................................... 24  
      4.4.2 Safety Feature Trade-off ...................... 24  
      4.4.3 Software Serial ............................... 25
4.4.4 Detailed Design .............................................. 25
4.5 Testing and Verification ............................................ 25

5 Summary and Conclusions ........................................... 27
5.1 Work Completed ..................................................... 27
5.2 Future Work .......................................................... 27
5.3 Lessons Learned ..................................................... 28

A Safety Documents ..................................................... 30
A.1 Vehicle Operation Procedures ...................................... 31
A.2 Operating Procedures while Car is on Jacks .................. 33

B Source Code ............................................................. 34
B.1 Auto Mega Code ........................................................ 34
B.2 Vehicle Mega Code ................................................. 37

C LIDAR Bracket ............................................................. 51
List of Figures

1.1 DARPA contestant ......................................................... 2
1.2 Google Car ................................................................. 2

2.1 RSL Rover Side View .................................................. 6
2.2 RSL Rover Front View .................................................. 7
2.3 RSL Rover Overhead View ........................................... 7
2.4 Full System Component Block Diagram ...................... 8
2.5 Previous Year’s Component Block Diagram ............... 8
2.6 Autonomous Mode Component Block Diagram .......... 9

3.1 SICK LIDAR LMS 111 ..................................................... 15
3.2 SICK LIDAR Detection Zone Example ..................... 17
3.3 LIDAR Test Results ..................................................... 18

4.1 Detailed Component Block Diagram ......................... 19
4.2 APM ................................................................. 20
4.3 Mission Planner ......................................................... 21
4.4 Sample Mavlink ......................................................... 21
4.5 Arduino Mega .......................................................... 22
4.6 Auto Mega Flowchart ................................................. 23
4.7 Example Command Strings .................................... 24
4.8 Vehicle Mega Flowchart ......................................... 25
4.9 Waypoint Navigation Test ....................................... 26
List of Tables

3.1 Comparison of different sensors .................................................. 16
Chapter 1

Introduction

The advancement of technology has allowed humans to venture into worlds that 10 years ago would not have even been possible. Increased processing power allows for the creation of more complex control algorithms to manage evolving systems. These control algorithms may be used in a variety of applications, such as the development of autonomous features in vehicles. All vehicles currently have driver-assist functions installed: cruise control, antilocking brakes, etc. These are pieces of technology that did not exist 50 years ago, but now they are common-place. However, imagine a world where everyone has a vehicle that can reach a destination with minimal human interaction and input. The driver would be able to sit back, relax, and use the driving time for other activities. The average American spends approximately 87 minutes behind the wheel each day [1]. This is time that could be used for more productive or enjoyable tasks. Autonomous cars will also lead to less congestion on the roadways and fewer accidents because, although many like to think that they are good drivers, statistics show that human error is the cause for around 85% of all accidents [2]. These accidents are usually caused by the driver getting distracted, not adjusting properly to driving conditions, or driving under the influence. The driver often has control over these issues but lacks the discipline to do so. Eliminating these human errors through autonomously controlled vehicles will reduce the accident rate. For these reasons, the dream of a fully autonomous vehicle is soon to become a reality.

In 2004, the Defense Advanced Research Projects Agency (DARPA) presented a challenge to create an autonomous vehicle with the ability to navigate to different waypoints in a preset course. The challenge was at first unsuccessful, as nobody was able to create a vehicle that was able to navigate the entire course. One example of an unsuccessful autonomous car entry in the DARPA Grand Challenge is shown in Figure 1.1. But in 2005, five teams were able to complete the 130 mile course. After only one year, there was a significant amount of progress in the field of vehicle autonomy.

In the decade since, four states have implemented laws that allow for the testing of autonomous vehicles, and this number is expected to grow as the technology improves. Google is leading the charge with a team of
15 engineers, many of whom were people who competed in the DARPA challenge in 2005 [4]. Their most recent prototype, shown in Figure 1.2 has no steering wheel or pedals installed; it essentially eliminates the driver’s ability to override the vehicle’s controls. This prototype currently has some limitations, however. It cannot be driven in heavy rain or snow, as those weather conditions interfere with the sensors, and it also has difficulty distinguishing and avoiding potholes and other uncommon objects. Google hopes to have all of these issues fixed by 2020 so that they can manufacture vehicles that can be used in all conditions.
The motivation behind our project was to use emerging technology in an educational manner to improve the lives of others. Future students and researchers will be able to build upon what the team has accomplished and make strides to improve this new technology.

1.1 Literature Review

The sources reviewed have provided valuable information for the development and advancement of this project. The articles focused on the possible applications for autonomous vehicles, different types of sensors used for autonomy, and complex data processing. Although the RSL Rover doesn’t have the same amount of complexity as these other projects, many of these concepts still apply.

Autonomous vehicles have a large number of possible uses that would have a great impact on society. Some of these uses would be: daily commuting, mapping unknown regions, transportation of dangerous material, and search and rescue missions [6]. Increased research and development into improving autonomous vehicles will be a great benefit towards society by increasing safety and efficiency. The possible applications for autonomous vehicles include decreasing the number of accidents on the road, searching deeper into the unknown depths of the oceans, and eliminating human risk in dangerous environments such as bomb diffusion, and detection of landmines. The applications for autonomous vehicles are seemingly endless.

Currently a large number of car manufacturers have started R&D on autonomous vehicles. As mentioned earlier, Google has developed a driverless car. With Google paving the way in lobbying and publicity, other car companies are starting to manufacture driverless cars. Although their cars are not yet fully autonomous, their technology seems well equipped to beat out Google in the autonomous car industry [7]. Car manufacturers Lexus, Volvo, Mercedes, and Infiniti all have vehicles on the market which incorporate a front-facing radar and adaptive cruise control in order to avoid collision with the vehicle in front of them. These vehicles will pave the way for more advanced technology, and eventually, affordable autonomous cars.

Different types of sensors are used for different tasks on autonomous vehicles. A combination of sensors are a necessity to ensure a reliable, safe system. The Google car has a total of 6 sensors: a LIDAR, GPS, sonar, odometry, radar and video camera [8]. Currently the price point of all this hardware is over $60,000. The price must be reduced drastically in order to bring this technology to the general market. One group attempted to lower cost by using a Microsoft Kinect sensor for object detection [9]. The Kinect was useful in its low cost and widespread availability, but faltered in other areas. In the study, different vegetation structures were processed as obstacles by the Kinect. The biggest drawback was that during daytime, the light from the sun overpowered the sensor and resulted in poor object detection. Also, the range of detection was only a few meters, which is a drawback for vehicles moving at high speeds. Another group in Portugal developed
a vehicle that could do basic automation with low resolution sonars and a camera mounted to the front of the vehicle \[10\]. They used simple filtering techniques for the sensors to remove possible interference. Even with this crude, low cost method, the vehicle was constantly reproducing accurate results. Sensors that use light waves, known as LIDAR, are used in a variety of applications, which include mapping, urban planning, navigation and law enforcement \[11\]. They can map out areas very quickly and accurately; light waves don’t face the same interference that other sensors like sonar or radar face, making it a much better choice in many applications.

In traditional autonomous vehicles, there is a large amount of data being inputted at the same time while the vehicle is running. The on-board processing system must be able to make use of all of that information in order to make a decision in a small window of time. Most vehicles with LIDAR technology use grid based processing to handle this information.

The basic idea of this type of data processing is that very small amounts of data are each stored in very specific ‘locations’, corresponding to their actual physical locations, within a gigantic grid. This allows the system to quickly and efficiently locate and analyze whatever specific data may be relevant at the current moment, and then update all of it in real time.\[11\]

The Audi A7 autonomous vehicle has 8 onboard PC’s, which are responsible for “controlling data logging, planning a path, controlling steering, braking, and acceleration, operating the near field cameras and fusing sensor data”\[12\]. Audi’s plan is to consolidate these sensors, but this shows how complex processing data could be. Multiple computers are necessary to meet safety standards set by different states. They allow failsafe tactics to be used if some hardware or software fails.

The approach taken on the RSL Rover was to use pre-existing software and interface it with the current system on the vehicle. This project is not nearly as complex as the vehicles being worked on by different car manufacturers, and thus it was appropriate to use a simpler system that the user would have no problem understanding. The application for the vehicle was for the detection of underground objects, but because the project only spanned a year, the team decided to focus on two main aspects of the project, navigation and obstacle avoidance. The rest of the tasks were left for future groups to implement. The waypoint navigation and GPS tracking are the main aspects of the navigation portion of the project and are handled by the 3D Robotics Ardupilot Mega (APM) unit. The APM stores the waypoints defined by the user and relays those to the vehicle actuators to start moving the vehicle to the desired waypoint. A LIDAR is used as the sensor for object detection and avoidance. It is not limited by range or by daylight, and does not cost as much as the equivalent hardware on the Google car. The on-board processing for the vehicle is handled by Arduino
microcontrollers, which allow for complex processing of data for navigation, obstacle avoidance, and safety, at a reasonable cost.

1.2 Problem Statement

The objective of this project was to use the existing RSL Rover vehicle, with its drive-by-wire control system, to implement an autonomous waypoint navigation system with the ability to avoid collisions. The tasks for the project were three-fold:

- Design and implement controllers that interface with the current system, in order to navigate along a path specified by a series of waypoints.
- Install and integrate sensors on the vehicle to detect and avoid collisions with objects.
- Develop an effective safety mechanism and emergency shutoff method that will prevent the rover from causing injury and damage.

The result of this project is a highly capable autonomous vehicle that may be used in a variety of capacities, including the detection of landmines, underground pipes, etc, to serve areas that face these obstacles.
Chapter 2

System-Level Design

2.1 Vehicle Background

The vehicle used for this project was originally built to compete in the 2004 & 2005 DARPA challenges by Team Overbot, a group of Silicon Valley engineers interested in vehicle autonomy. The team failed to qualify for the national competition both years and eventually donated the vehicle to a local university where it was used for educational purposes. The car sat idle for several years and was eventually donated to SCU’s Robotic Systems Laboratory in 2013. When the 2013-2014 team project obtained the vehicle, the internal components were in disarray. They were able to revert most of the changes over the years to set it back to factory standards. On top of that they “built a hierarchical control system, robust actuator mounts, and an effective safety system” [6] in a flexible manner that allows for upgrades. With these additions they were able to control the vehicle remotely by implementing a drive-by-wire interface. Figures 2.1, 2.2 and 2.3 show the existing vehicle.

Figure 2.1: RSL Rover Side View
2.2 Overview of the Project

For this project, an Autonomous mode was added to the vehicle on top of the Pre-existing Drive by Wire mode, seen in Figure 2.4. The vehicle traverses a field by following a set of waypoints defined by the user while avoiding any objects that may be in its way. This will be done with the sensors that are attached to the vehicle.
The Vehicle Mega has a software switch that chooses between the mode that the user will set. It dictates what commands will be passed on to the motor controllers. If Autonomous mode is switched on, the Vehicle Mega will block the commands from the drive by wire and allow the autonomous commands to pass. Otherwise, the motor controls will receive commands only from the Pre-existing Drive by Wire interface. Figure 2.5 gives us an understanding of the complexity of the motor control system.

Within the Autonomous Mode, there is an autopilot navigation subsystem and obstacle detection subsystem, both of which feed signals into a separate microcontroller. This microcontroller, known as the Auto Mega, processes this information and sends it into the Vehicle Mega. Figure 2.6 shows how the autonomous mode is connected.
2.3 Customer Definition and Needs

The objective of this project was to retrofit the current unmanned land vehicle with sensors and microprocessors to allow the vehicle to have autonomous capabilities. The process of doing so needed to be well documented and the final product had to be easy to use, so that future students can use the vehicle with a relatively small learning curve. The vehicle had some built-in constraints such as the weight, size, engine power, etc. However, the vehicle was still relatively open to modifications for our purposes. The team was able to customize the type of sensors that were used and design the user interface, vehicle speed, accuracy, robustness, and cost to fit the customer needs.

There were two customers in mind for the project. The first one was the Robotics Systems Laboratory. Future undergraduate, graduate and Ph.D. students will be able to use the vehicle to expand their knowledge of control systems, vehicle control, etc. The second customer was a group of farmers in rural locations in California. Their lack of resources makes it hard for them to compete against large farming corporations. This vehicle will be able to assist them in the detection of underground pipes that may be hidden in their plots of land.

The final product also needed to be easy to use and understandable. Thus, the algorithms used to control the vehicle had to be simple and robust enough that users with limited knowledge would not have any issues. A friendly user interface will allow the user to control aspects of the vehicle without actually having to change the code. Quality of the parts is also an issue. The system was retrofitted with the best parts that the budget allowed, to decrease the maintenance needed in the future and make it easier for the customer to trust the system. Also, by using parts manufactured by well-established companies, the components are less likely to become obsolete in the future.

Safety was a large concern for this project. Because this system is autonomous, extra precautions had
to be taken because a mistake could result in a serious injury or even death. Several safety systems must be implemented to protect both the user and the vehicle. These include dead man switches that shut off the system if the connection to a sensor is lost, warning signals/sounds from the vehicle, and a maximum speed limit in relation to the environmental conditions and sensor quality. The safety of the end user is the primary concern.

2.4 System Level Requirements

While the system requirements from the previous team are still valid. This year’s project will have five additional requirements:

- **Interface with Existing System:** The vehicle shall keep the existing drive-by-wire interface and add an autonomous mode on top of the existing system and interface with it.

- **Navigation Control:** The vehicle shall take in GPS coordinates, or waypoints, and use those to traverse a plot of land or a path. When following a predetermined route, the vehicle will not deviate more than 3 meters. A closed loop feedback control system will be implemented to read the feedback and make adjustments if necessary.

- **Velocity Control:** The vehicle shall maintain a 5 mph speed. The speed can be adjusted depending on the terrain, weather conditions and visibility. The maximum speed will not exceed 10 mph.

- **Sensor Control:** A digital flag shall be sent to a microcontroller when an object is detected. The latency between the sensor and microprocessor shall be less than 500ms.

- **Interface/Data Storage:** To make troubleshooting easy, the vehicle will generate a .txt file every time the vehicle starts, in order to log the vehicle diagnostics and performance, time, GPS, coordinates, activity and status.

2.5 Functional Analysis

The project was broken into four major subsystems: 1) Navigation, 2) Sensor Interfacing, 3) User Interface, and 4) Safety. All of these subsections are intertwined, but it’s easier to understand the work that needed to be done on each one if it is broken down. This also helped prioritize the most important subsystems and how they affected the other subsystems.

**Navigation System:** An off-the-shelf autopilot system was used. This ArduPilot (APM) comes pre-built with GPS capabilities and compass. The information being sent by the APM was interpreted by a microcontroller and be translated to the correct motor control values.
Sensor Interface: A LIDAR from SICK was used for collision avoidance. The LIDAR outputs a digital signal to a microcontroller if it detects an object within a set distance. This method does not burden the microprocessor with an excessive amount of data, allowing for quicker response times.

User Interface: The end goal is to have an easy-to-use user interface so that people without a large amount of technical training, have the ability to understand how to use it. The software used was Mission Planner, which is an open source software that is made specifically for interfacing with the APM.

Safety: Hardware and software automatic shutoffs were used. Because the system is autonomous, the vehicle must stop at a safe distance if it malfunctions. Code was implemented that will shut off the system if any of the sensors are unable to send information to the microcontrollers. The wiring is also important to safety; the connections between the components must not come loose in the driving process.

2.6 System Level Issues and Trade-offs

The team had to decide how much of the old vehicle to keep and if the same control commands would be used. These decisions were made early in order to design the system in an efficient manner. This operational drive-by-wire vehicle was inherited from last year’s team but the documentation in several cases was incorrect, and time was spent reverse engineering several existing subsystems. Eventually the team was able to get the system functioning as intended. The hardware added to the system was integrated into the previous team’s work. The team also improved the documentation of the vehicle, in both the previous work and the work for this year.

The previous year’s team already decided how the vehicle would receive, interpret and respond to motor command. They developed an algorithm that would specify the steering angle with either a brake or throttle commands. This year, the team wanted to preserve their code as much as possible, so it was decided to continue to use their command structure. The challenge was to get the external data translated into a form that the vehicle could process correctly. This issue was easier to solve than changing the entire original system.

2.7 Team and Project Management

2.7.1 Initial Team Goals

The team is comprised of 3 Electrical Engineers and a Mechanical/Computer Engineer double major who all have a passion for automotives and exploring control algorithms. At the beginning of the year several goals were established:
1. Improve the original control algorithm if needed, and add any code needed for the project.

2. Work with microcontrollers, sensors and safety mechanisms used in the industry.

3. Find a balance between the project, school work and the team members’ social lives.

4. Obtain a good grade, and have a working prototype done by the end of spring.

2.7.2 Mitigating Project Challenges

As with any big project there are a number of issues that arose in the decision making and design of the vehicle. These major challenges include: faulty mechanical equipment, unfamiliar code in regards to the autopilot system, and team disagreements.

Time management was another challenge that was faced. Deciding how the work would be split up and meeting goals became an issue because of lack of experience and manpower. When these issues arose, ways to fix these problems were implemented. They include:

1. **Having good communication with one another.** Completed work was reported so that the other members would know what issues had either been resolved or needed more work. This way no time was wasted doing unnecessary things. It was encouraged to share opinions and think about possible solutions to the current problem. This allowed us to see the problem from many different angles and to choose the best solution.

2. **Willingness to seek out help.** Many times others outside the project team were asked to assist with issues that the team was not have been familiar with. Using the available resources was the best way that the team could optimize the time spent on certain portions of the project.

3. **Effective documentation.** How the algorithm works, what parts were used, how the wiring is constructed was well documented. Because this is a legacy project, all the appropriate information needed to be organized for next year’s team. Having good documentation allows later groups to quickly refer to things when problems arise.

2.7.3 Timeline

When the team started in the Fall, a full understanding of the current vehicle and control structure was needed by the middle of the fall quarter. Incorrect documentation of the vehicle layout resulted in large amount of time reverse engineering the car. Eventually the vehicle was able to run by the end of the Fall, but by that time, 2 months of work time had been lost already. This setback caused the team to revise priorities and
determine how quickly certain components of the project were to be completed. Winter quarter consisted of obtaining all of the hardware necessary for integration, studying the documentation, and starting to write the algorithms that would integrate the autopilot system with the vehicle for autonomous driving. Delays in receiving parts and bugs in the code extended the schedule. Spring quarter consisted of installing sensors, integration of the entire system, and final testing.

2.7.4 Risks and Mitigations

The size of the vehicle necessitated extra safety precautions during testing. Every emergency stop button had to be able to be activated if something were to fail, and other safety measures were programmed into the control algorithms. Safety is one of main concerns for the project; therefore no effort was spared to ensure safe operation of the vehicle.

Another problem that could have arose was the clarity of work. As the team found out from trying to read the work and documentation from the previous year’s team, information may be presented that is not clear for future groups. By maintaining proper documentation of the work and procedures used to replicate results, the next team to work on this vehicle will be able to learn the basic workings of the vehicle in a minimal amount of time.

The final risk is the danger of the operators and bystanders around the vehicle. This hazard was minimized through the writing of safety procedures that were followed when working on the vehicle. Keeping civilians clear of the work area was also important to ensure safety. More information about these safety documents can be found in Appendix A.
Chapter 3

Sensors

3.1 Role and Requirements

In order for the RSL Rover to drive safely, the vehicle needs a robust sensor that has the ability to detect objects and send that information to the vehicle microcontroller. Due to the nature and size of the vehicle, the range of detection has to be large. This ensures that there is adequate space for the vehicle to stop if needed. The vehicle does not use that data to distinguish between different types of objects, but this may be expanded further in the future.

3.2 Options and Trade-off Analysis

A variety of sensors and their specifications were compared to see what would best suit our needs. Our primary need for the sensor was to be able identify and determine the location of obstacles that may be in the vehicle’s path. We had a multitude of options for sensors but a majority of them only outputted raw data that had to be processed with certain software. Other aspects that the team looked at were cost, maximum distance of measurement, size, and ease of use.

3.2.1 SICK - LMS 111

The LMS111, seen in Figure 3.1, is a compact, medium-sized LIDAR from SICK that has a maximum range of 20 meters. It has a robust encasing to protect it from any vibrations due to the vehicle’s operation. The weight of the sensor with the bracket is approximately 1.1 kg. The LMS111 has a field of vision (FOV) of 270 degrees (horizontal) with an angular resolution of .25 degrees. The precision of the measurements are in the millimeter range. The fact that this LIDAR only functions in a 2D plain limits the ability to expand into more complex processing. The output of the unit can be read via serial (USB), CAN and Ethernet. This is particularly useful because different systems will have different data interfaces and data transfer requirements.
It also has an onboard processing unit which greatly reduces the amount of work needed to implement this solution. The cost of the unit falls around $5,800. [14]

### 3.2.2 Velodyne VLP-16

The VLP-16 is the newest LIDAR from Velodyne. It was designed to be more compact and cheaper than other Velodyne products. With a price of $8000, it is several magnitudes cheaper than the older HDL-32E. A big advantage of this unit is that it has a maximum range of 100m [15]. This is several times larger than other LIDAR unit currently in the market. The FOV is 360 degrees (horizontal) and +/- 15 degrees (vertical). The precision of this unit is in the centimeter range. This allows the system to map out obstacles on a 3D scale, which could be helpful in the future if more complex processing is done. The output of this unit can be read via Ethernet. This limits the type of hardware that can be used to process the LIDAR data. This system does not have onboard processing capabilities, therefore processing would have to be designed by the team and run on an embedded system or laptop which would be too much work for the team to do in the allotted time.

### 3.2.3 Camera

Cameras are a piece of technology that have been used a variety of applications. The team looked at whether a camera could serve as the primary sensor for obstacle detection. The cost of cameras used on autonomous vehicles range from $100 to $200. Cameras have the ability to see in color, which could be used to distinguish
between various objects such as pedestrians, trees, and other vehicles. The field of vision is approximately 120 degrees with a resolution that allows for a large amount of detail. Like the Velodyne LIDAR, a camera would require image processing which the team would have to design on an embedded system or laptop which would be too much work for the team to do in the given time.

### 3.2.4 Trade-offs and Decision

All the sensors that could be used for primary obstacle detection have pros and cons. These were weighed against each other to come up with our decision. A comparison of the sensors can be seen in Table 3.1

<table>
<thead>
<tr>
<th>Sensor</th>
<th>Cost:</th>
<th>Distance:</th>
<th>Weight:</th>
<th>Ease of Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>LMS111</td>
<td>$5,800</td>
<td>20m</td>
<td>1.1 kg</td>
<td>High</td>
</tr>
<tr>
<td>VLP-16</td>
<td>$8,000</td>
<td>100m</td>
<td>.83 kg</td>
<td>Low</td>
</tr>
<tr>
<td>Camera</td>
<td>$200</td>
<td>Varies</td>
<td>Varies</td>
<td>Low</td>
</tr>
</tbody>
</table>

At first our team was going to use a camera for obstacle detection. The cost is low, and it was a proven way of detecting and distinguishing objects. The issue of processing data from the camera ultimately prevented us from using this hardware. Although software exists for this purpose, integrating that software with the current system would have been difficult. A laptop or embedded processor would have had to be installed in order to interpret the data and convert it to vehicle commands. “Computer vision is just not anywhere nearly good enough today to detect all important features with the reliability necessary for safe driving.” [16] Ultimately these two issues related to ease of use pushed the team away from this method.

Next the team took a look at the VLP-16. 360 degree vision and a large detection range are something that made this sensor favorable. A laptop or embedded system would have to be installed and interfaced with the current system. It is likely that there is driver software available thus integration with the system would have been relatively easy. However, the cost was also an issue; it was way out of the budget for this project.

The final sensor the team looked at, the LMS111, was a sensor that didn’t have the same problem that the other two choices had. We could use the onboard processing on the unit to set up detection zones that would send a digital output to a pin if an object is detected. This was perfect for our application because there was no need to distinguish between different types of objects. Interfacing was made easy because the LIDARs digital output signal could be directly fed into an existing microcontroller. Finally, SICK generously donated the unit to the Robotic Systems Lab. Cost was no longer an issue for this sensor, leading us to ultimately choose to work with this hardware.
3.3 Detailed Design Description

The detection zones are set on the user’s laptop using the SOPAS software provided by SICK. Figure 3.2 gives a general idea on the different shapes and locations that these regions can be placed. The darker regions represent the detection zones that were set for this example application. Once these are set, the parameters can be uploaded to the LIDAR where they were stored in its internal memory. One of the digital pins from the LIDAR is connected to a microcontroller on the vehicle. A signal will be sent to this pin if the sensor detects an object in the zone set by the user and the vehicle is in autonomous mode. More information about this will be mentioned in chapter 4. If the microcontroller reads a HIGH voltage (5V) on the pin, it will immediately trigger an emergency stop.

Limiting the amount of information sent to the microcontrollers was a design feature that was purposely implemented. Delays could be caused if an excessive amount of information was constantly sent to the processing units on the vehicle. Delays would reduce the reliability and safety of the vehicle.

3.4 Testing and Verification

The LIDAR was tested on a stretch of road outside the SCU parking garage. The car was manually driven to be approximately 60 feet away from a large, flat board that served as the obstacle. Then, running a script that directed the car to drive straight continuously, the car was started and set in autonomous mode. The car then drove straight down the road until it sensed the object. The LIDAR detection zone was set at 1 meter wide and 10 meters long. While 10 meters is a generous margin for the slow speeds used in this trial, we chose this
distance in case the LIDAR did not detect an object, giving the passenger enough time to manually engage the emergency stop sequence. We could test at 10 meters rather than the 3 meters we planned to use in the field because we were testing how close to the setpoint the car stopped rather than the actual setpoint. We found during testing that the LIDAR never failed to detect an object that was in the zone. A 100% detection rate is important to maintain safety and limit risk. Figure 3.3 is a graph of the results of the test.

Figure 3.3: Averaged stopping distance of the Rover from obstacle with a set stopping distance of 10.5 M. Each test was run 5 times.
Chapter 4

Autopilot

4.1 Overall Implementation

To drive the RSL Rover autonomously, a suitable autopilot system was designed. This system consisted of microcontrollers, the software of those microcontrollers, and sensors that work together to replace the human control of regular vehicles. The general implementation structure of the vehicle is shown in Figure 4.1.

![Figure 4.1: A detailed block diagram showing how all the components upstream of the Vehicle Mega interact with one another.](image)

Autonomous mode is made up of the APM, Auto Mega and LIDAR. The APM sends steering and throttle commands to the Auto Mega which converts these signals into commands that the vehicle can understand. The LIDAR sends a digital flag to a pin on the Vehicle Mega if it detects an object in front of it. The Auto Mega sends the commands from the APM to the Vehicle Mega, which feeds those commands to the...
appropriate actuators. The Vehicle Mega sends either commands from the Auto Mega or Console Mega to the rest of the vehicle, depending on which mode the vehicle is set in.

4.2 APM

4.2.1 Overview

The ArduPilot Module (APM), shown in Figure 4.2, is a microcontroller specifically designed to work as an autopilot and is the device that actually drives the Rover in autopilot mode. The APM comes with a compass and GPS module that serve as the inputs to the system. By processing these inputs with the desired waypoint destinations, the APM is able to generate signals that drive both steering and throttle actuators. The waypoints are chosen by the user in an open source software program created by 3D Robotics called Mission Planner. A picture of Mission Planner is shown in Figure 4.3. Mission Planner is integrated with Google Maps and allows the user to set waypoints on a satellite map of where they want the APM to direct their vehicle. Once the user is finished programming the route, it can be downloaded to the APM. Upon the APM being set in Auto mode, and the APM will direct the vehicle to those waypoints.

4.2.2 Mavlink Protocol

The APM is designed for use with hobby RC vehicles so the output signals are Pulse Width Modulation (PWM) voltages, good for driving the servo motors typically found in these RC vehicles. The actuators found on the RSL Rover are not run by PWM voltages, however. Therefore, the output voltage value is extracted from the APM and sent as serial data, as this is how the Rover was designed to drive the actuators. The team used the Mavlink protocol to send serial data.
Mavlink is the protocol that the APM uses to send out serial data. It consists of packaged messages that contain data to be sent between an autopilot and a ground control station. In the case of the Rover, the Auto Mega plays the part of the ground station. These messages are sent serially through the telemetry port on the APM. Figure 4.4 shows the Mavlink message format.

The most basic message is the heartbeat. This message verifies the connection between an autopilot and a ground control station. The ground control station sends out a heartbeat message and upon receipt of this message, the APM responds with another heartbeat. If heartbeats are sent and received every second, the communication works as normal. If after some time no heartbeat messages are received on either end, the APM goes into a designated safety mode.
4.2.3 Pre-built versus Making a Custom System

The key decision made about the APM was whether the team wanted to tackle designing the autopilot or buy a premade module. By creating a custom system, we could ensure that the exact specifications of the current Rover could be met, but it would be extremely difficult to design. By buying a premade module, we would avoid spending too much time designing, but whatever outputs the module yielded would have to be processed to be compatible with the system. The team chose a premade model because, given the time constraint of one school year, processing the outputs would be more feasible than building an entire autopilot system. The output processing proved to be one of the most time-consuming parts of the project but it was easier than designing a whole autopilot system.

4.3 Auto Mega

4.3.1 Overview

The Auto Mega takes Mavlink steering and throttle signals from the APM and translates them to the car commands that will be sent to the Vehicle Mega. The Rover does most of its processing with Arduino Megas, depicted in Figure 4.5. The Arduino Mega is a microcontroller that allows for digital and analog signals to be processed and controlled. The Auto Mega is one such Arduino Mega and its purpose is to translate the Mavlink messages to serial commands.

![Figure 4.5: Picture of the Arduino Mega](image)

4.3.2 Microcontroller Options

Research showed that the APM used the Mavlink protocol to send data over the serial line. To pick the data it sent out, a Mavlink message that sets telemetry rates must be sent into the APM. To generate this command, a choice had to be made whether to use a Mavlink library on the Arduino, or droneAPI on the Intel Edison. The Mavlink library offers a direct way to generate Mavlink commands. The user simply has to create the
Mavlink messages and design the algorithm to send them out according to the Mavlink protocol. In contrast, droneAPI offers a more high-level approach in which the hardware on the APM is abstracted to software objects and the user programs what he or she wants the hardware to do. The API deals with the low level algorithms, such as the sequence of Mavlink messages to send, that must be run in order to carry out the actions that the user specifies. While the coding of the Edison seemed simpler, we first chose to go with the Arduino because we were more familiar with it and the rest of the Rover was run by Arduinos. We had difficulty writing working code so we switched to the Edison. After suffering many setup issues and realizing that droneAPI could not even access the data wanted to send out, we reverted back to the Arduino. After reading through some Mavlink logs of messages sent between the APM and Mission Planner, we were able to pattern our Arduino code on these logs and successfully had the APM and the Arduino communicating. We therefore stuck with the Arduino to process the APM outputs.

4.3.3 Detailed Design

Figure 4.6 shows the algorithm that the Auto Mega uses to convert signals to vehicle commands. While the main purpose of the Auto Mega is to re-express the steering and throttle signals as car command strings, it must first generate and send a Mavlink message to the APM, requesting it to send out these signals at 10 times per second. The APM responds by sending the signal values in Mavlink packages over the UART channel to the the Auto Mega. The Mega, in turn, unpackages and extracts the data using a Mavlink library written for Arduino Megas. The data is then ready to be formatted into a vehicle command. A sample of the vehicle command structure is shown in Figure 4.7.

The steering and throttle signals from the APM are in the range of 1000 to 2000 with 1500 representing
the neutral position. To fit the structure of the existing car commands, these values had to be converted to a value between -1000 and 1000 with 0 as the neutral position. Moreover, to ensure the vehicle does not exceed the maximum speed, the value for the throttle command was truncated at 5% of maximum. The Auto Mega converts this value, inserts it into the proper serial string, and then sends it along to the Vehicle Mega.

### 4.4 Vehicle Mega

#### 4.4.1 Overview

The Vehicle Mega was designed by last year’s senior design group and serves as the entry point for vehicle commands; commands are fed into the Vehicle Mega and the vehicle responds with the appropriate action. In their design, the Vehicle Mega took commands as serial strings from the Console Mega and passed them along to the appropriate microcontroller. We kept this feature and added an Autonomous mode. When the Vehicle Mega senses that the designated Auto Switch has been pressed on the console, it reads in commands from the Auto Mega instead of reading and passing through commands from the Console Mega. These commands are connected to the software serial port on the Vehicle Mega. Thus, the stream of commands that the Auto Mega sends out are what the car actually does.

#### 4.4.2 Safety Feature Trade-off

In last year’s design, the Rover triggered the emergency stop sequence when it lost contact with the Console Mega. This is a safety feature to protect the car and the driver in case the console breaks, gets out of range of the Rover, or emergency stops due to the emergency stop button being pressed on the console. The team had to decide if we wanted to keep this feature in our new Autonomous mode. While it is not important to maintain communication between the Console Mega and the Vehicle Mega to control the car, we wanted the user to be able to trigger the stop sequence remotely using the emergency stop button on the console, in case any of the vehicle’s automatic safety features stop working. For this reason we decided to keep this feature.
### 4.4.3 Software Serial

In adding the autopilot function to the RSL Rover, we discovered that all three serial ports on the Vehicle Mega were already in use. After some research, the team found that by using a software library, two digital IO pins could be designated on the Mega to act as a serial port. The only caveat was that it had to be run at a baud rate of 1200. Otherwise, data corruption would occur. This was not much of a problem as it did not affect performance.

### 4.4.4 Detailed Design

Figure 4.8 shows the algorithm that the Vehicle Mega uses to choose between source microcontrollers and to sense the LIDAR. The main purpose of the Vehicle Mega is to route vehicle commands to the appropriate actuators. When we inherited the Rover, the Vehicle Mega ran the simple loop of reading in commands from the Console Mega and sending them to the appropriate destination, all the while verifying the connection between the Console Mega and the Vehicle Mega. If Vehicle Mega stops receiving commands, it E-Stops.

The current system now checks if the Auto Switch is pressed. If so, it checks for a signal from the LIDAR. This flag means an obstacle has been detected so the Rover should make an emergency stop. If no LIDAR flag is detected, the Vehicle Mega will continue to pass commands to the corresponding actuators from the Auto Mega. Note, that the LIDAR flag only triggers an emergency stop when the Auto Switch is pressed.

#### Figure 4.8: Flowchart of the Code run on the Vehicle Mega

![Flowchart of the Code run on the Vehicle Mega](image)

4.5 Testing and Verification

Testing was performed for the autopilot navigation on the third floor of the main parking garage at Santa Clara University. This location was chosen because of it is at close proximity to the lab and at night the testing site...
was free of cars and foot traffic. During the test, the Rover started out pointing away from the first waypoint. It drove forward, correcting its path, and then made a 90° turn and drove forward again. The accepted criteria for this test was that it hit the way points with a deviation of no greater than 3 meters. This tolerance was chosen based on the accuracy of the GPS that was used.

As can be seen in the results of the test in Figure 4.9, the acceptance criteria was met at all points, well within the tolerance.

![Figure 4.9](image)

**Figure 4.9**: Path of the ROVER during waypoint navigation test overlapped with expected path, the circles around the waypoints are a radius of 3m.

While we did not have any criteria for the Rover to stay on a path, it should be noted that it had a slight serpentining motion and seemed to go significantly off target right after the turn. However, the Rover quickly corrected its path and headed toward the next waypoint. We are not concerned about this as our focus was to reach the waypoints. In the end, the Rover did not finish its projected path because of a support pillar which obstructed its path and the obstacle detection safety feature stopped the vehicle. While we would have liked the vehicle to complete the designated path, it was reassuring to know that the Obstacle Avoidance process worked.
Chapter 5

Summary and Conclusions

5.1 Work Completed

Our initial goal for this project was to add an autonomous driving mode to a Gator class All Terrain Vehicle with an existing drive-by-wire system while maintaining all previous functionality. Furthermore, we ensured the safety of those operating the vehicle as well as the vehicle itself by adding obstacle detection. Over the course of the year we developed a fully autonomous driving system as well as a functioning obstacle detection system. All existing drive modes and safety feature were preserved and appropriate new safety features were added. To handle the navigation we added an APM autopilot and Arduino Mega. For the obstacle detection we added a LIDAR. When our vehicle detected an obstacle it made an emergency stop before it could cause damage or harm.

5.2 Future Work

The most pressing goal for a future group would be to create a more sophisticated obstacle avoidance system as opposed to the current obstacle detection system. In this new system instead of the vehicle stopping when detecting an obstacle it would drive around and avoid the obstacle. Currently the vehicle has a slight serpentine to its path and overshoots the tolerance on a sharp turn. While the team designed the Rover with waypoint navigation, it would be more useful if the vehicle followed a path. Using the groundwork from the waypoint navigation, a future group could implement path following for a more reliable Rover. Finally there are minor vehicle repairs to non-essential systems that a future team might want to fix.

A long term goal of the RSL Rover is to be able to autonomously detect underground objects. To do this a future team would have to attach a ground penetrating radar to the vehicle. Furthermore they would have to add a component to compile the data from the radar with GPS data to create a map showing the locations of the objects.
5.3 Lessons Learned

Going into this project our team had a limited knowledge of the Arduino language and the mechatronic design process and no knowledge of the LIDAR and APM software. In working with the APM, we learned that documentation on open source software is not the best, requiring us to infer functionality of other people’s work by examining their source code. This, however, will be an invaluable skill if we ever work in the mechatronics industry.

Another important lesson we learned was the necessity of getting help. We were very unfamiliar with certain aspects of the project, namely how the engine worked and certain part of the software. We tried to fix these problems on our own but we ended up just wasting time. It was only when we reached out to people familiar with those fields that we made any progress. From now on we will get help sooner.
Bibliography

[8] Alex Davies. “turns out the hardware in self-driving cars is pretty cheap”.
[12] Alex Davies. I rode 500 miles in a self-driving car and saw the future. it’s delightfully dull.
[16] Brad Templeton. “cameras or lasers?”. online.
Appendix A

Safety Documents

A.1 Vehicle Operation Procedures

VEHICLE OPERATION RULES

General Rules:
1. When operating the vehicle, one person must have a cell phone with enough battery to make a call. In case of emergency, call 911.
2. The vehicle is not street-legal. Do not drive on public roads.
3. Confirm explicit permission from the property owner before operating the vehicle on private property.
4. If trailering the vehicle, use proper loading and strapping procedures.
5. If operating in a parking lot or near buildings or people, use traffic cones to block off a testing zone.
6. Two people must be present when operating the vehicle. Always assign someone to watch surroundings for possibly safety concerns such as people walking or driving by. This person must be ready to stop and direct traffic if necessary.
7. Do not drive the vehicle in small buildings or confined spaces. The exhaust will quickly become hazardous.
8. Do not drive the vehicle when you are impaired. Do not drive when you are too tired, stressed or hurried to drive carefully.
9. Never annoy or distract the vehicle driver or operator unless there is a safety risk.
10. Do not operate the vehicle unless you have been properly trained and approved to do so.
11. Do not leave the vehicle unattended.
12. Always wear the seatbelt while driving or riding in the vehicle.

Pre-Drive Checklist:
1. Visually check the tires for low pressure or damage. If necessary, consult the tire’s specifications to check tire pressure (maximum pressure is marked on the sidewall).
2. Visually check the vehicle for fluid leaks. This includes:
   a. Brake fluid
   b. Differential or gear oil
   c. Motor oil
   d. Gasoline
3. Briefly test the horn.
4. Describe intended testing methods to all who are present; this will make operation malfunctions much easier to notice.
5. Confirm that all components and wiring are properly secured, especially near the wheels. No cables should be hanging down from the vehicle at any point.
6. Release all emergency stop buttons on the vehicle and console.
7. Power on the console and the vehicle electronics and wait for the ARMED light to come on. This signifies that the vehicle is ready to take commands and has gone through its power-up sequence before the motor is started.
8. Shift the transmission into Neutral (or Park on the console) before starting the motor.
9. Apply the pedal brake when starting the vehicle. The choke may need to be adjusted to start the vehicle and maintain idle speed. If this is not a familiar process, seek help.
**Autonomous Extra Testing Guidelines:**

1. All parties present must be informed that the vehicle will be switched to autonomous mode.
2. Someone should sit in the vehicle, ready to operate the brakes and electrical kill switch.
3. The vehicle must be tested in a fenced in field. The field must be in a location that has no bystanders nearby as this could pose a hazard for all parties involved.
4. The test area must be private property with authorization from the owner.
5. When operating in autonomous mode, the vehicle will not exceed 5mph.
6. Testing must be done on a clear day. Lighting conditions may have a negative effect on the sensors which could lead to delayed response times.

If for any reason one of the pre-drive checklist items is unconfirmed, take the appropriate action to fix the problem. If it cannot be fixed in the field, return to the lab. Operating with a compromised system is extremely dangerous.
A.2 Operating Procedures while Car is on Jacks

Safe Operating Rules

This document outlines the minimum Safety Standards for operating the RSL Rover on jacks while applying the throttle required by Santa Clara University and OSHA. RSL Rover safety should exceed these requirements. The purpose of RSL safety is to eliminate:

1. Accidents and Injuries
2. Property Damage
3. Equipment Abuses and Damage

Meeting these requirements requires understanding the system and the laws or regulations that may apply. These operating rules include all the rules required by OSHA, state OSHA and Santa Clara University. All operators need a copy of these Operating Rules and each operator must acknowledge them by signing that they have received a copy of these rules and understand them.

Training requires documentation. Undocumented training has the same basic legal effect as no training. The operator must acknowledge receipt of these rules for documentation to occur under OSHA, state OSHA and Santa Clara University requirements.

OPERATING RULES:

1. Only operators authorized by the management, and trained in the safe operation of the RSL rover will be permitted to operate and test such vehicle. Methods will be devised to train operators in safe operation of an off-road vehicle. A minimum of three trained operators must be present in order to perform testing.

2. Initial operating procedures must be performed at least once before commencing testing. Attention will be given to the proper functioning of tires, horn, lights, battery, controller, brakes, and steering mechanism. All emergency stop systems must be tested and checked to make sure they are all working as intended.

3. A minimum of 6 jacks must be used in the raising of the vehicle. Place these jacks in locations that distribute the weight of the vehicle equally. Apply some force to the side of the vehicle and shake it. The vehicle should stay securely on the jacks and hardly move.

4. A secondary safety measure must be implemented. Tether the car with a chain to either a secure spot on the ground or a spot in the testing area. This location must be able to withstand the force of the weight of the car and/or the movement. This failsafe system will only activate if the vehicle happens to fall off the jacks.

5. The testing area must be clear of any objects within 2 feet of the vehicle. The operators/testers need to be at least 6 feet away from the vehicle at all times when the throttle is being used. Have one of the operators making sure no bystanders are within 50 feet of the test area.

6. The vehicle will not exceed a speed of 8mph which is approximately 20% of full throttle.

7. NO RIDERS WILL BE PERMITTED ON THE VEHICLE. A person may not ride on the rover due to the possibly instability/shaking that may occur.
Appendix B

Source Code

B.1 Auto Mega Code

```c
// Arduino MAVLink test code.
#include <FastSerial.h>
#include "../mavlink/include/mavlink.h" // Mavlink interface

// The value will quickly become too large for an int to store
unsigned long previousMillis = 0; // will store last time LED was updated

// constants won’t change :
const long interval = 1000;
int LidarPin = 51;

FastSerialPort0(Serial);
FastSerialPort1(Serial1);
FastSerialPort2(Serial2);

uint8_t sysid = 255; // ID 20 for this airplane
uint8_t compid = MAV_COMP_ID_MISSIONPLANNER; // The component sending the message is
// the IMU, it could be also a Linux process
uint8_t type = MAV_TYPE_GCS;
uint8_t autopilot = MAV_AUTOPILOT_INVALID;
uint8_t base_mode = 0;
uint32_t custom_mode = 0;
uint8_t system_status = 0;

int Serial_timeout = 25; //Serial Timeout

//serial strings/
String commandType = "C";
String commandMode;
String throttleORsteer;
String toSend;

//flags
bool speedFlag = true;
bool eStop = false;
bool heartbeat = true;

void setup () {
    Serial.begin(9600);
    Serial1.begin(19200);
    Serial2.begin(1200);
    mavlink_message_t msg;
}
```c
uint8_t buf[MAVLINK_MAX_PACKET_LEN];
uint8_t target_system = 1;
uint8_t target_component = 1;
uint8_t req_stream_id = 3;
uint16_t req_message_rate = 10;
uint8_t start_stop = 1;

mavlink_msg_heartbeat_pack(sysid, compid, &msg, type, autopilot, base_mode, 
                          custom_mode, system_status);

// Copy the message to the send buffer
uint16_t len = mavlink_msg_to_send_buffer(buf, &msg);

// Send the message with the standard UART send function
Serial1.write(buf, len);

mavlink_msg_request_data_stream_pack(sysid, compid, &msg, target_system, 
                                      target_component, req_stream_id, req_message_rate, start_stop);

len = mavlink_msg_to_send_buffer(buf, &msg);
Serial1.write(buf, len);
Serial1.write(buf, len);
}

int mapAPM(int analog) {
    /*
    * this converts the 0-5V output of the APM to equivalent serial string commands, [-1000,1000]
    */
    return map(analog, 1100, 1900, -1000, 1000);
}

int rc_apm_function(int analog) {
    /*
    * This function turns the APM output into the serial strings the Rover understands
    */
    int desiredSpeed, desiredSteering;
    if (!eStop) //E-stop
    {
        commandMode = "A"; //We think we need it in actuator but are not sure
        if (speedFlag)
        {
            if (analog == 1500)
                desiredSpeed = -1000;
            else
                desiredSpeed = mapAPM(analog);
            if (desiredSpeed > 100)
                desiredSpeed = 100;
            throttleORsteer = "V";
            toSend = commandType + "," + commandMode + "," + throttleORsteer + "," + desiredSpeed;
            Serial2.println(toSend);
            //Serial.print(millis());
            //Serial.print(toSend);
        }
        else
        {
            desiredSteering = mapAPM(analog);
            throttleORsteer = "W";
            toSend = commandType + "," + commandMode + "," + throttleORsteer + "," + desiredSteering;
            Serial2.println(toSend);
            //Serial.print(toSend);
        }
        speedFlag = !speedFlag;
    }
```
else { Serial.print("Estop\n"); // if the desired gear does not match actual }

void loop() {
  unsigned long currentMillis = millis();
  if (currentMillis - previousMillis >= interval) {
    // save the last time you blinked the LED
    previousMillis = currentMillis;
    if (heartbeat)
      heartbeat = false;
    else
      eStop = false; // true; // If car randomly eStops this is why
    // Initialize the required buffers
    mavlink_message_t msg;
    uint8_t buf[MAVLINK_MAX_PACKET_LEN];

    // Pack the message
    mavlink_msg_heartbeat_pack(syseid, compid, &msg, type, autopilot, base_mode, custom_mode, system_status);

    // Copy the message to the send buffer
    uint16_t len = mavlink_msg_to_send_buffer(buf, &msg);

    // Send the message with the standard UART send function
    Serial1.write(buf, len);
  }
  // Looks for the Lidar digital pin
  if (digitalRead(LidarPin) == HIGH) {
    // Serial.print("Lidar");
    // eStop = true;
    // }
    comm_receive();
}

void comm_receive() {
  mavlink_message_t msg;
  mavlink_status_t status;

  // COMMUNICATION THROUGH EXTERNAL UART PORT (XBee serial)
  while (Serial1.available() > 0)
  {
    uint8_t c = Serial1.read();
    // Try to get a new message
    if (mavlink_parse_char(MAVLINK_COMM0, c, &msg, &status)) {
      // Handle message
      switch (msg.msgid) {
      case MAVLINK_MSG_ID_HEARTBEAT:
        heartbeat = true;
        eStop = false;
        // Serial.print("Heartbeat: ");
        // Serial.println(mavlink_msg_heartbeat_get_base_mode(&msg));
        break;
      case MAVLINK_MSG_ID_SERVO_OUTPUT_RAW:
        // Serial.print("servo3: ");
        rc_apm_function(mavlink_msg_servo_output_raw_get_servo3_raw(&msg));
      }
B.2 Vehicle Mega Code

//RSL Rover 2014
//Vehicle Mega code

#include <AltSoftSerial.h>
AltSoftSerial mySerial; //RX:48, TX:46

//Strings (for more info see string initializations in setup)
String console_input_string;
String selfDrive_string;
String command_type;
String command_mode;
String throttle_or_steer;
String steering_command;
String speed_string_to_send;
String steering_string_to_send;
String gear_string_to_send;
String desired_gear;
String current_gear;
String string_from_motor_controller;
String mc_state;
String steering_query;
String feedback_to_console;
String feedback_to_console_prefix;
int voltage_input = 0; //Integer placeholder
for analogRead of voltage divider circuits
float Twenty_Four_V_Voltage = 0; //24 Volt system voltage
int Twenty_Four_V_Voltage_pin = 8; //24 Volt system voltage input pin
float Twelve_V_Voltage = 0; //12 Volt system voltage
int Twelve_V_Voltage_pin = 9; //12 Volt system voltage input pin
int temp_count = 0; //Counter used to ensure temp_start_time begins timing when the temperature light first comes on
unsigned long temp_start_time = 0; //Absolute time recorded when the temperature light first turns on
unsigned long temp_end_time = 0; //Absolute time recorded every time an iteration occurs with the temperature light on
unsigned long temp_time = 0; //Time that the temperature light has been on (temp_end_time - temp_start_time)
int temp_time_limit = 20000; //Time in milliseconds to allow the temperature light to stay on before activating emergency stop (important safeguard to prevent serious engine damage)
int threshold_warning_12_v = 11; //12 Volt threshold to give low battery warning on console
int threshold_warning_24_v = 21; //24 Volt threshold to give low battery warning on console
int threshold_e_stop_12_v = 9; //12 volt threshold to activate emergency stop
int threshold_e_stop_24_v = 19; //24 volt threshold to activate emergency stop
int e_stop_state = LOW; //if high, e-stop will be activated (acts as a toggle and if statements can be added anywhere in the code to toggle emergency stop mode on)
int Serial_timeout = 100; //Set the serial timeout for hardware serial ports
int temp_warning = 10; //Digital input pin for vehicle’s temp warning light
int reverse = 9; //Digital input pin for vehicle’s reverse gear light
int neutral = 8; //Digital input pin for vehicle’s neutral gear light
int low = 7; //Digital input pin for vehicle’s low gear light
```c
int high = 6;  // Digital input pin for vehicle's high gear light
int ebrake_relay_pin = 2;  // Emergency brake relay pin
int horn_relay_pin = 3;  // Horn relay pin
int e_stop_relay_pin = 4;  // Emergency stop relay pin
int e_brake_state = HIGH;  // Emergency brake state: HIGH is on, LOW is off
int contact_with_console = LOW;  // Once the Vehicle Mega makes initial contact with the console, this state turns HIGH
int beacon_relay_pin = 5;  // Relay pin for beacon on rolloff
int LidarPin = 51;  // Used to control the intermittent sending of data
int counter = 0;  // Used to send a desired speed to the speed controller when changing gears
int wheel_speed = 0;  // Current wheel speed
int wheel_speed_pin = 11;  // From the Axle Tachometer Interpreter
int gear_position = 0;  // Desired position to send to gear actuator
int channel = 1;  // Channel used to formulate strings to send to motor controllers (either 1 for steering or 2 for transmission command)
int steering_position = 0;  // Desired steering position parsed from console
int act_steering_position = 0;  // Current steering position as queried from the steering motor controller (to be sent as feedback to the console)
int comma_index_1;  // Index of first comma in a string (for parsing)
int comma_index_2;  // Index of second comma in a string (for parsing)
int comma_index_3;  // Index of third comma in a string (for parsing)
int dead_man_timeout = 750;  // If the Vehicle Mega loses contact with the console for more than the dead_man_timeout (in milliseconds), the emergency stop will be hit
unsigned long e_stop_time_1 = 0;  // Records the start time when the last console contact occurred
unsigned long e_stop_time_2 = 0;  // Records the end time when the next console contact occurred
unsigned long e_stop_time_2 and e_stop_time_1 (compared to dead_man_timeout)
unsigned long a;
unsigned long b;

void setup ()  // Runs before the main loop to initialize everything
{
    pinMode(ebrake_relay_pin, OUTPUT);  // Sets the emergency brake relay pin to output (same as parking brake)
    digitalWrite(ebrake_relay_pin, e_brake_state);  // Writes the startup emergency brake state to emergency brake pin
    pinMode(e_stop_relay_pin, OUTPUT);  // Sets the emergency stop relay pin to output
    digitalWrite(e_stop_relay_pin, e_stop_state);  // Writes the startup emergency stop state to emergency stop pin
}
```

pinMode(horn_relay_pin, OUTPUT);  //Sets the horn relay pin to output
digitalWrite(horn_relay_pin, LOW);  //Writes the startup horn state to horn pin (LOW is off)

pinMode(beacon_relay_pin, OUTPUT);  //Sets the beacon relay pin to output
digitalWrite(beacon_relay_pin, LOW); //Writes the startup emergency stop state to emergency stop pin

// Open serial communications and wait for port to open:
Serial.begin(9600);  //Serial to/from USB or serial monitor (sets baud rate and opens serial port)
Serial.setTimeout(Serial_timeout); //If the serial buffer misses the '' character, it will read a really long string. Setting the timeout ensures that if the controller receives a long garbage string, it will not waste time reading it

Serial1.begin(9600);  //Serial to/from the Console (sets baud rate and opens serial port)
Serial1.setTimeout(Serial_timeout); //If the serial buffer misses the '' character, it will read a really long string. Setting the timeout ensures that if the controller receives a long garbage string, it will not waste time reading it

Serial2.begin(115200);  //Serial to/from steering and transmission motor controller (sets baud rate and opens serial port)
Serial2.setTimeout(Serial_timeout); //If the serial buffer misses the '' character, it will read a really long string. Setting the timeout ensures that if the controller receives a long garbage string, it will not waste time reading it

Serial3.begin(9600);  //Serial to/from Speed Controller (sets baud rate and opens serial port)
Serial3.setTimeout(Serial_timeout); //If the serial buffer misses the '' character, it will read a really long string. Setting the timeout ensures that if the controller receives a long garbage string, it will not waste time reading it

mySerial.begin(1200);  //Serial to/from Speed Controller (sets baud rate and opens serial port)
mySerial.setTimeout(Serial_timeout);

pinMode(48, OUTPUT);  //Sets the temp_warning pin as an input (HIGH or LOW)

pinMode(reverse, INPUT);   //Sets the reverse gear pin as an input (HIGH or LOW)
pinMode(neutral, INPUT);    //Sets the neutral gear pin as an input (HIGH or LOW)
pinMode(low, INPUT);        //Sets the low gear pin as an input (HIGH or LOW)
pinMode(high, INPUT);       //Sets the high gear pin as an input (HIGH or LOW)

pinMode(wheel_speed_pin, INPUT);  //Sets the wheel speed pin as an input (PWM)

consol_input_string = String("");    //String from console
selfDrive_string = "";    //Parsed from consol_input_string: C for command, ? for queries (not yet involved), etc
command_type = String("");     //Parsed from consol_input_string: A for actuator, S for speed control modes
command_mode = String("");    //Parsed from consol_input_string: W for steering command, V for speed related commands
steering_command = String(""); //Parsed from consol_input_string: Value from -1000 to 1000
steering_string_to_send = String("");  //Formulated string to send as a steering motor command to steering and transmission motor controller
speed_string_to_send = String(""");  // Formulated string to send as a command to speed controller
gear_string_to_send = String(""");  // Formulated string to send as a transmission motor command to steering and transmission motor controller
desired_gear = String("");  // Formulated string to send as a transmission motor command to steering and transmission motor controller
console_input_string: H for high, L for low, N for neutral, R for reverse, P for park
current_gear = String("N");  // Current gear that the vehicle is in: H for high, L for low, N for neutral, R for reverse, P for park
steering_query = String("?TR 1");  // Query to be sent to steering and transmission motor controller (Asks motor controller what the current steering position is as a value from -1000 to 1000)
string_from_motor_controller = String("");  // String sent from steering and transmission motor controller
mc_state = String("Ready");  // State of the steering and transmission motor controller: "Ready" when ready to take commands "Starting" when performing startup procedure
feedback_to_consol = String(""");  // Feedback string to console includes wheel speed, current gear, and current steering position
feedback_to_consol_prefix = String("F");  // Prefix for feedback to console so console recognizes this as feedback and not an error string
suffix = String("\r");  // Return character to send at the end of command or query to motor controller (denotes the end of a string of data)
space = String(" ");  // Space needed in motor controller commands
comma = String(",");  // Comma used mainly to separate variables for data logging
position_prefix = String("!g");  // "!g" is how absolute position commands to motor controllers begin
A_status = String("");  // String parsed from aux_string to indicate that auxiliary button A is in the on position ("A" if read HIGH, "X" if read LOW)
B_status = String(""");  // String parsed from aux_string to indicate that auxiliary button B is in the on position ("B" if read HIGH, "X" if read LOW)
C_status = String("");  // String parsed from aux_string to indicate that auxiliary button C is in the on position ("C" if read HIGH, "X" if read LOW)
D_status = String("");  // String parsed from aux_string to indicate that auxiliary button D is in the on position ("D" if read HIGH, "X" if read LOW)
E_status = String("");  // String parsed from aux_string to indicate that auxiliary button E is in the on position ("E" if read HIGH, "X" if read LOW)
F_status = String("");  // String parsed from aux_string to indicate that auxiliary button F is in the on position ("F" if read HIGH, "X" if read LOW)
horn_status = String("");  // String parsed from aux_string to indicate that auxiliary button H (Horn) is in the on position ("H" if read HIGH, "X" if read LOW)
error_string = String("XXXX");
error_string_previous = String("XXXX");
armed_status = String("X");
temp = String("X");
voltage = String("X");

while (!Serial1.available()) {
    delay(50);
}
void error_check() //Checks the vehicle’s systems for errors (currently the only errors being checked for are temperature light, under voltage, and letting the console know that the vehicle is armed)
{
if (digitalRead(temp_warning) == HIGH) //Checks the digital pin that is tapped into the temperature light on the dash (if the temperature light is on, the pin will read HIGH)
{
  temp = String("T"); //Puts a "T" in the designated temperature warning place in error_string (indicating that the temperature light is on)
  temp_count = temp_count + 1; //Counts iterations that the temperature light has been on
  if(temp_count == 1) //Ensures that the timer will start for the temperature light when the light first comes on
  {
    temp_start_time = millis(); //Assigns the start time on the first iteration that the temperature light has been on
  }
  temp_end_time = millis(); //Assigns the end time each iteration that the temperature light is on for
  temp_time = temp_end_time - temp_start_time; //Calculates the total time that the temperature light has been on for
  if(temp_time >= temp_time_limit) //Checks to see if the temperature light has been on for longer than the designated temperature time limit
  {
    e_stop_state = HIGH; //If the temperature light has been on for longer than the temperature time limit, the emergency stop is activated
    Serial.println("temp");
  }
else //If the temperature light is off
{
  temp = String("X"); //Puts a "X" in the designated temperature warning place in error_string (indicating that the temperature light is off)
  temp_count = 0; //Resets the counter
}
error_string = "E" + armed_status + voltage + temp; //Formulates error string
if(error_string != error_string_previous) //Only send error string to the console if the status of errors has changed
{
  Serial.println(error_string); //Send the error string to the console
  error_string_previous = error_string; //Set error_string_previous = error_string
}
```c
void check_voltage() //Function to check the on-vehicle voltage levels
{
  voltage_input = analogRead(Twelve_Voltage_pin); //Reads analog value
  Twelve_Voltage = voltage_input * .01468; //Scales voltage_input
  voltage_input = analogRead(Twenty_Four_Voltage_pin); //Reads analog value
  Twenty_Four_Voltage = voltage_input * .03205; //Scales voltage_input

  if(( Twelve_Voltage <= threshold_e_stop_12_v ) || ( Twenty_Four_Voltage <= threshold_e_stop_24_v )) //Checks to make sure voltage levels are above emergency stop threshold levels and activates emergency stop if they are not
  {
    Serial.println("voltage");
    e_stop_state = HIGH;
  }
  else if(( Twelve_Voltage <= threshold_warning_12_v ) || ( Twenty_Four_Voltage <= threshold_warning_24_v )) //Checks to make sure voltage levels are above temperature light warning threshold levels and activates temperature light if they are not
  {
    voltage = String("V"); //Puts a "V" in the designated temperature warning place in error_string (indicating that one of the vehicle’s systems is under voltage)
  }
  else
  {
    voltage = String("X"); //Puts a "X" in the designated temperature warning place in error_string (indicating that the vehicle’s voltage levels are not too low)
  }

  void aux_switch_parse() //Function to parse the auxiliary switch string from console
  {
    A_status = console_input_string.substring(2, 3); //Status of rocker
      switch A on console("A" for on, "X" for off)
    B_status = console_input_string.substring(3, 4); //Status of rocker
      switch B on console("B" for on, "X" for off)
    C_status = console_input_string.substring(4, 5); //Status of rocker
      switch C on console("C" for on, "X" for off)
    D_status = console_input_string.substring(5, 6); //Status of rocker
      switch D on console("D" for on, "X" for off)
    E_status = console_input_string.substring(6, 7); //Status of rocker
      switch E on console("E" for on, "X" for off)
    F_status = console_input_string.substring(7, 8); //Status of pushbutton F
      on console("F" for on, "X" for off)
    on console("F" for on, "X" for off)
    horn_status = console_input_string.substring(8, 9); //Status of pushbutton H
      on console("H" for on, "X" for off)

    if(horn_status == "H") //If the horn button on the console has been pressed, activate the horn relay
    {
      digitalWrite(horn relay_pin , HIGH);
    }
    else //If the horn button on the console has not been pressed, make sure the horn is off
    {
      digitalWrite(horn relay_pin , LOW);
    }
```

43
if (B_status == "B") {  // sets the autopilot
    flag
    brake = true;
}
else if (B_status == "X") {
    brake = false;
} 
Serial.println(consol_input_string);
if (A_status == "A") {  // sets the autopilot
    flag
    selfDrive = true;
}
else if (A_status == "X") {
    selfDrive = false;
}
}

void gear_change(String consol_input_string) {  // Function called when a
gear change is desired

desired_gear = consol_input_string.substring(comma_index_2+1);  // Parses the desired
gear from consol_input_string
if (desired_gear != current_gear) {  // Only changes if the
desired gear and current gear are different
    desired_speed = -500;  // Sets up a string to
    // sent to the Speed Controller to apply the brake while the gear is changed so that
    // throttle will be zero and the brake will be applied for you if you are on a hill
    // changing gears
    command_type = String("C");
    command_mode = String("A");
    throttle_or_steer = String("V");
    speed_string_to_send= space + command_type + comma + command_mode + comma +
    throttle_or_steer + comma + desired_speed;
    Serial3.println(speed_string_to_send);  // Sends string to Speed
    Controller to apply brakes

    if (desired_gear == "H") // If desired gear is
        High
        
        gear_position = 1000;  // Transmission actuator
        position associated with High
e_brake_state = LOW;  // Ensures emergency
        digitalWrite(ebrake_relay_pin, e_brake_state);
    }
else if (desired_gear == "L") // If desired gear is Low
        
        gear_position = 103;  // Transmission actuator
        position associated with Low
e_brake_state = LOW;  // Ensures emergency
        brake is off
        digitalWrite(ebrake_relay_pin, e_brake_state);
    }
else if (desired_gear == "N") // If desired gear is
        Neutral
        
        gear_position = -449;  // Transmission actuator
        position associated with Neutral
        
}
e_brake_state = LOW; //Ensures emergency brake is off
digitalWrite(ebrake_relay_pin, e_brake_state);

else if(desired_gear == "R") //If desired gear is Reverse
    gear_position = -1000; //Transmission actuator position associated with Reverse
e_brake_state = LOW; //Ensures emergency brake is off
digitalWrite(ebrake_relay_pin, e_brake_state);

else if(desired_gear == "P") //If desired gear is Park
    gear_position = -449; //Transmission actuator position associated with Neutral
e_brake_state = HIGH; //Applies parking brake
digitalWrite(ebrake_relay_pin, e_brake_state);

else
    desired_gear = current_gear;

channel = 2;
gear_string_to_send = position_prefix + space + channel + space + gear_position;
Serial2.println(gear_string_to_send);
current_gear_function();

else
    changing, the brake is released //When the gear is done changing, the brake is released
    desired_speed = 0; //Brake off and throttle at zero
    speed_string_to_send = space + space + command_type + comma + command_mode + comma + throttle_or_steer + comma + desired_speed;
    if(Serial1.available()) Serial1.readStringUntil('\r'); //clear serial buffer
    Serial3.println(speed_string_to_send + ' ' + channel + ' ' + serial_buffer + ' ' + desired_speed);

void current_gear_function() //Updates the current gear
{
    if(digitalRead(high) == HIGH) //If the high gear indicator light on the vehicle is on
        current_gear = "H";

    else if(digitalRead(low) == HIGH) //If the low gear indicator light on the vehicle is on
        current_gear = "L";
}
else if (digitalRead(neutral) == HIGH) // If the neutral gear indicator light on the vehicle is on
    if (e_brake_state == LOW) // If parking brake is off, and vehicle in neutral, the vehicle is simply in neutral
        current_gear = "N";
    else if (e_brake_state == HIGH) // If parking brake is on, and vehicle in neutral, the vehicle is in park
        current_gear = "P";
}
else if (digitalRead(reverse) == HIGH) // If the reverse gear indicator light on the vehicle is on
    current_gear = "R";
}

void control_command(String console_input_string) // Called when the string from Vehicle
    Mega is a control command
{
    comma_index_1 = console_input_string.indexOf(\',\'); // Index for parsing
    comma_index_2 = console_input_string.indexOf(\',\', comma_index_1 + 1); // Index for parsing
    command_mode = console_input_string.substring(comma_index_1 + 1, comma_index_2); // Parses the second character to determine the command mode
    if (command_mode == "A")
    {
        comma_index_3 = console_input_string.indexOf(\',\', comma_index_2 + 1); // Index for parsing
        throttle_or_steer = console_input_string.substring(comma_index_2 + 1, comma_index_3); // Parses the third character to determine whether the command is a speed or steering command (V or W)
        if (throttle_or_steer == "W") // If a steering command
        {
            channel = 1; // Channel 1 for steering commands to steering and transmission motor controller
            steering_command = console_input_string.substring(comma_index_3 + 1); // Parses steering command value from -1000 to 1000
            steering_position = steering_command.toInt(); // Transforms this value from string to integer
            steering_string_to_send = position_prefix + space + channel + space + steering_position; // Formulation of string
            Serial2.println(steering_string_to_send); // Sending steering command to steering and transmission motor controller
        }
    }
}
else if (throttle_or_steer == "V")
    //If a speed related command
    if (current_gear == desired_gear)
        //If the vehicle is in the correct gear, send on speed commands (helps ensure that vehicle doesn't give throttle input during gear change)
        Serial3.println(console_input_string);
    //String to send to speed controller

if (command_mode == "G" && abs(wheel_speed) < 1)
    //If the console input string is a gear change string and the absolute value of wheel speed is below 1 mph (Trying not to grind gears!)
    gear_change(console_input_string);
    //Call gear change function

void loop()
    //Main loop (iterates over and over)
{"SELF_PPCP"
    if (selfDrive && digitalRead(LidarPin) == HIGH)
        //Serial.println("Lidar");
        e_stop_state = HIGH;
    a = millis();
    if (Serial1.available())
        contact with console
        e_stop_time_1 = millis();
        //Record emergency stop start time (if contact with console is lost, the start time will stop updating itself)
        console_input_string = Serial1.readStringUntil(\'\r\');
        //Read console_input_string
    }
    //Serial.println(mySerial.available());
    if (mySerial.available())
        selfDrive_string = mySerial.readStringUntil(\'\r\');
        //Serial.println(selfDrive_string);
        //Serial.println("New LINE");
    }
    e_stop_time_2 = millis();
    //Record emergency stop end time
    e_stop_time = e_stop_time_2 - e_stop_time_1;
    //Time between start and stop for dead man switch
if (e_stop_time >= dead_man_timeout) // If difference in time
    { // is greater than the dead-man timeout, toggle on the emergency stop system
        Serial.println("commands");
        e_stop_state = HIGH; // Indicates the emergency stop system is engaged
    }

if (e_stop_state == HIGH) // If emergency stop state is HIGH, write low to the emergency stop relay to engage the emergency stop system
    {
        digitalWrite(e_stop_relay_pin, LOW);
    }
else // Otherwise write the emergency stop relay pin HIGH to keep the emergency stop system off
    {
        digitalWrite(e_stop_relay_pin, HIGH);
    }

if (mc_state == "Ready") // Ensuring that the motor controller is not executing its startup procedure
    {
        if (selfDrive)
        {
            //Serial.print(selfDrive_string);
            if(selfDrive_string.charAt(1)=='C'||selfDrive_string.startsWith("C"))
            {
                control_command(selfDrive_string);
            }
            else if(selfDrive_string.startsWith("C")
            {
                control_command(selfDrive_string);
            }
            else if(selfDrive_string.startsWith("A")||console_input_string.charAt(1)=='A')
            {
                aux_switch_parse();
            }
            else if(selfDrive_string != "" && selfDrive_string != "}
            {
                //Serial.println("Estop");
                e_stop_state = HIGH;
            }
        }
        else{ // These should both be identical if statements but this one is needed when the serial interface is used (Serial interface will not get through without this)
            if(console_input_string.charAt(1)=='C'||console_input_string.startsWith("C"))
            {
                console_input_string is a command, call control_command function
                {
                    if (console_input_string.charAt(5)=='V'&&brake)
                        control_command("C.A.V,-1000");
                    else
                        control_command(console_input_string);
                }
            }
        }"
*/
else if (consol_input_string.startsWith("A") || consol_input_string.charAt(1) == 'A')
    // If the consol_input_string is a auxiliary switch string, call aux_switch_parse function
    aux_switch_parse();
*/
else if (consol_input_string.charAt(1) == 'A')
    // These should both be identical if statements but this one is needed when the serial interface is used (Serial interface will not get through without this)
    aux_switch_parse();
*/
}

if (Serial2.available())
    // Read String from transmission and steering motor controller
    { string_from_motor_controller = Serial2.readStringUntil('r');
    }

if (string_from_motor_controller.startsWith("TR"))
    // If the string is a response to the steering position query
    { string_from_motor_controller = string_from_motor_controller.substring(3);
        // Parse the numerical value from the query response
        act_steering_position = string_from_motor_controller.toInt();
        // Convert this numerical value from a string to integer
    }

else if (string_from_motor_controller.startsWith("Starting"))
    // If the motor controller is executing its startup procedure
    { mc_state = string_from_motor_controller;
        // Set mc_state to "Starting"
        armed_status = String("X");
    }

else if (string_from_motor_controller.startsWith("Ready"))
    // When the motor controller is finished executing the startup procedure, the mc_state changes to "Ready"
    { mc_state = string_from_motor_controller;
        digitalWrite(beacon_relay_pin, HIGH);
        // Lights up the beacon to demonstrate vehicle is armed and ready
        armed_status = String("A");
    }

if (counter > 10)
    // Only sends the steering query and console feedback every 10 iterations
    { counter = 0;
        Serial2.println(steering_query);
    // Prints steering query
feedback_to_consol = feedback_to_consol_prefix + comma + act_steering_position + comma + wheel_speed + comma + current_gear;
Serial1.println(feedback_to_consol);  //Prints console feedback

counter = counter +1;  //Iteration

wheel_speed = pulseIn(wheel_speed_pin, HIGH, 5000);  //Read PWM (in nano-seconds) from Axle Tachometer Interpreter
wheel_speed = wheel_speed * (.0398) - 40;  //Turn PWM into a speed from -40 to +40 mph
current_gear_function();  //Function call to determine the current gear

check_voltage();  //Function call to check voltages
error_check();  //Function call to check the vehicle for errors and either alert the console or activate the emergency stop
b = millis();
Serial.println(b-a);
Appendix C

LIDAR Bracket

[Diagram of LIDAR Bracket with dimensions and notes]