Spring 2021

Light Loaded Automated Guided Vehicle

Marcus Radtka
mpr43@zips.uakron.edu

Nazar Paramashchuk
np52@zips.uakron.edu

Lawrence Shevock
lns51@zips.uakron.edu

Light Loaded Automated Guided Vehicle

Final Design Report

Design Team 15-A

Nazar Paramashchuk

Marcus Radtka

Lawrence Shevock

Faculty Advisor:

Dr. Hamad Bahrami

Date Submitted:

23-APR-2021

Individual Contribution

The following document outlines the locomotion system (LS) ultimately responsible for controls, movement of the vehicle, user input and feedback of the system. References will be made to the navigation system (NS), which is the other half of the team providing suggested routes and additional safety information. Together the locomotion and navigation systems comprise the lightly loaded automated guided vehicle (LLAGV) as a design project. As the computer engineer on the locomotion system (LS) I was responsible for implementing control theory for driving the LLAGV as well as the user interface and feedback mechanisms. The implementation of this software application was in the Python scripting language. With the complexity of this project a more capable embedded processor was permitted, namely a Raspberry Pi, to take advantage of its multiple cores and larger RAM options. This allowed for a multi-threaded design in the end application as well the use of an in-memory database. The software application built handled both a manual control and an autonomous control interface. A manual control program was made allowing the user to drive the LLAGV with the triggers of an Xbox controller. The autonomous or follow mode control program was built using the NS data to follow the user's cellular device.

As mentioned, being responsible for the user interface meant creating an interface that was responsive, intuitive and provided meaningful feedback. What does that really mean? Well, the design was desired to be such that one not knowing of the project could start and use the machine with little to no initial guidance, similar to say turning over a brand-new riding mower. I was able to accomplish this through the use of push button switches with LED indicators and individually addressable LED strips to provide feedback to the user acknowledging the requests. One could infer that this style of UI meant button debouncing, synchronization across multiple threads as well as keeping safety the top priority.

One final comment for the locomotion system specifically; much of the control theory was developed by Lawrence Shevock and implemented by me, Marcus Radtka, with help from Lawrence. The PCB design, harnessing, power source and the power distribution was handled by Nazar Paramashchuk. Again, my responsibilities remained within applying the control theory and the user interface of the LLAGV.

Frontal Material

Table of Contents (i):

List of Figures(ii):

List of Tables(iii):

Glossary(iv):

AGV – Autonomous Guided Vehicle

GPIO – General Purpose Inputs Outputs

I2C – Inter Integrated Circuit (I squared C) Communication Bus

LLAGV – Light Load Autonomous Guided Vehicle

MC – Motor Controller

MOSFET – Metal Oxide Semiconductor Field Effect Transistor

PDM – Power Distribution Module

PDMCSC – Power Distribution Module Charging State Controller

PDMOSC – Power Distribution Module Operational State Controller

Pi – Raspberry Pi

PIH – Power Interface Harness

RAS – Risk Addressed State

SoC – State of Charge

VIH – Vehicle Interface Harness

0. Abstract

The objective of the locomotion system was to design and implement the mechanical, electrical, and software related functions to ensure the LLAGV had the capability of maneuvering its surroundings. The LLAGV's motors were represented in an open loop transfer function to utilize RPM feedback and a compensator when needed. The modeled compensator helped control the LLAGV's speed and acceleration, enabling further control of the LLAGV. The internal circuitry has the means to properly distributed power to all components and allowed the user to control the LLAGV to their desire. The application software within the LLAGV locomotion system (LLAGV-LS) had consideration for distance and angle variation, provided by the navigation system (NS) team where this information was pulled from an in-memory database. Changing the angle and distance, from the user, was done using motor control theory and application. The data along with feedback from the system provided a reliable and predictable means of driving the LLAGV's traction control system as well as incorporating input from the user and delivering a source of feedback to the user, ultimately creating a cohesive, intuitive interface for the user to take advantage of the convenience the LLAGV offered. The LLAGV also had basic object detection features in which the NS informs the LS Team B would inform Team A of an object in front of the LLAGV. The LLAGV then conducts the actions necessary to avoid the object. Key features are as included below.

- LLAGV maintains a distance of 3 to 10 feet with an average of 3 ft/sec
- LED lights dictate the state of charge and state and direction intent
- LLAGV lasts for a minimum of two hours at a full charge
- LLAGV carries a light load of up to 30 pounds

1. Problem Statement

1.1. Need

According to *OSHA.gov "*Carrying loads on one shoulder, under an arm, or in one hand, creates uneven pressure on the spine." The need is to assist individual(s) in transporting light loads from a nonspecific source to destination.

1.2. Objective

Design an automated guided vehicle (AGV) capable of moving a light load placed on the vehicle from a source to destination. LLAGV will be capable of following an individual around to deliver the load to the individuals desired source. Detailed safety/object avoidance behavior while adaptability to changing surroundings.

1.3. Background

There has been an exponential growth of smart technology that changes how people interact with the world around them. Within the broad growth of smart technology, there is a development of automated guided vehicles (AGV's) that are seen within the work environment such as manufacturing and general public such as transportation. However, there is a lack of AGV's being utilized by the public in which the AGV assists in carrying light loads from a source to destination with little to no interaction in an efficient, safe, and cost-effective manner. The goal of our team's project is to create and demonstrate an AGV that will transport a light load, of up to thirty pounds, around most environments while accounting for efficiency, safety, and cost-effectiveness.

The first AGV was founded in 1954 when Mac Barrett was given credit for the invention of the AGV in which a simple towing tractor followed an overhead wire. The advancements within technology have led to more sophisticated designs that are seen within manufacturing and automation. The basic concept of an AGV can be described as: "An AGV is a mobile

robot/vehicle used to transport materials in manufacturing environments, designed to receive and execute instructions, follow a path, and receive and distribute materials. The vehicles generally follow a path that can go in many directions and can usually be easily reconfigured according to the manufacturer's plant." [1] The theory of how AGV's are constructed can be looked at through the electronic, mechanical, and software design. There is more than one correct way of designing and implementing an AGV. The design will be primarily constructed on what the AGV will be tasked to do. The electrical and mechanical design will need to include a supportive frame for operation, multiple input sensors to determine exterior parameters, motors, variable frequency drives, and controller(s) for processing operations. From a software perspective our design will implement more of a free-range routing. "This routing algorithm is based on the route choice methodology from a model called NOMAD [10][11], a microscopic pedestrian behavioral model also developed at the Delft University of Technology. The algorithm is dynamic, because it uses real time information on planned trajectories of other vehicles for the determination of new routes. These routes are free range because they make use Dynamic Free-Range Routing for Automated Guided Vehicles." [3] With the compatibility of mechanical, electrical, and software implementations the basic theory of an AGV will operate to provide value to customer by ensuring the AGV has the necessary means to transport light loads from a non-determined source to non-determined destination.

Currently, one of the most popular applications for AGVs is manufacturing facilities. Here the automated vehicle is used "to transport materials, designed to receive and execute instructions, follow a path, and receive and distribute materials", [1]. In this application, AGVs are widely utilized as material handlers to replace what once would have been manual labor of moving materials across a factory floor. Another source describes AGVs in the manufacturing setting [2] as an "intelligent logistics handling robot" with a "predetermined path of travel". A reoccurring theme from both aforementioned sources is that the automated vehicle has a predetermined path; this suggests the vehicle has a specific route that has been programmed into the LLAGV system. The AGV system here encapsulates a main controller or base station and the automated vehicle itself. Both [1] and [2] mention how an AGV system can be divided into three main functions: dispatching/navigation, routing/layout, and scheduling /guidance respectively. The first function described as dispatching or navigation provides the AGV with a task of

picking up a load and delivering to some point on the grounds of the facility. Secondly, routing or layout, provides the AGV with a designated path to complete the task given. The third function mentioned, scheduling or guidance is the control system involved with the vehicle navigating it's given path. The scheduling attribute to the third function adds synchronization with other manufacturing processes, while the guidance attribute encapsulates the sensor data on-board the vehicle to make aware of the vehicle's surroundings. An AGV must be outfitted with some sort of sensing devices to communicate back to the base station and make internal decisions. An example of this may be as simple as an external button to be manually pressed when loading or unloading has completed; but as complex as navigating and completing tasks as a self-sufficient worker robot. One source, [2], lists four common types of communication that may be outfitted on an AGV in the manufacturing application. Those types of communication include wired, infrared light, radio, and wireless LAN. Of course, comparing these four examples of communication yields many advantages and disadvantages. Interestingly, "Electromagnetic guide is one of the more traditional ways of AGV guidance", utilizing the hardwire communication scheme listed earlier, [2]. Not surprisingly, this common hardwired scheme is often the cheapest and easiest of the communications schemes listed to implement, however not the most robust for large scale applications.

Even with how advanced technology has become in the modern age there are still many limitations to an AGV. As mentioned in the above paragraph [1] and in most designs out right now all use "predetermined path of travel". Now this predetermined path method already has its own set of limitations as shown in [3], where in this article the optimization of route length when traveling between multiple points of a predetermined path is an issue. Our implementation of an AGV with have added limitations because we won't be using predetermined paths, we will be using the AGV to take its load from a non-determined source to non-determined destination. The challenges this adds are ones that more fully autonomous vehicles run into, those being navigation with by just using terrain senor data along with safety concerns with collision avoidance. With navigation unlike most other AGV in the industry today we will be having to use strictly positional and terrain mapping sensors to determine where our vehicle is in its environment instead of being able to rely on a set path that's preprogramed into it. As discussed in [4], which is an article about automated shuttle vehicles used in coal mines, they talk about the

issues with navigation precision when turning to avoid obstacles or to simply turn around a corner. Now our vehicle won't be confined to such small areas as the mining vehicle would be, but accuracy of navigation and precision is still a concern. The other main limitation would be safety concerns that being collision avoidance with obstacles of any type. This is a limitation for both the AGV's used in the field today along with autonomous vehicles in general. In the article "Safety aspects of autonomous guided vehicles in automated warehouses" [5] the author lays out a great design method to use. He lists five steps to follow: hazard analysis, identification of the safety related systems, determination of the required safety level, design of the safety related systems, safety analysis. Following these guidelines, we should be able to avoid safety issues.

Acknowledging these details about current systems and their implementations, there will be many similarities and differences between our concept AGV and current designs. Parallel to the main goal of the concept, the vehicle will serve as transport for different objects and materials commonly carried by hand, just like in manufacturing environments. The vehicle will support a load weight rating of 30 pounds. Construction of the vehicle will have similar, if not identical, physical structures such as chassis, wheels, and bucket construction. The source of power will come from on-board batteries that will feed motor controllers and microprocessors. Modern methods and designs used with AGVs will be referenced and used for charging, controlling discharge, logic behavior during charge, lifetime management, and power management (NISSAN MOTOR CO., LTD, US Patent No. 9,325,192), The drive systems will also be closely related since the request for movement and direction will be the same format in the end, no matter the data processing that comes before to produce the movement instruction. Other components such as suspension, braking, steering control, roll over protection, and emergency stopping will be incorporated in similar manner to current designs.

The concept will differ significantly in that it will follow a host using dynamic positioning algorithms, as mentioned earlier, instead of following a pre-set path. This will allow the vehicle to decide when it needs to start, stop, turn, and at what speed to follow. Another key difference is that the vehicle will be capable of operation not only indoors, but outdoors as well. Apart from smooth and flat floors, it will navigate through common terrain around the home, such as grass, pavement with a certain degree of allowable slope, gravel, semi-flat dirt, and light

snow. This allows the AGV to not only perform indoor duties of an indoor manufacturing environment robot but assist with outdoor labor done at various times of the year. Furthermore, the methods of guidance differ from traditional methods mentioned in [2]. Since there is no set path to follow, the vehicle will use radar positioning sensors to follow a target wherever it might turn. The AGV will also have to mind its' surroundings, as well as satisfy many additional safety requirements introduced with the freedom to choose a path of travel. These additions require more sensors to provide data for the safety monitors, and changes to safety software not used in current designs.

The vehicle will be powered with a battery-operated electric motor. The battery must be rechargeable and there will be a need for a charging station for the vehicle's battery. Efficiency in electronic devices is a very important part of any design. An invention to manage and control the charging of battery cells for AGVs was patented by NISSAN MOTOR CO., LTD (US Patent No. 9,325,192). The patented system monitors the voltage or charge of the battery cells and sets a threshold value for when the battery cell requires to be recharged. For any application, the knowledge of how long the AGV can operate and when it requires charging is very important.

Some applications of the light load AGV will require the loading platform of the vehicle to be balanced and stabilized. The vehicle will need to be able to compensate for an unbalanced load. A patented invention by Paul George Doan (US Patent No. 8,527,153) is being developed to keep a loading platform level and balanced. The system utilizes sensors that are coupled to an electronic control center to operate an extension member that extends to the ground to level and balance the AGV. The capability of the AGV keep its loading platform level is very useful in many applications to keep the load safely on the vehicle when no humans are operating the machine. Some loads could be fragile and be damage by falling off the AGV.

Automated Guided Vehicles have provided services which allow people to benefit from the advancement of technology. An AGV allows for the transportation of a product or a person from a specific source to destination. There are multiple types of AGV's however, there is a lack of AGV's that are used by the general public. With the creation of a light load cost effective

AGV, this machine will transport light loads from an undescriptive source to destination. This will allow for more robust transporting options over fitted terrain.

1.4. Marketing Requirements

1. The system will be able to navigate throughout the environment in which it's placed in order to follow an individual to its destination.
2. The system will maintain a set distance from the user.
3. The system will travel at an average walking speed.
4. The system will have multiple safety features.
5. The system will be able to carry a light load[1].
6. The system will be rechargeable via rechargeable battery.
7. The system will provide state information as user feedback.
8. The system will travel on light terrain[2].
9. The system will have an intuitive HMI[3] to operate the device.

2. Engineering Analysis

2.1. Circuits

The LLAGV incorporated multiple circuits to meet all design and marketing requirements. In this context, the main sub-circuits were:

- Power Distribution Module Operational State Circuitry (PDMOSC)
- Power Distribution Module Charging State Circuitry (PDMCSC)
- Vehicle Interface Harness (VIH)
- Power Interface Harness (PIH)

The Power Distribution Module (PDM) is the device that governed all power distribution, current protection, and charging. The circuitry on this device existed in a printed circuit board

[1] Light Load: Payload of less than 30lbs.
[2] Light Terrain: grass, gravel, asphalt, cement surfaces all of which have a relatively flat surface. No inclines > 10˚ and no severe undulations.
[3] HMI: Human to Machine Interface. In this application will be comprised of various labeled push buttons and LEDs.

format. The PDM Operational State Controller (PDMOSC) governed all power distribution and safety monitoring while the system was in the normal operation mode under the software states of RAS (Risk Addressed State), autonomous, and neutral. It made up for half of the functionality of the PDM. The system microcontroller was the controlling entity that communicated with the motor controller to receive and send data and commands. The circuits had both inputs and outputs interfacing with the main microcontroller to control different functions such as pre-charge, relay actuation, and switching. These inputs were conditioned and amplified by power transistors to allow the switching of high current peripherals using the small signal outputs of the microcontroller. DC levels of 14.8, 5V, and 3.3V were available and routed to which ever circuit needs them.

An important job that the PDM had to accommodate signal acquisition circuitry. Temperature and voltage measurements were the bulk of this type of signaling. For temperature measurements, the TMP6131LPGM linear thermistor was chosen to act as the temperature sensor. It is linear type thermistor with a resistance of 10kOhms at 25 degrees Celsius. The temperature vs. resistance graph is shown below in Figure 1.

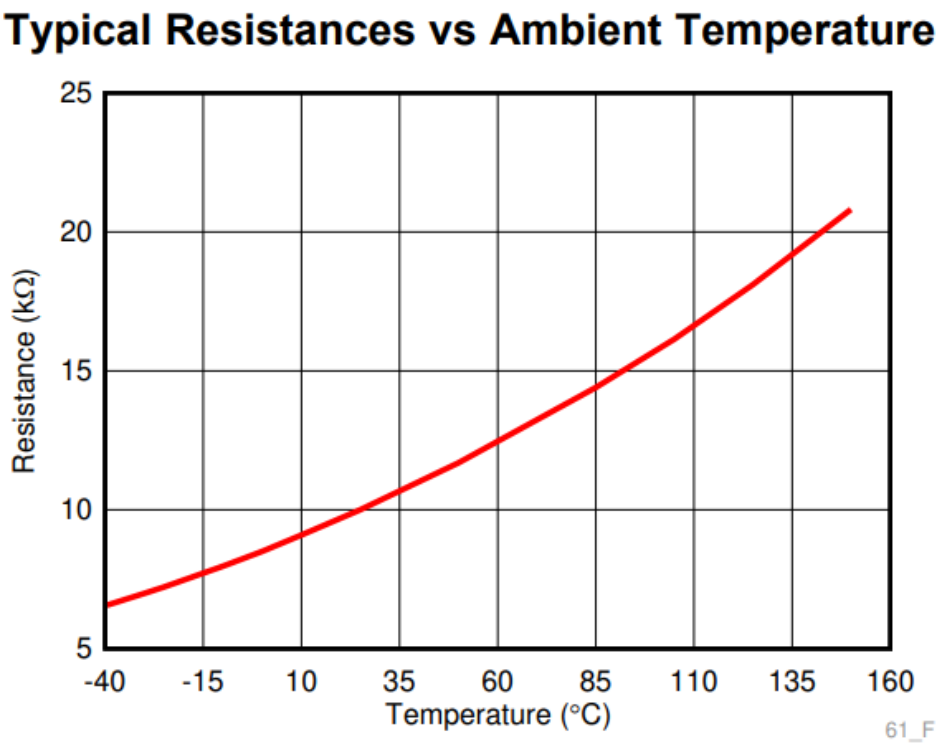


Figure 1: Resistance/temperature relations graph for TMP6131LPGM

With this thermistor, a simple voltage divider network was used to vary voltage proportional to the temperature exposed to the thermistor. The voltage can then be an input to the ADS1115 ADC chips that were used translate to digital signals and broadcasted to an I2C bus. A typical application of this network is shown in Figure 1.

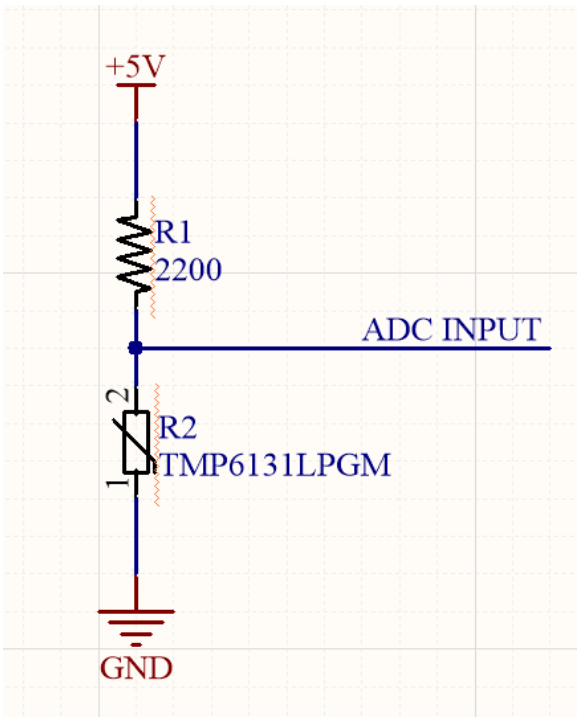


Figure 2: Typical application of the TMP6131LPGM

For the LLAGV, the range of measured temperature would range from 0 to 150 degrees Celsius. The ADS1115 ADC accepted an analog input of VDD-VSS. The PDM provided 5V for VDD and 0V for VSS, therefore the maximum measured analog value was 5V. To avoid saturation and measurement error, a range of 0.5 to 4.5V was applied globally to all temperature measurement types using this thermistor. The value of R1 is solved by simulating the highest maximum temperature event, which would provide a resistance of 20k ohms at 150 degrees Celsius. To solve for an appropriate value of R1, use the equation below.

$$\frac{5 * R_{\max temp\ of\ TPM6134}}{R_{\max temp\ of\ TPM6134} + R_1} = 4.5$$

In this case, an R1 value of 2200 ohms will allow an appropriate measuring range for the ADC proportional to the accepted range of temperatures the peripherals will operate at.

Many voltage measurements were made on different peripherals of the system. The measuring voltage is again at a range of 0.5V- 4.5V. The measured voltage was scaled down to what the ADC can measure, therefore Figure 2 Can be applied for this type of measurement.

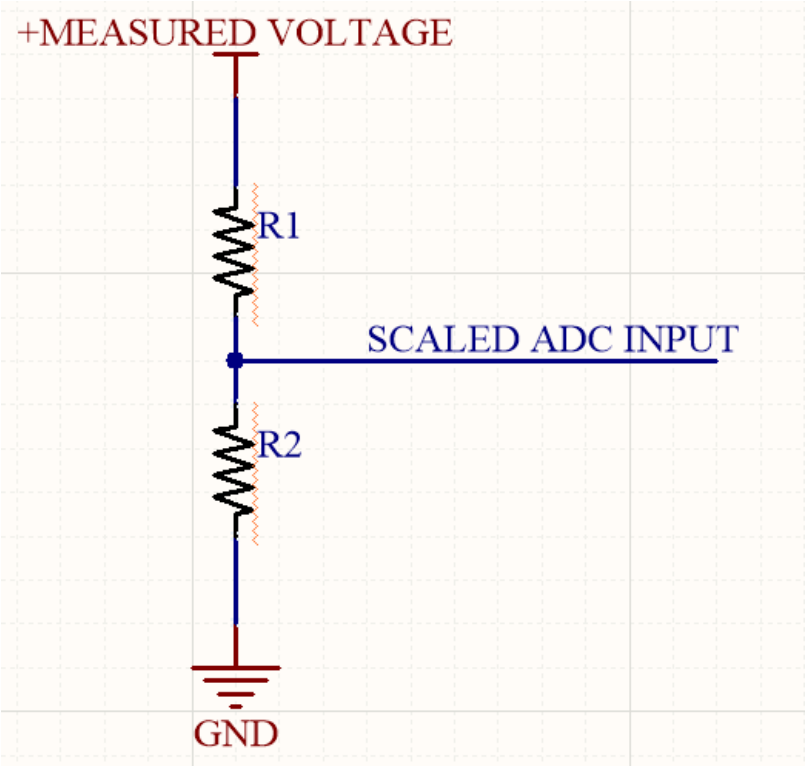


Figure 3 Typical application for voltage measurement scaling

One of the resistors will need to be pre-determined. A value of 10k ohms for R2 is appropriate to keep current consumption levels of the whole network to a minimum. The maximum voltage measured will need to produce an output of 4.5V at the ADC input, therefore the equation below can be used.

$$\frac{10{,}000 * Max\ Meausured\ Voltage}{R_1 + 10{,}000} = 4.5$$

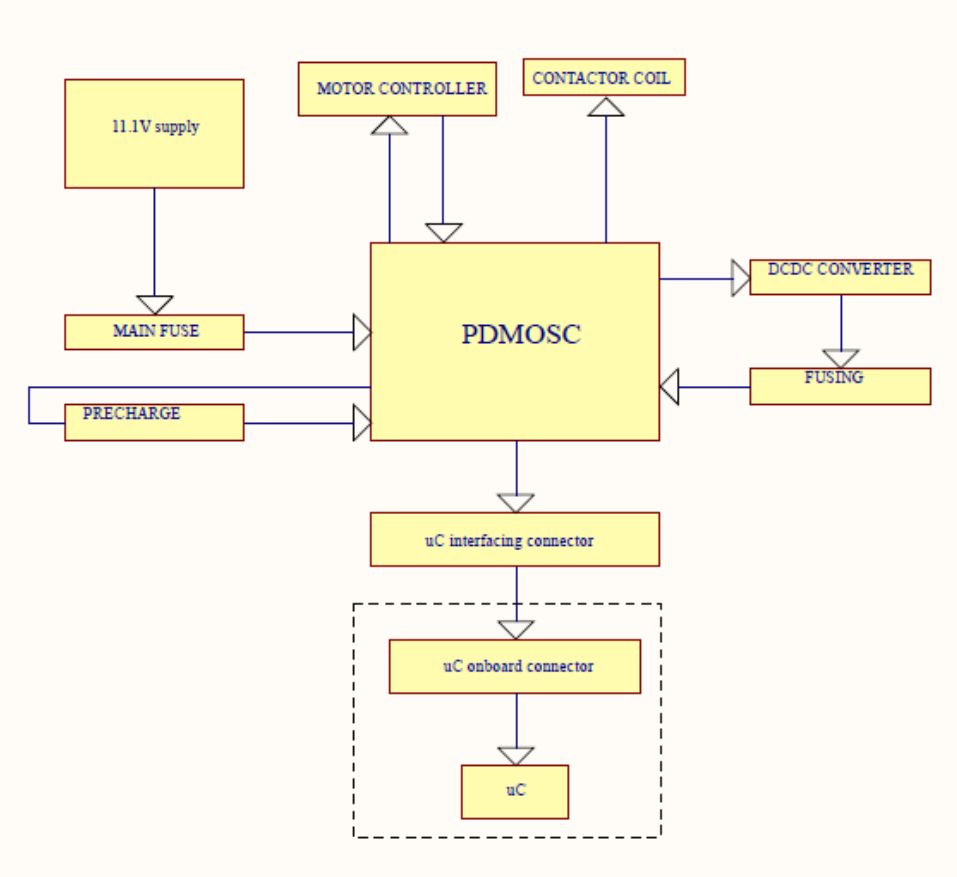


Figure 4 PDMOSC Level 2 Diagram

The PDM Operational State Controller (PDMOSC) was the main functional part of the PDM that brought all the systems of the LLAGV together. This system included all the measurement and most of the power distribution peripherals necessary for all subsystems within the AGV. The diagram of the system is shown in the figure above.

The PDM Charging State Controller (PDMCSC) functioned as the other half of the PDM. The charging function was provided by a charge controller and balancing was provided by onboard resistors, controlled by the main microcontroller. The PDMCSC circuitry is shown in the figure below.

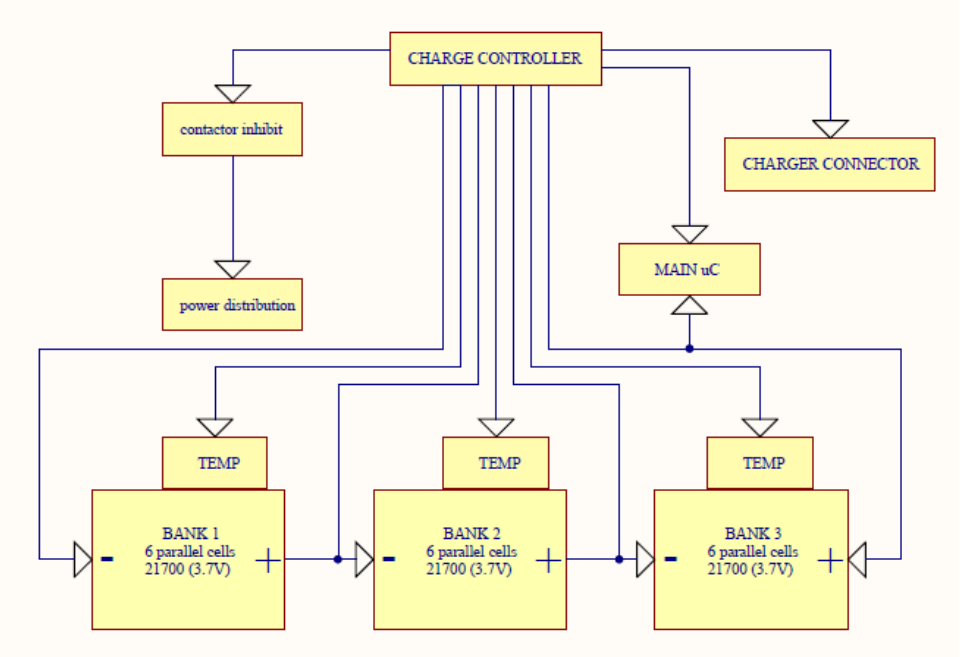


Figure 5 PDMCSC Level 2 Diagram

The Vehicle Interface Harness (VIH) interconnected the PDM, sensor/switch groups, and microcontrollers. It included communication lines between the system microcontrollers and the motor controller processor, as well as switch logic and analog sensor values.

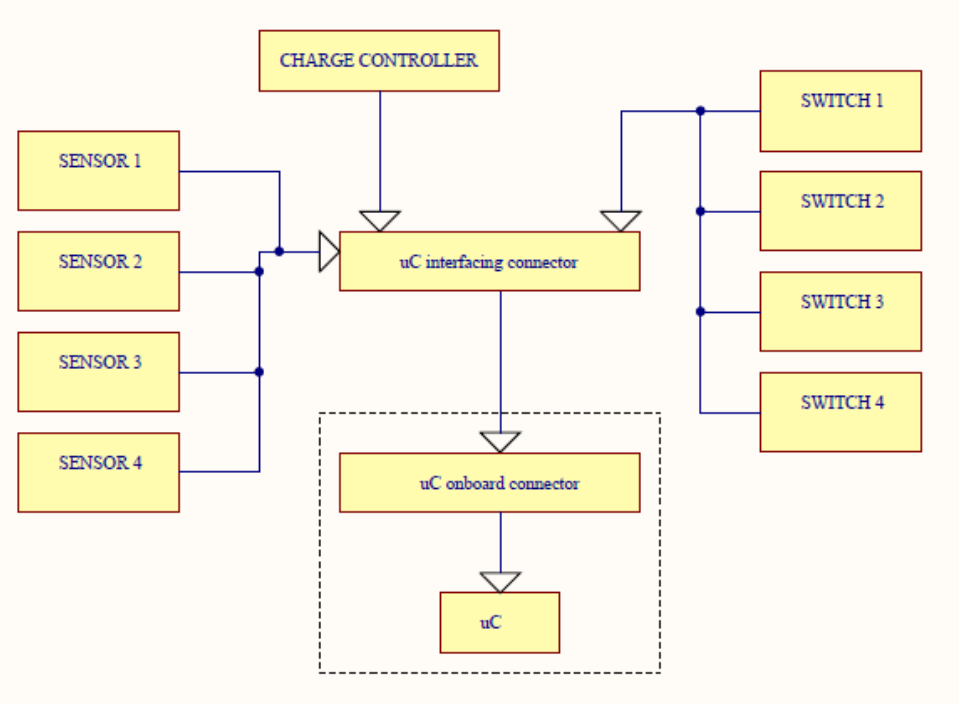


Figure 6 VIH Level 2 Diagram

The Power Interface Harness is the harness that was responsible for delivering high current between the battery pack, PDM, relay, motor controller, and motors. The main power net originated from the 14.8V battery pack and fed into F1, no more than 6” away from the pack. The net then split to lower-level fusing, and the contactor. The PIH continued to the motor controllers, where the motor controller provided two channels for locomotion, one channel per side.

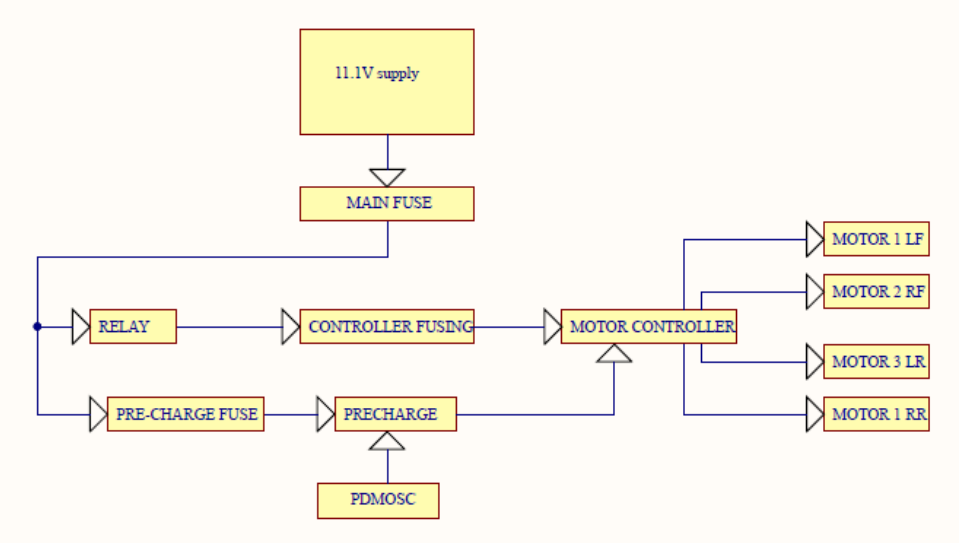


Figure 7 Level 2 Power Distribution and Charging

2.2. Electronics

The LLAGV contained a wide range of electronics, from PCB level components to the main microcontroller. All these parts were integrated within their respective subsystems, and communication will exist between these systems where needed.

Some analysis was conducted to compare various embedded processors for this LLAGV application. Ultimately, the decision was made based on processing capability, local storage (RAM) and various communication peripherals. After comparing several various embedded processors with similar capabilities, the embedded processor that meets or exceeds the needed specifications was the Raspberry Pi 4 Model B for the locomotion system - see Figure 8. This embedded processor provides the needed computational power, but more importantly the most RAM for the price with other peripherals. See **Error! Reference source not found.** comparing the embedded processors.

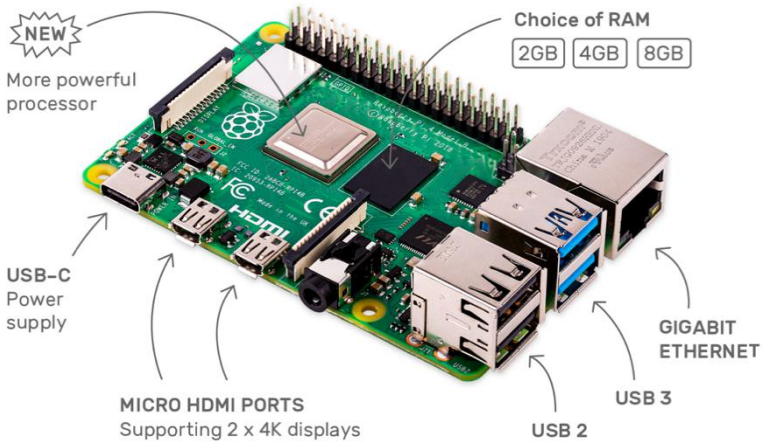


Figure 8 Raspberry Pi 4 Model B

Table 1 Embedded Processor Comparisons

Microcontroller:	Raspberry Pi 4B	Beagle Bone Blue (Beagle Bone Black + Robotics Cape)	ASUS Tinker Board	Arduino Tre
Processor	Cortex-A72 (ARM v8) 64-bit	Cortex-A8 AM335x	Rockchip Quad-Core RK3288 processor	TI Sitara AM335X ARM Cortex-A8
# of CPU Cores	4	1	4	1
Processor Speed	1.5GHz	1GHz	1.8Ghz	1GHz
RAM	1GB-8GB	512MB	2GB	512MB
I2C	YES	YES	YES	YES
UART	YES	YES	YES	YES
Serial/USB	USB 2.0/3.0	Serial/USB 2.0	USB 2.0	USB 2.0
CAN	NO	YES	NO	CAN
ADC's	NO	YES	NO	YES
GPIO	YES	YES	YES	YES
Operating Voltage	5V	5V-18V	5V	5V
Onboard Memory	SD Micro Card	SD Micro Card 4GB 8-bit eMMC Flash	SD Micro	SD Micro
Bluetooth	BT 5.0	BT 4.1	BT 4.0	NO
WiFi	2.4GHz/5.0GHz IEEE 802.11B/g/n/ac	2.4GHz IEEE 802.11B/g/n	2.4GHz IEEE 802.11B/g/n	NO
Ethernet	YES	NO	YES	YES
OS	Linux	Linux	TinkerOS/Linux/Android	Linux
Price	$80	$95	$45	$60

The Raspberry Pi 4 Model B (Pi) was the Linux based microcontroller chosen to govern the operations of the LLAGV. It was responsible for switching, measuring, communicating, and commanding different parts of the vehicle. This Pi has a quad-core 64-bit ARM-Cortex A72 processor that runs at 1.5Ghz with 4GB of LPDDR4 RAM. The features that the LLAGV utilized are the USB ports, I2C busses, PWM channels, and GPIO pins. The Pi required a stable 3A supply of 5V, which was provided by the PDM. The Pi processed multi-thread data in real-time. It was responsible for obtaining and converting sensor information into positioning data, deciding upon the best course of action, and relaying the information to the compensator which relayed commands to the motor controller to follow the user. Aside from the main highest

priority thread, the microcontroller monitored more data from the batteries and motors to ensure proper temperature and current levels, as well as provided the user with helpful feedback on what the LLAGV is doing.

The second microcontroller was the charging controller. It governed all aspects of charging the battery according to the traditional charge profile of a lithium-ion battery pack. The charge controller was the part of the main PDMCSC circuit and was given the means to communicate SoC, warning messages, faults, charging complete status, and charger cable inserted status. The PDMCSC had nets that go to the positive and negative leads of pack, as well as balancing leads that connect to each bank to ensure proper balancing during charge and discharge cycles. Thermocouple probes were also present to monitor pack temperature and create a high priority alert if the pack ever overheats. The charger cable connection incorporated a relay inhibit feature where the motor controller can't receive battery voltage if the charger is connected to the LLAGV.

The PDMOSC contained many active and passive electrical components within the PCB. DCDC converters stepped down the main pack voltage to a 5V rail to power devices that need this voltage as their maximum limit. MOSFETs allowed the PCB to utilize the small signal GPIO outputs from the Pi to drive large current loads, such as the motor controller relay. The PCB will include traditional passive components needed for signal conditioning and power smoothing.

The motor controller of choice was the Sabertooth 2x32A dual motor driver by Dimension Engineering. It provided 32A of continuous current per channel, with 64A burst current. The motor controller accepted a 6-30V range. The controller had many features to streamline the control process of the motor, but only the serial bus was used for sending and receiving commands and data, as well as utilize the protection features built in.

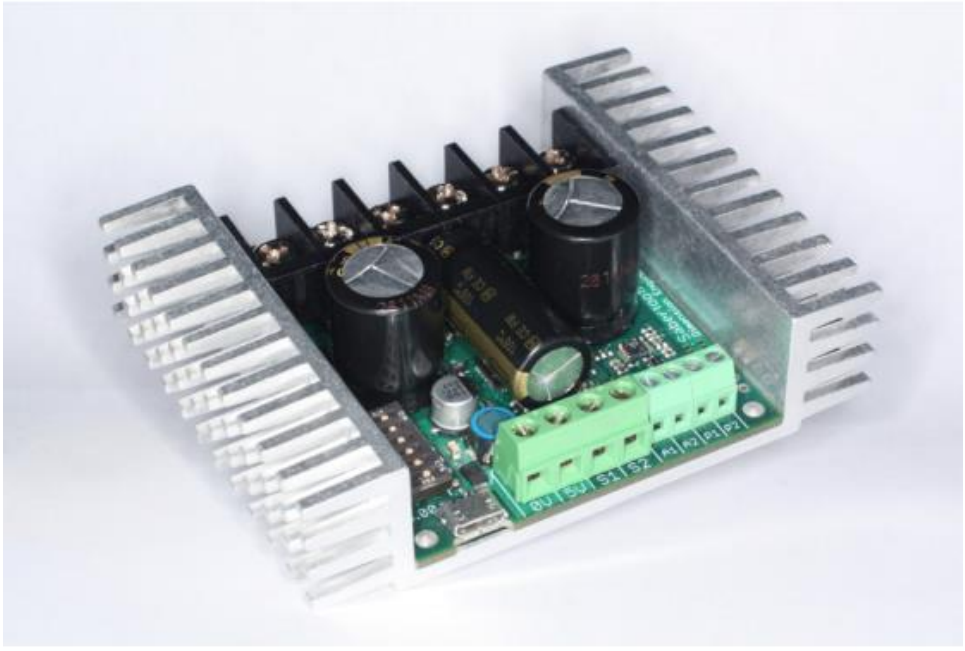


Figure 9 Sabertooth Motor Controller

The sensors of the LLAGV will provide positioning and guidance data for the system microcontroller and will provide the best path for the LLAGV to follow. The information on sensors is explained in more detail in the Navigation division of the project.

2.3. Signal Processing

Team 15B, Navigation Division, will handle Signal Processing.

2.4. Communications

The primary communications network will include the Pi and the motor controller. The communication protocol used will be Universal Asynchronous Receiver/Transmitter (UART) serial. This bus was used because other than sending motor commands, it received telemetry data from the controller, eliminating the needs for external sensors and meters for measurements that the motor controller already made. The communication structure consisted of destination addresses that would allow the Pi to command and receive data from different subsystems of the

motor controller. The commands sent are proprietary and defined in the datasheet for the Sabertooth 2x32 motor controller.

2.5. Electromechanics

In order to understand how the system will behave, use of the robotics design calculations provided was used. These calculations allowed for us to understand how the LLAGV would behave under certain characteristics. The following table will show the results of a fully loaded LLAGV going at the maximum speed with a minimum time to run. Hand calculations can be provided upon request.

Table 2 LLAGV Calculations

Constants:	
gravity	32.2000
static coefficient (max)	0.7000
Normal force	50.0000
Properties of LLGV:	
Weight of LLAGV	20.0000
Weight of Load (max)	30.0000
Weight (total) (lbs, kg)	50.0000
Velocity (max) (mph)	2.0000
Velocity (max) (ft/s, m/s)	2.9333
Mass of LLAGV	1.5528
radius of tire (ft, m)	0.3333
wheel (rad/sec)	8.8000
wheel (rev/sec)	1.4006
Wheel (RPM)	84.03407276
Force (lbs)	35.0000
Torque (ft lbs)	11.6667
Torque per motor (ft lbs)	2.9167
Mechanical Power (ft lbs/s)	102.6669
Electrical Power (W)	139.2536
Pushing Power (ft lbs/s)	102.6669
Torque sliding (ft lbs)	23.3333
wheel rotational velocity, no load (rpm)	17.6000
Total time ON (hour)	2.0000
Total time ON (sec)	7200.0000
Energy (J)	1002626.2787

Voltage (V)	14.4000
Current (A)	9.6704
Capacity of battery (C, Ah)	69626.8249
Km (desired induvial motor constant)	0.301607891
Acceleration from zero speed (ft/s^2)	1.16361E-05
Stall Torque (ft lbs)	23.3333331
No load (RPM)	168.0681455

The LLAGV receives angle of arrival and received signal strength indication data to give it a sense of distance and direction from the user it is trying to find. With the ability to know how far away the LLAGV is from the desired user, the LLAGV must adjust its speed and angle of rotation to ensure the LLAGV is within a proper following distance specified in the marketing requirements, three to ten feet. Of course, the LLAGV will not be designed to constantly vary between three to ten to three feet again however, this range of distance will give the LLAGV "wiggle room" to operate if needed.

As the LLAGV is constantly receiving updates on how far away the user becomes, the most efficient way of controlling how fast the LLAGV wished to go, with respect to a changing distance, would be implementing a compensator. A compensator will be is used to change the performance of our LLAGV system to achieve the desired performance, this performance will be calculated once motor controls, motors, and other specifications have been determined. The compensator will have a few characteristics that will be defined. The first being the Input Command. The primary purpose of the Input Command is that the compensator will be designed to reach the Input Command as its settling point. The Input Command will be determined based upon the distance away from the user. Converting data from distance to speed will be calculated later, upon further research and testing, however, the importance lies upon the application. In theory, the Input Command will be constantly changing depending upon the refresh rate. This means the LLAGV, with proper tuning, will be following the user close, but not too close. In example, if the LLAGV detects the user to be at the furthest distance away, the compensator will convert the distance to a high Input Command, allowing for the LLAGV to reach a max velocity to catch up. As the LLAGV gets closer to the user, the Input Command will decrease in which

the voltage across the motor will start to decrease, therefore slowing the LLAGV down. If the distance drastically decreases, the voltage seen across the motor compared to the input would be negative, therefore causing the motor to "lock up" and in result the LLAGV will break. The next characteristic of the compensator will be the Rise Time. The Rise Time is the amount of time it takes for the system to get from 10% to 90% of the Input Command. The motor will have a natural rise time if voltage is applied to the system however, if one wishes to change the rise time, increase, or decrease, a compensator is need. The Rise Time can also be thought of as the acceleration. The faster the Rise Time, the faster the acceleration. The next characteristic is the Settling Time. The Settling Time is the time required for the output to reach and stay within a given error. Overshoot is the difference of the peak output minus the Input Command. The goal is to have a minimal Overshoot depending upon the Rise Time. The last thing to mention for now is the Transient Response, as it is the system arriving at its destination. Without a compensator, the Transient Response will behave solely upon the characteristics of the physical system: motor, load, damping coefficient, and such. Therefore, a compensator will be used to alter the Transient Response.

In order to understand how the compensator will change the system, the system by itself must be observed. The electromechanical system can be represented within the time and frequency domain. A block diagram representing a single motor with an output of a wheel is given below.

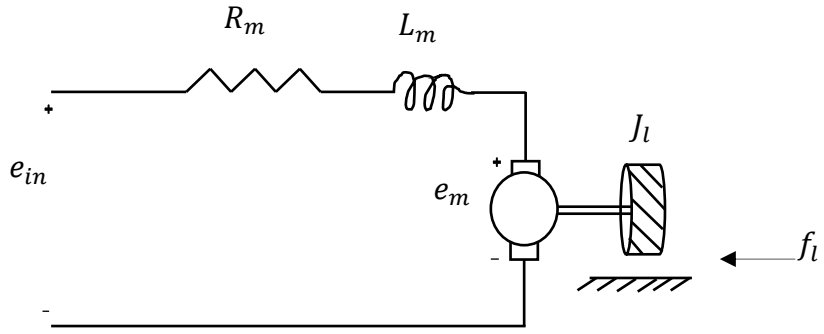


Figure 10 Electromechanical System

Here there a few characteristics that must be defined. e_{in} represents the applied input voltage applied to the motor. i_m represents the current from through the circuit. R_m is the

amateur resistance of the DC motor. L_m is the armature inductance of the motor. e_m is the back emf (electromotive force) from the motor. J_l is the moment of inertia of the wheel. f_l is the rotational damping coefficient. Each of these characteristics play an important role of defining the system. There can now be a set of equations that will represent the components of the system below.

$$e_{in}(t) = R_m * i_m(t) + L_m * \dot{\imath_m}(t) + e_m(t)$$

$$e_m(t) = K_m * \omega(t)$$

$$T_{per\ motor} = K_m * i_m(t)$$

$$T_{per\ motor} = J_l * \dot{\omega}(t) + f_l * \omega(t)$$

In these equations above, K_m represents the stiffness of the motor, $\omega(t)$ represents angular velocity, and $T_{per\ motor}$ represents the torque of a motor. These equations can now be represented within the frequency domain. These equations are restated below.

$$e_{in}(s) = (R_m + sL_m) * i_m(s) + e_m(s)$$

$$e_m(s) = K_m * \omega(s)$$

$$T_{per\ motor} = K_m * i_m(s)$$

$$T_{per\ motor} = (sJ_l + f_l) * \omega(s)$$

$$\theta(t) = \frac{1}{s}\,\omega(s)$$

Understanding the application of these equations allows for us to construct a block diagram for further implementation. The block diagram allows for a visual on the system. The following figure below represents this system.

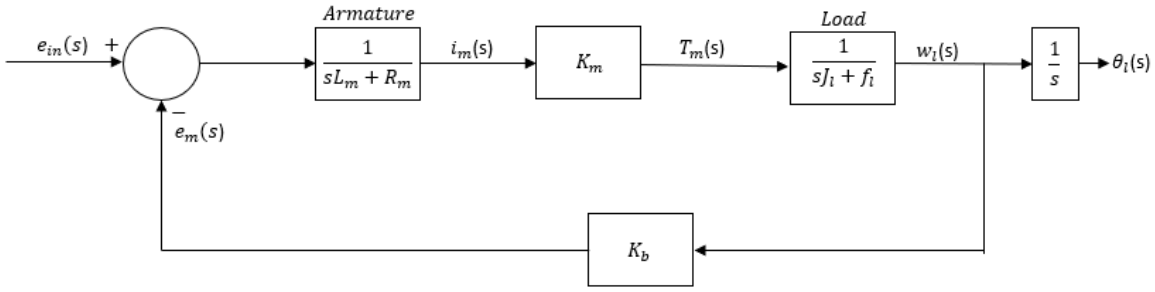


Figure 11 Block Diagram, Detailed

Here, K_b represents the back emf constant. To get a better understand on the value of these constants, motors must be chosen and purchased. All characteristics of the DC motors were not given by the supplier therefore, test must be run on the motors to build the transfer function. It is important to note that external torque is being neglected until real tests can be done on the LLAGV.

In Senior Design 1, tests were run on a cheaper DC motor to be ready to find the characteristics of the actual motor. Doing so provided useful, as a lot of that theory in application was used. It is important to note that I have removed the test results ran on the previous motor and have replaced them with the actual motor used in Senior Design 2.

The motors chosen were the Robotzone model #638276 with dual encoder feedback. This motor was decently priced at $60 and provided the necessary speed and torque for the system. A visual representation of the motor as well as the provided specs are included in the images below.

Figure 13 DC Motor

B. Electrical Characteristics:

1	Max. No-load Current	0.53	A	6	Max. Stall Current	20	A
2	No-load Speed	118±12	rpm	7	Insulation Resist.(500V)	20	MΩ
3	Rated-load Current	2	A	8	Dielectric Strength	250	VAC
4	Rated-load Speed	104±10	rpm	9	Motor Brush Type	Graphite	~
5	Min. Stall Torque	69	kgf-cm	10	Output Power at Max.Eff.	11	W

Figure 12 DC Motor Operating Characteristics

The motor purchased was a 12 Volts, 2 Ampere, and 69 Kg*cm (stall torque) motor. It was also noted that the reducer was a 1:71 gear ratio. The primary reason this motor was also chosen was due to the dual encoder feedback provided. This feedback allows us to convert voltage PWM feedback to RPM. With the recording of RPM, analysis was run to find the characteristics of that motor for equation purposes, as was talked about previously. Once the motor was properly connected and RPM was recorded by the Pi4, more discussion on how the RPM and voltage was recorded is talked about in the embedded systems, analyzing voltage input to the motor and RPM produced by the motor was done next. To properly analyze the motor, MATLAB was used. The data outputted by the motor was set into arrays to be ran. Please note that discussion of MATLAB code and outputs will be done throughout the report. To capture RPM, input voltage was varied by adjusting a dial on a DC supply from 0 to 12 volts, 12 Volts is the maximum rated voltage for the motor. By varying the input voltage, output RPM was

recorded at different steps. Once the data from these were saved, MATLAB ran the data. The following data is presented below in the figure.

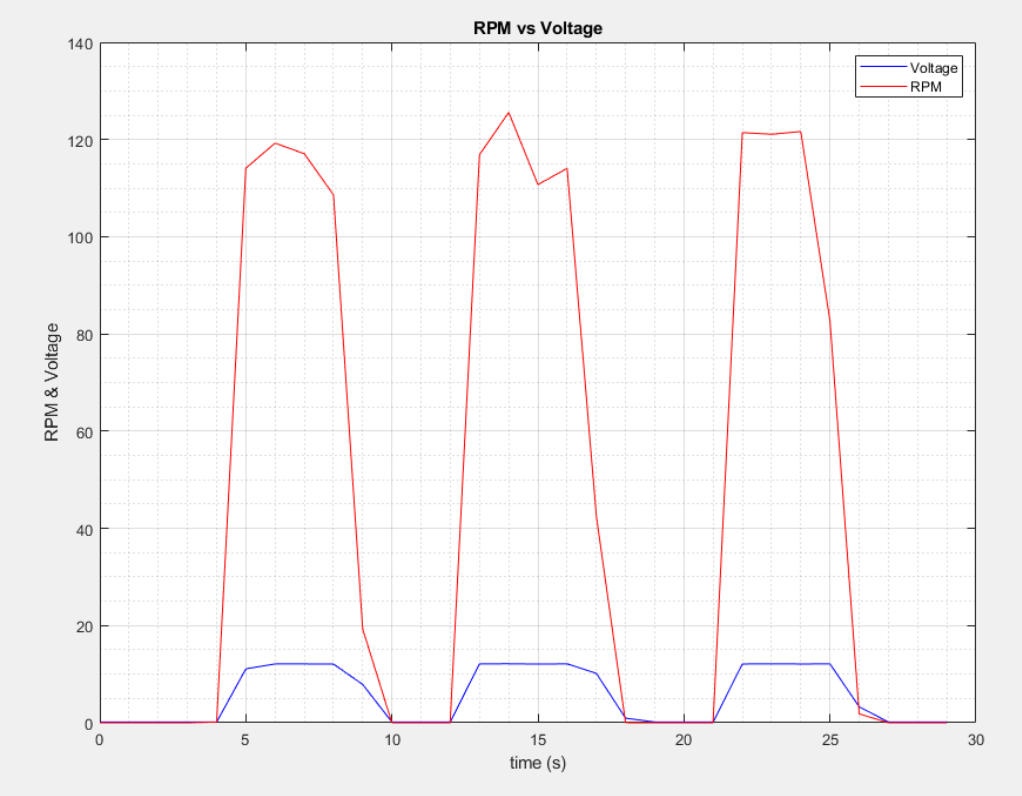


Figure 14 Input Voltage vs RPM

Correlation of RPM to speed was also found. One of our engineering requirements was to ensure a 3 $\frac{ft}{s}$ speed could be kept. The graph below shows the data as such. Note, a load to the system will not change the RPM (speed) as the motors have been chosen to handle the load. The only noticeable difference will be the settling time. Heavier load will result in a slower settling time.

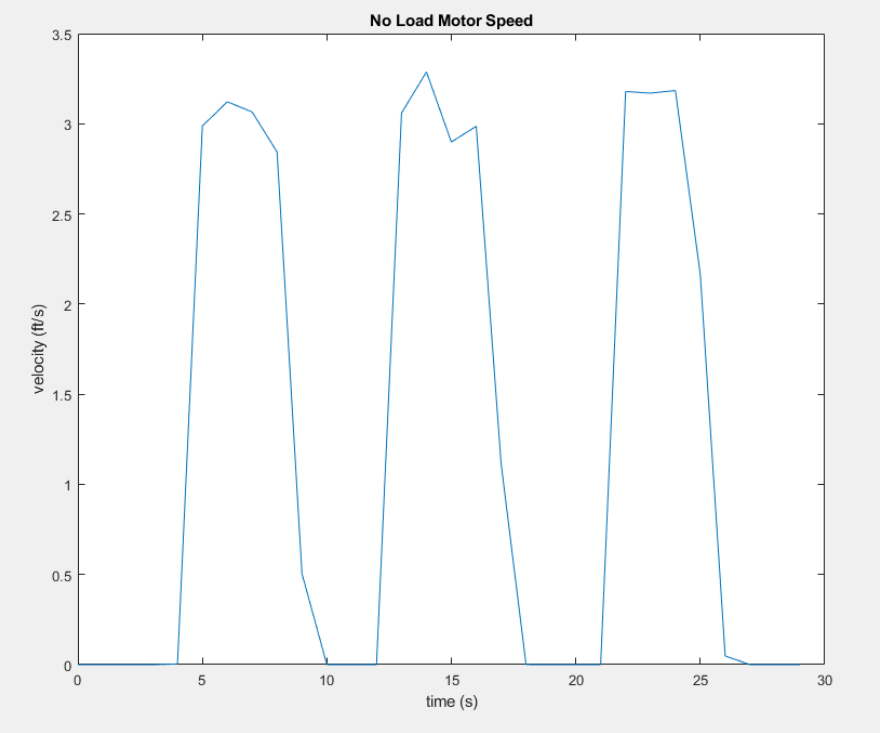


Figure 15 Speed of LLAGV

Here, visual confirmation of 12 volts relating to an RPM of 120 was recorded. After confirming data looked to be correct, the next step was to use this data to calculate the parameters of the DC motor. Once these parameters could be found, constructing the open-loop transfer function could be completed.

To find the parameters of the DC motor, the open loop transfer function relating output RPM to input voltage must be constructed. The constructed figure above was used to form the output equation. Using the following equations above, representation of the output rotation to input voltage is shown.

$$\frac{\omega(s)(sJ_l + f_l)}{K_m} = \left(e_{in}(s) - e_b w_l(s)\right)\left(\frac{1}{sL_m + R_m}\right)$$

$$\frac{\omega(s)(sJ_l + f_l)(sL_m + R_m)}{K_m} = \left(e_{in}(s) - K_b w_l(s)\right)$$

$$\omega(s)\left(\frac{(sJ_l + f_l)(sL_m + R_m)}{K_m} + \frac{K_b}{K_m}\right) = e_{in}(s)$$

$$\frac{\omega(s)}{e_{in}(s)} = \frac{K_m}{(sJ_l + f_l)(sL_m + R_m) + K_m K_b} = \frac{K_m}{s^2 J_l L_m + s(J_l R_m + L_m f_l) + K_m K_b}$$

Therefore, it can be seen by the equation above that the no load motor constants that need to be found are as follows: motor stiffness constant (K_m), moment of inertia (J_l), damping constant (f_l), resistance (R_m), inductance (L_m), and back emf constant (K_b). There are multiple ways to find these values.

In order to find the resistance of the motor, a multimeter was used to measure it. The resistance was measured to be 5.7 ohms. Next, the inductance of an RL circuit was found, also seen as the DC motor. To find the inductance, the following was done. First, a sinusoidal waveform was generated, the output of the waveform generator was connected to the input of the motor. The Analog Discovery Kit 2 was used to measure the voltage across the motor. The overall waveform observed with a DC square wave input was then analyzed to determine the inductance of the motor. The waveform recorded is shown below.

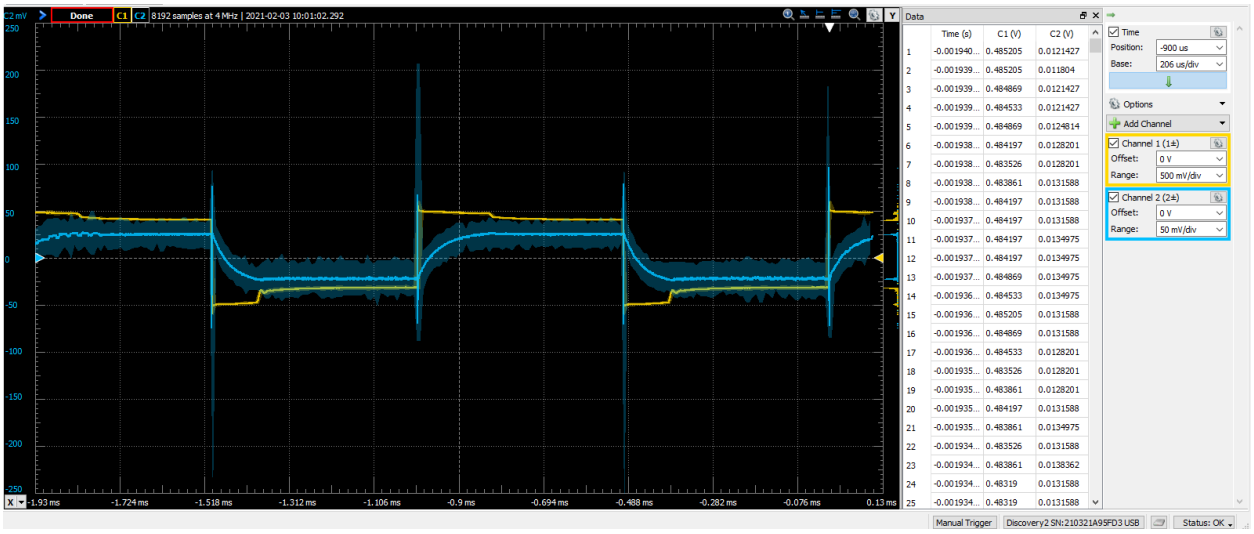


Figure 16 Single Period of DC square wave input

Analysis was then made on the waveform. Here the yellow (top) wave represents a square wave to the motor, the blue (bottom) represents the output of the motor. The amount of time it took for the blue wave to fall from peak to 37% of its peak value was the time constant, τ. The following equation was then used to calculate the inductance of the DC motor. The motors inductance was then measure at 71 mH.

$$\tau = \frac{L}{R}$$

$$L = \tau * \mathrm{R}$$

The next steps of measurements were the motor constant (K_m), and the back emf (K_b). The following equations were used to estimate these values.

$$K_b = \frac{1}{Kv} = \frac{\frac{V,\ peak}{9.543}}{\omega,\ no\ load} = \frac{\frac{12.1366}{9.543}}{125.5804} = 0.92289$$

$$K_m = \frac{K_T}{\sqrt{R}} = \frac{8.8129}{2\pi * Kv * \sqrt{R}} = \frac{8.8129}{2\pi * 1.0836 * \sqrt{5.7}} = 3.6913$$

After finding these constants, the only things left to find was the damping constant (f_l) and the moment of inertia constant (J_l). It is important to note that these constants were found under a no-load condition. When it comes time to implement the system, the weight of the system will play into account for the inertia constant. Implementation of no-load allows for the insight as to how the motor will behave. The following equations were used to find these values.

$$J_l = \frac{T, rated\ torque}{\propto, angular\ acceleration} = \frac{2\ Kg * cm}{14.565\ seconds} = 0.13731 Kg * m^2$$

$$f_l = \frac{T, rated\ torque}{\omega, angular\ velocity} = \frac{2\ Kg * cm}{13.151\ \frac{rad}{second}} = 0.15208\ Kg * m$$

Using these solved constants, a MATLAB script was written to give the open-loop transfer function. The following script below shows the constants and open-loop transfer function.

```
acc = 0.902879953; %time to go from 0 to max RPM
R = 5.7; % Resistance of DC motor
L = R*12.4654E-6; % Inductance of DC motor
torque = 2; % Torque from Data Sheet
```

```
Kv = (rpm_max/9.549296586)/v_max; % motor velocity constant
Kb = 1/Kv; % back EMF constant
Kt = 60/(2*pi*Kv); % torque constant
Km = Kt / sqrt(R); % motor constant

ang_vel = rpm_max/9.549296586; % angular velocity
ang_acc = ang_vel/acc; % angular acceleration
J = torque/ang_acc; % Inertia
B = torque/ang_vel; % Damping Coefficient

% Creating TF from data observed above, note: converted from
rad/s to rpm
H_s = tf([9.5492965964254*12*(Km)/(J*L)],[1 (J*R+L*B)/(J*L)
(Km*Kb)/(J*L)])
stepinfo(H_s)
figure(2)
step(H_s)
```

Here, it can be seen in the above MATLAB script used to generate the step information as well as step plot of the DC motor. Here, input voltage to the DC motor is 12 volts. The output step plot is recorded below.

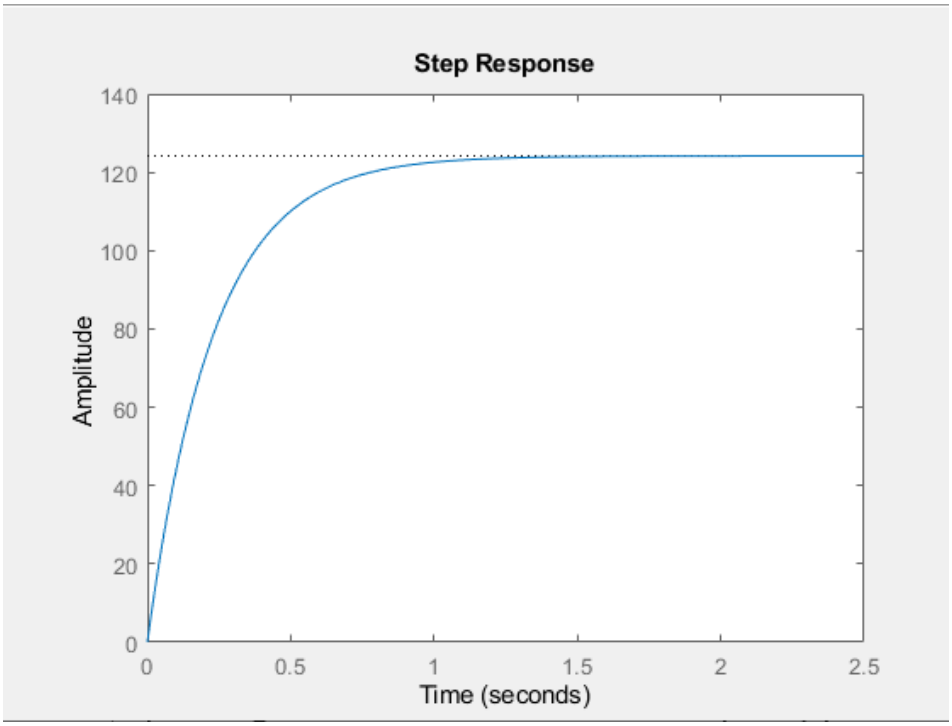


Figure 17 Step Plot showing no load rise time, settling time, RPM rises to values recorded

The following information was recorded and shown below.

```
    RiseTime: 0.5047
SettlingTime: 0.8988
 SettlingMin: 112.3082
 SettlingMax: 124.1634
   Overshoot: 0
  Undershoot: 0
        Peak: 124.1634
    PeakTime: 2.4228
```

Figure 18 Step information shown

Now, the open loop transfer function can be recorded

$$\frac{\omega(s)}{e_{in}(s)} = \frac{1.874E07}{s^2 + 8.022E04s + 1.509E05}$$

Now that the open loop characteristic equation was solved, it was needed to introduce and simulate load conditions to the motors. The reason this was done is so the compensator can be constructed. What if the load introduces a sinusoidal RPM or perhaps it is too slow with settling time; for this reason, we must simulate responses. In order to do so, it is needed to introduce a load to the system. This load will be called J_{tot}. An updated diagram is shown below.

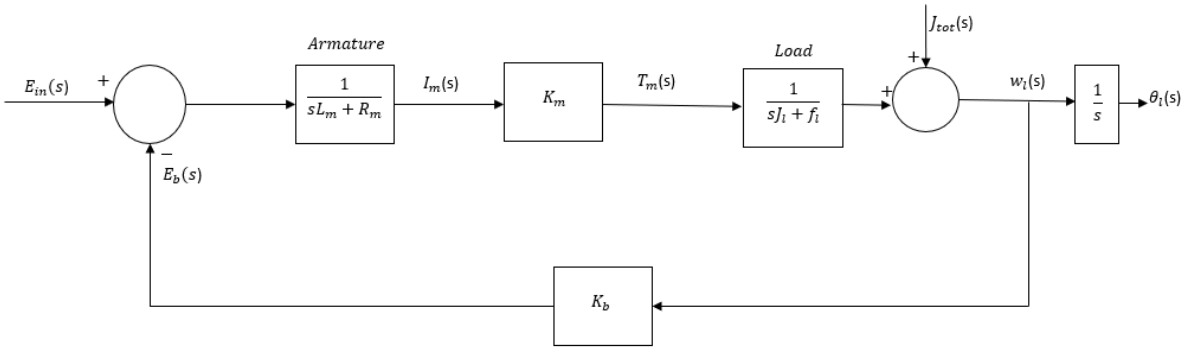


Figure 19 Block Diagram, including load torque

To calculate J_{tot}, it was needed to use some equations in order to solve for the four wheel inertia, J_w and the velocity, inertia, from strictly the load , J_v. These equations are solved below. To follow, w is the mass of one wheel. D, is the diameter of one wheel. l, is the LLAGV and the load in the LLAGV (the total load on all four wheels).

$$J_w = \frac{w * D^2}{2}$$

$$J_v = \frac{l * D^2}{4}$$

$$J_{tot} = J_w + J_w + J$$

Now, a new transfer function can be introduced. The transfer function is recorded below. It is important to note that this transfer function can change all the time. Yes, the LLAGV will have a weight of its own but the load can change from 0 to 30 lbs. For this reason, I will only represent a transfer function with a total load of 70 lbs. Forty pounds for the LLAGV and thirty pounds for the weight. I have also included the MATLAB code below that was used to simulate and find these values.

$$\frac{\omega(s)}{e_{in}(s)} = \frac{\frac{K_m}{L_m J_{tot}}}{s^2 + s\frac{(J_{tot}R_m + L_m f_l)}{L_m J_{tot}} + \frac{K_m K_b}{L_m J_{tot}}} = \frac{1.79907}{s^2 + 8.022E04s + 1.449E05}$$

```
% Calculating inertia of 4 wheel AGV with a load

D = 0.1524; % diameter of wheel, unit in m, 6 in diameter wheel
w1 = 0.793786; % mass of 1 wheel in kg

% Load of AGV in kg
lbs = 70;
w2 = lbs*0.453592;

% 4 wheel inertia
Jw = (1/2)*w1*D^2;

% velocity inertia from load of AGV
Jv = w2*(D/2)^2;

% motor shaft conversion load inertia
Jl = Jw+Jv;

% total inertia
Jtot =  Jl + J;

% TF of a load condition, note: converted from rad/s to rpm
num_load = [12*9.5492965964254*(Km)/(Jtot*L)];
den_load = [1 (Jtot*R+L*B)/(Jtot*L) (Km*Kb)/(Jtot*L)]
H_s_load = tf(num_load,den_load)
stepinfo(H_s_load)
```

Now that the transfer function was found, simulation was done to find the step response and step information. Doing so gave us the following information.

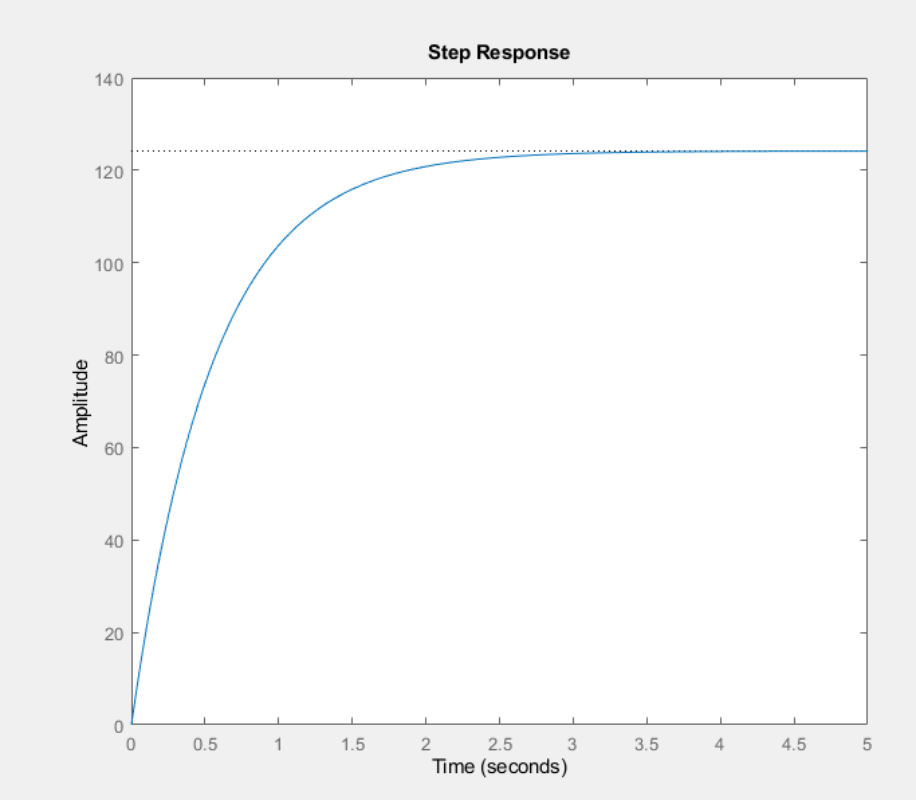


Figure 20 Step Response of a loaded condition

```
    RiseTime: 1.2163
SettlingTime: 2.1659
 SettlingMin: 112.3086
 SettlingMax: 124.1634
   Overshoot: 0
  Undershoot: 0
        Peak: 124.1634
    PeakTime: 5.8386
```

Figure 21 Characteristics of the loaded condition

All MATLAB Code for this section is included below.

```
clc; close all; clear all;

% Time recorded
temp_t = [1612479288    1612479289  1612479290  1612479291
1612479291  1612479292  1612479293  1612479294  1612479295
1612479296  1612479297  1612479298  1612479299  1612479300
1612479300  1612479301  1612479302  1612479303  1612479304
1612479305  1612479306  1612479307  1612479308  1612479309
1612479309  1612479310  1612479311  1612479312  1612479313
1612479314  1612479315  1612479316  1612479317]';

% While loop below scales time from 0 to 29 seconds
i=1;
while i <= length(temp_t)
t(i) = abs(temp_t(i) - temp_t(1));
i = i+1;
end
t = t';

% Input Voltage
v = [0.15   0.142739778 0.15    0.142739778 0.142739778
0.15    11.08389368 12.11484514 12.10758492 12.07854403
7.882135966 0.157260222 0.15    0.15    0.15    12.12936558
12.1366258  12.07854403 12.12210536 10.16910576 0.948624372
0.15    0.142739778 0.142739778 0.142739778 12.10032469
12.12936558 12.10032469 12.11484514 3.293675937 0.15
0.15    0.15]';
% Recorded RPM
rpm = [0    0   0   0   0   0.143229167 114.1102248
119.2555068 117.0820278 108.6230175 19.294812   0   0   0
0.658854167 116.8714913 125.5803642 110.7199869 114.0656442
42.54782097 0.016469173 0   0   0   0.369240052 121.4233971
121.0830765 121.6334722 82.65857881 1.829819965 0   0
0]';

acc = 0.902879953; %time to go from 0 to max RPM
R = 5.7; % Resistance of DC motor
L = R*12.4654E-6; % Inductance of DC motor
torque = 2; % Torque from Data Sheet


figure(1)
plot(t,v,'b') % Plot time vs input voltage
hold on
plot(t,rpm,'r') % Plot Time vs RPM output
hold off
```

```
title('RPM vs Voltage')
xlabel('time (s)')
ylabel('RPM & Voltage')
legend('Voltage','RPM')
grid on
grid minor

% Identify maximum voltage and associated RPM
[v_max,placement] = max(v);
rpm_max = rpm(placement);

v_max
rpm_max
set_voltage = 12;

Kv = (rpm_max/9.549296586)/v_max; % motor velocity constant
Kb = 1/Kv; % back EMF constant
Kt = 60/(2*pi*Kv); % torque constant
Km = Kt / sqrt(R); % motor constant

ang_vel = rpm_max/9.549296586; % angular velocity
ang_acc = ang_vel/acc; % angular acceleration
J = torque/ang_acc; % Inertia
B = torque/ang_vel; % Damping Coefficient

name =
{'Resistance','Inductance','Voltage','RPM','Torque','Kv','K
e','Kt','Km','Angular Velocity','Angular
Acceleration','Inertia (J)', 'Damping Coefficient (B)'}';
value =
[R,L,v_max,rpm_max,torque,Kv,Kb,Kt,Km,ang_vel,ang_acc,J,B]'
;
table = table(name,value)


% Creating TF from data observed above, note: converted
from rad/s to rpm
H_s = tf([9.5492965964254*set_voltage*(Km)/(J*L)],[1
(J*R+L*B)/(J*L) (Km*Kb)/(J*L)])
stepinfo(H_s)
figure(2)
step(H_s)
% figure(3)
% rlocus(H_s)

% Calculating inertia of 4 wheel AGV with a load

D = 0.1524; % diameter of wheel, unit in m, 6 in diameter
wheel
w1 = 0.793786; % mass of 1 wheel in kg

% Load of AGV in kg
lbs = 65;
```

```
w2 = lbs*0.453592;

% 4 wheel inertia
Jw = (1/2)*w1*D^2;

% velocity inertia from load of AGV
Jv = w2*(D/2)^2;

% motor shaft conversion load inertia
Jl = Jw+Jv;

% total inertia
Jtot =  Jl + J;

% TF of a load condition, note: converted from rad/s to rpm
num_load = [set_voltage*9.5492965964254*(Km)/(Jtot*L)];
den_load = [1 (Jtot*R+L*B)/(Jtot*L) (Km*Kb)/(Jtot*L)]
H_s_load = tf(num_load,den_load)
stepinfo(H_s_load)

%P compensator
Kp = 0.45;
R = tf([num_load(1)*Kp],[1 den_load(2) den_load(3)*Kp]);

figure(4)
hold on
step(H_s_load)
step(R)
hold off
legend('No Compensator','P-Compensator')

velocity_load = (3.28084*125*pi*D)/60
velocity_load1 = (3.28084*140*pi*D)/60


% Converting RPM (from no load condition) to velocity
(ft/s)
i=1;
while i <= length(rpm)
velocity(i) = (3.28084*rpm(i)*pi*D)/60;
i = i+1;
end

figure(6)
plot(t,velocity)
title('No Load Motor Speed');
xlabel('time (s)')
ylabel('velocity (ft/s)')
```

Now, let's summarize and analyze what the last few pages of work were about. It is needed to find out how our step response would look when a load is introduced to four motors. To do so, the no-load transfer function was figured out. Once this was found, calculations of the load-transfer function were done to see what exactly changes. For both the no-load and load condition, the overshoot of the motors was 0% so no oscillation was recorded. To further analyze, the settling time at its highest load would be roughly 2.2 seconds. Having such high-rise time and low overshoot was unplanned. In the real world, an engineer might be happy without any compensator as the settling time and overshoot are "good". This is to say for this application I would most likely not use a compensator as it is not needed.

However, for the intention of learning and application a compensator would be used. This compensator would be designed differently from the initial thought. The motors simply must slow down the settling time. Doing so would allow for less power consumption, as the motors would not be accelerating so fast. The choice was made to design multiple compensators for a simple purpose, to delay the settling time to a max of four seconds.

During Senior Design 1, a few compensators were evaluated and a PID was eventually chosen. The reason the PID was chosen was because a PID could speed up the rise time of the system and lower the steady-state error (this could be the overshoot of the system if presented). Without knowing how the transfer function of the DC motor would behave, this was a safe choice. Like stated above, there really wasn't a practical need for a PID in this application therefore, a different compensator should be chosen to best fit the application.

After researching what compensators would be best to use, the choice was made to use either one of the two chosen: feed forward controller or P-compensator. In order to be ready to implement, I designed both. I will now discuss the design process of both controllers.

To design a feed forward controller, the following mathematics was done. To save time, the explanation of how a feed forward controller is not included, just simply the results and MATLAB simulations.

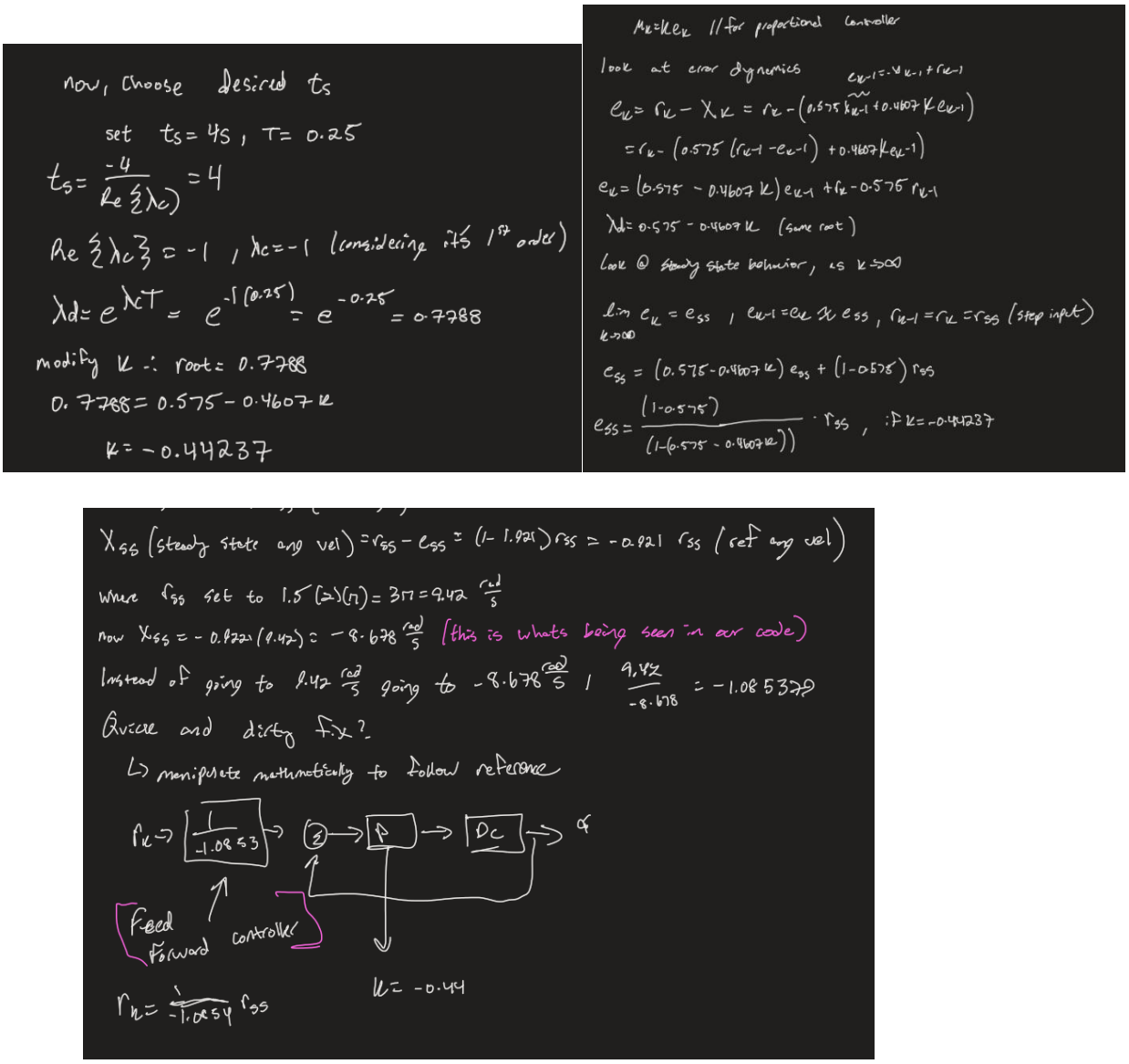


Figure 22 Multiple captures of mathematical work

Now, representing the controller in MATLAB was done. It was first represented within MATLAB such that confirmation of the controller could be made. All the MATLAB code is shown below to implement said controller. This code uses the characteristics of the motor and a desired settling time of four seconds.

```
clc; clear all; close all;

%% Model Current Motor
T = 0.25; %Sampling time

ts = 1.73; %Simulated settling time [s]
re_lamda_c = -4/ts; %Continuous lambda
lamda_d = exp(re_lamda_c*T); %Discrete lambda

if lamda_d <1
    if lamda_d > -1
    A = lamda_d;
    B = ((1-A)*(5.42))/5;
    end
end

%% Simulating Current motor with 4 second delay
tfinal=20;
tvec=[0:T:tfinal];
N=length(tvec);
x_des=zeros(3,N);
f_des=zeros(3,N);
u_in=12*ones(1,N);

ts = 4; %Desired settling time
l_d = exp((4/-ts)*T); %Discrete pole location
k = -(l_d - A)/B; % Initial Gain
ess = (1-A)/(1-(A-B*k)); %Error stead state
xss = 1-ess;
rss = 1.5*2*pi; %desired [rad/s] Input is RPM
xss = xss*rss; %Settling point
ffc = 1/(rss/xss); %Feed forward controller
R = (A-B*k);
U = (ffc*k*B);

%% Plotting
% First calculate what RPM and Voltage is needed to
change u_in. Then run
% it through this filter.

for k=1:N-1

    f_des(1,k+1) = A*f_des(1,k) + B*u_in(k); %
Original
    x_des(1,k+1) = 9.5492965964254*f_des(1,k+1);

    f_des(2,k+1) = R*f_des(2,k) + 1.13*U*u_in(k); %
Delay 4 s
    x_des(2,k+1) = 9.5492965964254*f_des(2,k+1);
```

```
end


figure(1)
plot(tvec, x_des(1,:), 'b')
hold on
plot(tvec, x_des(2,:), 'r')
hold off
legend('Original','Delay of 4 sec')
title('Senior Design Motor')
ylabel('RPM')
xlabel('time')
```

The RPM is sampled at 0.25 seconds to represent the discrete time system of the Raspberry Pi 4 sampling time. Now, the step response is recorded for a 12-volt input.

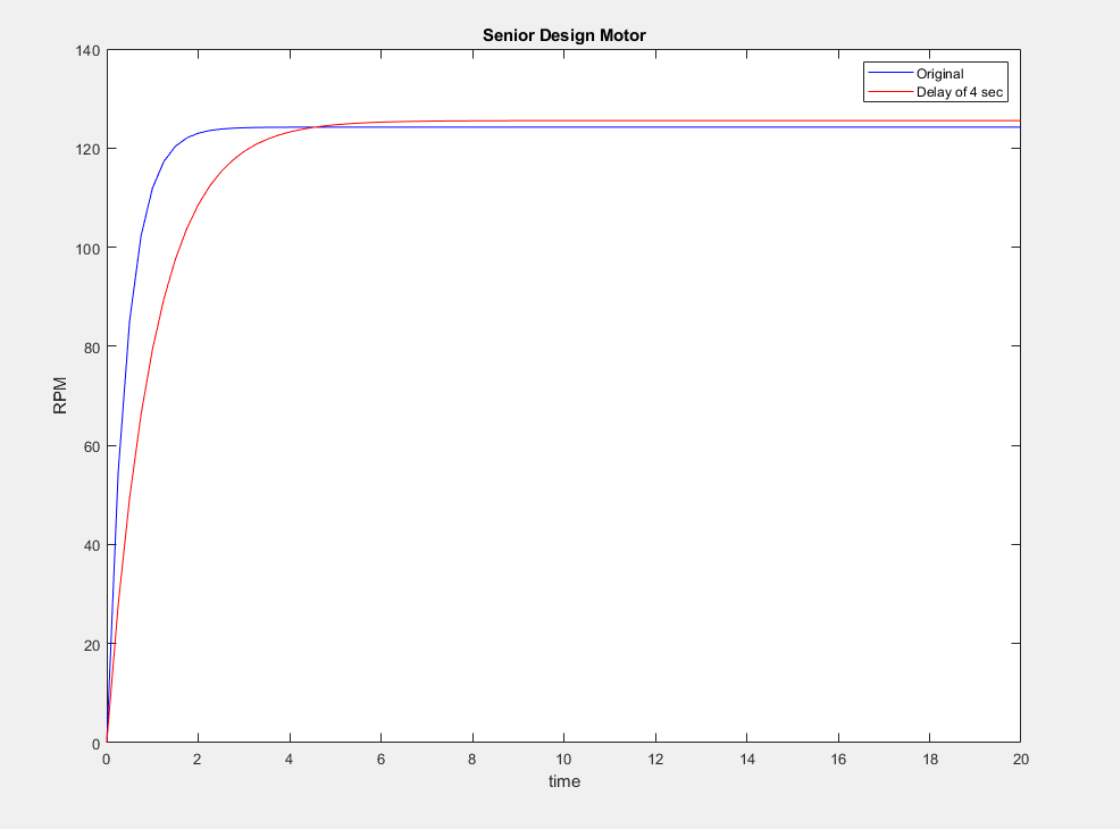


Figure 23 Recorded RPM for feed forward compensator

To summarize this controller, it is easy mathematically to represent. A settling time of 4 seconds is achievable however, the error between the original and delay is small. This compensator can be slightly unstable the more the simulated settling time (depending on the load) changes. Here, it is designed for the max load condition however, if the load changes, the error also changes. The code, however, was ready to be implemented in the microcontroller, and just needed to add RPM feedback and it would be ready to go.

The next compensator to discuss is the P-type compensator. To calculate the: K_p, the use of control system designer tool to find an appropriate pole location. Doing so, an appropriate gain was able to be found. The MATLAB code is shown below, and the step response is also recorded comparing the original to the four second delay.

```
%P compensator
Kp = 0.45;
R = tf([num_load(1)*Kp],[1 den_load(2) den_load(3)*Kp]);
```

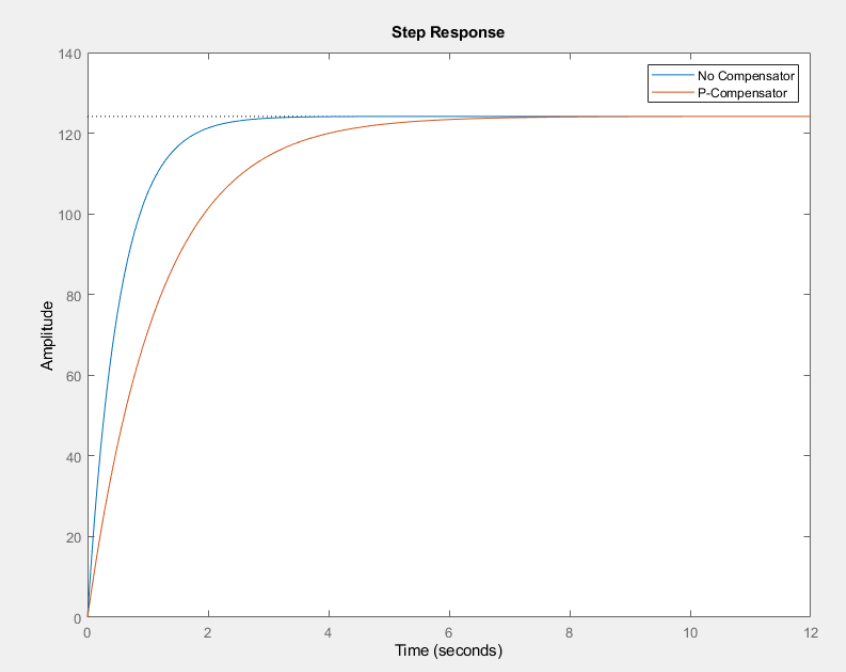


Figure 24 P-type compensator vs original

Here, the compensator is more stable, and the error is very small. This compensator would then be implemented in the system. The Raspberry Pi also had a built-in library which would help configure this compensator.

Once the compensator was chosen and the code was ready to be implemented, the team ran into some bigger issues. When finding that the ultrasonics did not properly work with the Nano-Jetson, the team had to make a decision. With only a week left to implement the team had to decide between object detection and safety versus RPM feedback. Unfortunately, the RPM feedback had to be eliminated and instead, the ultrasonic sensors would take its place on the Raspberry Pi's GPIO pins. This allowed for the project to continue. Even without a compensator, the motors still ran however, without the ultrasonics sensors, the object detection and safety orientation would not work.

2.6. Computer Networks

There is no information at this for the Locomotion division with computer networks.

2.7. Embedded Systems

The LLAGV in general was an embedded system in that effectively combines interfaces the many types of hardware present on the machine. The main system microcontroller was an embedded single board Linux based computer that communicated and interacted with all the different peripherals. Through multiple communication busses, it interfaced with the motor controller through a serial bus, sensors and switches through the GPIO pins and I2C expansion boards.

Each section of the LLAGV specialized in one type of action or responsibility within the machine. The system microcontroller either controlled or obtained data from each section. This means that the embedded system effectively converted any type of data, whether it be voltage, current, speed, distance, direction, temperature, status, etc., to use inside the software code and logic to make computations, make decisions and act upon them, as well as return similar values, or create different types of responses for different systems.

2.8 Mechanical System

The mechanical system of the LLAGV consisted of a cubic shaped main body that formed the superstructure upon which all other components integrated into. The vehicle consisted of four driving wheels, utilizing a tank style control. The left and right side were coupled through identical motor connections, which allowed one channel to control both wheels that drove the respective half of the machine.

The main chassis of the machine was a 18x22x4” form factor, made of 6061 aluminum 1/8” thick. This material was light and strong enough to hold the requirements, while allowing easy drilling for mounting different components. The chassis was grounded. The top cover consisted of metal grating to allow easy access into the chassis area, and also held the load pod. The top cover ideally would’ve been secured using hinges on the rear side of the chassis to allow the top cover to open like a car hood, though the hinges were never incorporated as the top cover needed to be constantly fully removed.

The electrical powertrain mounting consisted of a 12V DC motor, coupled to a gearbox. The 6mm gearbox output shaft was run through a 6 to 8mm flexible coupler that the main axle attached to from the opposite input side. This axle was held up by a pillow block and self-aligning bearing blocks on the walls of the body. Thrust bearings separated these features to prevent horizontal movement of the axle during turns. The 6” wheels bolted onto the axle, about an inch away from the outer wall of the LLAGV.

The overall operation of the mechanical system was satisfactory aside from the couplers. While the couplers were intact, the machine moved efficiently, and the powertrain had minimal load from driveline drag or friction. Adding the marketed load on top of the machine made only a minor increase in the current consumption to carry the load, indicating the mechanical system was not operating beyond any limits.

The single component failure experienced was the coupler from the motor gearbox to axle. These devices were of the spring type, allowing both some angle between conjoining shafts and damping from sharp speed changes. These couplers allowed the driveline to move very

freely. When executing zero turn maneuvers, the radial load on the couplers was beyond limits. This was not because of a heavy load, but of high traction provided by the tires. A zero turn is a precise maneuver that does not move any distance, but spins in place for quick directional changes. This means that all circular movement purely needed to break traction and slip all four wheels to execute, which led to the failure of one coupler. Tesa tape was added to all four tires to reduce their friction with the ground to ease some of the stress off of the driveline components while executing this maneuver. A coupler with more radial strength would have been the perfect alternative for the spring type, allowing more reliable operating during various movements.

3. Engineering Requirements Specification

Marketing Requirement	Engineering Requirements	Justification
6	LLAGV-LS will operate, from a full charge, for a minimum of 2 hours before needing recharged.	Because of frequent stops and often long idle position, a 2-hour run-time will be sufficient for the use case.
1	LLAGV-LS will be able to adjust its angle of rotation based on positioning data polled from navigation team.	The vehicle can adjust its travel angle of direction based on positioning data and navigation around objects.
2,3	LLAGV-LS will have an average speed of 3 feet/second while maintaining an overall distance of 3 to 10 feet.	The vehicle is capable of following the user at a comfortable walking speed, while slow enough to accurately assess its environment.
4,6	LLAGV-LS will incorporate protection circuitry to allow safe power distribution, power consumption, and charging.	Circuitry will be constructed to allow for no over-charge of batteries.
7,9	LLAGV-LS will display system state, state-of-charge (SoC) and faults to the user.	The vehicle needs a way to communicate its metrics and operating conditions to the user.
1,4,7	LLAGV-LS will be able to hold its position when unsafe conditions are sensed. Command vehicle speed 0.	In an emergency situation (user initiated or sensed from navigation team) the vehicle needs to stop ASAP and maintain its position until the user can accommodate the vehicle.
1,2,3	LLAGV-LS will compensate its motion and steering mechanisms by sampling its current speed in real time >1kHz or sampled every 0.001s at a minimum.	In order to accurately control the vehicle speed and direction, the vehicle will monitor its current speed at a high frequency >1kHz.
4,9	LLAGV-LS will respond to manual requests from the user hardware interface <500ms.	For safety and to avoid unnecessary lag the system will respond to state change requests quickly <500ms.
4,7	LLAGV-LS will be < 25lbs, excluding the load, so the user can move the LLAGV if needed.	Should the vehicle enter a situation where it cannot navigate (according to nav team data) the user may need to pick up the vehicle and move it out of the situation.
5,8,9	LLAGV-LS will have an easy access holding cell to transport the light load (up to 30lbs or ~13.6kg)	A hopper design will allow for a light load to be held on top of the LLAGV. Designed to allow for the center of gravity to be as close to the ground as possible.

Marketing Requirements References:

1. The system will be able to navigate throughout the environment in which its placed in order to follow an individual to its destination.
2. The system will maintain a set distance from the user.
3. The system will travel at an average walking speed.
4. The system will have multiple safety features.
5. The system will be able to carry a light load.
6. The system will be rechargeable via rechargeable battery.
7. The system will provide state information as user feedback.
8. The system will travel on light terrain.
9. The system will have an intuitive HMI to operate the device.

4. Engineering Standards Specification

4.1. Safety

Table 3 Safety Standards

Safety Standards	Usage
ANSI B56.5	Warning Alarm/Lights Emergency Stop Collision Avoidance

4.2. Communication

Table 4 Communication Protocols & Usages

Protocol	Usage
USB 2.0	Communication between embedded processor and motor controller
GPIO	Inputs from Switches Output to LEDs
I2C	ADC/Weight Sensor
1-Wire	Signal for Individually Addressable LEDs

4.3. Data Formats

Table 5 Data Formats & Usage

Format	Usage
Binary (En/Decode via Python Struct Pack)	Format for all locally stored data

4.4. Design Methods

Table 6 Design Tools

Language	Usage
AutoCAD	Mechanical Sketch
MATLAB	Motor Control
Microsoft Visual Studio Code	Software Development IDE

4.5. Programming Languages

Table 7 Programming Languages

Language	Usage
Python 3.6+	Application Language
Redis Database	Local Data Storage

4.6. Connector Standards

Table 8 Connector Standards

Connection	Usage
USB	5V Power/Data Transfer
Molex-MX150	PDM power and I/O connections
Generic copper ring terminals	High current connections
Generic shrouded header connectors	Module/sensor connections

5. Accepted Technical Design

5.1. Hardware Design:

The overall hardware architecture was split into many parts. All control system commands, communications, measurements, and switching governed by software or hardware was either interfaced or propagated through one of four circuits: the PDMOSC, PDMCSC, VIH, or PIH. These circuits provided the skeleton for the full vehicle wiring harness and enabled the full embedded system to be interconnected between all destinations. These analysis for these circuits have been covered in section 2.1.

The PDM was the major power distribution and switching module that included the PDMOSC and PDMCSC. The PDMOSC was responsible for distributing most of the power to the different subsystems in the LLAGV. All the incorporated circuits in the PDMOSC were fused. The circuits in the PDMOSC allowed the Pi to send GPIO level outputs that were amplified to control high current peripherals. The PDMCSC was responsible for charging and maintaining the battery pack. It also provided user feedback, charge controller status to the Pi, as well as inhibiting any dangerous activation of the motor controller. The workings of the PDM are explained below.

The circuits to illuminate the neutral and autonomous mode LEDs are shown below in **Error! Reference source not found.**. The LEDs were low side switched with an n-channel MOSFET, using dedicated GPIO ports from the Pi. The return nets were the cathode side of the LEDs, with a current limiting resistor in series. The FET gates were pulled down to enable the channels to work like switches.

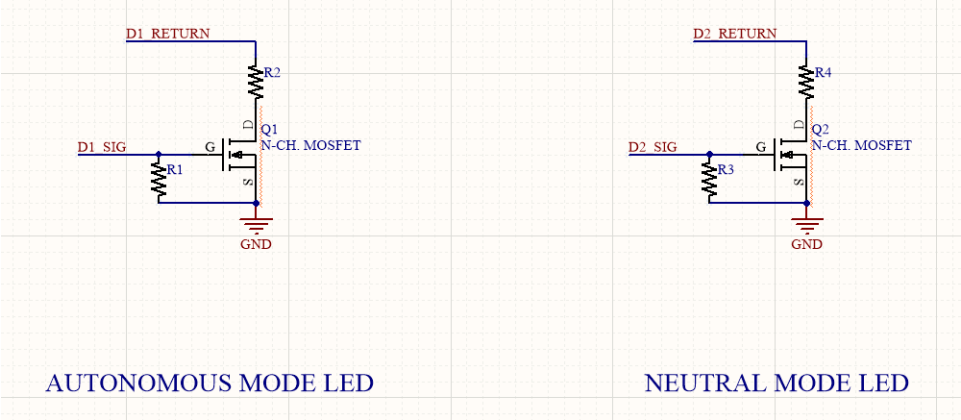


Figure 25 LED actuation circuits

The pre-charging circuits in **Error! Reference source not found.** contained an n-channel and p-channel MOSFET. The pre-charge actuation signal from the Pi named PC_SIG activated Q3, which drove the gate of Q4 low to activate its channel, allowing a path from the fused battery net through F5, a PTC fuse, to a pre-charge resistor that allowed the motor controller voltage rail to slowly rise and build a charge on the capacitors. This circuit were deactivated by turning off PC_SIG, which pulls the gate of Q3 low to break the channel.

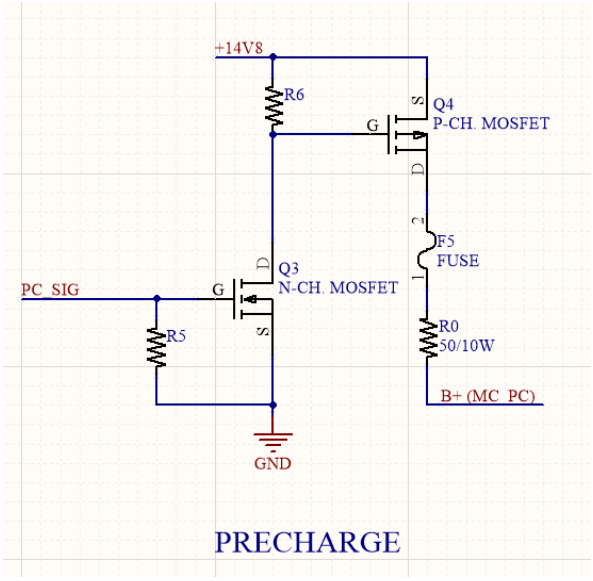


Figure 26 Pre-charge circuit

The main relay switching circuit is shown in Figure 27. This circuit existed to actuate the relay coil (net RLC+) of the high current relay that connected the B+ terminal of the battery to the high side of the motor controller. This circuit had the same principle as the pre-charge circuit,

where the B+ terminal was connected to the common terminal of RL1. This relay stayed in the open state if the charger is not plugged in. This net continued to the normally closed terminal, which is connected to the high side of the main relay coil. When the charger is plugged in, the supply closed the relay, which disconnected the common terminal from the normally closed terminal, therefore opening the main relay.

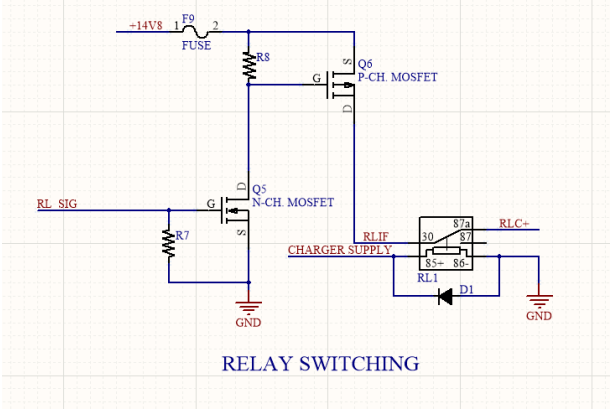


Figure 27: Main relay switching circuit

The PDM included an array of diagnostic LEDs that will indicate whether certain circuits have powered up, such as voltage rails and motor controller's actuation. The setup for these LEDs is shown below.

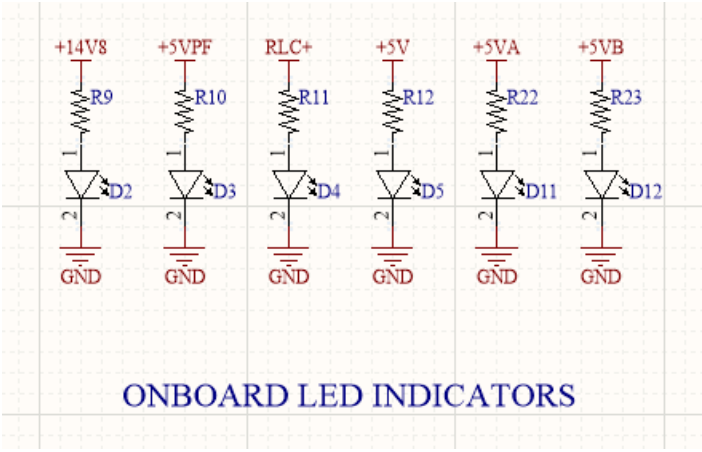


Figure 28: Diagnostic LED indicators

The net current of the system was an available measurement to the Pi. By using the MCP6231 op amp, a differentiator can be built calculate the voltage drop across a current shunt and use the shunt parameters to calculate current going through the device. The current shunt

was a 100mV/100A shunt, with a Kelvin connection at the input and output of the shunt. The input connection was the I_NET FEED net, and output net was I_NET RETURN.

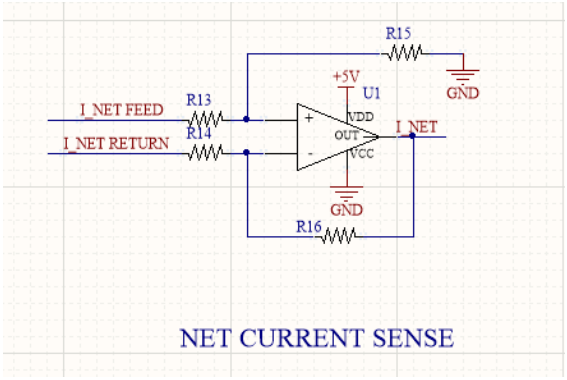


Figure 29: Net current sense circuit

The latching/unlatching circuit used a momentary push button activated by the user to start the LLAGV. The button was connected to B+, and this switched net was S1. By closing RL3, the B+ net spread to whichever ports require it. The net also continued through the normally closed-common connection of RL2 and into the high side coil of RL3, which allowed the relay to essentially feed itself. To break this circuit, RL2 became closed when SHUTDOWN 1 or 2 drive the gate of Q7 high, which introduced a path to ground for the low side of RL2. This disconnected the normally closed contact from the common contact and broke the original B+ net from latching itself, therefore turning of the B+ power to the system and microcontrollers. The system was once again be latched on only if the user pressed the ON/OFF switch.

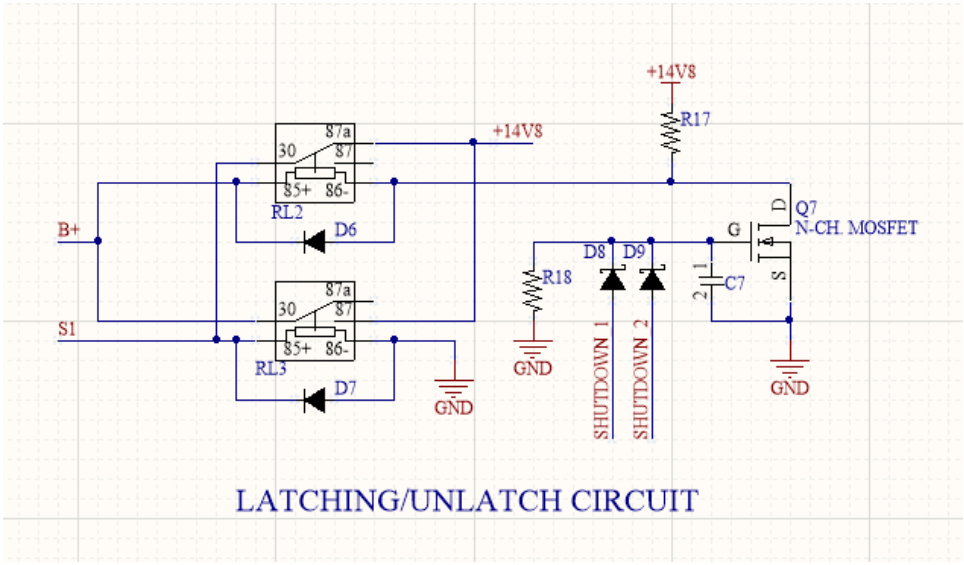


Figure 30 Latching/unlatching circuit

The PDM contained power rail conditioning through two parallel capacitors of 1 and 100uF. This helped smooth out any voltage spikes. The conditioning circuits are shown below.

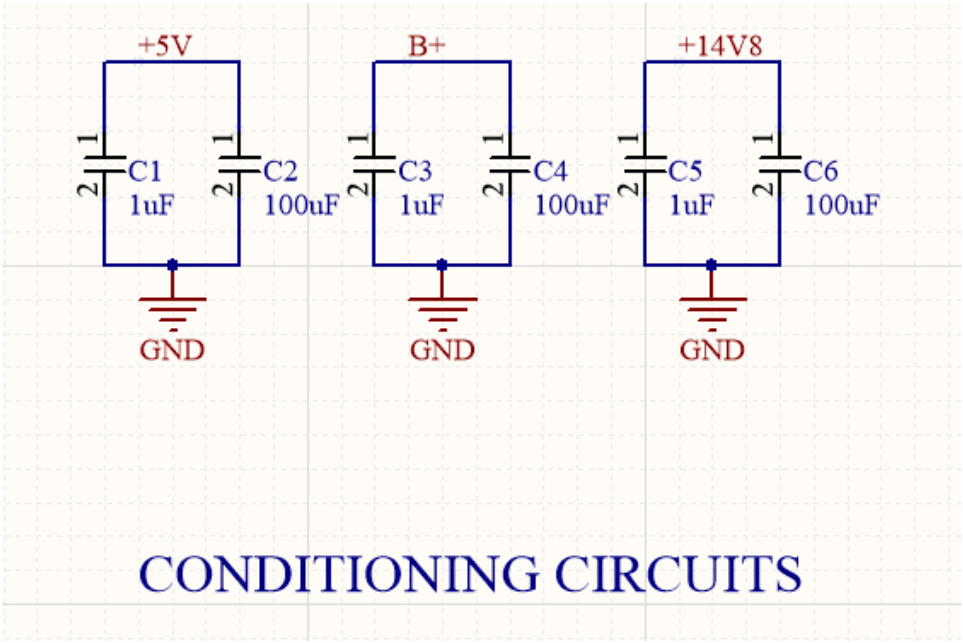


Figure 31 Conditioning circuitry

The PDM contained temperature monitoring and incorporated a cooling fan to control the internal temperature of the main housing. The cooling fan was actuated similarly to the LEDs of

the system, where the low side of the fan was given a path to ground when a channel was made in the n-channel MOSFET. The gate of this transistor was controlled by the FAN_EN net controlled by the Pi.

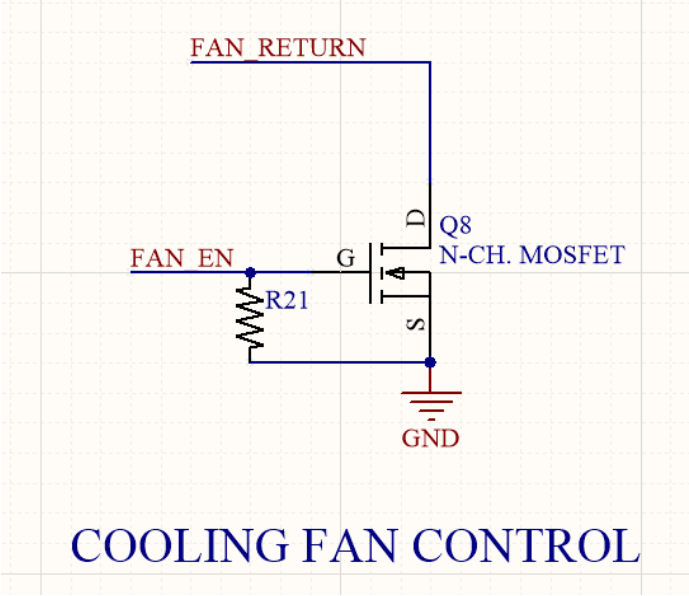


Figure 32 Cooling fan actuation circuit

The charger supply actuation circuit contained a relay that allowed charging to start, as well as sending an inhibiting signal for the system to start driving. This was a safety feature required to never let the AGV not move while charging

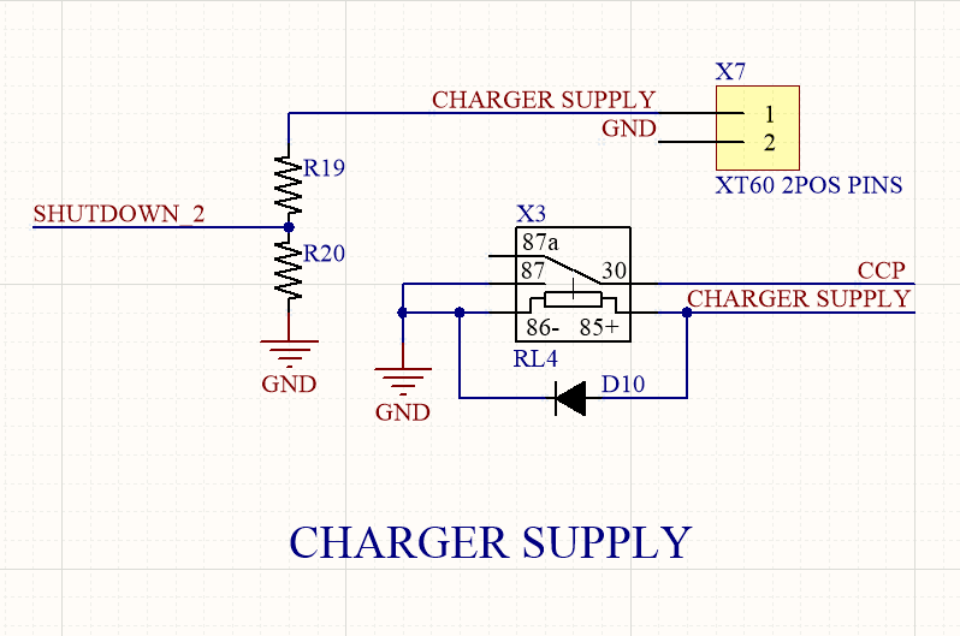


Figure 33 Charger supply circuit

The PDM contained ADS1115 ADCs onboard. U4 contained the resistor networks for the 4 motor temperatures. The first resistor leg of each network was tied to 5V. The center point of each network was routed to its respective analog input on the ADC. The second resistor leg was connected to the center point and ground to complete the circuit. These resistors were routed out from the PDM to the body of the motors; therefore, they are not drawn here.

The individual bank voltage of the 4-cell battery were monitored through the BK nets on U5 and U7. By setting the ADC input range from 0.5-4.5V, the BK nets were the source voltage for their resistor networks. Once again, the midpoint is tied the inputs to the ADCs.

The bank temperatures were set up similarly to the voltage measuring circuits. There max/min values were predetermined to specify the value of resistors required.

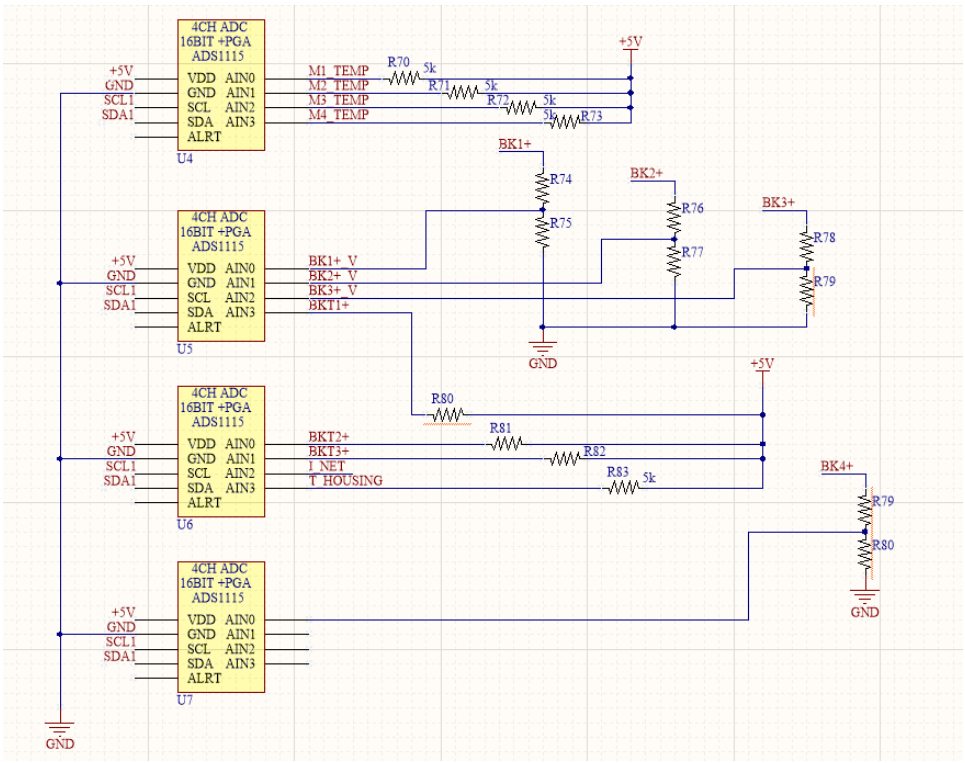


Figure 34 ADC division

The PDMCSC is shown in the figure below. The whole process is governed by the BQ24600 lithium-ion battery charger. The circuit was set up to meet the requirements of the manufacturer, with some calculations needed for calibrating the chip. The battery voltage wasprogrammed through the resistor divider network onto the VFB pin. The current regulation was set to the ISET input of the chip. The setup used two power MOSFETs to charge the battery pack at an 8A rate.

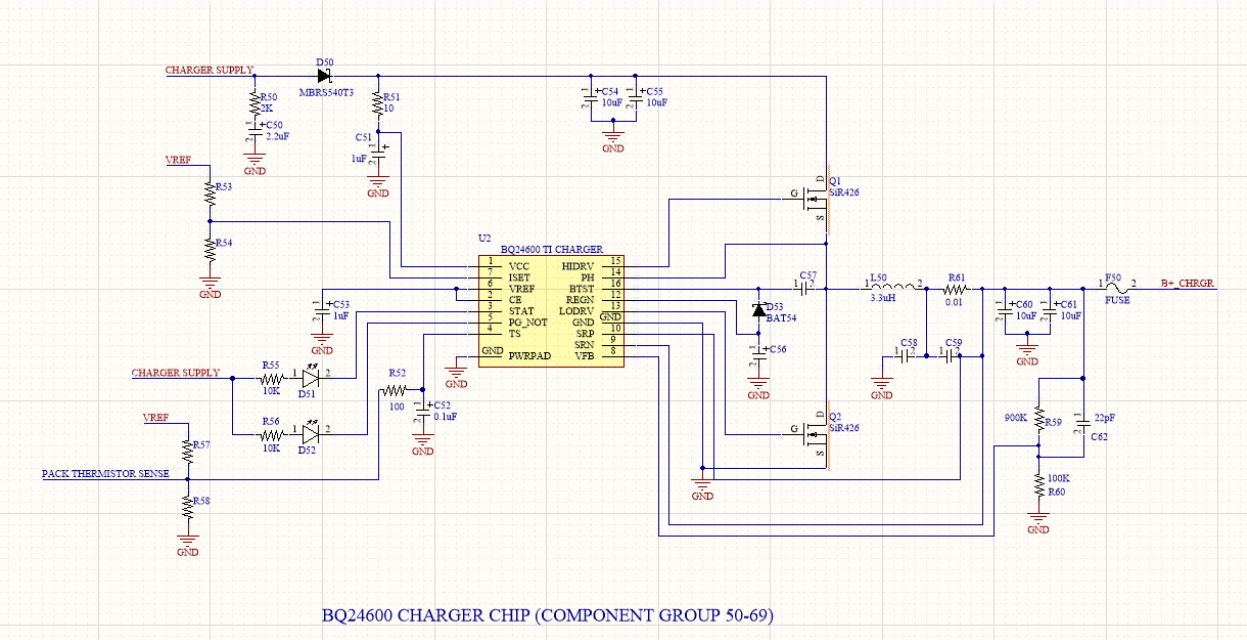


Figure 35 PDMCSC

In the figure below, the system overview of the two main harnesses are shown. The VIH is shown in the top half of the drawing, where it starts at the PDM and splits off for power distribution and signal acquisition for the various sensors, switches, and motor controller. The PIH is shown in the bottom half of the drawing. It also starts at the PDM, and interconnects the motors to the motor controllers, as well as the battery pack and 12-5V DCDC converter. The middle harness shows the connection of PCB to panel mounted charger connection.

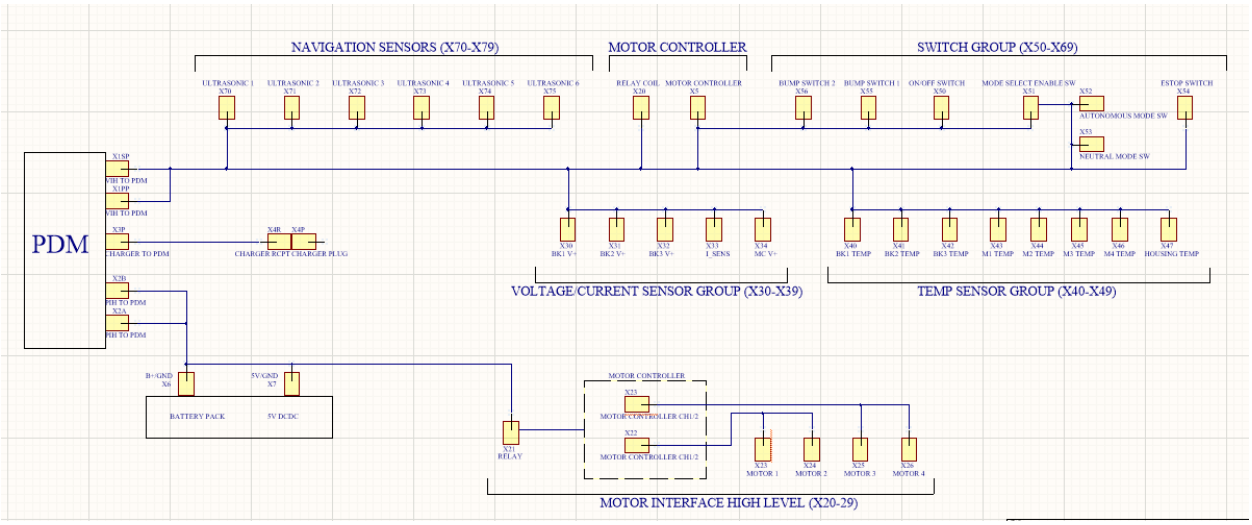


Figure 36: Vehicle Interface Harness

The figure below shows the Vehicle Interface Harness in more detail. At the left, X10 and X11 represent the mating connectors to the PCB. The coil of the RL5 main power relay are supplied by the VIH. B1 shows the battery pack and all the peripherals that will be inside of it. All switches, LEDs, and sensors are shown with their connections to the PCB or power. The VIH will also connect the Jetson Nano to the Raspberry Pi via an ethernet connection. The single large wire is the bus that connected power, ground, and signaling to the 6 ping sensors, shown on the right.

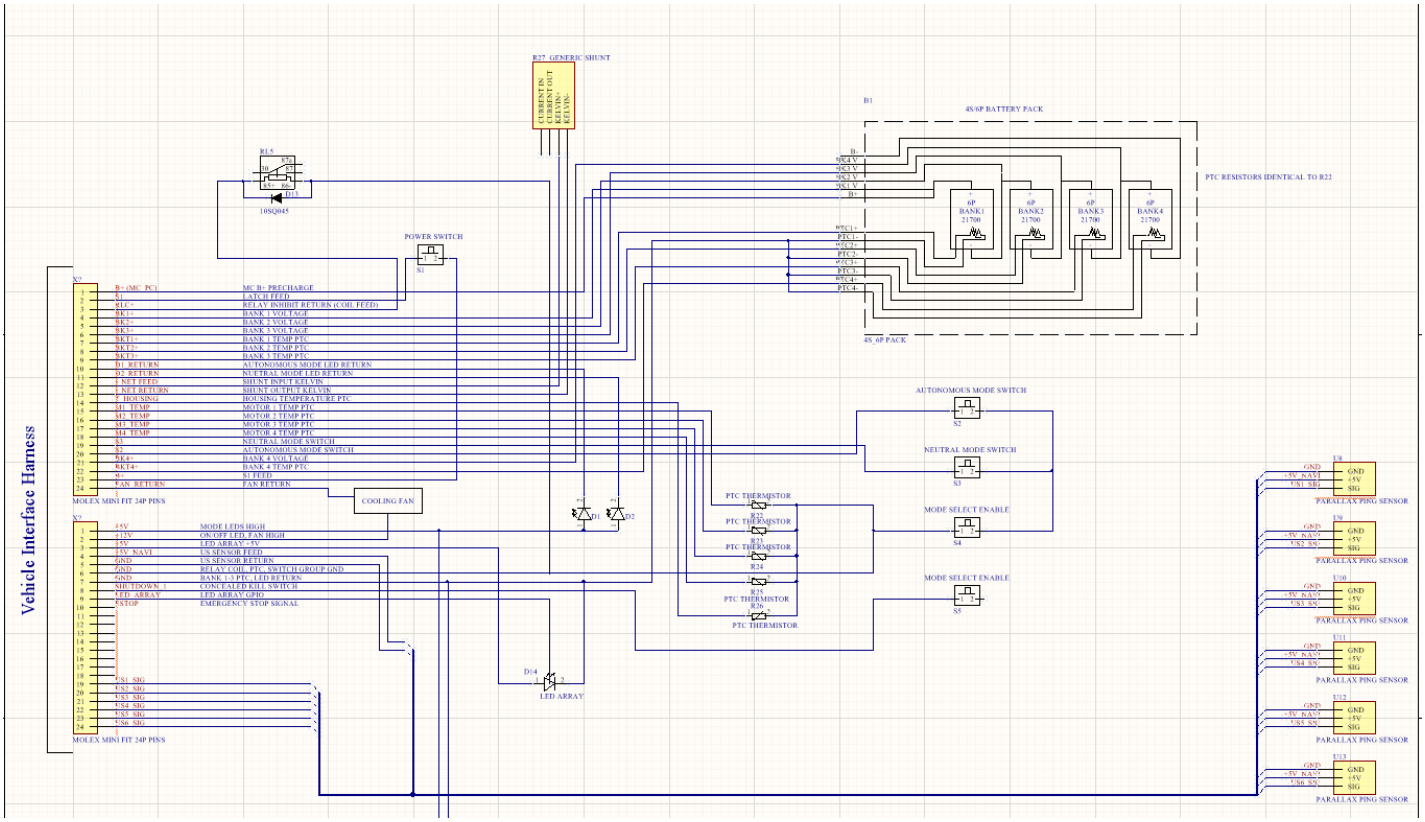


Figure 37 VIH in detail

The PIH shown in Figure 38 shows all high current connections in detail. The B+ was switched by RL5 to the motor controller once it is has been pre-charged. The right-side components show the motor controller and the connections that go from controller to motor. The DCDC converter connection are shown with the 4-pin device on the lower left.

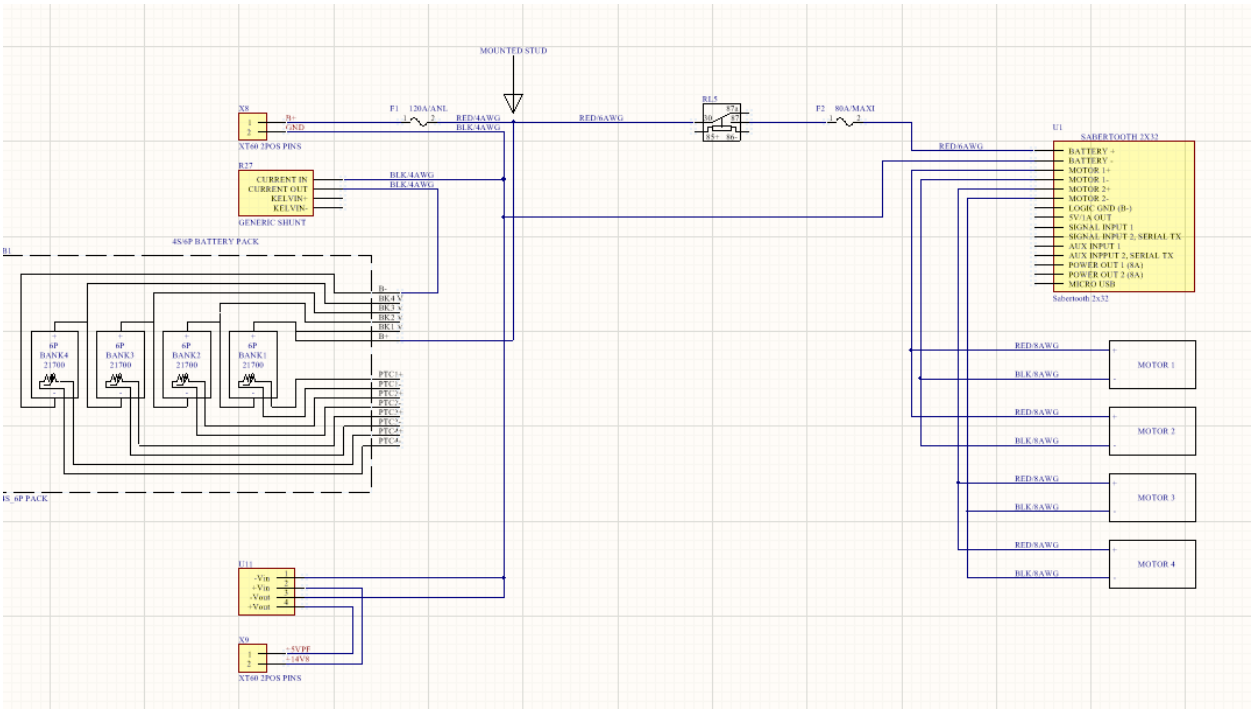


Figure 38 Motor Control to Motors

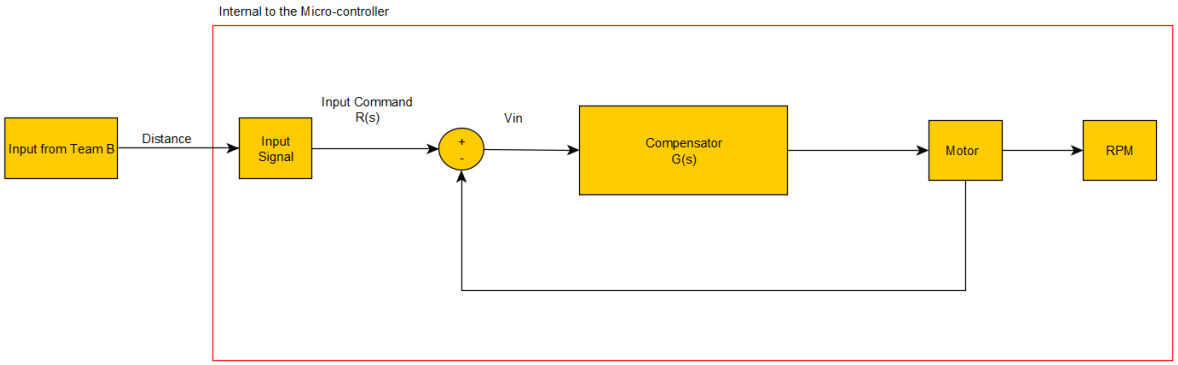


Figure 39: Compensator Level 2 Design

Here, the Level 2 Diagram shows the input signal referring to the Input Command. The P Compensator will allow for the LLAGV speed to be varied upon distance from user. The Transfer Function will be calculated and stored within the program. The output signal will drive to two motor control channels. Data from Team 15B will allow for the LLAGV to detect Line of

Sight, angle of LLAGV to user, and adjust the RPM to allow for necessary changes. Theory of the compensator and theory of design implementation is stated above in engineering analysis.

Table 9 Functional Requirements Hardware Compensator

Module	Compensator Control Loop Level 2
Designer	Lawrence S.
Inputs	Distance from User Motor Control Feedback Transfer Function of Electromechanical System
Outputs	Voltage(s) to Motor Controls
Description	Distance of LLAGV to user, from Team B sensors, will be used as an Input Command to the compensator. Compensator will be designed to allow for the Transient Response, output voltage to the motors, to react in a way that allows for the LLAGV to have proper speed and distance from the user.

5.2. Software Design:

To describe the software architecture for the LLAGV, there exists four active states and one powered off state. The following describes the states found in the figure below representing the system states.

1. Off
2. Startup (State LEDs: Blue)
3. Neutral State (State LEDs: Yellow)
4. Auto/Follow (State LEDs: Green)
5. Help (State LEDs: Red)

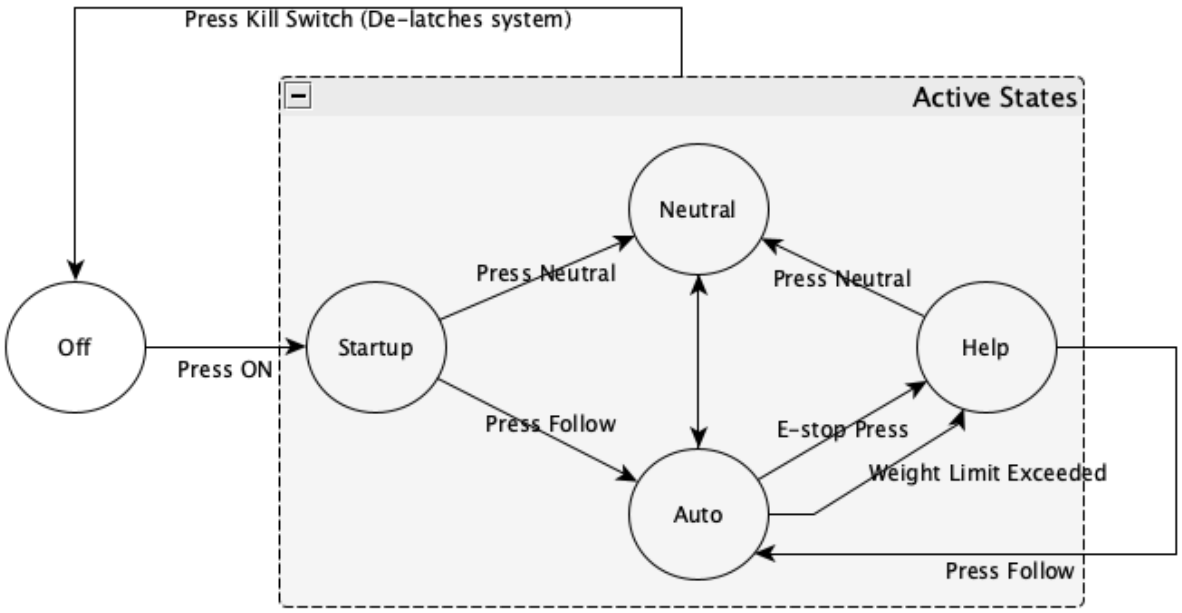


Figure 40 System State Diagram

In the powered-off state, no software systems are active. In this case the embedded processor is turned off, all sensor networks are powered off and the LLAGV can be freely moved as no motor control systems are active. To leave the powered off state, the user must press the power on button held within the push button switch interface.

The startup state exists to prompt the user to select an action. In order to avoid the system booting into follow (Auto) or manual (Neutral) mode, the startup state was created to initialize the database values to safe initial conditions and also prompt the user to select follow or manual control by flashing the dedicated push button switched on the user interface of the LLAGV.

The neutral state was initially created as a state where the user could simply push the LLAGV, by hand, out of a situation deemed non navigable by the LLAGV's sensory system. However, due to the end size and girth of the machine this state now allows the user to drive the LLAGV with a remote control, namely an Xbox 360 controller. This is a feature added to enhance the user experience and will be discussed in more detail later.

The autonomous state implements the main objective, providing a mechanism to deliver a light load for the user. It is here where all systems are active. The sensing network is active to determine a clear path of travel to follow the user; these navigation instructions are made

available to the compensator processes implemented in software to follow the user. Ideally, the vehicle will follow the user as the he/she moves away from the front of the LLAGV while maintaining the set distance. While in the autonomous state, the LLAGV will have the capability to navigate around obstacles and overall achieve the task of reaching the user. In the worst case where the LLAGV gets cornered, or senses objects surrounding the vehicle and cannot determine a clear path of travel to the user; the LLAGV will transition from autonomous to the help state and wait for user intervention to clear the help request.

Lastly, in the discussion of states, there is the help state. This state is entered when the vehicle deems the path of travel non navigable (decision made available by the navigation team) or the machine senses an overweight condition measured via load cell. When the LLAGV determines unsafe conditions as mentioned: overweight, loss of sight, unable to traverse; the LLAGV brakes all motors, holds its position and displays the help state via the dedicated state LEDs on the front of the vehicle (in red). At this point the vehicle is requesting the user's help to proceed. The user can switch to manual control mode to drive it out of the area, remove the obstacles or remove weight from the bed of the LLAGV to correct the situation. In any case, until the situation is resolved attempts made to re-enter Auto (follow) mode will be denied and remain in the help state.

In an attempt to provide overviews pictorially, the overall architecture of both hardware and software will ideally be straightforward. So before jumping into the Python code, several high-level views of the system will be explored.

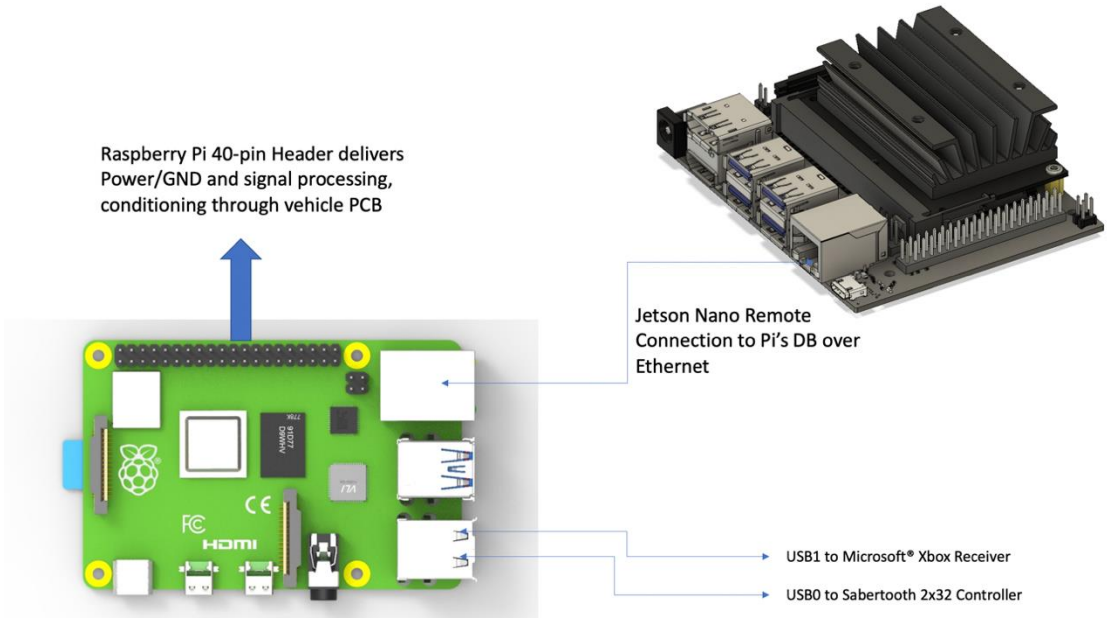


Figure 41 Processor High Level Overview

To help illustrate the hardware connections that will be mentioned repetitively in the software discussion, the above depiction shows a high-level overview of some of the basic connections. The focus of this overview is on the Raspberry Pi 4 (pictured lower left) as this was the processor used on the locomotion system of the project. The Jetson Nano was exclusively used for the navigation "brain" of the LLAGV but is necessary to mention here as for this integration discussion. Depicted in this illustration, there is a few highlights on connections. The entire 40-pin header is taken to the PCB by way of forty position ribbon cable. This connection provides the 5V (up to 3A) power supply and establishes the connections to the vehicle's LED lighting system, push button switches and telemetry circuitry (localized temperatures of motors and battery cell voltage). Moving to the lower center of the illustration, there are two universal serial bus (USB) connections for the Raspberry Pi to communicate with both the Sabertooth motor controller and the Microsoft® Xbox 360 receiver. Communication between Team A and B (Locomotion and Navigation) is one datalink. As depicted, from the Ethernet jack of both the Pi and Nano is a connection made with a standard CAT6 cable. This connection allows the Nano to read and write into the in-memory database hosted on the Pi. This is the only communication method between locomotion and navigation processors used.

The developed application for the LLAGV is implemented by way of a multi-threaded design. Utilizing all four cores of the Raspberry Pi this is made possible. Each thread or process specifically carrying out its respective task will have access to global internal storage via an internal database. The user interface process with the highest priority on the embedded processor encapsulates management of state transitions based upon events detected through push buttons or events detected from the sensor networks.

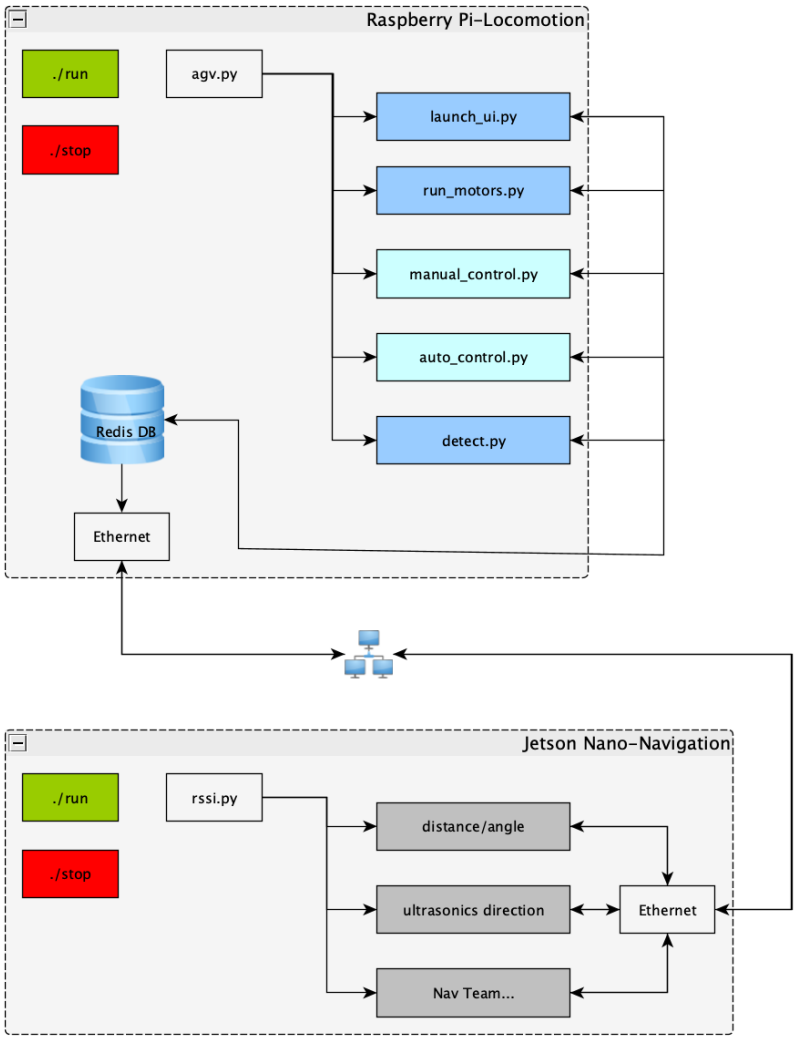


Figure 42 Locomotion Software Overview (Process/Thread Architecture)

This multi-threaded application is outlined above, with focus again on the Raspberry Pi. As a clarification, Python threading and actual threading in the traditional sense are two different things. To explain, in the above illustration the script launch_ui.py behaves as a thread or process

in the traditional sense, occupying one core of the processor. Within that script there are several Python threads kicked off, these do not occupy additional cores, but act as multiprocessing within that overarching process. Ignoring all other scripts mentioned in the above illustration, say just launch_ui.py, if one were to use a program to view active programs/processes it would show as one python process, even when in reality there is a Python thread for each individual switch, LED lighting system and other attributes of the user interface. Perhaps a better word to differentiate is a **process** defines part of the application occupying a core on the processor, whereas a **thread** (*python thread*) lives within a process and there can be many of these per process. This is highlighted by the two different blue hues in the above figure. The darker blue hue signifies a process, whereas the lighter blue hue signifies a major python thread. Noting that there is some detail not shown here, this will be expanded upon later when those individual files are explored and the associated attribute functions.

As one last piece of overview before getting into the specifics of the control processes and threads within, the in-memory database should be covered. Hosted on the Raspberry Pi there is an application running a server that handles read and write requests to a pool of data located in the Pi's on-board random-access memory (RAM). The reason for implementing this in-memory database was to simplify communication among threads and in this case also processors. Normally, all of the control threads within the application would need to be synchronized or how else would the motor control process know when the user switches from manual to follow mode? Implementing this Redis database bypasses the challenge of multiprocessing synchronization. To view the data base values, the sever responds to requests made in the command line interface as well as what the LLAGV application will be using and that is the Redis Python library imported into the Python scripts. The interface is simple, for starters a connection can be made to the database by simply calling the following from the Pi's terminal:

```
redis-cli
```

This establishes a local connection (127.0.0.1) to the Pi's network and permits the viewing, setting and getting of keys within the database. It really is as simple as the following setting and getting of the test key: "SD_test".

```
pi@raspberrypi:~/letsdothis $ redis-cli
127.0.0.1:6379> set SD_test "YAY"
OK
127.0.0.1:6379> get SD_test
"YAY"
127.0.0.1:6379>
```

Figure 43 Redis Command Line Interface Test

Now all keys used in the application are available for manipulation and viewing as well:

```
127.0.0.1:6379> keys *
 1) "right_fts"
 2) "left_percent"
 3) "tsig"
 4) "weight"
 5) "us_2"
 6) "right_rpm"
 7) "soc"
 8) "auto_led"
 9) "neutral"
10) "neutral_led"
11) "turn"
12) "user_assist_req"
13) "auto"
14) "angle"
15) "us_4"
16) "headlight"
17) "right_percent"
18) "test"
19) "us_5"
20) "stop"
21) "too_heavy"
22) "stop_ack"
23) "left_fts"
24) "left_rpm"
25) "state"
26) "us_3"
27) "us_1"
28) "distance"
29) "direction"
30) "system_voltage"
127.0.0.1:6379>
```

Figure 44 Redis Database Keys

Unfortunately, the database keys are not displayed in any reasonable order. This is likely due to how the server handles the storage of the key names and their respective values. Because this is Python, they are likely stored in a dictionary data type which does not have any ordering like you would expect in say a list data type for example. Anyway, the above captures the thirty keys used in the LLAGV software application. This does include a few keys that the Jetson Nano watches and sets. Recall, the Nano is remoting into this hosted database to view data and set some keys pertaining to navigation system information. This is how the locomotion and navigation systems communicate. This command line interface is intuitive and was essential to our integration phase as it allowed quick development and experimentation setting specific keys' values and recording the LLAGV's reaction. As mentioned, the CLI is what was used in software development, but the actual implementation in the LLAGV's application was very similar. For instance, the following shows how to get a value from the database, typecast it to something of value and ultimately set a value in the database.

```
45  def plausibility():
46      state = str(db.get('state').decode('UTF=8'))
47      if state == 'auto':
48          db.set('auto_led', "on")
```

Figure 45 set/get DB Example

Here, a snippet from the plausibility check gets the current state from the key "state", converts from raw binary to a string type. If that decoded string comes back as "auto" (follow mode), then the application will also command the LED ring around the Auto/Follow switch to turn on, acknowledging the state change request. This is one small example, and of course are many more instances of these set/get occurrences.

```
1   import os, redis
2
3   db = redis.Redis(host='localhost', port=6379, db=0)
4
5   # init db
6   db.set('state', 'startup')
7   db.set('auto', 0)
8   db.set('neutral', 0)
9   db.set('left_percent', 0)
10  db.set('right_percent', 0)
11  db.set('distance', 0.0)
12  db.set('angle', 0.0)
13  db.set('state', 'startup')
14  db.set('stop', 0)
15
16  try:
17      os.system('sudo python3 -B /home/pi/letsdothis/ui/launch_ui.py &')
18      os.system('sudo python3 -B /home/pi/letsdothis/motor/manual_control.py &')
19      os.system('sudo python3 -B /home/pi/letsdothis/motor/auto_control.py &')
20      os.system('sudo python3 -B /home/pi/letsdothis/motor/run_motors.py &')
21      os.system('sudo python3 -B /home/pi/letsdothis/object_detection/detect.py &')
22  except KeyboardInterrupt:
23      exit()
24
```

Figure 46 agv.py (overarching start script)

Now for the discussion of the actual scripts and Python code written for the locomotion system. The above captures the entirety of the software triggered to run the locomotion application. The script starts by making necessary imports and creating an object that links to a database connection. The following lines set specific variables to make sure the system boots in a known state. Within the try block is where using Raspbian OS to make system calls and start the processes mentioned in the thread explanation figure in the beginning of this section. Each of these calls are blocking which is why each line ends with an ampersand character telling the OS to run the script in the background. There is a script built to find and kill processes started when the LLAGV is shutdown or if the application is desired to be killed. This is enacted by calling "stop" in the command line. Similarly, this script above can be enacted by calling "run".

User Interface

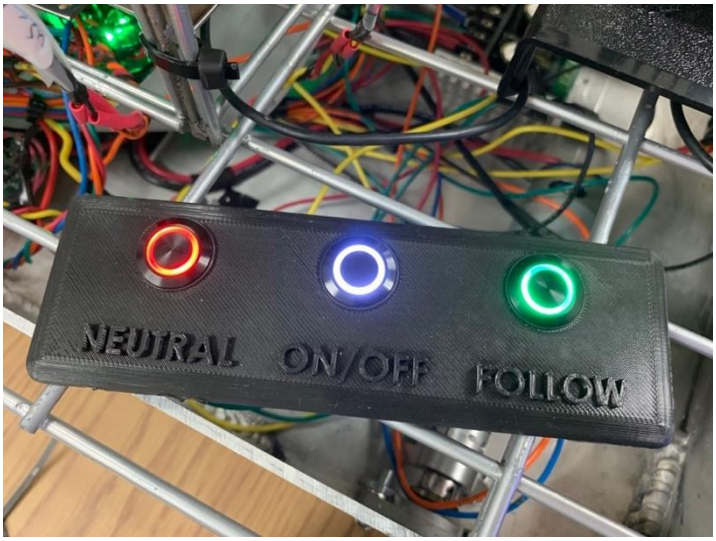


Figure 47 User Interface (Button Interface)

A user interface is needed to gather information from the user. For example, what does the user want to do? A series of push buttons were used for the user to select an operation. These include ON, NEUTRAL, and FOLLOW (Auto). As discussed before, these buttons will force the software to move to different states and if all is well, that is, the respective state. The above photo illustrates the interface used in the final design. The center push button is ON with the system. When the application begins the left and right (Neutral & Follow) push button's LED rings flash at 1Hz prompting the user to make a selection. Once a selection is made the respective LED ring turns solid confirming the user's choice. From there the user can select the other mode and the state will transition. That LED will now turn on solid and the previous states will turn off acknowledging the new selection. The two outer push buttons for neutral and follow also have software behind them.

```python
import RPi.GPIO as GPIO
import board, redis
from time import time, sleep

debounce_ms = 350
debounce = debounce_ms / 1000


class Switch():

    def __init__(self, name, pin, led_pin=None, default_state=None):
        self.switch_name = name
        self.pin = pin
        self.led = led_pin
        self.latched = False
        self.db = redis.Redis(host='localhost', port=6379, db=0)
        self.setup()

    def setup(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
        if self.led is not None:
            GPIO.setup(self.led, GPIO.OUT)

    def cleanup(self):
        GPIO.cleanup()

    def latch(self):
        # invert the latched state
        self.latched = not self.latched
        self.db.set(self.switch_name, int(self.latched))

        if self.latched is True:
            self.db.set(self.switch_name + "_led", "on")
            self.db.set('state', self.switch_name)
        else:
            self.db.set(self.switch_name + "_led", "off")
        return

    def drive_feedback_led(self, state=False):
        GPIO.output(self.led, state)

    def strobe_led(self, interval=0.5):
        self.drive_feedback_led(state=False)
        sleep(interval)
        self.drive_feedback_led(state=True)
        sleep(interval)

    def monitor_switch(self):
        while True:
            action = str(self.db.get(self.switch_name + '_led').decode('UTF=8'))
            if action == 'strobe':
                self.strobe_led()
            elif action == 'off':
                self.drive_feedback_led(False)
            else:
                self.drive_feedback_led(True)
            try:
                if GPIO.input(self.pin) == 0:
                    sleep(debounce)
                    if GPIO.input(self.pin) == 0:
                        print("Button Pressed!")
                        self.latch()
                sleep(0.1)
            except KeyboardInterrupt:
                self.cleanup()
                exit()
```

Figure 48 Switch class implementation

To simplify how the Python code creates the threads monitoring the switch states, a class was defined and within it were functions generic to each switch. The most significant part being the monitor_switch() function which produces the behavior as described. The script will carry out the necessary action for the switch based on the action returned from the database. the following flow chart helps illustrate this feature of driving the LED ring as well as debouncing the switch input and when to update the switch state.

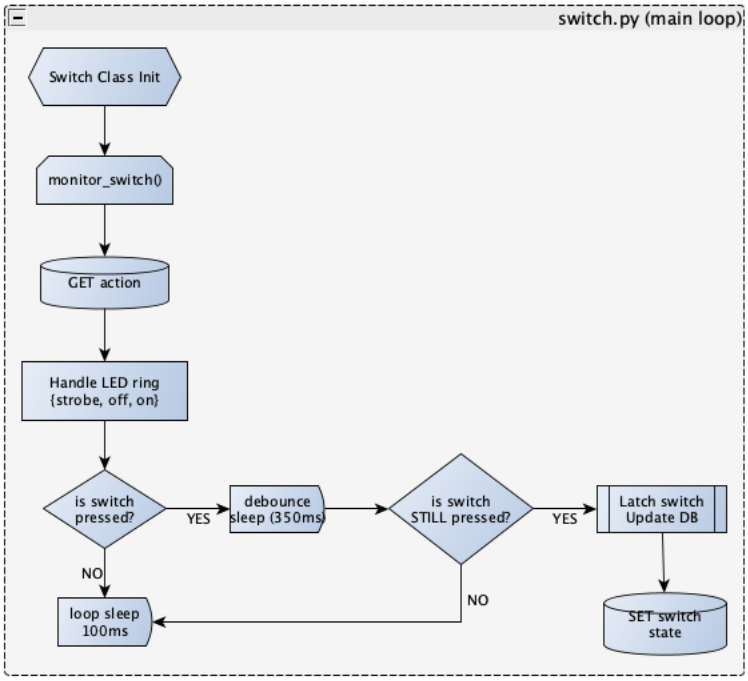


Figure 49 Switch class main flowchart

There are two other buttons apart of the interface. Those are the emergency stop and kill switch buttons (pictured below) on the front and left side of the LLAGV. The emergency stop breaks the connection between the battery input and the controller, disabling the vehicle's movement. The placement of this button was chosen because of how low the vehicle sits and its preference to drive forward. Should something go wrong it is easy to use one's foot to press the e-stop switch in, immediately disabling the movement. The kill switch is smaller and slightly harder to access. This is because the kill switch de-latches the PCB, turning off the entire system: both processors, PCB circuitry and completely turns off the vehicle. These two buttons are hardware only, meaning they have no relevance to software. The e-stop is indirectly detected when pressed due to the serial exceptions raised as communication to the motor controller is lost

at that point. When this happens the state transitions to the help state, requesting the user to cycle power.

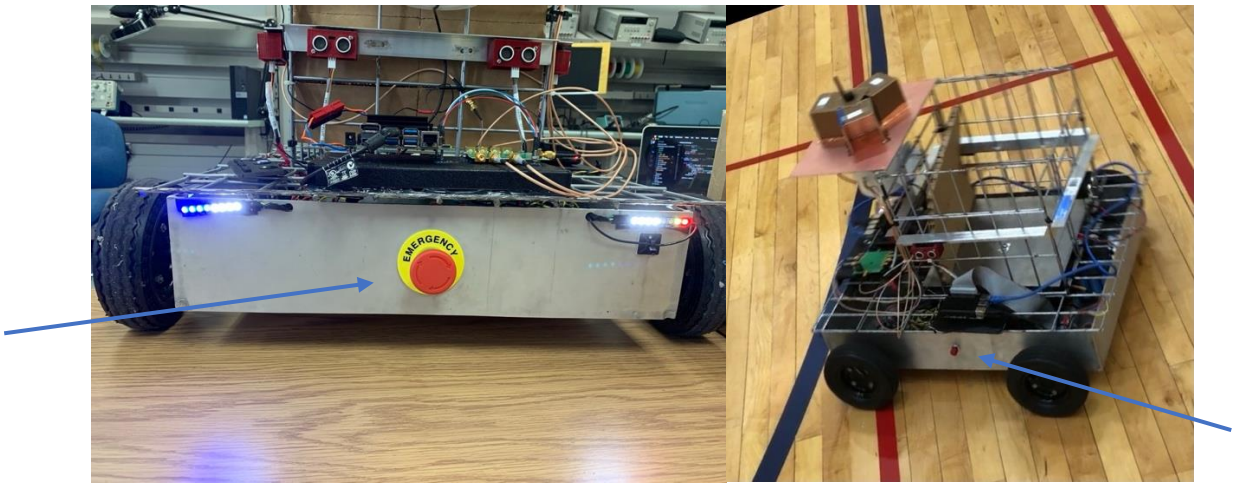


Figure 50 Emergency Stop and Kill Switch

Additionally, a mechanism is needed to display to the user that their request was acknowledged via current state, state-of-charge (SOC) of the device and also headlights and directional indicators were added. An LED lighting system will be used to satisfy this need and perform within the user interface. Individually addressable LEDs will be a sufficient method of displaying this information in the form of light strips. This information will be boldly displayed for the user to quickly verify their requests are acknowledged and monitor the overall state of the system. The photo below is titled with the LED blocks and their associated purpose. The LED strips double as a safety mechanism to alert intelligent bystanders to avoid intentionally obstructing the LLAGV, a warning mechanism if you will.

State/Fault | Headlight/Signal O Headlight/Signal | SOC

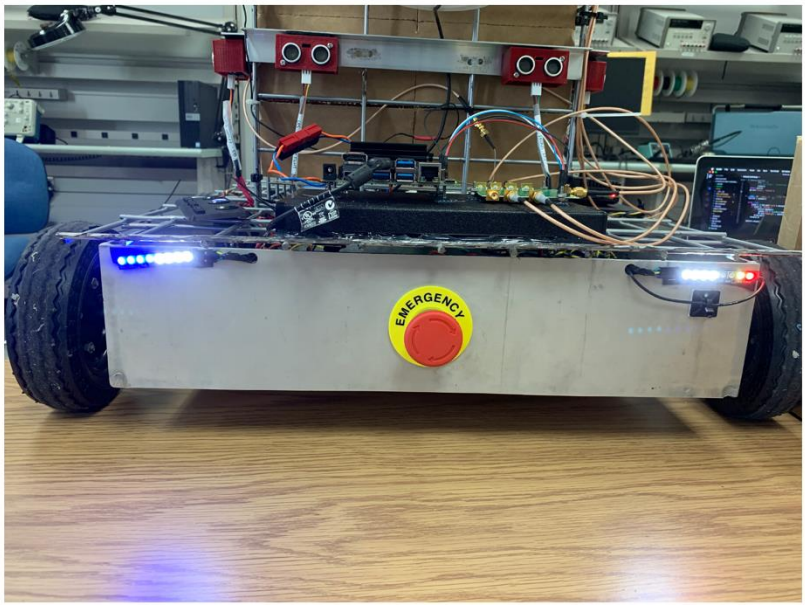


Figure 51 LED lighting system

Per the photo above, from left to right, the left most four LEDs indicate the state of the system, the inner eight LEDs act as headlights and the left/right turn signals. The "O" in the title is just a reference to the big e-stop. The four rightmost LEDs in the photo indicate the system SOC. Note that headlights and directional signals were not part of the initial design but added as they provide feedback as to where the vehicle plans to go and helped to confirm software written without actually moving the unit to keep risk of the LLAGV from ramming into something or someone in the development and integration phases.

Table 10 SOC Functions

Pixel Index	SOC	Color	Brightness
LED[0]	0-24%	Red	Scaled 0-255
LED[1]	25-49%	Yellow	Scaled 0-255
LED[2]	50-74%	Green	Scaled 0-255
LED[3]	75-100%	Green	Scaled 0-255

The breakdown of the LED lighting system starts with one of the most important, the state of charge of the vehicle. To help illustrate this, the table above outlines functions of the rightmost four individually addressable LEDs that will illuminate to quickly and intuitively display the system state of charge to the user. Notice that this is a part of the display user feedback subsystem. In the above functions for displaying SoC there is four, each representing 25% of 100. Initial testing was done on a string of 50 individually addressable RGB LEDs just because it was available at the time. Early testing showed that this was indeed a viable option for displaying SOC to the user. Below simulated percentages were displayed through the display algorithm. For the demonstration below and translated to the final design, a dummy variable to scale the input voltage was used to force an accelerated time lapse of the LED indicators. Now the four LEDs below are all on at full intensity when the unit is fully charged. As the LLAGV is used the SOC depletes. This is shown as the system voltage drops the intensity of the LED scales and dims with the depleting voltage. For example, as voltage decreases from 100% to 75% the left most green LED slowly dims as the voltage decreases and eventually turns completely off once the voltage level drops below the representable SoC sector. Then the next LED begins the same process, until reaching the last sector of 25% -> 0%.

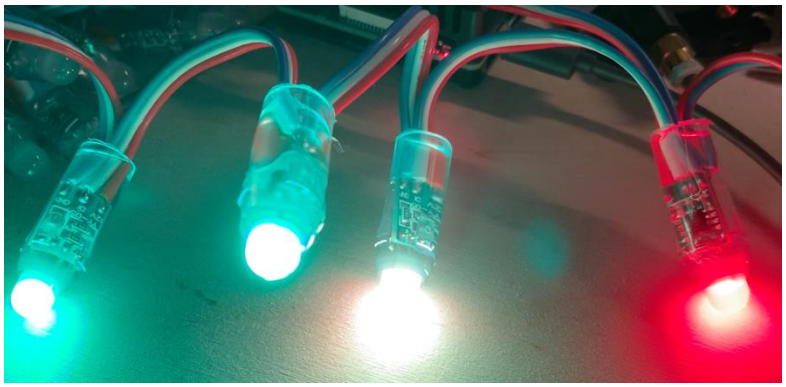


Figure 52 Test RGB SOC Display

This implies that this feedback thread knows what the system voltage is. This information was intended to be measured using an ADC to monitor the cell voltage of each bank. As it turns out the system voltage can be retrieved from the Sabertooth motor controller by a simple ASCII command as follows:

```
34  def post_battery_voltage():
35      volts = 0
36      try:
37          volts = float(int(str(saber.textGet(b'm1:getb'))[:-2].split('B')[1][:3]) / 10)
38      except ValueError:
39          volts = 10
40      if volts < DEPLETED:
41          percent = 0
42
43      percent = round(((volts - 13.2) / 2.0) * 100)
44      db.set('system_voltage', volts)
45      db.set('soc', percent)
```

Figure 53 Query Sabertooth for Battery Voltage and Post to DB

A function was created to collect the voltage reported by the Sabertooth and type cast the string data type to a float. Then based on this value the state of charge could be calculated based on the max and min voltage expected to see on the battery pack. The maximum charged value for the designed battery pack is 16.8V and the minimum voltage one should stop operation and recharge at is 13.2V. A formula was used to calculate the percent value as shown above. This voltage value is made available in the local data storage, that is, the Redis database. The algorithm for displaying the SOC is shown in the snippet below, reiterating what has already been described. There are four sectors where the LEDs behave in a specific way and this is reflected in the following four if/elif blocks.

```
15      def display_soc(self, soc):
16          if 0 < soc <= 25:
17              leds[1] = OFF
18              leds[2] = OFF
19              leds[3] = OFF
20              intensity = soc * 10
21              leds[0] = (intensity, 0, 0, 0)
22          elif 26 <= soc <= 50:
23              leds[0] = RED
24              leds[2] = OFF
25              leds[3] = OFF
26              intensity = (soc - 25) * 10
27              leds[1] = (intensity, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
28          elif 51 <= soc <= 75:
29              leds[0] = RED
30              leds[1] = YELLOW
31              leds[3] = OFF
32              intensity = (soc - 50) * 10
33              leds[2] = (0, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
34          elif 76 <= soc <= 100:
35              leds[0] = RED
36              leds[1] = YELLOW
37              leds[2] = GREEN
38              intensity = (soc - 75) * 10
39              leds[3] = (0, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
40          else:
41              raise AttributeError(f"Impossible SOC: {soc}")
42          leds.show()
```

Figure 54 Display SOC algorithm

Next, the LED lighting system also displays the system state on the far left, pictured in Figure 51. The table below represents the colors of the four LEDs representing system state. Note that all four left most LEDs are acted on in the same way at 100% brightness.

Table 11 State Display

Pixel Index	State	Color	Brightness
LED[12:15]	Startup	Blue	100% (255)
LED[12:15]	Neutral	Yellow	100% (255)
LED[12:15]	Follow	Green	100% (255)
LED[12:15]	Help	Red	100% (255)

This is implemented in Python in the following function. Here, the function takes a string argument pulled from the in-memory database and changes the color of the indexed LEDs accordingly.

```
83      def display_state(self, state):
84          color = OFF
85          if state == 'neutral':
86              color = YELLOW
87          elif state == 'auto':
88              color = GREEN
89          elif state == 'startup':
90              color = BLUE
91          elif state == 'help':
92              color = RED
93          for led in range(n-4, n):
94              leds[led] = color
```

Figure 55 Display State Function

The last feature of the LED lighting system is the additional “bonus” feature of headlights and directional signals to inform the user of the vehicles directional intent. This really came about when sourcing LED strips for the project, as RGBW small LED “sticks” were found to work great in this application. The RGBW has four dedicated diodes within the chipset allowing for a bright white option in place of using all three RGB to simulate a white hue. The white diode allowed for a very bright white and thus came about the feature of headlights. In Follow mode the headlights turn on automatically when the vehicle wants to drive forward to follow the user. In manual mode the user can use the DPAD up (^) key to toggle the headlights is desired. In the process of development, there was worry of unnecessary stressing of the system doing zero turns when testing the angle calculations, as a mechanical motor shaft coupler had already broken. To help with this the introduction of turn signal directional indicators were added such that the intent was shown when the LLAGV should turn right or left. This helped in development as further progress could be made without actually stressing the mechanical system. This feature was left as it also helps the user see the intent of the vehicle. The turn signals are implemented in a sequential pattern slowly indexing the respective four LEDs in an outward motion. The following table explains the pixel index and associated colors. These are the center eight LEDs.

Table 12 Directional Functions (LED Lighting System)

Pixel Index	Position	Color	Brightness
LED[4:7]	Left	White/Orange	100% (255)
LED[8:11]	Right	White/Orange	100% (255)

To summarize the user interface process, here is a flowchart outlining the features.

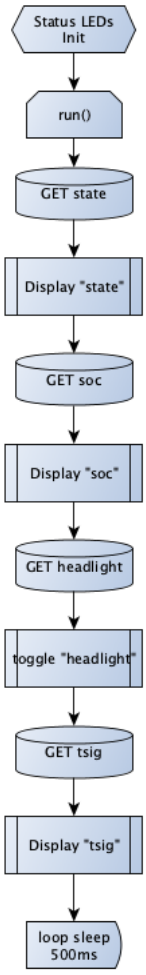


Figure 56 Status LEDs main loop Flowchart

The above flowchart implemented in Python code:

```
105     def run(self):
106         while True:
107             #get current state from redis
108             self.display_state(state=str(self.db.get('state').decode('UTF=8')))
109             #get soc from redis
110             self.display_soc(soc=float(self.db.get('soc')))
111 
112             #headlights?
113             if bool(int(self.db.get('headlight'))):
114                 self.turn_on_headlights()
115             else:
116                 self.turn_off_headlights()
117             #signals?
118             tsig = str(self.db.get('tsig').decode('UTF=8'))
119             if tsig != "off":
120                 self.display_tsignal(directional=tsig)
121             sleep(0.5)
```

Figure 57 Status LEDs Main Loop

The status_leds.py script is too large to be readable in line so it will be included in the appendix section with the software for locomotion as a whole. One of the major components not described in detail here is how the LEDs are commanded. These individually addressable LEDs are commanded through a protocol named 1-wire. GPIO 18 is one of three GPIO on the Raspberry Pi that can communicate 1-wire and is the one used here. This communication is literally over one signal wire. Each LED has an IC and if the packet reaches that individual LED and matches the index, it carries out that action. If the index does not match, the packet of information is not meant for that LED and the information is passed to the next LED and its IC. There are many times in the script that reference a specific color i.e., RED, BLUE, GREEN, etc. These colors are defined in an led_colors.py file like so:

```
1   RED = (255, 0, 0, 0)
2   GREEN = (0, 255, 0, 0)
3   BLUE = (0, 0, 255, 0)
4   WHITE = (0, 0, 0, 255)
5   YELLOW = (255, 150, 0, 0)
6   ORANGE = (255, 128, 0, 0)
7   OFF = (0,0,0,0)
```

Figure 58 LED Colors (RGBW codes)

This just happens to be a cleaner way of defining the RGBW color codes assigned to each LED in the lighting system as a whole.

```
31  def watch_weight():
32      # start a thread to monitor the weight in bed
33      w = Thread(target=post_weight, daemon=True)
34      w.start()
35
36  def watch_switches():
37      # start a thread for all push button switches
38      for switch in switches:
39          sw = Thread(target=switch.monitor_switch, daemon=True)
40          sw.start()
41
42  def begin_feedback():
43      # start a thread to monitor state, soc and headlights/directionals
44      led = Thread(target=leds.run, daemon=True)
45      led.start()
46
47  def plausibility():
48      # to make sure the UI updates properly
49      state = str(db.get('state').decode('UTF=8'))
50      if state == 'auto':
51          db.set('auto_led', "on")
52          db.set('auto', 1)
53          db.set('neutral_led', "off")
54          db.set('neutral', 0)
55      elif state == 'neutral':
56          db.set('auto_led', "off")
57          db.set('auto', 0)
58          db.set('neutral_led', "on")
59      elif state == 'startup':
60          db.set('auto_led', "strobe")
61          db.set('neutral_led', "strobe")
62      if bool(int(db.get('too_heavy'))):
63          db.set('state', 'help')
64
65  def main():
66      watch_weight()
67      watch_switches()
68      begin_feedback()
69      while True:
70          plausibility()
71          sleep(0.1)
```

Figure 59 launch_ui.py Python Script

This concludes the user interface section, noting that a combination of push button switches and two LED sticks for a total of sixteen individually addressable LEDs comprise the UI for the LLAGV. Recall from the beginning of this section, the threading architecture, one of the processes spawned is launch_ui.py. This script is responsible for creating necessary class instances of all push button switches on the LLAGV, starts a thread for the LED lighting feedback mechanism, and lastly which will be discussed later it handles a process that monitors the weight placed in the vehicle and starts threads associated with ultrasonics. This will be discussed in more detail when the detect.py script is mentioned, in short, due to unforeseen issues with the Nano the sampling of ultrasonics had to be moved to the Raspberry Pi.

Motor Control

Next, following the application outline there is a script that spawns the threads associated with left and right motor control, posts localized RPM feedback (deactivated), and posts the battery voltage reported by the Sabertooth motor controller. For simplicity there was a motor class written in Python. Two class instances are created for left and right motor control in the run_motors.py script. Starting with the motor class explanation, reference the following Python code Figure 60. Here the motor instance has several functions and attributes. Referencing from top to bottom, there is first the init() method for the motor class. This accepts the Sabertooth library object to permit motor control, the encoder feedback pins, gear ratio and pulses per revolution unique to the motor used. Following this is two functions handing the setup and teardown of the GPIO used in association. The A and B pins from the encoder feedback need to be configured as inputs to the Raspberry Pi. Should this thread exit for any reason a cleanup method was created to clean up the GPIO (reset to kernel known states) and make sure the motor control is stopped for good measure.

```python
class Motor():

    def __init__(self, name, controller_instance, A_pin, B_pin, volts_feedback_pin=0, gear_ratio=71, ppr=48):
        self.name = name
        self.saber = controller_instance
        self.A = A_pin
        self.B = B_pin
        self.volt_feedback = volts_feedback_pin
        self.gear_ratio = gear_ratio
        self.ppr = ppr
        self.counts_per_min = ppr * gear_ratio
        self.setup_io()

        self.db = redis.Redis(host='localhost', port=6379, db=0)

    def setup_io(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.A, GPIO.IN)
        GPIO.setup(self.B, GPIO.IN)

    def cleanup(self):
        GPIO.cleanup()
        self.saber.stop()

    def get_supply_voltage(self):
        raw = ads.read_adc(0, gain=1)
        return round(((raw * 4.096) / 32767) * 3.636363, 4)

    def get_instantaneous_rpm(self):
        counter = 0
        last_AB = 0b00
        start = time()

        while (time() - start) <= 1:
            A = GPIO.input(self.A)
            B = GPIO.input(self.B)
            current_AB = (A << 1) | B
            position = (last_AB << 2) | current_AB
            counter += outcome[position]
            last_AB = current_AB
        # print(counter)
        try:
            rpm = round(abs(((counter * 60) / self.counts_per_min)), 4)
        except ZeroDivisionError:
            rpm = 0.0
        return rpm

    def get_velocity(self, rpm: float) -> float:
        return round(VELOCITY_CONVERT * rpm, 2) # converts rpm to velocity of motor

    def log_rpm(self):
        list_of_tuples = []
        while True:
            try:
                volts = self.get_supply_voltage()
                time, rpm = self.get_instantaneous_rpm(supply_voltage=volts)
                csv_entry = (time, volts, rpm)
                list_of_tuples.append(csv_entry)
                # self.write_to_csv([time, volts, rpm])
                print(csv_entry)

            except KeyboardInterrupt:
                for item in list_of_tuples:
                    self.write_to_csv(item)
                del list_of_tuples
                GPIO.cleanup()
                exit()

    def post_motor_speeds(self):
        while True:
            rpm = self.get_instantaneous_rpm()
            self.db.set(self.name + '_rpm', rpm)
            self.db.set(self.name + '_fts', self.get_velocity(rpm=rpm))
            sleep(0.001)

    def drive(self, channel=1, percent=0):
        channel = 1 if self.name == 'left' else 2
        while True:
            try:
                percent = float(self.db.get(self.name + '_percent'))
            except TypeError:
                percent = 0

            try:
                self.saber.drive(channel, percent)
            except serial.serialutil.SerialException:
                pass
            sleep(0.1)
```

Figure 60 Motor class Implementation

A few words on the rotations per minute (RPM) feedback. There was a lot of effort and design that went into the design theory and control aspect in anticipation of using some sort of PID or compensator in the locomotion system. In order to implement a compensator loop or even a P type controller for the system, two things at a minimum are required. One being the system level voltage to know exactly what voltage is supplied to the motors as it does affect the RPM calculations. This was carried from the initial testing into the final design as shown above a part of the Motor class get_supply_voltage(). The second being counting and determining position of the quadrature encoders attached to two of the four motors on the LLAGV. Again carried from initial test scripts to the final design, get_instantaneous_rpm() and helper functions get_velocity() and log_rpm(). A new addition to simply post these values to the in-memory database was the post_motor_speeds() function. Two motors have quadrature encoders attached as feedback from the left and right side of the vehicle. Quadrature encoders provide two pulse signals that reveal where the shaft is in four quadrants of the rotational geometry. One source describes quadrature encoders as the following: “A quadrature encoder is an incremental encoder with 2 out-of-phase output channels used in many general automation applications where sensing the direction of movement is required. Each channel provides a specific number of equally spaced pulses per revolution (PPR) and the direction of motion is detected by the phase relationship of one channel leading or trailing the other channel” [1]. With the voltage feedback as well as the encoder feedback for one motor, the process can easily translate to another (different) motor and used in combination to satisfy the need for two motors with the encoder feedback. Note that this is the goal, do some testing with a motor, verify the calculations, then apply the findings to the motors used in the final design. Early testing was completed with a small simple test motor to characterize the motor and paired with its datasheet could gauge the formulations and calculations made to use that data collected and calculate RPM. Procedure wise this early testing had a manually selectable variable power supply to drive the motor (0-24V). this voltage was read in through an ADC breakout board into the embedded processor to sense the voltage provided to the motor. The breakout board chosen does not accept 24V signals, rather 5V maximum; a voltage divider circuit was breadboarded to step the 0-24V to 0-3.3V. Obviously, there is some voltage loss here due to discrepancies with the resistors used and interpretation of

the signal, but for the sake of testing was deemed sufficient as the scale 0-22.5V was observed. See the initial test setup used below.

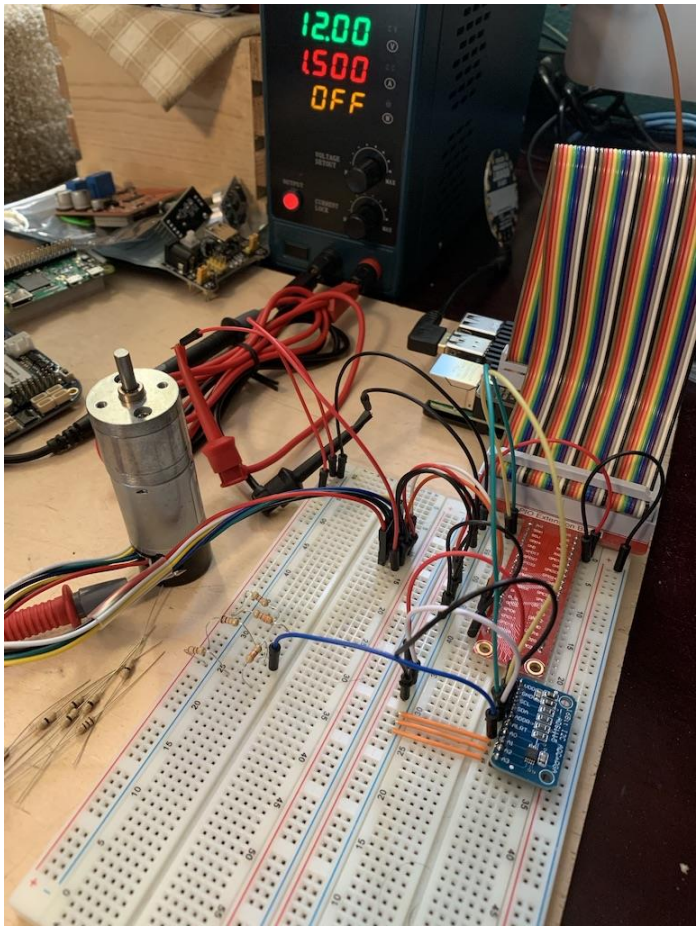


Figure 61 Bench Test Setup Characterize Motor

In the background there is the variable power supply shown. In the foreground the motor to left is socketed into the breadboarded circuit shown. The blue breakout board shown is the ADS1115 16-bit ADC. This ADC device communicates over I2C to the embedded processor. Only one single-ended channel was used for the demonstration, all others were tied to GND, to reduce noise. With some research it was found rotations per minute of a motor shaft can be calculated with the following equation:

$$Output\ RPM\ =\ ((Pulses\ Received\ *\ 60)\ /\ PPR)\ /\ Gear\ Ratio$$

The pulses per revolution and the gear ratio are both constants given in the data sheet for the motor. Again, the overall goal here is to enter the necessary information into the equation, calculate an RPM for a supplied voltage and ideally match what was given in the datasheet to back-up the claims made that this algorithm does indeed calculate instantaneous RPM.

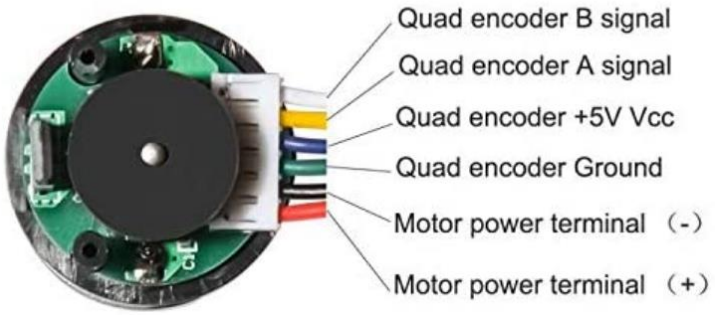


Figure 62 Motor Pinout

There is an A and a B signal connected to the embedded processor (assume grounds and all other supplies where necessary), there needs to be a correlation of the two signals to determine where the motor shaft is relative to permanent magnets are, which is what drives the pulses observed from an encoder. The following table provides a look-up table of sorts. In the algorithm the previous value of A and B are stored. The new (current) A and B signal is sampled and then shifted onto the previous yielding a new combination of values. This value is then used to determine a position, which is the index in the result matrix. This is shown exactly in the algorithm shown below that determines what quadrant the encoder is reading, based on the last and current quadrant reading, it can determine whether the motor is turning clock or counterclockwise and based on the count of pulses sampled in a set interval of time, the RPM can be calculated using the aforementioned equation for output RPM.

Table 13 Quadrature Encoder Signal Lookup Table

Previous A	Previous B	Current A	Current B	Result
0	0	0	0	0
0	0	0	1	1
0	0	1	0	-1
0	0	1	1	0
0	1	0	0	-1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	-1
1	1	0	0	0
1	1	0	1	-1
1	1	1	0	1
1	1	1	1	0

```
A_pin = 5
B_pin = 6
ads = ADS.ADS1115(i2c)
gear_reduction_ratio = 1 / 75
ppr = 12 # @ 24V

GPIO.setmode(GPIO.BCM)
GPIO.setup(A_pin, GPIO.IN)
GPIO.setup(B_pin, GPIO.IN)

outcome = [0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0]


def get_instantaneous_rpm(supply_voltage):
    counter = 0
    last_AB = 0b00
    start = time()

    while (time() - start) < .25:
        A = GPIO.input(A_pin)
        B = GPIO.input(B_pin)
        current_AB = (A << 1) | B
        position = (last_AB << 2) | current_AB
        counter += outcome[position]
        last_AB = current_AB

    try:
        rpm = abs((counter * 60) / ppr * gear_reduction_ratio * (supply_voltage / 24))
    except ZeroDivisionError:
        rpm = 0.0
    return time(), rpm
```

Figure 63 Python Motor Characterization Script

As already explained, the above code snippet demonstrates how this RPM calculation was done using both the supply voltage to the motor (read back in through the ADC) and using

two GPIO pins to read the A and B signals of the encoder connected to the motor. Ultimately, after operations performed on the previous and current AB signals, a timestamp and the calculated RPM are returned. Using this function in combination with the following functions pictured below, these values can be recorded, written to file and imported to be post-processed to determine a model for the motor as well as a transfer function to be implemented in conjunction with a P controller which is the ultimate goal.

```
def get_supply_voltage():
    chan = AnalogIn(ads, ADS.P0)
    return chan.voltage * 7.27272727273

def average_sample_range(start=0, stop=12, increment=1):
    voltage = start
    samples = (stop - start) / increment

    while start <= voltage <= stop and samples != 0:
        volts = get_supply_voltage()
        time, rpm = get_instantaneous_rpm(supply_voltage=volts)
        write_to_csv([time, volts, rpm])
        print(time, volts, rpm)
        samples -= 1
        sleep(0.25)

def log_rpm():
    while True:
        volts = get_supply_voltage()
        time, rpm = get_instantaneous_rpm(supply_voltage=volts)
        write_to_csv([time, volts, rpm])

def write_to_csv(data_list=[], append=True):
    operation = 'w' if append is False else 'a'
    with open('test.csv', operation, newline='') as file:
        rpm_record = csv.writer(file, delimiter=',', quoting=csv.QUOTE_MINIMAL)
        rpm_record.writerow(data_list)


if __name__ == "__main__":

    header = ['Time', 'PS Voltage', 'Calc RPM']
    write_to_csv(data_list=header, append=False)
    input("Ready? Press [ENTER] when ready!")
    log_rpm()
```

Figure 64 Python Script to Log Motor Characterization Parameters

Now the entirety of the above script is not all included as a part of the final motor control loop, however it is used to characterize the motors specified for the final design, as a determination of RPM versus time and voltage will be necessary. With the above software the instantaneous RPM can be logged to a file to later be graphed. All of this initial work was

translated and applied to the motors used in the final design. Updated bench test setup is pictured in Figure 66.

Now recall the Motor class code snippet. The initial testing, calculations and formulations were translated to the final design. The get_supply_voltage() was used heavily in both the initial bench test setup (described above) and the updated one with both final test motors (pictured below). This voltage function simply reads an ADC channel to gather the voltage value sent to the motor, which is then used in the RPM calculation. It turns out, like the battery supply voltage value, the Sabertooth provides this information with an ASCII command so the ADC reading is not necessary. The next function, get_instantaneous_rpm() polls those A and B signals and counts the pulses witnessed from the encoders attached to the two rear motors. The RPM feedback process is illustrated in the following flowchart.

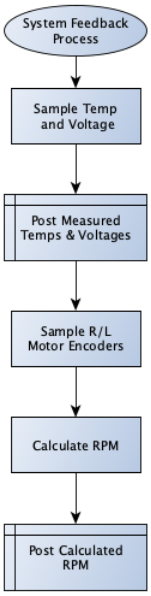


Figure 65 RPM feedback loop

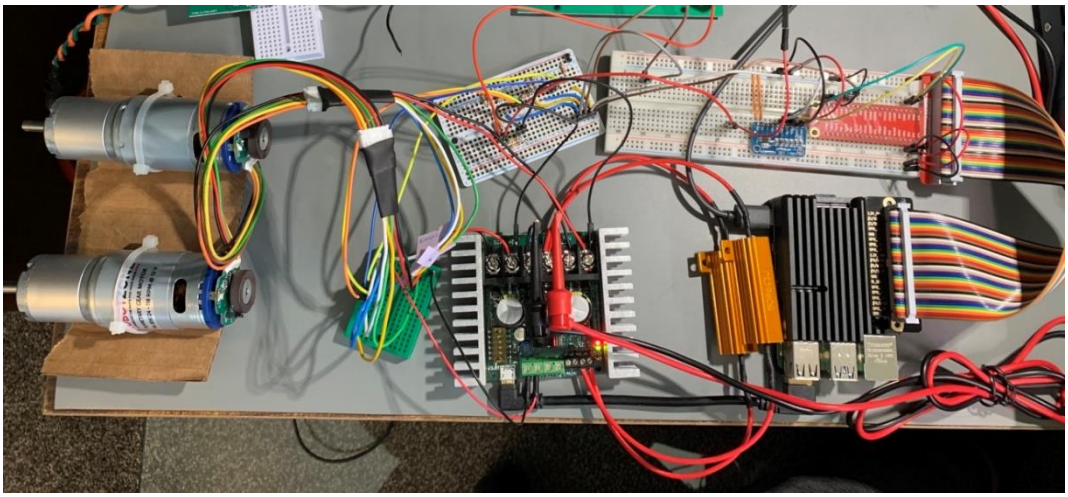


Figure 66 Bench Test Setup with Final Design Motors

This RPM feedback function is fully developed and working however, this monitoring for the RPM feedback was cut. This was due to unforeseen issues with signal processing on the Jetson Nano and the ultrasonic processing had to be moved to the Raspberry Pi. Because the cores and processing on the Pi was already dedicated and maxed out with other features, it was decided to cut the RPM feedback. Discussion on what this was replaced by will come later. The following three functions all have to do with logging and posting the recorded RPM feedback and interpreting that as a velocity for the overall vehicle, which was discussed above on how to use those functions to characterize the test and final motors used.

The last function outlined in the Motor class of the final design captured below has to do with driving the respective side of the vehicle. This is really where the use of the in-memory database shines. These motor class instances are constantly running and the loop within the drive() function is always watching a percent field in the database. These threads do not care whether this command is from manual or follow mode control. This database permits complete independence of other threads. The fields can even be updated manually by connecting to the command line interface as shown earlier in the software discussion.

```
94      def drive(self, channel=1, percent=0):
95          channel = 1 if self.name == 'left' else 2
96          while True:
97              try:
98                  percent = float(self.db.get(self.name + '_percent'))
99              except TypeError:
100                 percent = 0
101
102             try:
103                 self.saber.drive(channel, percent)
104             except serial.serialutil.SerialException:
105                 pass
106             sleep(0.1)
```

Figure 67 Motor class main drive() loop

As shown is an emphasis on the drive() function taken from the Motor class pictured below. This design makes the control of the respective sides simple and streamlined. Within the thread the percent command is captured from the database, type casted to percent (float data type) and passed to the saber object to carry out the commanded percent. This thread is updated every 100ms. This can also be shown in the form of a flow chart to help illustrate the loop's behavior below.

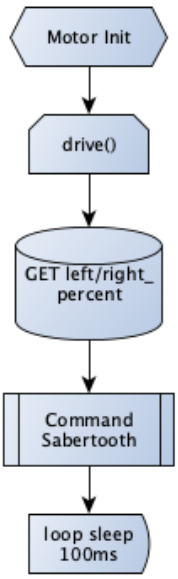


Figure 68 Motor drive() Flowchart Representation

As mentioned, there are two Motor class (pictured above) instances. The overarching process that creates these instances is the run_motors() script. This script is outlined below.

```python
import os, redis
from motor import Motor
from time import time, sleep
from pysabertooth import Sabertooth
from threading import Thread, Lock
from multiprocessing import Process

FULLY_CHARGED = 16.8 # full charge nominal voltage
DEPLETED = 13.6 # CHARGE ME

saber = Sabertooth('/dev/ttyACM0', baudrate=9600, address=128, timeout=0.1)
db = redis.Redis(host='localhost', port=6379, db=0)
left = Motor(name='left', controller_instance=saber, A_pin=23, B_pin=24, volts_feedback_pin=0)
right = Motor(name='right', controller_instance=saber, A_pin=16, B_pin=20, volts_feedback_pin=1)

def drive_left_motor():
    if left is not None:
        l = Thread(target=left.drive, daemon=True)
        l.start()

def drive_right_motor():
    if right is not None:
        r = Thread(target=right.drive, daemon=True)
        r.start()

def left_rpm_feedback():
    lrpm = Process(target=left.post_motor_speeds)
    lrpm.start()

def right_rpm_feedback():
    rrpm = Process(target=right.post_motor_speeds)
    rrpm.start()

def post_battery_voltage():
    volts = 0
    try:
        volts = float(int(str(saber.textGet(b'm1:getb'))[:-2].split('B')[1][:3]) / 10)
    except ValueError:
        volts = 10
    if volts < DEPLETED:
        percent = 0

    percent = round(((volts - 13.2) / 2.0) * 100)
    db.set('system_voltage', volts)
    db.set('soc', percent)

def main():
    drive_left_motor()
    drive_right_motor()
    # left_rpm_feedback()
    # right_rpm_feedback()
    while True:
        post_battery_voltage()
        sleep(30)
        # pass

if __name__ == "__main__":
    pid = os.getpid()
    print(pid)
    main()
```

Figure 69 run_motors() script

The above Python script shows creating the two motor class instances on lines 13 and 14. These both are passed the attributes described above. Focusing on the main() function in this run_motors() script is shown that the threads to drive left and right motors are commenced, while rpm feedback is commented out and not running. Lastly, as mentioned because the battery supply voltage is obtained from the Sabertooth motor controller every 30 seconds the voltage is queried and updated in the database. The separate user interface thread, namely status_leds.py will poll this value and its update is carried out through the system.

Thus far for motor control the final design Motor class has been covered as well as some important notes on efforts for RPM feedback in both initial testing and translated to the final design. Again, although much effort was put toward RPM feedback it was ultimately cut for another system taking precedence.

Now, a look at the manual control aspect of this LLAGV-Locomotion System. To better understand movement capabilities, top speeds and in general to prove out the locomotion hardware, a software system control loop was designed to manually control the LLAGV via an Xbox controller.

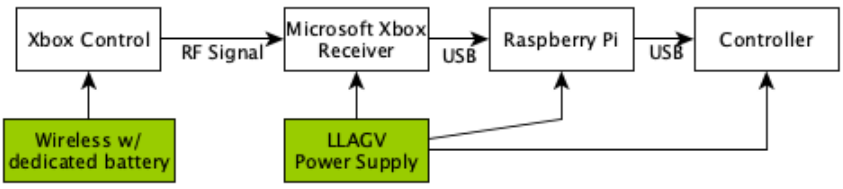


Figure 70 Functional Block Diagram Manual Control

Table 14 Functional Requirements Table for Manual Control

Module	Manual Motor Control Loop
Designer	Marcus R.
Inputs	Xbox Controller Buttons L/R Trigger L/R Bumper

	DPAD UP
Outputs	L/R Traction Commanded %
Description	The program will take inputs from the Xbox 360 controller and apply them to scale output of commanded motor in terms of percent.

This was achieved by connecting a Microsoft® Xbox receiver to one of the Raspberry Pi's USB ports. A python library was pulled, and another interface was written on top of that to control the LLAGV. The Python library handles the dynamics of connecting and interpreting the signals from the controller. The manual control loop is its own process that acts should the system state change to Neutral (manual). This control method also took over as the neutral mode due to the size and girth of the end machine. This added feature makes it much easier for the user to navigate the machine if needed.

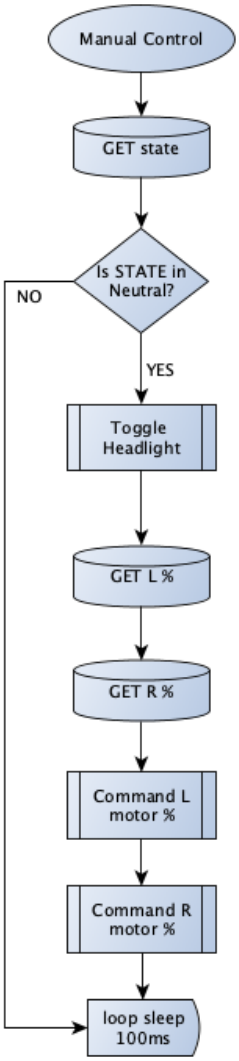


Figure 71 Manual Control Flowchart

Outlined in the above flowchart, when the state is “neutral” it sits in a small loop that checks the DPADUP button on the Xbox controller, and calls a helper function, drive_agv(). If the UP button is pressed the program with toggle the headlights from whatever the prior state was. In the drive_agv() function is where the commanded percent is pulled from the database. Noting that in this function it is checked to see if one of the L/R bumpers are pressed on the controller. If so, then the commanded percent is inverted, and the motor(s) are commanded reverse. See the following illustration of the controller and table outlining the functions of used buttons on the controller with designated reference numbers.

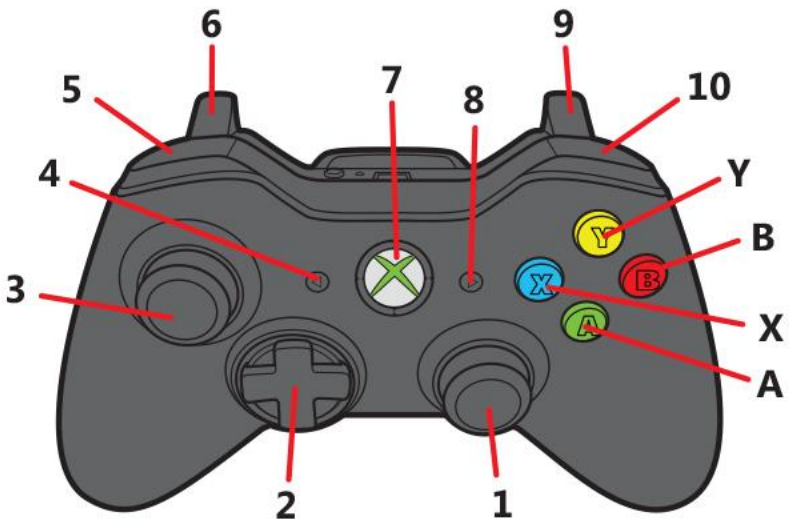


Figure 72 Xbox 360 Control Button Mapping

Table 15 Manual Control Buttons and Functions

Button	Function	Type	Value Range	Controller Ref#
Left Trigger	Drive Left Motors	Float	0 to 1	6
Right Trigger	Drive Right Motors	Float	0 to 1	9
Left Bumper	Invert Left %	Boolean	0 or 1	5
Right Bumper	Invert Right %	Boolean	0 or 1	10
DPAD UP	Toggle Headlights	Boolean	0 or 1	2^

This concludes the discussion of the manual control loop for the locomotion system. Again, this was an added “bonus” feature to compensate for the weight and size of the machine so the user does not have to pick it up should there be unnavigable terrain according to the navigation sensors and data or just a desire to move the LLAGV oneself.

For auto motor control, the systems and design are more complex as expected with having to automate something the user would normally be doing. Revisiting some of the theory related to motor control, below shows a basic outline via functional block diagram.

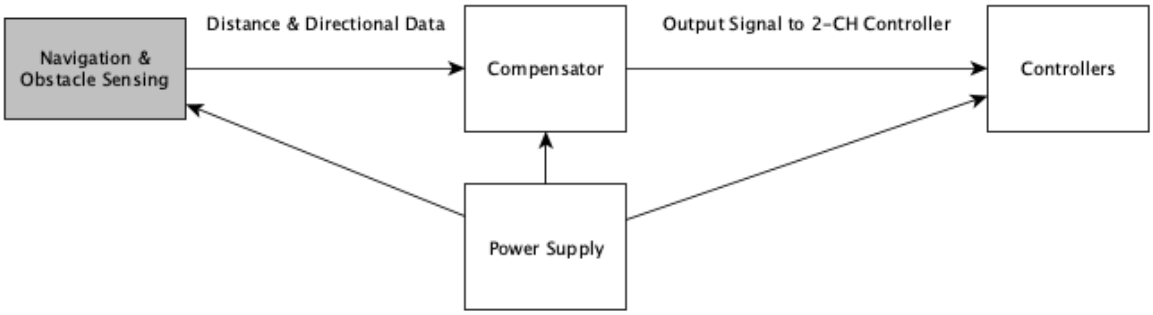


Figure 73 Level 1 Functional Block Diagram for Motor Control

Here, this area of the program will sample navigation and obstacle sensing data from an internal database of the embedded processor. The motor control loop will use this data to dynamically change the setpoint of the compensator. The compensator will attempt to reach its new goal. Meanwhile, as the LLAGV moves the program will continue to sample the navigation and obstacle sensing data. The end result of the compensator box (loop) will be specific commands for the 2-channel motor controller which ultimately decides where the LLAGV goes in direction and speed. Of course, also shown in Figure 73 is the power supply given to all three blocks. Table 16 describes the inputs, outputs and overall thinking behind the functional block diagram for this compensator subsystem.

Table 16 Functional Requirements for Level 1 Compensator

Module	SW Motor Control Loop
Designer	Marcus R. Lawrence S.
Inputs	Distance from User Obstacle Detected Recommended Path Direction
Outputs	L/R Traction Signal

Description	Control loop considers distance and direction data from external sensors (team B) to determine setpoints for compensator. Based on gain values and characteristics of chosen motors, the compensator will drive the 2-CH Motor Controller.

To expand, another iteration on the block diagram above to achieve a deeper understanding of how the motor control loop will operate. Functionally, what is shown in Figure 74 is the same as the prior iteration. That is, the main compensator loop with inputs to the left and outputs to the right. However, in Figure 74, there is a more detailed view of how the control collects its data. Starting from the left, navigation and obstacle sensor data is inserted into a local database where the motor control loop will poll said data. This user distance data, along with obstacle detection data, recommended direction and motor controller feedback will all be taken into consideration for the ultimate setpoint of the compensator as shown in Figure 74. Here, it is shown that the compensator will iterate on itself until the ultimate objective is carried out or complete. As the compensator iterates it communicates over serial (USB) to command the individual motors respectively. The motion of the LLAGV will operate as a tank tread design but with four wheels. If the recommended direction is left, for example, the left wheel would be driven at a slower or stopped position as the right side of the traction will move faster to compensate and overall achieve the “to the left” motion. This of course translates to the right side and will repeat as necessary. Two independent serial commands for channels 1 and 2 drive the respective sides of the LLAGV. The embedded processor can then request the current motor speed and adjust the compensator setpoint as needed for each iteration.

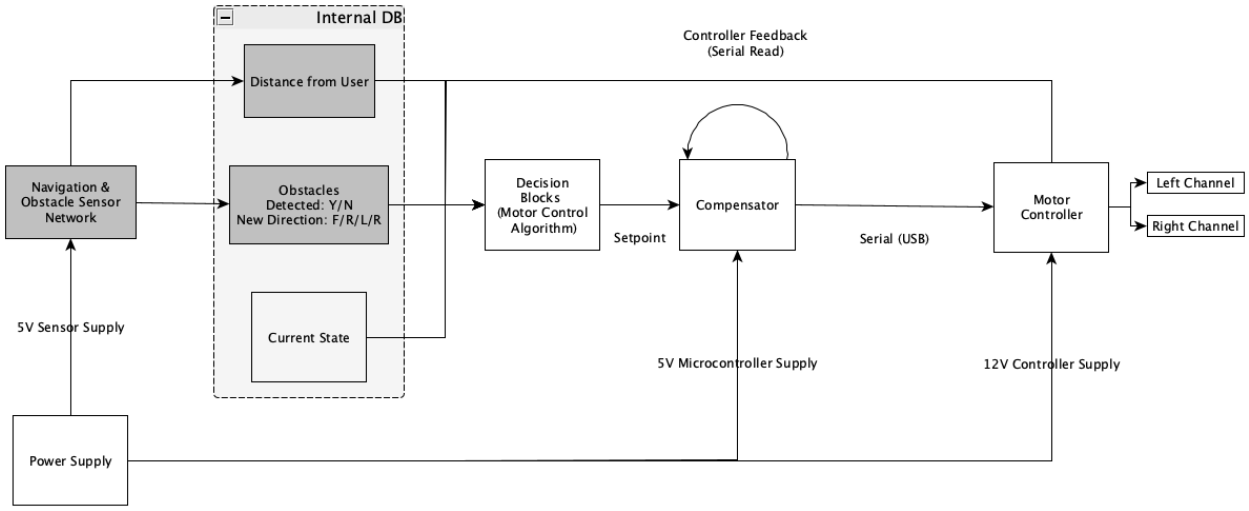


Figure 74 Level 2 Functional Block Diagram for Motor Control

Overall, this process is described in the Table 17 below. The compensator loop accounts for the inputs shown to meet the overall objective of following the user and delivering the load the user placed inside the LLAGV. This is accomplished by polling the listed inputs and adjusting vehicle speed and angle as needed to maneuver itself to the user.

Table 17 Functional Requirements Table for Level 2 Motor Control Block Diagram

Module	SW Motor Control Loop Level 2
Designer	Marcus R.
Inputs	Distance from User Obstacle Detected Recommended Path Direction Current System State Motor Control Feedback
Outputs	L/R Traction Signal
Description	Control loop considers distance and direction data from external sensors (team B), current vehicle speed, and current system state to determine setpoints for compensator. Based on gain values and

	characteristics of chosen motors, the compensator will drive the 2-CH Motor Controller.

For the actual design a look into the python script driving the automated motor control, representing Follow mode, associated flow charts and detailed explanations will follow. Like the manual control loop, this is a process that is always running but does not produce any motor commands unless the switch state changes to Auto. Mentioned on several occasions in other areas of the technical design the RPM feedback had to be turned off. This directly affected the compensator design as it was dependent on that field to create the closed-loop system. To adjust, information from the navigation team was used to fill this missing data. Namely using the distance data from the antenna array. This value was used to linearly scale the drive forward percent commanded to the motors of the LLAGV. As the vehicle approaches the user the vehicle slows and as the user walks away the vehicle speeds up eventually to its max speed to keep up. See the below flow chart for this auto control loop commanding the motors in follow mode.

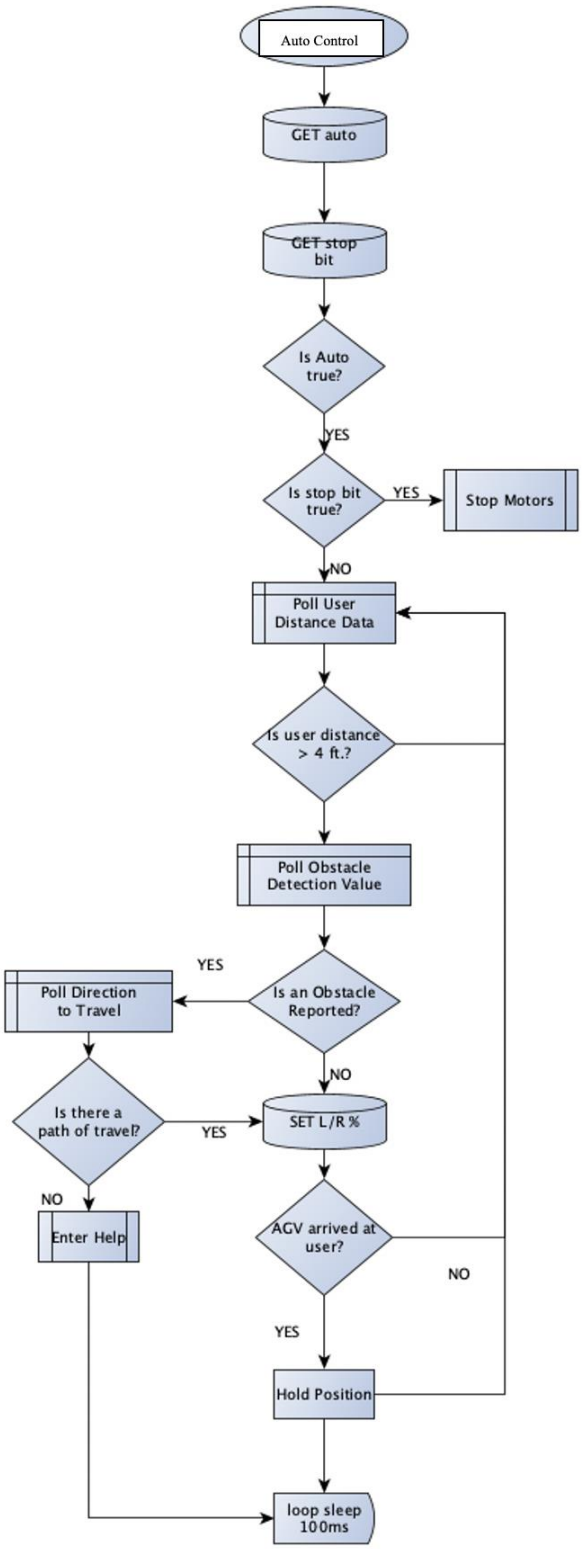


Figure 75 Auto Control Flowchart

The flowchart above illustrates how the overall motor control loop will operate. As mentioned, the motor control loop will be a self-contained thread within the overall application. Here, the motor control loop has access to the data to make decisions by polling navigation and sensor data from a locally stored database. If the system state is in autonomous then the control loop samples the internal database to obtain a distance between the user and the LLAGV. If that reported distance is greater than the set threshold (4 ft.), then the control loop samples the internal database again to obtain obstacle data and acquire a path of travel. If no object is reported, the set distance is simply passed to the compensator to carry out its duty of driving the two motors. **Here, the term "compensator" means the linear scale formulation derived from the lack of RPM feedback.** Note that until the compensator receives an updated setpoint it is continually iterating on itself to get to the user and the embedded processor is consistently sampling the motor controller for feedback. Also, simultaneously while the compensator is working, the control loop is querying the internal database to check if an object is detected, and if yes also poll the new recommended path of travel. This information is continually updates and is passed to the compensator, ultimately adjusting the path of the LLAGV until the vehicle has reached the user. Ideally, the control loop returns to poll the user distance again once the vehicle has successfully arrived at the user and the user is stationary. At this point the vehicle is still in the autonomous state and is simply waiting to follow (user needs to break the threshold again). In a worst-case scenario, while the vehicle is following the user, should there be a loss of sight between the vehicle and user, or the vehicle is in a surrounded situation and there is no recommended path of travel; the application enters the help state. The vehicle in this situation requires the user to maneuver it out of the situation and reset the vehicle in a combination of button pushes to re-enter the autonomous state.

To elaborate on the discussion of autonomous locomotion, it is important to note once again that functions would have been used if the feedback from the motors was not cut. The idea is simple, use the data received from the navigation systems (NS) team to decide whether to change the angle of the LLAGV or drive forward.

To break it down even further, the hierarchy would go as follows. The LLAGV would be constantly received angle variation and distance from the user (NS). In doing so, code was written that handled angle variation as the main priority. It is important to think that the LLAGV

must be within an allowable angle from the user to drive if not, the LLAGV could go anywhere. For this reason, an “if” loop was always checking a threshold before driving forward. The code is represented below.

```
if angle < (ANGLE_DEFAULT - ANGLE_THRESHOLD):
    db.set('headlight', 0)
    turn_right(duration=determine_duration(angle=angle))
if angle > (ANGLE_DEFAULT + ANGLE_THRESHOLD):
    db.set('headlight', 0)
    turn_left(duration=determine_duration(angle=angle))
```

Figure 76: Angle Detection

Both he “ANGLE_DEFAULT” and “ANGLE_THRESHOLD” could be changed inside the code at any moment. Due to accuracy from the navigation team, it was found that ±90° degrees would give the most accurate data. If the current angle detected was outside of that threshold, the LLAGV would initiate a turn command, shown below.

```
def turn_left(duration=0):
    stop()
    db.set("tsig", "left")
    update_left_command(-20)
    update_right_command(20)
    sleep(duration)
    stop()
    db.set("tsig", "off")

def turn_right(duration=0):
    stop()
    db.set("tsig", "right")
    update_left_command(20)
    update_right_command(-20)
    sleep(duration)
    stop()
    db.set("tsig", "off")
```

Figure 77: Turn commands

This above snippet simply does a zero turn in the direction of the desired angle. The important thing to note is the “sleep(duration)” command. This is important as it tells the LLAGV how long to turn for. To find the sleep command, the LLAGV’s measurements were written down and a little bit of trigonometry was used. As you can see from the image below, it

was easiest to think of the LLAGV as a large circle where the four corners of the LLAGV would be the radius of the circle.

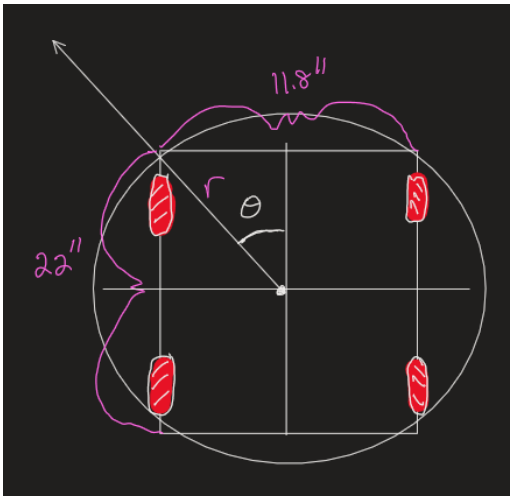


Figure 78: Representation of LLAGV

Given that graphical representation, the radius of that circle was found. Through some calculations it was determined that applying ~2V to the motors (inverted of each other for a zero turn) would yield the following:

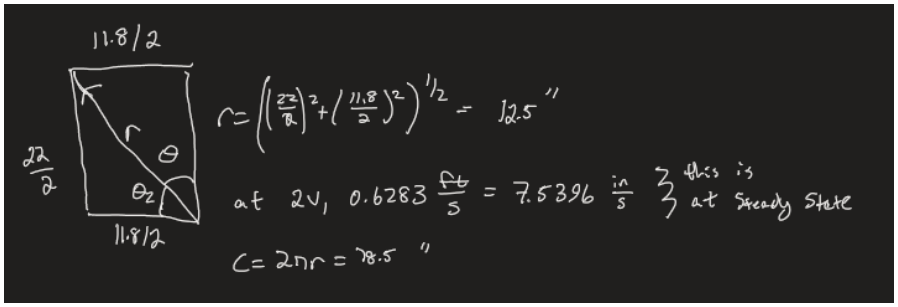


Figure 79: Calculating speed based on voltage

To finish this off, an equation to relate this information to the amount of time needed to turn an exact angle was necessary. Velocity, distance, and time was used to determine said equation. Using the radius of the circle and the angle would also prove useful.

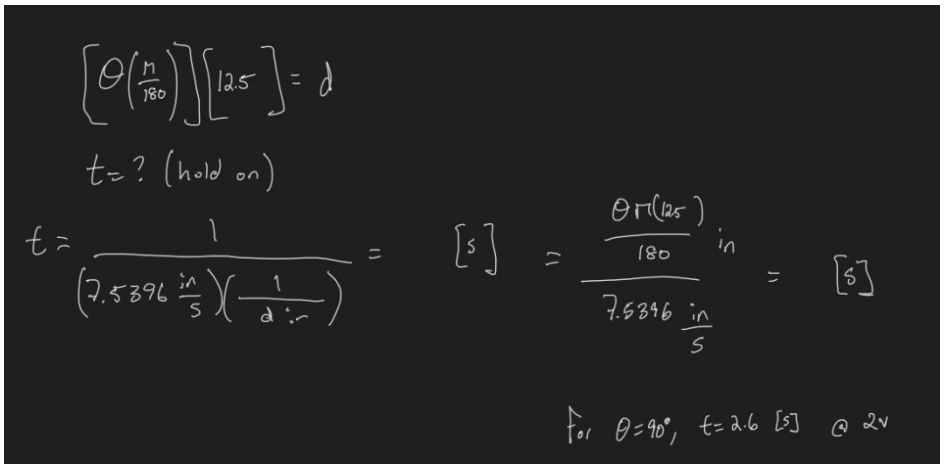


Figure 80: Time related to angle

It can now be seen that time and angle are related. In the example shown, a 90° rotation at 2V to the motor would require roughly 2.6 seconds. The sleep length for the "sleep(duration)" command for exact angle variation needed was found this way.

To summarize this formulation created to replace the PID, it was determined that in practice, giving the LLAGV an exact angle to turn was difficult. The reason being so was the data received from Team B (NS). As found in implementation, due to hardware, Team B's data could sometimes be inconsistent. For example, say this array of data was received from Team B [0 , 0, 0, 50 , 0]. Here, the value "50" would be an error. If the LLAGV's program were to see the "50" and do a 50° turn, the LLAGV would turn the incorrect way and then notice it was wrong and try to turn back. The theory of turning to the exact angle is great if and only if there is no error or lag between data and user. For this reason, it was better to scale the angle. To scale the angle, simply decreased the voltage to the motor by a scale of 0.25. This meant that it would run at ¼ the speed for the required time. Doing so allowed for the LLAGV to "keep up" with (NS) data and work more appropriately.

```
def determine_duration(angle=0) -> float:
    try:
        duration = float((abs(angle) * 3.14 * 12.5 / 180) / 7.54)
    except ZeroDivisionError:
        duration = 0.0
    return duration
```

Figure 81: Implementation of theory

Now that angle variation is concluded, discussion on moving forward can be discussed. The goal was to allow for the LLAGV to be within 3 to 10 feet of the user, when the LLAGV is running. Obviously, if the LLAGV is turned on and the user is further, the LLAGV will catch up. To do so, that distance values from the navigation system (NS) team was used. With the lack of RPM feedback (previously discussed), the distance data from the (NS) was relied on. The distance, known as delta in the code, was greater than 7 feet, 100% speed command to try to catch up. If the distance was then between 4 to 7 feet, the voltage is scaled to the motor to maintain a 4 to 7-foot distance at all times. If the user slows down and stops and the distance from the LLAGV to the user is less than 4 feet away, stop the LLAGV and pause for 5 seconds. Sleeping will allow the LLAGV to ensure it does not hit the user and that the data from the user can be refreshed properly. The code that did this process is included below.

```
# drive forward
if (ANGLE_DEFAULT - ANGLE_THRESHOLD) <= angle <= (ANGLE_DEFAULT + ANGLE_THRESHOLD):
    db.set('headlight', 1)
    # if user is less than 3 ft away
    if delta < 4:
        stop()
        sleep(5)
    elif 4 <= delta <= 7:
        drive_forward(percent=determine_percent(delta - 4)) # operate btw 4-7ft scaled 0-3.2ft/s
    elif delta > 7:
        if str(db.get('turn').decode('UTF=8')) == "turn":
            stop()
            turn_right(duration=determine_duration(angle=160))
            stop()
            drive_forward(100)
            sleep(3)
            stop()
            turn_left(duration=determine_duration(angle=160))
            stop()
            drive_forward(20)
            sleep(2)
            stop()
        drive_forward(100)
```

Figure 82: Drive forward command

The next thing to point out in this auto_control.py script is the “if” statement after the “elif delta > 7” command. This is the object detection. The idea of the object detection was as so. If the user was more than 7 feet away from the LLAGV and the Pi read an ultrasonic sensor had detected something in front, then the LLAGV would stop. The LLAGV would then turn right to 90° (previously mentioned scaling angles due to sharp turns) and would drive forward for three seconds. This would give the LLAGV enough time to clear the object in front of itself. The

LLAGV would then rotate 90° back and go forward for 2 seconds. This gave the LLAGV enough time to slowly creep up and make sure the object is cleared on its side. The program would then resume and the LLAGV would be on its way. If for some reason the LLAGV did not clear the object in front, the program would run again until the object is cleared. This concludes the explanations surrounding the implementation of the autonomous control of the LLAGV.

Recall the process outline in the beginning of the software discussion. So far,

1. launch_ui.py
2. run_motors.py
3. manual_control.py
4. auto_control.py

have been covered. The final major process in need of discussion in the detect.py script. This is a special area which is mostly why it has been saved for the end. Ideally, the navigation system (NS) team would have handled the ultrasonic sensor processing on the Jetson Nano end and the necessary directional information would be made available to the locomotion system (LS) via setting keys in the database. As mentioned in prior sections there was an unforeseen issue with processing ultrasonic Ping® sensors on the Jetson Nano. After browsing development forums, it was determined that the implementation of the Linux OS variant for the Nano (from NVIDIA) did not provision for accurate timing associated with the GPIO pins. The decision was then made to move the sampling of the ultrasonic Ping® sensors to the Raspberry Pi. Unfortunately, this does complicate the discussion and separation between the NS and LS, but it was a compromise that had to be made. This is where the detect.py script comes into play. Below is the detect.py Python script.

```python
import logging, os
from ping import Ping
from threading import Thread, Lock
from time import sleep

ping1 = Ping(num=1, pin=5, location='right')
ping2 = Ping(num=2, pin=9, location='frontR')
ping3 = Ping(num=3, pin=25, location='frontL')
ping4 = Ping(num=4, pin=11, location='left')
ping5 = Ping(num=5, pin=8, location='rear')

pings = [ping1, ping2, ping3, ping4, ping5]

def begin_us_detection():
    for ping in pings:
        p = Thread(target=ping.ping_distance, daemon=True)
        p.start()

def main():
    begin_us_detection()
    while True:
        pass

if __name__ == '__main__':
    pid = os.getpid()
    print(pid)
    main()
```

Figure 83 detect.py script

Here it is shown that there are five class instances of the Ping class. Then each of those sensors act as a python thread within the detect process continually updating the distance read every 1Hz. This is made clear within the actual Ping sensor class written to grab information from the sensors and post that distance data into the database. Again, confusing but yes, the Raspberry Pi (the LS embedded processor) is posting the data of the five sensors. That's it. Because there is the shared database the NS can pick up and actually process the data and set other variables for the motor control threads to act on. The idea has always been for the LS to not act on raw data read from the Ping® sensors, but processed data from the NS. This is still the case, the Raspberry Pi is just taking the place of sampling the sensors, as the Nano was deemed incapable. Once again, because of this shift RPM feedback was removed to allow for this unplanned change.

Next, the Ping Python class was created to handle actions in association with triggering and receiving the pulse from the sensor.

```python
import RPi.GPIO as GPIO
import board, redis
from time import time, sleep

class Ping():

    def __init__(self, num, pin, location):
        self.name = "us_" + str(num)
        self.pin = int(pin)
        self.location = str(location)
        self.db = redis.Redis(host='localhost', port=6379, db=0)

    def trigger(self):
        GPIO.setup(self.pin, GPIO.OUT)
        GPIO.output(self.pin, 0)
        sleep(0.000002)
        GPIO.output(self.pin, 1)
        sleep(0.000005)
        GPIO.output(self.pin, 0)

    def echo(self) -> float:
        starttime = 0
        endtime = 0
        GPIO.setup(self.pin, GPIO.IN)

        while GPIO.input(self.pin) == 0:
            starttime = time()

        while GPIO.input(self.pin) == 1:
            endtime = time()

        duration = endtime - starttime
        return round(duration * 343 / 2, 4) #meters

    def ping_distance(self):
        while True:
            self.trigger()
            distance = self.echo()
            # print(distance, "meters")
            self.db.set(self.name, distance)
            sleep(1)


if __name__ == "__main__":
    ping1 = Ping(num=1, pin=5)
    ping1.ping_distance()
```

Figure 84 Ping class

As shown the Ping sensor class has trigger and echo functions. This functionality was provided from the NS research and development. The ping_distance() function was developed and is what runs as a python thread for each sensor. Here the determined distance is simply pushed to the database every one second. This data is made available to the Nano for logical processing. This

concludes the detect.py script discussion. The process/thread overview in the early software discussion has now been covered in its entirety. Thus, concluding the software application for the locomotion system.

In summary, the locomotion system (LS) software application really deals with two major features. That is, the user interface (UI) and the movement of the LLAGV. Specifically, related to the UI a handful of Python scripts were discussed from the individual classes, the scripts that created the instances of those classes and ultimately how the multi-processing structure was handled on the Raspberry Pi (LS's embedded processor). Motor control, naturally more complex had two other main control loops, manual and auto. The drivability of the LLAGV was first explored with an Xbox controller interface, which later ended up keeping as a "bonus" added feature for the user to more easily move. A major challenge was discussed with dropping the RPM feedback for the motors, eliminating the PID work. This again was due to the need for ultrasonic Ping® sensor sampling to be done on the Raspberry Pi because the Nano was deemed incapable. This RPM feedback information was replaced with less accurate distance data from the NS. The move from a more precise and accurate compensator was switched to a much simpler linear scaling of vehicle speed based on this user distance data. Lastly, there has been many mentions of multi-threading and one may wonder how it is monitored. Unfortunately, a capture was not collected while running on the unit, but a program named dstat was utilized to monitor the four cores of the CPU on the Raspberry Pi.

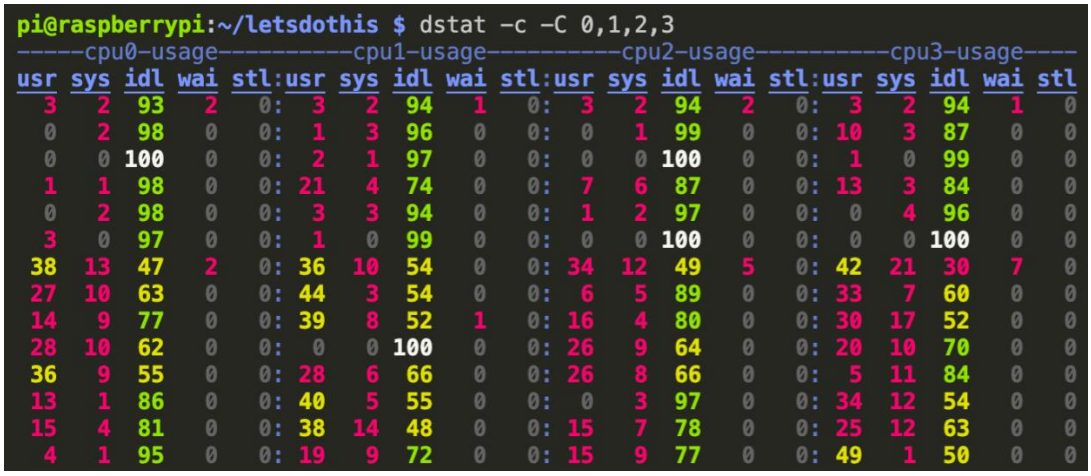


Figure 85 dstat CPU monitoring

Paying specific attention to the idl field in the photo. Ideally this field is close to 100%. When this program was running with the entire LS application running three of the cores were running between 0 and 15%. This is not great, but the fourth CPU core was left to be lighter with idl times in the 70% range. This was necessary because there still needs to be idle time for the OS to carry out its tasks like networking, memory management and handling access to hardware like the GPIO requests in those processes.

6. Mechanical Sketch

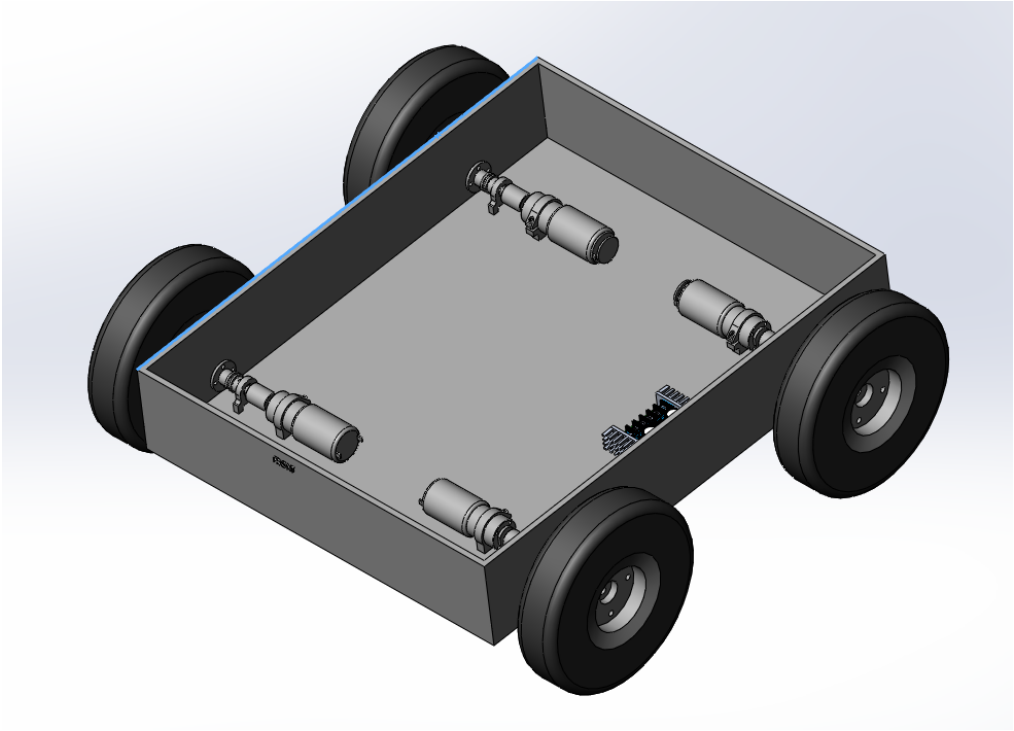


Figure 86:Main housing with drivetrain and wheels

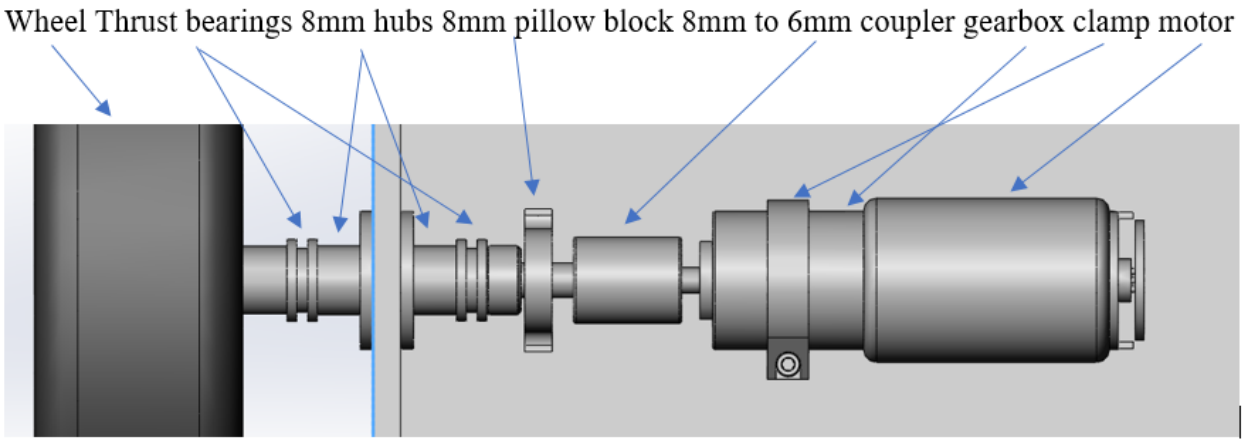


Figure 87: Drivetrain assembly

7. Team Information

Team A – Hardware Lead - Nazar Paramashchuk, EE
Team A – Archivist - Marcus Radtka, CpE
Team A - Project Lead - Lawrence Shevock, EE
Team B – Hardware Lead - John Downey, EE
Team B – Software Lead - Charles Werling, CpE
Team B – Hardware -Robert J Williams II, EE

8. Parts Lists

Table 18 Parts List

5	CBL1-5	415-0031-036	SMA Co-axial cable M-F, 3ft
1	CBL6	415-0029-036	SMA Co-axial cable M-M, 3ft
1	AD1	CAB.S02	SMA to IPEX MHF4 Adapter
1	BT1	8265.NGWMG	Intel Dual Band Wireless-Ac 8265w/Bluetooth
2	M1, M3	RB-Sct-1012	MOTOR WITH ENCODER
2	M2, M4	638324	120 RPM motor without encoder encoder
2	N/A		2 6' wheels with hubs included
1	N/A		Sabertooth dual 32A motor driver
1	N/A		DC converter
4	N/A	OPLC10K	Shaft collar
4	N/A	OPLC10K	Shaft collar
4	N/A	4002-0006-0008	6/8mm shaft coupler
2	N/A	YJ0274MA-AN-M	8mm shaft hub (LINK HAS 4)

4	N/A	555176	Clamping motor mount
4	N/A	1602-0032-0008	8mm pillow block
1	N/A	a19110100ux0846	8mm thrust bearing (LINK HAS 10)
3	F7, F8, F9	0ZCF0500FF2A	PTC RESET FUSE 16V 5A 2920
5	R1-R4	EC2-12NU	RELAY GEN PURPOSE DPDT 2A 12VDC
10	D1-D10	SDURD1040TR	DIODE GEN PURP 400V DPAK
10		Nexperia USA Inc.	MOSFET N-CH 30V 37A LFPAK
10		ON Semiconductor	MOSFET P-CH 30V 25A ATPAK
2		Texas Instruments	IC SYNC SW-MODE BAT CHRGR 16VQFN
2		442060001	CONN HEADER VERT 24POS 4.2MM
1	N/A		Micro SD Card (32GB)
2		39012245	CONN RECEPT 24POS DUAL
100		457503112	CONN SOCKET 16AWG CRIMP TIN
10		TMP6131LPGM	SENSOR PTC 10K OHM 1% TO92S
1		A00000226	100A 100mV SHUNT
20		INR-21700-M50A	21700 5ah 15a
1		B08HR916WK	2 per link
1	S1	EK42442-01	PE42442 4-channel RF switch
1	N/A	6061ASHT125	Aluminum Base
16	N/A	1276-6733-1-ND	CAP CER 0.1UF 100V X7R 0805
10	N/A	1276-2569-1-ND	CAP CER 100PF 100V C0G/NP0 0805
4	N/A	478-8502-1-ND	CAP TANT 2.2UF 20% 25V 0805
6	N/A	478-8927-1-ND	CAP TANT 1UF 10% 20V 0805
3	N/A	478-3286-1-ND	CAP TANT 0.1UF 10% 20V 0805
10	N/A	511-1794-1-ND	CAP TANT 10UF 20% 20V 0805
10	N/A	399-C0805C220K5HAC7800CT-ND	CAP CER 0805 22PF 50V ULTRA STAB
20	N/A	497-10392-1-ND	DIODE SCHOTTKY 100V 3A SMB
20	N/A	732-4990-1-ND	LED GREEN CLEAR 1206 SMD
10	N/A	160-1170-1-ND	LED YELLOW CLEAR SMD
4	N/A	SSC54-E3/57TGICT-ND	DIODE SCHOTTKY 40V 5A DO214AB
4	N/A	BAT54SCT-ND	DIODE ARRAY SCHOTTKY 30V SOT23-3
5	N/A	18-1812L300/24SLERCT-ND	PTC RESET FUSE 3.0A 24V 1812
3	N/A	507-1768-1-ND	PTC RESET FUSE 24V 1.1A 1812
3	N/A	541-1008-1-ND	FIXED IND 2.2UH 8A 20 MOHM SMD
25	N/A	1727-5910-1-ND	MOSFET N-CH 30V 44A LFPAK56
5	N/A	SIR426DP-T1-GE3CT-ND	MOSFET N-CH 40V 30A PPAK SO-8

5	N/A	1727-BUK6Y33-60PXCT-ND	MOSFET P-CH 60V 30A LFPAK56
2	N/A	A143679CT-ND	RES 3550 100R 1%
40	N/A	10-ERA-6VRW1002VCT-ND	RES 10K OHM 0.05% 1/4W 0805
4	N/A	A102096CT-ND	RES SMD 2.49KOHM 0.1% 1/10W 0805
10	N/A	RMCF0805FT383RCT-ND	RES 383 OHM 1% 1/8W 0805
10	N/A	YAG5980CT-ND	RES SMD 330K OHM 1% 1/8W 0805
5	N/A	13-RT0805BRE079KLCT-ND	RES SMD 9K OHM 0.1% 1/8W 0805
3	N/A	311-2813-1-ND	RES SMD 1K OHM 0.5% 1/8W 0805
10	N/A	RMCF0805JT2K00CT-ND	RES 2K OHM 5% 1/8W 0805
3	N/A	541-4132-1-ND	RES SMD 10 OHM 1% 1/8W 0805
3	N/A	541-4166-1-ND	RES SMD 100 OHM 5% 1/8W 0805
10	N/A	541-3978-1-ND	RES SMD 100K OHM 1% 1/8W 0805
3	N/A	311-2822-1-ND	RES SMD 21K OHM 0.5% 1/8W 0805
3	N/A	YAG2002CT-ND	RES SMD 9.31K OHM 0.1% 1/8W 0805
3	N/A	P430KDACT-ND	RES 430K OHM 0.1% 1/8W 0805
3	N/A	YAG1992CT-ND	RES SMD 909K OHM 0.1% 1/8W 0805
20	N/A	RNCF0805BTE10K4CT-ND	RES 10.4K OHM 0.1% 1/8W 0805
10	N/A	13-RT0805FRE072K74LCT-ND	RES SMD 2.74K OHM 1% 1/8W 0805
10	N/A	A143808CT-ND	RES 3550 4R7 5%
2	N/A	696-1268-1-ND	RES 0.01 OHM 5% 35W TO263 DPAK
8	N/A	Z2352-ND	RELAY GEN PURPOSE SPST 10A 12V
2	N/A	296-TLV271QDRG4Q1CT-ND	IC OPAMP GP 1 CIRCUIT 8SOIC
4	N/A	296-45221-1-ND	IC ADC 16BIT SIGMA-DELTA 10VSSOP
2	N/A	296-47740-1-ND	IC BATT CHG LI-ION 1-6CEL 16VQFN
5	N/A	732-5401-ND	CONN HEADER VERT 40POS 2.54MM
4	N/A	H3CCS-4036G-ND	IDC CBL - HHKC40S/AE40G/HHKC40S

The following table includes the final materials budget list corresponding to the budget expense. The total budget was $900.00. This comes from $150.00 per person allowance. All total expenses are below. The total expense was $1,300 as the project also required mechanical expenses. For the size of this project, the overall expense was low.

5	415-0031-036	SMA Co-axial cable M-F, 3ft	$17.79	$88.95
1	415-0029-036	SMA Co-axial cable M-M, 3ft	15.64	15.64
1	CAB.S02	SMA to IPEX MHF4 Adapter	9.00	9.00

1	8265.NGWMG	Intel Dual Band Wireless-Ac 8265w/Bluetooth	23.99	23.99
2	RB-Sct-1012	MOTOR WITH ENCODER	59.99	119.98
2	638324	120 RPM motor without encoder	39.99	79.98
2		2 6' wheels with hubs included	38.92	77.84
1		Sabertooth dual 32A motor driver	124.99	124.99
1		DC converter	16.99	16.99
4	OPLC10K	Shaft collar	1.10	4.40
4	OPLC10K	Shaft collar	1.10	4.40
4	4002-0006-0008	6/8mm shaft coupler	5.00	20.00
2	YJ0274MA-AN-M	8mm shaft hub (LINK HAS 4)	14.00	28.00
4	555176	Clamping motor mount	7.00	28.00
4	1602-0032-0008	8mm pillow block	7.00	28.00
1	a19110100ux0846	8mm thrust bearing (LINK HAS 10)	10.39	10.39
3	0ZCF0500FF2A	PTC RESET FUSE 16V 5A 2920	1.20	3.60
5	EC2-12NU	RELAY GEN PURPOSE DPDT 2A 12VDC	1.97	9.85
10	SDURD1040TR	DIODE GEN PURP 400V DPAK	0.56	5.60
10	Nexperia USA Inc.	MOSFET N-CH 30V 37A LFPAK	0.51	5.10
10	ON Semiconductor	MOSFET P-CH 30V 25A ATPAK	0.61	6.10
2	Texas Instruments	IC SYNC SW-MODE BAT CHRGR 16VQFN	4.77	9.54
2	442060001	CONN HEADER VERT 24POS 4.2MM	2.01	4.02
1		Micro SD Card (32GB)	8.49	8.49
2	39012245	CONN RECEPT 24POS DUAL	1.56	3.12
100	457503112	CONN SOCKET 16AWG CRIMP TIN	0.26	26.00
10	TMP6131LPGM	SENSOR PTC 10K OHM 1% TO92S	0.61	6.10
1	A00000226	100A 100mV SHUNT	13.99	13.99
20	INR-21700-M50A	21700 5ah 15a	7.50	150.00
1	B08HR916WK	2 per link	13.95	13.95
1	EK42442-01	PE42442 4-channel RF switch	95.00	95.00
1	6061ASHT125	Aluminum Base	53.78	53.78
16	1276-6733-1-ND	CAP CER 0.1UF 100V X7R 0805	0.12	1.87
10	1276-2569-1-ND	CAP CER 100PF 100V C0G/NP0 0805	0.07	0.70
4	478-8502-1-ND	CAP TANT 2.2UF 20% 25V 0805	0.85	3.40
6	478-8927-1-ND	CAP TANT 1UF 10% 20V 0805	0.40	2.40
3	478-3286-1-ND	CAP TANT 0.1UF 10% 20V 0805	0.62	1.86
10	511-1794-1-ND	CAP TANT 10UF 20% 20V 0805	0.72	7.15
10	399-C0805C220K5HAC7800CT-ND	CAP CER 0805 22PF 50V ULTRA STAB	0.07	0.71
20	497-10392-1-ND	DIODE SCHOTTKY 100V 3A SMB	0.53	10.56
20	732-4990-1-ND	LED GREEN CLEAR 1206 SMD	0.21	4.20
10	160-1170-1-ND	LED YELLOW CLEAR SMD	0.20	1.98
4	SSC54-E3/57TGICT-ND	DIODE SCHOTTKY 40V 5A DO214AB	0.61	2.44

4	BAT54SCT-ND	DIODE ARRAY SCHOTTKY 30V SOT23-3	0.31	1.24
5	18-1812L300/24SLERCT-ND	PTC RESET FUSE 3.0A 24V 1812	1.95	9.75
3	507-1768-1-ND	PTC RESET FUSE 24V 1.1A 1812	0.22	0.66
3	541-1008-1-ND	FIXED IND 2.2UH 8A 20 MOHM SMD	1.36	4.08
25	1727-5910-1-ND	MOSFET N-CH 30V 44A LFPAK56	0.37	9.34
5	SIR426DP-T1-GE3CT-ND	MOSFET N-CH 40V 30A PPAK SO-8	1.05	5.25
5	1727-BUK6Y33-60PXCT-ND	MOSFET P-CH 60V 30A LFPAK56	0.83	4.15
2	A143679CT-ND	RES 3550 100R 1%	1.61	3.22
40	10-ERA-6VRW1002VCT-ND	RES 10K OHM 0.05% 1/4W 0805	0.59	23.40
4	A102096CT-ND	RES SMD 2.49KOHM 0.1% 1/10W 0805	0.94	3.76
10	RMCF0805FT383RCT-ND	RES 383 OHM 1% 1/8W 0805	0.03	0.26
10	YAG5980CT-ND	RES SMD 330K OHM 1% 1/8W 0805	0.08	0.80
5	13-RT0805BRE079KLCT-ND	RES SMD 9K OHM 0.1% 1/8W 0805	0.40	2.00
3	311-2813-1-ND	RES SMD 1K OHM 0.5% 1/8W 0805	0.12	0.36
10	RMCF0805JT2K00CT-ND	RES 2K OHM 5% 1/8W 0805	0.02	0.17
3	541-4132-1-ND	RES SMD 10 OHM 1% 1/8W 0805	0.10	0.30
3	541-4166-1-ND	RES SMD 100 OHM 5% 1/8W 0805	0.10	0.30
10	541-3978-1-ND	RES SMD 100K OHM 1% 1/8W 0805	0.06	0.60
3	311-2822-1-ND	RES SMD 21K OHM 0.5% 1/8W 0805	0.12	0.36
3	YAG2002CT-ND	RES SMD 9.31K OHM 0.1% 1/8W 0805	0.36	1.08
3	P430KDACT-ND	RES 430K OHM 0.1% 1/8W 0805	0.31	0.93
3	YAG1992CT-ND	RES SMD 909K OHM 0.1% 1/8W 0805	0.36	1.08
20	RNCF0805BTE10K4CT-ND	RES 10.4K OHM 0.1% 1/8W 0805	0.28	5.52
10	13-RT0805FRE072K74LCT-ND	RES SMD 2.74K OHM 1% 1/8W 0805	0.08	0.80
10	A143808CT-ND	RES 3550 4R7 5%	1.45	14.48
2	696-1268-1-ND	RES 0.01 OHM 5% 35W TO263 DPAK	3.63	7.26
8	Z2352-ND	RELAY GEN PURPOSE SPST 10A 12V	1.34	10.72
2	296-TLV271QDRG4Q1CT-ND	IC OPAMP GP 1 CIRCUIT 8SOIC	0.92	1.84
4	296-45221-1-ND	IC ADC 16BIT SIGMA-DELTA 10VSSOP	6.74	26.96
2	296-47740-1-ND	IC BATT CHG LI-ION 1-6CEL 16VQFN	5.06	10.12
5	732-5401-ND	CONN HEADER VERT 40POS 2.54MM	1.13	5.65
4	H3CCS-4036G-ND	IDC CBL - HHKC40S/AE40G/HHKC40S	4.03	16.12
			Total	$1,304.62

9. Project Schedules

A. Final Proposed Gantt Chart

One can see below that the original versus actual Gantt charts were different. The biggest difference is how in the original, most of the "due dates" for testing, implementation, and revision were evenly spread out throughout the semester. However, the team found that testing took most of the time as product delivery and individual testing was behind. Therefore, implementation and revision were the primary focus the last few weeks, prior to design day. This proved to be a challenge however, the team worked efficiently, and we were happy with the time allocated for deliveries.

Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
Manually Scheduled	**SDP2 Implementation 2020**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Manually Scheduled	Revise Gantt Chart	14 days	**Mon 1/11/21**	Sun 1/24/21		
Auto Scheduled	**Implement Project Design**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Auto Scheduled	**Hardware Implementation**	**86 days**	**Mon 1/11/21**	**Tue 4/6/21**		
Manually Scheduled	**Layout and Generate PCB(s)**	**21 days**	**Mon 1/11/21**	**Sun 1/31/21**		
Manually Scheduled	Library Consolidation	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Auto Scheduled	Power Supplies and Distribution	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	Microprocessor Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	Charger Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP

Manually Scheduled	Measurement Peripheral Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	**Assemble Hardware**	**38 days**	**Mon 1/11/21**	**Wed 2/17/21**		
Auto Scheduled	Mechanical LLAGV Housing and Harnessing	38 days	Mon 1/11/21	Wed 2/17/21		LS,NP,RW
Auto Scheduled	Mechanical Mount(s) and Motor(s)	38 days	Mon 1/11/21	Wed 2/17/21		LS,NP,RW
Auto Scheduled	Sensor(s) Mounting	38 days	Mon 1/11/21	Wed 2/17/21		JD,RW,NP
Manually Scheduled	Antenna Mounting	38 days	Mon 1/11/21	Wed 2/17/21		JD,NP
Manually Scheduled	Microprocessor/PCB Mounting	38 days	Mon 1/11/21	Wed 2/17/21		CW,MR,NP,JD

Manually Scheduled	Switches/LEDs UI	38 days	Mon 1/11/21	Wed 2/17/21		MR,NP
Manually Scheduled	**Test Hardware**	**49 days**	**Wed 2/17/21**	**Tue 4/6/21**		
Auto Scheduled	Verify PCB/PDM Connections	12 days	Wed 2/17/21	Sun 2/28/21		MR,NP
Auto Scheduled	Motor Characteristics	12 days	Wed 2/17/21	Sun 2/28/21		LS,MR
Auto Scheduled	Sensor Testing	12 days	Wed 2/17/21	Sun 2/28/21		CW,MR,RW
Manually Scheduled	Antenna Broadcast/BT Receiver	41 days	Wed 2/17/21	Mon 3/29/21		JD
Manually Scheduled	**Revise Hardware**	**12 days**	**Wed 2/17/21**	**Sun 2/28/21**		
Auto Scheduled	Verify PCB/PDM Connections	15 days	Sun 2/14/21	**Sun 2/28/21**		MR,NP
Auto Scheduled	Motor Characteristics	15 days	Sun 2/14/21	**Sun 2/28/21**		LS,MR

Auto Scheduled	Sensor Testing	15 days	Sun 2/14/21	**Sun 2/28/21**		CW,MR,RW
Manually Scheduled	Antenna Broadcast/BT Receiver	15 days	Sun 2/14/21	**Sun 2/28/21**		JD
Auto Scheduled	*MIDTERM: Demonstrate Hardware Subsystems*	5 days	Mon 2/22/21	Fri 2/26/21		CW,JD,LS,MR,NP,RW
Manually Scheduled	SDC & FA Hardware Approval	0 days	Sat 2/27/21	Sat 2/27/21	28	
Manually Scheduled	**Software Implementation**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Manually Scheduled	**Develop Software**	**49 days**	**Mon 1/11/21**	**Sun 2/28/21**		
Auto Scheduled	Overall Architecture (Redis, Multi-threaded Design)	49 days	Mon 1/11/21	Sun 2/28/21		CW,MR

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation Controller Interface (Serial)	16 days	Mon 1/11/21	Sun 2/28/21		LS,MR,RW
Auto Scheduled	UI (Sensory inputs/Display Status)	49 days	Mon 1/11/21	Sun 2/28/21		MR
Manually Scheduled	Camera Object Tracking/Detection	49 days	Mon 1/11/21	Sun 2/28/21		CW
Manually Scheduled	BT RSSI Reading - place in Redis	49 days	Mon 1/11/21	Sun 2/28/21		CW,JD
Manually Scheduled	Obstacle Avoidance (Ultrasonic - last resort safety sys)	49 days	Mon 1/11/21	Sun 2/28/21		CW,MR,RW
Manually Scheduled	**Test Software**	**29 days**	**Sun 2/21/21**	**Sun 3/21/21**		

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation	28 days	Mon 2/22/21	**Sun 3/21/21**		LS,MR
Manually Scheduled	Camera Object Tracking/Detection	12 days	Wed 3/10/21	**Sun 3/21/21**		CW
Auto Scheduled	UI (Sensory inputs/Display Status)	5 days	Wed 3/17/21	**Sun 3/21/21**		MR
Manually Scheduled	BT RSSI Reading - place in Redis	12 days	Wed 3/10/21	**Sun 3/21/21**		CW,JD
Manually Scheduled	Obstacle Avoidance (Ultrasonic - last resort safety sys)	12 days	Wed 3/10/21	**Sun 3/21/21**		CW,MR,RW
Auto Scheduled	**Revise Software**	**28 days**	**Mon 3/8/21**	**Sun 4/4/21**	**31**	

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation	28 days	Mon 3/8/21	Sun 4/4/21		LS,MR
Manually Scheduled	Camera Object Tracking/Detection	19 days	Wed 3/17/21	Sun 4/4/21		CW
Auto Scheduled	UI (Sensory inputs/Display Status)	5 days	Wed 3/31/21	Sun 4/4/21		MR
Manually Scheduled	BT RSSI Reading - place in Redis	19 days	Wed 3/17/21	Sun 4/4/21		CW,JD
Auto Scheduled	*MIDTERM: Demonstrate Software Subsystems*	5 days	Mon 2/22/21	Fri 2/26/21		
Manually Scheduled	SDC & FA Software Approval	0 days	Sat 2/27/21	Sat 2/27/21	49	
Manually Scheduled	**System Integration**	**45 days**	**Mon 1/11/21**	**Wed 2/24/21**		

Manually Scheduled	**Assemble Complete System Integration**	**45 days**	**Mon 1/11/21**	**Wed 2/24/21**		
Manually Scheduled	Power and PCB Layout	45 days	**Mon 1/11/21**	**Wed 2/24/21**		LS,NP
Manually Scheduled	Motor Control	45 days	**Mon 1/11/21**	**Wed 2/24/21**	49	LS,MR
Manually Scheduled	Sensor(s) Integration	45 days	Mon 1/11/21	**Wed 2/24/21**	49	MR,RW
Manually Scheduled	Object Detection	45 days	Mon 1/11/21	**Wed 2/24/21**		CW,MR
Manually Scheduled	Angle of Arrival & Distance	45 days	Mon 1/11/21	**Wed 2/24/21**		JD
Manually Scheduled	**Test Complete System Integration**	**35 days**	**Mon 2/1/21**	**Sun 3/7/21**	**52**	
Manually Scheduled	Motor Control, Compensator Design, & Angle Variation	35 days	**Mon 2/1/21**	**Sun 3/7/21**	52	LS,MR,RW

Manually Scheduled	Object Detection & Course Variation	49 days	**Mon 2/1/21**	**Sun 3/21/21**	52	CW,MR
Manually Scheduled	PCB Communication	35 days	**Mon 2/1/21**	**Sun 3/7/21**	52	LS,NP
Manually Scheduled	Angle of Arrival & Distance	49 days	**Mon 2/1/21**	**Sun 3/21/21**		JD
Manually Scheduled	Sensor(s) Communication	35 days	**Mon 2/1/21**	**Sun 3/7/21**		MR,RW
Manually Scheduled	**Revise Complete System Integration**	**63 days**	**Mon 2/1/21**	**Sun 4/4/21**	**58**	
Manually Scheduled	Motor Control	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	LS,MR,RW
Manually Scheduled	Object Detection & Course Variation	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	CW,MR
Manually Scheduled	PCB & Power	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	LS,NP
Manually Scheduled	Sensor(s)	63 days	**Mon 2/1/21**	**Sun 4/4/21**		MR,RW

Manually Scheduled	Angle of Arrival & Distance	63 days	**Mon 2/1/21**	**Sun 4/4/21**		JD
Auto Scheduled	*Demonstration of Complete System*	5 days	Mon 4/5/21	Fri 4/9/21	64	CW,JD,LS,MR,NP,RW
Manually Scheduled	**Develop Final Report**	**15 days**	**Fri 4/9/21**	**Fri 4/23/21**		CW,JD,LS,MR,NP,RW
Manually Scheduled	Write Final Report	15 days	Fri 4/9/21	**Fri 4/23/21**		CW,JD,LS,MR,NP,RW
Manually Scheduled	Submit Final Report	10 days	Fri 4/9/21	**Sun 4/18/21**	72	CW,JD,LS,MR,NP,RW
Manually Scheduled	*Project Demonstration and Presentation*	5 days	Mon 4/5/21	Fri 4/9/21		CW,JD,LS,MR,NP,RW

B. Final Revised Gantt Chart

Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
Manually Scheduled	**SDP2 Implementation 2020**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Manually Scheduled	Revise Gantt Chart	14 days	**Mon 1/11/21**	Sun 1/24/21		

Auto Scheduled	**Implement Project Design**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Auto Scheduled	**Hardware Implementation**	**86 days**	**Mon 1/11/21**	**Tue 4/6/21**		
Manually Scheduled	**Layout and Generate PCB(s)**	**21 days**	**Mon 1/11/21**	**Sun 1/31/21**		
Manually Scheduled	Library Consolidation	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Auto Scheduled	Power Supplies and Distribution	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	Microprocessor Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	Charger Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP
Manually Scheduled	Measurement Peripheral Integration	21 days	Mon 1/11/21	Sun 1/31/21		LS,NP

Manually Scheduled	**Assemble Hardware**	**38 days**	**Mon 1/11/21**	**Wed 2/17/21**		
Auto Scheduled	Mechanical LLAGV Housing and Harnessing	38 days	Mon 1/11/21	Wed 2/17/21		LS,NP,RW
Auto Scheduled	Mechanical Mount(s) and Motor(s)	38 days	Mon 1/11/21	Wed 2/17/21		LS,NP,RW
Auto Scheduled	Sensor(s) Mounting	38 days	Mon 1/11/21	Wed 2/17/21		JD,RW,NP
Manually Scheduled	Antenna Mounting	38 days	Mon 1/11/21	Wed 2/17/21		JD,NP
Manually Scheduled	Microprocessor/PCB Mounting	38 days	Mon 1/11/21	Wed 2/17/21		CW,MR,NP,JD
Manually Scheduled	Switches/LEDs UI	38 days	Mon 1/11/21	Wed 2/17/21		MR,NP
Manually Scheduled	**Test Hardware**	**49 days**	**Wed 2/17/21**	**Tue 4/6/21**		

Auto Scheduled	Verify PCB/PDM Connections	12 days	Wed 2/17/21	Sun 2/28/21		MR,NP
Auto Scheduled	Motor Characteristics	12 days	Wed 2/17/21	Sun 2/28/21		LS,MR
Auto Scheduled	Sensor Testing	12 days	Wed 2/17/21	Sun 2/28/21		CW,MR,RW
Manually Scheduled	Antenna Broadcast/BT Receiver	41 days	Wed 2/17/21	Mon 3/29/21		JD
Manually Scheduled	**Revise Hardware**	**12 days**	**Wed 2/17/21**	**Sun 2/28/21**		
Auto Scheduled	Verify PCB/PDM Connections	15 days	Sun 2/14/21	**Sun 2/28/21**		MR,NP
Auto Scheduled	Motor Characteristics	15 days	Sun 2/14/21	**Sun 2/28/21**		LS,MR
Auto Scheduled	Sensor Testing	15 days	Sun 2/14/21	**Sun 2/28/21**		CW,MR,RW
Manually Scheduled	Antenna Broadcast/BT Receiver	15 days	Sun 2/14/21	**Sun 2/28/21**		JD

Auto Scheduled	*MIDTERM: Demonstrate Hardware Subsystems*	5 days	Mon 2/22/21	Fri 2/26/21		CW,JD,LS,MR,NP,RW
Manually Scheduled	SDC & FA Hardware Approval	0 days	Sat 2/27/21	Sat 2/27/21	28	
Manually Scheduled	**Software Implementation**	**89 days**	**Mon 1/11/21**	**Fri 4/9/21**		
Manually Scheduled	**Develop Software**	**49 days**	**Mon 1/11/21**	**Sun 2/28/21**		
Auto Scheduled	Overall Architecture (Redis, Multi-threaded Design)	49 days	Mon 1/11/21	Sun 2/28/21		CW,MR

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation Controller Interface (Serial)	16 days	Mon 1/11/21	Sun 2/28/21		LS,MR,RW
Auto Scheduled	UI (Sensory inputs/Display Status)	49 days	Mon 1/11/21	Sun 2/28/21		MR
Manually Scheduled	Camera Object Tracking/Detection	49 days	Mon 1/11/21	Sun 2/28/21		CW
Manually Scheduled	BT RSSI Reading - place in Redis	49 days	Mon 1/11/21	Sun 2/28/21		CW,JD
Manually Scheduled	Obstacle Avoidance (Ultrasonic - last resort safety sys)	49 days	Mon 1/11/21	Sun 2/28/21		CW,MR,RW
Manually Scheduled	**Test Software**	**29 days**	**Sun 2/21/21**	**Sun 3/21/21**		

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation	28 days	Mon 2/22/21	**Sun 3/21/21**		LS,MR
Manually Scheduled	Camera Object Tracking/Detection	12 days	Wed 3/10/21	**Sun 3/21/21**		CW
Auto Scheduled	UI (Sensory inputs/Display Status)	5 days	Wed 3/17/21	**Sun 3/21/21**		MR
Manually Scheduled	BT RSSI Reading - place in Redis	12 days	Wed 3/10/21	**Sun 3/21/21**		CW,JD
Manually Scheduled	Obstacle Avoidance (Ultrasonic - last resort safety sys)	12 days	Wed 3/10/21	**Sun 3/21/21**		CW,MR,RW
Auto Scheduled	**Revise Software**	**28 days**	**Mon 3/8/21**	**Sun 4/4/21**	**31**	

Auto Scheduled	Motor Control, Compensator Design, & Angle Variation	28 days	Mon 3/8/21	Sun 4/4/21		LS,MR
Manually Scheduled	Camera Object Tracking/Detection	19 days	Wed 3/17/21	Sun 4/4/21		CW
Auto Scheduled	UI (Sensory inputs/Display Status)	5 days	Wed 3/31/21	Sun 4/4/21		MR
Manually Scheduled	BT RSSI Reading - place in Redis	19 days	Wed 3/17/21	Sun 4/4/21		CW,JD
Auto Scheduled	*MIDTERM: Demonstrate Software Subsystems*	5 days	Mon 2/22/21	Fri 2/26/21		
Manually Scheduled	SDC & FA Software Approval	0 days	Sat 2/27/21	Sat 2/27/21	49	
Manually Scheduled	**System Integration**	**45 days**	**Mon 1/11/21**	**Wed 2/24/21**		

Manually Scheduled	**Assemble Complete System Integration**	**45 days**	**Mon 1/11/21**	**Wed 2/24/21**		
Manually Scheduled	Power and PCB Layout	45 days	**Mon 1/11/21**	**Wed 2/24/21**		LS,NP
Manually Scheduled	Motor Control	45 days	**Mon 1/11/21**	**Wed 2/24/21**	49	LS,MR
Manually Scheduled	Sensor(s) Integration	45 days	Mon 1/11/21	**Wed 2/24/21**	49	MR,RW
Manually Scheduled	Object Detection	45 days	Mon 1/11/21	**Wed 2/24/21**		CW,MR
Manually Scheduled	Angle of Arrival & Distance	45 days	Mon 1/11/21	**Wed 2/24/21**		JD
Manually Scheduled	**Test Complete System Integration**	**35 days**	**Mon 2/1/21**	**Sun 3/7/21**	**52**	
Manually Scheduled	Motor Control, Compensator Design, & Angle Variation	35 days	**Mon 2/1/21**	**Sun 3/7/21**	52	LS,MR,RW

Manually Scheduled	Object Detection & Course Variation	49 days	**Mon 2/1/21**	**Sun 3/21/21**	52	CW,MR
Manually Scheduled	PCB Communication	35 days	**Mon 2/1/21**	**Sun 3/7/21**	52	LS,NP
Manually Scheduled	Angle of Arrival & Distance	49 days	**Mon 2/1/21**	**Sun 3/21/21**		JD
Manually Scheduled	Sensor(s) Communication	35 days	**Mon 2/1/21**	**Sun 3/7/21**		MR,RW
Manually Scheduled	**Revise Complete System Integration**	**63 days**	**Mon 2/1/21**	**Sun 4/4/21**	**58**	
Manually Scheduled	Motor Control	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	LS,MR,RW
Manually Scheduled	Object Detection & Course Variation	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	CW,MR
Manually Scheduled	PCB & Power	63 days	**Mon 2/1/21**	**Sun 4/4/21**	58	LS,NP
Manually Scheduled	Sensor(s)	63 days	**Mon 2/1/21**	**Sun 4/4/21**		MR,RW

Manually Scheduled	Angle of Arrival & Distance	63 days	**Mon 2/1/21**	**Sun 4/4/21**		JD
Auto Scheduled	*Demonstration of Complete System*	5 days	Mon 4/5/21	Fri 4/9/21	64	CW,JD,LS,MR,NP,RW
Manually Scheduled	**Develop Final Report**	**15 days**	**Fri 4/9/21**	**Fri 4/23/21**		CW,JD,LS,MR,NP,RW
Manually Scheduled	Write Final Report	15 days	Fri 4/9/21	**Fri 4/23/21**		CW,JD,LS,MR,NP,RW
Manually Scheduled	Submit Final Report	10 days	Fri 4/9/21	**Sun 4/18/21**	72	CW,JD,LS,MR,NP,RW
Manually Scheduled	*Project Demonstration and Presentation*	5 days	Mon 4/5/21	Fri 4/9/21		CW,JD,LS,MR,NP,RW

10. Conclusions and Recommendations

In conclusion for compensation, the primary objective was to have an input command related to distance of LLAGV to user and an output, transient response, of a voltage across the DC motor(s) controls the speed in correlation to distance. Characteristics of the DC motor were used to model a transfer function. A P-type compensator was designed to slow the settling time of the transient response to achieve the acceleration and velocity needed. Each channel, on the motor control, controlled its own side of the LLAGV. MATLAB was used to find the step plot

and step info. Unfortunately, the compensator could not be used as the ultrasonic sensors did not work on the Jetson (Team B microcontroller). This meant that the team had to decide between using a compensator or having ultrasonic data, which provided safety and object detection. The raspberry pi 4 did not have the computing power to do both encoder feedback and ultrasonics. For this reason, the compensator had to be cut out. This information was replaced less accurate distance data from the antenna array from the NS team. The LLAGV boasted an intuitive user interface to accept inputs via multiple push button switches. An extension of this UI, the LLAGV displays acknowledgments as feedback to ensure the overall user experience executes well. The application for the LLAGV successfully executed a responsive user interface and drive commands both manually and autonomously. One of the most successful parts of the project taking both the NS and LS into account was the implementation of the Redis database. The introduction of the database worked very efficiently to not only to access data across multiple threads but also between two processors. Leveraging the built-in networking capabilities of the Pi and Nano they were connected via an Ethernet CAT6 cable. This allowed a direct connection between the two and a shared in-memory database with no need for an encoding/decoding protocol between the two processors, say a serial UART connection for example. The VIH and PIH allowed the LLAGV to have all necessary circuitry and power distribution to execute locomotion and navigation. The PDM contained the necessary signal measurement and conditioning required for the microcontrollers onboard to calculate positioning and movement commands. It also contained all the circuit protection required for safe operation. The VIH also interfaced with all the switches on the machine and allowed the microcontroller to know the desired user state, as well as incorporated a kill switch for stopping the machine abruptly.

With further recommendations, the team would push for more integration earlier in the semester. Each member found that they needed more time until proper integration could be done. This idea put strain on the integration of the final project. Moving forward, even if team members would not be done themselves, a push towards early integration would be highly recommended. The team also found that although a processor, released by a respected company, seems to work fine to verify each sensor operation before integration. With the complications of the Jetson, sensor issues forced the Raspberry Pi to use its GPIO pins therefore forcing other aspects of the project to be forfeited. This project was a great experience for each team member

as it not only sharpened engineering theory and application but also improved soft skills. As mentioned as a team all would have made changes if to do the project over again which is a sign of growth and learning in general.

11. References

[1] "DYNAPAR," 2020. [Online]. Available: https://www.dynapar.com/technology/encoder_basics/quadrature_encoder/. [Accessed 20 November 2020].

[2] A. Fethi and S. Mehdi, "The effect of AGVs number on a flexible manufacturing system," IEEE, Elazig, 2019.

[3] B. Y. Qi, Q. L. Yang and Y. Y. Zhou, "Application of AGV in intelligent logistics system," IET, Guilin, China, 2015.

[4] L. Sai-nan, "Optimization problem for AGV in automated warehouse system," in *2008 IEEE International Conference on Service Operations and Logistics, and Informatics*, Beijing, China, 2008.

[5] V. Androulakis, J. Sottile, S. Schafrik and Z. Agioutantis, "Elements of Autonomous Shuttle Car Operation in Underground Coal Mines," in *2019 IEEE Industry Applications Society Annual Meeting*, Baltimore, MD, USA, 2019.

[6] P. Mortimer, "Safety aspects of autonomous guided vehicles in automated warehouses," in *IEE Colloquium on Autonomous Guided Vehicles*, Salford, UK, 1991.

[7] M. Z. a. G. L. M. Duinkerken, "Dynamic Free Range Routing for Automated Guided Vehicles," *IEEE,* pp. 312-313, 2006.

[8] H. Almeida, C. Júnior, D. Santos and M. Leles, "Autonomous Navigation of a Small-Scale Ground Vehicle Using Low-Cost IMU/GPS Integration for Outdoor Applications," in *2019 IEEE International Systems Conference (SysCon)*, Orlando, FL, USA, 2019.

Patent References:

Fukui: Toshihito (Machida, JP), Hirayama; Mitsuru (Isehara, JP). US Patent No. 9,325,192. United States Patent and Trademark Office. 4-26-2016. https://patents.justia.com/patent/9325192 . Assignee: NISSAN MOTOR CO., LTD

Doan, Paul George (Macomb, MI). US Patent No. 8,527,153. United States Patent and Trademark Office. 9-13-2013. https://patents.google.com/patent/US8527153B2/en . Assignee: Fori Automation Inc

12. Appendices

Charger

TEXAS INSTRUMENTS

bq24600

SLUS891B – FEBRUARY 2010 – REVISED NOVEMBER 2014

bq24600 Stand-Alone Synchronous Switch-Mode Li-Ion or Li-Polymer Battery Charger with Low I_q

1 Features

- 1.2-MHz NMOS-NMOS Synchronous Buck Converter
- Stand-Alone Charger Support for Li-Ion or Li-Polymer
- 5-V – 28-V VCC Input Voltage Range and Supports 1S-6S Battery Cells
- Up to 10-A Charge Current and Adapter Current
- High-Accuracy Voltage and Current Regulation
  - ±0.5% Charge-Voltage Accuracy
  - ±3% Charge-Current Accuracy
- Integration
  - Internal Loop Compensation
  - Internal Soft Start
- Safety
  - Input Overvoltage Protection
  - Battery Thermistor Sense Hot/Cold Charge Suspend
  - Battery Detection
  - Built-In Safety Timer
  - Charge Overcurrent Protection
  - Battery Short Protection
  - Battery Overvoltage Protection
  - Thermal Shutdown
- Status Outputs
  - Adapter Present
  - Charger Operation Status
- Charge Enable Pin
- 6-V Gate Drive for Synchronous Power Converter
- 30-ns Driver Dead-Time and 99.5% Max. Effective Duty Cycle
- 16-Pin 3.5-mm × 3.5-mm QFN package
- Energy Star Low Quiescent Current I_q
  - < 15-µA Off-State Battery Discharge Current
  - < 1.5-mA Off-State Input Quiescent Current

2 Applications

- Portable Equipment Handle Terminals
- Industrial and Medical Equipment
- Power Tools Appliance
- Mobile Internet Device, and Ultra-Mobile PC

3 Description

The bq24600 is a highly integrated Li-ion or Li-polymer switch-mode battery-charge controller. It offers a constant-frequency synchronous PWM controller with high-accuracy charge current and voltage regulation, charge preconditioning, termination, and charge status monitoring,

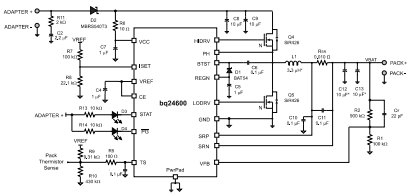
The bq24600 charges the battery in three phases: preconditioning, constant current, and constant voltage. Charge is terminated when the current reaches a minimum level. An internal charge timer provides a safety backup. The bq24600 automatically restarts the charge cycle if the battery voltage falls below an internal threshold, and enters a low-quiescent-current sleep mode when the input voltage falls below the battery voltage.

Device Information(1)

PART NUMBER	PACKAGE	BODY SIZE (NOM)
bq24600	VQFN (16)	3.50 mm x 3.50 mm

(1) For all available packages, see the orderable addendum at the end of the datasheet.

Simplified Schematic



Raspberry Pi 4B

3 Mechanical Specification

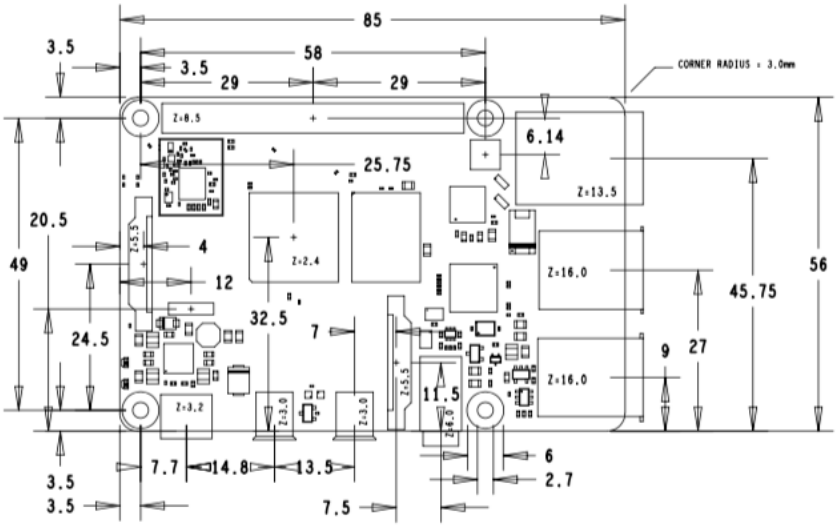


Figure 1: Mechanical Dimensions

ADS1115 - ADC

Ultra-Small, Low-Power, 16-Bit Analog-to-Digital Converter with Internal Reference

Check for Samples: ADS1113 ADS1114 ADS1115

FEATURES

- ULTRA-SMALL QFN PACKAGE: 2mm × 1,5mm × 0,4mm
- WIDE SUPPLY RANGE: 2.0V to 5.5V
- LOW CURRENT CONSUMPTION: Continuous Mode: Only 150µA Single-Shot Mode: Auto Shut-Down
- PROGRAMMABLE DATA RATE: 8SPS to 860SPS
- INTERNAL LOW-DRIFT VOLTAGE REFERENCE
- INTERNAL OSCILLATOR
- INTERNAL PGA
- I²C™ INTERFACE: Pin-Selectable Addresses
- FOUR SINGLE-ENDED OR TWO DIFFERENTIAL INPUTS (ADS1115)
- PROGRAMMABLE COMPARATOR (ADS1114 and ADS1115)

APPLICATIONS

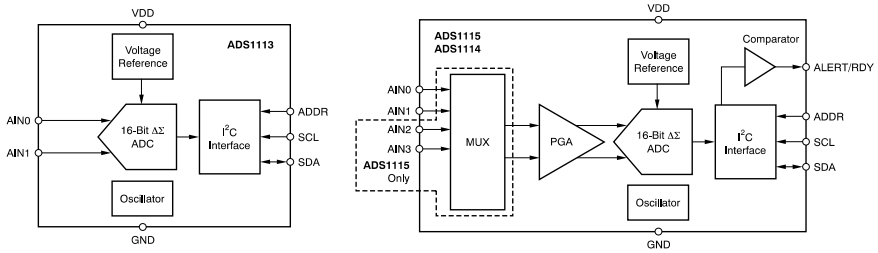
- PORTABLE INSTRUMENTATION
- CONSUMER GOODS
- BATTERY MONITORING
- TEMPERATURE MEASUREMENT
- FACTORY AUTOMATION AND PROCESS CONTROLS

DESCRIPTION

The ADS1113, ADS1114, and ADS1115 are precision analog-to-digital converters (ADCs) with 16 bits of resolution offered in an ultra-small, leadless QFN-10 package or an MSOP-10 package. The ADS1113/4/5 are designed with precision, power, and ease of implementation in mind. The ADS1113/4/5 feature an onboard reference and oscillator. Data are transferred via an I²C-compatible serial interface; four I²C slave addresses can be selected. The ADS1113/4/5 operate from a single power supply ranging from 2.0V to 5.5V.

The ADS1113/4/5 can perform conversions at rates up to 860 samples per second (SPS). An onboard PGA is available on the ADS1114 and ADS1115 that offers input ranges from the supply to as low as ±256mV, allowing both large and small signals to be measured with high resolution. The ADS1115 also features an input multiplexer (MUX) that provides two differential or four single-ended inputs.

The ADS1113/4/5 operate either in continuous conversion mode or a single-shot mode that automatically powers down after a conversion and greatly reduces current consumption during idle periods. The ADS1113/4/5 are specified from –40°C to +125°C.



Software Application Code

battery.py

```
import RPi.GPIO as GPIO
import board, redis
from time import time, sleep

class Battery():

    def __init__(self):
        self.PRECHARGE_PIN = 15
        self.CONTACTOR_PIN = 16
        self.BANK_FDBCK = None

        self.db = redis.Redis(host='localhost', port=6379, db=0)
        self.setup()
        self.disable_precharge()
        self.disable_contactor()

    def setup(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.CONTACTOR_PIN, GPIO.OUT)
        GPIO.setup(self.PRECHARGE_PIN, GPIO.OUT)

    def cleanup(self):
        GPIO.cleanup()

    def enable_contactor(self):
        GPIO.output(self.CONTACTOR_PIN, 0)

    def disable_contactor(self):
        GPIO.output(self.CONTACTOR_PIN, 1)

    def enable_precharge(self):
        GPIO.output(self.PRECHARGE_PIN, 1)

    def disable_precharge(self):
        GPIO.output(self.PRECHARGE_PIN, 0)

    def get_battery_voltage(self):
        pass

if __name__ == '__main__':
    battery = Battery()
    battery.enable_contactor()
```

auto_control.py

```
# Motor Control Script
import math, redis
from time import time, sleep
from pysabertooth import Sabertooth


# TODO make into class
# TODO make sure to avoid __pycache__
ANGLE_DEFAULT = 0
ANGLE_THRESHOLD = 10
wheel_diameter = 0.1524
velocity_ms = 0
velocity_fts = 0.0262 # constant to convert RPM to ft/s
max_rpm = 124
MAX_FORWARD_SPEED = velocity_fts * max_rpm # ft/s
update_rate = 1 # receive a sample every second
PREVIOUS_ANGLE = 0.0
MAX_PERCENT_FORWARD = 25 / 100

saber = Sabertooth('/dev/ttyACM0', baudrate=9600, address=128, timeout=0.1)
db = redis.Redis(host='localhost', port=6379, db=0)

def target_rpm(ft_per_s):
    exptected_rpm = 38.197 * ft_per_s
    return exptected_rpm if exptected_rpm < max_rpm else max_rpm

def convert_rpm_to_out_volts(rpm: float) -> float:
    return rpm / 10.337544 # input voltage should be to reach this RPM

def sample_angle(samples=5) -> float:
    avg = 0.0
    for x in range(0, samples):
        angle = float(str(db.get('angle').decode('UTF=8')))
        avg += angle
        sleep(0.1)
    return avg / samples

def stop():
    db.set('stop_ack', 1)
    update_left_command(0)
    update_right_command(0)
    sleep(1)

def drive_forward(percent):
    update_left_command(percent=(percent*MAX_PERCENT_FORWARD))
```

```
    update_right_command(percent=(percent*MAX_PERCENT_FORWARD))

def turn_left(duration=0):
    stop()
    db.set("tsig", "left")
    update_left_command(-20)
    update_right_command(20)
    sleep(duration)
    stop()
    db.set("tsig", "off")

def turn_right(duration=0):
    stop()
    db.set("tsig", "right")
    update_left_command(20)
    update_right_command(-20)
    sleep(duration)
    stop()
    db.set("tsig", "off")

def drive_agv():
    current_distance = 0.0

    # get values from db
    # angle = sample_angle()
    angle = float(str(db.get('angle').decode('UTF=8')))
    # angle = 0.0 # HACK
    target_distance = float(str(db.get('distance').decode('UTF=8')))
    # target_distance = 10 # HACK

    # if db.get('turn').decode('UTF=8') == 'Left':
    #     turn_left(duration=1)
    #     angle_override = True

    # if db.get('turn').decode('UTF=8') == 'Right':
    #     turn_right(duration=1)
    #     angle_override = True

    # turn if needed
    if angle < (ANGLE_DEFAULT - ANGLE_THRESHOLD):
        db.set('headlight', 0)
        turn_right(duration=determine_duration(angle=angle))
    if angle > (ANGLE_DEFAULT + ANGLE_THRESHOLD):
        db.set('headlight', 0)
        turn_left(duration=determine_duration(angle=angle))
```

```
    PREVIOUS_ANGLE = angle

    delta = abs(target_distance - current_distance)
    # print(delta)
    velocity_of_user = delta / update_rate
    left_fts = float(db.get('left_fts'))
    right_fts = float(db.get('right_fts'))
    plausible = bool(abs(left_fts - right_fts) < 0.5) # check wheel speeds are close to each other, if not maybe stuck
    if not plausible:
        stop()
        db.set("state", "help")
    elif plausible:
        db.set('user_assist_req', 0)
    velocity_of_agv = max(left_fts, right_fts) # take the max of the 2 wheel speeds (if plausible)

    # drive forward
    if (ANGLE_DEFAULT - ANGLE_THRESHOLD) <= angle <= (ANGLE_DEFAULT + ANGLE_THRESHOLD):
        db.set('headlight', 1)
        # if user is less than 3 ft away
        if delta < 4:
            stop()
            sleep(5)
        elif 4 <= delta <= 7:
            drive_forward(percent=determine_percent(delta - 4)) # operate btw 2-5ft scaled 0-3.2ft/s
        elif delta > 7:
            if str(db.get('turn').decode('UTF=8')) == "turn":
                stop()
                turn_right(duration=determine_duration(angle=160))
                stop()
                drive_forward(100)
                sleep(3)
                stop()
                turn_left(duration=determine_duration(angle=160))
                stop()
                drive_forward(20)
                sleep(2)
                stop()
            drive_forward(100)

    # current_distance = target_distance

def determine_duration(angle=0) -> float:
    try:
        duration = float((abs(angle) * 3.14 * 6.23 / 180) / 7.5396) # NOTE Tell Larry to fix
    except ZeroDivisionError:
        duration = 0.0
```

```
    return duration

def determine_percent(command=0):
    rpm = target_rpm(command)
    volts = convert_rpm_to_out_volts(rpm)
    percent = round(volts / 12 * 100)
    if percent >= 100:
        percent = 100
    if percent <=0:
        percent = 0
    return percent

def update_left_command(percent=0):
    db.set('left_percent', percent)

def update_right_command(percent=0):
    db.set('right_percent', percent)

if __name__ == '__main__':
    stop()
    while True:
        auto = bool(int(db.get('auto')))
        stop_bit = bool(int(db.get('stop').decode('UTF=8')))
        # print(stop, type(stop))
        if auto and not stop_bit:
            drive_agv()
        elif auto and stop_bit:
            stop()
        sleep(0.1)
    stop()
```

manual_control.py

```
import RPi.GPIO as GPIO
import xbox, redis
from time import sleep, time

db = redis.Redis(host='localhost', port=6379, db=0)
MAX_PERCENT = 100
MIN_PERCENT = -100

def determine_left_percent(value=0):
    return round(value * MAX_PERCENT)

def determine_right_percent(value=0):
    return round(value * MAX_PERCENT)

def update_left_command(percent=0):
    db.set('left_percent', percent)

def update_right_command(percent=0):
    db.set('right_percent', percent)

def drive_agv():
    lpercent = determine_left_percent(control.leftTrigger()) if not control.leftBumper() else
determine_left_percent(control.leftTrigger()) * -1
    rpercent = determine_right_percent(control.rightTrigger()) if not control.rightBumper() else
determine_right_percent(control.rightTrigger()) * -1

    update_left_command(percent=lpercent)
    update_right_command(percent=rpercent)

def connect():
    control = None
    wait = True

    while wait:
        try:
            control = xbox.Joystick()
            wait = False
        except IOError:
            wait = True
            # db.set('state', 'connecting')
            sleep(2)
    return control
    db.set('state', 'neutral')
```

```
if __name__ == '__main__':
    control = connect()

    while True:
        state = str(db.get('state').decode('UTF=8'))
        if state == 'neutral':
            if control.dpadUp():
                update = 0 if bool((int(db.get('headlight')))) else 1
                db.set('headlight', update)
            drive_agv()
        sleep(0.1)

    control.close()
```

motor.py

```
# imports
import RPi.GPIO as GPIO
import board, busio, csv, redis
from time import sleep, time
import Adafruit_ADS1x15
# from pysabertooth import Sabertooth
# from adafruit_ads1x15.analog_in import AnalogIn

i2c = busio.I2C(board.SCL, board.SDA)
ads = Adafruit_ADS1x15.ADS1115(busnum=1, address=0x48)
outcome = [0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0]

WHEEL_DIAMETER = 0.1524
MAX_RPM = 124
VELOCITY_CONVERT = 0.0262
MAX_FORWARD_SPEED = MAX_RPM * VELOCITY_CONVERT


class Motor():

    def __init__(self, name, controller_instance, A_pin, B_pin, volts_feedback_pin=0, gear_ratio=71, ppr=48):
        self.name = name
        self.saber = controller_instance
        self.A = A_pin
        self.B = B_pin
        self.volt_feedback = volts_feedback_pin
        self.gear_ratio = gear_ratio
        self.ppr = ppr
        self.counts_per_min = ppr * gear_ratio
        self.setup_io()

        self.db = redis.Redis(host='localhost', port=6379, db=0)

    def setup_io(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.A, GPIO.IN)
        GPIO.setup(self.B, GPIO.IN)

    def cleanup(self):
        GPIO.cleanup()
        self.saber.stop()

    def get_supply_voltage(self):
        raw = ads.read_adc(0, gain=1)
        return round(((raw * 4.096) / 32767) * 3.636363, 4)
```

```
def get_instantaneous_rpm(self):
    counter = 0
    last_AB = 0b00
    start = time()

    while (time() - start) <= 1:
        A = GPIO.input(self.A)
        B = GPIO.input(self.B)
        current_AB = (A << 1) | B
        position = (last_AB << 2) | current_AB
        counter += outcome[position]
        last_AB = current_AB
    # print(counter)
    try:
        rpm = round(abs(((counter * 60) / self.counts_per_min)), 4)
    except ZeroDivisionError:
        rpm = 0.0
    return rpm

def get_velocity(self, rpm: float) -> float:
    return round(VELOCITY_CONVERT * rpm, 2) # converts rpm to velocity of motor

def log_rpm(self):
    list_of_tuples = []
    while True:
        try:
            volts = self.get_supply_voltage()
            time, rpm = self.get_instantaneous_rpm(supply_voltage=volts)
            csv_entry = (time, volts, rpm)
            list_of_tuples.append(csv_entry)
            # self.write_to_csv([time, volts, rpm])
            print(csv_entry)

        except KeyboardInterrupt:
            for item in list_of_tuples:
                self.write_to_csv(item)
            del list_of_tuples
            GPIO.cleanup()
            exit()

def post_motor_speeds(self):
    while True:
        rpm = self.get_instantaneous_rpm()
        self.db.set(self.name + '_rpm', rpm)
        self.db.set(self.name + '_fts', self.get_velocity(rpm=rpm))
```

```
        sleep(0.001)

    def drive(self, channel=1, percent=0):
        channel = 1 if self.name == 'left' else 2
        while True:
            try:
                percent = float(self.db.get(self.name + '_percent'))
            except TypeError:
                percent = 0

            try:
                self.saber.drive(channel, percent)
            except serial.serialutil.SerialException:
                pass
            sleep(0.1)
```

run_motors.py

```
import os, redis
from motor import Motor
from time import time, sleep
from pysabertooth import Sabertooth
from threading import Thread, Lock
from multiprocessing import Process

FULLY_CHARGED = 16.8 # full charge nominal voltage
DEPLETED = 13.6 # CHARGE ME

saber = Sabertooth('/dev/ttyACM0', baudrate=9600, address=128, timeout=0.1)
db = redis.Redis(host='localhost', port=6379, db=0)
left = Motor(name='left', controller_instance=saber, A_pin=23, B_pin=24, volts_feedback_pin=0)
right = Motor(name='right', controller_instance=saber, A_pin=16, B_pin=20, volts_feedback_pin=1)

def drive_left_motor():
    if left is not None:
        l = Thread(target=left.drive, daemon=True)
        l.start()

def drive_right_motor():
    if right is not None:
        r = Thread(target=right.drive, daemon=True)
        r.start()

def left_rpm_feedback():
    lrpm = Process(target=left.post_motor_speeds)
    lrpm.start()

def right_rpm_feedback():
    rrpm = Process(target=right.post_motor_speeds)
    rrpm.start()

def post_battery_voltage():
    volts = 0
    try:
        volts = float(int(str(saber.textGet(b'm1:getb'))[:-2].split('B')[1][:3]) / 10)
    except ValueError:
        volts = 10
    if volts < DEPLETED:
        percent = 0

    percent = round(((volts - 13.2) / 2.0) * 100)
    db.set('system_voltage', volts)
    db.set('soc', percent)
```

```
def main():
    drive_left_motor()
    drive_right_motor()
    # left_rpm_feedback()
    # right_rpm_feedback()
    while True:
        post_battery_voltage()
        sleep(30)
        # pass

if __name__ == "__main__":
    pid = os.getpid()
    print(pid)
    main()
```

sim_angle.py

```
import redis
from time import time, sleep

db = redis.Redis(host='localhost', port=6379, db=0)

angles = [0, 0, 0, 0, 0]
distances = [10, 10, 10, 10, 4, 3, 1, 0]


def populate_distances():
    for distance in distances:
        db.set('distance', distance)
        print('Distance', db.get('distance'))
        sleep(1)

def populate_angles():
    for angle in angles:
        db.set('angle', angle)
        print('Angle', db.get('angle'))

if __name__ == '__main__':
    populate_distances()
```

detect.py

```
import logging, os
from ping import Ping
from threading import Thread, Lock
from time import sleep

ping1 = Ping(num=1, pin=5, location='right')
ping2 = Ping(num=2, pin=9, location='frontR')
ping3 = Ping(num=3, pin=25, location='frontL')
ping4 = Ping(num=4, pin=11, location='left')
ping5 = Ping(num=5, pin=8, location='rear')

pings = [ping1, ping2, ping3, ping4, ping5]

def begin_us_detection():
    for ping in pings:
        p = Thread(target=ping.ping_distance, daemon=True)
        p.start()

def main():
    begin_us_detection()
    while True:
        pass

if __name__ == '__main__':
    pid = os.getpid()
    print(pid)
    main()
```

ping.py

```
import RPi.GPIO as GPIO
import board, redis
from time import time, sleep

class Ping():

    def __init__(self, num, pin, location):
        self.name = "us_" + str(num)
        self.pin = int(pin)
        self.location = str(location)
        self.db = redis.Redis(host='localhost', port=6379, db=0)

    def trigger(self):
        GPIO.setup(self.pin, GPIO.OUT)
        GPIO.output(self.pin, 0)
        sleep(0.000002)
        GPIO.output(self.pin, 1)
        sleep(0.000005)
        GPIO.output(self.pin, 0)

    def echo(self) -> float:
        starttime = 0
        endtime = 0
        GPIO.setup(self.pin, GPIO.IN)

        while GPIO.input(self.pin) == 0:
            starttime = time()

        while GPIO.input(self.pin) == 1:
            endtime = time()

        duration = endtime - starttime
        return round(duration * 343 / 2, 4) #meters

    def ping_distance(self):
        while True:
            self.trigger()
            distance = self.echo()
            # print(distance, "meters")
            self.db.set(self.name, distance)
            sleep(1)


if __name__ == "__main__":
    ping1 = Ping(num=1, pin=5)
```

```
ping1.ping_distance()
```

launch_ui.py

```
import RPi.GPIO as GPIO  # import GPIO
import logging, os, redis
from switch import Switch
from status_leds import StatusLEDs
from hx711 import HX711
from threading import Thread, Lock
from multiprocessing import Process
from time import sleep
GPIO.setmode(GPIO.BCM)  # set GPIO pin mode to Board Based numbering

logging.basicConfig(filename='/home/pi/Documents/agv.txt', level=logging.DEBUG)
db = redis.Redis(host='localhost', port=6379, db=0)
hx = HX711(dout_pin=10, pd_sck_pin=7)

neutral = Switch(name='neutral', pin=22, led_pin=14)
auto = Switch(name='auto', pin=27, led_pin=4)
switches = [neutral, auto]
leds = StatusLEDs()

def post_weight():
    while True:
        hx.set_scale_ratio(55615.92)  # set ratio for current channel
        weight = round(abs(hx.get_weight_mean(20)), 2)
        db.set('weight', weight)
        if weight >= 27:
            db.set('too_heavy', 1)
        else:
            db.set('too_heavy', 0)
        sleep(2)

def watch_weight():
    # start a thread to monitor the weight in bed
    w = Thread(target=post_weight, daemon=True)
    w.start()

def watch_switches():
    # start a thread for all push button switches
    for switch in switches:
        sw = Thread(target=switch.monitor_switch, daemon=True)
        sw.start()

def begin_feedback():
    # start a thread to monitor state, soc and headlights/directionals
    led = Thread(target=leds.run, daemon=True)
    led.start()
```

```
def plausibility():
    # to make sure the UI updates properly
    state = str(db.get('state').decode('UTF=8'))
    if state == 'auto':
        db.set('auto_led', "on")
        db.set('auto', 1)
        db.set('neutral_led', "off")
        db.set('neutral', 0)
    elif state == 'neutral':
        db.set('auto_led', "off")
        db.set('auto', 0)
        db.set('neutral_led', "on")
    elif state == 'startup':
        db.set('auto_led', "strobe")
        db.set('neutral_led', "strobe")
    if bool(int(db.get('too_heavy'))):
        db.set('state', 'help')

def main():
    watch_weight()
    watch_switches()
    begin_feedback()
    while True:
        plausibility()
        sleep(0.1)

if __name__ == '__main__':
    pid = os.getpid()
    print(pid)
    main()
```

led_colors.py

```
RED = (255, 0, 0, 0)
GREEN = (0, 255, 0, 0)
BLUE = (0, 0, 255, 0)
WHITE = (0, 0, 0, 255)
YELLOW = (255, 150, 0, 0)
ORANGE = (255, 128, 0, 0)
OFF = (0,0,0,0)
```

status_leds.py

```
import board, neopixel, redis
from led_colors import *
from time import sleep, time

n = 16
leds = neopixel.NeoPixel(pin=board.D18, n=n, brightness=1, pixel_order=(1,0,2,3), auto_write=False)
hdlt_left = list(range(4, 7))
hdlt_right = list(range(8, 12))

class StatusLEDs():

    def __init__(self):
        self.db = redis.Redis(host='localhost', port=6379, db=0)

    def display_soc(self, soc):
        if 0 < soc <= 25:
            leds[1] = OFF
            leds[2] = OFF
            leds[3] = OFF
            intensity = soc * 10
            leds[0] = (intensity, 0, 0, 0)
        elif 26 <= soc <= 50:
            leds[0] = RED
            leds[2] = OFF
            leds[3] = OFF
            intensity = (soc - 25) * 10
            leds[1] = (intensity, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
        elif 51 <= soc <= 75:
            leds[0] = RED
            leds[1] = YELLOW
            leds[3] = OFF
            intensity = (soc - 50) * 10
            leds[2] = (0, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
        elif 76 <= soc <= 100:
            leds[0] = RED
            leds[1] = YELLOW
            leds[2] = GREEN
            intensity = (soc - 75) * 10
            leds[3] = (0, intensity, 0, 0) if intensity >= 11 else (0,0,0,0)
        else:
            raise AttributeError(f"Impossible SOC: {soc}")
        leds.show()

    def chase(self, start_led, end_led, color=GREEN, forward=True, delay=0.1):
        if forward:
```

```
        for led in range(start_led, end_led):
            leds[led] = OFF
            leds.show()
            sleep(delay)
            leds[led] = color
            leds.show()
            sleep(delay)

    elif not forward:
        for led in range(end_led, start_led, -1):
            leds[led] = OFF
            leds.show()
            sleep(delay)
            leds[led] = color
            leds.show()
            sleep(delay)

def strobe(self, start_led, end_led, delay=0.25, color=RED):
    for led in range(start_led, end_led):
        leds[led] = OFF
    leds.show()
    sleep(delay)

def turn_on_headlights(self):
    for led in range(4, n-4):
        leds[led] = WHITE

def turn_off_headlights(self):
    for led in range(4, n-4):
        leds[led] = OFF

def signal_left(self):
    self.chase(start_led=3, end_led=7, color=ORANGE, forward=False)

def signal_right(self):
    self.chase(start_led=8, end_led=12, color=ORANGE)

def display_state(self, state):
    color = OFF
    if state == 'neutral':
        color = YELLOW
    elif state == 'auto':
        color = GREEN
    elif state == 'startup':
        color = BLUE
    elif state == 'help':
```

```
            color = RED
        for led in range(n-4, n):
            leds[led] = color

    def display_tsignal(self, directional='off'):
        if directional == 'left':
            self.signal_left()
        elif directional == 'right':
            self.signal_right()

    def cleanup(self):
        leds.fill(OFF)

    def run(self):
        while True:
            #get current state from redis
            self.display_state(state=str(self.db.get('state').decode('UTF=8')))
            #get soc from redis
            self.display_soc(soc=float(self.db.get('soc')))

            #headlights?
            if bool(int(self.db.get('headlight'))):
                self.turn_on_headlights()
            else:
                self.turn_off_headlights()
            #signals?
            tsig = str(self.db.get('tsig').decode('UTF=8'))
            if tsig != "off":
                self.display_tsignal(directional=tsig)
            sleep(0.5)



if __name__ == "__main__":
    feedback = StatusLEDs()

    for soc in range(100, 0, -1):
        feedback.display_soc(soc)
        sleep(5)
    feedback.cleanup()
```

switch.py

```
import RPi.GPIO as GPIO
import board, redis
from time import time, sleep

debounce_ms = 350
debounce = debounce_ms / 1000


class Switch():

    def __init__(self, name, pin, led_pin=None, default_state=None):
        self.switch_name = name
        self.pin = pin
        self.led = led_pin
        self.latched = False
        self.db = redis.Redis(host='localhost', port=6379, db=0)
        self.setup()

    def setup(self):
        GPIO.setmode(GPIO.BCM)
        GPIO.setup(self.pin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
        if self.led is not None:
            GPIO.setup(self.led, GPIO.OUT)

    def cleanup(self):
        GPIO.cleanup()

    def latch(self):
        # invert the latched state
        self.latched = not self.latched
        self.db.set(self.switch_name, int(self.latched))

        if self.latched is True:
            self.db.set(self.switch_name + "_led", "on")
            self.db.set('state', self.switch_name)
        else:
            self.db.set(self.switch_name + "_led", "off")
        return

    def drive_feedback_led(self, state=False):
        GPIO.output(self.led, state)

    def strobe_led(self, interval=0.5):
        self.drive_feedback_led(state=False)
        sleep(interval)
```

```
        self.drive_feedback_led(state=True)
        sleep(interval)

    def monitor_switch(self):
        while True:
            action = str(self.db.get(self.switch_name + '_led').decode('UTF=8'))
            if action == 'strobe':
                self.strobe_led()
            elif action == 'off':
                self.drive_feedback_led(False)
            else:
                self.drive_feedback_led(True)
            try:
                if GPIO.input(self.pin) == 0:
                    sleep(debounce)
                    if GPIO.input(self.pin) == 0:
                        print("Button Pressed!")
                        self.latch()
                sleep(0.1)
            except KeyboardInterrupt:
                self.cleanup()
                exit()

if __name__ == '__main__':
    power = Switch(name='power', pin=5, led_pin=6)
    power.monitor_switch()
```

agv.py

```
import os, redis

db = redis.Redis(host='localhost', port=6379, db=0)

# init db
db.set('state', 'startup')
db.set('auto', 0)
db.set('neutral', 0)
db.set('left_percent', 0)
db.set('right_percent', 0)
db.set('distance', 0.0)
db.set('angle', 0.0)
db.set('state', 'startup')
db.set('stop', 0)

try:
    os.system('sudo python3 -B /home/pi/letsdothis/ui/launch_ui.py &')
    os.system('sudo python3 -B /home/pi/letsdothis/motor/manual_control.py &')
    os.system('sudo python3 -B /home/pi/letsdothis/motor/auto_control.py &')
    os.system('sudo python3 -B /home/pi/letsdothis/motor/run_motors.py &')
    os.system('sudo python3 -B /home/pi/letsdothis/object_detection/detect.py &')
except KeyboardInterrupt:
    exit()
```

Test Scripts

characterize_motor.py

```
import RPi.GPIO as GPIO
import board, busio, csv
import adafruit_ads1x15.ads1115 as ADS
from adafruit_ads1x15.analog_in import AnalogIn
from pysabertooth import Sabertooth
from time import sleep, time
# import time
i2c = busio.I2C(board.SCL, board.SDA)

A_pin = 5
B_pin = 6
ads = ADS.ADS1115(i2c)
gear_reduction_ratio = 1/71
ppr = 48
countable = ppr / gear_reduction_ratio

GPIO.setmode(GPIO.BCM)
GPIO.setup(A_pin, GPIO.IN)
GPIO.setup(B_pin, GPIO.IN)

outcome = [0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0]


def get_instantaneous_rpm(supply_voltage):
    counter = 0
    last_AB = 0b00
    start = time()

    while (time() - start) <= 1:
        A = GPIO.input(A_pin)
        B = GPIO.input(B_pin)
        current_AB = (A << 1) | B
        position = (last_AB << 2) | current_AB
        counter += outcome[position]
        last_AB = current_AB
    print(counter)

    try:
        rpm = round(abs(((counter * 60) / countable)), 4)
    except ZeroDivisionError:
        rpm = 0.0
    return time(), rpm

def get_supply_voltage():
```

```
    chan = AnalogIn(ads, ADS.P1)
    return round(chan.voltage * 3.63636363, 4)

def log_rpm():
    list_of_tuples = []
    while True:
        try:
            volts = get_supply_voltage()
            time, rpm = get_instantaneous_rpm(supply_voltage=volts)
            csv_entry = (time, volts, rpm)
            list_of_tuples.append(csv_entry)
            write_to_csv([time, volts, rpm])
            print(csv_entry)

        except KeyboardInterrupt:
            for item in list_of_tuples:
                write_to_csv(item)
            del list_of_tuples
            GPIO.cleanup()
            exit()

def write_to_csv(data_list=[], append=True):
    operation = 'w' if append is False else 'a'
    with open('test.csv', operation, newline='') as file:
        rpm_record = csv.writer(file, delimiter=',', quoting=csv.QUOTE_MINIMAL)
        rpm_record.writerow(data_list)


if __name__ == "__main__":
    saber = Sabertooth('/dev/ttyACM0', baudrate=9600, address=128, timeout=0.1)
    header = ['Time', 'PS Voltage', 'Calc RPM']
    write_to_csv(data_list=header, append=False)
    input("Ready? Press [ENTER] when ready!")
    for percent in range(0, 110, 10):
        saber.drive(2, percent)
        sleep(1)
        volts = get_supply_voltage()
        timestamp, rpm = get_instantaneous_rpm(supply_voltage=volts)
        print(f"%: {percent} Time: {timestamp} Volts: {volts} RPM: {rpm}", flush=True)
    # log_rpm()
```

encoder_feedback.py

```
import RPi.GPIO as GPIO
import board, busio
import adafruit_ads1x15.ads1115 as ADS
from adafruit_ads1x15.analog_in import AnalogIn
from time import sleep, time
i2c = busio.I2C(board.SCL, board.SDA)

A_pin = 5
B_pin = 6
ads = ADS.ADS1115(i2c)

GPIO.setmode(GPIO.BCM)
GPIO.setup(A_pin, GPIO.IN)
GPIO.setup(B_pin, GPIO.IN)

def run():

    outcome = [0,1,-1,0,-1,0,0,1,1,0,0,-1,0,-1,1,0]
    last_AB = 0b00
    counter = 0

    while True:
        counter = 0
        start = time()
        while time() - start < 1.0:
            A = GPIO.input(A_pin)
            B = GPIO.input(B_pin)
            current_AB = (A << 1) | B
            #print(f"A: {A} B: {B} AB: {current_AB}", end='\r', flush=True)
            position = (last_AB << 2) | current_AB
            counter += outcome[position]
            last_AB = current_AB
            chan = AnalogIn(ads, ADS.P0)
            sleep(0.000001)

    print(f"A0: {chan.voltage} A: {A} B: {B} Pos: {position} Count: {counter} RPM: {abs(counter*60/900*0.083)}", end="\r",
flush=True)

if __name__ == "__main__":
    run()
```

read_db.py

```
import redis, time

db = redis.Redis(host='localhost', port=6379, db=0)

while True:
    print('Angle', db.get('angle'))
    print('Dist', db.get('distance'))
    print('SOC', db.get('soc'))
    print('RIGHT', db.get('us_1'))
    print('FRONTL', db.get('us_2'))
    print('FRONTR', db.get('us_3'))
    print('LEFT', db.get('us_4'))
    print('REAR', db.get('us_5'))
    print('STOP', db.get('stop'))
    print('TURN', db.get('turn'))
    print('TSIG', db.get('tsig'))
    time.sleep(0.5)
```