Spring 2023

Furniture Mover

Zachariah Burkhardt
The University of Akron, zmb14@uakron.edu

Gary Mucciarone
The University of Akron, gam74@uakron.edu

Gino Mucciarone
The University of Akron, gvm6@uakron.edu

Juan Soto
The University of Akron, jgs65@uakron.edu

David Kotyk
The University of Akron, dmk155@uakron.edu

Senior Design Final Report

Design Project: Furniture Mover

Zachariah Burkhardt

David Kotyk

Gary Anthony Mucciarone

Gino Vincent Mucciarone

Juan Soto

Faculty Advisor: Dr. Mohammed Ali

Date Submitted: 04/24/2023

Table of Contents

List of Figures

List of Tables

0. Abstract

The movement of furniture is an often-overlooked pain point for physically challenged individuals, especially when it comes to rearranging furniture in a room. These individuals may try to minimize the risk of tip-over or strain-related injuries by seeking assistance from others. With that said, some individuals are limited in finding volunteers forthcoming and capable of helping. This can lead to a dilemma of either being able to find individuals willing to assist in the process or risking one's own personal safety when moving furniture. The proposed design project implements the use of multiple, independent platforms with mecanum wheels placed underneath each corner of a piece of furniture by a user. These devices will work together synchronously to move furniture based on commands from a wireless device, allowing for the adjustment of furniture with more flexible positioning and will negate the requirement of human force, preventing physical strain on one's body. [ZB, GVM, JS]

Key Features:

- Mobile app communication
- Rechargeable battery (45-minute runtime)
- Ability to hold up to 100 pounds
- Automatic alignment
- Compatible with multiple furniture leg sizes
- Safe and manageable speed of movement

1. Problem Statement

This section will introduce an overview of the value the project can add to societal needs, the design and conceptual outlines to be implemented, and the wants and needs of the project from a consumer's point of view. [JS]

1.1 Need Statement

According to the U.S Consumer Product Safety Commission, out of 25,500 related tip-over emergency department-treated injuries, adults from the age 18 to 59 years old made up 39% (10,000) of these injuries in 2020 (Division of Hazard Analysis, 2021). These tip-over-related injuries include interaction with furniture where external force was applied. The most common scenario when a great deal of external force is applied to furniture is when it comes to rearranging. Individuals try to minimize this risk of harm by seeking assistance when it comes to adjusting furniture. The only limitation is finding individuals who are forthcoming and capable to help. A system that could assist with the mobile component of remodeling could not only reduce the risks but also help reduce the limitations of this task. [JS]

1.2 Objective Statement

The objective of this project is to design a system to assist individuals when rearranging or moving furniture and eliminate some of the dangerous aspects of these processes. One or more platforms will be placed beneath a piece of furniture and then can be controlled hands-free via a user-friendly interface. The user interface will be implemented on the user’s mobile device for easy use. The application will allow the user to communicate with the device to adjust the furniture with more flexible positioning and prevent physical strain on an individual's body. [ZB, GAM, GVM, JS]

1.3 Background

The movement of furniture is an often-overlooked pain point for those who cannot move furniture. This leads to a dilemma of either finding individuals who are willing to assist in the process or risking one's own personal safety when moving furniture. These challenges were amplified during the COVID-19 pandemic due to social distancing and widespread quarantining. This system is designed to allow individuals to move furniture around with ease and to eliminate the challenge of having to find assistance elsewhere. The basic theory behind this design is to implement an omnidirectional, motorized wheel system that can help individuals eliminate the physical and social struggles of moving furniture using today's available technology. The proposed design implements the use of multiple, independent mecanum wheel devices placed underneath each corner of a piece of furniture by a user. These devices would work together to move furniture based on commands from a wireless device, negating the requirement of human force. With this theory in mind, there are already a few pre-existing solutions that can assist in the movement of furniture. [JS, ZB]

Some of the more basic solutions for furniture movement include using moving blankets, stretch wraps, dollies, lifting straps, and furniture sliders. The use of all these products does make the movement of furniture easier to do, but they still require someone to exert physical work and possibly cause injury (if proper precautions are not observed). One specific patented idea is titled *Dolly for handling furniture* and was invented by Frank J. Mcnamara (U.S. Patent No. 2,627,425, 1953).

The invention by Mcnamara claimed to make the movement of many different styles and types of furniture easier. This was done by designing a compact platform with rotational wheels that could be placed under furniture so that once the platform was set up, the furniture could be

pushed to the designated location with less required effort. While this idea was patented in 1950, the concepts behind it are still being used today. The proposed design objectives are to implement a more electronic approach and to eliminate as much physical stress as possible; however, there are many limitations to what can be done with those two goals in mind. [GVM]

An important consideration when designing a motorized furniture mover is to make sure that the system can handle the weight of the furniture. According to *How much does a couch actually weigh*, the average weight of a sofa can range between 280 to 350 lb or more. This means that to move a piece of furniture evenly distributed between four platforms, each platform would need to handle between 70 and 87.5 pounds or more.

With the system operating in a high-weight scenario, the wheel's durability must also be considered. According to *Design and development of steered active wheel casters and its application*, if there is not a durable wheel to help move around the furniture, it will lead to frequently replacing the wheel that is in use (Ueno et al., 2017). Also, with this motor design, the motor will be working against friction which can also cause wear and tear of the component over time and cause rotational losses and lower speeds for the wheel. Not only will this require replacing the wheels, but this could also cause damage to various types of flooring if they are damaged and not replaced. While considering the limitations of the wheels, the proposed utilization of mecanum wheels was introduced. The use of mecanum wheels does solve some of the limitations discussed above, thus the wheel component in this project will utilize a simplified motorized mecanum wheel design.

The main thing that is looked for here is how the wheel and certain surface areas affect the system. If it can be noticed that the wheels are constantly getting caught on some sort of carpet, it will be known that this system will not be able to operate on that surface. This is

because it will cause further damage to the gears and cause things to get jammed up. According to the article *Floor-types identification method for wheel robot using impedance variation*, with all these varying floors, there is a different amount of energy loss due to slip depending on the surface (Lee et al., 2008). Another potential cause of damage would be if the system is operating on a wooden surface and the tires start to leave marks or scratches on the wooden floor. Based on the previously mentioned article, a method to fix this is having a "brush" follow behind the wheel and polish or wipe off the carpet, wood, and marble floors. This could either be a result of too much force being pushed onto the floor or a flawed wheel design.

There also needs to be a very strong signal range for this system so that the user can use this system anywhere within his or her network range. According to *Networked control systems with delay [tutorial]*, networked systems are not subject to the same design as non-networked systems (Tsoulkas, 2010). This is a common way to connect to devices in today's world and the proposed design will implement the same method to follow marketing standards today. Finally, it is essential to make sure that all four wheels are synchronized when in operation. If there were to be one wheel moving at either a higher speed or a different time, it could damage the furniture itself. [GAM, GVM, JS]

The limitations discussed above were heavily weighed when it came to designing the proposed system. The design of this system has unique aspects but does have similarities to other implemented solutions. In the ICCES article *Furniture that learns to move itself*, it discusses using vibration modules that were designed to attach to any leg of furniture and link together to coordinate the movement to reach a specified location (Parshakova et al., 2017). Similarly, in this design, the system will be able to attach to the bottom of the furniture and link together to coordinate the furniture's movement. The key difference between these two designs is that the

ICCES article design utilizes vibrations to achieve the movement of furniture, while the proposed design will implement a mechanical system with mecanum wheels to achieve this movement.

Another approach is that of a research article titled *RoomShift: Room-scale dynamic haptics for VR with furniture-moving swarm robots* which created robots that move furniture based on a dynamically changing virtual reality (VR) environment (Suzuki et al., 2020). The robots employed mechanical scissor lifts that lifted furniture from underneath and then moved and placed it in a new, specified location. The proposed design solution would also use the idea of the robot moving the furniture from underneath using motorized wheels, but the platforms would not drive themselves beneath the furniture nor lift the furniture using a scissor lift.

In the article *Raspberry pi based remote controlled car using smartphone accelerometer,* it discusses controlling motorized vehicles using a smartphone application to utilize the accelerometer, gyrometer, and magnetometer of a phone to calculate the speed and orientation of the remote-controlled wheels (Kiran & Santhanalakshmi, 2019). Similarly, the idea will utilize a smartphone application to wirelessly control the motorized mecanum wheels but be designed with a user input field to control the orientation and speed of the wheels. However, there are currently existing patented technologies in which their designs can be relevant to what the proposed design is attempting to accomplish. [JS, ZB]

One example of an idea relevant to the proposed design is the *Automotive dolly system* (U.S. Patent No. 7,543,830, 2009). The invention was created by Doug Symiczek and its overall purpose of the invention is to assist in the movement/maneuverability of disabled vehicles without the use of another vehicle, whether it's due to a dead battery, a flat tire, etc. The dolly system also comes with a safety lock to ensure that once the system is set up, it will firmly hold

any vehicle in place, whether it be a sedan or a truck, even if the vehicle is disassembled. There are a few reasons why this invention can be considered relevant to these design goals.

When comparing the *Automotive dolly system* to the design proposed in this paper, the dolly has a number of features needed in this design. First, one of the system's limitations is the weight the wheels would be able to handle. As for this dolly system, being able to hold the weight of a vehicle would suffice in regard to handling the average shared weight on the four platforms. Another design feature used in this idea is its customizable safety lock system for the dolly system to fit all different car models, regardless of whether the vehicle is assembled or not. As part of the proposed design, there needs to be the inclusion of a universal non-slip point of contact, as there are many different designs to furniture. The proposed design uses multiple supports and a non-slip surface that can be attached to the platform at a furniture's point of contact. [GVM, ZB]

The movement of furniture is an industry in which there are many existing designs, but those designs have not employed the benefits of implementing mechanical movement and microcontrollers. Other solutions to furniture relocation, such as moving blankets and furniture straps, do make moving pieces of furniture easier on individuals; however, physical exertion is still necessary for the bulk of the movement even in furniture mover designs that use casters to aid in horizontal movement. By adding motorized casters and wireless control, safety is increased (by reducing risks of furniture falling or lifting-related injuries) and accessibility is expanded for those in which significant physical exertion of energy on furniture is dangerous or impossible. [ZB]

1.4 Marketing Requirements

The marketing requirements for this project are what define what functionality/operation the overall design should implement by the end of the academic year. The requirements include user control, power, and design limitation specifications that will be implemented throughout the entire design phase. [JS]

Table 1: Marketing Requirements

1. The system should be able to remotely move a given piece of furniture
2. The system should be controlled by an external device
3. The system should have a portable, rechargeable energy storage device
4. The system should be compatible with various types of furniture
5. The system should sense and alert when the power source is low/needs replaced

[GVM, ZB, JS]

2. Engineering Analysis

This section will cover the necessities of each of the various subsystems involved with this design, with the analysis examining both the hardware and software theories of operation. The analysis includes calculations along with requirement specifications for each subsystem in order for the completed system to be functionally operational by the end of the academic year. [GVM, JS]

2.1 Circuits

The overall goal of the power subsystem is to store energy after the system has been charged. The energy will then be stored into the batteries as a DC voltage. Another important aspect to this subsystem is to convert the DC voltage stored in the battery to different voltage levels that fit the requirements for the various subsystems involved in the design. The various voltages will be issued where necessary and will lead to powering the system and have it operate through external user inputs. [GVM]

2.1.1 Power Analysis

Looking as there are a few requirements to how the system will be powered for this project, there are a few considerations to be put into place. For starters, as stated in the circuits section above (section 2.1), the system's four platforms will be DC battery powered to allow a clean design within the platforms and will be able to operate with a maximum power draw of 80 watts. This eliminates the worry of power wiring from outlets interfering with the path of movement for the wheels/platforms. As for the battery being used for the system, it is required that the batteries in use must be able to recharge without any concerns of damaging the circuit. There were a few options to consider when thinking of the charging methods that could be used and implemented for the system—these different methods are discussed further in detail in

section 2.2.1. Section 2.2.1 will address the options of using a standard 120VAC 60Hz outlet or using a constant current constant voltage (CCCV) charging method.

Looking into more specific requirements for the power subsystem, the design being implemented is to have a required voltage of approximately 14.8 volts for each platform. As stated in the previous paragraph, the 14.8V will be supplied from a battery source, with the battery in use to be rechargeable from an external power supply to allow for long-term use before the replacement of batteries is necessary. The recharging of the batteries for the system should be easy and have low risk of damage to any subsystems in each platform. Lastly, the batteries in use must be able to operate for a runtime of approximately 45 minutes. This allows plenty of the time for the user to move any furniture necessary before having to recharge the system. [GVM]

2.1.2 Power Calculations

Elaborating on the requirements for the power source of the system, there are a few calculations required to see how much power is necessary for the system to operate. Starting with the use of the electric power equation.

$$P = V * I \quad (1)$$

Looking into equation (1) above and discussing the variables above, with variable P representing power, in units of watts, variable V representing voltage, in units of volts, and finally variable I representing the current, in units of amperes.

Assuming the use of a 14.8V battery, along with knowing the wattage required for the four motors in each platform is 54.8 Watts [W] (this value will be further discussed in section 2.4 below). It can be determined that the current draw from the motors by rearranging equation (1) above gives the following result:

$$I\,(motor) = \frac{P}{V} = \frac{54.8\,[W]}{14.8\,[V]} = 3.70\,[A] \quad (2)$$

Including the current draw for the four alignment sensors (this is up to 100 nA), along with the current draw necessary to operate the microcontroller in the design (this is approximately 20mA or 0.02A), it is calculated that the total current draw for each of the platforms is as follows:

$$I\ (total)\ =\ 3.70\ +\ (4 * 0.0000001)\ +\ 0.02\ =\ 3.73\ [A] \quad (3)$$

With still using the assumption of a 14.8-volt battery, we are able to determine the power necessary for each subsystem within each platform by using equation (1) from above and equation (17) from below.

$$Power\ Needed\ for\ All\ Four\ Motors\ =\ 13.7 * 4 = 54.8\ [W] \quad (4)$$

$$Power\ for\ All\ Four\ Sensors\ =\ 14.8\ [V] *\ 100 * 10^{-9}\ [A] =\ 1.48 * 10^{-6}\ [W] \quad (5)$$

$$Power\ for\ Microcontroller\ =\ 14.8\ [V]\ *\ 0.02\ [A]\ =\ 0.30\ [W] \quad (6)$$

After determining the power necessary for each of the subsystems, it is necessary to determine the total power requirement for each platform as a whole, which goes as seen in equation (7) below: [GVM]

$$Total\ Power\ for\ Each\ Platform\ =\ 54.8 + 1.48 * 10^{-6} + 0.30\ = 55.2\ [W] \quad (7)$$

2.2 Electronics

Within the discussion of electronics, the methods of charging and voltage regulation will be addressed as well as some of the problems faced when considering charging methods and voltage regulation within the system. [JS]

2.2.1 Battery Management System

When working with Lithium-Ion batteries, it is recommended to implement a battery management system (BMS) to safely discharge these types of batteries. The main purpose behind a battery management system is to protect lithium-ion batteries from overheating, as overheating

lithium-ion batteries can lead to a fire hazard. Many different integrated circuits are designed to work specifically with lithium-ion batteries. An integrated circuit is not always required to produce a battery management system, but they are highly recommended to use.

For this project, the integrated circuit (IC) used to generate the battery pack's BMS was the BQ7791500PWT. This IC allows multiple different protection features to safely use and implement the lithium-ion batteries. The protection features included in the BQ7791500PWT are as follows: overvoltage/undervoltage protection, overcurrent charge/discharge, short circuit discharge protection, open wire detection, and overtemperature charge/discharge. Along with these protection features, the BQ7791500PWT also has a programmable cell balancing feature during charging. The BQ7791500PWT allows the battery pack to be protected and for each of the cells within the pack to be equal in charge. [GVM]

2.2.2 Voltage Regulation and Power Output

The 79M05 voltage regulator is characterized by a set of essential parameters that define its performance and functionality. One of the most critical parameters is the output voltage, which, for this specific regulator, is a fixed at 5V. The output voltage accuracy is within 2% of the nominal value, ensuring a stable and precise voltage supply for the connected devices. Another key parameter of the 79M05 regulator is the maximum output current, which is 500mA. This current rating indicates the maximum load that the regulator can safely handle without compromising its performance or reliability. [DMK]

It was also made a requirement to indicate the battery life to the user so they know when to charge the platforms. This was implemented by using a linear light emitting diode (LED) voltage divider, which displays simple red, yellow, and green LEDs to show the user the battery life once power is turned on. However, this is one of the less efficient ways to show the battery

life, as there is voltage sag when the batteries are running, so the LED indicator would be slightly inaccurate while the platform is running. A better approach would have been to design a coulomb counting circuit to keep track of the coulomb level within the pack instead of just reading the voltage of the battery pack. This would allow for a more accurate battery reading and allow the user to see what the pack is at in total. [GVM]

2.3 Communications

The system will consist of multiple communication technologies, including systems for communicating between multiple microprocessors and alignment sensors. This includes communication protocols such as I^2C, SPI, and UART; other communication protocols for communication with a user's mobile device will be discussed further in the Computer Networks section. [ZB]

2.3.1 I^2C

One communication protocol under consideration for usage is Inter-Intergrated Circuit (I^2C). According to *Circuits Basics*, under the I^2C protocol, you are able to have multiple master devices controlling a variable amount of slave devices. The number of masters is not limited in I^2C but, there is a limitation for slaves within I^2C. I^2C utilized Serial Data (SDA) and Serial Clock (SCL) to transmit data between master and slave devices. I^2C is classified as a serial communication which means the SDA line will be used to transfer data bit by bit. One important aspect that I^2C uses while transferring data is the acknowledgment (ACK) bit. The advantage of using ACK within data communication is it lets the master controller know when data has been successfully transferred to the slave controller(s) and when data communication has failed to reach the slaves (Campbell, 2021). [JS]

2.3.2 SPI

Another protocol considered for use for communicating between the microcontroller and sensors/modules is Serial Peripheral Interface (SPI). According to *Seeed Studio,* SPI is a communication protocol typically used for communications between multiple microcontrollers. It has a higher bandwidth than I^2C at rates of 8 Mbps or higher, and it connects with a theoretically unlimited number of SPI devices by using a Chip Select (CS) line. However, by design, SPI requires more pins and wires than I^2C. Additionally, SPI does not have an acknowledgment system to confirm data transmission (Yida, 2022). [ZB]

2.3.3 UART

A final protocol considered for this design was Universal Asynchronous Reception and Transmission (UART). It is also a serial communication protocol that uses a transmit and receive data line and supports two-way data transmission at simplex (one-way transmission), half-duplex (two-way transmission but only one device at a time), or full-duplex (two-way transmission with both devices transmitting simultaneously). There are no clocks in this protocol and instead UART uses a start and stop bit and a set baud rate (transmission rate).

This protocol is appealing due to its simple design, use in many modules (including many commercially-available Bluetooth communication modules), no required clock, and error checking. However, this protocol has low data transmission speeds (slower than I^2C and SPI) and only allows for two devices to connect directly together. In considering these characteristics, the design will use UART for intra-platform communications between local microcontrollers. [ZB]

2.4 Electromechanics

The system will be powered through a battery source supplying DC voltage to the various subsystems. It holds the responsibility of energizing the microcontroller, multiple receiving and

transmitting alignment sensors, and the wheel motors. For this section specifically, an analysis of the wheel motors will be discussed to show the maximum required values of the motor along with the power necessary to operate the motor subsystem. [GVM]

2.4.1 Motor Requirements and Analysis

There are quite a few features and characteristics that can be analyzed to describe the motor requirements to fit the needs of the design. To simplify the search when selecting a specific motor, the characteristics that were focused on were the motor's torque, the motor's revolutions per minute (rpm), and the power necessary to run the motor. The system will operate at a variable speed, meaning the speed will be adjustable by the user, with a maximum operating speed of 0.5 meters per second (m/s). It is also worth considering the weight each wheel has to uphold since each motor will be directly connected to the mecanum wheels on the platform. Each wheel on the platform must be able to hold and move a weight of 8.75 lbs. This weight was calculated assuming 25 lbs on each platform, with each platform weighing about 10 lbs. This leads to an evenly distributed weight between the four wheels on the platform: [GVM]

$$Weight\ on\ Each\ Wheel\ = \frac{(25 + 10)[lbs]}{4[wheels]}\ =\ 8.75\ [lbs/wheel] \quad (8)$$

2.4.2 Motor Calculations

When performing the motor calculations, there are a few values that need to be considered to know the conditions in which the motors have to rotate the wheels they are attached to. These values include but are not limited to: the frictional force that needs to be overcome, the torque of the motor required, and the angular velocity. To begin these calculations, it was first necessary to find the normal force on the wheels of the platform. The equation below will be used to help determine the normal force on the wheels.

$$Normal\ Force\ on\ Wheels: F_N\ [N]\ =\ m\ [kg]\ *\ g\ [m/s^2] \quad (9)$$

With the value being in newtons, we multiply the mass (m) on each wheel by the gravitational acceleration (g) which is equal to 9.81 [m/s^2]. Having each wheel required to hold a maximum weight of 8.75 lbs. each, the first step is converting this value into kilograms (kg). Converting 8.75 lbs. to kilograms gives the value of 3.97 kg. This will be the number used to determine the normal force on each wheel. Plugging these two values into equation (9), the following results are acquired:

$$F_N\ [N] =\ 3.97[kg]\ *\ 9.81\ [m/s^2]\ =\ 38.95\ [N] \quad (10)$$

Once normal force is determined, we can use this to find the frictional force that the wheels need to overcome. Equation (11) below will be the calculation performed to find the force of friction.

$$Frictional\ Force\ to\ overcome: F_{FR}\ [N]\ =\ \mu\ *\ F_N\ [N] \quad (11)$$

With the result being in newtons again, equation (11) states to multiply the normal force that was previously determined (F_N) by the coefficient of friction between the surface of the wheel and the surface in which the wheel is moving on (μ). For this calculation, the coefficient of friction between thermoplastic polyurethane rubber (TPU) on carpeting was used, which is equal to 0.7. Using this along with the recently calculated normal force, the frictional force is determined as:

$$F_{FR}\ [N] = 0.7\ *38.95\ [N]\ =\ 27.27\ [N] \quad (12)$$

The next value necessary for calculations is the torque required by each motor, and it should be noted that in order to find the torque, the radius of the wheel must be chosen and multiplied by the frictional force determined from equation (12). Equation (13) below will be used to determine the torque requirement for the motors.

$$Required\ Torque\ (T)\ [N-m]\ = r_{wheel}\ [m]\ *\ F_{FR}\ [N] \quad (13)$$

Having already determined the frictional force (F_{FR}), all that is left to calculate the torque is plugging in a value for the wheel radius (r_{wheel}). The wheels being used in the design have a radius of 2 inches. Converting this value to meters the radius of the wheel is 0.0508 meters. Multiplying the frictional force by the wheel radius, the torque is calculated as such:

$$Torque\ (T)\ [N-m] = \ 0.0508\ [m]\ *\ 27.27\ [N]\ =\ 1.39\ [N-m] \quad (14)$$

Having calculated the frictional force the motors have to overcome along with the torque requirement for each of the motors, there are two more values that need to be determined. These values are the angular velocity (ω) of the motor, and the power (P_{motor}) each motor needs to operate under the given conditions calculated in equations (9) through (14) above. Moving on to the calculation for angular velocity, to calculate, the use of equation (15) below will be implemented.

$$Angular\ Velocity\ (\omega)\ [rad/sec] = \frac{v[m/s]}{r_{wheel}\ [m]} \quad (15)$$

As stated in section 2.5.1 above, the speed of the system will be able to change, with the maximum velocity (v) of the system being at 0.5 m/s. The radius of the wheel was also discussed earlier and was stated to be 0.0508 meters, so these values can be plugged into equation (15) above to produce the angular velocity calculation.

$$Angular\ Velocity\ (\omega)\ =\ \frac{0.5\ \left[\frac{m}{s}\right]}{0.0508\ [m]}\ =\ 9.843\ [rad/sec] \quad (16)$$

Using equation (15) gives us the angular velocity value in radians/second, however, with most motors on the market, this value is normally stated in revolutions per minute (rpm), so a conversion is necessary. When converting the value of ω = 9.843 [rad/sec] to rpm, the equivalent value comes out to be 93.994 revolutions per minute (rpm). So, it can be determined that the motors to be selected can rotate up to 100 rpm. This should be a high enough rating to account for any error in the calculations above.

For the last calculation in determining the power (P_{motor}) required to run each motor, the calculated torque (T) and angular velocity (ω) are used to find this value. These two values are used to find the power in equations (17) and (18) below.

$$Power\ per\ Motor\ (P_{motor})\ [W]\ =\ T\ [N-m]\ *\ \omega\ [rad/sec] \quad (17)$$

$$P_{motor}\ [W]\ =\ 1.39\ [N-m]\ *\ 9.843\ [rad/sec]\ =\ 13.7\ [W] \quad (18)$$

With the power requirement calculated from equations (17) and (18) above, the selection of a motor is achievable such that the selected motor has the qualifications stated throughout this section. [GVM]

2.4.3 Motor Driver

During operation, the control chip must be able to do logic control on the motion of the motors. As logic/data inputs and outputs are not high in energy, a motor driver must be used to power the motion.

The motor driver at the root is an L298n chip which consists of 2 H-bridges. The basic logic is pairs of high and low. The purpose of a motor driver is to allow the full voltage and current to power the motors because logic voltage and current is way too small. For instance, our logic voltage is 5 V and the max current is limited to 0.5 amps which means the max theoretical power available with logic current and voltage is

$$P_{logic}\ [W]\ =\ 5\ V\ *.5A\ =\ 2.5\ [W]$$

Compared to the H-Bridge that operates at battery pack voltage and has a max current of 2A per channel.

$$P_{H-bridge\ max}\ [W]\ =\ 14\ V\ *2A\ =\ 28\ [W]$$

The H-Bridge allows the motor to operate well within the required power range. [DK]

2.5 Computer Networks

As previously mentioned, the system will consist of multiple communication technologies. Several technologies were considered for communications between the external control device and master platform as well as communications between the master and multiple slave platforms.

Multiple network topologies must be considered when designing the network for this project, including point-to-point, star, and mesh. A point-to-point network topology is used for communications directly between two devices. The two devices act as a transmitter and receiver to each other. A star network topology is similar to a point-to-point topology, but all devices connect to one hub. The stations or nodes wanting to communicate with another station must send their data to the hub which forwards the data to the appropriate recipient. Finally, a mesh network topology is designed so that all nodes are connected to each other, and each node can act as a router.

Point-to-point networks are limited to two devices directly communicating, while star and mesh networks allow for many devices to communicate. Mesh networks are resilient networks and can allow for a lower number of hops between nodes in certain scenarios. However, mesh networks require all nodes to be active and ready to relay messages which increases energy consumption. In contrast, star networks have a single point of failure–the central hub–but allow nodes to conserve energy usage by resting between message transmissions (Proctor, n.d.). [ZB]

2.5.1 Wi-Fi (IEEE 802.11)

The Wi-Fi standard was created by the Institute of Electrical and Electronics Engineers (IEEE). It is commonly used in wireless local area networks (WLAN) in businesses, schools, and homes and utilizes both the 2.4 GHz and 5.0 GHz Industrial, Scientific, and Medical (ISM)

frequency bands, though newer iterations of Wi-Fi, such as Wi-Fi 6E, expand into the 6 GHz band (Afaneh, 2022).

Recent Wi-Fi iterations allow for very high bandwidth, ranging from 54 Mbps to 1.3 Gbps. While this technology would facilitate communications for external user control and inter-platform communications, Wi-Fi is a power-intensive protocol, which is not ideal for a system running on a limited power storage device (such as a battery). [ZB]

2.5.2 Zigbee

Zigbee is a smart home automation protocol based on the IEEE 802.15.4 standard. It operates in the 2.4 GHz ISM band and is typically used in a mesh topology. While Zigbee boasts small power requirements, it only supports small amounts of data transmission (20 - 250 kbps) and is not intended for continuous data throughput. Therefore, Zigbee was not chosen for the design of this system. [ZB]

2.5.3 Thread

Similar to Zigbee, Thread is a recent smart home protocol also based on the IEEE 802.15.4 standard and operates within the 2.4 GHz ISM band. It, too, requires minimal amounts of power but also is only capable of low data transmission rates (20 - 250 kbps) and is intended to be used for sporadic transmission of sensor data and device control. Thus, Thread is inadequate for this system design. [ZB]

2.5.4 Z-Wave

Z-Wave is another protocol for home automation and was created by the Z-Wave Alliance. Notably, it operates in the 908 and 915 MHz bands instead of the congested 2.4 GHz ISM band. However, similar to Zigbee and Thread, this protocol is intended for use in the smart

home space, boasting a throughput of 10 - 100 kbps, and is not designed for continuous data throughput, so Z-Wave was not chosen for this system. [ZB]

2.5.5 LoRa

LoRa is a communication protocol designed for low power usage and long distances, with range estimates between 2 and 20 km. However, to achieve this range and power consumption, LoRa is capable of only 10 - 50 kbps of throughput and infrequent data transmission. Additionally, LoRa capabilities are not included in most phones currently manufactured. Because of the throughput requirements of this project, LoRa is not sufficient for this project's communications. [ZB]

2.5.6 Bluetooth

Bluetooth is a wireless communication protocol that can be used in a variety of scenarios. It includes two operation modes: Bluetooth Classic and Bluetooth Low Energy (LE). Bluetooth Classic is the older Bluetooth operation mode and was the only operating mode used in Bluetooth 3.0 and older Bluetooth standards. In modern applications, it is still used for backward compatibility with older Bluetooth devices as well as bandwidth-intensive applications, such as for streaming audio. Bluetooth LE was introduced in the Bluetooth 4.0 standard as a low-powered alternative to the Bluetooth Classic mode. It is intended for applications with infrequent or low data bandwidth requirements and where minimal power usage is prioritized.

The Bluetooth Classic mode operates in a point-to-point or star network topology, but Bluetooth LE introduces support for two additional topology designs: mesh and broadcast.

The Bluetooth 5 LE standard allows for modulation rates of 1 Mbps and 2 Mbps. Based on the 2 Mbps rate, the maximum data packet length is 265 octets, based on Table 2 below (Ren, 2022).

Table 2: Number of Octets for a Maximum-Size Packet in Bluetooth 5

Preamble	Access Address	Header	Payload	MIC	CRC
1	4	2	251	4	3

To calculate the theoretical effective throughput, the "R" slot (used for confirming successful packet transmission by the receiver) and inter-frame space (T_IFS, which is the interval between consecutive packets) must be considered. The "R" slot is a minimal, empty packet, and the required octets per the Bluetooth 5 specification are displayed in Table 3.

Table 3: Required Octets of an Empty Transmission Packet

Preamble	Access Address	Header	CRC
1	4	2	3

The minimum 10 octets allow for a calculation of the "R" slot time cost, which is 40 μs, as shown in equation 19.

$$\frac{10\ octets\ *\ 8\ bits}{2\ Mbps} = 40\ \mu s \tag{19}$$

Additionally, the T_IFS time cost is 150 μs per the Bluetooth 5 core specification. Finally, based on the total octets of the maximum payload in Table 2, the maximum packet length time is calculated in equation (20).

$$\frac{(1\ +\ 4\ +\ 2\ +\ 251\ +\ 4\ +\ 3)\ octets\ *\ 8\ bits}{2\ Mbps} = 1060\ \mu s \tag{20}$$

These time calculations can then be summed to determine the time of one complete period and used to calculate an effective throughput, as shown in equation 21.

$$Throughput_{effective} = \frac{payload}{one\ complete\ period} = \frac{251\ payload\ octets * 8\ bits}{(40 + 150 + 1060 + 150)\ \mu s} \approx 1.4\ Mbps \quad (21)$$

As an estimation of total bandwidth requirements for this system, consider four data streams (external controller to the master platform and three slaves connected to the master platform). Each device is sent data 1000 times per second. If the external controller transmits one variable of type double for device speed and one variable of type integer for direction, and if the master platform transmits one variable of type integer per platform wheel for each motor's direction and speed, the approximated bandwidth requirements for this system can be estimated as 480 kbps in equation 22, less than the 1.4 Mbps of theoretical throughput.

$$\frac{[(8\ bytes\ +\ 4\ bytes)\ +\ (12\ motors\ *\ 4\ bytes)]\ *\ 8\ bits\ *\ (1000\ updates\ per\ second)}{1000}$$

$$= 480\ kbps \quad (22)$$

Due to this system's requirement for minimal power usage and bandwidth requirements as well as a standard for which many development libraries and modules exist, this project will utilize a Bluetooth 5 network for communications between the external input control and the master platform as well as for inter-platform communications. Additionally, the inter-platform Bluetooth network will be designed as a star network topology to support the master-slave model for platform control (this model is discussed in further detail in section 2.8.1). [ZB]

2.6 Embedded Systems

2.6.1 PIC24

The Microchip PIC24FJ128GA010 microcontroller is a 16-bit general-purpose microcontroller based on a modified Harvard Architecture. The required voltage level to operate the PIC24 falls with the range of 2.4 - 3.6 V. The PIC24 support also extends to I^2C and UART protocol communication. Pin communication for the PIC24 is a simple I/O standard with high

current sinking and sourcing. The microcontroller also utilizes C-based coding to implement any procedural program. [JS]

The microcontroller's CPU supports up to 16 MIPS operations at 32 MHz (32,000,000 cycles per second). It also includes an 8 MHz Internal Oscillator with multiple divide options to slow down the clock. By using a 16x divisor, the microcontroller will be able to process 2,000,000 operations per second, which will be sufficient for processing the control signals of the external user input and calculating control signals for the slave platforms. [ZB]

2.6.2 ESP32

The ESP32 is a popular microcontroller featuring low power draw requirements, inexpensive cost, and expansive connectivity options. The ESP32-C3-Mini-1 system on a chip (SoC) is a 32-bit, single-core processor that follows the RISC-V instruction set architecture. The system firmware is written in C language using an ESP-IDF SDK provided by the manufacturer: Espressif. This SoC requires a power supply voltage between 3.0 and 3.6 V_{DC}. It includes a built-in voltage regulator to step an ~5 V power supply down to 3.3 V, so it can be powered via an included micro-USB connector. This SoC supports multiple communication protocols, including SPI, UART, and I^2C. It can also generate up to six independent pulse-width modulation (PWM) signals with 14 bits of accuracy. Additionally, the ESP32-C3-Mini-1 includes hardware to support WiFi 802.11 b/g/n and Bluetooth 5 LE communications.

Finally, it also supports up to six independent analog-to-digital converters (ADC), each of which provide up to 12-bit sampling resolution at up to 3.3 V. Five of these channels are factory calibrated. The sixth channel is not factory calibrated and is used for WiFi transmission and receiving, by default. However, WiFi functionality can be disabled to utilize this sixth ADC channel.

This processor's clock can be set to up to 160 MHz, allowing for the processing of up to 160,000,000 operations per second. This is sufficient for processing the external user control and slave platform signals. Additionally, the support for Bluetooth 5 LE and multiple ADC channels meets the requirements for wireless communications and the designed alignment system. Finally, the ESP32 boards have many pre-existing tutorials and documentation. For these reasons, the decision was made, following the proof of concept, to transition from the Explorer 16/32 to the ESP32. [ZB]

2.7 Controls

A feedback control allows the robots to automatically adjust their behavior based on the difference between their current state and a desired reference value. This design is a simple feedback control system designed for a robot that relies on infrared (IR) emitters, microcontrollers, and pull-up/pull-down networks of PNP infrared (IR) receivers as sensors.

This system enables the robot to rotate counterclockwise or clockwise to correct its orientation based on the error between the reference value and the current input.

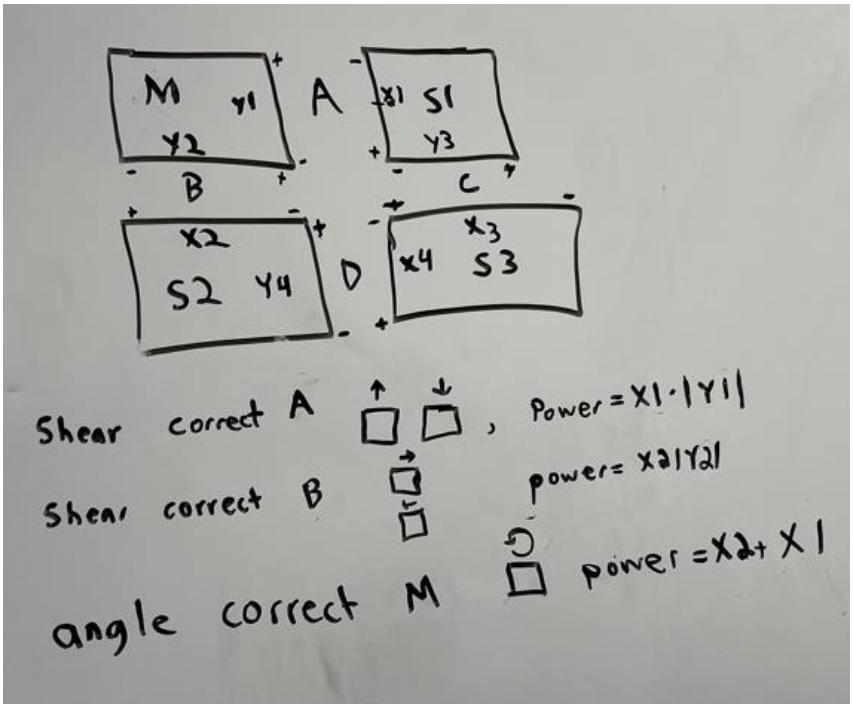
The microcontroller processes the feedback signal and makes software decisions. The microcontroller is the brain of the entire system. Some functions that the microcontroller will operate include signal processing, software decisions, communication protocols, and motor control. Section 2.6 discussed the many considerations of which microcontroller is well suited to accomplish all these given controls.

The motor control system, under the command of the microcontroller, interprets the feedback signal from the pull-up/pull-down network sensors and directs the robot to rotate counterclockwise or clockwise accordingly. If the error is negative (i.e., the robot is misaligned

to the left of the reference), the motor control system rotates the robot clockwise to correct its orientation. Conversely, if the error is positive (i.e., the robot is misaligned to the right of the reference), the motor control system rotates the robot counterclockwise.

[DMK]

Figure 1 Control System State Space Logic



2.7.1 External User Control

The system will consist of control signals from a user-controlled external input device. This external user control will take in the desired movement direction and a variable speed control (0 - 100%). The user control will then be transmitted wirelessly to the platform identified

as the master platform. The master platform will take these signals and calculate each slave platform's controls for each motor's direction and speed. Additionally, the external controller will provide feedback to the user on whether it is paired to the master platform, whether the slave platforms are paired to the master platform and information on the initial alignment of the platforms.

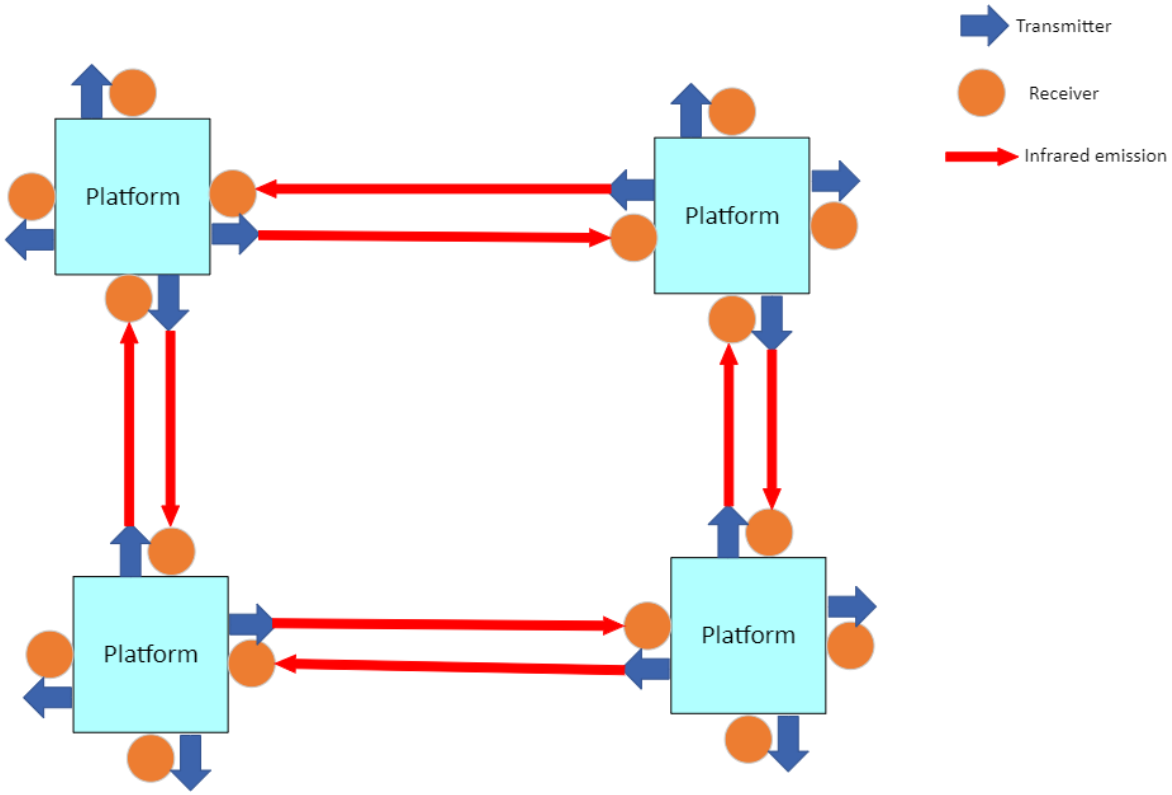
When designing this system, two user control systems were considered. First, an external, handheld controller (such as those used for remote control toy cars) could be utilized so long as it could receive variable X and Y direction commands from a user. The controller would then wirelessly transmit these signals to the master platform. This would be appealing as development time would be reduced, as one would only need to know the wireless signals transmitted by the pre-made controller.

A second control system considered was that of a mobile application (app) on a phone. The phone or other smart device would need to be paired with the master platform and would require development of an app, requiring further development time. However, this solution is appealing because Bluetooth or other communication protocols built into common smart devices could be utilized for wireless communication. Existing communication development stacks on platforms such as iOS or Android could be used in development. Additionally, this approach allows the team to customize the user interface and communication protocols (such as transmitted data) to meet the needs of this system. Due to these advantages, the design will develop a mobile companion application to take in and transmit user control. [ZB]

2.7.2 Alignment

A crucial aspect of the project is the platform alignment automation. This is a key aspect due to wheel movement. The movement and rotation of using mecanum wheels relies on each wheel being within a rectangular pattern comprised of the remaining platforms. The user should not need to adjust the robots every few feet, so an internal control system will make sure the robots are always in line. This will be achieved using infrared light emitters and receivers. [JS, DK].

Figure 2: Platform Alignment Illustration



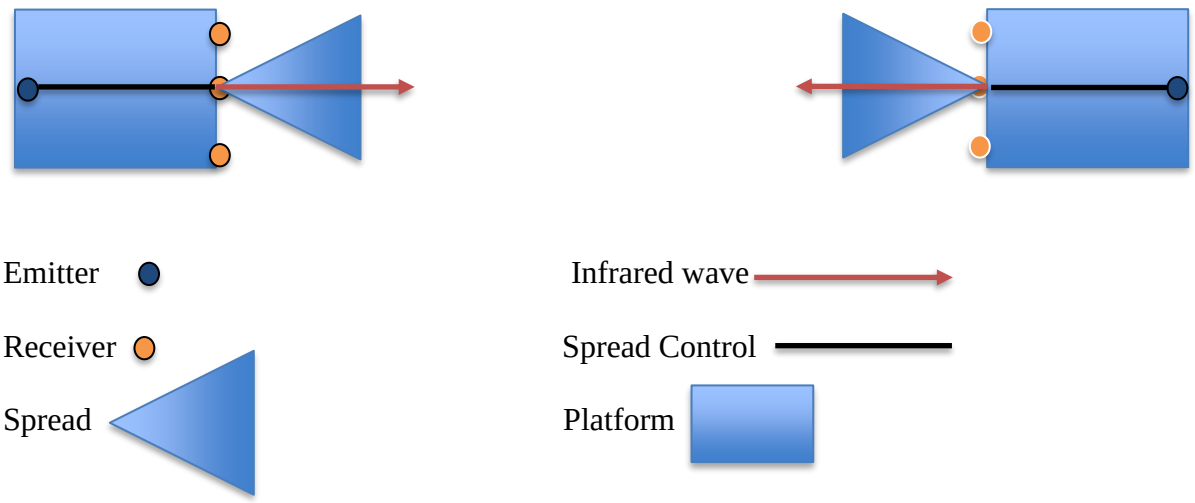
In Figure 2, it can be seen how each side of a platform will have one transmitter and receivers will be aligned along the outside of the platform. This design implementation will allow users to dynamically place each platform without having to worrying about aligning the correct sides of each platform to match a receiver and transmitter. Throughout user operation receiver voltage

levels will be read by an ESP32. The software running the alignment will have an open loop discrete time controller. A positive error will make the controller move the robot towards a negative position and negative error will make the robot move toward the positive. The range of voltages that will be used for the error detection can be seen in Table 4. The difficult part of these voltage level readings is controlling the emitters spread to prevent false alignment reading within receiver voltage readings. In order to control the emitter, spread the design in Figure 3 will be implemented to help control this emitter spread. [JS, DK]

Table 4: Alignment Error Correction

+3.3v	+1.65v	+0
Move negative	No adjust	Move positive

Figure 3: Emitter Design



Because the emitters have 32 degrees of spread, the emitters will be put in a drinking straw [Spread control] to direct the wave in a fiberoptic like manner. The Final spread will need to be less than 1.5 inches at 6 feet. Because the straw opening [6mm] is many times larger than the wavelength [1mm], diffusion will not need to be accounted for. This means the spread control required length can be estimated by the human eye.

[DK]

3. Engineering Requirements Specification

The following table of engineering requirements, Table 5, will establish the technical guidelines for this project along with justifications for each requirement. These requirements were created based on the marketing requirements in Table 1 and are intended to meet or exceed one or more of the established marketing requirements. [ZB]

Table 5: Engineering Requirements

Marketing Requirements	Engineering Requirements	Justification
4	1. The system will be able to withstand a maximum weight of 100 lbs. split between four platforms (25 lbs. per platform)	This weight limit is a reasonable design limitation due to the overall cost of higher weight-bearing parts.
4	2. The system will be able to support the furniture via one point of contact per platform	The project's design is intended to use multiple platforms in tandem supporting each corner of the furniture. Thus, the platform does not need to hold an entire piece of furniture.

1, 2	3. The system will implement directional control along the X & Y axis via an external device with an accuracy of ±2 cm	The external device communication allows for easy motor directional control by any user.
1, 2	4. The system will be able to move 360 degrees with an accuracy of ±10 degree via controlling x and y axis of each platform	Allowing the system to move accurately through 360 degrees motion provides easier rotation of furniture.
3	5. Each platform will operate from an onboard rechargeable energy storage device with a maximum power draw of 150 W for a minimum operating time of 45 minutes	With the amount of time it takes to adjust the position of furniture, this operating time seems reasonable to implement for the design.
5	6. The system will visually indicate the energy storage device's state of charge for each platform	Giving the user knowledge of when the system's batteries are running low allows the user to know when to halt operation and begin the recharge process.
1	7. The system must be able to operate under a coefficient of friction ranging between 0.5 and 1.0 (Thermoplastic Polyurethane on various carpeting and hardwood floors)	For the system to be able to move the given furniture, it must be able to overcome the weight of what is being moved through a high frictional surface.
1, 2	8. The system must operate at a variable speed with the max speed being 0.5 m/s	This is a reasonable maximum speed to prevent any harm and damage to the furniture or users.
1	9. Each platform will maintain a constant alignment with two other platforms with an accuracy of ±2 cm	This range of accuracy provides a margin of error for small delays in inter-platform communications amongst platform movement or insignificant differences in wheel movement.
1, 2	10. Each platform will communicate	Having each platform

	with each other via wireless communications with a minimum range of 5 meters	communicate with one another will allow the overall system to monitor each platform's alignment and motor speed.
Marketing Requirements 1. The system should be able to remotely move a given piece of furniture 2. The system should be controlled by an external device 3. The system should have a portable, rechargeable energy storage device 4. The system should be compatible with various types of furniture 5. The system should sense and alert when the power source is low/needs replaced		

[ZB, GVM, JS]

4. Engineering Standards Specification

This section will go into the specified safety and standards that will be used throughout this project. To ensure that the project adheres to industry standards and safety measures, engineering standards have been considered in the design of this system around the topics of safety, communication, data formats, design methods, and programming languages. [ZB, JS]

4.1 Safety

When working with any electrical system, it is always necessary to take the proper safety precautions in any given scenario or situation. For this system specifically, there are various subsystems that require a close eye during the development stages. The specific subsystems that need attention when designing/developing are the power/battery subsystem along with the motor subsystem design and the control subsystem. This portion of the report will cover the safety standards that are required for the given design project. [GVM]

Specific safety features have been implemented into this project's design. Regarding safety regulations with the power subsystem of the project, there are a few precautions to take when working with a battery-powered system. Specifically, having protection for overcurrent and overvoltage for the batteries in use is necessary to minimize any concerns about overcharging the battery. Another safety aspect taken into consideration for this project design is motor movement. The motor has been limited to 0.5 m/s to protect the user and furniture from any harm while operating the system. This variable system speed will allow the user to control the movement of furniture slowly and safely without fear of jolting movement that could cause a tip-over or collision with another object. [GVM, JS]

Additionally, considerations for safety in the platform's control systems have been included. The software on each platform's microcontroller will be designed to default to a halt

state, meaning that all motors on that platform will cease movement. There will be a watchdog added to ensure that firmware crashes are detected, and that the microcontroller is consequently rebooted. On boot (and after a watchdog reboot), the microcontroller will reset its motor signals to a braking state and to a speed of zero. This design is intended to both reduce potential damage to the platforms and lessen the risk of user injury and/or room damage when operating the system with an inactive or physically-removed platform. [ZB]

The University of Akron Engineering standards of safety will also be followed throughout the entire design process. Alcoholic beverages of any kind will not be allowed in the Senior Design facility. All injuries are required to be reported to the Senior Design Coordinator (SDC). Proper Personal Protection Equipment (PPE) will be worn when any chance of harm may occur. Broken tools or equipment will be reported to the SDC in a timely manner. All circuitries will be approved by the Faculty Advisor (FA) or SDC before fusing or connecting the main power supply. Voltages greater than 48 volts will not be energized until the circuit is approved by the FA. These rules will be followed when designing any subsystem of the furniture mover. [JS]

4.2 Communication

The design will use Bluetooth for communications between the external user controller and the master platform as well as between the master and slave platforms. Bluetooth allows for two operating modes–classic and low energy. This system will use the Bluetooth standard's low energy mode. Using the Bluetooth protocol allows for use of many pre-existing libraries. Further details on Bluetooth can be found in section 2.6.6.

The system will also use UART for local communication within a local platform between microcontrollers. For further details, see section 2.4. [ZB]

4.3 Data Formats

The Espressif Systems ESP32-C3 microcontroller is a 32-bit general-purpose microcontroller. To that end, up to 32-bit data types can be used. It uses the RISC-V instruction set and a modified C-based language. This reduced instruction set is formatted with one of six instruction formats: register, immediate, upper immediate, store, brand, or jump. [ZB]

4.4 Design Methods

The process of designing and creating this project involved several phases, starting with defining a problem statement. Once a problem statement was defined, this led to the creation of both an objective and a need statement for the project (see sections 1.0-1.2 above). Research for similar technologies was then conducted to understand the solutions already out in the market today. Once knowledge of the existing technologies was completed the marketing requirement phase for the project began. After consideration of the marketing requirements was completed, then began the brainstorming phase for the project design. [GVM, JS]

An initial project idea was then formed to meet the project needs. Through further research of academic articles, mathematical calculations, and consultation with departmental professors, the specifications of the project were honed using n-level block diagrams. This was to specify inputs and outputs needed from the system. Engineering analysis was then performed on each identified subsystem to specify technical requirements and hardware/software requirements. [ZB]

4.5 Programming Languages

4.5.1 Swift

Swift is a programming language released in 2015 and built for general-purpose development on Apple platforms (iOS, iPadOS, macOS, etc.). Because it was intended to be a

replacement for the C language, the language was built to be closely comparable to C in terms of performance. It also includes many safety features including checking arrays for overflow and ensuring that variables are initialized before being used. This programming language will be used to develop the iOS application to provide external user input for the system (Apple, n.d.). [ZB]

4.5.2 C

Another programming aspect behind this system is C programming. C is also known as a procedural language which essentially means that all operations have to be laid out step-by-step. According to *GeeksforGeeks,* C programming not only allows for the direct manipulation of computer hardware but also allows for fast and efficient performance rates. C programming is not only utilized by embedded systems. Operating systems like iOS, Android, and Windows utilize C applications. In this project, C will not only be utilized within the embedded system—C will be used to communicate between the embedded system and the external user controller (GeeksforGeeks, 2021).

5. Accepted Technical Design

5.1 Hardware Design

5.1.1 Level 0 Hardware Block Diagram

Figure 4 below illustrates this system's Level 0 Hardware Block Diagram. This system diagram takes an input of DC power, input from alignment sensors, and speed/direction input from the user. The system then outputs movement in the X and Y directions. [ZB]

Figure 4: Level 0 Hardware Block Diagram

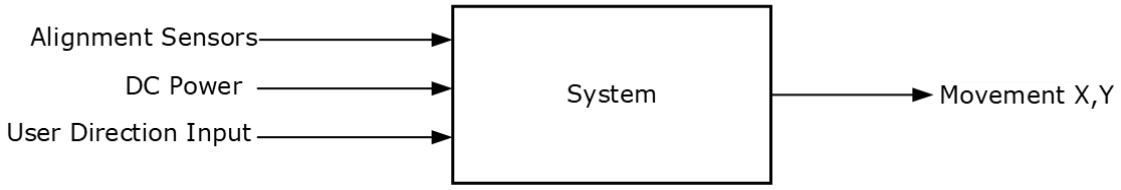


Table 6: Level 0 System Function Requirements

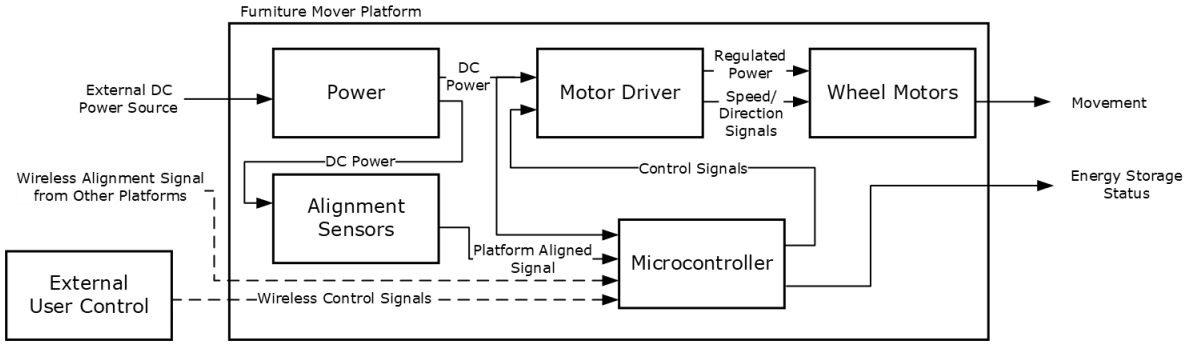
Module	System
Designers	Zach Burkhardt, Juan Soto
Inputs	Alignment sensors: binary input of sensor alignment Power: DC User direction input: variable direction and speed control
Outputs	X,Y Movement: Wheel Motors
Description	The microcontroller takes in various input signals and calculates output signals. The alignment sensors will have binary input. The control signal for motors will have input from variable user direction input.

5.1.2 Level 1 Hardware Block Diagram

Figure 5 below shows the Level 1 Hardware Block Diagram for the user-controlled furniture mover system. The following Figure 5 goes over the examination of Figure 4 from above into more depth of the overall system inputs, subsystems, and outputs.

As it can be seen the three initial inputs have been broken down into more detail showing how the external user control and alignment sensor signals will be read as an input to the microcontroller, along with the power as an additional input. The output for the microcontroller is then directed to the input of the wheel motor to control the X & Y movement of the wheels, it can also be seen how power is being supplied to the wheel motors as an input. Finally, the expected output for the system should be motor movement and battery level status. [JS, GVM]

Figure 5: Level 1 Hardware Block Diagram



The tables below detail each subsystem of the Level 1 Hardware Block Diagram with each subsystem's inputs and outputs. Table 7 details the inputs and outputs of the power module, which regulates power input voltage in order to supply proper voltage levels to all other subsystems of the overall system. Table 8 describes the alignment sensor signals which are transmitted as input to the microcontroller to help coordinate the system to the proper orientation configuration. Table 9 details the microcontroller's inputs from external user control, power, and alignment sensors and uses this information to control the wheel motors to determine the desired user direction. Table 10 shows the movement subsystem inputs/outputs for the user-specified X and Y movement. [ZB]

Table 7: Level 1 Power Module Function Requirements

Module	Power
Designers	Gino Mucciarone
Inputs	External DC power source: DC voltage
Outputs	Power, DC: Power distributed to various subsystems
Description	The module will accept DC power from the external power supply, which will allow the battery to recharge. The output of the power module will then be converted to DC regulated voltage levels to distribute to the various subsystems.

Table 8: Level 1 Alignment Sensor Module Function Requirements

Module	Alignment Sensor
Designers	Zach Burkhardt, Juan Soto
Inputs	Alignment signal: Received signals from other platforms receivers. Power: DC User Direction Input
Outputs	Platform Aligned Signal: Indicates whether the sensor is aligned with the transmitter
Description	The alignment sensor will both transmit and receive signal information. The transmission will be of an infrared ray beam and the receiver signal will determine if the alignment of the transmission beam has been detected.

Table 9: Level 1 Microcontroller Module Function Requirements

Module	Microcontroller
Designers	Zach Burkhardt, Juan Soto
Inputs	Power: DC Wireless Control Signals: Wireless input signals from external user controller Platform Aligned Signal: Received signals from alignment sensors
Outputs	Control Signals: To be sent to the wheel motors to produce movement in the X and Y directions Energy Storage Status: Gives the user a reading on the battery percentage and alerts the user when batteries require a recharge.
Description	The microcontroller takes in various input signals and calculates appropriate output signals. The output signal will determine the action for each motor controller and the indication for the energy storage status.

Table 10: Level 1 Wheel Motor Module Function Requirements

Module	Wheel Motors
Designers	Gino Mucciarone
Inputs	Power: DC Control Signals: Direction and speed signals from microcontroller
Outputs	Movement: Control motor movement for moving the platform
Description	The wheel motors take in power and signals to control the motor's movement. The output of this module moved each motor in a direction at a set speed.

5.1.3 Level 2 Hardware Block Diagrams

Figure 6 below shows the level 2 hardware block diagram for the power subsystem. The power subsystem is split into 3 modules, which include the battery charger, the batteries, and the power output module. The battery charger module is responsible for taking the external DC power source to recharge the battery when low on power. This then sends DC voltage to the batteries, and then into the power output module, which will then convert the voltage from the batteries into the different voltage levels required to power all other subsystems involved in the design. Tables 11, 12, and 13 describe the requirements for the battery charger, battery, and power output modules, in that order. [GVM, JS]

Figure 6: Level 2 Power Module Block Diagram

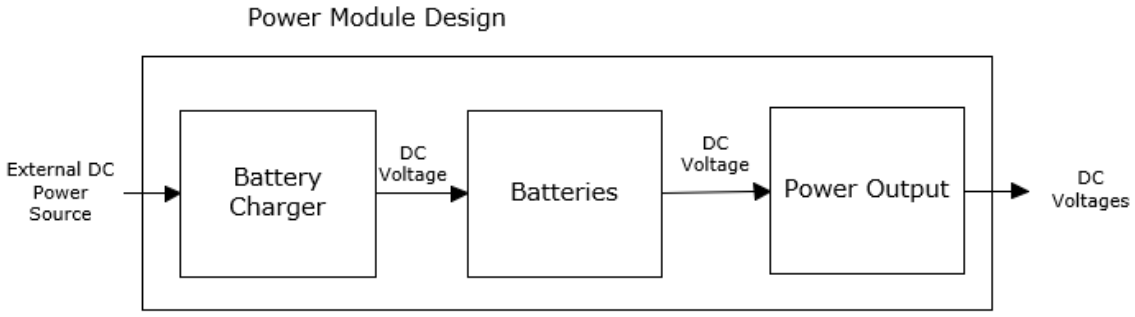


Table 11: Level 2 Battery Charger Module Function Requirements

Module	Battery Charger
Designers	Gino Mucciarone
Inputs	External DC Power Source
Outputs	DC Voltage:
Description	Will allow for the system's batteries to be recharged when necessary, this will be indicated to the user when required to do so.

Table 12: Level 2 Battery Module Function Requirements

Module	Battery
Designers	Gino Mucciarone
Inputs	Battery Charger: DC voltage supplied
Outputs	DC Power: 14.8V DC
Description	Will be the main power source for the system and allow other subsystems to run as wanted.

Table 13: Level 2 Power Output Module Function Requirements

Module	Power Output
Designers	Gino Mucciarone
Inputs	DC Voltage: 14.8V DC
Outputs	DC Voltages: Various voltage levels for powering all subsystems (12V, 5V, 3.3V)
Description	The Power output module will output a various set of voltages in order to supply each subsystem of the design.

Figure 7 shows the level 2 hardware block diagram for the wheel motor subsystem. This block diagram specifies the DC motor being driven by the motor controller which is supplying a DC voltage. The motor controller is then supplied with a DC voltage and control signals sent from the microcontroller. [GVM]

Figure 7 : Level 2 Wheel Motor Module Block Diagram

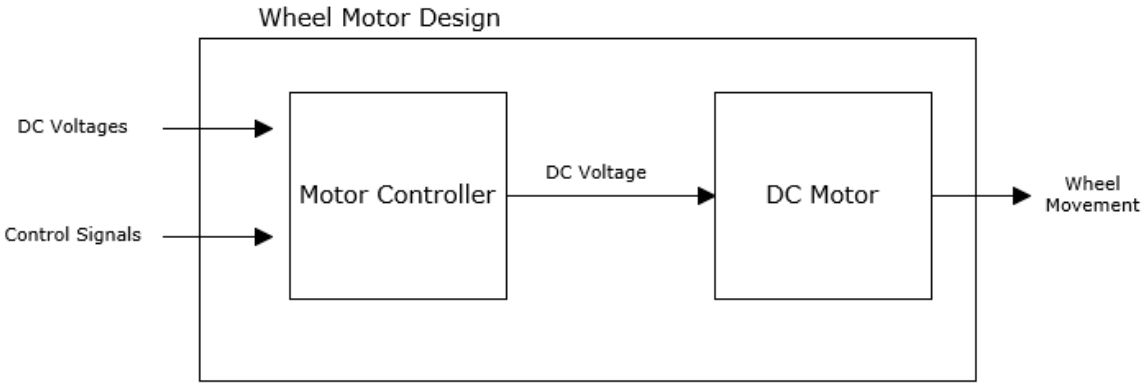


Table 14 describes all the functional requirements for the motor controller to implement to each of the wheel motors. Each motor controller must be capable of driving the motor it is

correlated with. These motors need to be able to move each platform of the system in the X and Y directions when control signals are received to do so. Further requirements for the DC motor module can be seen in Table 15 below. [GVM]

Table 14: Motor Controller Module Function Requirements

Module	Motor Controller
Designers	Gino Mucciarone
Inputs	DC Voltage: The motor controller will take in a DC regulated voltage from the Power output module. Control Signals: Signals from the microcontroller module will be taken in to determine motor movement.
Outputs	DC Voltage: A positive/negative voltage will be output to determine direction of the DC motor
Description	The Motor controller will interpret control signals received from the microcontroller to determine the appropriate positive or negative voltage output to determine the direction of the DC motor.

Table 15: DC Motor Module Function Requirements

Module	DC Motor
Designers	Gino Mucciarone
Inputs	DC Voltage: The DC motor will take in a positive or negative DC voltage to determine motor direction.
Outputs	Wheel Movement: X and Y directional movement
Description	Each motor spins at the speed and direction dependent on the user's choice.

5.1.4 Level 3 Hardware Block Diagrams

Figure 8 below shows the Level 3 hardware block diagram for the power module of this project. This block diagram goes into slightly more detail through the necessary components

used to power each of the platforms. Tables 16, 17, 18 and 19 are shown to describe each of the components that are included in the level 3 hardware block diagram. [GVM]

Figure 8: Level 3 Power Module Block Diagram

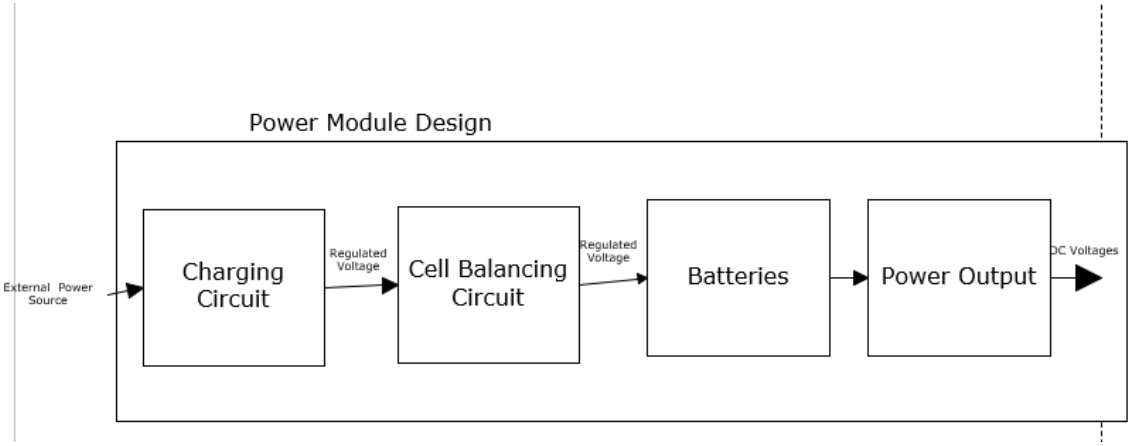


Table 16: Level 3 Charging Circuit Module Function Requirements

Module	Charging Circuit
Designers	Gino Mucciarone
Inputs	External Power Source
Outputs	Regulated Voltage
Description	This module is responsible for creating the regulated voltage that will be sent through the battery management system, which will be 14.8V DC.

Table 17: Level 3 Cell Balancing Circuit Function Requirements

Module	Battery Management System
Designers	Gino Mucciarone
Inputs	Regulated Voltage
Outputs	Safe Charging to Batteries
Description	Will allow for safer charging to the battery pack being used to power each of the platform, allowing for longer battery life overall

	and better safety to the user.

Table 18: Level 3 Battery Module Function Requirements

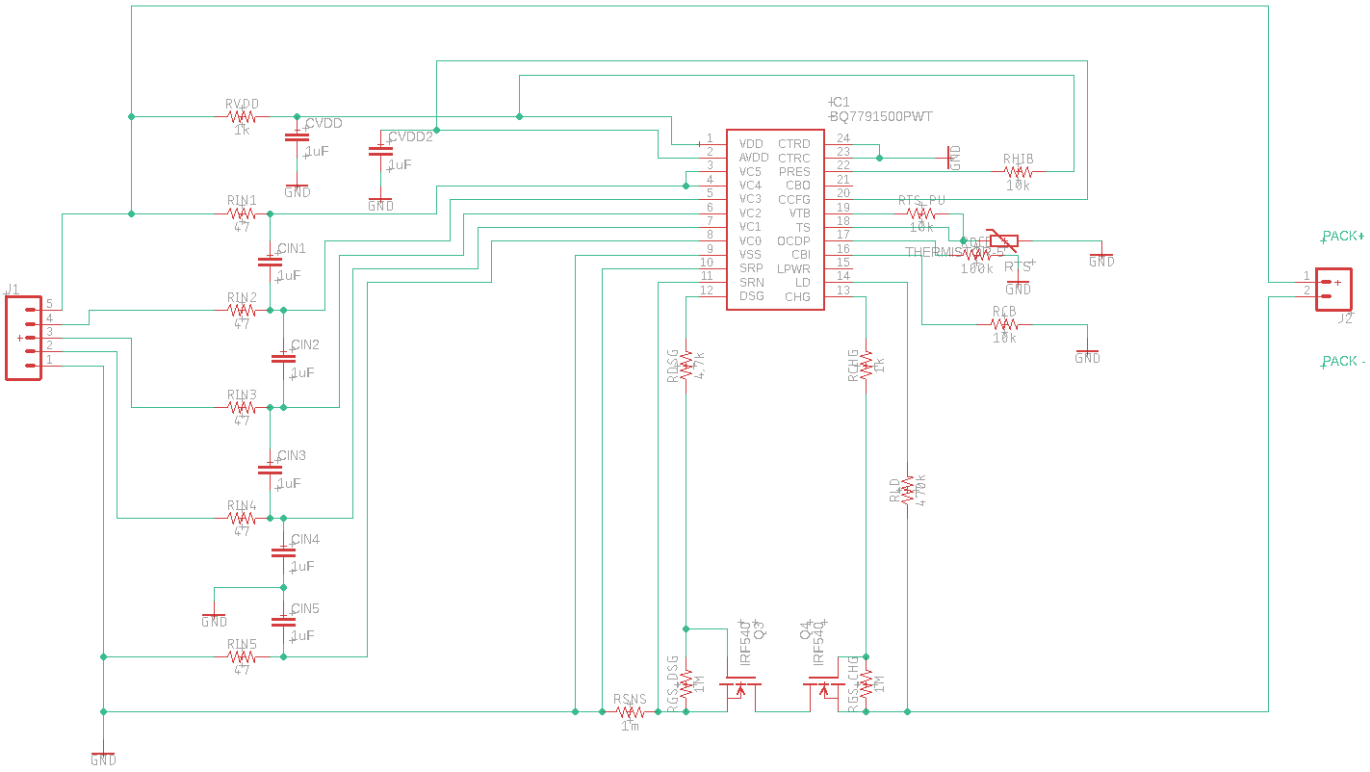
Module	Battery
Designers	Gino Mucciarone
Inputs	Battery Management System Safe Charging
Outputs	DC Power: 14.8V DC
Description	Will be the main power source for the system and allow other subsystems to run as wanted.

Table 19: Level 3 Power Output Module Function Requirements

Module	Power Output
Designers	Gino Mucciarone
Inputs	DC Voltage: 14.8V DC
Outputs	DC Voltages: Various voltage levels for powering all subsystems (12V, 5V, 3.3V)
Description	The Power output module will output a various set of voltages in order to supply each subsystem of the design.

The following figures shown below are schematics design using EAGLE drawings, these include schematics of charging circuits, battery management systems, and motor driver. Details of each schematic are discussed below each of their images as shown in the pages below. [GVM]

Figure 9: Battery Management Schematic



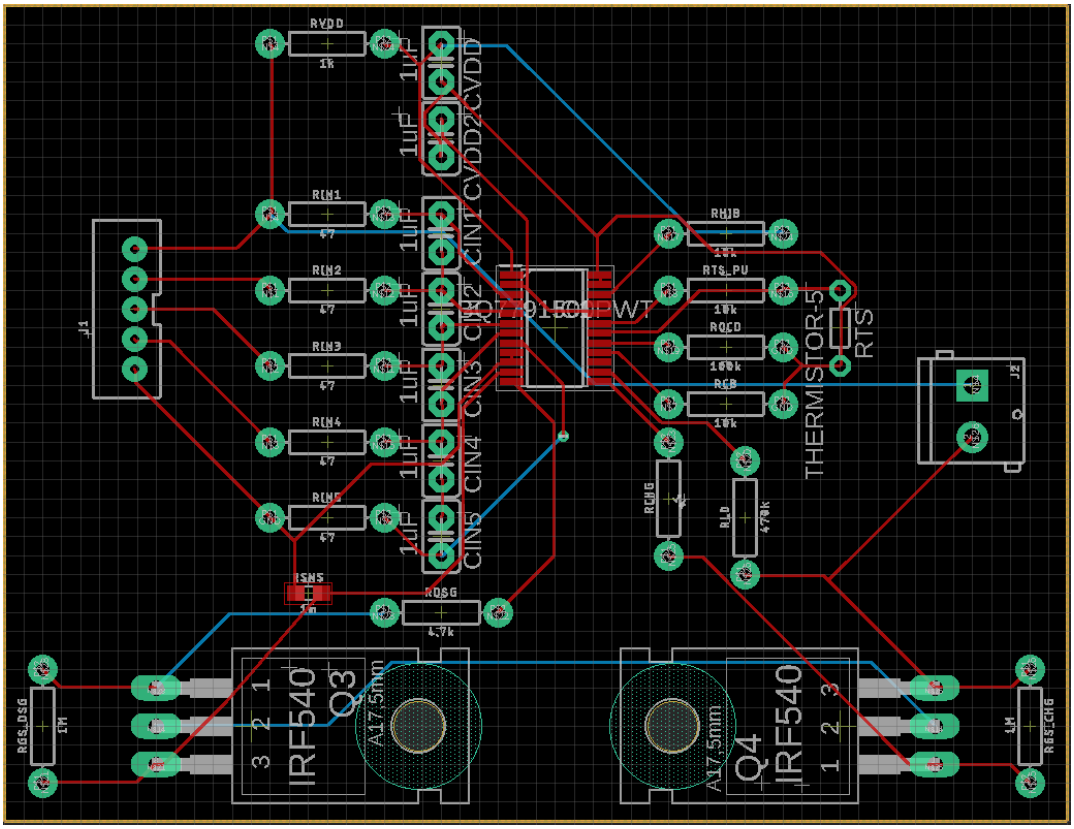
In figure 9 above is the schematic of each platforms battery management system. As previously discussed in section 2.2.1, the IC used for this circuit was the BQ7791500PWT, which provided many different protection features in order to use the batteries safely in the system. Going into more detail about the schematic itself, including the protection features and the adjacent cell balancing algorithm programmed into the BQ7791500PWT, there are also two N-channel MOSFET drives in the chip, which controls charging and discharging, labeled as CHG and DSG on the device pins.

Moving on to talk about how some of these protection features work in the circuit, starting with the over/undervoltage detections. Labeled in the schematic as RIN“X” and CIN“X,” the BQ7791500PWT reads each cells voltage and compares it with the device threshold. The overvoltage threshold is set to 4.2V per cell and the undervoltage threshold is set to 2.9V per

cell. In the event that a cell reads 4.2V, the charging MOSFET driver brings the CHG MOSFET to ground and stops the battery pack from charging the batteries any further. Similarly, for an undervoltage case and a cell is read at 2.9V, the discharging MOSFET driver pulls the entire pack to ground and does not allow the pack to discharge any further.

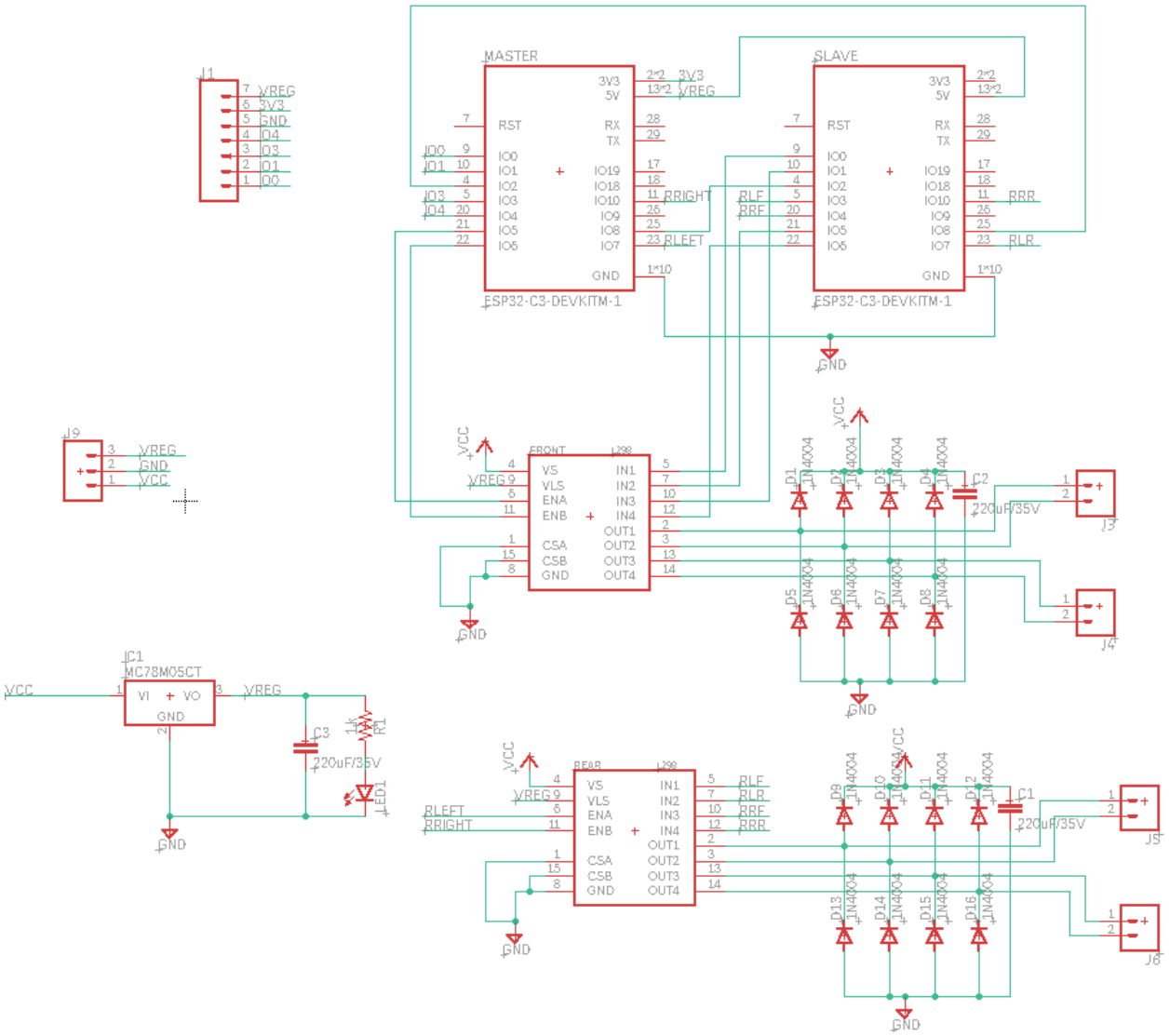
Discussing the current protection features now, which include short circuit discharge and overcurrent discharge. These features are controlled through the SRP and SRN pins of the BQ7791500PWT, which are placed across the current sense resistor (labeled RSNS in schematic). Depending on the value picked for the resistor, you can select a current threshold so that if it is ever hit, the battery pack will stop discharging completely. There are also temperature protection features held within the BQ7791500PWT. These features are controlled by the thermistor in the schematic (labeled as RTS). Below will be the layout of the PCB that was implemented into the project. [GVM]

Figure 10: Battery Management System Layout.



The board layout of the battery management system can be seen in the figure above, all values used for resistors and capacitors were selected based off the BQ7791500PWT datasheet recommended values. The selection of the MOSFETS to use came with the requirements for it to have a 20V Vgs with a specific RDSON at 10V. It was recommended to keep all RC filters close to the device pins to get as accurate readings as possible. [GVM]

Figure 11: Motor Driver/Esp32 Schematic



Motor driver circuit is a simple power stabilizing design. Capacitors are used to ground any alternating current. Diodes are used to keep the maximum voltage from spiking past supply

voltage. A 5-volt regulator is built on the chip to control the logic/data. For example, A simple jumper connects the 5-volt regulator to turn on the enable switch if a PWM wave in not inputted. [DK]

The circuit also includes header pins for two ESP32 microcontrollers. The GPIO pin outputs for PWM signals and direction signals connect to the two motor driver chips on the board. Additionally, the four alignment sensor inputs are routed to external pins to connect to the controls system circuit. Finally, the two microcontrollers have traces between the two to accommodate the local serial communication between microcontrollers. [ZB]

Figure 12: Motor Driver/Esp32 Layout

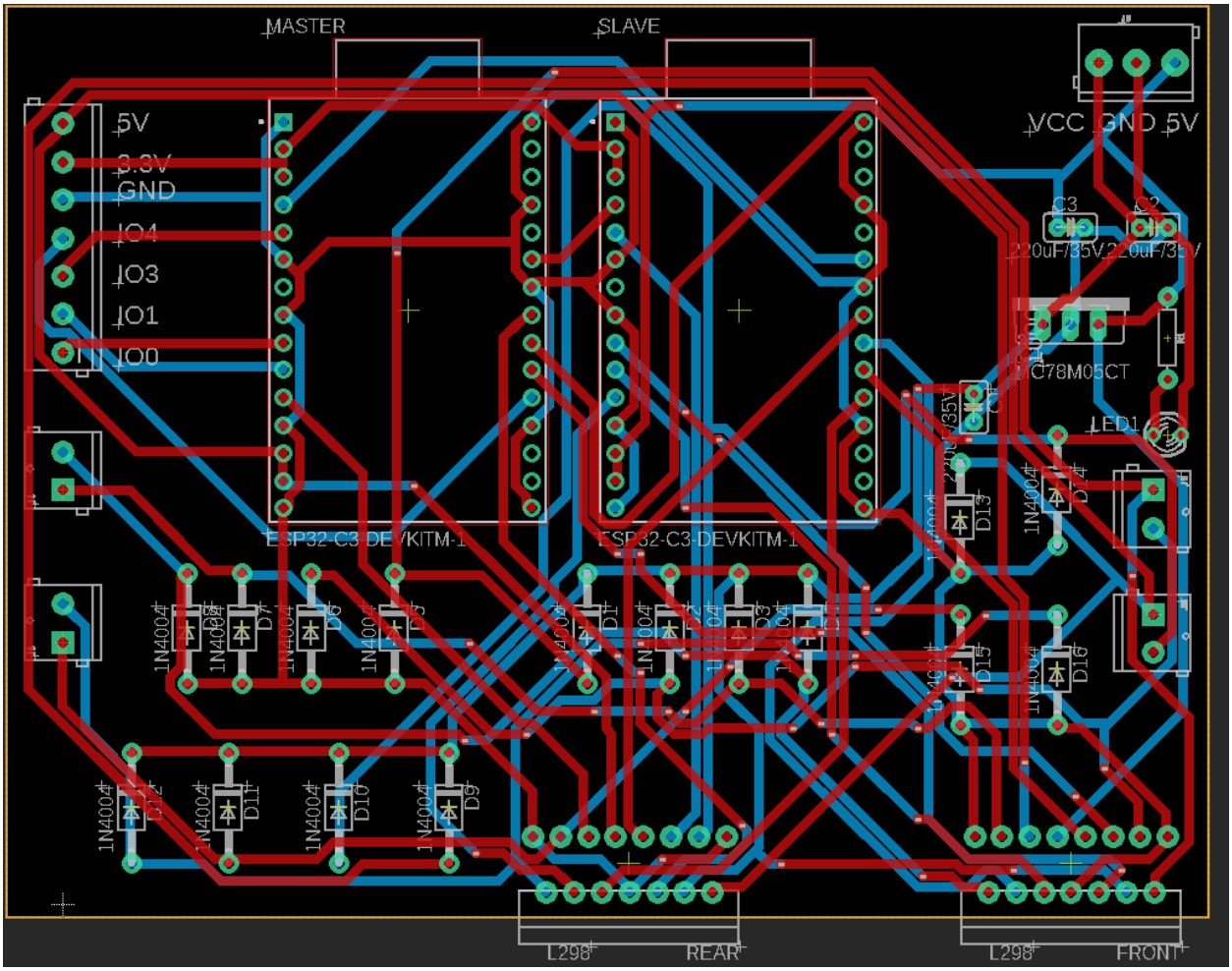
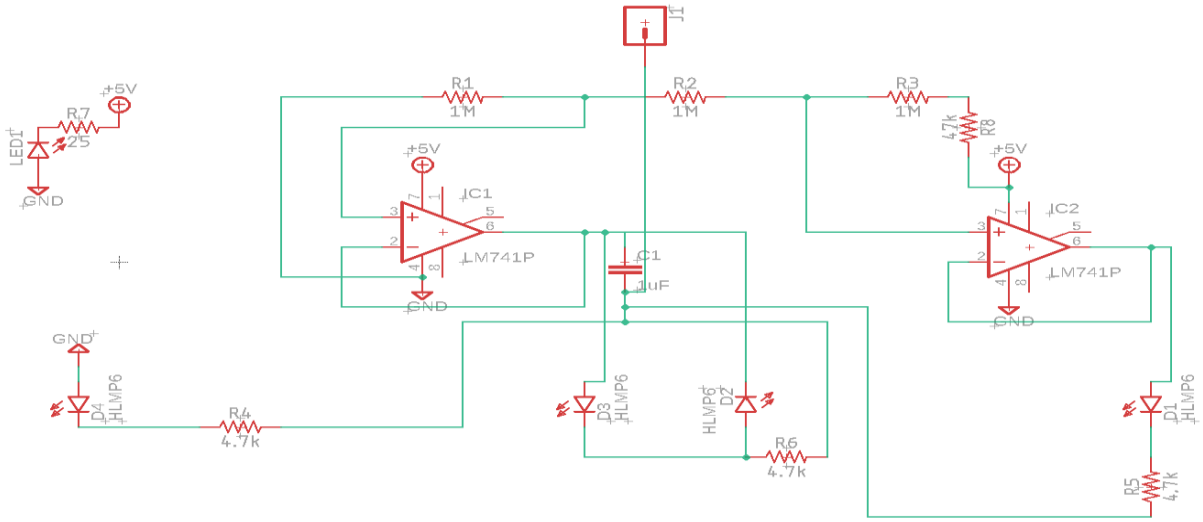


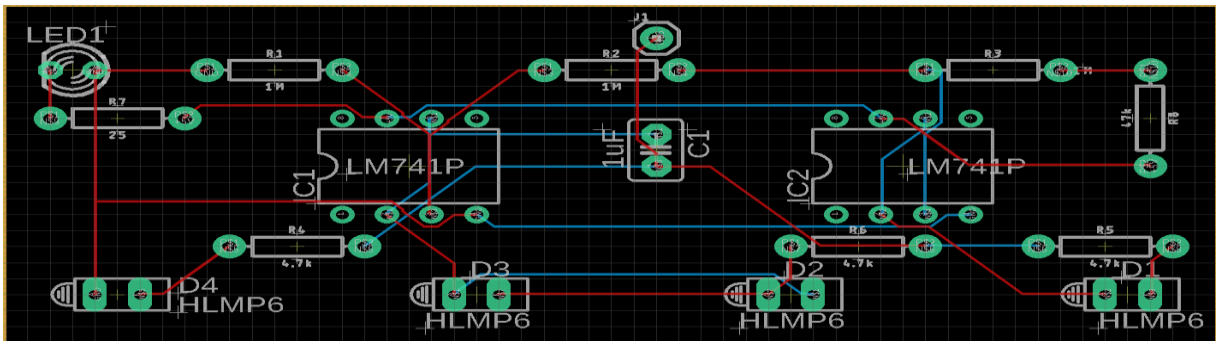
Figure 13: Alignment Sensor Schematic



The control system utilizes an innovative sensor arrangement consisting of four infrared PNP receivers, designed to work collaboratively in managing the output voltage levels. When the right PNP receivers is activated, it pulls the voltage high, while the left receiver is responsible for pulling the voltage low. The remaining two PNP receivers in the center function to maintain a middle voltage level when activated.

To preserve the sensor's final value for a short duration even when the sensor is not actively receiving signals, a capacitor is integrated into the system. This capacitor serves as a temporary storage element, allowing the control system to retain the last known value. [DMK]

Figure 14: Alignment Sensor Layout



5.2 Software Design

5.2.1 Level 0 Software Behavior Models

Figure 15 below illustrates the system's level 0 software behavior model. The software behavior in the following Figure 15 first starts off with a start phase. Next, an inactive or idle time is then activated to halt operations for a certain amount of time. Once the idle time has passed, the operation phase then begins. In the operation phase user input, alignment, and motor control is taken care of and executed. Next, a kill request is then processed where the software can either return to the inactive/idle stage or terminate completely. [JS, ZB]

Figure 15: Level 0 Software Behavior

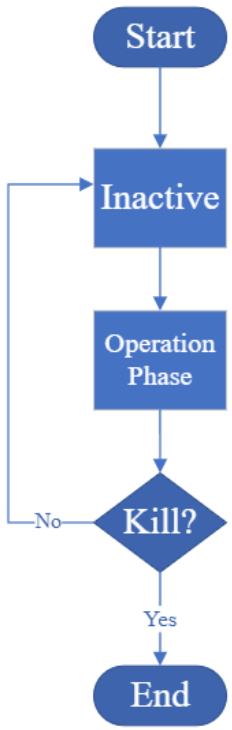


Table 20: Level 0 Software Behavior Functional Requirement Table

Module	Software Behavior
Designers	Juan Soto
Inputs	External User Control: Take in user speed and movement input from external device. Platform Communication: The master platform will establish multiple connections with slave platforms. Alignment Sensor: Proper alignment signals will be processed by the master platform to determine correct platform orientation.
Outputs	Wheel movement: Directions in the X & Y coordinates Alignment: Output proper alignment requirement message and error message. Energy storage level: An indication of the platforms remaining energy will be displayed to the user.
Description	The software will maintain the proper inputs from both user and sensors to determine the appropriate action to take when it comes to any furniture mover platforms.

5.2.2 Level 1 Software Behavior Models

As shown in Figure 16, when starting the software, the system will initially prompt the user to pair the external user controller to a master platform. Once paired, the master platform will check if it is paired with the slave platforms. If not, the user will be prompted on the external controller to pair the platforms. Then, the software will check the alignment sensor signal to check if all platform alignment sensor receivers report being aligned to a transmitter sensor. If one or more alignment sensors report issues, the software will prompt the user to fix the alignment.

Finally, the system enters an idle state while waiting for movement controls from the external user input. The system will idle for 30 ms if movement signals are not received. Once a control signal is received from the external input, the master platform calculates each motor’s

directions and speed and wirelessly sends the control signals to each slave platform to control the motors. The software will then verify that the master platform has not disconnected from the slave platforms (due to connectivity issues, platform energy source depletion, etc.). If it depletes, the user will be prompted to re-pair the device. Otherwise, it will continue to wait for a new control signal.

The software on each platform's microcontroller will be designed to default to a halt state, meaning that all motors on that platform will cease movement. There will be a watchdog added to ensure that firmware crashes are detected, and that the microcontroller is consequently rebooted. On boot (and after a watchdog reboot), the microcontroller will reset to its motor signals to brake and to a speed of zero. This software design ensures the safety of users and furniture should there be an unexpected malfunction. [ZB]

Figure 16: Level 1 Software Behavior

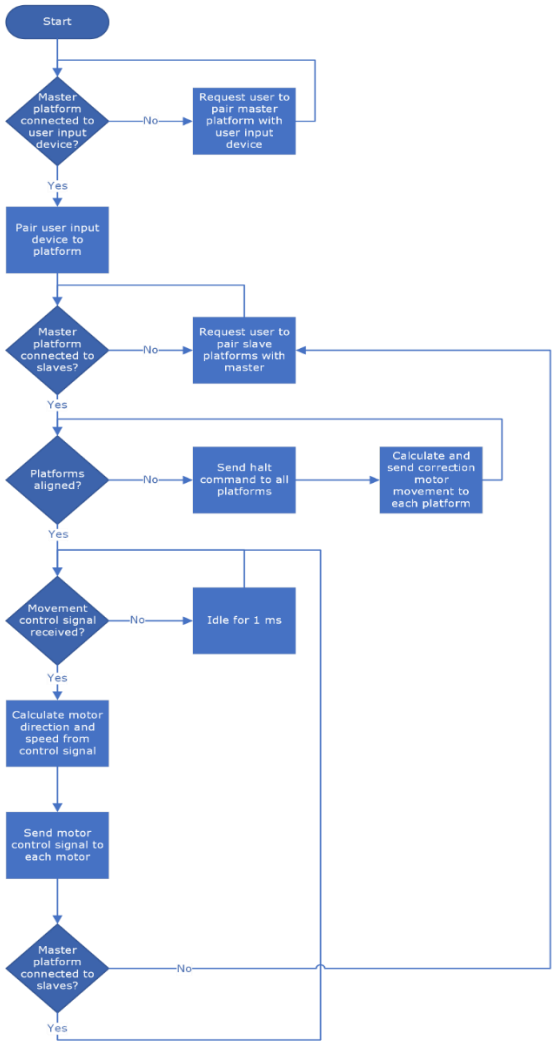


Table 21: Level 1 Software Behavior Functional Requirement Table

Module	Microcontroller/External User Input
Designers	Zach Burkhardt
Inputs	External User Control: Take in user input of speed and movement from external device. Platform Communication: The master platform will establish multiple connections with slave platforms. Alignment Sensor: Proper alignments signals will be processed by the master platform to determine correct platform orientation.
Outputs	Wheel movement: Motor direction and speed signals Alignment: Output proper alignment requirement message and error

	message. Energy storage level: An indication of the platforms remaining energy will be displayed to the user.
Description	The software will take input from the external user controller, alignment sensor, and other platforms to determine system status and movement. The software will send motor control signals based on its calculated direction.

5.2.3 Level 2 Bluetooth/Embedded System Software Behavior

As shown in Figures 17 and 18, the software for the embedded systems and communications between platforms and the user input device have been broken up into two modes: initial pairing and the main operation mode. Figure 17 describes the initial start-up behavior. The microcontroller's Bluetooth module enters pairing mode and checks for a new connection. If the new device is not a phone, the device is paired to another platform and is assumed to be a slave; an internal flag is set on the microcontroller for this identity. However, if the newly connected device is a phone, the platform is presumed to be the master—as the master platform ingests data from the user input device and slave platforms—and marks an internal flag to indicate this identity. The device then re-enters pairing mode to allow the slave platforms to connect to itself. Once the user input device indicates to exit pairing mode and start main operation, the platform enters the behavior indicated in Figure 18.

For slave platforms in the main operation mode, as depicted in Figure 18, the software reads in each of the four alignment sensor values and transmits these values to the master platform. If a movement command from the master platform is not received at this point, the software idles for 1 ms to reduce wasted power on communication checks. When a movement command is received, the platform calculates the direction and power to move each motor and

generates the corresponding PWM waves. The platform verifies it is still connected to the master platform; if it is disconnected, the platform halts all motors and attempts to reconnect.

For master platforms in the main operation mode, the platform first checks that all slaves are still connected. If any slave has disconnected, the master will transmit this state to the user input device along with the respective platform ID that has disconnected to notify the user. If all slaves are connected, the master platform checks in 1 ms intervals for new data from the user input device or slave platforms. If the received data is from the input device, the master platform calculates each platform's movement commands and transmits to each respective slave platform. If the received data is from a slave platform, the alignment values are processed. If the platform requires correction, the master platform calculates the correction movements and transmits these movements to each platform. The master platform checks for updated alignment values from the unaligned platform(s) and continues to send corrections until alignment is restored. If corrections are no longer required or if corrections were never required, the master again checks the slave pairing status and for newly received data.

The code in Appendix A was written for the proof-of-concept demonstration. This code was intended for use with an Explorer 16/32 microcontroller but still represents a scaled-down representation of what the final code for the ESP32 will be. [ZB]

Figure 17: Level 2 Embedded System Software Behavior (Part 1 – Initial Pairing)

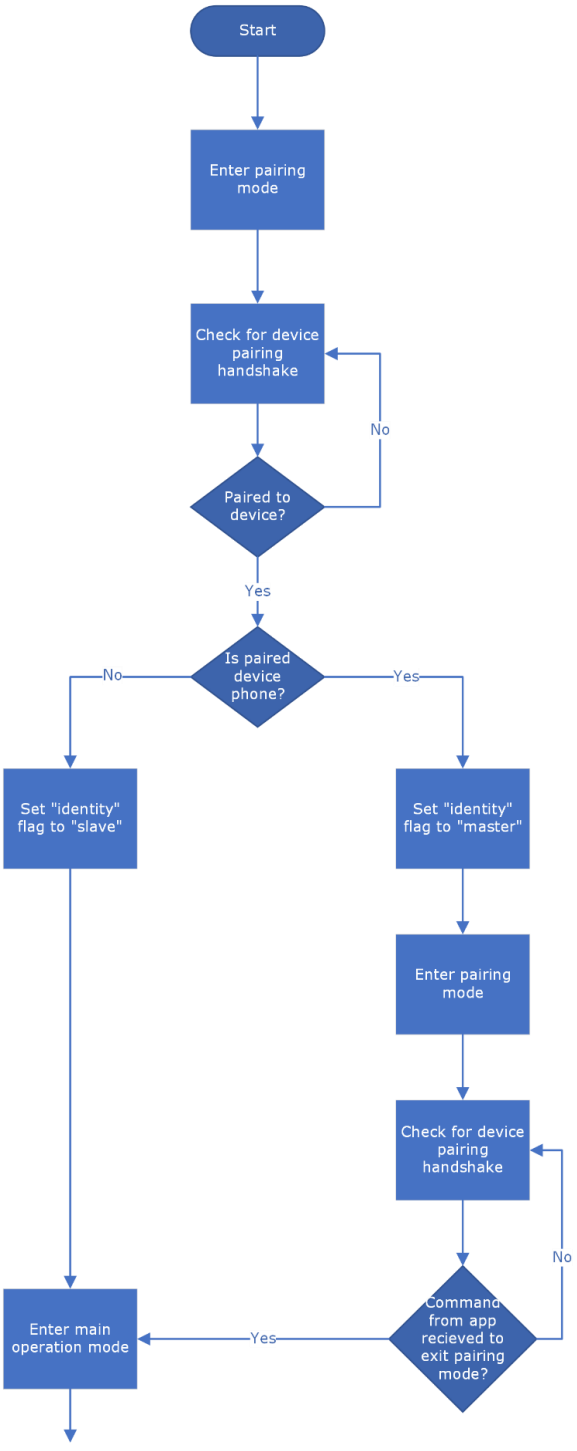


Figure 18: Level 2 Embedded System Software Behavior (Part 2 – Main Operation)

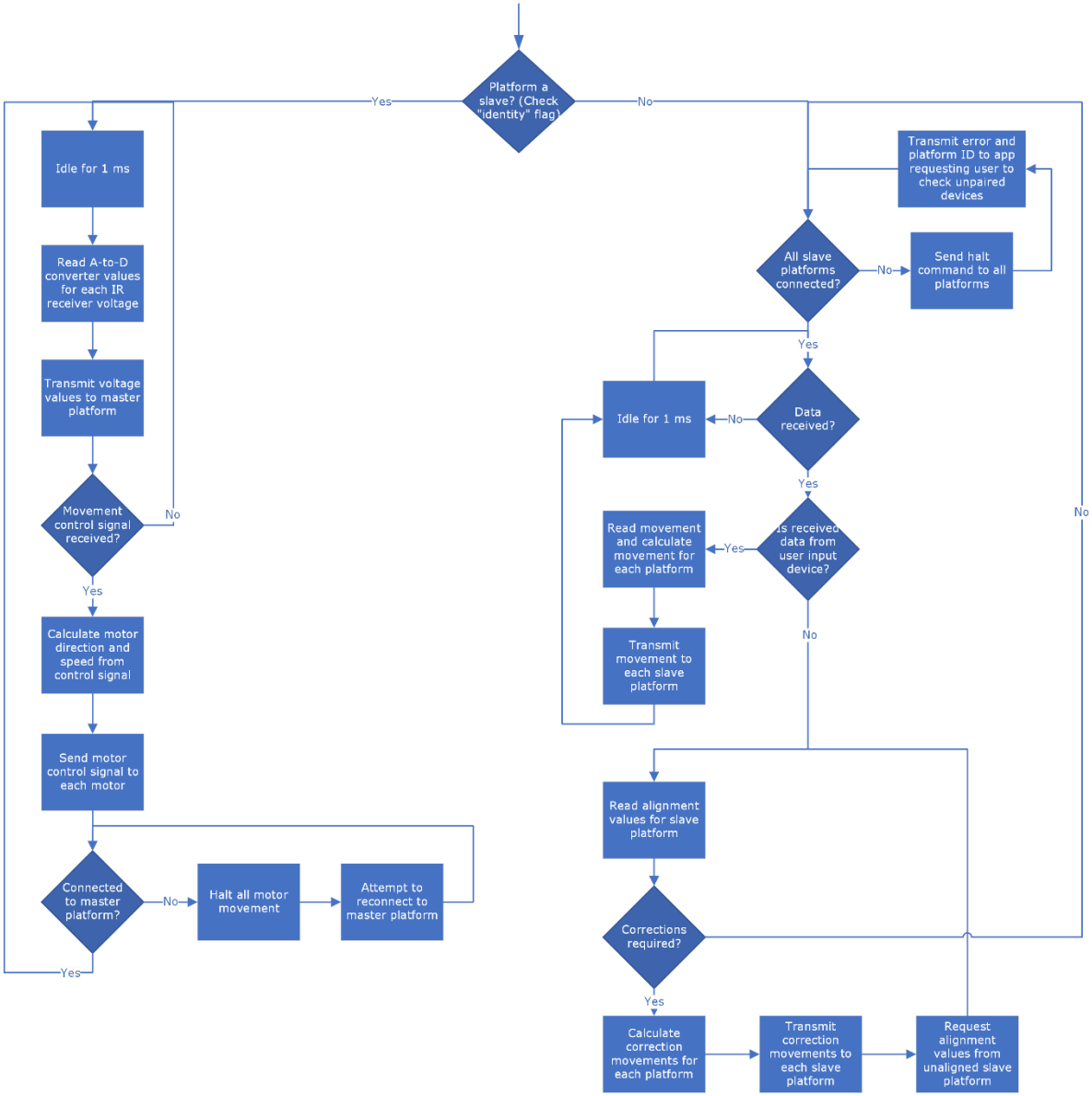


Table 22: Level 2 Embedded System Behavior Functional Requirement Table

Module	Bluetooth/Embedded System Behavior
Designers	Zach Burkhardt
Inputs	External User Control: Take in user input of speed and movement from external device. Platform Communication: The master platform will establish multiple connections with slave platforms. Alignment Sensor: Proper alignments signals will be processed by the master platform to determine correct platform orientation.
Outputs	Wheel movement: The master platform transmits motor direction and speed signals to the slave platforms. Platform Communication: The slave platforms will transmit alignment sensor readings to the master platform.
Description	The software will take input from the external user controller, alignment sensors, and other platforms to determine system status and movement. The master platform software will send motor control signals to each slave platform based on its calculated direction.

5.2.4 Level 2 Alignment/Controls Software Behavior

In Figure 19, it can be seen how alignment control software should behave when the operation phase has begun. Challenges that have to be faced within the logic control include the alignment receiver voltage readings and platform communication between the master platform and slave platforms. [JS]

Figure 19: Level 2 Alignment Controls Software Behavior

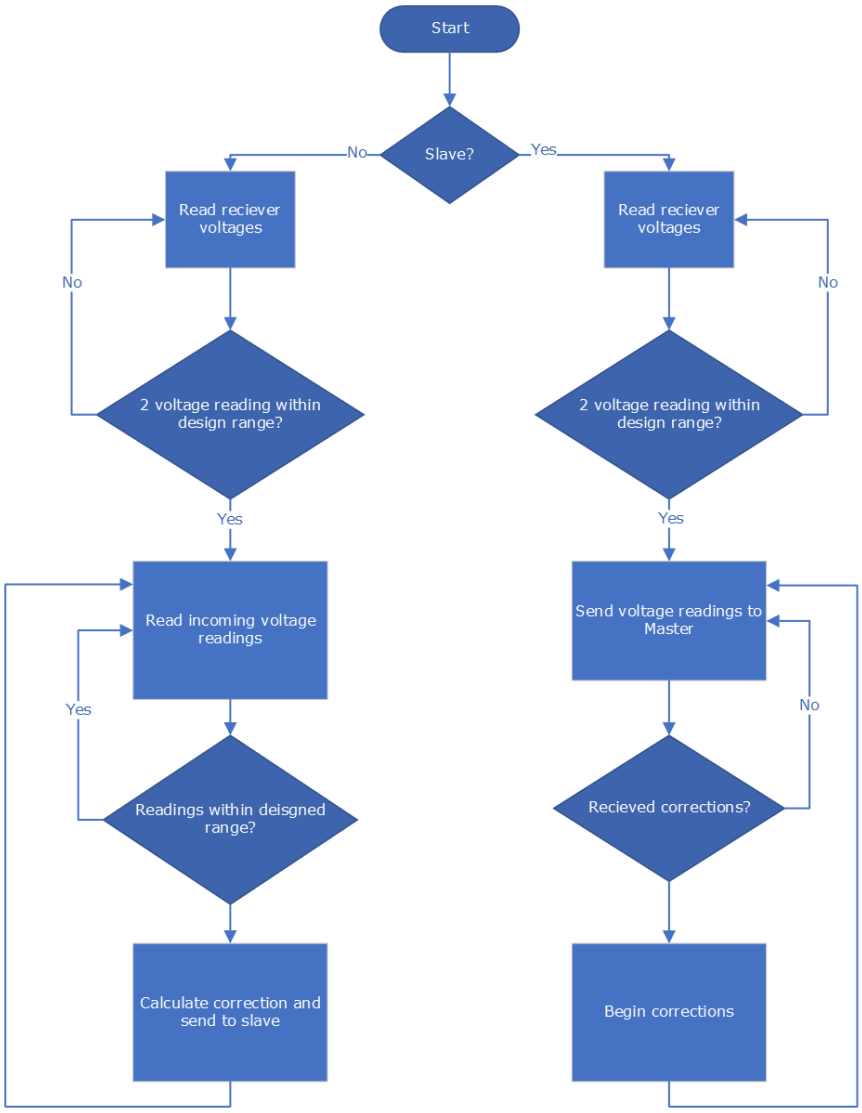
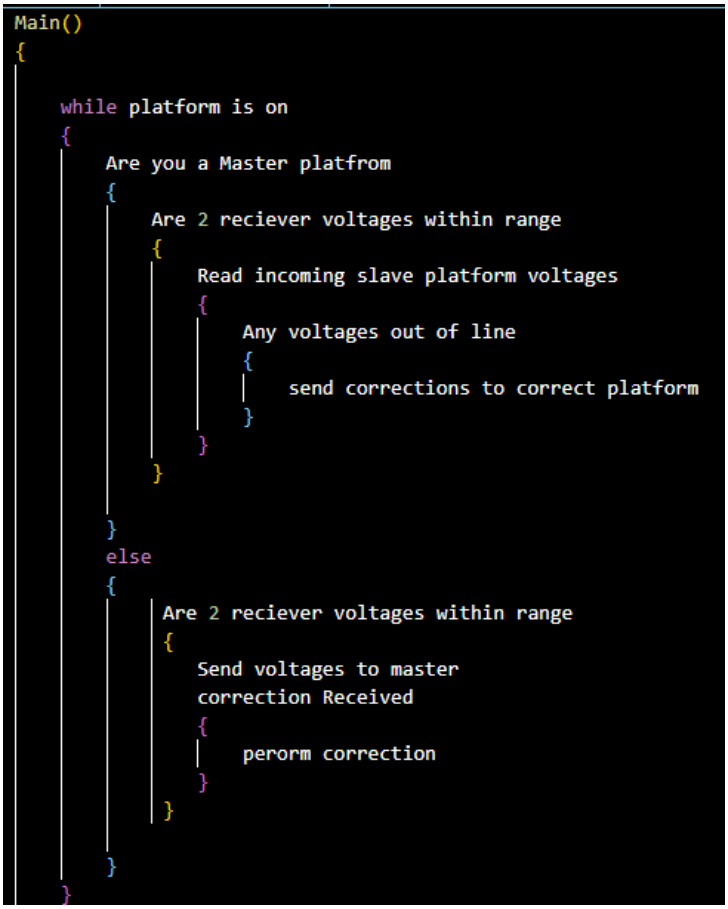


Table 23: Level 2 Alignment Control Behavior Functional Requirement Table

Module	Alignment Control Behavior
Designers	Juan Soto
Inputs	Alignment Receivers: Master platform will read receivers voltage levels to calculate any correction movement that is needed

	Platform communication: Master platforms will take in voltage alignment readings from slave platforms. Slave platforms will take in alignment corrections from the master platform when unalignment has occurred.
Outputs	Wheel movement: Motor direction and speed signals Alignment: Output proper alignment requirement message and error message.
Description	The alignment control software will take in inputs from alignment receiver voltages and platform communications. Based on these inputs, platform alignment can be automated with the software to correctly direct wheel movement and prevent and need for user manual correction.

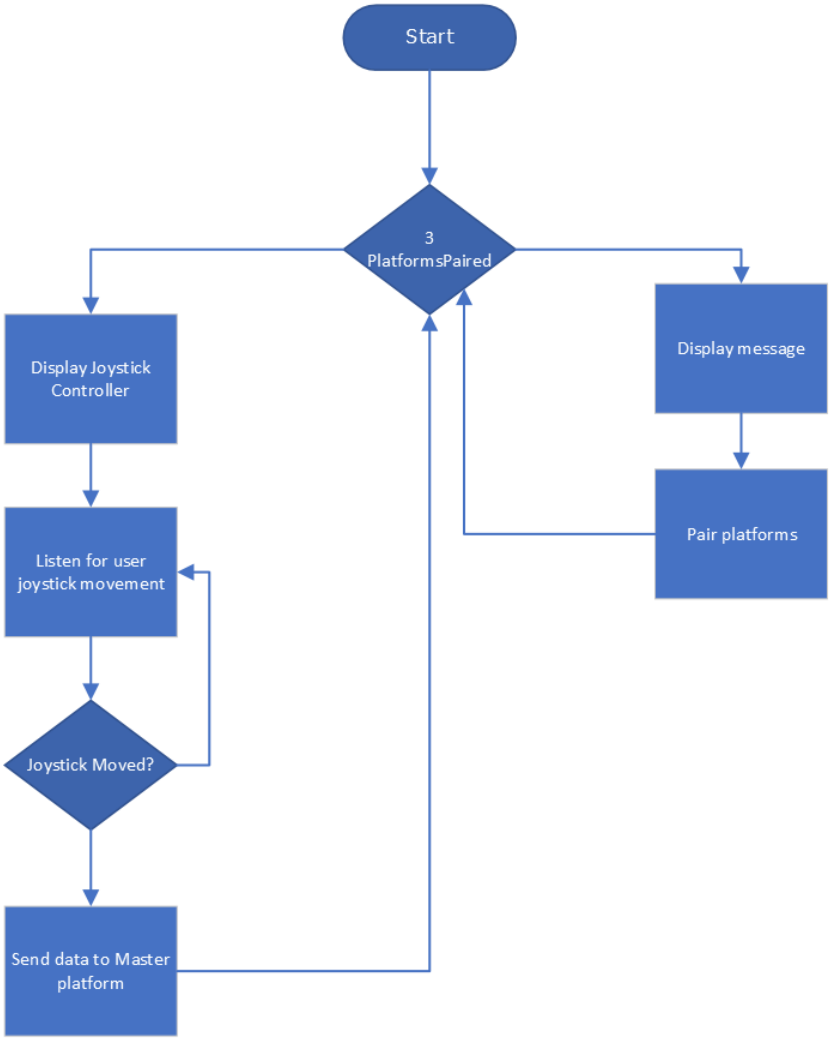
Figure 20: Control Alignment Pseudo Code



5.2.5 Level 2 User Input App Software Behavior

In Figure 21, it can be seen how app software behavior will act during user operation. The software will need to be able to connect and communicate with a master platform. Another aspect that needs to be taken into consideration is user input. User input will need to be obtained via joystick operation and speed control and communicated to the master platform. Lastly, the software will need to take communication from the master platform to alert the user if any platform has been disconnected. [JS]

Figure 21: Level 2 User Interface Software Behavior



Pseudo Code for User Input App Software Behavior

```
# Initial Bluetooth pairing/connection
While Bluetooth device not connected:
    Open system prompt to connect Bluetooth devices
If connected device not platform:
    Notify connected device not platform

# Wait for master to pair to remaining slaves
While user still in pairing mode:
    Check for message from master platform indicating new connected device
    Store platform MAC address of newly connected device

# Get initial alignment values from slaves
While initial alignment values not received:
    Check for message from master platform with alignment values for each platform

# Determine locatino of each platform for displaying on app
Calculate each platform's rectangular location based on alignment values
Display each platform's location and its ID visually

# Wait for initial auto-alignment to finish
Wait for message from master platform that platforms have been auto-aligned
Notify user initial platform alignment finished

# Main operation
Show user direction and speed controls
While app running:
    While master platform disconnected:
        Notify user that platform is disconnected from app
    While direction pressed:
        Send direction and speed to master platform
        Check for message from master platform with alignment values for each platform
        Wait 1 ms
```

Table 24: Level 2 User Input App Software Functional Requirement Table

Module	User Input App Software Behavior
Designers	Juan Soto
Inputs	Platform Communication: Master platform will communicate with the app to provide information on Bluetooth connection and alignment of platforms. User input: User touch input will be taken for direction and speed control.
Outputs	Direction Control: X and Y coordinates to communicate to the master platform along with speed.
Description	The software for the app will take in user input to supply data to the master platform to communicate wheel movement and motor speed.

5.3 Implemented Code

5.3.1 Server – Backend

The code below in Table 25 represents the primary microcontroller's firmware for the master platform. This firmware acts as the server in the network of platforms; it takes in alignment readings for all slave platforms, takes in the user's direction and speed selection from the mobile app, and calculates and sends each platform's direction and motor movement.

The firmware starts a Bluetooth LE GATT server on boot, registering many different characteristics for storing the direction and speed for each platform and the app as well as the alignment sensor values for each platform. The Bluetooth connection handle (to identify which platform to send updated instructions to) is dynamically assigned as clients (platforms or the app) are connected. This allows for any client to serve in any position in the four-platform design.

An interrupt function is called when a Bluetooth event occurs, such as a client connecting, disconnecting, or when scanning for devices to connect to. These functions house the dynamic assignment logic. When an updated speed or direction is ready to be sent to a client, this interrupt function also includes logic for determining which Bluetooth connection handle to send a Bluetooth indicate message to.

In the main program loop, the global variables storing the app's last sent direction and speed are checked for a new value. If the values have not changed, a utility function is called to calculate alignment correction values (discussed further in section 5.3.2). If the direction or speed values have changed, a separate utility function is called to start updating the direction and speed for all platforms.

This direction and speed update function checks what platforms are connected and sends the new direction and speed to these platforms using a Bluetooth characteristic indication to verify that the platform has received the updated command. Prior to the indicate, the speed is scaled, and each motor's direction of movement is calculated. [ZB]

Table 25: Implemented Server Firmware

```
# MASTER PLATFORM - PRIMARY BOARD
#1.0

import machine
from machine import Pin, PWM, UART, ADC, WDT
import ubluetooth
import time
from micropython import const


# Custom measured value of center (varies by minor variations in resistors)
#CALIBRATION = 1.35 # Master: 1.35



BLE_NAME = "ESP32"
PWM_MAX_VAL = 1023
APP_MAX_VAL = 100
MAX_COAST_SPEED = 5 # if speed below this value, coast motors
BLE_MAX_SIZE = 50 # max size of bluetooth buffer size in bytes



UUID_SERVICE_MASTER = "4fafc201-1fb5-459e-8fcc-c5c9c331914b"
UUID_CHAR_APP_DIR = "beb5483e-36e1-4688-b7f5-ea07361b26a8"
UUID_CHAR_APP_SPEED = "3954a328-b182-4c2e-a9c0-ad8535c5a30b"
UUID_CHAR_S1_DIR = "049ddb2a-786a-4967-82f5-fd6c446daf67"
UUID_CHAR_S1_SPEED = "a432948b-2968-4380-802e-542586878d68"
#UUID_CHAR_S2_DIR = "dccea7d6-27c4-44a1-b569-ca78cccbd3df"
#UUID_CHAR_S2_SPEED = "ed1ad57f-c75c-474c-b629-e0d693a838e6"
#UUID_CHAR_S3_DIR = "628cec5b-d59b-4295-bc02-1279ddeb231c"
#UUID_CHAR_S3_SPEED = "211dd0a0-41bb-4732-8121-bf0310b5da2b"

UUID_SERVICE_ALIGNMENT = "c763102a-8287-4dfa-8ccc-8c02a1492b12"
UUID_CHAR_MASTER_A1 = "72a01b4c-2a60-4561-8a60-6ff3c33e69bd"
UUID_CHAR_MASTER_A2 = "cd416615-cc8e-4f1e-a39b-cf61a21bde16"
UUID_CHAR_S1_A1 = "de48a269-f563-4a83-a77e-107c1a772c8f"
UUID_CHAR_S1_A2 = "bdfaa817-78ca-4980-a39a-4122a0affcbb"
```

```
UUID_CHAR_S2_A1 = "e19d9120-6fd9-4163-935b-cd40d09f26ff"
UUID_CHAR_S2_A2 = "966842dc-d53b-4808-9b27-b7ac390339a4"
UUID_CHAR_S3_A1 = "5ccd9477-f1af-4216-ad14-73a9b039c560"
UUID_CHAR_S3_A2 = "9118cbe6-edb2-4625-9a71-c457afe64db2"

UUID_SERVICE_SLAVE = "21f985bc-4fd3-48d7-92b7-5c40182ea6db"
UUID_CHAR_SLAVENUMBER = "f448f711-8af4-445a-b00a-4f8e20004f55"

# PROBABLY DON'T NEED MOST/ALL OF THESE, TEMPORARY?
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
_IRQ_GATTC_SERVICE_RESULT = const(9)
_IRQ_GATTC_SERVICE_DONE = const(10)
_IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
_IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
_IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
_IRQ_GATTC_DESCRIPTOR_DONE = const(14)
_IRQ_GATTC_READ_RESULT = const(15)
_IRQ_GATTC_READ_DONE = const(16)
_IRQ_GATTC_WRITE_DONE = const(17)
_IRQ_GATTC_NOTIFY = const(18)
_IRQ_GATTC_INDICATE = const(19)
_IRQ_GATTS_INDICATE_DONE = const(20)
_IRQ_MTU_EXCHANGED = const(21)

direction = "stop"
speed = "0"

# Global (dynamic) values for connection handles and characteristic value handles
connHandleApp = -1
connHandleS1 = -1
connHandleS2 = -1
connHandleS3 = -1
#valHandleS1 = None
#valHandleS2 = None
#valHandleS3 = None
S1A1 = -0.1
S1A2 = -0.1
S2A1 = -0.1
S2A2 = -0.1
S3A1 = -0.1
```

```
S3A2 = -0.1
lastConnected = -1
previousSend = 0

indicateReadyS1 = True



def bleIrqHandler(event, data):
    if event == _IRQ_CENTRAL_CONNECT:
        # A central has connected to this peripheral.
        conn_handle, addr_type, addr = data

        global connHandleS1
        global connHandleS2
        global connHandleS3
        global connHandleApp

        clientAddress = bytes(addr)

        print("\n\n**********")
        print(clientAddress)

        print("CENTRAL CLIENT CONNECTED")

        if addr_type == 0: # ESP32 BLE address type
          global lastConnected

          # Find next unassigned slave ID and assign
          if connHandleS1 == -1 and (clientAddress == b'4\x85\x18"aZ' or clientAddress == b'\xec\xda;\x0eK>'):
              connHandleS1 = conn_handle
              lastConnected = 1
              BLEConnection.gatts_indicate(connHandleS1, char_S1Direction)
              BLEConnection.gatts_indicate(connHandleS1, char_S1Speed)
              print("New Connection Handle (Slave 1): " + str(connHandleS1))
          elif connHandleS2 == -1 and (clientAddress == b'\xf4\x12\xfa\x18Z\x06' or clientAddress ==
b'\xec\xda;\x0eK>'):
              connHandleS2 = conn_handle
              lastConnected = 2
              print("New Connection Handle (Slave 2): " + str(connHandleS2))
          elif connHandleS3 == -1 and (clientAddress == b'4\x85\x18">\xfa' or clientAddress == b'\xec\xda;\x0eK>'):
              connHandleS3 = conn_handle
              lastConnected = 3
              print("New Connection Handle (Slave 3): " + str(connHandleS3))
          else:
              print("ERROR: Slaves 1-3 are already connected and assigned")
              print("Connection Handle: " + str(conn_handle))
```

```
        print("***********")

        # Convert UUID of slave number characteristic to correct variable type, search for characteristic
        slaveNumberUUID = ubluetooth.UUID(UUID_CHAR_SLAVENUMBER)
        BLEConnection.gattc_discover_characteristics(conn_handle, 1, 0xffff, slaveNumberUUID)

      elif addr_type == 1: # control app BLE address type
        global connHandleApp
        connHandleApp = conn_handle
        print("New Connection Handle (App): " + str(conn_handle))


      #print("connHandleS1" + str(connHandleS1))
      #print("connHandleS2" + str(connHandleS2))
      #print("connHandleS3" + str(connHandleS3))
      #print("connHandleApp" + str(connHandleApp))

      if (connHandleS1 > -1 and connHandleS2 > -1 and connHandleS3 > -1 and connHandleApp > -1):
        print("MAX BLUETOOTH DEVICES")
        print("BLUETOOTH NOT ADVERTISING")
      else:
        BLEConnection.gap_advertise(20000) # have to start advertising BLE again, defaults to stopping after central
client connects
        print("BLUETOOTH ADVERTISING")
      print("***********")

    elif event == _IRQ_CENTRAL_DISCONNECT:
      # A central has disconnected from this peripheral.
      conn_handle, addr_type, addr = data

      print("\n\n***********")
      print("CENTRAL CLIENT DISCONNECTED")
      BLEConnection.gap_advertise(20000, adv_data=None) # Start advertising again
      print("BLUETOOTH ADVERTISING")
      print("***********")

      # Determine which slave disconnected and reset to -1 to indicate unset/disconnected
      # And reset alignment characteristics back to "empty" to indicate unset/disconnected
      if conn_handle == connHandleApp:
        connHandleApp = -1
        print("App has disconnected")
      elif conn_handle == connHandleS1:
        connHandleS1 = -1
        BLEConnection.gatts_write(char_S1A1, "empty_A1")
        BLEConnection.gatts_write(char_S1A2, "empty_A2")
        print("Slave 1 has disconnected")
      elif conn_handle == connHandleS2:
        connHandleS2 = -1
```

```
      BLEConnection.gatts_write(char_S2A1, "empty_A1")
      BLEConnection.gatts_write(char_S2A2, "empty_A2")
      print("Slave 2 has disconnected")
    elif conn_handle == connHandleS3:
      connHandleS3 = -1
      BLEConnection.gatts_write(char_S3A1, "empty_A1")
      BLEConnection.gatts_write(char_S3A2, "empty_A2")
      print("Slave 3 has disconnected")

    print("***********")

  elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
    # Called for each characteristic found by gattc_discover_services().
    conn_handle, def_handle, value_handle, properties, uuid = data

    print("\n\n***********")
    print("NEW CHARACTERISTIC FOUND")
    print("VALUE HANDLE:")
    print(value_handle)
    print("UUID:")
    print(uuid)
    print("CONN_HANDLE:")
    print(conn_handle)
    print("***********")

    BLEConnection.gattc_read(conn_handle, value_handle)

  elif event == _IRQ_GATTC_READ_RESULT:
    # A gattc_read() has completed.
    conn_handle, value_handle, char_data = data

    global lastConnected

    if lastConnected == 1:
      BLEConnection.gattc_write(conn_handle, value_handle, "1", )
    elif lastConnected == 2:
      BLEConnection.gattc_write(conn_handle, value_handle, "2", )
    elif lastConnected == 3:
      BLEConnection.gattc_write(conn_handle, value_handle, "3", )

    #print("Calibration offset: " + str(bytes(char_data).decode('utf-8')))
    #print("***********")

  elif event == _IRQ_GATTC_INDICATE:
    # A server has sent an indicate request.
    conn_handle, value_handle, notify_data = data
```

```
    recMessage = str(bytes(notify_data).decode('utf-8'))
    alignmentValueElem = recMessage.split(',')

    if conn_handle == connHandleS1:
      global S1A1, S1A2
      S1A1 = float(alignmentValueElem[0])
      S1A2 = float(alignmentValueElem[1])
      #print("S1A1: " + str(S1A1) + ", S1A2: " + str(S1A2))
    elif conn_handle == connHandleS2:
      global S2A1, S2A2
      S2A1 = float(alignmentValueElem[0])
      S2A2 = float(alignmentValueElem[1])
      #print("S2A1: " + str(S2A1) + ", S2A2: " + str(S2A2))
    elif conn_handle == connHandleS3:
      global S3A1, S3A2
      S3A1 = float(alignmentValueElem[0])
      S3A2 = float(alignmentValueElem[1])
      #print("S3A1: " + str(S3A1) + ", S3A2: " + str(S3A2))
    else:
      print("ERROR: Unknown source of indication")

    return


  elif event == _IRQ_GATTS_INDICATE_DONE:
    # A client has acknowledged the indication.
    # Note: Status will be zero on successful acknowledgment, implementation-specific value otherwise.
    conn_handle, value_handle, status = data

    global indicateReadyS1
    indicateReadyS1 = True

    #if status == 0:
      #print("\n\n**********")
      #print("RECEIVED INDICATION ACKNOWLEDGEMENT")

    #else:
     # print("\n\n**********")
      #print("ERROR: INDICATION ACKNOWLEDGEMENT NOT RECEIVED")
      # MAYBE CODE IN HERE TO RETRY INDICATION?
      # AND IF ACK NOT RECEIVED AFTER A COUPLE RETRIES,
      # ENTER CRITICAL ERROR STATUS

    #print("**********")
    #print("STATUS:")
    #print(status)
```

```
        #print("CONN_HANDLE:")
        #print(conn_handle)
        #print("***********")


def criticalError(errMsg):
    print("\n\n~~~CRITICAL ERROR~~~")
    print("Message: " + str(errMsg))

    # BLINK BUILT-IN RGB LED (NEVERMIND, PIN 8 IN USE)

    return

def runDirection(message):
    uart.write(message) # send command to local secondary ESP32
    #print(message)

    messageElem = message.split(',') # Decode elements from encoded message (direction and scaled speeds)

    # Convert message strings to integers
    dutyFL = int(messageElem[1])
    dutyFR = int(messageElem[3])
    dutyRL = int(messageElem[5])
    dutyRR = int(messageElem[7])

    if dutyFL == -1:
        # CODE HERE TO RUN CRITICAL COMMAND (NO ACK, RUN IMMEDIATELY)
        pass
    else:
        # Update duty cycles for enable switch PWMs
        pwm_fl.duty(dutyFL)
        pwm_fr.duty(dutyFR)
        pwm_rl.duty(dutyRL)
        pwm_rr.duty(dutyRR)
        #print("\n\n***********")
        #print("Updated motor PWM signals")
        #print("***********")

    # will need code to check if speed "-1" (critical command, no ack, run immediately)
    return


def sendDirection(dir_fl, speed_fl, dir_fr, speed_fr, dir_rl, speed_rl, dir_rr, speed_rr, recipient):
    msgBoth_ForServer = dir_fl + ',' + str(int(speed_fl )) + ',' + dir_fr + ',' + str(int(speed_fr )) + ',' + dir_rl + ',' +
str(int(speed_rl )) + ',' + dir_rr + ',' + str(int(speed_rr ))
    msgDirection = dir_fl + ',' + dir_fr + ',' + dir_rl + ',' + dir_rr
    msgSpeed = str(speed_fl) + ',' + str(speed_fr) + ',' + str(speed_rl) + ',' + str(speed_rr)
```

```
if recipient == "all":
    # Send to connected slaves
    if connHandleS1 > -1:
        global indicateReadyS1

        if indicateReadyS1:
            print("Sending S1 new instruction")
            print(msgDirection)
            print(msgSpeed)
            indicateReadyS1 = False
            BLEConnection.gatts_write(char_S1Direction, msgDirection)
            BLEConnection.gatts_write(char_S1Speed, msgSpeed)
            BLEConnection.gatts_indicate(connHandleS1, char_S1Direction)
            BLEConnection.gatts_indicate(connHandleS1, char_S1Speed)
    if connHandleS2 > -1:
        pass # CODE HERE FOR WRITING AND INDICATING TO SPEED/DIRECTION CHARACTERISTICS FOR S2
    if connHandleS3 > -1:
        pass # CODE HERE FOR WRITING AND INDICATING TO SPEED/DIRECTION CHARACTERISTICS FOR S3

    # Run locally
    runDirection(msgBoth_ForServer)

elif recipient == "master":
   # Run locally
   runDirection(msgBoth_ForServer)

elif recipient == "slave1":
    if connHandleS1 > -1:
        global indicateReadyS1

        if indicateReadyS1:
            #print("Sending S1 new instruction")
            #print(msgDirection)
            #print(msgSpeed)
            # Send to slave 1
            BLEConnection.gatts_write(char_S1Direction, msgDirection)
            BLEConnection.gatts_write(char_S1Speed, msgSpeed)
            print("Diagnose")
            BLEConnection.gatts_indicate(connHandleS1, char_S1Direction)
            BLEConnection.gatts_indicate(connHandleS1, char_S1Speed)

else:
    print("\n\n**********")
    print("ERROR: Unrecognized recipient value - " + str(recipient))
    print("**********")


# newInstructionAck = False
```

```
    # while not newInstructionAck:
    #     if uart.any() > 0: # NEED TO CONSIDER SENARIO WHERE TWO BOARD'S ACK RECEIVED AT SAME TIME!!
    #         recMsg = uart.read() # PROB WILL NEED TO SPECIFY NUM OF BYTES TO READ AS ONE PLATFORM'S
WORTH
    #         print(str(recMsg))

    #         if recMsg == b'ok':
    #             newInstructionAck = True

    # msgRun = "run"
    # uart.write(msgRun)
    # print(msgRun)

    # newInstructionAck = False
    # while not newInstructionAck:
    #     if uart.any() > 0: # NEED TO CONSIDER SENARIO WHERE TWO BOARD'S ACK RECEIVED AT SAME TIME!!
    #         recMsg = uart.read() # PROB WILL NEED TO SPECIFY NUM OF BYTES TO READ AS ONE PLATFORM'S
WORTH
    #         print(str(recMsg))

    #         if recMsg == b'ok':
    #             newInstructionAck = True
    #         elif recMsg == b'err':
    #             criticalError("Error received from platform #")
    #             sendDirectionCritical('n',-1,'n',-1,'n',-1,'n',-1)


    # SAME CODE AS ABOVE TO CHECK FOR ACK. HALT ALL IF NOT RECEIVED FROM ONE PLATFORM OR IF MSG
IS ERR

    # Code to check for acknowledgement of new instructions
    # Code to send "run" command to all platforms
    # Code to check for acknowledgement of successful execution

    return

def scaleSpeed(originalSpeed):
    newSpeed = int( ( int(originalSpeed) * PWM_MAX_VAL) / 100 ) # scale from 0-100 to 0-PWM_MAX_VAL
                                            # also convert our original speed from bytes to int

    return newSpeed

def changeDirSpeed(appSpeed, newDirection):
    newSpeed = scaleSpeed(appSpeed) # scale to PWM max value
    #print("\n\n**********")
    #print("New direction: " + newDirection.decode('UTF-8')) # decode bytes to string for printing new direction
    #print("New scaled speed: " + str(newSpeed)) # convert int to string to print new speed
    #print("**********")
```

```
    # Set all motor PWM signals to 0 until direction is updated
    pwm_fl.duty(0)
    pwm_fr.duty(0)
    pwm_rl.duty(0)
    pwm_rr.duty(0)

    # Transmit updated direction and speed (front left, front right, rear left, rear right)
    # f = forward, r = reverse, a = both fwd/reverse active (coast), n = neither fwd/reverse active (brake)
    if newDirection == b'stop': # Break
        sendDirection('n',0,'n',0,'n',0,'n',0,"all")

    elif newSpeed < MAX_COAST_SPEED: # Coast
        sendDirection('a',0,'a',0,'a',0,'a',0,"all")

    else:
        if newDirection == b'up':
            sendDirection('f',newSpeed,'f',newSpeed,'f',newSpeed,'f',newSpeed,"all")
        elif newDirection == b'down':
            sendDirection('r',newSpeed,'r',newSpeed,'r',newSpeed,'r',newSpeed,"all")
        elif newDirection == b'left':
            sendDirection('r',newSpeed,'f',newSpeed,'f',newSpeed,'r',newSpeed,"all")
        elif newDirection == b'right':
            sendDirection('f',newSpeed,'r',newSpeed,'r',newSpeed,'f',newSpeed,"all")
        elif newDirection == b'top left':
            sendDirection('a',newSpeed,'f',newSpeed,'f',newSpeed,'a',newSpeed,"all")
        elif newDirection == b'top right':
            sendDirection('f',newSpeed,'a',newSpeed,'a',newSpeed,'f',newSpeed,"all")
        elif newDirection == b'bottom left':
            sendDirection('r',newSpeed,'a',newSpeed,'a',newSpeed,'r',newSpeed,"all")
        elif newDirection == b'bottom right':
            sendDirection('a',newSpeed,'r',newSpeed,'r',newSpeed,'a',newSpeed,"all")
        elif newDirection == b'turn left':
            sendDirection('r',newSpeed,'r',newSpeed,'r',newSpeed,'r',newSpeed,"master")
            sendDirection('f',newSpeed,'f',newSpeed,'f',newSpeed,'f',newSpeed,"slave1")
        elif newDirection == b'turn right':
            sendDirection('f',newSpeed,'f',newSpeed,'f',newSpeed,'f',newSpeed,"master")
            sendDirection('r',newSpeed,'r',newSpeed,'r',newSpeed,'r',newSpeed,"slave1")


        else:
            print("ERROR: Unknown direction: " + str(newDirection))

    return

def Alignment():

    #Declaring Local speed Variabels
```

```
    master_shear_adjust = scaleSpeed(speed)
    slave_shear_adjust = scaleSpeed(speed)

    #Grabbing all alignment readings from  stationary platforms platforms
    S2A2_Value = float(S2A2) #Station 1 Bottom Readings
    S3A2_Value = float(S3A2) #Station 2 Bottom Readings

    #Angle correct for Master and Slave1
    i = 0
    platform = ["master","slave1"]
    platformSpeeds = [master_shear_adjust, slave_shear_adjust]
    sensorReadings = [S2A2, S3A2]
    if S2A2 != -0.1 and S3A2 != -0.1: #Making sure station 1 and station 2 are connected
        for i in range(0,2):
            rotation = 1 #CW
            angle_diff = (sensorReadings[i]-1.53)*100
            if angle_diff >0:
                rotation = 0 #CCW
            angle_diff_scaled = (abs(angle_diff) *250)/100 #Scaling the sensor value from 0-50
            angle_adjusted_speed = platformSpeeds[i] + angle_diff_scaled #adding calculated speed adjustment to
the platorm speed
            if angle_adjusted_speed > (platformSpeeds[i] + 50) and (angle_adjusted_speed>15 or
angle_adjusted_speed <-15): #if out of tolerance then overcorrect speed
                AngleMaxCorrectSpeed = platformSpeeds[i] + 70
                if AngleMaxCorrectSpeed > 1023:
                    AngleCorrection(rotation,1023,platformSpeeds[i],platform[i])
                else:
                    AngleCorrection(rotation, AngleMaxCorrectSpeed, platformSpeeds[i], platform[i])
            else:
                AngleCorrection(rotation,angle_adjusted_speed, platformSpeeds[i], platform[i]) #Sending the
alignment correction to the designated platform
    return


def AngleCorrection(newdirection, adjustedspeed, currentPlatformSpeed ,platform):
   global previousSend

   if direction == b"up":

        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),
platform)

        elif newdirection == 1:
```

```
            #print("CW")

sendDirection('f',int(adjustedspeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),
platform)
    elif direction == b"down":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),
platform)

        elif newdirection == 1:
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),
platform)
    elif direction == b"left":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'r',int(currentPlatformSpeed),
platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'r',int(adjustedspeed),
platform)
    elif direction == b"right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(adjustedspeed),'f',int(adjustedspeed),platfor
m)
        elif newdirection == 1:
            #print("CW")

sendDirection('f',int(adjustedspeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),
platform)
    elif direction == b"top left":
        previousSend = 0
        # 0-CCW, 1-CW
```

```
        if newdirection == 0:
            #print("CCW")

sendDirection('a',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'a',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('a',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'a',int(currentPlatfor
mSpeed),platform)
    elif direction == b"top right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'f',int(adjuste
dspeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('f',int(adjustedspeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'f',int(currentPlatfor
mSpeed),platform)
    elif direction == b"bottom left":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'r',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'r',int(adjuste
dspeed),platform)
    elif direction == b"bottom right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('a',int(currentPlatformSpeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),'a',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")
```

```
sendDirection('a',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),platform)
    elif direction == b"turn right":
        # 0-CCW, 1-CW
        previousSend = 0
        if newdirection == 0:
            #print("CCW")
            if platform == "master":
                sendDirection('f',int(0),'f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),platform)
            else:
                #time.sleep(.5)
                sendDirection('r',int(adjustedspeed-50),'a',int(0),'r',int(adjustedspeed-50),'a',int(0),platform)


        elif newdirection == 1:
            #print("CW")
            if platform == "master":
                sendDirection('f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),'f',int(0),platform)
            else:
                #time.sleep(.5)
                sendDirection('a',int(0),'r',int(adjustedspeed-50),'a',int(0),'r',int(adjustedspeed-50),platform)
    elif direction == b"turn left":
        previousSend = 0
        if newdirection == 0:
            #print("CCW")
            if platform == "master":
                sendDirection('r',int(adjustedspeed-50),'r',int(0),'r',int(adjustedspeed-50),'r',int(0),platform)
            else:
                sendDirection('f',int(0),'f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),platform)


        elif newdirection == 1:
            #print("CW")
            if platform == "master":
                sendDirection('r',int(0),'r',int(adjustedspeed-50),'r',int(0),'r',int(adjustedspeed-50),platform)
            else:
                sendDirection('f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),'f',int(0),platform)

    elif direction == b"stop":
        if previousSend < 4:
            previousSend += 1
            print("SEND STOP ANGLE")
            sendDirection('n',0,'n',0,'n',0,'n',0,"all")
    return
```

```
time.sleep(2) # boot delay to avoid sending serial data to
            # secondary ESP32 before it is booted

uart = UART(1) # construct instance of UART class for intraplatform communications
uart.init(baudrate=115200, tx=2, rx=8) # configure baudrate and assign transmit pins
BLEConnection = ubluetooth.BLE() # construct instance of Bluetooth class
BLEConnection.active(True) # set the Bluetooth radio to active
BLEConnection.irq(bleIrqHandler) # configure interrupt handler


# Bluetooth advertisement
advertiseName = bytes(BLE_NAME, 'UTF-8')
BLEConnection.config(addr_mode=0x00,
                     gap_name=advertiseName,
                     rxbuf=BLE_MAX_SIZE)


# Register Bluetooth service and characteristics - direction and speed
serviceUUID_dirSpeed = ubluetooth.UUID(UUID_SERVICE_MASTER)
char_Direction = (ubluetooth.UUID(UUID_CHAR_APP_DIR),
                  ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY,)
char_Speed = (ubluetooth.UUID(UUID_CHAR_APP_SPEED),
              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY,)
char_S1Direction = (ubluetooth.UUID(UUID_CHAR_S1_DIR),
              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S1Speed = (ubluetooth.UUID(UUID_CHAR_S1_SPEED),
              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
#char_S2Direction = (ubluetooth.UUID(UUID_CHAR_S2_DIR),
#              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
#char_S2Speed = (ubluetooth.UUID(UUID_CHAR_S2_SPEED),
#              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
#char_S3Direction = (ubluetooth.UUID(UUID_CHAR_S3_DIR),
#              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
#char_S3Speed = (ubluetooth.UUID(UUID_CHAR_S3_SPEED),
#              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
#serviceTuple_dirSpeed = (serviceUUID_dirSpeed, (char_Direction, char_Speed, char_S1Direction, char_S1Speed,
char_S2Direction, char_S2Speed, char_S3Direction, char_S3Speed,))
serviceTuple_dirSpeed = (serviceUUID_dirSpeed, (char_Direction, char_Speed, char_S1Direction, char_S1Speed,))

# Register Bluetooth service and characteristics - alignment
serviceUUID_alignment = ubluetooth.UUID(UUID_SERVICE_ALIGNMENT)
char_MasterA1 = (ubluetooth.UUID(UUID_CHAR_MASTER_A1),
              ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_MasterA2 = (ubluetooth.UUID(UUID_CHAR_MASTER_A2),
```

```
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S1A1 = (ubluetooth.UUID(UUID_CHAR_S1_A1),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S1A2 = (ubluetooth.UUID(UUID_CHAR_S1_A2),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S2A1 = (ubluetooth.UUID(UUID_CHAR_S2_A1),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S2A2 = (ubluetooth.UUID(UUID_CHAR_S2_A2),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S3A1 = (ubluetooth.UUID(UUID_CHAR_S3_A1),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
char_S3A2 = (ubluetooth.UUID(UUID_CHAR_S3_A2),
            ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY | ubluetooth.FLAG_INDICATE,)
serviceTuple_alignment = (serviceUUID_alignment, (char_MasterA1, char_MasterA2, char_S1A1, char_S1A2,
char_S2A1, char_S2A2, char_S3A1, char_S3A2,))

#allServices = (serviceTuple_dirSpeed, serviceTuple_alignment, )
#( (char_Direction, char_Speed, char_S1Direction, char_S1Speed, char_S2Direction, char_S2Speed, char_S3Direction,
char_S3Speed,), (char_MasterA1, char_MasterA2, char_S1A1, char_S1A2, char_S2A1, char_S2A2, char_S3A1,
char_S3A2,),) = BLEConnection.gatts_register_services(allServices)
allServices = (serviceTuple_dirSpeed, serviceTuple_alignment, )
( (char_Direction, char_Speed, char_S1Direction, char_S1Speed,), (char_MasterA1, char_MasterA2, char_S1A1,
char_S1A2, char_S2A1, char_S2A2, char_S3A1, char_S3A2,),) = BLEConnection.gatts_register_services(allServices)


BLEConnection.gap_advertise(20000, bytearray('\x02\x01\x02') + bytearray((len(advertiseName) + 1, 0x09)) +
BLE_NAME) # advertise 20 ms (in microseconds) to be discovered by iOS
            # can resume advertising with gap_advertise(internal_in_microsec), don't have to repass advertisement
data
            # can stop advertising with gap_advertise(None?????)


# Initial values for direction and speed
BLEConnection.gatts_write(char_Direction, "stop")
BLEConnection.gatts_write(char_Speed, "0")
BLEConnection.gatts_write(char_S1Direction, "n,n,n,n")
BLEConnection.gatts_write(char_S1Speed, "0,0,0,0")
#BLEConnection.gatts_write(char_S2Direction, "n,n,n,n")
#BLEConnection.gatts_write(char_S2Speed, "0,0,0,0")
#BLEConnection.gatts_write(char_S3Direction, "n,n,n,n")
#BLEConnection.gatts_write(char_S3Speed, "0,0,0,0")
lastDirection = BLEConnection.gatts_read(char_Direction)
lastSpeed = BLEConnection.gatts_read(char_Speed)

# Initial values for alignment
BLEConnection.gatts_write(char_MasterA1, "0")
BLEConnection.gatts_write(char_MasterA2, "0")
BLEConnection.gatts_write(char_S1A1, "empty_A1")
```

```
BLEConnection.gatts_write(char_S1A2, "empty_A2")
BLEConnection.gatts_write(char_S2A1, "empty_A1")
BLEConnection.gatts_write(char_S2A2, "empty_A2")
BLEConnection.gatts_write(char_S3A1, "empty_A1")
BLEConnection.gatts_write(char_S3A2, "empty_A2")



# Setup PWM pins
pwm_fl = PWM(Pin(5), freq=5000, duty=0) # Front left motor
pwm_fr = PWM(Pin(6), freq=5000, duty=0) # Front right motor
pwm_rl = PWM(Pin(7), freq=5000, duty=0) # Rear left motor
pwm_rr = PWM(Pin(10), freq=5000, duty=0) # Rear right motor


# Set initial PWM signals to 0
pwm_fl.duty(0)
pwm_fr.duty(0)
pwm_rl.duty(0)
pwm_rr.duty(0)

# Set Alignment Pins
MA1 = ADC(Pin(0,mode=Pin.IN))
MA1.atten(ADC.ATTN_11DB)
MA2 = ADC(Pin(1,mode=Pin.IN))
MA2.atten(ADC.ATTN_11DB)

# Start watchdog monitor
wdt = WDT(timeout=2000)


print("\n\n***********")
print("FIRMWARE FOR SERVER (VERSION 1.3.0)")
print("BOOT COMPLETE")
print("***********")

print("\n\n***********")
print("BLUETOOTH ADVERTISING")
print("***********")



while True:
    # Read updated direction/speed values from characteristics
    direction = BLEConnection.gatts_read(char_Direction)
    speed = BLEConnection.gatts_read(char_Speed) # kept speed as bytes instead of converting to int for efficiency
                                    # bytes are converted to int within the speed scaling function
```

```
    # If speed or direction are updated,
    # run function to send updated instructions
    if direction != lastDirection:
        lastDirection = direction
        changeDirSpeed(speed, direction)

    if speed != lastSpeed:
        if int(speed) > APP_MAX_VAL:
            print("\n\n**********")
            print("ERROR: Bad speed received")
            print("Value: " + str(speed))
            print("**********")
            speed = lastSpeed
        else:
            lastSpeed = speed
            changeDirSpeed(speed, direction)
    else:
       #MasterAngle()
       #SlaveAngle()
       Alignment()

    wdt.feed() # Feed watchdog timer
    time.sleep(.3)
```

5.3.2 Server – Alignment

In Figure 22 the tolerance range for the alignment sensors is set to be +/- 0.20 from the center value. The alignment process first begins with the master reading in the stationary platform's alignment values. Once a value is read, the difference between the received and center value is calculated and scaled by one hundred to avoid floating point math. Based on the sign of the difference value, the platform will know either to correct counterclockwise or clockwise. Next the platform speed is increased from a range of zero to fifty. If the increased speed is greater than the platform speed plus fifty, then the platform has fallen off tolerance and now a faster-increased speed is added for counterclockwise or clockwise correction. This logic can also be seen in table 26, which shows the implementation alignment code for the final design. [JS]

Figure 22: Alignment Tolerance Range

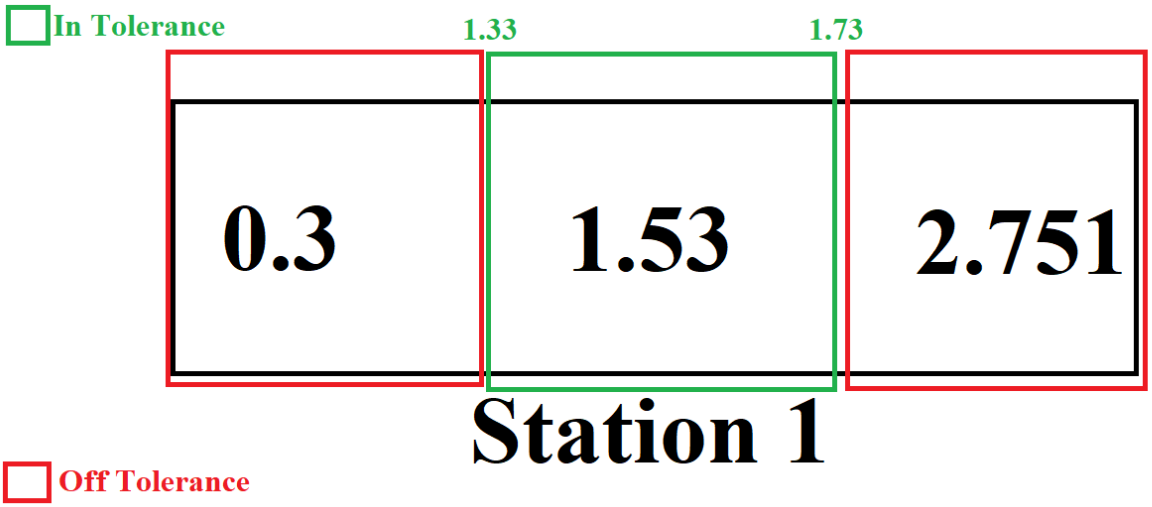


Table 26: Implemented Alignment Code

```
def Alignment():

    #Declaring Local speed Variabels
    master_shear_adjust = scaleSpeed(speed)
    slave_shear_adjust = scaleSpeed(speed)

    #Grabbing all alignment readings from  stationary platforms platforms
    S2A2_Value = float(S2A2) #Station 1 Bottom Readings
    S3A2_Value = float(S3A2) #Station 2 Bottom Readings

    #Angle correct for Master and Slave1
    i = 0
    platform = ["master","slave1"]
    platformSpeeds = [master_shear_adjust, slave_shear_adjust]
    sensorReadings = [S2A2, S3A2]
    if S2A2 != -0.1 and S3A2 != -0.1: #Making sure station 1 and station 2 are connected
        for i in range(0,2):
            rotation = 1 #CW
            angle_diff = (sensorReadings[i]-1.53)*100
```

```
            if angle_diff >0:
                rotation = 0 #CCW
            angle_diff_scaled = (abs(angle_diff) *250)/100 #Scaling the sensor value from 0-50
            angle_adjusted_speed = platformSpeeds[i] + angle_diff_scaled #adding calculated speed adjustment to
the platorm speed
            if angle_adjusted_speed > (platformSpeeds[i] + 50) and (angle_adjusted_speed>15 or
angle_adjusted_speed <-15): #if out of tolerance then overcorrect speed
                AngleMaxCorrectSpeed = platformSpeeds[i] + 70
                if AngleMaxCorrectSpeed > 1023:
                    AngleCorrection(rotation,1023,platformSpeeds[i],platform[i])
                else:
                    AngleCorrection(rotation, AngleMaxCorrectSpeed, platformSpeeds[i], platform[i])
            else:
                AngleCorrection(rotation,angle_adjusted_speed, platformSpeeds[i], platform[i]) #Sending the
alignment correction to the designated platform
    return
```

```
def AngleCorrection(newdirection, adjustedspeed, currentPlatformSpeed ,platform):
   global previousSend

   if direction == b"up":

        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),
platform)

        elif newdirection == 1:
            #print("CW")

sendDirection('f',int(adjustedspeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),
platform)
   elif direction == b"down":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),
platform)

        elif newdirection == 1:
```

```
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),
platform)
    elif direction == b"left":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'r',int(currentPlatformSpeed),
platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'r',int(adjustedspeed),
platform)
    elif direction == b"right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(adjustedspeed),'f',int(adjustedspeed),platfor
m)
        elif newdirection == 1:
            #print("CW")

sendDirection('f',int(adjustedspeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),
platform)
    elif direction == b"top left":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('a',int(currentPlatformSpeed),'f',int(currentPlatformSpeed),'f',int(adjustedspeed),'a',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('a',int(currentPlatformSpeed),'f',int(adjustedspeed),'f',int(currentPlatformSpeed),'a',int(currentPlatfor
mSpeed),platform)
    elif direction == b"top right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
```

```
            #print("CCW")

sendDirection('f',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'f',int(adjusted
dspeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('f',int(adjustedspeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'f',int(currentPlatfor
mSpeed),platform)
    elif direction == b"bottom left":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('r',int(adjustedspeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'r',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('r',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'a',int(currentPlatformSpeed),'r',int(adjusted
dspeed),platform)
    elif direction == b"bottom right":
        previousSend = 0
        # 0-CCW, 1-CW
        if newdirection == 0:
            #print("CCW")

sendDirection('a',int(currentPlatformSpeed),'r',int(currentPlatformSpeed),'r',int(adjustedspeed),'a',int(currentPlatfor
mSpeed),platform)
        elif newdirection == 1:
            #print("CW")

sendDirection('a',int(currentPlatformSpeed),'r',int(adjustedspeed),'r',int(currentPlatformSpeed),'a',int(currentPlatfor
mSpeed),platform)
    elif direction == b"turn right":
        # 0-CCW, 1-CW
        previousSend = 0
        if newdirection == 0:
            #print("CCW")
            if platform == "master":
                sendDirection('f',int(0),'f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),platform)
            else:
                #time.sleep(.5)
                sendDirection('r',int(adjustedspeed-50),'a',int(0),'r',int(adjustedspeed-50),'a',int(0),platform)



        elif newdirection == 1:
```

```
            #print("CW")
            if platform == "master":
                sendDirection('f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),'f',int(0),platform)
            else:
                #time.sleep(.5)
                sendDirection('a',int(0),'r',int(adjustedspeed-50),'a',int(0),'r',int(adjustedspeed-50),platform)
    elif direction == b"turn left":
        previousSend = 0
        if newdirection == 0:
            #print("CCW")
            if platform == "master":
                sendDirection('r',int(adjustedspeed-50),'r',int(0),'r',int(adjustedspeed-50),'r',int(0),platform)
            else:
                sendDirection('f',int(0),'f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),platform)


        elif newdirection == 1:
            #print("CW")
            if platform == "master":
                sendDirection('r',int(0),'r',int(adjustedspeed-50),'r',int(0),'r',int(adjustedspeed-50),platform)
            else:
                sendDirection('f',int(adjustedspeed-50),'f',int(0),'f',int(adjustedspeed-50),'f',int(0),platform)

    elif direction == b"stop":
        if previousSend < 4:
            previousSend += 1
            print("SEND STOP ANGLE")
            sendDirection('n',0,'n',0,'n',0,'n',0,"all")
    return
```

5.3.3 Secondary

The secondary firmware in Table 27 is flashed on each secondary microcontroller in both server and client platforms. It receives updated motor direction instructions from each platform's primary microcontroller. There are eight GPIO signal pins configured, with two pins for each motor to indicate four states (forward, reverse, brake, or coast). The firmware communicates locally via serial UART with the primary microcontroller. [ZB]

Table 27: Implemented Secondary Firmware

```
# MASTER/SLAVE PLATFORM - SECONDARY BOARD
```

```
import machine
from machine import Pin, PWM, UART
import time


def setDirection(direction, fwdPin, revPin):
    if direction == 'a':
        fwdPin.value(1)
        revPin.value(1)
    elif direction == 'n':
        fwdPin.value(0)
        revPin.value(0)
    elif direction == 'f':
        fwdPin.value(1)
        revPin.value(0)
    elif direction == 'r':
        fwdPin.value(0)
        revPin.value(1)
    else:
        criticalError("Unrecognized direction value")


def criticalError(errMsg):
    print("\n\n~~~CRITICAL ERROR~~~")
    print("Message: " + str(errMsg))

    # BLINK BUILT-IN RGB LED (NEVERMIND, PIN 8 IN USE)

    return

def runDirection(message):
    messageElem = message.split(',') # Decode elements from encoded message (direction and scaled speeds)

    # Convert a speed strings to integers to check for critical command
    dutyFL = int(messageElem[1])

    if dutyFL == -1:
        # CODE HERE TO RUN CRITICAL COMMAND (NO ACK, RUN IMMEDIATELY)
        pass
    else: # run commands normally with acknowledgements
        # SEND ACK OF MESSAGE
        # WAIT FOR EXECUTION COMMAND
        # EXECUTE (CODE BELOW)

        # Update duty cycles for enable switch PWMs
        setDirection(messageElem[0], fl_forward, fl_reverse)
        setDirection(messageElem[2], fr_forward, fr_reverse)
```

```
        setDirection(messageElem[4], rl_forward, rl_reverse)
        setDirection(messageElem[6], rr_forward, rr_reverse)
        print("***********")
        print("Updated motor direction signals")
        print("***********")

        # SEND ACK OF SUCCESSFUL EXECUTION

    return



time.sleep(2) # boot delay to avoid sending serial data to
            # primary ESP32 before it is booted

uart = UART(1) # construct instance of UART class for intraplatform communications
uart.init(baudrate=115200, tx=2, rx=8) # configure baudrate and assign transmit pins



# Setup motor control direction pins
fl_forward = machine.Pin(0, machine.Pin.OUT) # Front left motor
fl_reverse = machine.Pin(5, machine.Pin.OUT)
fr_forward = machine.Pin(1, machine.Pin.OUT) # Front right motor
fr_reverse = machine.Pin(6, machine.Pin.OUT)
rl_forward = machine.Pin(3, machine.Pin.OUT) # Rear left motor
rl_reverse = machine.Pin(7, machine.Pin.OUT)
rr_forward = machine.Pin(4, machine.Pin.OUT) # Rear right motor
rr_reverse = machine.Pin(10, machine.Pin.OUT)


# Set initial output signals to low
fl_forward.value(0)
fl_reverse.value(0)
fr_forward.value(0)
fr_reverse.value(0)
rl_forward.value(0)
rl_reverse.value(0)
rr_forward.value(0)
rr_reverse.value(0)



print("\n\n***********")
print("FIRMWARE FOR SECONDARY (VERSION 1.0.3)")
print("BOOT COMPLETE")
print("***********")
```

```
print("\n\n***********")
print("WAITING FOR SERIAL COMMAND")
print("***********")

while True:
    while uart.any() > 0:
        msg = uart.readline() # Read in UART message
        print("\n\n***********")
        print("Message: " + str(msg))


        # Run new direction/speed if message is correct
        # ASCII value 252 (\xfc) gets sent via UART when server/client reboots
        if (msg[0] == 252):
            print("***********")
            print("BAD MESSAGE")
            print("***********")
        else:
            runDirection(msg.decode('UTF-8')) # Decode and run (we don't want to decode until
                                    # we verify it's a valid message)

    #time.sleep(1)
```

5.3.4 Client

The client firmware in Table 28 is flashed on each client platform's primary microcontroller. On boot, the firmware starts a Bluetooth LE GATT client library instance that scans for devices advertising under the name "ESP32". On each discovered Bluetooth server, the Bluetooth interrupt function is called to determine if this is a server platform, and if so, connect to and dynamically assign Bluetooth connection handle numbers.

The main loop waits for an updated direction and speed instruction from the server platform. On reception of a Bluetooth characteristic indication from the server, the instruction is parsed for the new direction and speed and the corresponding global variables are updated. The main loop then identifies the updated values and calls a utility function to begin changing direction and speed. The utility function sends the new motor direction instruction to the

secondary microcontroller via serial UART. Then, the PWM output signals are also updated to the new scaled speed. [ZB]

Table 28: Implemented Client Firmware

```
# SLAVE PLATFORM - PRIMARY BOARD

import machine
from machine import Pin, PWM, UART, ADC, WDT
import ubluetooth
import time
from micropython import const


# Custom measured value of center (varies by minor variations in resistors)
# Update before flashing firmware
CALIBRATION = 1.25 # Slave A: ???, Slave B: ???, Slave C: ???



UUID_SERVICE_ALIGNMENT = "c763102a-8287-4dfa-8ccc-8c02a1492b12"
UUID_CHAR_MASTER_A1 = "72a01b4c-2a60-4561-8a60-6ff3c33e69bd"
UUID_CHAR_MASTER_A2 = "cd416615-cc8e-4f1e-a39b-cf61a21bde16"
UUID_CHAR_S1_A1 = "de48a269-f563-4a83-a77e-107c1a772c8f"
UUID_CHAR_S1_A2 = "bdfaa817-78ca-4980-a39a-4122a0affcbb"
UUID_CHAR_S2_A1 = "e19d9120-6fd9-4163-935b-cd40d09f26ff"
UUID_CHAR_S2_A2 = "966842dc-d53b-4808-9b27-b7ac390339a4"
UUID_CHAR_S3_A1 = "5ccd9477-f1af-4216-ad14-73a9b039c560"
UUID_CHAR_S3_A2 = "9118cbe6-edb2-4625-9a71-c457afe64db2"

UUID_SERVICE_SLAVE = "21f985bc-4fd3-48d7-92b7-5c40182ea6db"
UUID_CHAR_SLAVENUMBER = "f448f711-8af4-445a-b00a-4f8e20004f55"
UUID_CHAR_ALIGNMENT = "0f5ff9ec-b18f-4b3c-9eab-48c38ce54b9f"

BLE_ADV_STRING = b'\x02\x01\x02\x06\tESP32'

# PROBABLY DON'T NEED MOST/ALL OF THESE, TEMPORARY?
_IRQ_CENTRAL_CONNECT = const(1)
_IRQ_CENTRAL_DISCONNECT = const(2)
_IRQ_GATTS_WRITE = const(3)
_IRQ_GATTS_READ_REQUEST = const(4)
_IRQ_SCAN_RESULT = const(5)
_IRQ_SCAN_DONE = const(6)
_IRQ_PERIPHERAL_CONNECT = const(7)
_IRQ_PERIPHERAL_DISCONNECT = const(8)
_IRQ_GATTC_SERVICE_RESULT = const(9)
_IRQ_GATTC_SERVICE_DONE = const(10)
```

```
_IRQ_GATTC_CHARACTERISTIC_RESULT = const(11)
_IRQ_GATTC_CHARACTERISTIC_DONE = const(12)
_IRQ_GATTC_DESCRIPTOR_RESULT = const(13)
_IRQ_GATTC_DESCRIPTOR_DONE = const(14)
_IRQ_GATTC_READ_RESULT = const(15)
_IRQ_GATTC_READ_DONE = const(16)
_IRQ_GATTC_WRITE_DONE = const(17)
_IRQ_GATTC_NOTIFY = const(18)
_IRQ_GATTC_INDICATE = const(19)
_IRQ_GATTS_INDICATE_DONE = const(20)
_IRQ_MTU_EXCHANGED = const(21)

addressMaster = b'0' # global variable for master platform MAC address
            # allows for dynamic assignment of master platform
connHandleMaster = -1 # Connection Handle/ID for master platform connection (-1 indicates unset/disconnected)
valHandleA1 = -1
valHandleA2 = -1
slaveNumber = -1
direction = "n,n,n,n"
speed = "0,0,0,0"
connected = False
#lastSlaveNumber = -1

#indicateReadyMaster = True


def bleIrqHandler(event, data):
    if event == _IRQ_SCAN_RESULT:
      # A single scan result.
      addr_type, addr, adv_type, rssi, adv_data = data

      print("\n\n***********")
      print("NEW DEVICE DISCOVERED")
      print("***********")

      print("Address Type: " + str(addr_type))
      print("Address: " + str(bytes(addr)))
      print("Advertisement Type: " + str(bytes(adv_type)))
      print("Advertisement Data: " + str(bytes(adv_data)))
      print("RSSI: " + str(rssi))
      print("***********")

      if bytes(adv_data) == BLE_ADV_STRING:
        print("\n\n***********")
        print("FOUND MASTER ESP32")
        print("SCANNING STOPPED")
        print("***********")
```

```
        BLEConnection.gap_scan(None) # stop BLE scanning
        addressMaster = bytes(addr) # update variable with discovered master platform MAC address

        BLEConnection.gap_connect(0, addressMaster)

    elif event == _IRQ_PERIPHERAL_CONNECT:
        # A successful gap_connect().
        conn_handle, addr_type, addr = data

        global connHandleMaster
        connHandleMaster = conn_handle # Set master platform handle ID back to -1 (unset)

        print("\n\n***********")
        print("CONNECTION SUCCESSFUL")
        print("***********")
        print("Connection Handle (Master): " + str(connHandleMaster))
        print("***********")

        UUID = ubluetooth.UUID(UUID_CHAR_S1_A1)
        BLEConnection.gattc_discover_characteristics(connHandleMaster, 1, 0xffff, UUID,)
        time.sleep(2)

        #UUID = ubluetooth.UUID(UUID_CHAR_S1_A2)
        #BLEConnection.gattc_discover_characteristics(connHandleMaster, 1, 0xffff, UUID,)
        #time.sleep(2)

    elif event == _IRQ_PERIPHERAL_DISCONNECT:
        # Connected peripheral has disconnected.
        conn_handle, addr_type, addr = data
        print("\n\n***********")
        print("DISCONNECTED")
        print("SCANNING FOR MASTER ESP32")
        print("***********")

        # Stop platform if disconnects from master
        pwm_fl.duty(0)
        pwm_fr.duty(0)
        pwm_rl.duty(0)
        pwm_rr.duty(0)

        global connHandleMaster
        connHandleMaster = -1 # Set master platform handle ID back to -1 (unset)

        global slaveNumber
        BLEConnection.gatts_write(charSlaveNumber, "-1") # Set slave number back to -1
        slaveNumber = -1
```

```
    global valHandleA1
    global valHandleA2
    valHandleA1 = -1 # Reset value handles for alignment back to default (-1)
    valHandleA2 = -1

    BLEConnection.gap_scan(0) # start scanning again to reconnect

  elif event == _IRQ_GATTC_CHARACTERISTIC_RESULT:
    # Called for each characteristic found by gattc_discover_services().
    conn_handle, def_handle, value_handle, properties, uuid = data

    print("\n\n***********")
    print("NEW CHARACTERISTIC FOUND")
    print("VALUE HANDLE:")
    print(value_handle)
    print("UUID:")
    print(uuid)
    print("CONN_HANDLE:")
    print(conn_handle)
    print("***********")

    global valHandleA1
    global valHandleA2
    if valHandleA1 == -1:
      valHandleA1 = value_handle
    elif valHandleA2 == -1:
      valHandleA2 = value_handle
    else:
       print("\n\n***********")
       print("ERROR: Value handle 1 & 2 already assigned")
       print("Value Handle A1:")
       print(valHandleA1)
       print("Value Handle A2:")
       print(valHandleA2)
       print("New Value Handle:")
       print(value_handle)
       print("***********")

  elif event == _IRQ_GATTC_INDICATE:
    # A server has sent an indicate request.
    conn_handle, value_handle, notify_data = data

    recMessage = str(bytes(notify_data).decode('utf-8'))
    #print("\n\n***********")
    #print("NEW CHARACTERISTIC INDICATION")
    #print("***********")
    #print("NOTIFY_DATA:")
```

```
        #print(recMessage)
        #print("***********")

        global speed
        global direction

        firstCharacter = recMessage[0]
        if firstCharacter.isdigit():
          speed = recMessage
        else:
          direction = recMessage

        messageDirElem = direction.split(',') # Decode elements from encoded message (direction)
        messageSpeedElem = speed.split(',') # Decode elements from encoded message (scaled speeds)

        newMessage = messageDirElem[0] + ',' + messageSpeedElem[0] + ',' + messageDirElem[1] + ',' +
messageSpeedElem[1] + ',' + messageDirElem[2] + ',' + messageSpeedElem[2] + ',' + messageDirElem[3] + ',' +
messageSpeedElem[3]

        runDirection(newMessage)

      elif event == _IRQ_GATTS_INDICATE_DONE:
        # A client has acknowledged the indication.
        # Note: Status will be zero on successful acknowledgment, implementation-specific value otherwise.
        conn_handle, value_handle, status = data

        #global indicateReadyMaster
        #indicateReadyMaster = True

def runDirection(message):
    uart.write(message) # send command to local secondary ESP32
    #print(message)

    messageElem = message.split(',') # Decode elements from encoded message (direction and scaled speeds)

    # Convert message strings to integers
    dutyFL = int(messageElem[1])
    dutyFR = int(messageElem[3])
    dutyRL = int(messageElem[5])
    dutyRR = int(messageElem[7])

    if dutyFL == -1:
        # CODE HERE TO RUN CRITICAL COMMAND (NO ACK, RUN IMMEDIATELY)
        pass
    else: # run commands normally with acknowledgements
        # SEND ACK OF MESSAGE
        # WAIT FOR EXECUTION COMMAND
```

```
        # EXECUTE (CODE BELOW)
        #print("\n\n***********")
        #print("NEW SPEED/DIRECTION:")
        #print(message)
        #print("***********")
        # Update duty cycles for enable switch PWMs
        pwm_fl.duty(dutyFL)
        pwm_fr.duty(dutyFR)
        pwm_rl.duty(dutyRL)
        pwm_rr.duty(dutyRR)
        #print("\n\n***********")
        #print("Updated motor PWM signals")
        #print("***********")

        # SEND ACK OF SUCCESSFUL EXECUTION

    return

def Readalignment():
    #global indicateReadyMaster

    #print("indicateReadyMaster: " + str(indicateReadyMaster))

    #if indicateReadyMaster: # Don't send new alignment value until indicate received (or not received)
    #indicateReadyMaster = False

    print("***********")
    A1_Value = A1.read_uv() * 0.000001
    print("A1 Value: " + str(A1_Value))

    A2_Value = A2.read_uv() * 0.000001
    print("A2 Value: " + str(A2_Value))

    msgCombined = str(A1_Value) + ',' + str(A2_Value)

    BLEConnection.gatts_write(charAlignment, msgCombined)
    BLEConnection.gatts_indicate(connHandleMaster, charAlignment)
    return


time.sleep(2) # boot delay to avoid sending serial data to
            # secondary ESP32 before it is booted

uart = UART(1) # construct instance of UART class for intraplatform communications
uart.init(baudrate=115200, tx=2, rx=8) # configure baudrate and assign transmit pins
BLEConnection = ubluetooth.BLE() # construct instance of Bluetooth class
BLEConnection.active(True) # set the Bluetooth radio to active
BLEConnection.irq(bleIrqHandler) # configure interrupt handler
```

```
# Bluetooth configuration options
BLEConnection.config(rxbuf=50)


# Register Bluetooth service and characteristic
serviceUUID = ubluetooth.UUID(UUID_SERVICE_SLAVE)
charSlaveNumber = (ubluetooth.UUID(UUID_CHAR_SLAVENUMBER),
                   ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
charAlignment = (ubluetooth.UUID(UUID_CHAR_ALIGNMENT),
                   ubluetooth.FLAG_READ | ubluetooth.FLAG_WRITE | ubluetooth.FLAG_NOTIFY |
ubluetooth.FLAG_INDICATE,)
serviceTuple = (serviceUUID, (charSlaveNumber, charAlignment,),)
allServices = (serviceTuple,)
( (charSlaveNumber, charAlignment,), ) = BLEConnection.gatts_register_services(allServices)


# Write value of the slave's calibration/center offset to local characteristic
BLEConnection.gatts_write(charSlaveNumber, str(slaveNumber))
BLEConnection.gatts_write(charSlaveNumber, str(slaveNumber))


# Setup PWM pins
pwm_fl = PWM(Pin(5), freq=5000, duty=0) # Front left motor
pwm_fr = PWM(Pin(6), freq=5000, duty=0) # Front right motor
pwm_rl = PWM(Pin(7), freq=5000, duty=0) # Rear left motor
pwm_rr = PWM(Pin(10), freq=5000, duty=0) # Rear right motor


# Set initial PWM signals to 0
pwm_fl.duty(0)
pwm_fr.duty(0)
pwm_rl.duty(0)
pwm_rr.duty(0)

A1 = ADC(Pin(0,mode=Pin.IN))
A1.atten(ADC.ATTN_11DB)
A2 = ADC(Pin(1,mode=Pin.IN))
A2.atten(ADC.ATTN_11DB)

# Start watchdog monitor
wdt = WDT(timeout=4000)

print("\n\n**********")
print("FIRMWARE FOR CLIENT (VERSION 1.3.0)")
print("BOOT COMPLETE")
```

```
print("***********")

print("\n\n***********")
print("SCANNING FOR MASTER ESP32")
print("***********")
BLEConnection.gap_scan(0)


while True:
  slaveNumber = int(BLEConnection.gatts_read(charSlaveNumber))
  #print('1')

  while uart.any() > 0:
    msg = uart.readline()
    #print("Message: " + str(msg))
  #print('2')
  wdt.feed() # Feed watchdog timer
  if slaveNumber > -1: # Indicates the platform is connected to a master, so start checking/transmitting alignment
values
    #print('3')
    #if lastSlaveNumber == slaveNumber:
    Readalignment()
    #else:
    #  lastSlaveNumber = slaveNumber
    #  time.sleep(1) # Delay to avoid attempting to indicate before characteristic is fully configured

  #print('4')
  wdt.feed() # Feed watchdog timer
  time.sleep(.3)
```

5.3.5 App Code

In the table 29 the app code can be seen. The functionality of the app will connect to the server being hosted on the master platform ESP32 server and then write to certain characteristics to update platform speed and direction. The app will first start off by scanning all available Bluetooth devices to connect to. When the app has found the Bluetooth device advertising as ESP32 it will then attempt to connect to this peripheral. Once connected the app will search for all services within the peripheral and then search for the speed and motor direction characteristics by their CBUUID. The buttons on the app will then write to the discovered

characteristics when pressed. Finally, if the app is disconnected at any point, then the peripheral search process will begin again and reestablish a connection. [JS]

Table 29: Implemented App Code

```
//
//  ViewController.swift
//  App2
//
//  Created by Juan Soto on 12/21/22.
//

import
import
class ParticlePeripheral  NSObject
   public static let id    CBUUID init string  "4FAFC201-1FB5-459E-8FCC-C5C9C331914B"
   public static let charuuid    CBUUID string  "beb5483e-36e1-4688-b7f5-ea07361b26a8"
   public static let speeduuid    CBUUID string  "3954a328-b182-4c2e-a9c0-ad8535c5a30b"


class ViewController  UIViewController  CBCentralManagerDelegate  CBPeripheralDelegate
   var manager  CBCentralManager
   var peripheral  CBPeripheral
   var connectedPeripheral  CBPeripheral
   var writableCharacteristic  CBCharacteristic
   var writableSpeedCharacteristic  CBCharacteristic
   var speed String
   var value String
   var timer    Timer

   @IBOutlet weak var test  UITextField
   @IBOutlet weak var speedtxt  UILabel


   override func viewDidLoad
      Thread sleep forTimeInterval  1.0
      super viewDidLoad
      speed    "50"
      manager    CBCentralManager delegate  self  queue                    global




   func centralManagerDidUpdateState _          CBCentralManager

      switch          state


      case  poweredOn
         print "on"
         manager scanForPeripherals withServices nil
```

```
    case resetting
        print "resetting"
    case unsupported
        print "unsupported"
    case unauthorized
        print "unauthorized"
    case poweredOff
        print "off"

    case unknown
        print "UNKONWN"
    @unknown default
        print "default"



  func centralManager _          CBCentralManager didDiscover              CBPeripheral advertisementData
String  Any  rssi       NSNumber

    let                      name



    //print("Peripheral Discovered: \(peripheral)")

    if              "ESP32"

        self connectedPeripheral

    //self.peripheral.delegate = self
        print "Peripheral Discovered:                "
        print "Peripheral name:                name "
        print  "Advertisement Data :                          "
        manager connect              options  nil




  func centralManager _          CBCentralManager didFailToConnect              CBPeripheral error Error
    //let alert =  UIAlertController(title: "title", message: "message", preferredStyle: .alert)
    //self.present(alert, animated: true, completion: nil)


  func centralManager _          CBCentralManager didConnect              CBPeripheral

    self connectedPeripheral
    self connectedPeripheral delegate    self
    print "connected to:                "
                discoverServices  ParticlePeripheral id
```

```
    func peripheral _                CBPeripheral  didDiscoverServices          Error
        //print(peripheral.services)

        if let                              services
            for             in
                if             uuid        ParticlePeripheral id
                    print "found service:                    services "
                    //Now kick off discovery of characteristics
                                discoverCharacteristics nil  for




    func peripheral _                CBPeripheral  didDiscoverCharacteristicsFor           CBService  error  Error
        guard let                                  characteristics else
            return


        for                    in
            //Setting the direction characterisitc to write to
            if                    uuid        ParticlePeripheral charuuid

                writableCharacteristic

            //Setting the speed characterisitic to write to
            else if                    uuid        ParticlePeripheral speeduuid

                writableSpeedCharacteristic



    func centralManager _            CBCentralManager  didDisconnectPeripheral                CBPeripheral  error
Error
        manager scanForPeripherals withServices nil


    func peripheral _                CBPeripheral  didUpdateValueFor                  CBCharacteristic  error  Error
        let             String data                     value   encoding   utf8
        print



    func senddatadirection direction  String  speedinfo  String

        let                             as NSString  data using  String Encoding utf8 rawValue
        let                               as NSString  data using  String Encoding utf8 rawValue
        connectedPeripheral  writeValue                for  writableCharacteristic   type   withResponse
        connectedPeripheral  writeValue                  for  writableSpeedCharacteristic   type   withResponse

    func updatespeed speedinfo  String

        let                               as NSString  data using  String Encoding utf8 rawValue
        connectedPeripheral  writeValue                for  writableSpeedCharacteristic   type   withResponse
```

```
@IBAction func speedslider _            UISlider

   speedtxt text    String Int          value
   speed    speedtxt text



//Checking to see if the any button is pressed.

@IBAction func Up _             UIButton
   senddatadirection direction "up"  speedinfo  speed


@IBAction func down _           UIButton
   senddatadirection direction "down"  speedinfo  speed


@IBAction func left _           UIButton
   senddatadirection direction  "left"  speedinfo  speed


@IBAction func right _          UIButton
   senddatadirection direction  "right"  speedinfo  speed


@IBAction func topleft _        UIButton
   senddatadirection direction  "top left"  speedinfo  speed


@IBAction func topright _       UIButton
   senddatadirection direction  "top right"  speedinfo  speed


@IBAction func bottomleft _         UIButton
   senddatadirection direction  "bottom left"  speedinfo  speed


@IBAction func bottomright _        UIButton
   senddatadirection direction  "bottom right"  speedinfo  speed


@IBAction func turnleft _       UIButton
   senddatadirection direction  "turn left"  speedinfo  speed


@IBAction func turnright _      UIButton
   senddatadirection direction  "turn right"  speedinfo  speed


@IBAction func stop _           UIButton
   senddatadirection direction  "stop"  speedinfo  speed
```

6. Mechanical Sketch

In the following Figures 23, 24, 25, and 26 the final design of the furniture mover platforms can be seen. Figure 23 shows the overall final design of one furniture mover platform. Figure 24 shows the internals of one platform. Figures 25 and 26 also show the final design for the furniture leg holder that may be implemented on the furniture mover platform. [JS]

Figure 23: Furniture Mover Platform

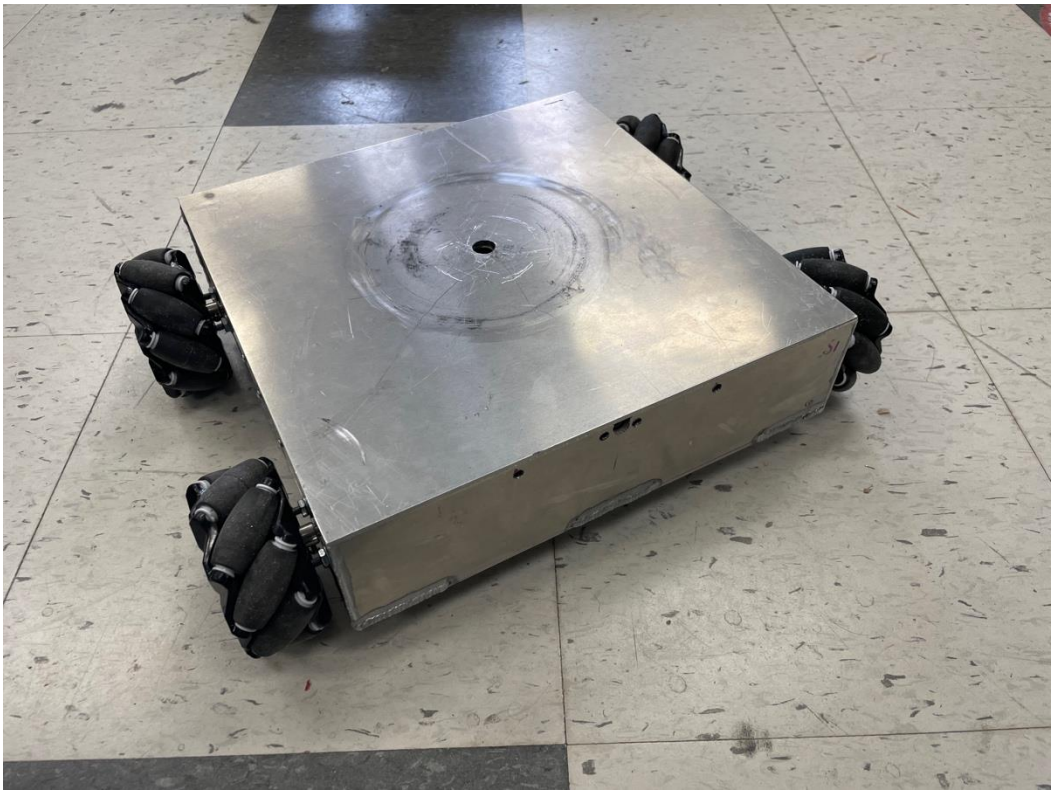


Figure 24: Internals of Furniture Mover Platform

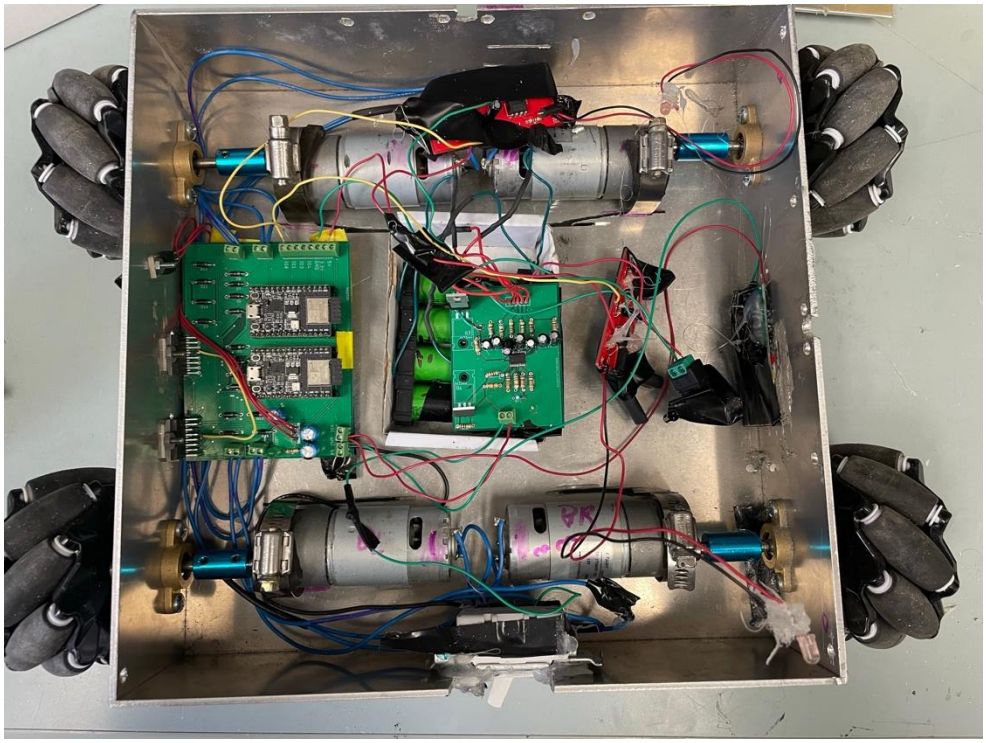
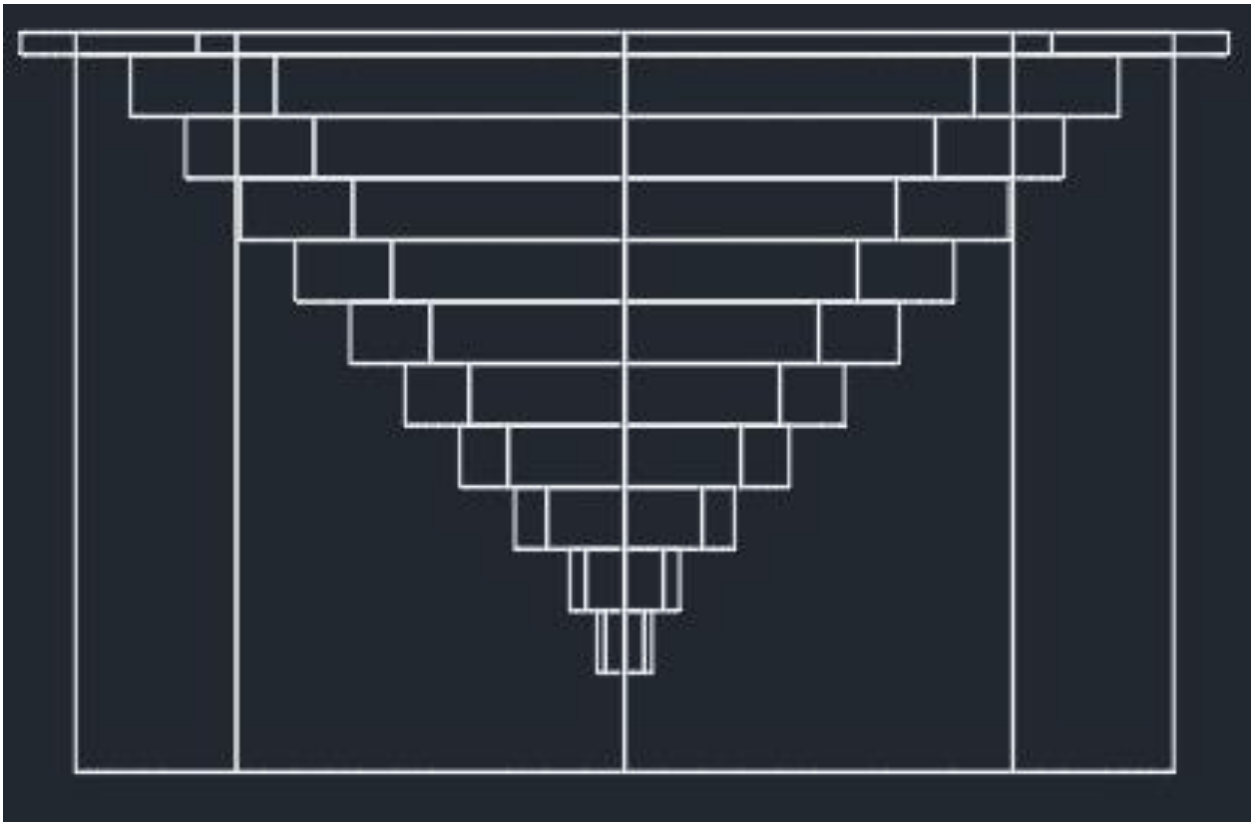


Figure 25: Stepped Holder

Figure 26: Internal Design of Stepped Holder



7. Parts List

7.1 Parts List

Table 30 below represents the parts used in and corresponding to the schematics in the Accepted Technical Design. [ZB]

Table 30: Accepted Technical Design Parts List

Qty.	Refdes	Part Num.	Description
14			MOSFET MOSFT 100V 33A 44mOhm 47.3nC
2			Battery Management 3-Series to 5-Series Stackable Ultra-Low-Power Primary Protector With Autonomous Cell Balancing 24-TSSOP -40 to 85
4			NTC Thermistors 10kohm 1%
4			Current Sense Resistors - SMD 1/4watt 0.001ohm 1%
8			LG MJ1 18650 3500mAh 10A Battery
3		MF-2P	12V DC Power Connector 5.5mm x 2.1mm 24V Power Jack Plug Barrel Adapter for CCTV Security Camera Led Strip Light (2 x Male + 2 x Female)
1			Alignment PCB
32			Wirewound Resistors - Through Hole 2W 25 ohms 1%
32		TL081	IC, OP-AMP, General Purpose JFET, High Slew
16		QED123	OPTO, LED, Infared, 880nm, AlGaAs, T-1 3/4
85		QSD123	6mA photo trans
20			1uF, 50V Capacitor
24			Aluminum Electrolytic Capacitors - Radial Leaded 220uF 25V 8x11,5mm 85 C 2500h
1			4 ft by 4 ft 6061 Aluminum sheet
70			1N4007 Diode
10			L298N Chips
1			Nickel Strip
16		--	TSINY Small 12v DC 100 RPM Spur High Torque Gear Box Electric Motor
4		--	ESP32-C3-DevKitM-1 Development Board
4		--	97mm Mecanum Wheel + 6mm Motor Coupling Motor Connector Kit
16		--	Twidec/2Pcs 6mm to 6mm Bore Flexible Shaft Coupling
1			Multipurpose 304 Stainless Steel Rod 3ft
4		--	CFsunbird 3D Printer wholesales 5pcs 6mm Caliber zinc Alloy Bearing
4		30496	220 uF 25 V Capacitor
32			30V RF Diode
14			Sanyo NCR18650GA 3500mAh 10A Battery

7.2 Materials Budget

In order to complete the impletmentation of the project, certain parts were required to be bought in order to implment certain functionalities. Additionally, a record of project expenses and budget had to be tracked. These subsystems included the power charging, motor driver, and microcontroller. Below in Tables 31, 32, 33, and 34 is a list of components that were purchased for the project implementation. [ZB, JS]

Table 31: Parts Request Order Form 1

			Unit	Total
Qty.	**Part Num.**	**Description**	**Cost**	**Cost**
2	RN4871-V	Microchip RN4871 Click Bluetooth Module	$33.00	$66.00
1	--	HiLetgo ESP-WROOM-32 ESP32 ESP-32S Development Board	$10.99	$10.99
1	--	Greartisan DC 12V 100RPM Gear Motor High Torque Electric Micro Speed Reduction	$14.99	$14.99
1	--	L298N Motor Driver Controller Board Module Stepper Motor DC	$11.49	$11.49
1	J302-1682000UX	16.8V 2A Charger, AC 100-240V DC 16.8V 1A 2A Replacement Power Supply Adapter	$14.23	$14.23
1	--	AITRIP 3 pcs 18650 Li-ion Lithium Battery Protection Board 25A 4S 16.8V BMS Charging Module	$9.99	$9.99
1	--	50PCS 18650 Lithium Battery Double Holder Bracket	$8.99	$8.99
2	--	18650 3.7V 3600mAh Li-ion Rechargeable Battery (2pcs)	$14.95	$29.90
1	MF-2P	12V DC Power Connector 5.5mm x 2.1mm 24V Power Jack Plug Barrel Adapter for CCTV Security Camera Led Strip Light (2 x Male + 2 x Female)	$4.99	$4.99

Table 32: Parts Request Order Form 2

			Unit	Total
Qty.	**Part Num.**	**Description**	**Cost**	**Cost**
16	--	TSINY Small 12v DC 100 RPM Spur High Torque Gear Box Electric Motor	$17.88	$286.08
4	--	ESP32-C3-DevKitM-1 Development Board	$8.00	$32.00

4	--	97mm Mecanum Wheel + 6mm Motor Coupling Motor Connector Kit	$39.99	$159.96
16	--	Twidec/2Pcs 6mm to 6mm Bore Flexible Shaft Coupling	$7.99	$127.84
1		Multipurpose 304 Stainless Steel Rod 3ft	$19.10	$19.10
4	--	CFsunbird 3D Printer wholesales 5pcs 6mm Caliber zinc Alloy Bearing	$16.10	$64.40
4	30496	220 uF 25 V Capacitor	$0.00	$0.00
32		30V RF Diode	$0.00	$0.00
14		Sanyo NCR18650GA 3500mAh 10A Battery	$5.50	$77.00

Table 33: Parts Request Order Form 3

			Unit	Total
Qty.	Part Num.	Description	Cost	Cost
10	--	TSINY Small 12v DC 100 RPM Spur High Torque Gear Box Electric Motor	$17.88	$178.80
2	--	2 Pack 6061 T651 Aluminum Sheet Metal 12x12x1/8	$25.99	$51.98
2	--	97mm Mecanum Wheel + 4mm Motor Coupling Motor Connector Kit	$38.99	$77.98
4	--	Twidec/2Pcs 6mm to 6mm Bore Flexible Shaft Coupling	$7.99	$31.96
2	--	CFsunbird 3D Printer wholesales 5pcs 6mm Caliber zinc Alloy Bearing	$16.00	$32.00

Table 34: Parts Request Order Form 4

			Unit	Total
Qty.	Part Num.	Description	Cost	Cost
14		MOSFET MOSFT 100V 33A 44mOhm 47.3nC	$1.35	$18.90
2		Battery Management 3-Series to 5-Series Stackable Ultra-Low-Power Primary Protector With Autonomous Cell Balancing 24-TSSOP -40 to 85	$2.16	$4.32
4		NTC Thermistors 10kohm 1%	$0.96	$3.84
4		Current Sense Resistors - SMD 1/4watt 0.001ohm 1%	$1.43	$5.72
8		LG MJ1 18650 3500mAh 10A Battery	$5.99	$47.92
3	MF-2P	12V DC Power Connector 5.5mm x 2.1mm 24V Power Jack Plug Barrel Adapter for CCTV Security Camera Led Strip Light (2 x Male + 2 x Female)	$4.99	$14.97
1		Alignment PCB	$6.20	$6.20
32		Wirewound Resistors - Through Hole 2W 25 ohms 1%	$1.34	$42.88
32	TL081	IC, OP-AMP, General Purpose JFET, High Slew		$0.00
16	QED123	OPTO, LED, Infared, 880nm, AlGaAs, T-1 3/4		$0.00

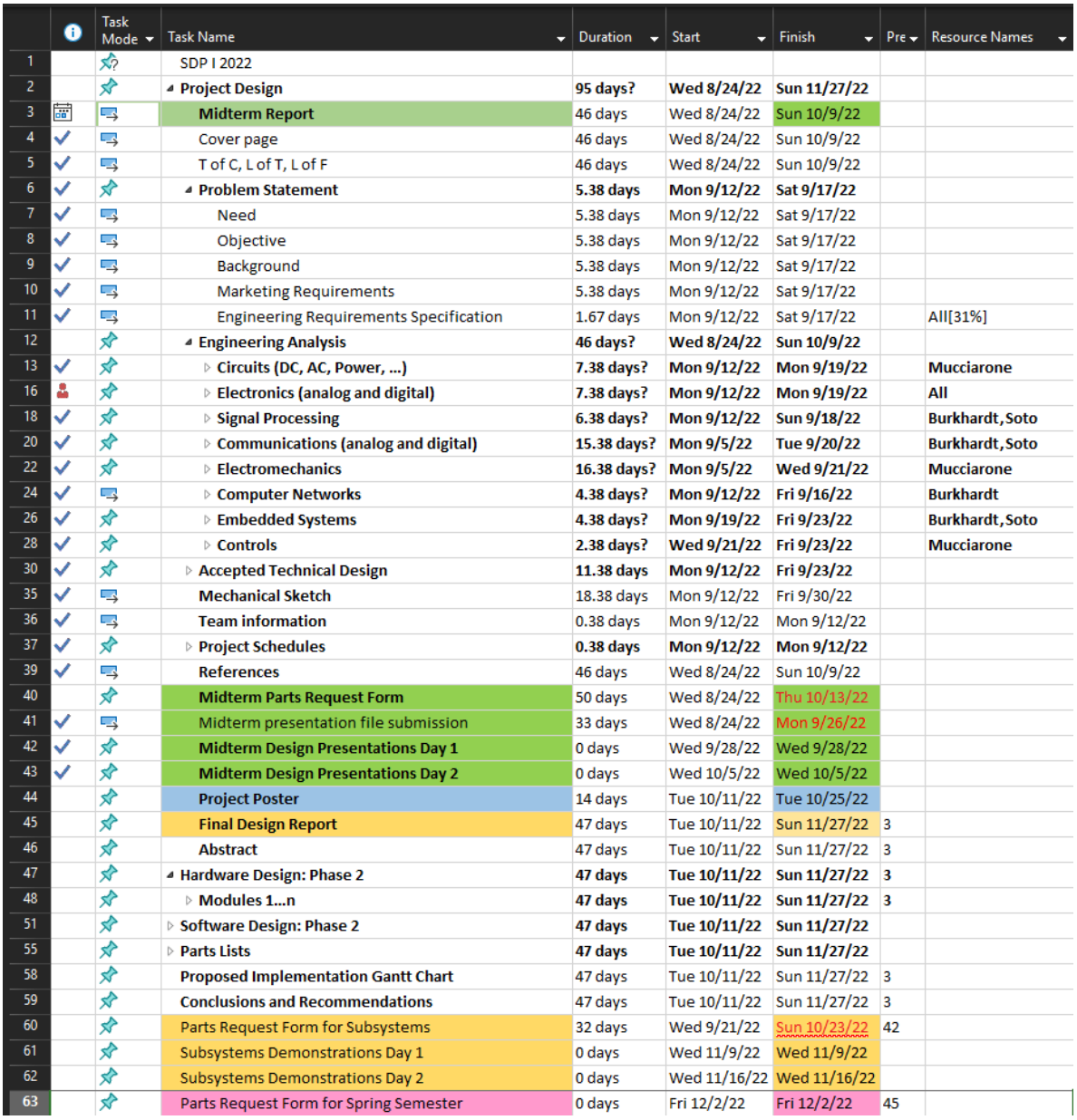
85	QSD123	6mA photo trans		$0.00
20		1uF, 50V Capacitor		$0.00
24		Aluminum Electrolytic Capacitors - Radial Leaded 220uF 25V 8x11,5mm 85 C 2500h	$0.48	$11.54
1		4 ft by 4 ft 6061 Aluminum sheet	$273.68	$273.68
70		1N4007 Diode		$0.00
10		L298N Chips	$10.81	$108.10
1		Nickel Strip		$0.00

8. Project Schedules

8.1 Midterm Design Gantt Chart

The following Figure 22 illustrates the project research subsections deadlines and each individual's responsibilities. [JS]

Figure 22: Midterm Gantt Chart



	Task Mode	Task Name	Duration	Start	Finish	Pre	Resource Names
1		SDP I 2022					
2		**Project Design**	**95 days?**	**Wed 8/24/22**	**Sun 11/27/22**		
3		**Midterm Report**	46 days	Wed 8/24/22	Sun 10/9/22		
4		Cover page	46 days	Wed 8/24/22	Sun 10/9/22		
5		T of C, L of T, L of F	46 days	Wed 8/24/22	Sun 10/9/22		
6		**Problem Statement**	**5.38 days**	**Mon 9/12/22**	**Sat 9/17/22**		
7		Need	5.38 days	Mon 9/12/22	Sat 9/17/22		
8		Objective	5.38 days	Mon 9/12/22	Sat 9/17/22		
9		Background	5.38 days	Mon 9/12/22	Sat 9/17/22		
10		Marketing Requirements	5.38 days	Mon 9/12/22	Sat 9/17/22		
11		Engineering Requirements Specification	1.67 days	Mon 9/12/22	Sat 9/17/22		All[31%]
12		**Engineering Analysis**	**46 days?**	**Wed 8/24/22**	**Sun 10/9/22**		
13		**Circuits (DC, AC, Power, ...)**	**7.38 days?**	**Mon 9/12/22**	**Mon 9/19/22**		**Mucciarone**
16		**Electronics (analog and digital)**	**7.38 days?**	**Mon 9/12/22**	**Mon 9/19/22**		**All**
18		**Signal Processing**	**6.38 days?**	**Mon 9/12/22**	**Sun 9/18/22**		**Burkhardt,Soto**
20		**Communications (analog and digital)**	**15.38 days?**	**Mon 9/5/22**	**Tue 9/20/22**		**Burkhardt,Soto**
22		**Electromechanics**	**16.38 days?**	**Mon 9/5/22**	**Wed 9/21/22**		**Mucciarone**
24		**Computer Networks**	**4.38 days?**	**Mon 9/12/22**	**Fri 9/16/22**		**Burkhardt**
26		**Embedded Systems**	**4.38 days?**	**Mon 9/19/22**	**Fri 9/23/22**		**Burkhardt,Soto**
28		**Controls**	**2.38 days?**	**Wed 9/21/22**	**Fri 9/23/22**		**Mucciarone**
30		**Accepted Technical Design**	**11.38 days**	**Mon 9/12/22**	**Fri 9/23/22**		
35		**Mechanical Sketch**	18.38 days	Mon 9/12/22	Fri 9/30/22		
36		**Team information**	0.38 days	Mon 9/12/22	Mon 9/12/22		
37		**Project Schedules**	**0.38 days**	**Mon 9/12/22**	**Mon 9/12/22**		
39		**References**	46 days	Wed 8/24/22	Sun 10/9/22		
40		**Midterm Parts Request Form**	50 days	Wed 8/24/22	Thu 10/13/22		
41		Midterm presentation file submission	33 days	Wed 8/24/22	Mon 9/26/22		
42		**Midterm Design Presentations Day 1**	0 days	Wed 9/28/22	Wed 9/28/22		
43		**Midterm Design Presentations Day 2**	0 days	Wed 10/5/22	Wed 10/5/22		
44		**Project Poster**	14 days	Tue 10/11/22	Tue 10/25/22		
45		**Final Design Report**	47 days	Tue 10/11/22	Sun 11/27/22	3	
46		**Abstract**	47 days	Tue 10/11/22	Sun 11/27/22	3	
47		**Hardware Design: Phase 2**	**47 days**	**Tue 10/11/22**	**Sun 11/27/22**	**3**	
48		**Modules 1...n**	**47 days**	**Tue 10/11/22**	**Sun 11/27/22**	**3**	
51		**Software Design: Phase 2**	**47 days**	**Tue 10/11/22**	**Sun 11/27/22**		
55		**Parts Lists**	**47 days**	**Tue 10/11/22**	**Sun 11/27/22**		
58		**Proposed Implementation Gantt Chart**	47 days	Tue 10/11/22	Sun 11/27/22	3	
59		**Conclusions and Recommendations**	47 days	Tue 10/11/22	Sun 11/27/22	3	
60		Parts Request Form for Subsystems	32 days	Wed 9/21/22	Sun 10/23/22	42	
61		Subsystems Demonstrations Day 1	0 days	Wed 11/9/22	Wed 11/9/22		
62		Subsystems Demonstrations Day 2	0 days	Wed 11/16/22	Wed 11/16/22		
63		Parts Request Form for Spring Semester	0 days	Fri 12/2/22	Fri 12/2/22	45	

8.2 Final Design Gantt Chart

The following Figure 23 illustrates the project implementation deadlines and each individual's responsibilities. [ZB]

Figure 23: Final Gantt Chart

	Task Mode	Task Name	Duration	Start	Finish	Predecessors	Resource Names
1		**SDP2 Implementation 2023**	**453 days**	**Mon 1/9/23**	**Fri 4/5/24**		
2		Revise Gantt Chart	14 days	**Mon 1/9/23**	Sun 1/22/23		
3		**Implement Project Design**	**92 days?**	**Mon 1/9/23**	**Mon 4/10/23**		
4		**Hardware Implementation**	**43 days?**	**Mon 1/9/23**	**Tue 2/21/23**		
5		**Breadboard Components**	**14 days**	**Mon 1/9/23**	**Sun 1/22/23**		
6		Serial/Bluetooth Communication	14 days	Mon 1/9/23	Sun 1/22/23		Zach Burkhardt
7		Alignment Sensors	14 days	Mon 1/9/23	Sun 1/22/23		Juan Soto
8		**Layout and Generate PCB(s)**	**14 days**	**Mon 1/9/23**	**Sun 1/22/23**		
9		Power Management	14 days	Mon 1/9/23	Sun 1/22/23		Gino Mucciarone
10		Microcontroller Shield	14 days	Mon 1/9/23	Sun 1/22/23		Zach Burkhardt,Gino Mucciarone
11		Motor Driver	14 days	Mon 1/9/23	Sun 1/22/23		David Kotyk
12		**Assemble Hardware**	**28 days**	**Mon 1/9/23**	**Sun 2/5/23**	**5,8**	
13		Design Frame Topper	28 days	Mon 1/9/23	Sun 2/5/23		Juan Soto
14		**Test Hardware**	**7 days**	**Mon 2/6/23**	**Sun 2/12/23**	**12**	
15		Weight Test (25 lbs)	7 days	Mon 2/6/23	Sun 2/12/23		Juan Soto
16		Alignment Test	7 days	Mon 2/6/23	Sun 2/12/23		David Kotyk,Juan Soto
17		Battery Test (45 mins)	7 days	Mon 2/6/23	Sun 2/12/23		Gino Mucciarone
18		**Revise Hardware**	**7 days**	**Mon 2/13/23**	**Sun 2/19/23**	**14**	
19		Alignment Sensor Placement	7 days	Mon 2/13/23	Sun 2/19/23		David Kotyk
20		Motor Driver/Microcontroller Placement (Avoid Heat)	7 days	Mon 2/13/23	Sun 2/19/23		Zach Burkhardt
21		*MIDTERM: Demonstrate Hardware Subs*	0 days	Tue 2/21/23	Tue 2/21/23		
22		**Software Implementation**	**43 days**	**Mon 1/9/23**	**Tue 2/21/23**		
39		**System Integration**	**49 days**	**Tue 2/21/23**	**Mon 4/10/23**		
40		**Assemble Complete System Integration**	**14 days**	**Tue 2/21/23**	**Mon 3/6/23**	**38**	
41		Connect Motors and Wheels	14 days	Tue 2/21/23	Mon 3/6/23	38	David Kotyk
42		Connect Alignment Sensors	14 days	Tue 2/21/23	Mon 3/6/23	38	Juan Soto
43		Connect Power	14 days	Tue 2/21/23	Mon 3/6/23	38	Gino Mucciarone
44		Connect Microcontrollers	14 days	Tue 2/21/23	Mon 3/6/23	38	Juan Soto
45		**Test Complete System Integration**	**7 days**	**Tue 3/7/23**	**Mon 3/13/23**	**40**	
46		Platform Movement Syncronization	7 days	Tue 3/7/23	Mon 3/13/23	40	Juan Soto,Zach Burkhardt
47		Battery Lifespan	7 days	Tue 3/7/23	Mon 3/13/23	40	Gino Mucciarone
48		Weight (25 lbs)	7 days	Tue 3/7/23	Mon 3/13/23	40	Juan Soto
49		Alignment Sensors	7 days	Tue 3/7/23	Mon 3/13/23	40	David Kotyk,Juan Soto
50		**Revise Complete System Integration**	**24 days**	**Tue 3/14/23**	**Thu 4/6/23**	**45**	
51		Alignment Algorithm Adjustment	24 days	Tue 3/14/23	Thu 4/6/23	45	Juan Soto,David Kotyk
52		Communication protocols	24 days	Tue 3/14/23	Thu 4/6/23	45	Zach Burkhardt
53		Battery Charge rate/ Drain time	24 days	Tue 3/14/23	Thu 4/6/23	45	Gino Mucciarone
54		*Preliminary Demonstration of Complete*	4 days	Fri 4/7/23	Mon 4/10/23		
55		**Develop Final Report**	**106 days**	**Mon 1/9/23**	**Mon 4/24/23**		
56		Write Final Report	106 days	Mon 1/9/23	Mon 4/24/23		All
57		Submit Final Report	0 days	Mon 4/24/23	Mon 4/24/23	56	Juan Soto
58		Spring Recess	7 days	Mon 3/20/23	Sun 3/26/23		
59		*Project Demonstration and Presentation*	0 days	Mon 4/17/23	Mon 4/17/23		All

9. Design Team Information

Zachariah Burkhardt, Computer Engineer

David Kotyk, Electrical Engineer

Gary Anthony Mucciarone, Electrical Engineer

Gino Vincent Mucciarone, Electrical Engineer

Juan Soto, Computer Engineer

10. Conclusions and Recommendations

At this time, the team has completed the design implementation of each major subsystem and integrated this system together. The system was demonstrated to the engineering design coordinator and other faculty members.

Due to constraints in the university sources used for welding, only two of the four platforms were able to be constructed. The remaining two platforms were built as stationary boxes with caster wheels, and all four platforms were connected via a rectangular frame. This design still allowed all subsystems to be demonstrated, including control algorithms for alignment.

Should this project be continued, the team would finish constructing two additional, independent platforms to complete the design. [ZB]

11. References

Afaneh, M. (2022, March 25). *Wireless connectivity options for IOT applications - technology comparison*. Bluetooth® Technology. Retrieved October 5, 2022, from https://www.bluetooth.com/blog/wireless-connectivity-options-for-iot-applications-technology-comparison/

Apple Inc. (n.d.). *About Swift*. Swift.org. Retrieved October 11, 2022, from https://www.swift.org/about/

Campbell, S. (2021, November 14). *Basics of the I2C communication protocol*. Circuit Basics. Retrieved October 9, 2022, from https://www.circuitbasics.com/basics-of-the-i2c-communication-protocol/

Division of Hazard Analysis, & Suchy, A., Product Instability or Tip-Over Injuries and Fatalities Associated with Televisions, Furniture, and Appliances: 2020 Report (2021). U.S. Consumer Product Safety Commission. Retrieved February 2022, from https://www.cpsc.gov/s3fs-public/2020_Tip_Over_Report.pdf?nhwAgmMt9YXGhkqfsN75hMCNYgBTe5OJ.

GeeksforGeeks. (2021, June 8). *Features of C Programming Language*. Retrieved October 11, 2022, from https://www.geeksforgeeks.org/features-of-c-programming-language/

Gudino, M. (2020, April 30). *Types of switching DC to DC converters*. Arrow.com. Retrieved October 11, 2022, from https://www.arrow.com/en/research-and-events/articles/types-of-switching-dc-dc-converters

Kiran, V. V., & Santhanalakshmi, S. (2019). Raspberry pi based remote controlled car using smartphone accelerometer. *2019 International Conference on Communication and Electronics Systems (ICCES)*. https://doi.org/10.1109/icces45898.2019.9002079

Lee, E. C., Choi, H. D., Kim, S. H., & Kwak, Y. K. (2008). Floor-types identification method for wheel robot using impedance variation. 2008 International Conference on Control, Automation and Systems. https://doi.org/10.1109/iccas.2008.4694197

Mcnamara, F. J. (1953). Dolly for handling furniture. *U.S. Patent No. 2627425*. Washington, DC: U.S. Patent and Trademark Office.

Parshakova, T., Cho, M., Cassinelli, A., & Saakes, D. (2017). Furniture that learns to move itself. *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. https://doi.org/10.1145/3027063.3049778

Peterson, D. (2022, March 9). *Understanding NPN vs PNP for sensors and devices - technical articles*. Control Automation. Retrieved October 11, 2022, from https://control.com/technical-articles/understanding-pnp-vs-npn-for-sensors-and-devices/

Proctor, B. (n.d.). *Mesh, star and point-to-point topology in IOT: Link labs*. Mesh, Star and Point-To-Point Topology In IoT | Link Labs. Retrieved October 8, 2022, from https://www.link-labs.com/blog/iot-topology

Ren, K. (2022, March 29). *Higher speedhow fast can it be?* Bluetooth® Technology. Retrieved October 6, 2022, from https://www.bluetooth.com/blog/exploring-bluetooth-5-how-fast-can-it-be/

Salter, J. (2022, February 3). *How much does a couch actually weigh?* Home Of Cozy. Retrieved March 13, 2022, from https://homeofcozy.com/guides/average-couch-weight/

Suzuki, R., Hedayati, H., Zheng, C., Bohn, J. L., Szafir, D., Do, E. Y.-L., Gross, M. D., & Leithinger, D. (2020). Roomshift: Room-scale dynamic haptics for VR with furniture-moving swarm robots. *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. https://doi.org/10.1145/3313831.3376523

Symiczek, D. (2009). Automotive dolly system. *U.S. Patent No. 7543830*. Washington, DC: U.S. Patent and Trademark Office.

Tsoulkas, V. (2010). Networked control systems with delay [tutorial]. *2010 2nd International Conference on Computational Intelligence, Communication Systems and Networks*. https://doi.org/10.1109/cicsyn.2010.78

Ueno, Y., Watanabe, K., & Nagai, I. (2017). Design and development of steered active wheel casters and its application. 2017 IEEE International Conference on Mechatronics and Automation (ICMA). https://doi.org/10.1109/icma.2017.8015869

Yida. (2022, September 13). *Uart vs I2C VS SPI – Communication Protocols and uses*. Seeed Studio. Retrieved October 10, 2022, from https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/

Zhang, H. (n.d.). *140: Basic concepts of linear regulator and switching mode power supplies*. AN-140: Basic Concepts of Linear Regulator and Switching Mode Power Supplies. Retrieved October 11, 2022, from https://www.analog.com/en/app-notes/an-140.html

12. Appendices

Appendix A – Code for Bluetooth Control Subsystem

```
/*
 * Senior Design Project - Team 13 - Proof of Concept
 * Zach Burkhardt (zmb14)
 * November 09, 2022
 * This program is used to demonstrate the feasibility of wireless inter-platform communications
*/

#include "mcc_generated_files/system.h"
#define RTS _RF13 // Output, For potential hardware handshaking.
#define CTS _RF12 // Input, For potential hardware handshaking.


void ms_delay(int ms) {
    T1CON = 0x8030; // Timer 1 On, prescale 1:256, Tcy as clock
    TMR1 = 0; // Clear TMR1
    while (TMR1 < ms * 62.5) {
    }
}

void InitU2(void) {
    U2BRG = 34; // PIC24FJ128GA010 data sheet, 17.1 for calculation, Fcy = 16MHz. Baud Rate = 115200
    U2MODE = 0x8008; // See data sheet, pg 148. Enable UART2, BRGH = 1,
    // Idle state = 1, 8 data, No parity, 1 Stop bit
    U2STA = 0x0400; // See data sheet, pg. 150, Transmit Enable
    // Following lines pertain Hardware handshaking
    TRISFbits.TRISF13 = 1; // enable RTS , output
    RTS = 1; // default status , not ready to send
}

char putU2(char c) {
    //    while (CTS); //wait for !CTS (active low)
    while (U2STAbits.UTXBF); // Wait if transmit buffer full.
    U2TXREG = c; // Write value to transmit FIFO
    return c;
}

char getU2(char c) {
     c = U2RXREG; // Write value to transmit FIFO
    return c;
}

unsigned int getButton(unsigned int mask) {
    unsigned int button;
    switch (mask) {
        case 0x0008: button = !PORTDbits.RD6; // S3 button
            break;
        case 0x0004: button = !PORTDbits.RD7; // S6 button
            break;
        case 0x0002: TRISAbits.TRISA7 = 1;
            button = !PORTAbits.RA7;
            break;
        case 0x0001: button = !PORTDbits.RD13;
```

```
            break;
        default:
            button = 0;
    }
    return (button);
}


int main(void) {
    // initialize the device
    SYSTEM_Initialize();
    InitU2();
    char str[80], temp = 'p';
    int i = 0;

    // config bits
    TRISA = 0x00; // set Port A to output
    _T1IP = 4; // this is the default value anyway
    TMR1 = 0; // clear the timer


    ms_delay(1000);
    ms_delay(1000);


    // Config Bluetooth module
    PORTA = 0x01;
    putU2(0x24); // $$$ (no \r)
    putU2(0x24); // Enter command mode
    putU2(0x24);
    ms_delay(500);

    PORTA = 0x01;
    putU2(0x53); // SS,C0\r
    putU2(0x53); // Set bitmap of services - support device info and UART transparent
    putU2(0x2C);
    putU2(0x43);
    putU2(0x30);
    putU2(0x0D); // Carriage return
    putU2(0x0A); // Line feed
    ms_delay(500);

    putU2(0x52); // R,1\r
    putU2(0x2C); // Restart Bluetooth module to confirm config change
    putU2(0x31);
    putU2(0x0D); // Carriage return
    putU2(0x0A); // Line feed
    ms_delay(1000);


    // START MASTER PLATFORM CODE
    PORTA = 0x01;
    putU2(0x24); // $$$ (no \r)
    putU2(0x24); // Enter command mode
    putU2(0x24);
    ms_delay(500);
```

```
PORTA = 0x03;
sprintf(str, "C,0,3481f406e298"); // Connect to slave Bluetooth module
for(i = 0; i < 16; ++i)
   putU2(str[i]);
putU2(0x0D); // Carriage return
putU2(0x0A); // Line feed
ms_delay(1000); // Wait for 4 seconds for connection to establish
ms_delay(1000);
ms_delay(1000);
ms_delay(1000);

PORTA = 0x07;
ms_delay(500);
// END MASTER PLATFORM CODE

//PORTA = 0x80; // MASTER
PORTA = 0x40; // SLAVE

while (1) {
   if (getButton(0x008)) {
      //PORTA = 0x8F; // MASTER
      PORTA = 0x4F; // SLAVE
      putU2(0x4C);
      while (getButton(0x008))
      {
         ms_delay(50); // Wait 50 ms before transmitting again
         putU2(0x4C); // Transmit character 'L' to instruct slave to turn on light
      }
      //PORTA = 0x80;  // MASTER
      PORTA = 0x40; // SLAVE
   }
   if (U2STAbits.URXDA) // If UART data available to read
   {
      for (i = 0; i < 80; ++i) // Fill string variable with '%' character to empty
      {
         str[i] = '%';
      }
      i = 0; // Reset iterator variable
      for (i = 0; U2STAbits.URXDA; ++i) // Read in UART data
      {
         str[i] = getU2(temp);
      }

      if (str[0] != '%') // If first character of string isn't the filler variable
      {
         //PORTA = 0x8F; // MASTER
         PORTA = 0x4F; // SLAVE
         ms_delay(75);

         while (U2STAbits.URXDA) // If more UART data available, read in again
         {
            i = 0;
            for (i = 0; U2STAbits.URXDA; ++i)
            {
               str[i] = getU2(temp);
            }
            ms_delay(75);
```

```
            }

            //PORTA = 0x80;  // MASTER
            PORTA = 0x40; // SLAVE
          }
        }
      }
    }
```