EG2310

Group 8

G2 Report

Student Team Members:

Chen Jiawei	A0258695H
Moey Sean Jean	A0255813A
Joey Lee Leyi	A0259107A
Benedict Chan	A0249460B

Table of Contents

1. Problem Definition

The main objective of this project is to develop a state-of-the-art automated drinks delivery system for a restaurant that seamlessly integrates a dispenser and a robot, creating a cohesive and efficient solution for delivering drinks to customers. The system needs to be able to work flawlessly in tandem, ensuring that the dispenser and robot are synchronised in their operations to deliver drinks in a timely and accurate manner.

1.1. Delivery Robot

The problem statement entails the development of a precise docking mechanism for the TurtleBot to receive a can from the dispenser, with a focus on accurately positioning the TurtleBot underneath the dispenser while accounting for factors such as alignment, stability, and reliability. Additionally, the TurtleBot must be equipped with sensors or buttons to detect the presence of the can after it has been received, triggering its departure from the dispenser.

The docking mechanism must be robust and reliable, capable of consistently and accurately aligning the TurtleBot with the dispenser to receive the can. Factors such as height, width, and position of the dispenser, as well as the size and shape of the can, must be taken into consideration during the design of the docking mechanism. Precise control over the movement and positioning of the TurtleBot is essential to ensure seamless docking without any misalignment. This can be achieved using optical sensors, weight sensors, buttons, or other appropriate detection mechanisms. The sensor data should be processed in real-time to trigger the departure of the TurtleBot from the dispenser, ensuring that it only departs when a can is successfully received.

1.2. Autonomous Navigation

The problem at hand is to develop a robust and efficient autonomous navigation system for the TurtleBot in the restaurant environment, with the main challenge being accurate obstacle detection and avoidance. The system needs to utilise advanced technologies such as Light Detection and Ranging (LiDAR) -based point cloud processing and onboard Inertial Measurement Unit (IMU)-based motion estimation to ensure precise navigation to different waypoints for delivering drinks to designated tables. The goal is to safely avoid

obstacles, accurately estimate the TurtleBot's current position, in order to achieve smooth and reliable autonomous delivery operations within the restaurant.

1.3. Dispenser

The dispenser system is a crucial component in the TurtleBot's operation, as it needs to reliably and efficiently dispense a drink can into the TurtleBot's designated receptacle upon returning to the dispenser. To achieve this, precise control over the dispensing mechanism is necessary to ensure that the drink can is accurately released without getting stuck or damaged halfway. The dispenser should only release a canned drink if the TurtleBot is deemed to have docked into the dispenser. The integrity of the drink can also has to be considered to prevent it from getting damaged halfway through dispensing. This can be done by considering the dispenser release mechanism or by adding some form of cushion into the TurtleBot's receptacle to lessen the impact on the canned drink.

Additionally, the dispenser system should also have a user-friendly interface, such as a set of buttons, which will be operated by a Teaching Assistant (TA) to select the table to deliver the can to. It is also essential that the dispenser system is designed for seamless integration with the TurtleBot, ensuring reliable communication between the two. FInally, the dispenser should fit within the designated 50cm by 50cm zone, with a maximum height of 100cm.

1.4. Communication

The communication protocol used between the dispenser and the TurtleBot should be reliable and efficient, ensuring that information is delivered in a timely manner and without loss of data. This may involve implementing appropriate Quality of Service (QoS) levels, handling message acknowledgements, and retransmission of lost messages to ensure reliable communication.

The communication protocol used should be optimised for the resources of the Microcontroller Unit (MCU) on the dispenser and Raspberry Pi (RPi) , considering the limited processing power, memory and bandwidth of these devices. This may involve optimising the communication protocol used, to minimise the overhead and reduce the impact on system performance.

Lastly, the protocol should be scalable and extensible, to allow for future enhancements or modifications. This is to mimic an actual restaurant setting where there could be multiple waiters. This may involve designing a flexible communication architecture that can accommodate changes in requirements, protocols or devices, and providing appropriate Application Programming Interfaces (APIs) for easy integration and customisation.

2. Literature Review

2.1. Delivery Robot

2.1.1. Sensor

For the dispenser to work seamlessly with the TurtleBot, it is crucial for the TurtleBot to accurately detect when a canned drink has dropped into the designated receptacle so that it would know when to start moving off and initiate delivery to the desired table. There are several methods to achieve this detection, including both mechanical and electronic approaches. The mechanical approach involves some form of physical contact between the TurtleBot and the drink can, while the electronic approach employs the use of sensors to detect the can's presence without any direct contact.

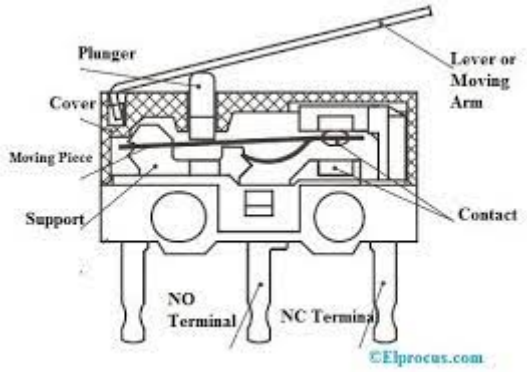


Figure 1

A mechanical approach to detect the presence of a drink can in the TurtleBot's receptacle is the use of a microswitch. This type of mechanical switch is triggered whenever the can makes contact with the TurtleBot's receptacle. As seen in Figure 1 above, the microswitch is able to detect the physical presence of a can by the act of it pressing against the switch's lever to touch the contact point, causing a change in logic level. It is important to

note that the positioning of the microswitch in the TurtleBot's receptacle is crucial as the canned drink is cylindrical in shape. Misplacement of the microswitch would mean that the microswitch will not be triggered when the canned drink falls into the TurtleBot's receptacle. Additionally, mechanical switch debouncing is also a concern that should be addressed should it be chosen as the method to be implemented. Mechanical switch debouncing occurs when the contacts of the microswitch rapidly make and break contact when it is pressed or released. This could lead to multiple triggers and false readings that we may not want. Nevertheless, implementing a microswitch is relatively simple and straightforward.

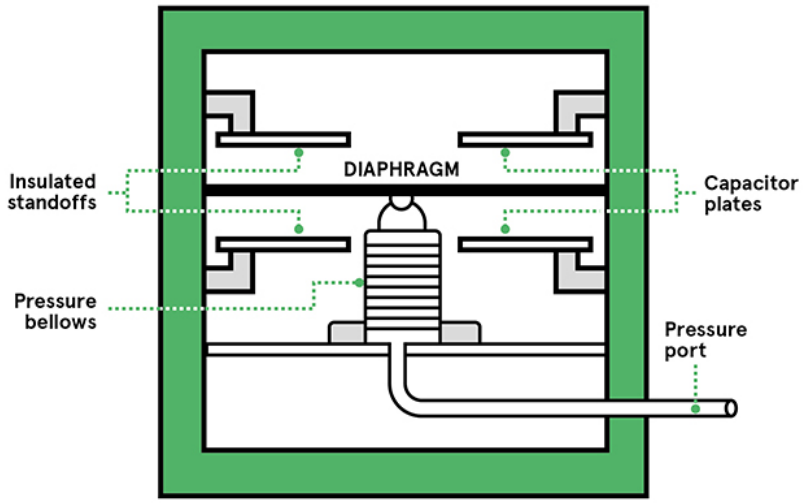


Figure: 2

A pressure sensor can also be used to detect the presence of a drink can in the TurtleBot's receptacle. The pressure sensor works by detecting a change in pressure and this can be achieved by placing the sensor at the bottom of the TurtleBot's receptacle. More specifically, a capacitive pressure sensor as shown in Figure 2 works based on the principle that a change in pressure alters the capacitance between two conductive surfaces. When the canned drink is dropped into the TurtleBot's receptacle, the distance between the two conductive surfaces will decrease, causing a change in the capacitance. This change is then converted into an electrical signal which can then be used as an indication of the canned drink. Despite its high accuracy and resolution to detect small changes in pressure, they are rather expensive to implement and require careful calibration to prevent false positive readings.

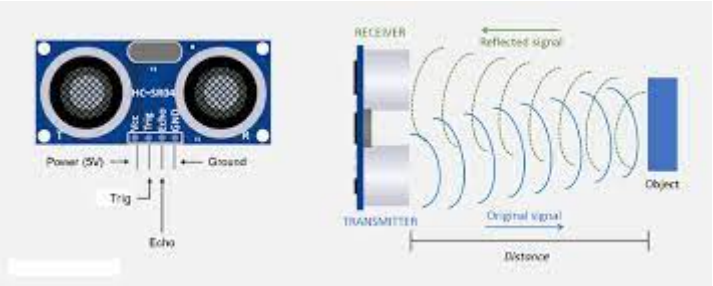


Figure 3

Ultrasonic sensors is another highly popular choice to detect the presence of the drink can in the TurtleBot's receptacle. Ultrasonic sensors work by emitting a high-frequency sound wave through the trigger pin and then measuring the time taken for the sound wave to be received by the echo pin. The distance between the ultrasonic sensor and object detected can then be derived. The ultrasonic sensor can be mounted above the receptacle to detect the can dropping/ passing by. Ultrasonic sensors are relatively inexpensive and can be used for a multitude of applications.

Considering the nature of our project and the mission objective, the use of a microswitch would be the most suitable for us. Since we only need to know when the can has dropped into the TurtleBot's receptacle, we do not have to implement such a cost heavy solution by using a capacitive pressure sensor. To add on, the ultrasonic sensor has a limited operating range and may not work accurately when that threshold is not respected. Considering that the ultrasonic sensor would have to be placed somewhere on the TurtleBot, we would have to design a large enough receptacle to accommodate for the limitations and ensure that the ultrasonic sensor would work reliably every single time. The microswitch is an easy to implement and cheap option that would work fine for the purpose of our project.

2.1.2. Docking

The docking system is a crucial aspect of the entire delivery system as it ensures that the TurtleBot is accurately positioned to dock in the dispenser. This is to make sure that the can will fall into the TurtleBot properly and not get stuck halfway. There are various hardware and software methods that can be implemented to achieve a consistent and precise docking.

The most straightforward method would be to design a physical barrier to guide the TurtleBot back into the dispenser. After the TurtleBot is deemed to have reached the

designated point, the angle of the shortest distance obtained from the LiDAR will then be used to rotate the TurtleBot to ensure that it faces the dispenser at a normal angle. However, the LDS-01 LiDAR unit onboard the TurtleBot has a minimum operating range of 12cm. The physical barrier used would have to slowly converge into the dispenser system and the distance between the walls used for the physical barrier would be narrow. As such, the LiDAR will not be accurate when docking and this may cause the TurtleBot to move in an unpredictable manner.

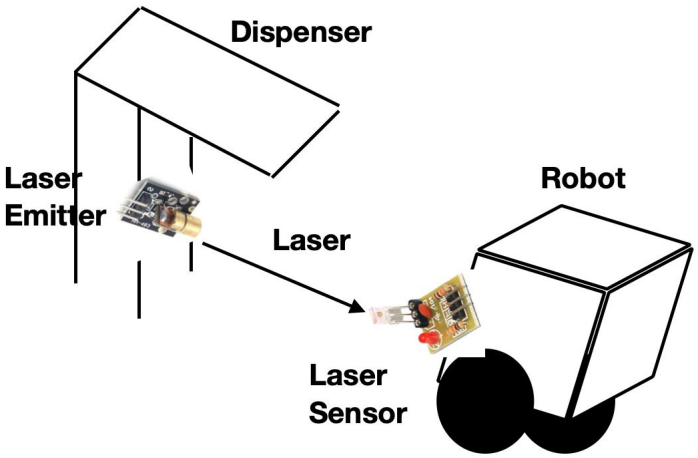


Figure 4

A more elaborate implementation using a laser emitter and laser sensor can be used to help guide the TurtleBot to dock accurately, as seen in figure 4 above. This involves emitting a thin and straight laser beam from a laser emitter positioned somewhere on the dispenser system. The TurtleBot will first have to navigate to a predetermined waypoint and rotate on the dime until the onboard laser sensor detects the laser beam. The TurtleBot will then move towards that direction and align itself with the laser beam until it docks. However, the intricacy of this design requires a clear line of sight between the laser emitter and laser sensor. Any obstacles or interference will prevent the TurtleBot from docking properly. Any minor shift in the positioning of the laser pointer or sensor could cause the entire system to fail.

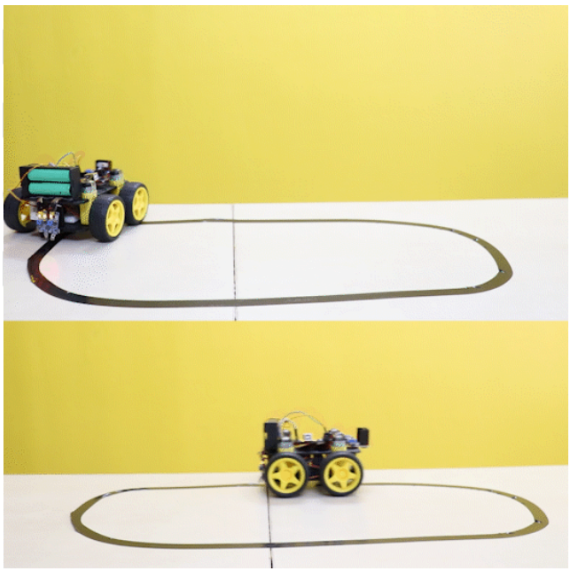


Figure 5

A common approach in the field of robotics is the use of Infrared Red (IR) sensors with a line-following algorithm to get to its destination. A similar approach can be used to aid in the docking of the TurtleBot by placing a strip of black line on the ground leading to the dispenser. Using two IR sensors, the robot will keep following the black line until it reaches the dispenser system and stops. Figures 6 and 7 below demonstrate how the TurtleBot will correct its path if it were to deviate in one direction. This way, the TurtleBot will keep tracking the black line until it reaches the end where both IR sensors would detect a black strip, indicating that it has reached its destination.

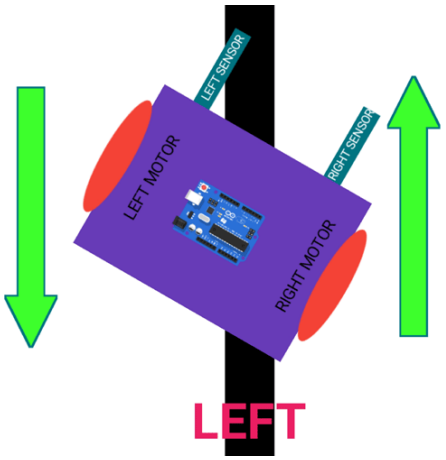


Figure 6

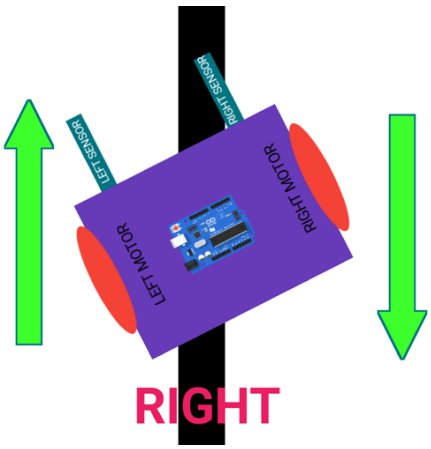


Figure 7

The implementation of IR sensors on the TurtleBot is the preferred choice amongst the three listed due to its simplicity in implementation and the consistency in its workings. Compared to the other two methods, the IR sensors along with the line-following algorithm does not require a high degree of calibration like the laser emitter and sensor, and also does not use a brute force method like the physical barrier to dock into the dispenser. The IR sensors are in a way almost foolproof if implemented correctly and will always ensure that the TurtleBot is docked precisely at where it is supposed to be.

2.2. Autonomous Navigation

The TurtleBot is tasked to deliver canned drinks to different tables autonomously and navigate the environment while avoiding collision. Manual navigation is time-consuming and prone to errors, which can lead to delays in service and customer dissatisfaction. To achieve autonomous navigation in the restaurant setting, the TurtleBot requires a robust collision avoidance system and a path planning algorithm to enable it to navigate to specific locations such as our tables. Herein, the path planning task explored is based on a two-wheel differential mobile robot. The robot can control the speed of its two driving wheels to achieve arbitrary trajectory movements such as linear movement, turning, and turning around in circles.

Path planning algorithms utilise various techniques to generate feasible paths, such as reinforcement learning algorithms, Simultaneous Localisation and Mapping (SLAM) algorithms, and potential field methods. These include the Potential Deep Q-Network

(PDQN), Vector Field Histogram (VFH) and HectorSLAM algorithms, which will be explored in subsequent sections.

2.2.1. PDQN

PDQN is a reinforcement learning algorithm that combines elements of both potential field methods and Q-learning, to enable robots to learn optimal path planning strategies. It builds upon the DQN algorithm which uses deep neural networks to learn Q-values which are used for making decisions in an agent's action selection process. The Q-value function can be represented as

$$Q(s, \alpha, \theta) \text{ ----- } (1)$$

where s represents the current state of the TurtleBot in the form of a vector; α represents the set of potential field parameters; and θ represents the parameters used in the deep neural network. Figure 8 illustrates the process by which the DQN algorithm utilises a neural network to approximate the value function. It showcases the iterative updating of potential field parameters during the neural network training until convergence is achieved, resulting in the formation of a final output.

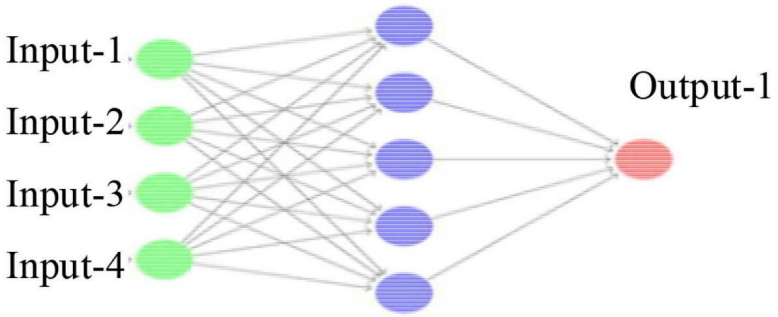


Figure 8

Lastly, the potential field method is used for goal-driven behaviour to guide the TurtleBot towards the table. This potential field can be calculated as

$$U(X) = \frac{1}{2}k(X - X_g)^2 \text{ ------ } (2)$$

where k is the gain coefficient, X is the current state of the TurtleBot and X_g is the goal position. The TurtleBot will then exploit the potential field to generate a gradient that points towards the direction of decreasing potential, and ultimately move towards the goal position.

With the aforementioned methods used, the TurtleBot would be able to learn optimal path planning strategies while incorporating goal-driven behaviour and adapting to dynamic environments.

2.2.2. Vector Field Histogram

VFH is a real-time obstacle avoidance technique used for path planning by dividing the space around the robot into a set of polar coordinates using a 2-dimensional Cartesian histogram grid. The range data gathered through the robot's onboard proximity sensors will update continuously to generate a vector field that indicates the direction of free space around the robot. This enables the TurtleBot to move in the direction of free space and explore its environment safely and efficiently, all while avoiding obstacles and following a planned path.

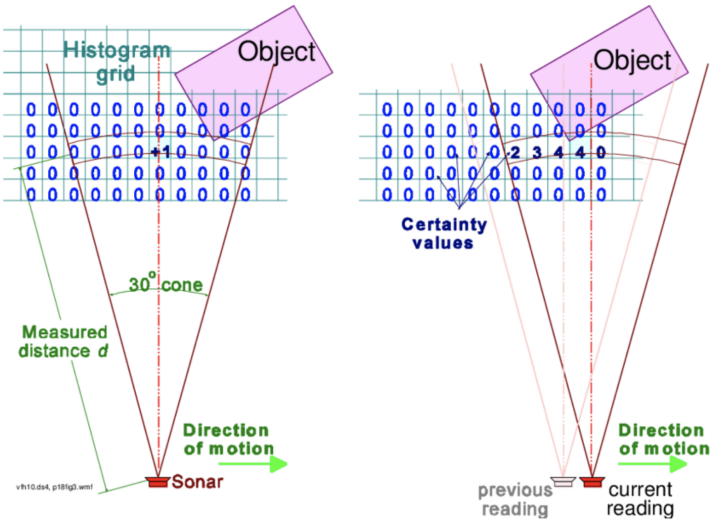


Figure 9

The VFH algorithm for autonomous navigation encompasses two primary phases. In the first phase, the 2-dimensional Cartesian histogram grid around the robot's current location is downsampled into a one-dimensional polar histogram. This histogram characterises the polar obstacle density in each section around the robot, providing a comprehensive view of the robot's environment, as illustrated in Figure 9. In the second phase, the algorithm selects the optimal direction for the robot to move in by identifying the section with the lowest polar obstacle density. The steering angle and the velocity controls from the 1-dimensional polar histogram will be computed and sent to the TurtleBot.

The vector field generated from the 1-dimensional polar histogram is susceptible to noise and possible introduction of local minimas in the vector field, causing the robot to be trapped in the map, resulting in suboptimal path planning. As such, a Gaussian filter is used to eliminate these errors by smoothening the vector field using a kernel to remove the high frequency components.

2.2.3. HectorSLAM

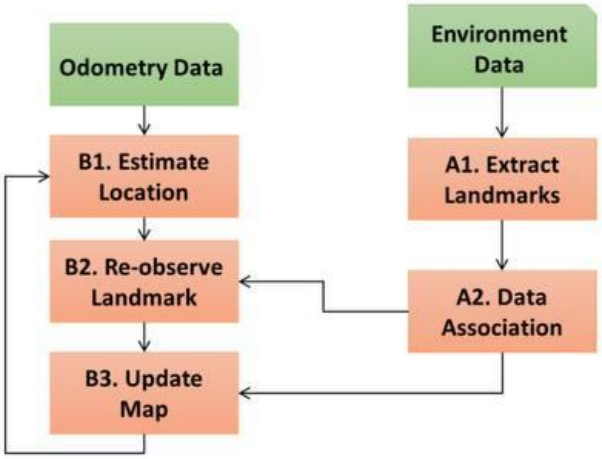


Figure 10

For our two-wheel differential mobile robot to navigate the restaurant, mapping and localisation are two of the main tasks involved. On their own, both indoor and outdoor localisation/ mapping is not an issue. However, it becomes much trickier when we need to address both localisation and mapping at the same time while the robot is moving. SLAM is a popular technique used in robotics to construct a map of an unknown environment while simultaneously keeping track of the robot's location. As seen in Figure 10 above, SLAM requires environment data such as prominent landmarks and also odometry data to associate the data with the landmarks picked up. This allows the robot to estimate its current location and build upon the current mapping that it has.

HectorSLAM is one of more widely used variants of SLAM as it does not require odometry data to work. Instead of using odometry data to estimate its location, it uses the new observation data acquired to match it with a known map in order to figure out its current position. Removing the reliance on odometry data helps to improve the robustness and accuracy of the map generated by relying more on data from the LiDAR unit and IMU sensor.

HectorSLAM is thus known for its ability to generate high-quality maps that are visually informative and recognizable. By combining the map generated using HectorSLAM, together with a waypoint following algorithm, the robot can then navigate autonomously from one point to another by following a predetermined set of waypoints saved beforehand. This is especially relevant for us to enable the TurtleBot to perform periodic tasks (i.e. delivering canned drinks to tables) that require autonomous navigation between predefined waypoints.

2.2.4. Choice and Justification

Among the three algorithms discussed, HectorSLAM along with a waypoint following algorithm is the most appropriate path planning approach to adopt for our group to accomplish the mission objective. Both PDQN and VFH algorithms have inherent flaws that could cause the autonomous navigation to become unpredictable and compromise our mission objective.

PDQN requires an accurate estimate of the Q-value functions to work, which can be challenging to achieve consistently. As the Q-values are used to make decisions, the inaccuracy could lead to suboptimal performance. As described in equation 2, the PDQN algorithm could also be sensitive to the choice of hyperparameters used, and could pose a challenge to achieve good performance. Lastly, the overall PDQN algorithm will be computationally expensive as it requires the calculation of the Q-value of the robot's each state.

Although VFH uses a Gaussian filter to eliminate noise and the possibility of introducing local minimas, it still relies heavily on accurate sensor data to generate an accurate map. Moreover, VFH may not be well-suited for complex environments due to inherent limitations in its implementation. While the absence of repulsive or attractive forces in VFH reduces the likelihood of the robot from becoming imprisoned, it also implies that the algorithm may steer the robot away from the goal, which is undesirable for our mission.

In contrast, HectorSLAM is capable of simultaneous mapping and localization without relying on odometry data, making it robust and accurate in dynamic environments where odometry data may be noisy or unreliable. This is a significant advantage, as it enables the TurtleBot to navigate autonomously in real-time without the need for external localization systems or markers in the environment.

Furthermore, HectorSLAM utilises data from the LiDAR unit and IMU sensor, which are common sensors available in most robotic platforms, making it accessible and easy to implement. It does not require additional sensors or hardware, reducing the overall cost and complexity of the system, and hence it is the most practical for our mission objective.

2.3. Dispenser

Automation has become increasingly important in various industries, including the delivery of items. In this project, our group has been tasked to design and implement an automated dispenser system that is capable of releasing canned drinks to the TurtleBot for delivery to the designated table. The primary challenge of this task is to ensure accurate and efficient release of the canned drink directly into the TurtleBot and also knowing when to release it. This would entail a precise release mechanism and also a way to figure out when the TurtleBot has returned 'home'. Besides that, a user-friendly interface would also be required for users to select the table to deliver the canned drinks to.

2.3.1. Microcontroller

The TurtleBot uses a RaspberryPi (RPi) 3 model B+ which is a Single Board Computer (SBC) that acts as the TurtleBot's brain. The RPi is responsible for the autonomy of the TurtleBot and also to communicate with the MCU used on the dispenser.

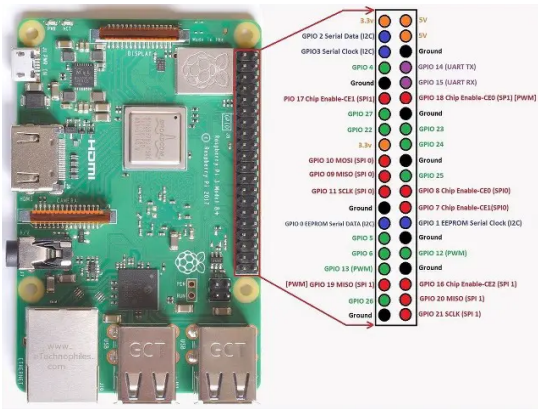


Figure 11

An MCU is required on the dispenser to control the electronics required for the dispenser to work and also ensure a seamless integration with the TurtleBot. Two popular choices for a MCU would be the Arduino UNO and ESP32 devices. Arduino is an open-source hardware and software platform that is widely used in the makers community and it is a low-cost and effective choice for student projects. This versatile MCU is equipped

with a ATmega328P and can be programmed using the Arduino Integrated Development Environment (IDE) that has a lot of standard libraries that can be used for our purposes. The Arduino UNO R3, however, does not have built-in bluetooth or Wi-Fi communication capabilities. In our application, we require the MCU on the dispenser to communicate with the RPi on the TurtleBot so the Arduino UNO R3 may not be the most suitable choice.

The ESP32 is a powerful bluetooth and Wi-Fi enabled MCU from Espressif systems. It has a wide range of peripherals, including the General Purpose Input Output (GPIO) pins, Inter-Integrated Circuit (I2C), Serial Peripheral Interface (SPI), Analog-to-Digital Converter (ADC) and supports various communication protocols such as Wi-Fi, bluetooth and Bluetooth Low Energy (BLE). Therefore, the ESP32 would prove to be a better choice over the Arduino UNO R3 to allow for the establishment of communication between the dispenser and the TurtleBot.

2.3.2. User-friendly interface

The interface used to select the table to deliver the canned drink needs to be intuitive and easy to navigate, with clear instructions and options presented in a simple and straightforward manner. One potential option for the interface is a mobile application. The app can be designed with a simple and intuitive layout, allowing the user to easily input the table number for delivery, at their convenience. The app could also provide real-time status updates of the delivery progress and a confirmation notification once the canned drink has been successfully delivered. Another potential option is a voice activated/ gesture control interface to allow for a contactless method of selecting the table number to deliver the canned drink. This would be particularly convenient for users with limited mobility. Lastly, the dispenser system can be equipped with physical buttons that allow the user to easily select the table number for the canned drink delivery. The buttons should be clearly labelled to provide a clear visual representation of the tables for the user, ensuring that the correct table number is selected.

Ultimately, considering the uncomplicated nature of our mission objective, a button interface would be the best option. A button interface, compared to the other previously mentioned interfaces, would be more intuitive for the restaurant staff, allowing them to quickly and easily select the table they wish to deliver the drinks to. Furthermore, physical buttons are more reliable and less prone to errors than a more complex interface, and is

therefore a practical solution for the task at hand. Figure 12 below shows a possible button interface which can be implemented for our user interface.

Figure 12

2.3.3. “ I’m Home! ”

Another important aspect of the dispenser system is to know that the robot is home to prevent the dispenser from releasing the canned drinks when the turtlebot is not home. To do so, it is important to establish some form of communication between the two entities. One possible way to achieve this is through the use of sensors. The dispenser can be equipped with a range sensor (e.g. ultrasonic sensor) to detect the presence or absence of the TurtleBot. However, this hardware implementation would require careful calibration of the hardware component being used. In addition, many range sensors have limited operating range and would not work well with close distances.

The use of communication protocols between the TurtleBot and the dispenser’s “brain” could prove to be a better choice. Once the Turtlebot has completed its docking manoeuvre, it would send an acknowledgement message to the dispenser’s “brain” to inform it, saying “I’m Home!” The specific communication protocols that could be used will be further discussed in later sections. This would ensure that the can is released only when the TurtleBot is in the correct position, reducing the risk of the drinks being released prematurely.

2.3.4. Release Mechanism

Designing a canned drinks dispenser for a TurtleBot requires careful consideration of factors such as the payload size and weight, mechanism for releasing the cans, and control method for the dispenser. In this section, we will review several existing designs for canned drinks dispensers and discuss their advantages and limitations in the context of our mission objective.

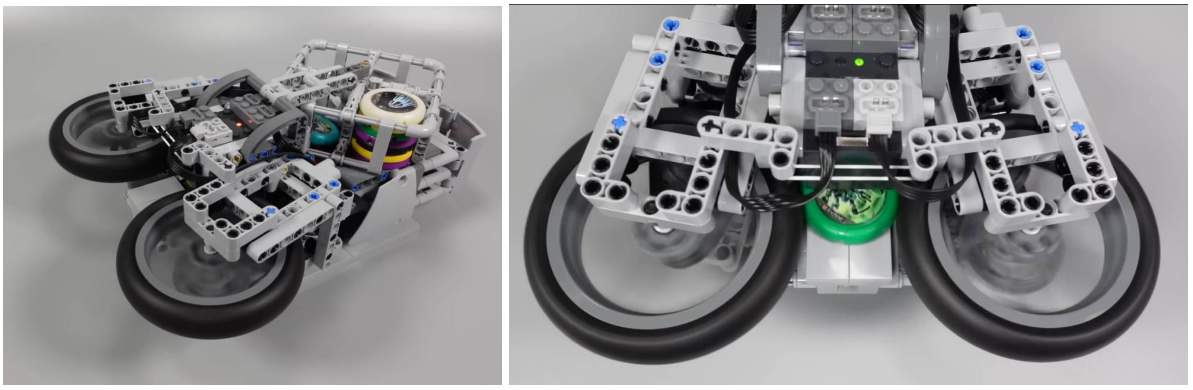


Figure 13

Figure 13 illustrates how a conveyor belt with a rotary mechanism can be used to propel a rubber discus forward. The high speed turning of the flywheels act as a launcher which will launch the rubber discus in the direction specified. Despite its high accuracy in launching the item in the correct direction, this method may not be suitable for our mission objective. Our payload is a drink can which is made out of a smooth surface. In addition, in a real-world scenario, the drink cans are usually condensed on the outside which makes friction an issue. The flywheels will not be able to launch the can as intended due to the loss of grip. More importantly, our payload is considered fragile and that the drink can should reach the table intact. Implementing this method could damage the can and potentially hurt the customer as well.

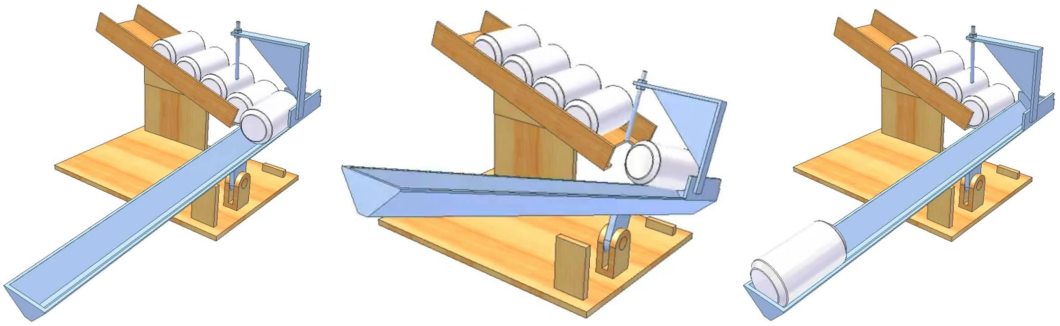


Figure 14

Another possible mechanism we could use is a two-part can dispenser mechanism as depicted above. This operates by storing multiple cans on a ramp, which would roll down the ramp until the current can reaches the blue track. Once it reaches the blue track, the track itself would tilt backwards as seen in Figure 14. Using a servo motor or pulley system, the track will rotate back to its original orientation, allowing for the current can to slide down the blue track, onto the TurtleBot.

This is a simple mechanism to implement. However, this method is not suitable for our mission, given the requirement that our dispenser must lie within a 50cm by 50cm dispenser zone. The servo motor also has to be strong enough to rotate the blue track back and forth with sufficient torque. The standard issued SG90 servo motor may not be strong enough to handle the job, and hence, this mechanism is impractical for us to implement.

Figure 15

Figure 15 illustrates a third method for our dispenser's release mechanism. A rotating can dispenser mechanism consists of a cylindrical open can holder that is initially positioned faced up. When a button is pressed and the TurtleBot is docked, it activates the servo motor to rotate the cylindrical can holder by 180 degrees around its central axis. This rotation causes the can to be released directly onto the TurtleBot. After the can has been dispensed, the cylindrical can holder returns to its initial position, in order to receive the next can.

In conclusion, the third mechanism mentioned is the most suitable for our mission objective. It is compact, and able to fit within the 50cm by 50cm demarcation, a requirement which the other mechanisms may not meet. In addition, the gradual rotation of the can in the third mechanism also ensures that the can is released onto the TurtleBot safely, and is the easiest to design and implement.

2.4. Communication

Communication protocols are a set of standards that govern the exchange of data packets between devices and are essential for ensuring data is transmitted accurately and reliably regardless of the platform used. Communication protocols can be divided into two broad categories: wired and wireless protocols.

Wired protocols typically include Ethernet, Controller Area Network (CAN) bus, and Serial Peripheral Interface (SPI) and are generally much faster and more reliable than wireless protocols. They are less susceptible to electronic interference and can transmit data over longer distances without having to consider the attenuation of signals. However, wired protocols may not be suitable for our mission objective since the two systems (i.e. dispenser and TurtleBot) will be far apart from each other. The distance between the two systems would be dynamic and having a wired network will not be feasible as the cable will be dragged around the restaurant and pose safety risks in a dynamic setting. Also, the cable may provide resistance to the movement of the TurtleBot and mess up the map generated by the TurtleBot.

Wireless communication, on the other hand, allows for seamless and convenient communication between the dispenser system and the TurtleBot without the need for physical connections or tethering. This enables the users to send cues or instructions to the TurtleBot regarding which table to go to, in real-time, without being hindered by physical constraints. This can significantly enhance the system's flexibility and adaptability in navigating the restaurant environment. Examples of wireless communication protocols include Bluetooth, Wi-Fi , and Zigbee. Compared to wired protocols, wireless communication has the advantage of being more flexible and convenient. Although it is arguable that wireless communication may be easily influenced by other devices using the same frequency band, we are working in an enclosed environment where there will not be any significant interference from other devices. The issue of suboptimal wireless connection at large distances that is an inherent limitation of wireless connections will also not pose as an issue for us since the area of concern is not huge.

2.4.1. Serial Peripheral Interface

SPI is a communication protocol used for communicating with microcontrollers and other devices. It is a synchronous serial communication protocol where data is transferred in

a synchronised manner using a clock signal. SPI is a full-duplex protocol, meaning that data can be sent and received simultaneously. This is an essential feature for our mission objective where both the dispenser and TurtleBot have to receive and transmit data simultaneously without interference or delay.

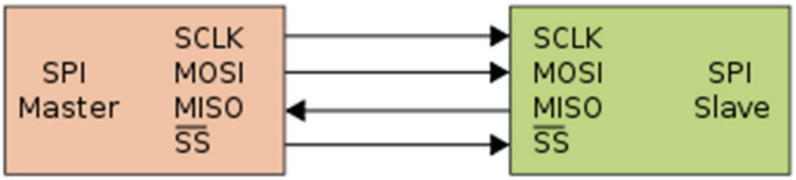


Figure 16

The SPI protocol uses four lines for communication: Master Out Slave In (MOSI), Master In Slave Out (MISO), Serial Clock (SCLK), Slave Select (SS). MOSI is used to send data from the master to the slave, while MISO is used to send data from the slave to the master. SCLK is the clock signal that synchronises the data transfer and SS is used to select the slave device that the master wants to communicate with.

One of the disadvantages of SPI is that it is a point-to-point communication protocol, meaning that it can only communicate between a single master and a single slave. In addition, SPI does not have built-in error checking functions (e.g. checksums) to ensure that data packets received are not corrupted. This may compromise data integrity and cause the TurtleBot to receive a table number that is different from the one chosen by the user.

2.4.2. Hypertext Transfer Protocol

Hypertext Transfer Protocol (HTTP) is the protocol used by the World Wide Web to transfer data between web servers and clients. It is an application layer protocol that is built on top of the Transmission Control Protocol/Internet Protocol (TCP/IP) stack, which provides reliable communication over the Internet. The HTTP protocol communicates by sending out a request and response packet. The communication sequence is shown in Figure 17 below.

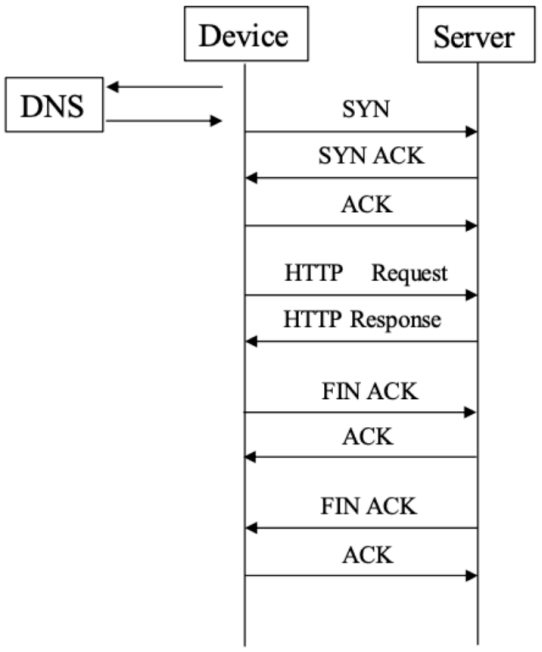


Figure 17

However, HTTP is a request-response protocol, which means that clients can only request information from servers, but cannot initiate communication on their own. In addition, HTTP has a significant overhead due to the additional data that must be transmitted along with each request and response. The additional data includes headers, which contain metadata such as the content type, content length, and other information required for the server and client to properly handle the request and response. The additional data required to be sent along with the actual message packet may slow down the overall communication speed and increase latency which is not ideal.

2.4.3. Message Queuing Telemetry Transport (MQTT)

The MQTT protocol defines two types of network entities: a message broker and a number of clients. An MQTT broker is a server that receives all messages from the clients and then routes the messages to the appropriate destination clients. An MQTT client is any device (from a micro controller up to a fully-fledged server) that runs an MQTT library and connects to an MQTT broker over a network.

Information is organised in a hierarchy of topics. When a publisher has a new item of data to distribute, it sends a control message with the data to the connected broker. The broker then distributes the information to any clients that have subscribed to that topic. The

publisher does not need to have any data on the number or locations of subscribers, and subscribers, in turn, do not have to be configured with any data about the publishers.

A MQTT broker can be set up on a laptop. Then, both the ESP32 on the dispenser and RPi on the delivery robot can be connected to this broker. The ESP32 and RPi can then communicate to each other by publishing and subscribing to different topics.

3. Preliminary Design

3.1. Delivery Robot

The delivery robot is constructed on the TurtleBot3 Burger model, with the addition of a container to hold the can drink during transportation. The container is fabricated using acrylic material and is fixed to the TurtleBot using screws and nuts. It contains a microswitch that is responsible for detecting the presence of the can drink within the container. Upon detection of the can drink, the TurtleBot will initiate its navigation algorithm and proceed to the designated table. Once the delivery is complete and the TurtleBot has reached the table, it will wait until the can drink is removed and the microswitch is released before returning back to the dispenser.

In addition to the microswitch, the container is also equipped with two IR sensors to aid the line following algorithm to dock precisely in the dispenser. To support the weight of the can drink and prevent the acrylic container from snapping, a ball caster wheel is also integrated into the design to provide additional support.

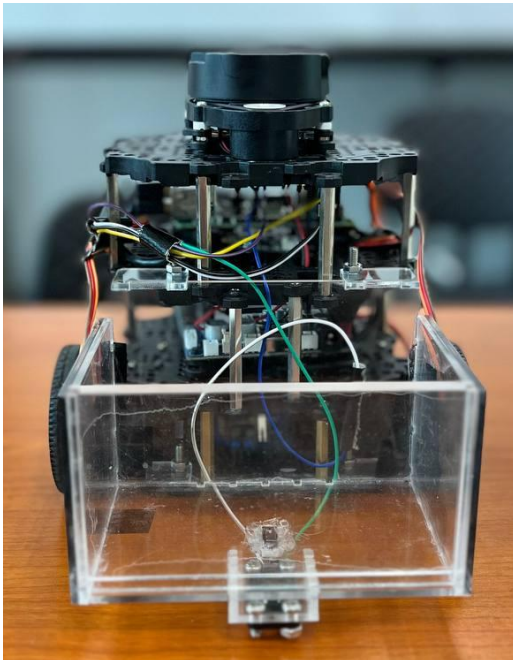


Figure 18

3.2. Autonomous Navigation

The autonomous navigation to deliver the can drinks will start once the TurtleBot senses the can drink in the container. A table number will be sent over from the dispenser to the TurtleBot through MQTT communication to inform the TurtleBot of the route to take. The TurtleBot will then follow a set of waypoints that was saved beforehand to navigate autonomously and finally reach the designated table. Once all the waypoints for that specific route have been traversed, the TurtleBot will stop and wait for the can drink to be removed before finding its way back to the dispenser. The navigation back to the dispenser is following the same concept, except that the waypoints to travel are now reversed. The docking of the TurtleBot into the dispenser system is done through a line-following algorithm to ensure that the TurtleBot is exactly at the location that we want it to be.

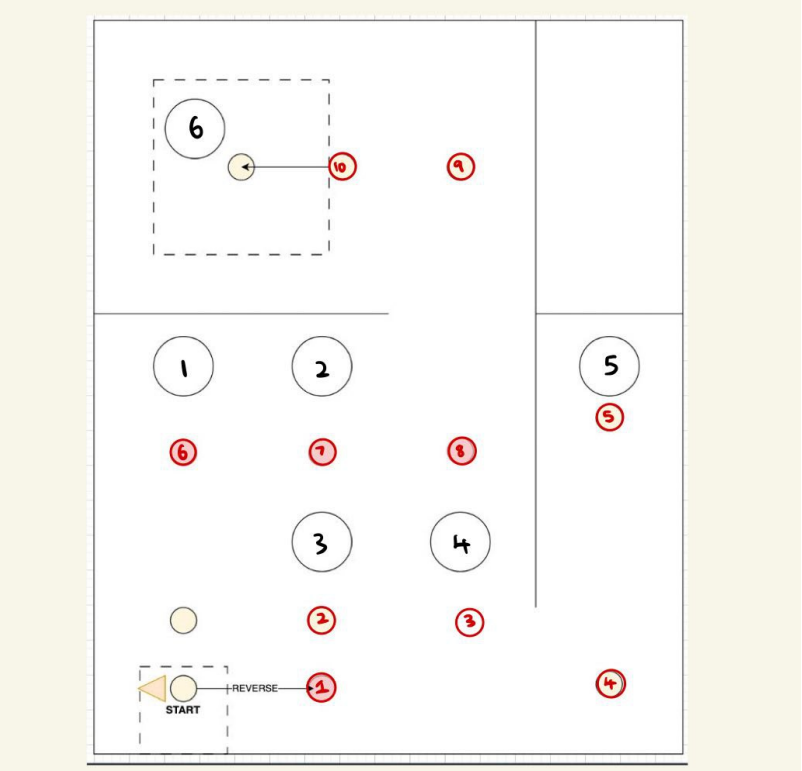


Figure 19

Figure 19 above shows a rough understanding of the restaurant layout, with the tables marked in black and the waypoints marked in red. The waypoints marked in red will be taken at the start of each run to save the waypoints to be used for autonomous navigation. The function to collect waypoints is detailed in section 7.3. As seen, each table has its own set of waypoints that can be used to navigate to. For example, table 5 would entail the TurtleBot to travel waypoints 1, 4, 5.

For tables 1 to 5, the table locations are made known to us. Once the TurtleBot reaches the final waypoint for its respective table, it will then scan its peripheral surroundings to find the angle of the shortest distance and rotate in that direction. The TurtleBot will then move in that direction until the distance from the front of the TurtleBot and the table reaches below a certain threshold such that it is less than 15 cm away. For table 6, the only information on the location made known to us is that it will be somewhere within a 1m by 1m zone. As such, additional aspects of autonomy need to be implemented to reach table 6.

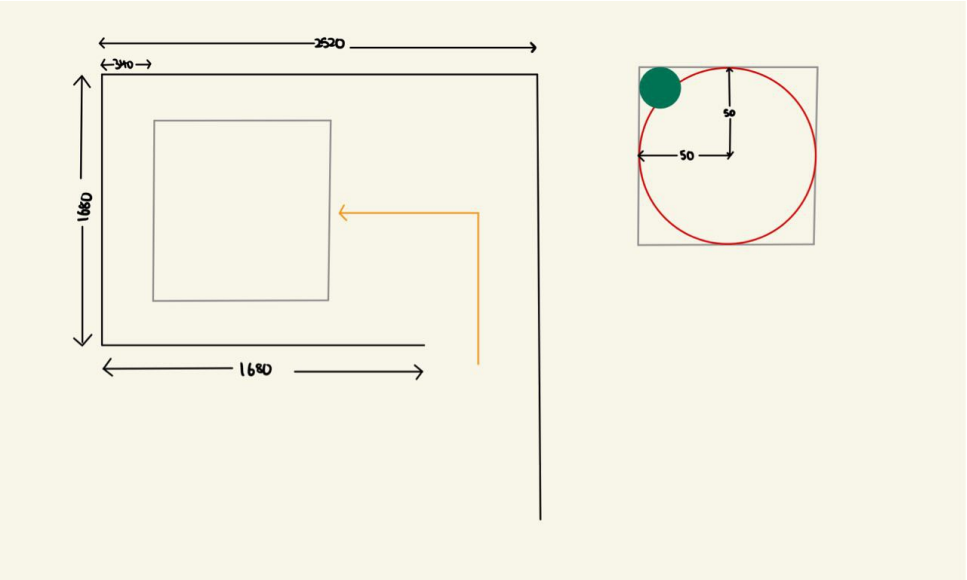


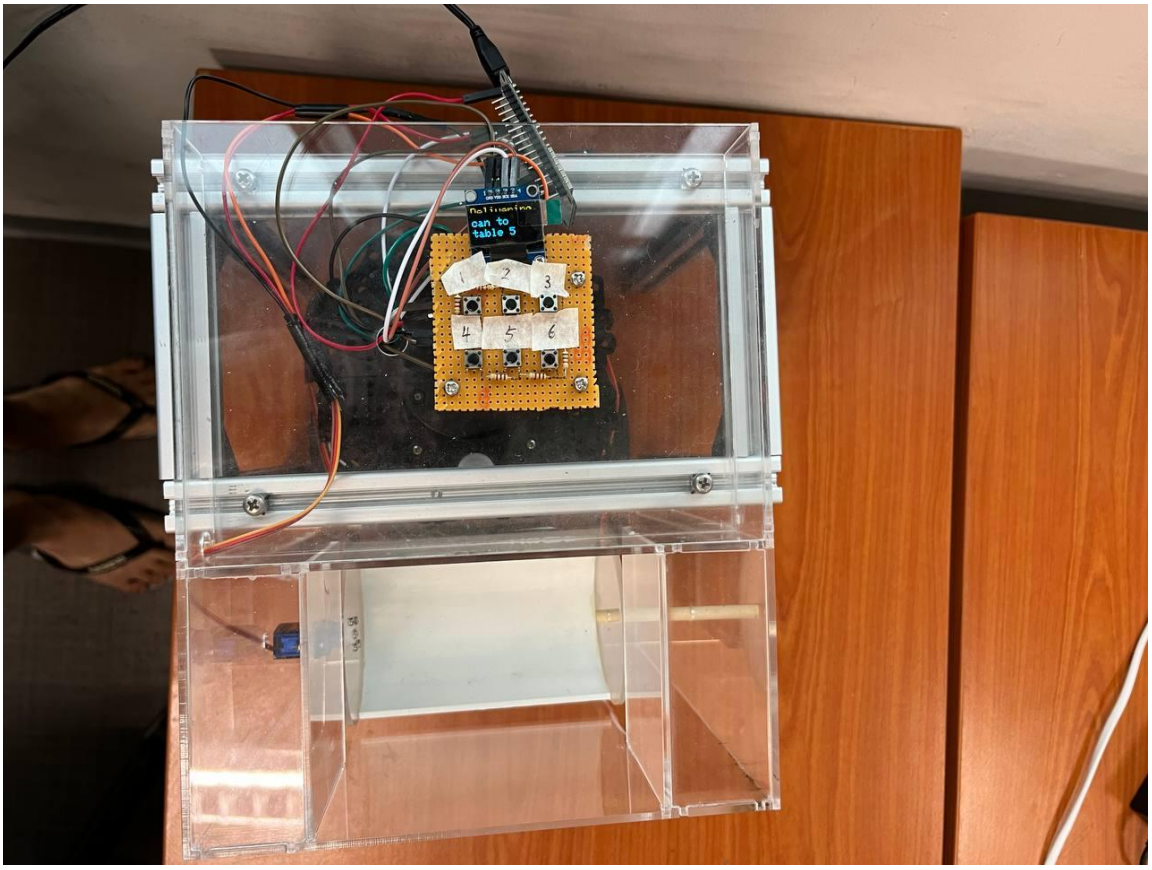
Figure 20

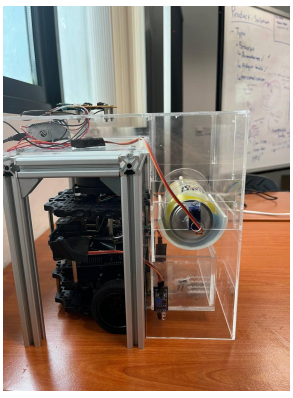
The Turtlebot will first reach waypoint 10 in figure 20 before calling another function to navigate to table 6. Once outside the 1m by 1m zone, the Turtlebot will keep moving for another 50 cm forward, while scanning its peripheral surroundings for any object detected that has a distance less than 50 cm. Once detected, the TurtleBot will stop moving and rotate towards the object found and move towards it. In the event that no object is detected, the TurtleBot will stop after moving 50cm forward to the centre of the 1m by 1m zone. We can then conclude that table 6 is most likely non-existent. The edge case here would be that table 6 is placed at one of the four corners of the 1m by 1m zone. By taking into account the minimum diameter of the table and the scanning radius of the TurtleBot, table 6 will always be picked up by the TurtleBot and move towards it. A visual representation of table 6's algorithm can be seen in Figure 20.

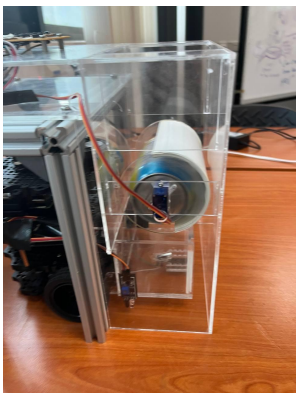
3.3. Dispenser

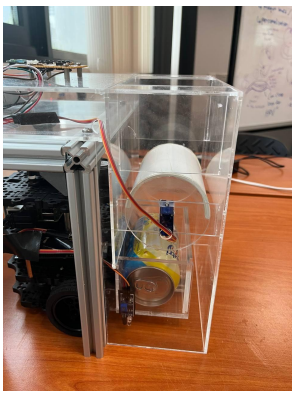
The power supplied to our dispenser system should be able to provide sufficient power for the microcontroller, servo motor and the user interface on the dispenser. In our design, the servo motor and the user interface will be powered by the 5V/3.3.V output pin of the microcontroller —- an ESP32 development board while the board itself will be powered from the wall plug using a power adapter. The user interface contains one OLED screen and six buttons. Each button represents one table and the table selected will be displayed on the OLED screen. The OLED screen will also show if the ESP32 is connected to wifi and MQTT

broker. After initialising, the ESP32 will send a signal to rotate the servo when it receives a “home” message from the delivery robot via the MQTT broker. It will then send the table number to the delivery robot via the same MQTT broker. The combination of servo motor and a cylindrical can holder achieve the dispensing purpose of our dispenser. The can holder can hold the can while waiting for the robot to return and the can will be released by the servo turning 180 degrees after the robot docks into the dispenser.









3.4. Communication

MQTT is used for communication between the ESP32 on the dispenser and the RPi on the delivery robot. A broker will be set up on a laptop and both ESP32 and RPi will connect to this broker. The laptop, ESP32 and RPi need to connect to the same wifi or mobile data hotspot. The table number and the docking status of the robot can be sent via MQTT between the ESP32 and RPi.

4. System Technical Specifications

4.1. Delivery Robot Specifications

Max Translational Velocity	~ 0.22 m/s
Max Rotational Velocity	~ 162.72°/s (2.84 rad/s)
Dimensions (L x W x H) (mm)	~ (218.77 x 176.6 x 194.74)
Total Mass	~ 1050.8 g
Center of Mass (cm)	~ (0, 7.3, 4.8)
Wheel Base	~ 80.77 mm [Horizontal distance between the centres of the 2 sprocket wheels & the metal ball caster wheel]
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C
Operating voltage	~ 11.1V
Operating current	~ 0.91A during start-up ~ 0.59A during standby ~ 0.77A during normal operation
Expected charging time	~ 2h 30m

Expected operating time	~ 140 mins
Single Board Computer	Raspberry Pi 3B+
Wheel Motors	Dynamixel XL430-W250
Lidar Sensor	360 Laser Distance Sensor LDS-01
Microswitch	Panasonic AVT342061
IR Sensor	3Pcs IR Infrared Obstacle Avoidance Sensor Modules
Inertial Measurement Unit	Gyroscope 3-Axis Accelerometer 3-Axis
Equipments Needed	A laptop and wifi or mobile data hotspot. ESP32 on the dispenser, RPi on the delivery robot and laptop need to connect to the same wifi or mobile data hotspot.

4.2. Dispenser Specifications

Dimensions (L x W x H) (mm)	~ (246 x 240 x 258)
Total Mass	~ 1645 g
Center of Mass (cm)	~ (11.9, 16.5, 12.5)
Maximum Capacity	1 Can
Operating voltage	5V from USB
Operating current	~ 0.55A

Microcontroller Unit (MCU)	ESP32
OLED Screen	OLED-128O064D-BPP3N00000
Servo Motor	SG90

5. Electrical

5.1. Electrical Component List

Component	Quantity	Function	Specification	Remark
Push buttons TE 1825910-6	6	Vary the voltage output of the button interface to achieve the selection of table	• Dimension: 0.65 * 0.65 * 0.4 cm • Max Contact Rating: 50mA at 24VDC • Min Contact Rating: 10uA at 1VDC • Actuation Force 100 ± 50GF	
OLED screen OLED-128 O064D-BPP 3N00000	1	Display table information in the dispenser	• Operating Voltage: 2.8 - 3.3 V • Operating Current 9 - 12 mA • Dimension: 26.7 x 19.26 x 1.65 mm • Display format: 128 x 64 dots	Powered by ESP32 3.3V output

			• Interface: 6800, 8080, serial, and I2C • Built-in controller: SSD1306BZ	
ESP32 ELECbee ESP32 Developme nt Board	1	Dispenser control and communicatio n	• Operating Voltage: 5V • Operating current: 0.24A • 1 x 3.3V output pin (while powered by 5V USB, Vin will output 5V power) • 25 digital pins (including PWM x 21, I2C x 1, SPI x 3) • ADC Channels (12-bits SAR) x 15 • 2.4 GHz dual-mode Wi-Fi and bluetooth chips	Powered by wall plug
Servo motor SG90	1	Turn the can holder to dispense the can	• Operating Voltage: 4.8V (~5V) • Operating Current: 0.3A	Powered by ESP32 5V output

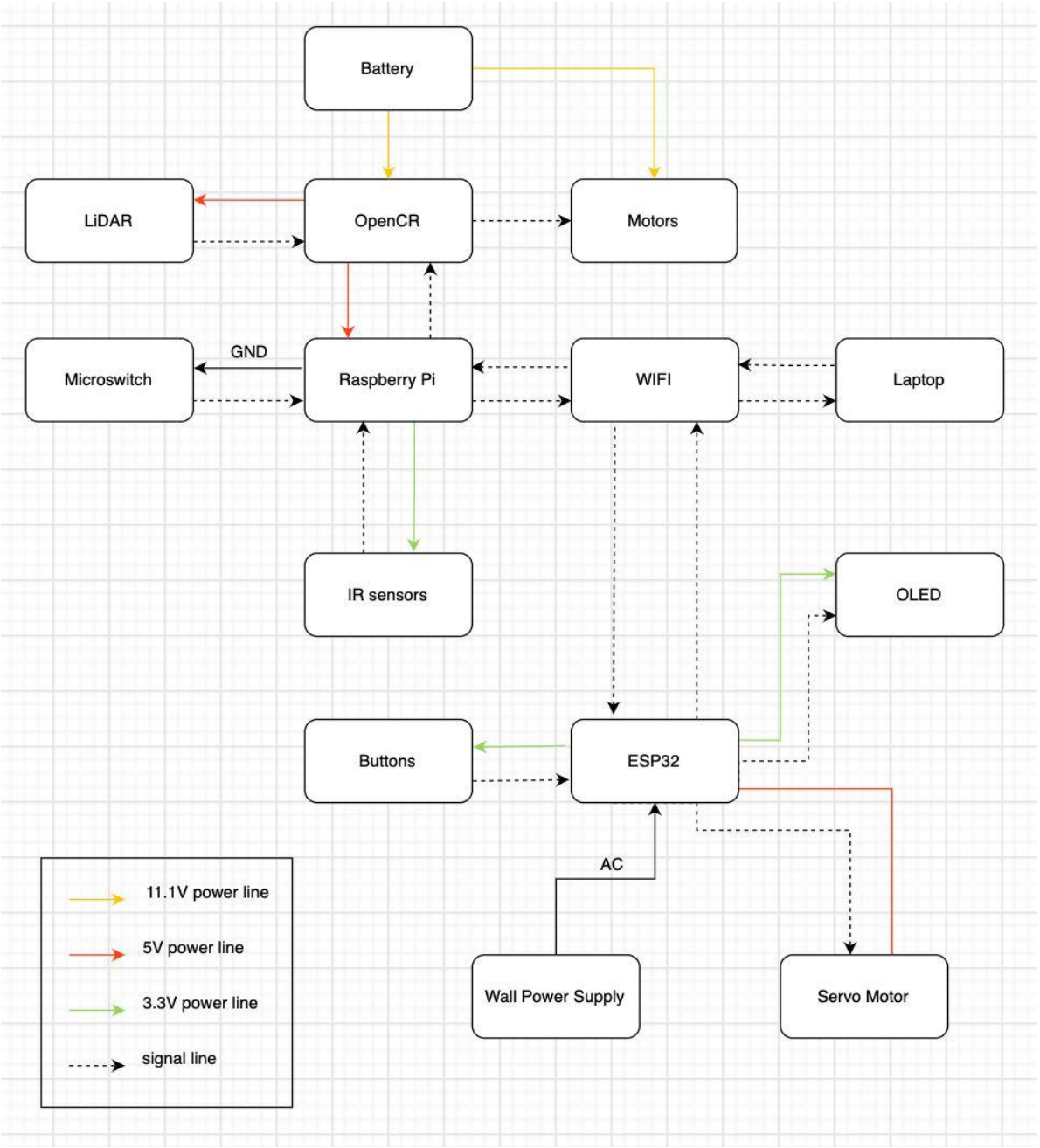
			• Dimension: 22.2 x 11.8 x 31 mm approx • Weight: 9g • Stall torque: 1.8 kgf·cm • Rotational Speed: 60°/0.1s • Position "0" (1.5 ms pulse) is in the middle, "90" (~2ms pulse) is all the way to the left. ms pulse) is all the way to the right, ""-90" (~1ms pulse) is all the way to the left.	
Raspberry Pi 3B+	1	Robot control and communicatio n	• Operating Voltage and Current: 5V/2.5A DC via micro USB connector • 2 x 3.3V and 2 x 5V output pins • 26 GPIO digital pins (of which 2 pins (GPIO 18 and 19) can be used for PWM output) • No ADC Channels	Powered by OpenCR 5V output
OpenCR 1.0	1	Processing of LiDAR	• Operating Voltage: 5 - 24V (default battery:	Powered directly by

		information	11.1V) ● Operating current: 4.5A ● 32 digital pins (including PWM x 6, I2C x 1, SPI x 1) ● ADC Channels (Max 12bit) x 6	battery
LiDAR LDS-01	1	Perform mapping and navigation of the robot	● Operating Voltage: 5V ± 5% ● Current consumption: 400mA or less (Rush current 1A) ● Detection Distance: 120 ~ 3500 mm ● Dimension: 69.5(W) X 95.5(D) X 39.5(H)mm ● Mass: 131g ● Distance Accuracy: ±15mm (120 - 499mm); ±5.0% (500 - 6000mm) ● Distance Precision: ±10mm (120 - 499mm); ±3.5% (500 - 6000mm) ● Scan rate: 300 ±10 rpm	Powered by OpenCR 5V output
Motor	2	Drive the robot	● Operating Voltage: 6.5	Powered

XL430-W250			~ 12.0 [V] (Recommended : 11.1 [V]) ● Operating Current 0.052A when stand by; 0.15A when moving with no load; 1.4A when stalled ● Dimension: 28.5 x 46.5 x 34 mm ● Gear Ratio: 258.5 : 1 ● Stall Torque: 1.4 [N.m] (at 11.1 [V], 1.3 [A]) ● No Load Speed: 57 [rev/min] (at 11.1 [V]) ● Mass: 57.2g	directly by battery
Infrared Sensor 3Pcs IR Infrared Obstacle Avoidance Sensor Modules	2	Detection of black line for the docking of the robot	● Operating Voltage: 3-5V ● Operating Current: 0.06mA ● Detection distance: 2-30cm ● Detection angle: 35 ° ● Board size: 3.2CM * 1.4CM ● With the screw holes	Powered by Raspberry Pi 3B+ 3.3V output

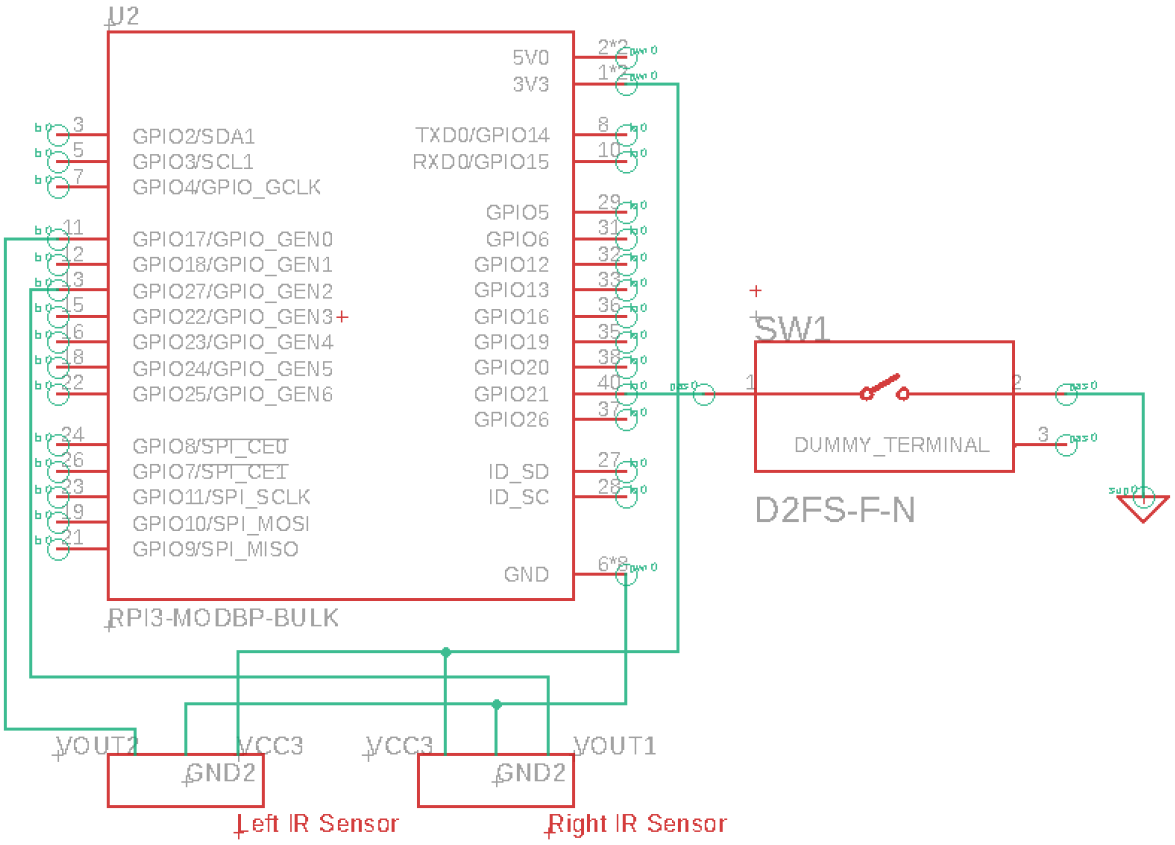
			3mm, easy fixed installation	
Microswitch Panasonic AVT342061	1	Detection of the condition of the can	• Dimension: 1.3 x 0.3 x 0.6 cm • Max Contact Rating: 100mA at 30V DC • Min Reactive Force: 0.078N • Max Operation Force: 0.49N	

5.2. Electronic System Architecture



5.3. Robot Electrical Design (including schematics)

5.3.1. Robot Schematics



5.3.2. Raspberry Pi 3B+

The Single Board Computer(SBC) used in our robot is a Raspberry Pi 3B+ and is placed on the second layer of the robot. There are no breadboards or PCBs presented in the robot. Henec, all the wires of different electrical components of this system is directly connected to the Raspberry Pi 3B+

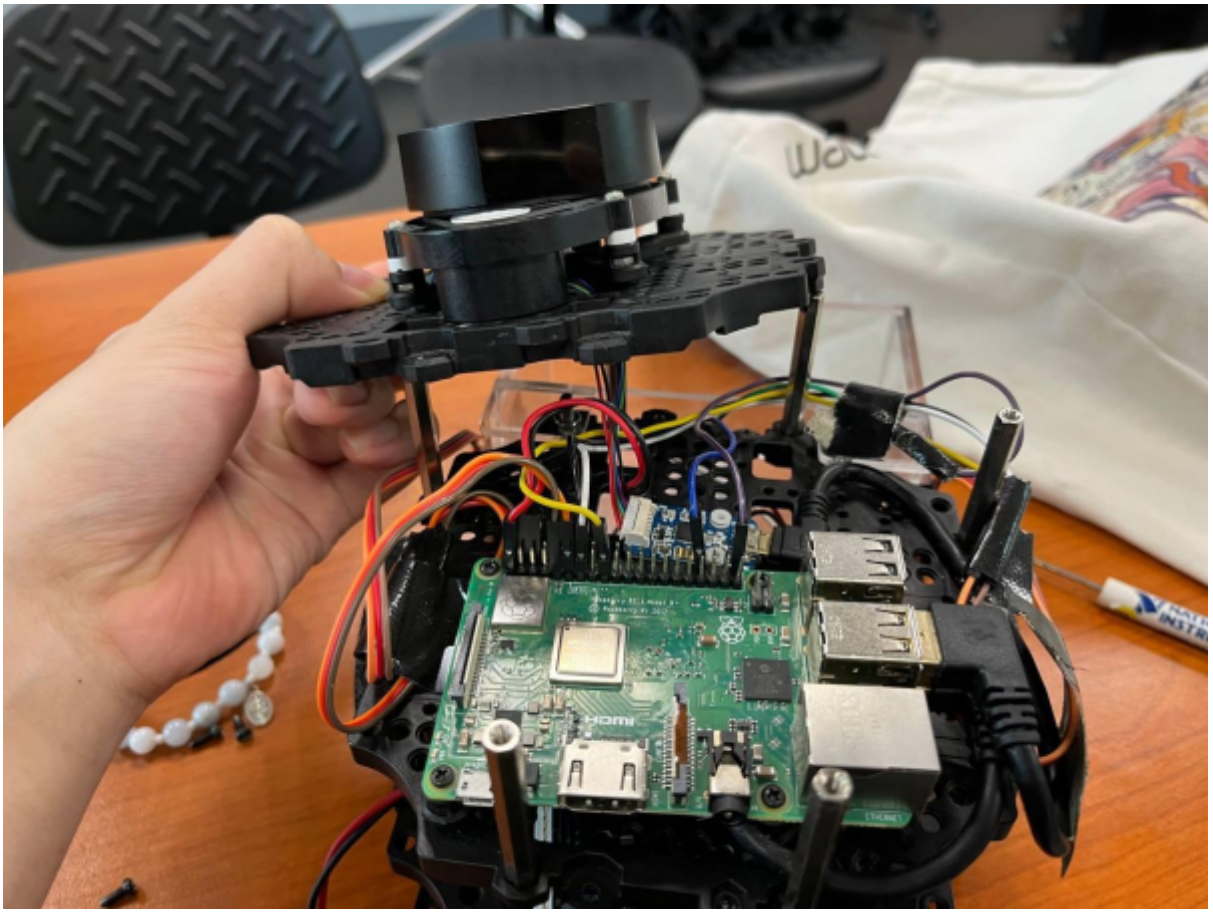


Fig 21: The placement of Raspberry Pi 3B+ on the second layer of the robot

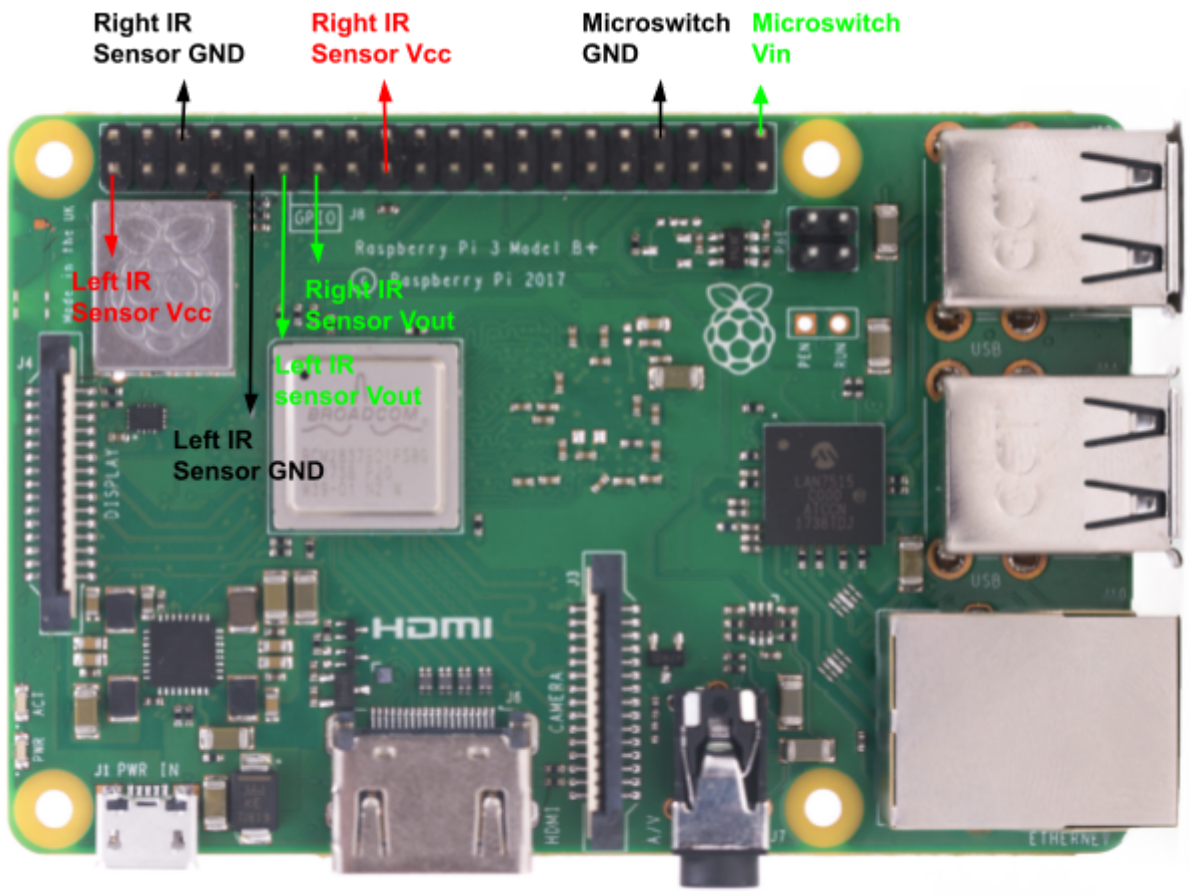
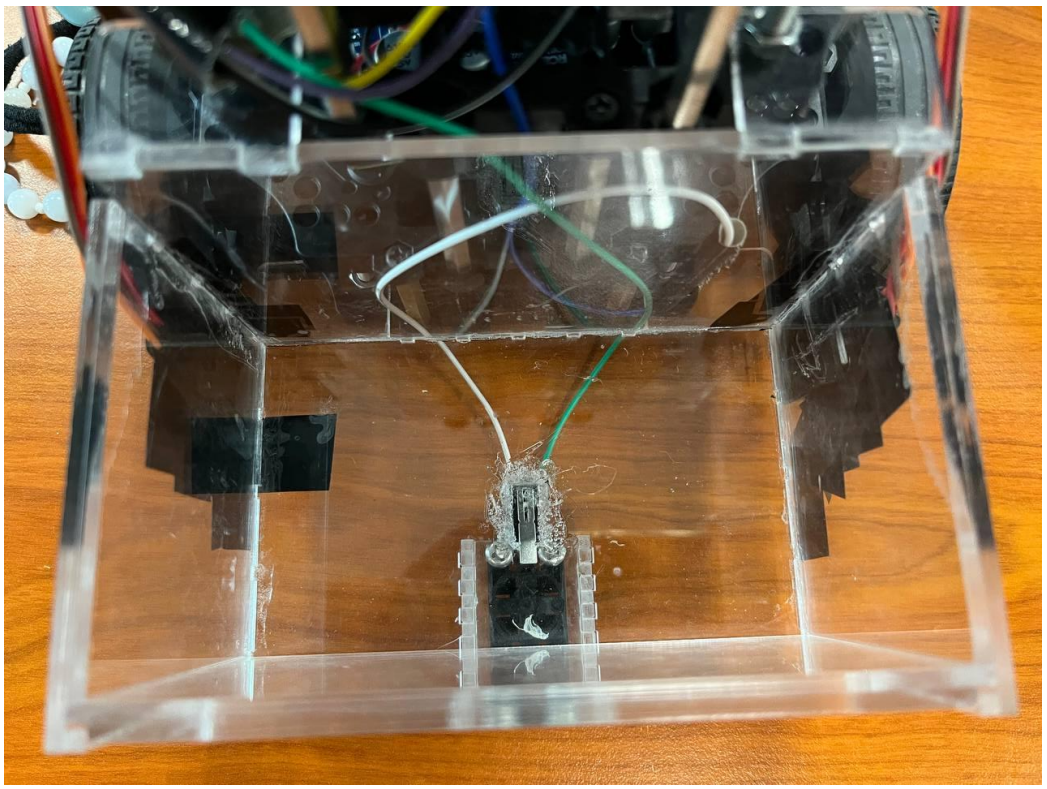


Fig 22: The wire connection on Raspberry Pi 3B+

5.3.3. Microswitch

A microswitch Panasonic AVT342061 is attached to the bottom of the robot container using hot glue to detect the state of the can in the container. Once a can drop onto it, the switch will be closed. The two pins on Raspberry Pi 3B+ connected to the microswitch are the Ground and GPIO Pin 21 (which is set to level low). When the switch is closed, GPIO Pin 21 will be grounded which will results in a “low” digital signal level which triggers the pin.



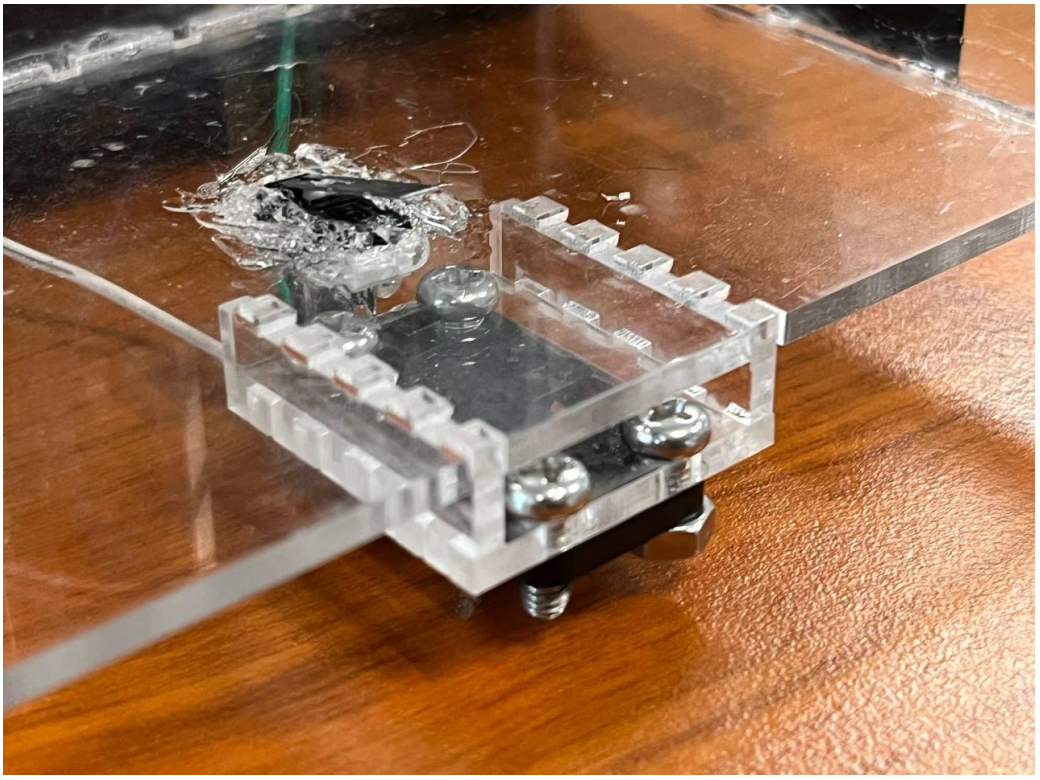


Fig 23 and 24: The placement of microswitch at the bottom of the robot container

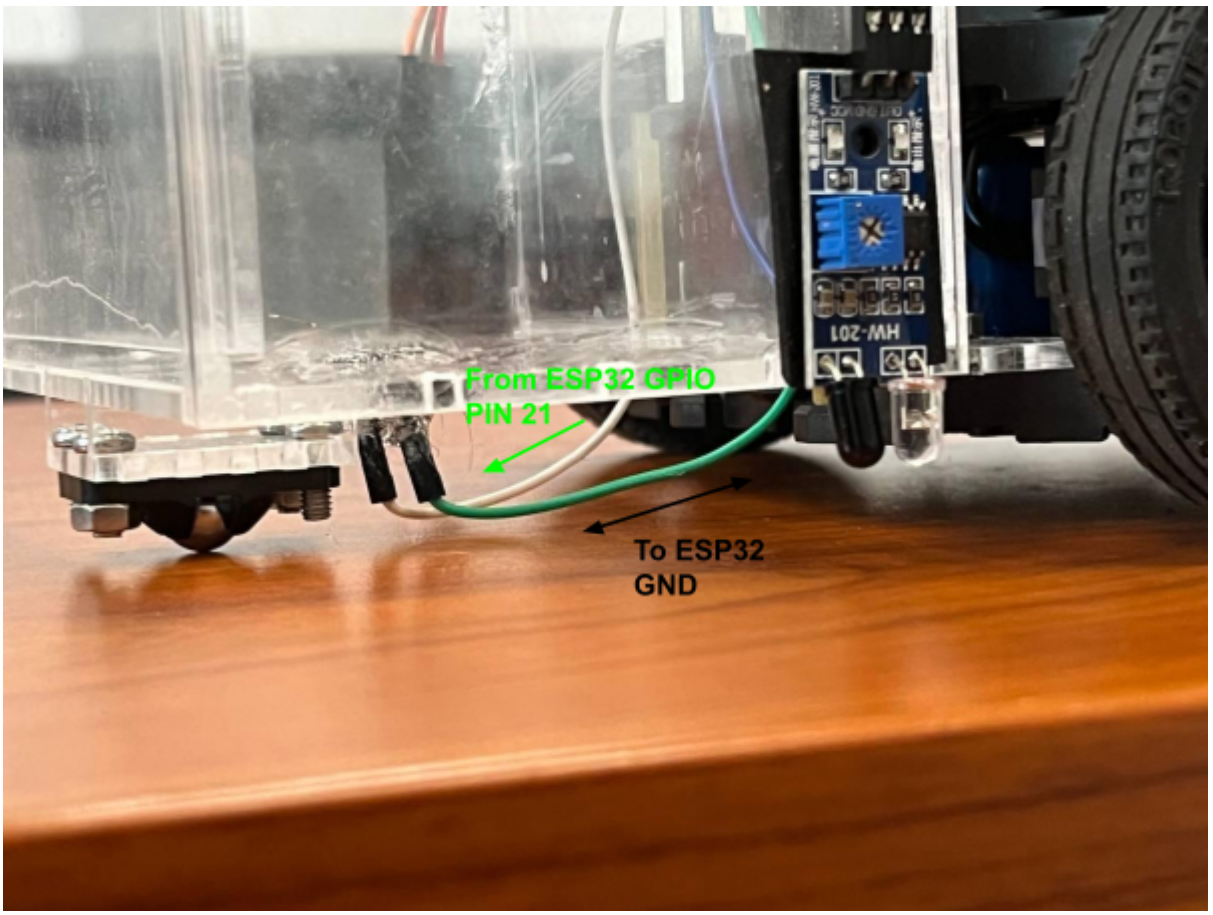
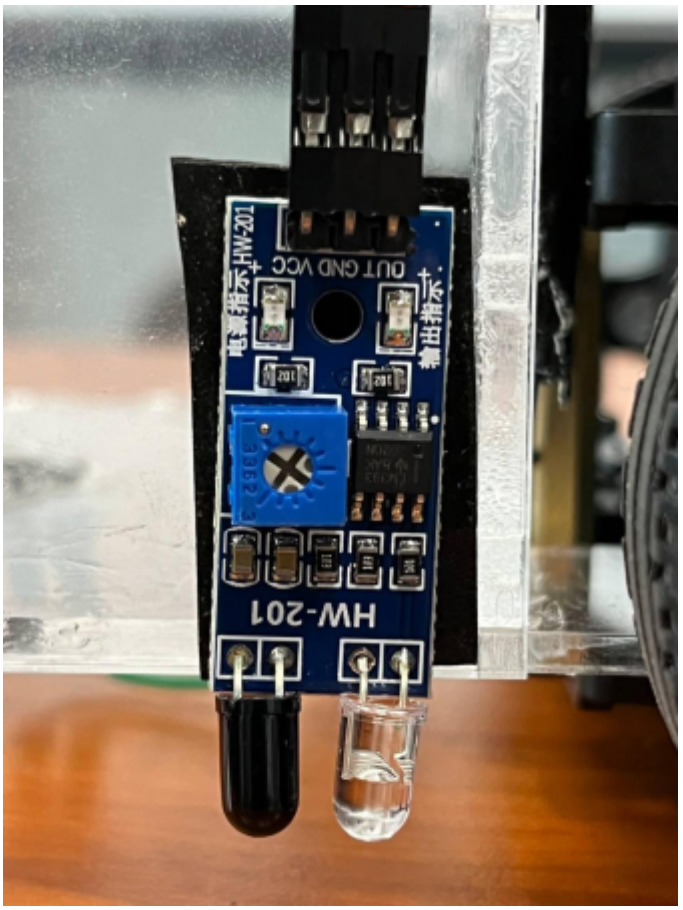


Fig 25: The wire connection of Microswitch

5.3.4. IR Sensor



Two 3Pcs IR Infrared Obstacle Avoidance Sensor Modules are attached to the sides of the container closer to the robot to assist the docking process of the robot. The IR sensor will send a digital signal “high” if it senses a black line underneath and “low” if it senses white line otherwise. Both IR sensors are powered by the 3.3V output pins of Raspberry Pi 3B+ and the signal output pin of the left and right sensor is connected to

GPIO Pin 27 and GPIO Pin 17 of the board respectively.

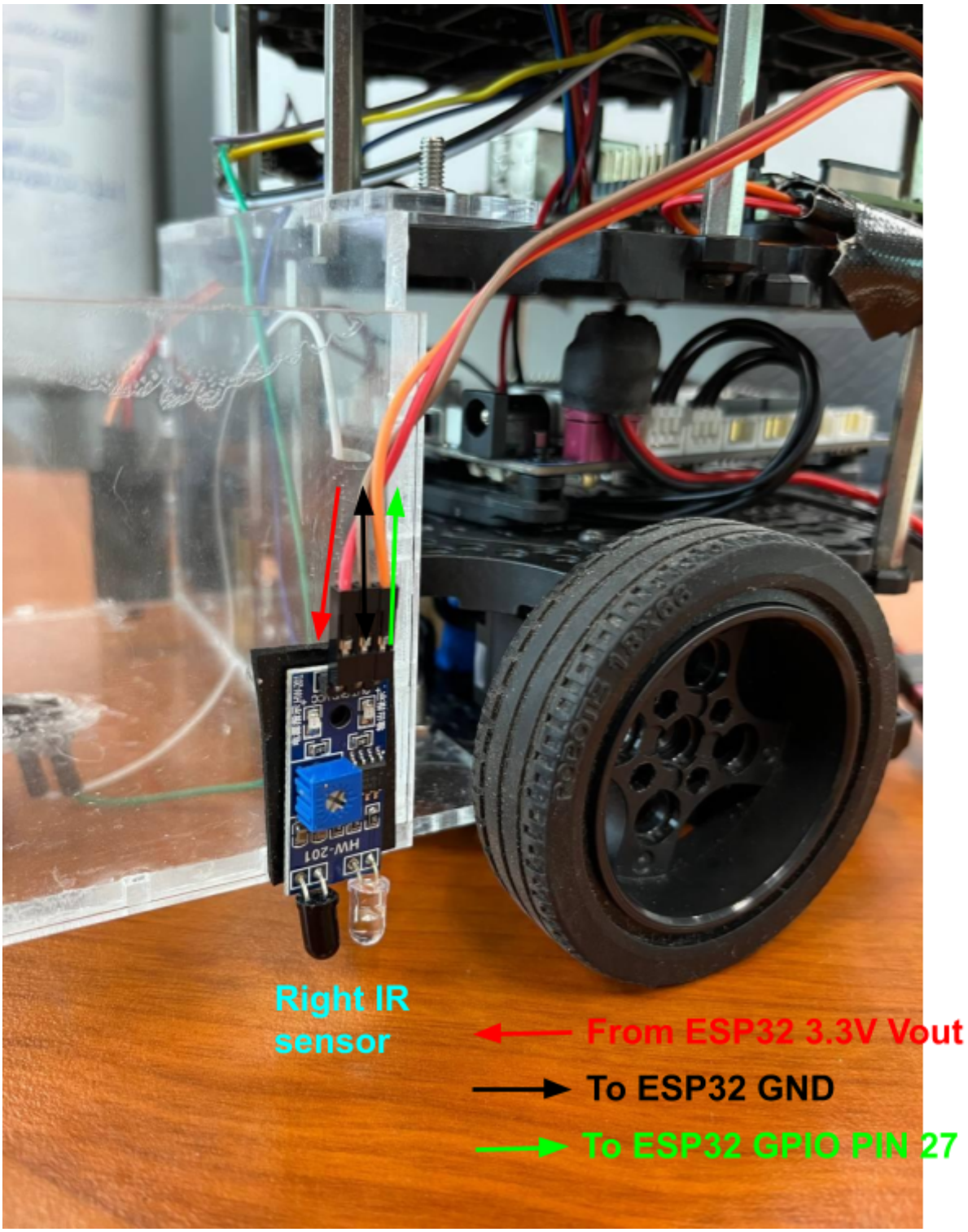
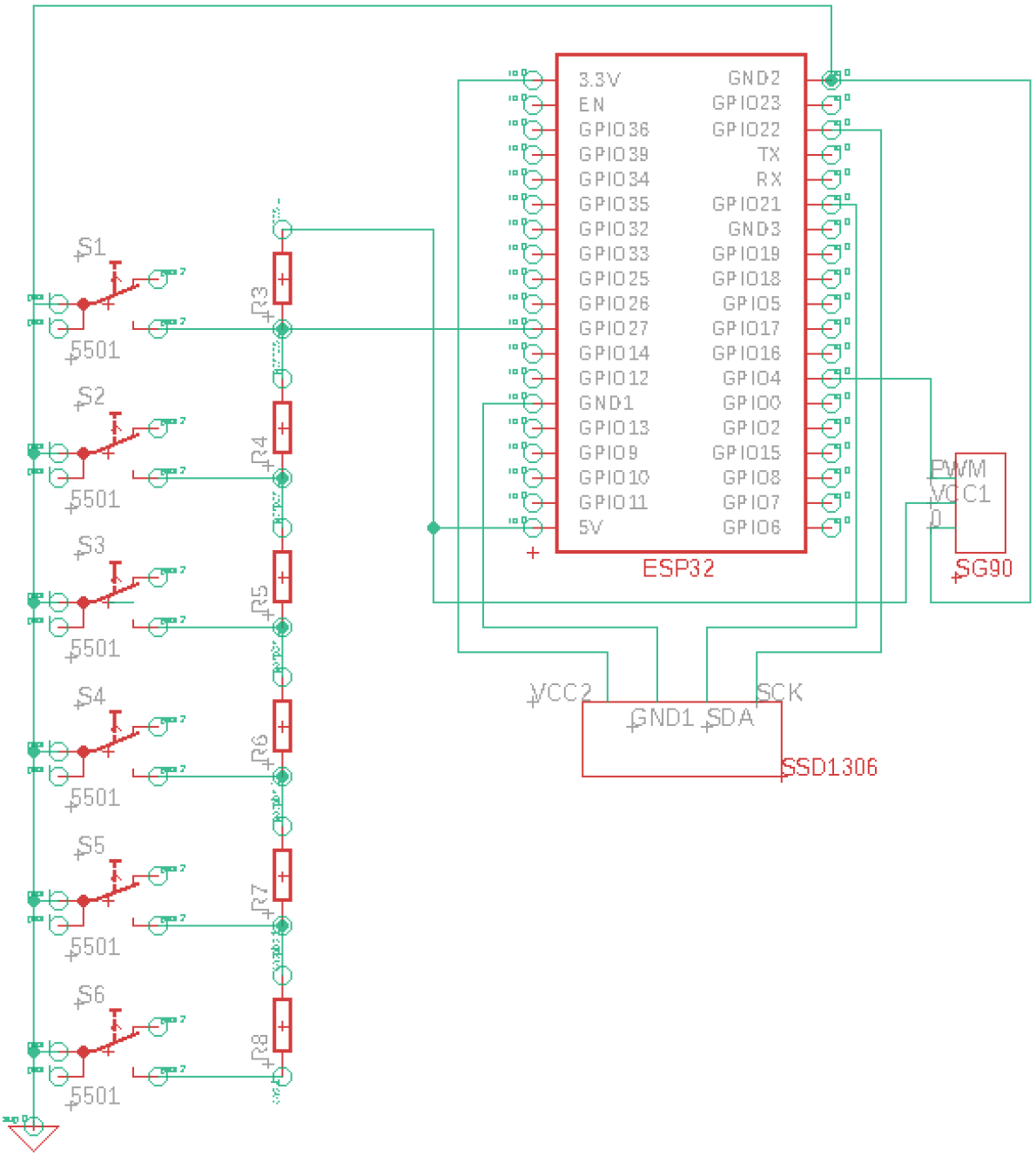


Fig 26: The placement and wire connection of IR sensor modules on the side of the robot container

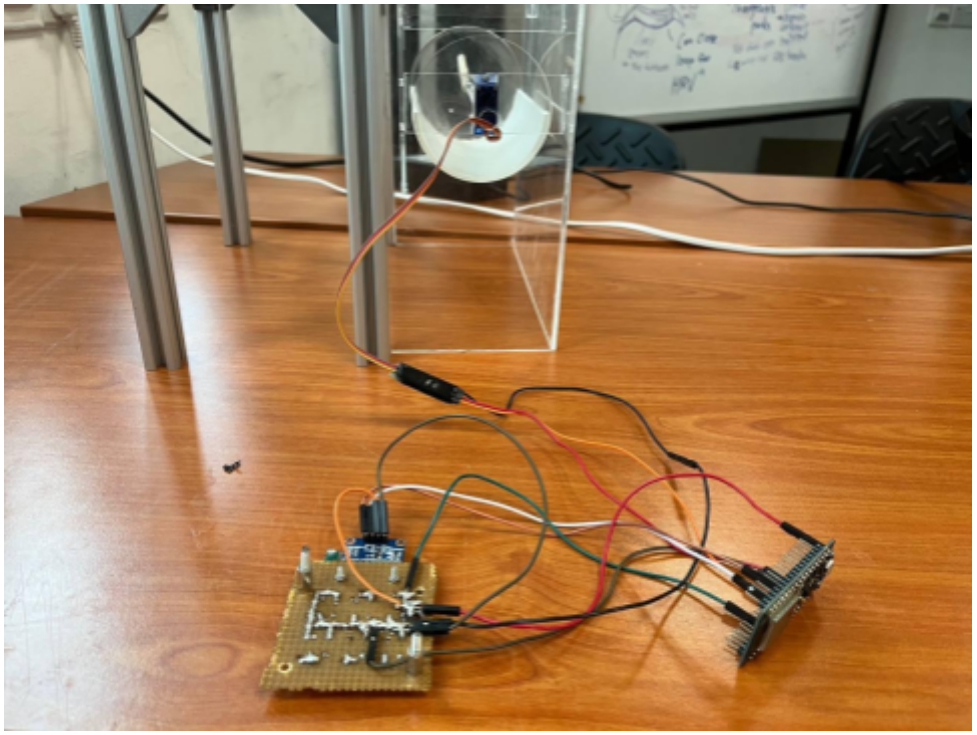
5.4. Dispenser Electrical Design (including schematics)

5.4.1. Dispenser Schematics



5.4.2. ESP32

The microcontroller used in our dispenser is an ELECbee ESP32 Development Board. The ESP32 board is placed in between the top and the middle plate of the dispenser.



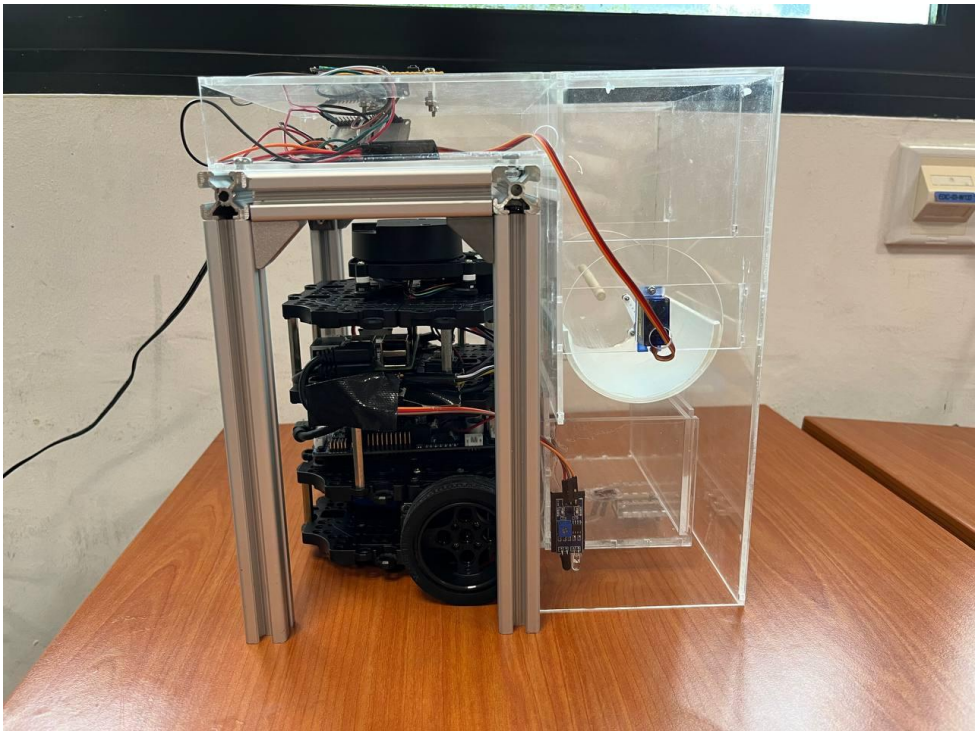


Fig 27 and 28: The placement of ESP32 in the dispenser

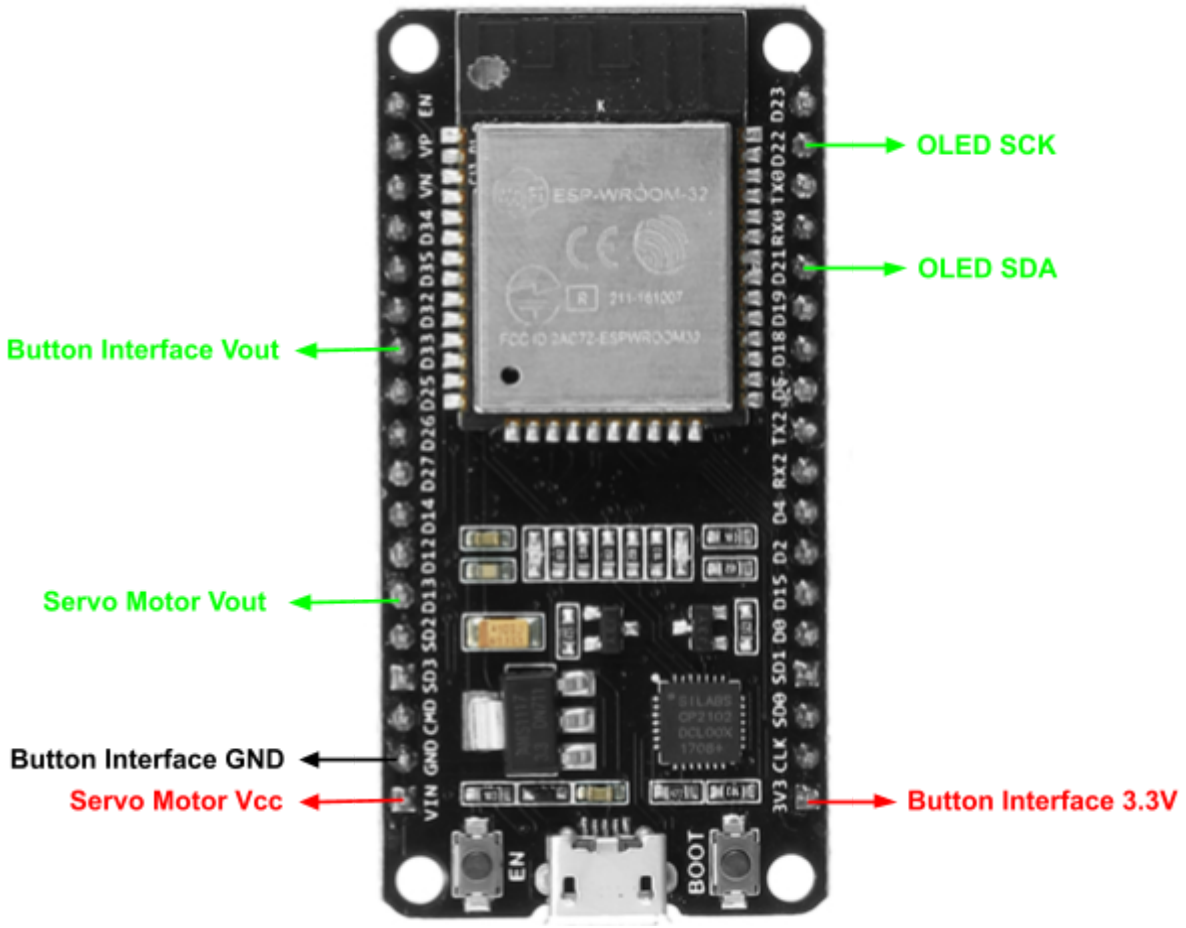


Fig 29: The wire connection of ESP32

5.4.3. Button Interface

A button interface is designed and mounted on the top plate of our dispenser for the selection of the table number. There are six push buttons TE 1825910-6 in total on the interface. One side of any two buttons are connected in series with one resistor in between them. They are supplied with 3.3V from the ESP32 board. The other side of all the buttons are grounded. The signal output (which is the voltage across the first resistor) of the button set is connected to Analog Pin 33 of ESP32 board. When different buttons are pressed, a different number of resistors are presented in between the ground and the power supply. This will result in a change in the voltage reading of the Analog Pin which will be used for distinguishing the Table selected.

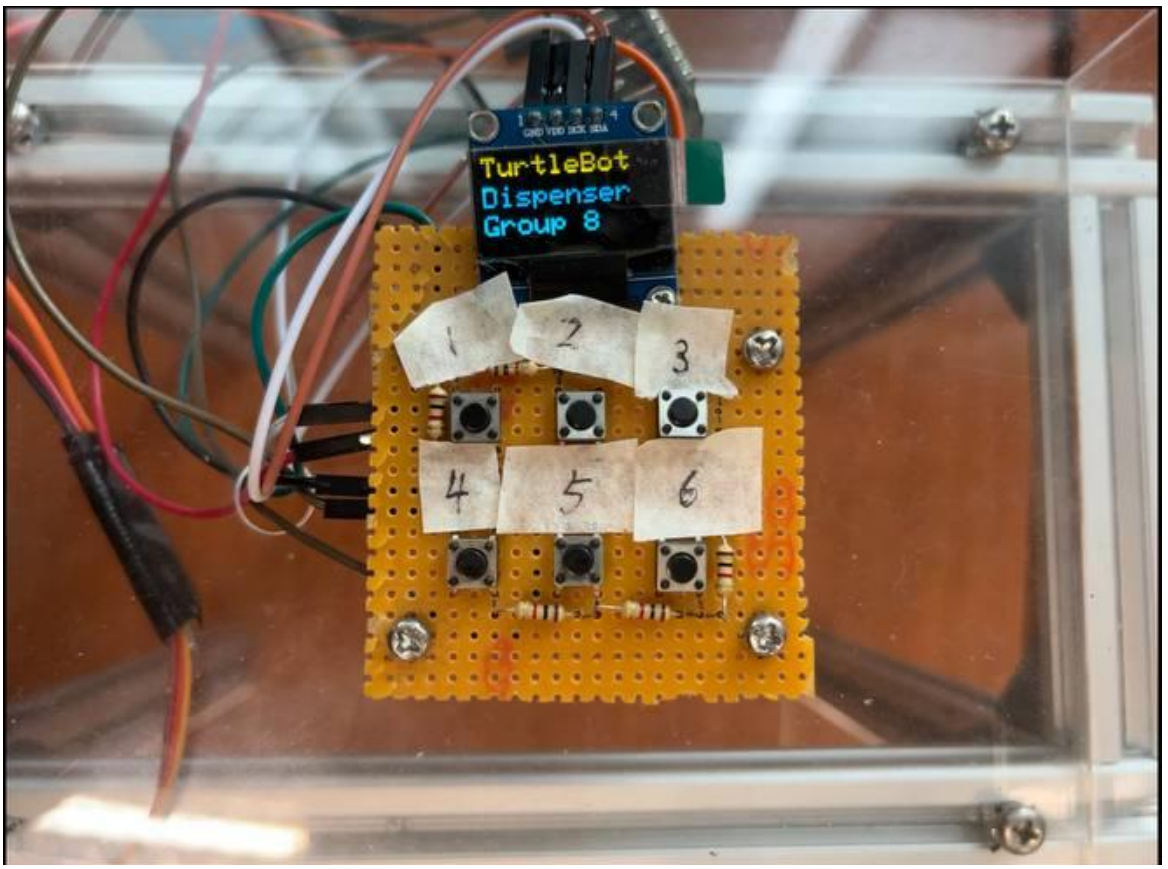


Figure 30: The Placement of button interface on the top plate of the dispenser

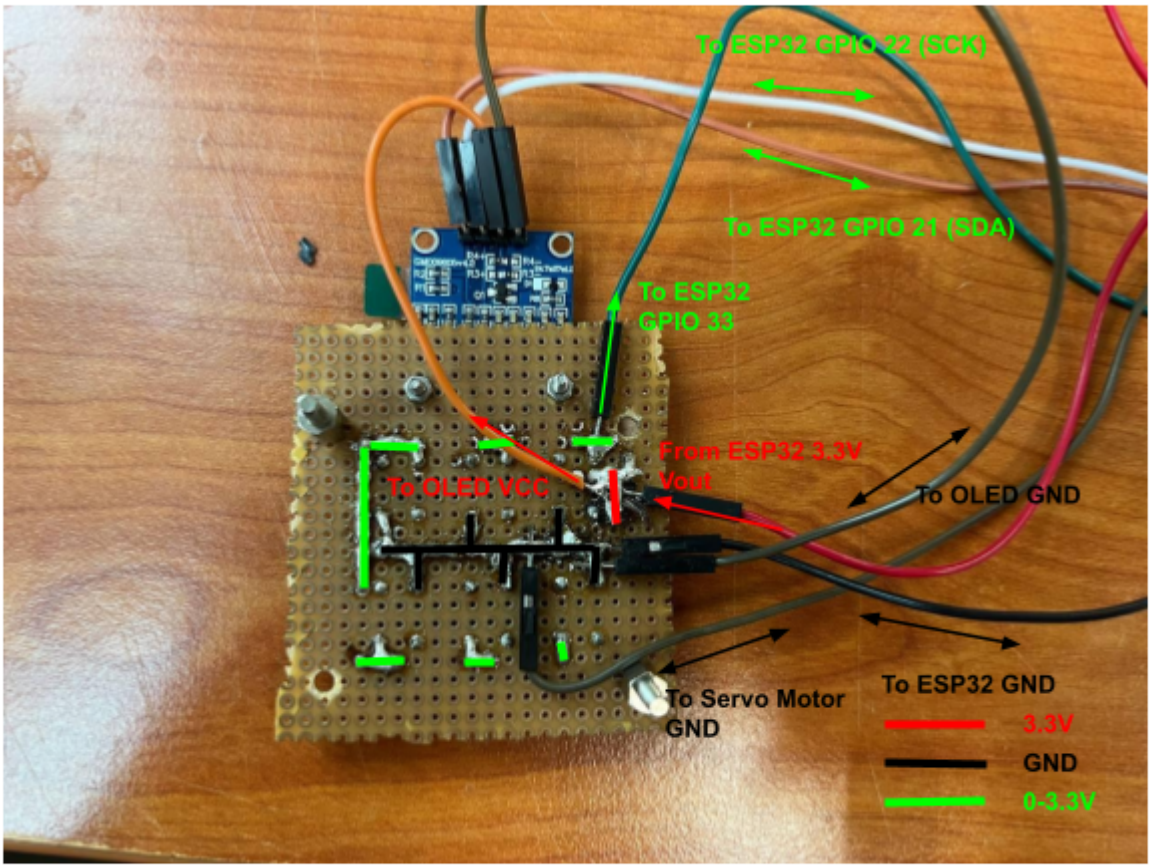
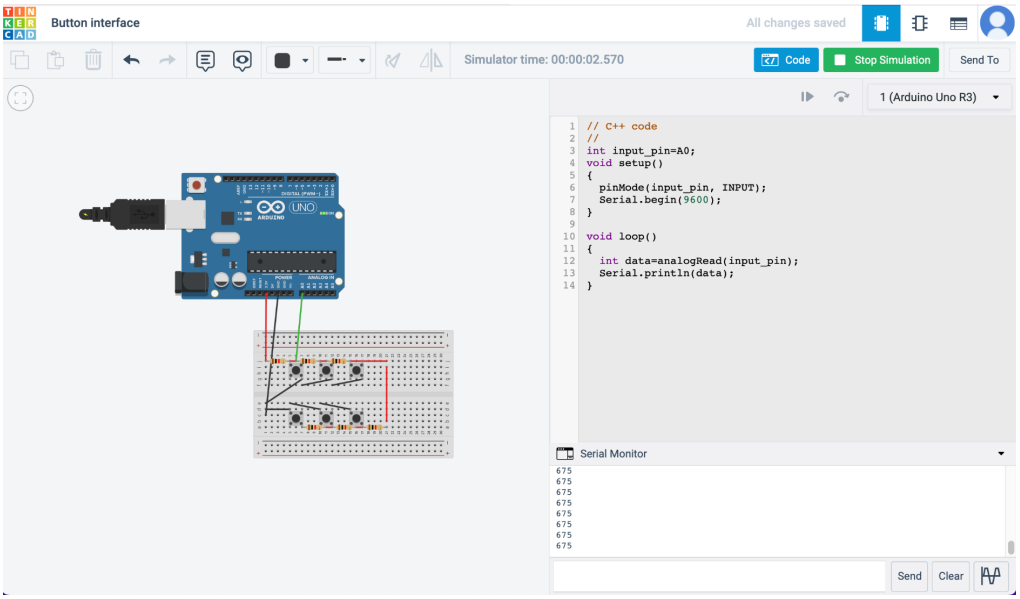
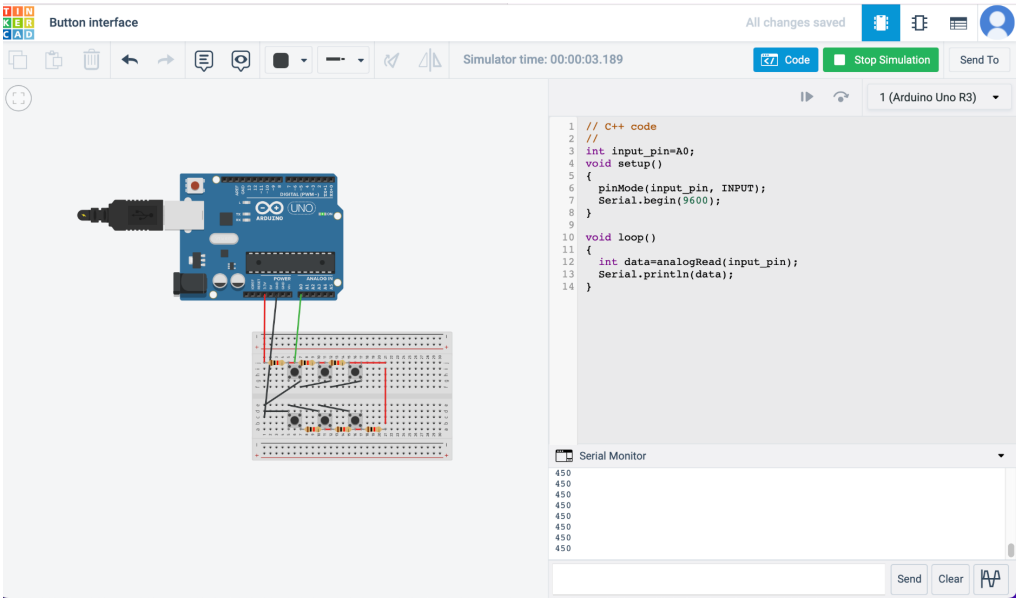
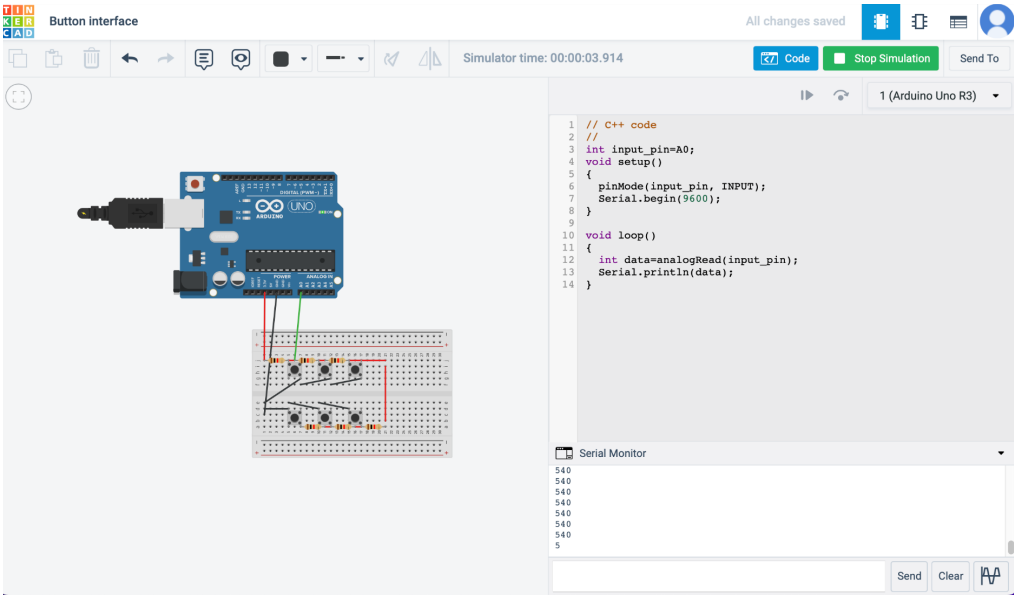


Fig 31: The wire connection of the button interface







Above figures are TinkerCAD illustrations on how the reading from analog Pin A0 shown on the serial monitor changes when different buttons on the interface are pressed. (The first figure is the reading when no button is pressed, The second figure is the reading when button 3 is preseed while the third figure is the reading when button 5 is pressed)

Here, an Arduino Uno is used to represent the ESP32 board used in our robot and the analog Pin A0 in Arduino Uno corresponds to the analog Pin 33 of the ESP32 board.

5.4.4. OLED Screen

One OLED-128O064D-BPP3N00000 screen is mounted on our button interface to display the state of the dispenser. It is powered by the 3.3V output pin of ESP32 board together with the button interface. The message “WIFI not connected” and “MQTT broker not connected” will be displayed on the screen if the dispenser is not connected to a hotspot or MQTT broker respectively. If both connections are secured, the OLED will display the lines “TurtleBot Dispenser Group 8” continuously. When any of the buttons on the interface is pressed, the message will change into “Delivering can to table X” (X is the number of the table corresponding to the button pressed) and the message will stay there until the button of a different table is pressed.

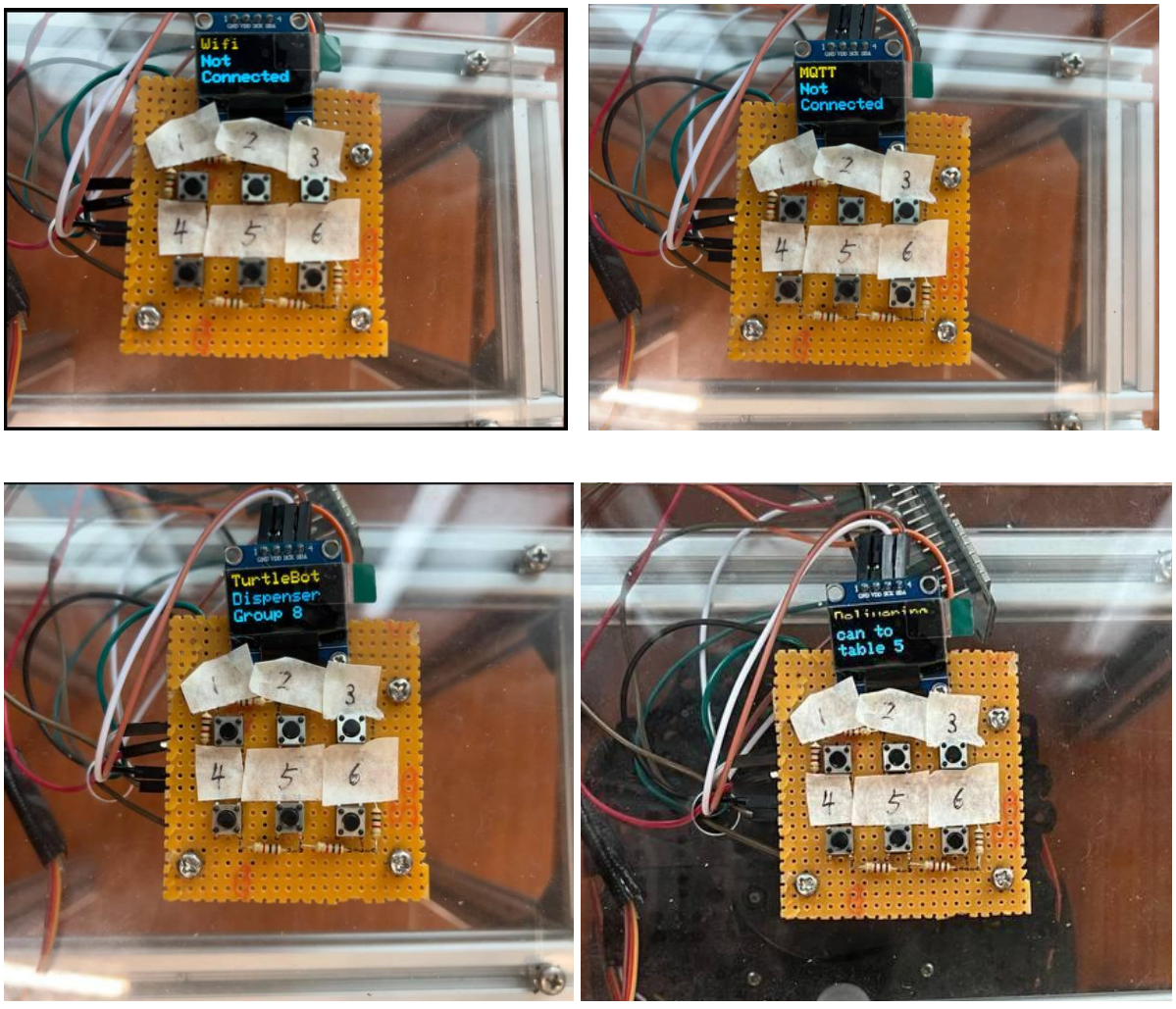
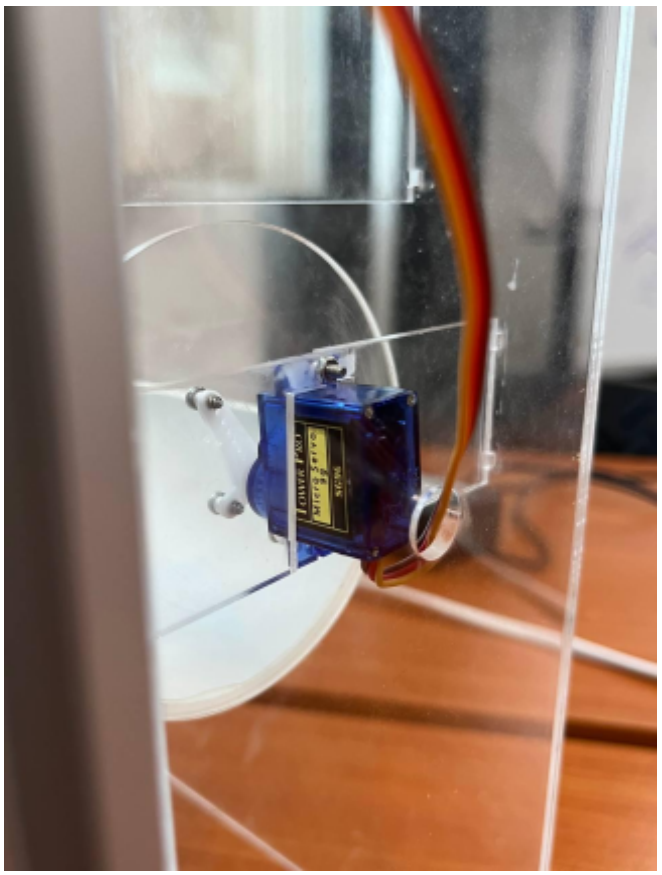


Figure 32 -35: The display on the LED screen when the dispenser is not connected to WIFI; not connected to MQTT broker; in standby mode and when Table 5 button is pressed

5.4.5. Servo Motor

The servo motor is placed on a stand in the middle of the back part of the dispenser, holding and turning the can holder to release the can to the robot. It is controlled by PWM signals from Digital Pin 13 of ESP32 board and powered by the Vin pin of the board (which will output 5V when the board is powered by 5V via USB). It is set to turn 180 degrees every time it is activated and return to the starting position for the next can after 3 seconds.



5.5. Electrical Calculation

5.5.1. Power Consumption Calculation for Robot

Compoenent	Quntity	Operating Current/A	Operating Voltage/V	Power Consumption/W
Robot(start-up)	1	0.91	11.1	10.101
Robot(standby)	1	0.59	11.1	6.549
Robot(normal operation)	1	0.77	11.1	8.547
IR sensor module	2	0.00006	3.3	0.000396

5.5.2. Maximum Operation Time for Continuous Operation

Assume start up takes about 30s, and after starting up, the robot continues in normal operations

Assume also that the battery will need to be charged after its current capacity drop below 20% (as any capacity lower than 20% will cause damage to the battery)

Maximum operation time

$= (19.98 \times 60 \times 60 - 10.101 \times 30) \div 8.547 = 8380.13s = 139.67mins$

5.5.3. Maximum Number of Delivery Mission Completable

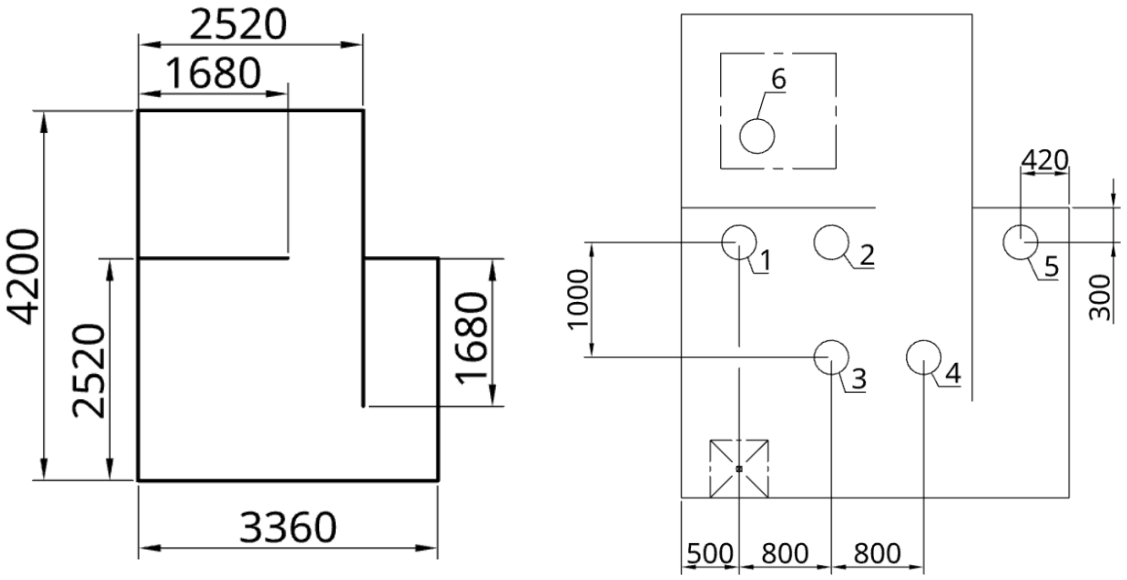


Fig 36: Layout map of the restaurant

Assuming that the robot will always try to move in straight line and turn 90 degrees when moving between the dispenser and the robot in the restaurant (which means that no rerouting due to obstacles)

Assuming also that the robot will always stop 15 cm away from the table when it delivers the can and reaches the middle of the dispenser zone every time it docks into the dispenser

From the restaurant layout map, we can hence calculate the average distance travelled by the robot and the average number of 90 degrees rotation that the robot has to take to one table

The total distance travelled

$= 1820\ (to\ Table\ 1) + 2620\ (to\ Table\ 2) + 1620\ (to\ Table\ 3) + 2420\ (to\ Table\ 4)$

$+ 4410\ (to\ Table\ 5) + 6300\ (to\ Table\ 6,\ estimated\ using\ the\ middle\ of\ the\ Table\ 6\ area)$

$= 19190mm$

The average distance travelled $= 19190 \div 6 = 3198.3mm$

The average number of 90 degrees rotation

$= (0 + 1 + 1 + 1 + 2 + 3) \div 6 = 1.3333$

Assuming 30s waiting time in the dispenser and the table for the can to be loaded by the waiter or taken away by the customers

The average power consumption to complete one delivery mission

$$= 2 \times (30 \times 6.549 + (3198.3 \div 220 + 90 \times 1.3333 \div 155) \times 8.547$$

$$= 423.56W$$

Max number of mission cycle that can be completed before charging

$$= 19.98 \times 60 \times 60 \times 0.8 \div 423.56 = 135$$

5.5.4. Power Consumption Calculation for Dispenser

Assume the servo motor consume the same amount of power in standby mode as the operation mode

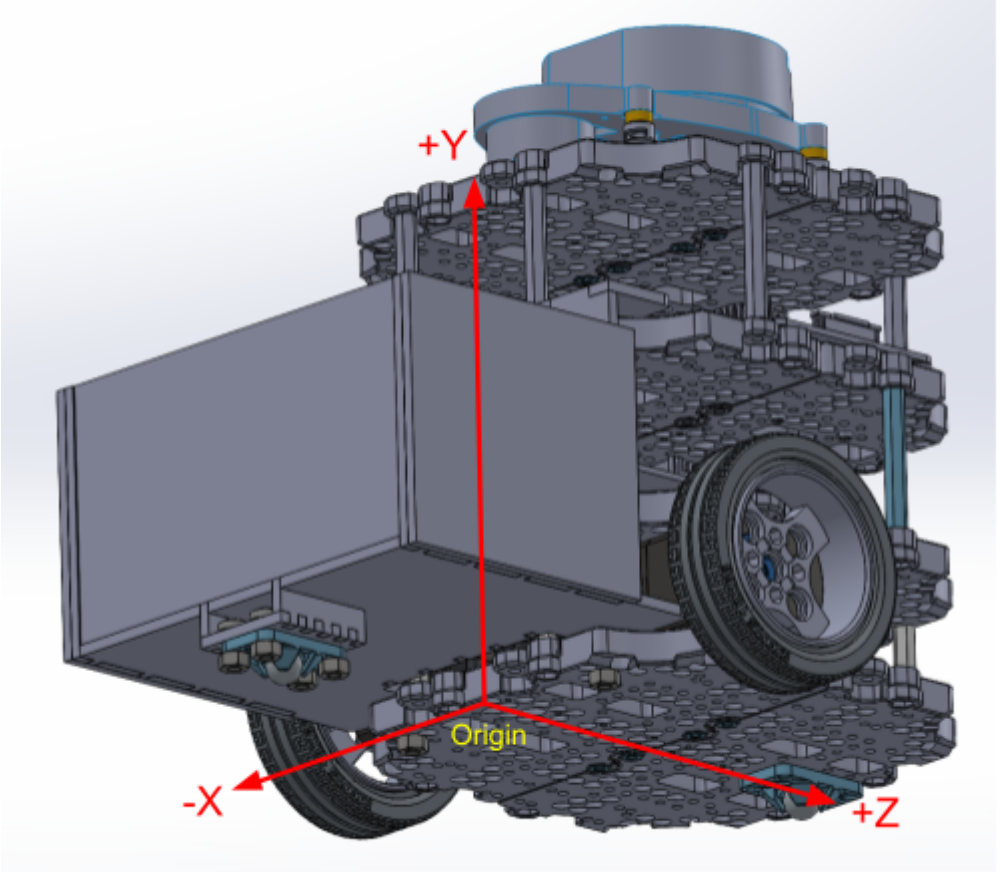
Compoenent	Quntity	Operating Current/A	Operating Voltage/V	Power Consumption/W
ESP32 Board	1	0.24	5	1.2
Servo motor	1	0.3	5	1.5
OLED screen	1	0.01	3.3	0.033
			Overall power consumption by dispenser/W	2.733

6. Mechanical

6.1. Mechanical Calculation

*Fasteners are excluded from calculation

6.1.1. Delivery Robot



S/No.	Part	Mass (g)	Part's CG from Origin (cm)			m*x	m*y	m*z
			X	Y	Z			
1	ball wheel	4.6	0	0.5	2	0	2.3	9.2
2	left motor	56	-5	4	9	-280	224	504
3	right motor	56	5	4	9	280	224	504
4	left wheel	29.5	-8	4	10	-236	118	295
5	right wheel	29.5	8	4	10	236	118	295
6	battery	137.5	0	4	7	0	550	962.5
7	OpenCR	60	0	7	7	0	420	420
8	Rpi 3B+	50	0	11.5	4	0	575	200
9	USB2LDS	2.6	0	11	8.5	0	28.6	22.1
10	LIDAR	111.6	0	18	8	0	2008.8	892.8
11	1st plate	39.5	0	1.5	7	0	59.25	276.5
12	2nd plate	39.5	0	5	7	0	197.5	276.5
13	3rd plate	39.5	0	10	7	0	395	276.5
14	4th plate	39.5	0	14.5	7	0	572.75	276.5
15	black standoff	1.6	2	6.5	3	3.2	10.4	4.8
16	black standoff	1.6	-2	6.5	3	-3.2	10.4	4.8
17	black standoff	1.6	2	6.5	11	3.2	10.4	17.6
18	black standoff	1.6	-2	6.5	11	-3.2	10.4	17.6

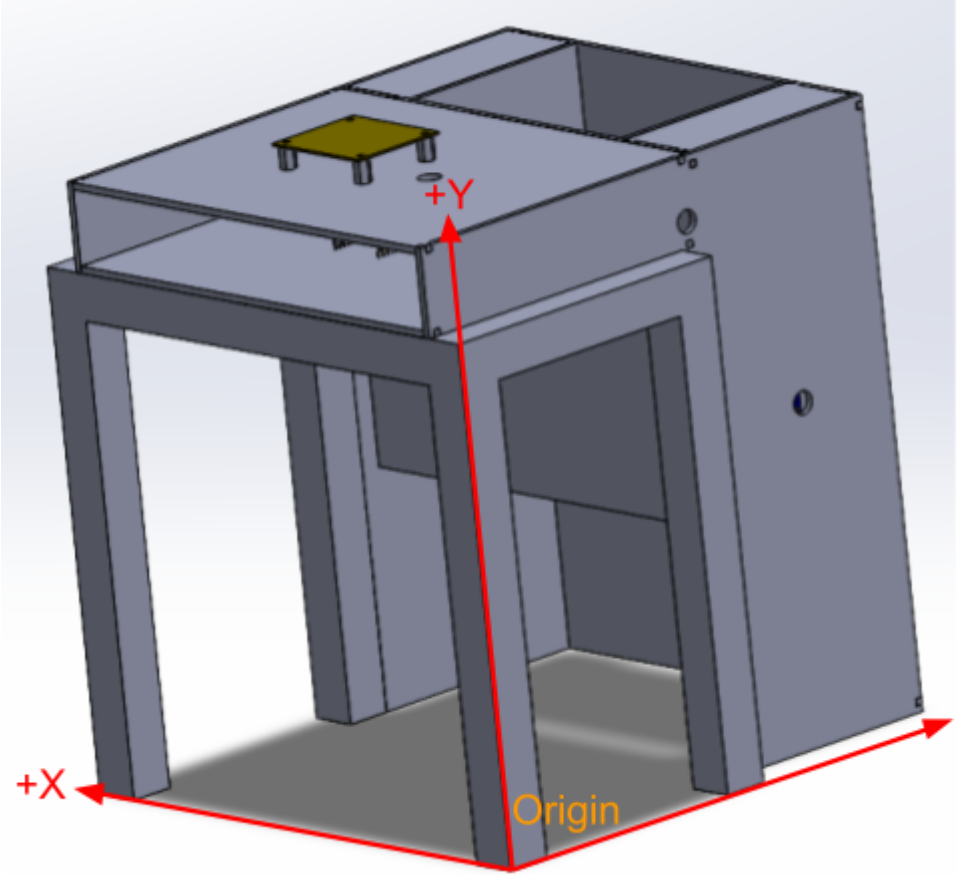
19	black standoff	1.6	3	11	2	4.8	17.6	3.2
20	black standoff	1.6	-3	11	2	-4.8	17.6	3.2
21	black standoff	1.6	3	11	6	4.8	17.6	9.6
22	black standoff	1.6	-3	11	6	-4.8	17.6	9.6
23	black standoff	1.6	2	15.5	5.5	3.2	24.8	8.8
24	black standoff	1.6	-2	15.5	5.5	-3.2	24.8	8.8
25	black standoff	1.6	2	15.5	8.5	3.2	24.8	13.6
26	black standoff	1.6	-2	15.5	8.5	-3.2	24.8	13.6
27	35mm standoff	4.9	2.5	3	0.5	12.25	14.7	2.45
28	35mm standoff	4.9	-2.5	3	0.5	-12.25	14.7	2.45
29	35mm standoff	4.9	2.5	3	13	12.25	14.7	63.7
30	35mm standoff	4.9	-2.5	3	13	-12.25	14.7	63.7
31	45mm standoff	6.6	6.5	8	4.5	42.9	52.8	29.7
32	45mm standoff	6.6	-6.5	8	4.5	-42.9	52.8	29.7
33	45mm standoff	6.6	1.25	8	13	8.25	52.8	85.8
34	45mm standoff	6.6	-1.25	8	13	-8.25	52.8	85.8
35	45mm standoff	6.6	2.5	12	0.5	16.5	79.2	3.3
36	45mm standoff	6.6	-2.5	12	0.5	-16.5	79.2	3.3
37	45mm standoff	6.6	2.5	12	13	16.5	79.2	85.8
38	45mm standoff	6.6	-2.5	12	13	-16.5	79.2	85.8
39	45mm standoff	6.6	6.5	12	9.5	42.9	79.2	62.7

40	45mm standoff	6.6	-6.5	12	9.5	-42.9	79.2	62.7
41	battery holder	0.8	0	2	2	0	1.6	1.6
42	battery holder	0.8	1	2	12	0.8	1.6	9.6
43	battery holder	0.8	-1	2	12	-0.8	1.6	9.6
44	battery holder	0.8	2.5	2	6	2	1.6	4.8
45	battery holder	0.8	-2.5	2	6	-2	1.6	4.8
46	payload	223.6	0	5.2	-4.3	0	1162.72	-961.48
47	ball wheel	4.6	0	0.5	-6.6	0	2.3	-30.36
48	microswitch	2.5	0	1.9	-4.6	0	4.75	-11.5
49	left IR sensor	8	-6.7	3.5	-1.6	-53.6	28	-12.8
50	right IR sensor	8	6.7	3.5	-1.6	53.6	28	-12.8

Total mass (g): 1050.8

Center of mass (cm): (0, 7.3, 4.8)

6.1.2. Dispenser



S/No.	Part	Mass (g)	Part's CG from Origin (cm)			m*x	m*y	m*z
			X	Y	Z			
1	Aluminium Profile	692.7	12	15.1	7.1	8312.4	10459.77	4918.17
2	Acrylic Piece	816.1	11.9	17.9	16.3	9711.59	14608.19	13302.43
3	Can Holder	100.5	12	12.6	19.8	1206	1266.3	1989.9
4	Wooden Rod	0.7	20.2	14.2	19.8	14.14	9.94	13.86
5	Protoboard	10	13.2	26.9	7.5	132	269	75
6	ESP32	10	12	23.7	7.5	120	237	75

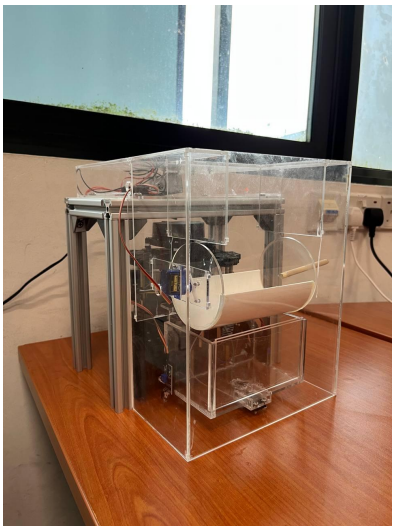
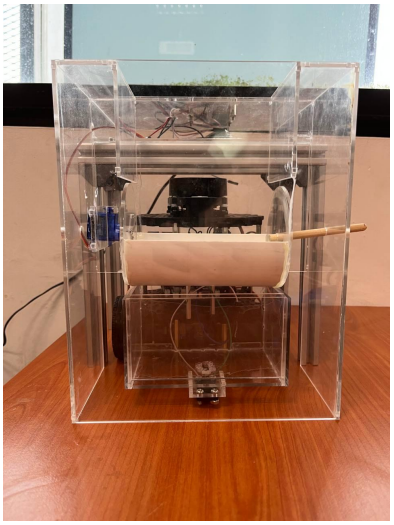
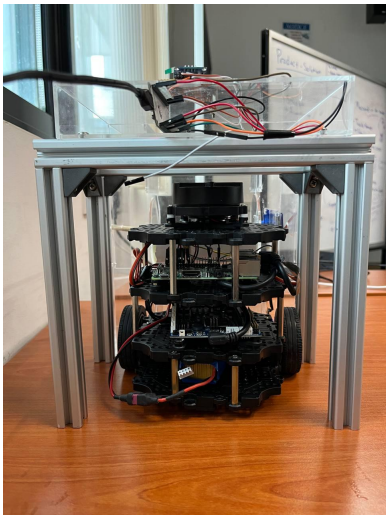
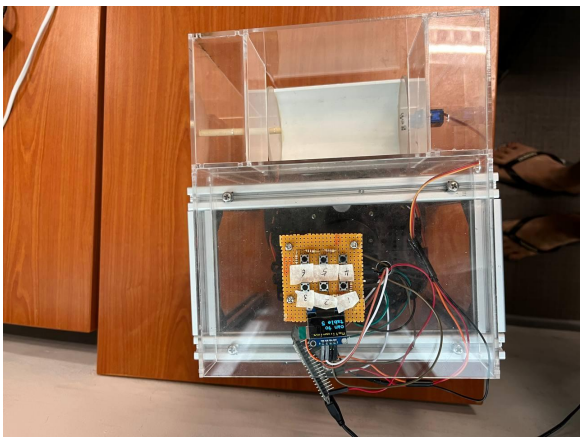
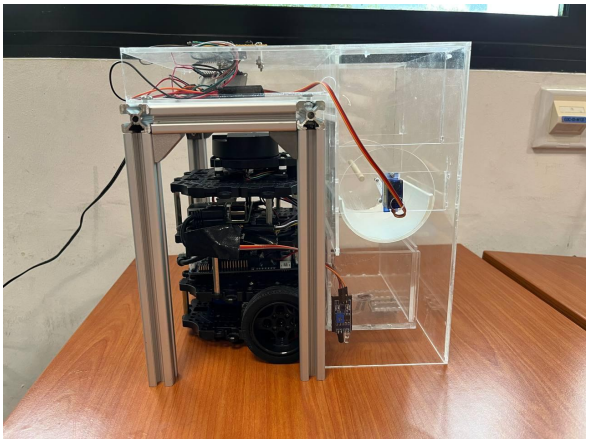
7	Servo	9	3.8	14.7	19.8	34.2	132.3	178.2
8	Standoffs	6	13.2	26.3	7.5	79.2	157.8	45

Total mass (g): 1645

Center of mass (cm): (11.9, 16.5, 12.5)

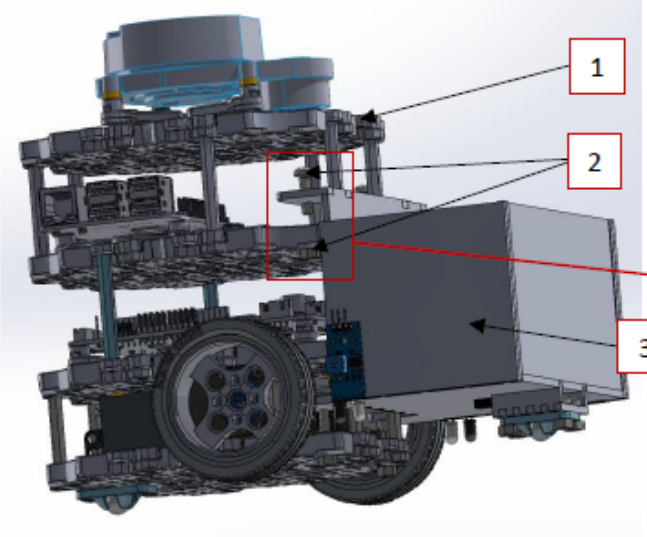
6.2. Mechanical System Assembly

All acrylic pieces are glued together by acrylic glue.

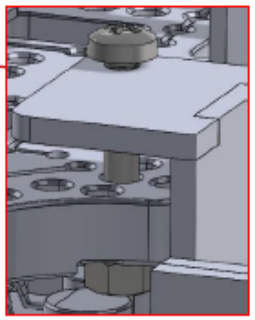






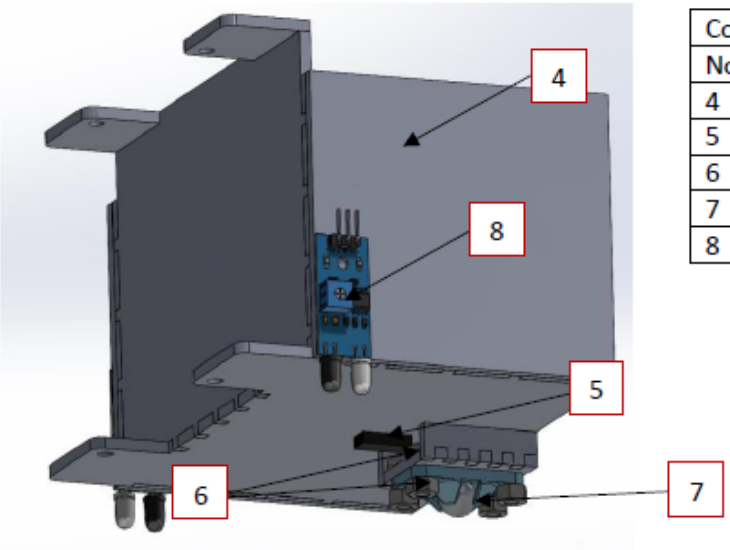
6.2.1. Delivery Robot

Container is attached to the robot by M4 screws and nuts. Ball caster wheel is attached to the container by M4 screws and nuts. Microcontroller is glued to the container by hot glue. The IR sensors are taped to both sides of the container.

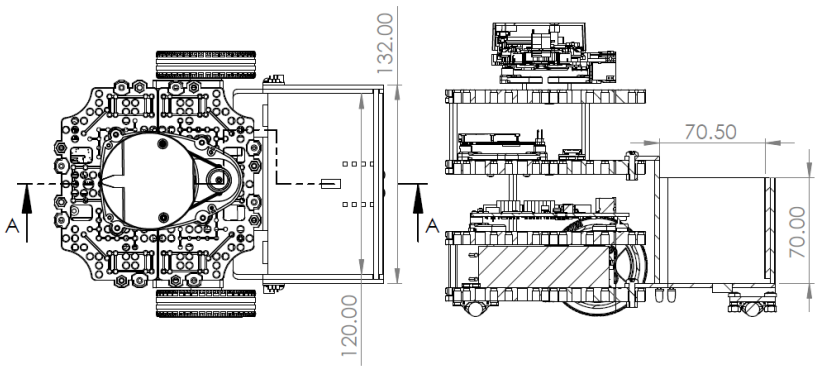


Robot		
No.	Description	Qty
1	Turtlebot3 Burger Model	1
2	M4 Screw & Hex Nut	4
3	Payload	1



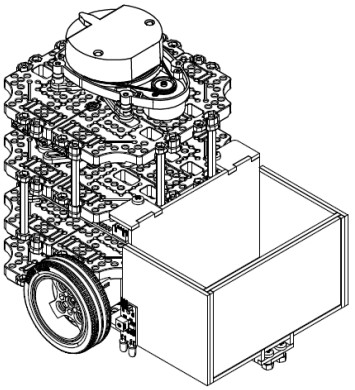


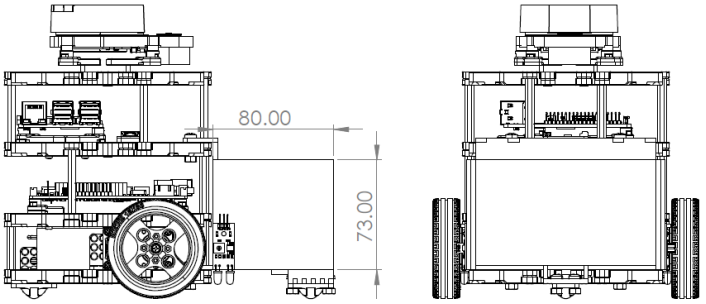
Container		
No.	Description	Qty
4	3mm Acrylic	13
5	Microswitch	1
6	M4 Screw & Hex Nut	4
7	Ball Caster Wheel	1
8	IR Sensor	2



SECTION A-A
SCALE 1 : 3

ALL UNITS IN MMS

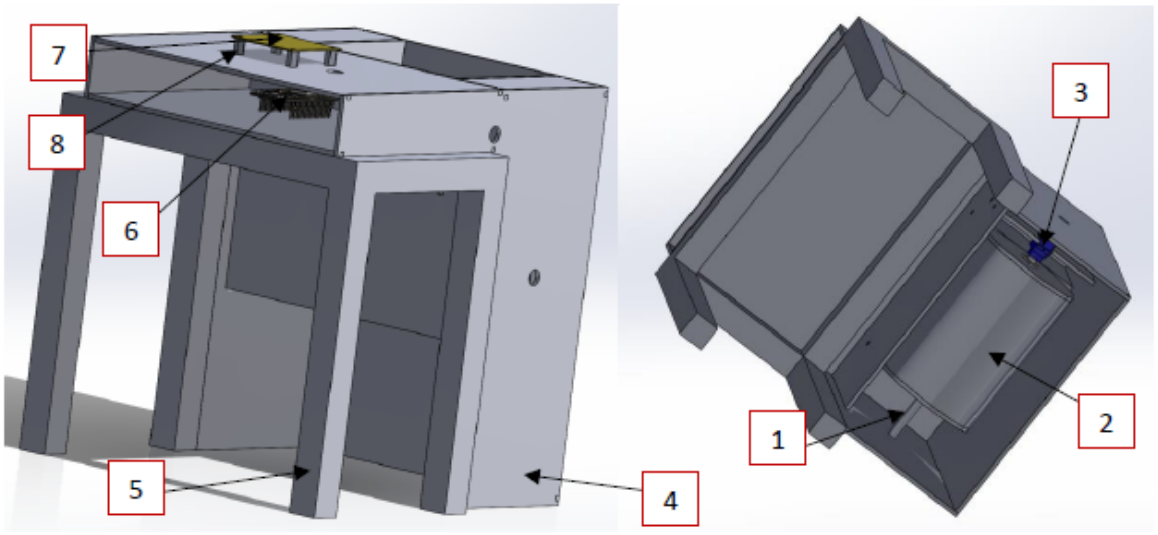




6.2.2. Dispenser

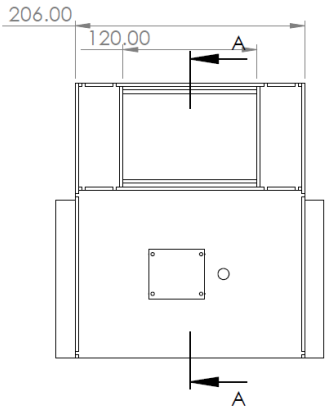
The aluminium profile is held together by rhombus nuts and screws. The dispenser rests upon the aluminium profile and is held by rhombus nuts and screws. Servo is attached to acrylic by M2 screw and nut. Can holder is attached to the servo by M2 self tapping screws. The protoboard is attached to the dispenser by M3 screws, nuts and standoffs.

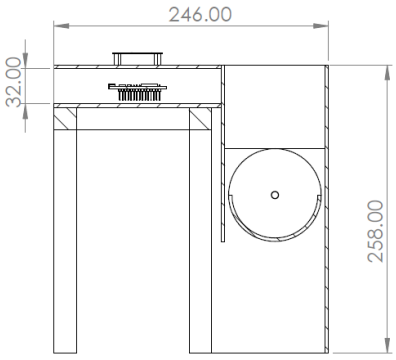
*Fasteners are excluded from drawing



Dispenser		
No.	Description	Qty
1	43mm Wooden Rod ø¼"	1
2	120mm PVC Pipe (ID 80mm)	1
3	Servo SG90	1
4	3mm Acrylic	13
5	20x20 Aluminium Profile	8
6	ESP32	1
7	Button Interface	1
8	M3 Standoff	4

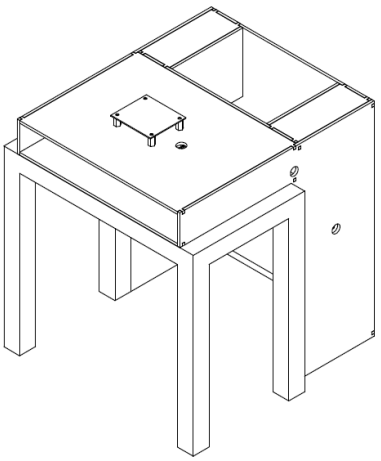
Fasteners		
No.	Description	Qty
9	M2 Self Tapping Screw	5
10	M2 Screw	1
11	M2 Nut	1
12	M3 Screw	4
13	M3 Nut	4
14	M5 Screw	12
15	M5 Rhombus Nut	12
16	Aluminium Profile Bracket	8

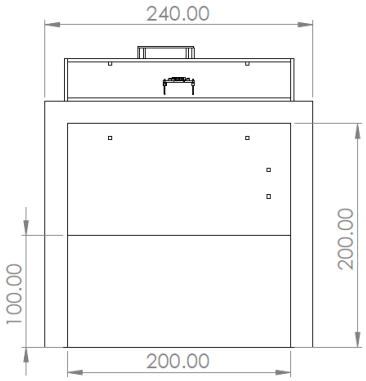


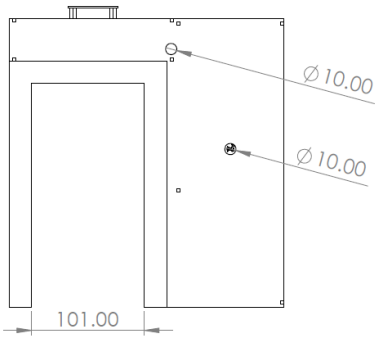


SECTION A-A
SCALE 1 : 4

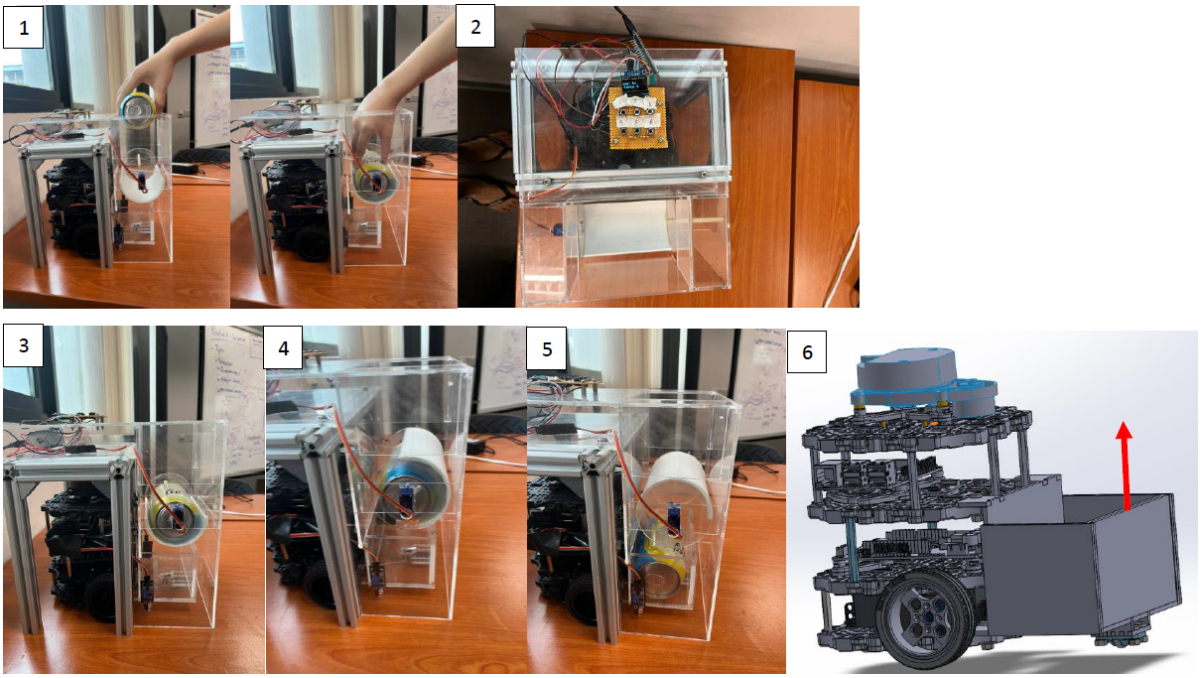
ALL UNITS IN MMS







6.3. Delivery Robot and Dispenser Mechanism



1. Place the can inside the dispenser as gently as possible.
2. Press the corresponding button on the interface.
3. The can will rest inside the dispenser until the dispenser detects that the robot has returned.
4. Once the robot has returned, the servo will turn the can holder to release the can.
5. The can falls into the container of the robot, pressing the microswitch on the container.
6. Once the robot has reached the table, the back can be removed to take out the can.

7. Software

7.1. Software Assembly

7.1.1. Setting Up the Environment

1. Firstly, follow the instructions provided in this link, specifically the "Foxy" tab, starting from Section 3.1.1, to install Ubuntu 20.04 and ROS 2 Foxy on your laptop.

2. Verify the functionality of the ROS development environment by creating a basic working publisher and subscriber, following the instructions provided in this link.

3. Afterwards, proceed to flash the ROS2 Foxy image to the SD card on the TurtleBot3's RPi, utilising the Ubuntu Operating System (OS), as per the instructions provided in this link.

4. Repeat step 2 to ensure that the ROS development environment on the RPi is also functioning as intended.

5. Execute the publisher on the RPi, run the subscriber on your laptop, and confirm that they are communicating without any errors. Then, swap the devices to validate that they function effectively as both publishers and subscribers.

7.1.2. Message Queuing Telemetry Transport Setup

1. On the RPi, install the 'paho-mqtt' package, the Eclipse Paho MQTT Python client library, using either one of the following methods by entering the following into the command-line interface (CLI)

    a. apt install
    `sudo apt install paho-mqtt`
    b. pip install
    `pip install paho-mqtt`

2. On the laptop, in the Ubuntu OS, install 'mosquitto', a broker for the MQTT protocol version 5.0/3.1.1/3.1 using the following command.

    `sudo snap install`

3. Copy the mosquitto_example.conf file into a new file named mosquitto.conf.

```
cd /var/snap/mosquitto/common
sudo cp mosquitto_example.conf mosquitto.conf
```

4. Edit the mosquitto.conf file by adding the following lines at the end of the file.

```
allow_anonymous true
listener 1883
```

5. Stop and start the mosquitto service.

```
sudo systemctl stop snap.mosquitto.mosquitto.service
sudo systemctl start snap.mosquitto.mosquitto.service
```

6. Check that the mosquitto broker is running

```
sudo systemctl status snap.mosquitto.mosquitto
```

7.1.3. Installing the program

1. Before cloning the program from the GitHub repository, we need to create a ROS2 package on our laptop with the following commands.

```
cd ~/colcon_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd auto_nav/auto_nav
```

2. We will then move the `__init__.py` file from the current directory temporarily to the parent directory, using the below command

```
mv __init__.py ..
```

3. Now, we can clone the repository onto our laptop

```
git clone git@github.com:MoeySeanJean/r2auto_nav.git .
```

4. Then, we can move our __init__.py file back into the current working directory

```
mv ../__init__.py .
```

5. Add the following lines to the .bashrc file on the laptop

```
export TURTLEBOT3_MODEL=burger
alias rteleop='ros2 run turtlebot3_teleop teleop_keyboard'
alias rslam='ros2 launch turtlebot3_cartographer cartographer.launch.py'
```

6. Now, we will move on to install the code for the RPi. 'ssh' into the RPi, using the following command, where the Xs that follow 'ubuntu@' is the IP address of the RPi.

```
ssh ubuntu@XXX.XX.XX.X
```

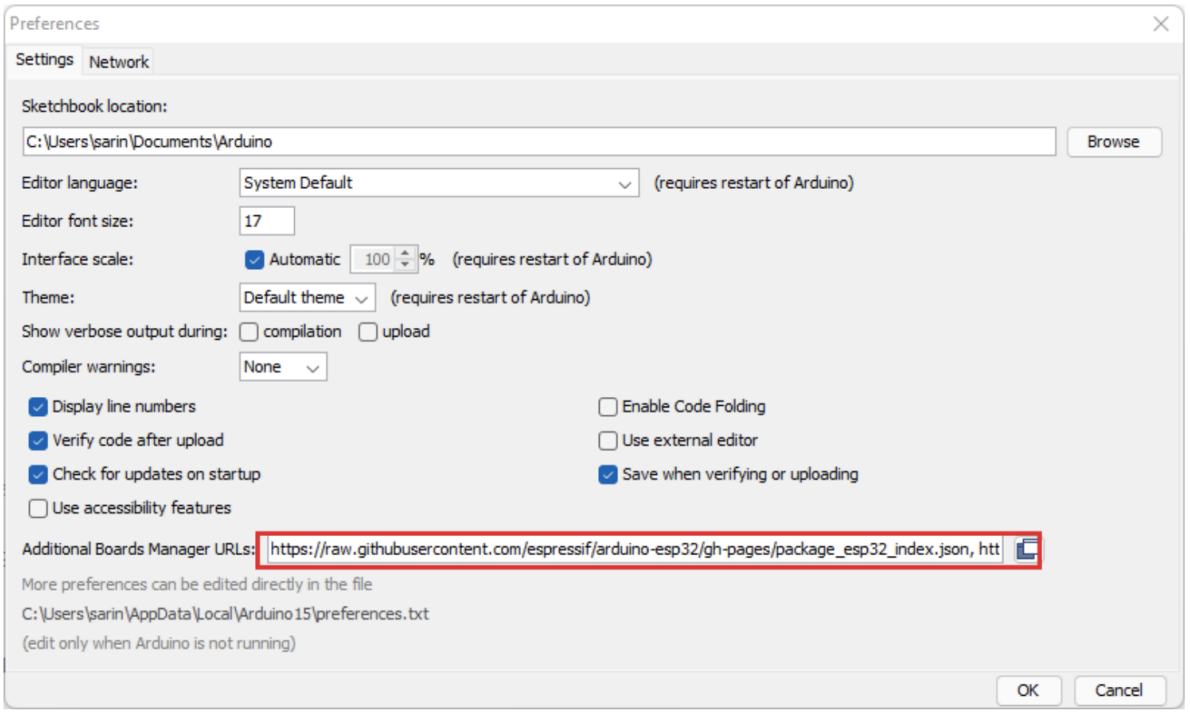
7. Repeat steps 1 to 4, but with the following highlighted changes. The series of commands to be entered should be as follows.

```
cd ~/turtlebot3_ws/src
ros2 pkg create --build-type ament_python auto_nav
cd navigation_eg2310/navigation_eg2310
mv __init__.py ..
git clone git@github.com:applepiofmyeye/navigation_eg2310.git .
mv ../__init__.py .
```

7.1.4. ESP32 Setup

1. (Using Arduino Integrated Development Environment (IDE), on Windows OS) In your Arduino IDE, go to File> Preferences, and enter the following into the "Additional Board Manager URLs" field:

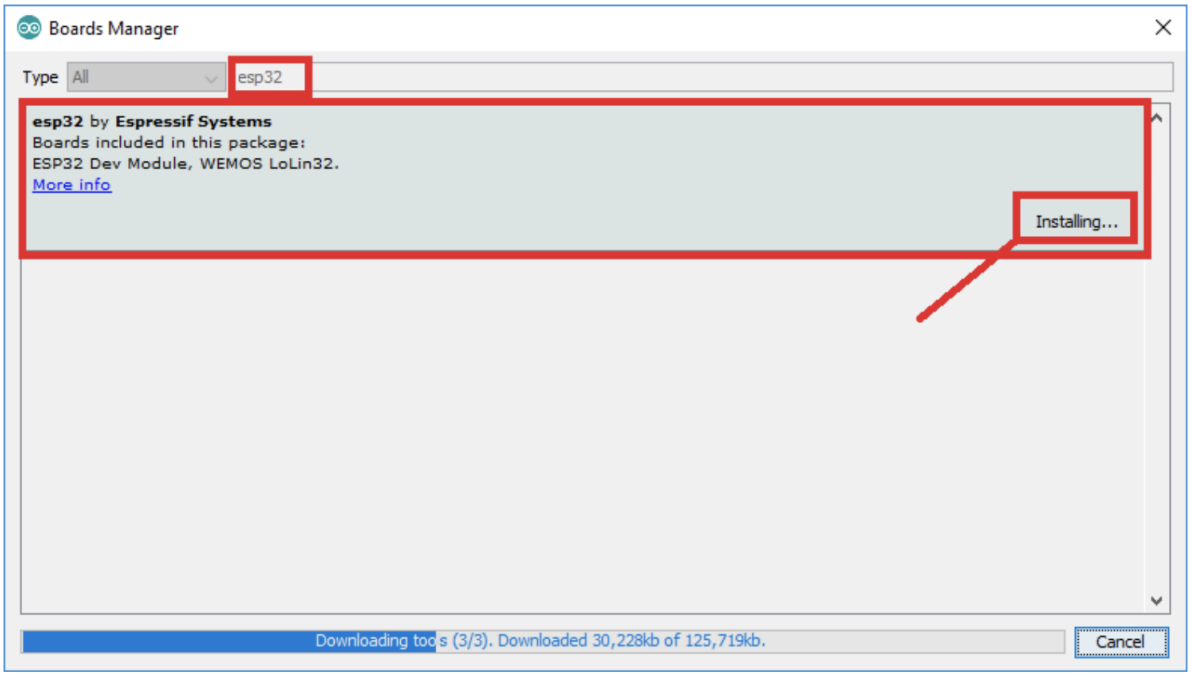
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_esp32_index.json



Then click the “OK” button.

2. Open the Boards Manager. Go to Tools > Board > Boards Manager…

3. Search for ESP32 and press install button for the “ESP32 by Espressif Systems“:



4. Plug the ESP32 board to the laptop. With your Arduino IDE open, Select your Board in Tools > Board menu (for our case, we chose the DOIT ESP32 DEVKIT V1).

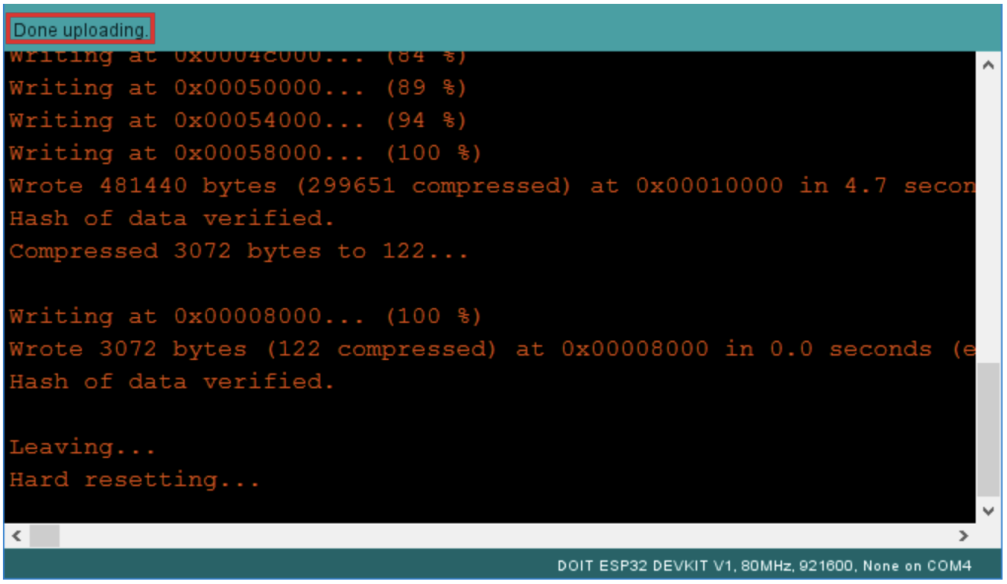
5. Select the port connected to the ESP32, under Tools > Port, an

6. From this link, copy the file titled “ESP.ino” and create a new Arduino sketch with this code.

7. Press the Upload button on the Arduino IDE and wait for the code to compile.

8. After it compiles, the IDE should display “Connecting…”. Press and hold the Boot button on the ESP32 until the upload begins.

9. If everything went as expected, the IDE should display a “Done uploading” message.



10. If the ESP32 has been connected to the OLED screen as mentioned in Section 4.4.4, it should show the displays as per mentioned.

7.2. Overall Logical Flow

7.2.1. Overview of Algorithm

On the RPi, the flowchart for the general algorithm is as shown in the following figure.

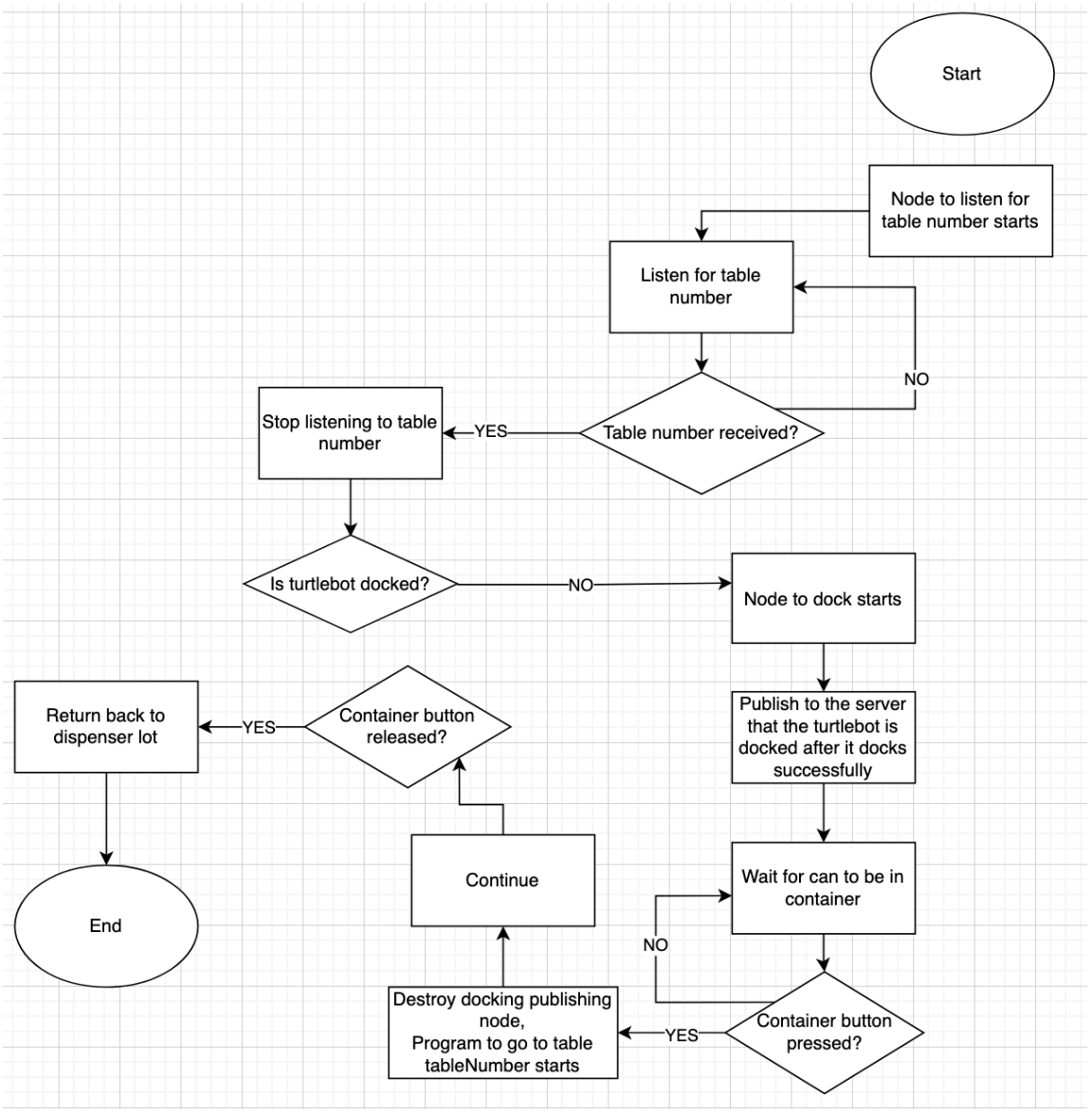


Figure 37

On the other hand, on the ESP32, the flowchart of the general algorithm is as illustrated below in Figure 38

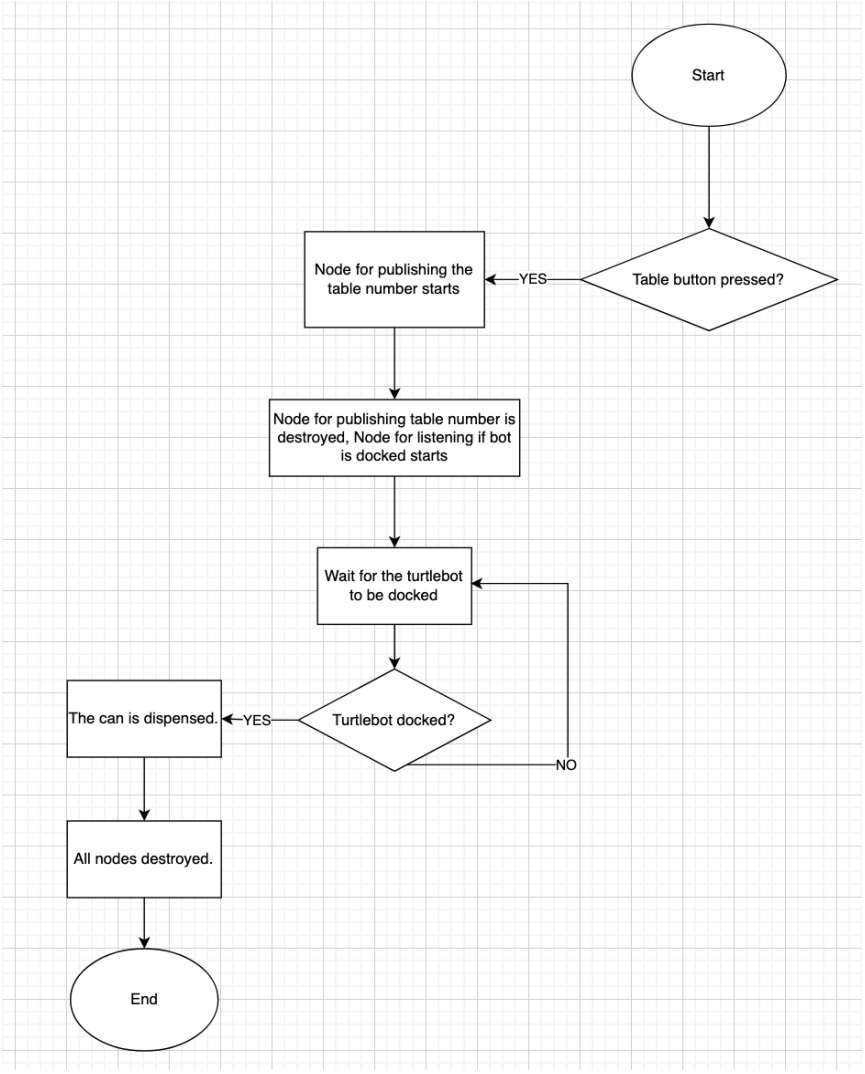


Figure 38

7.2.2. System Architecture

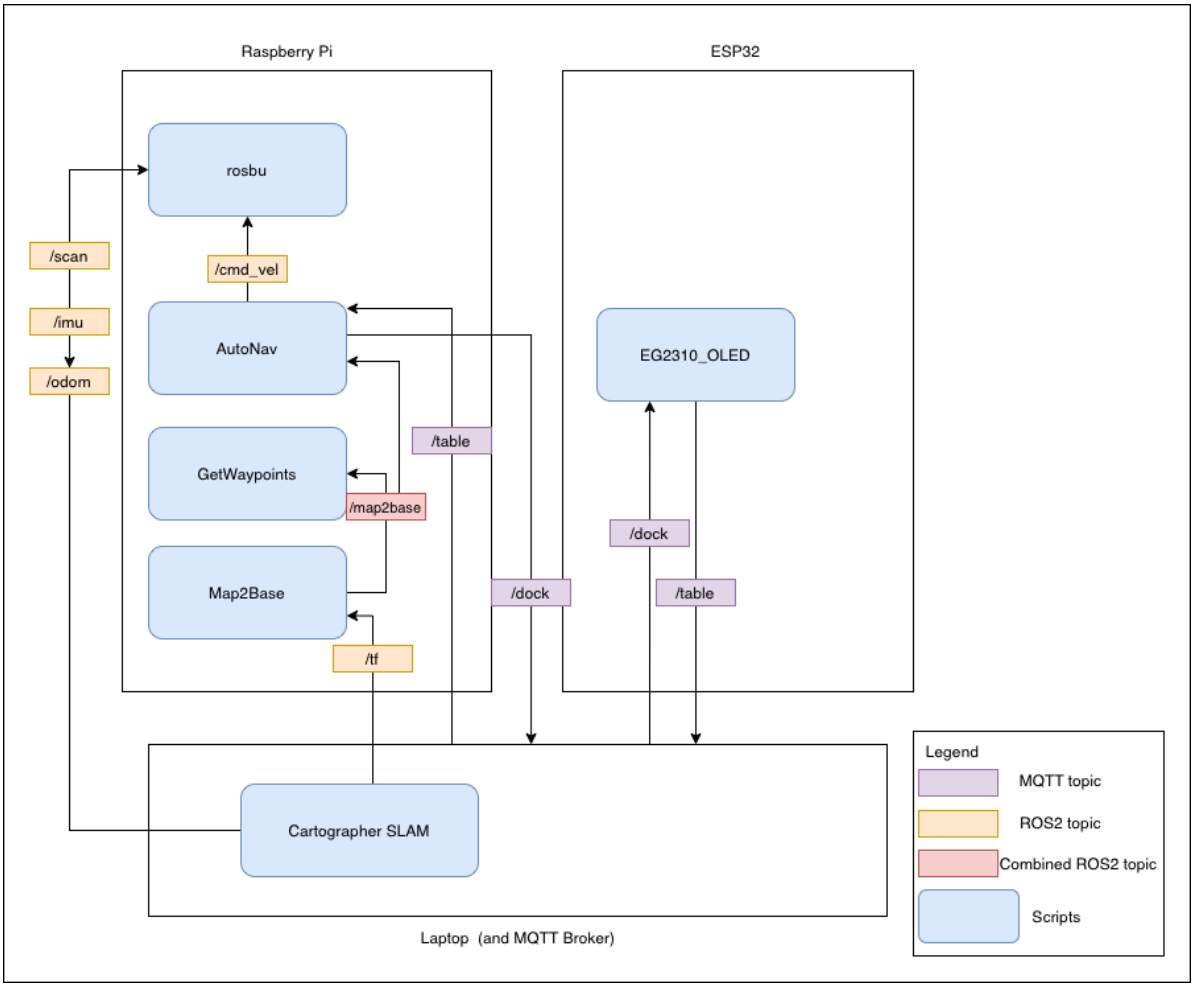


Figure 39

7.3. Mapping Software Design

Upon running the code, the main() function is run. This creates a new Waypoint node, and then prompts the user the following.

```
Press s to start:
```

If the user enters ‘s’, the get_waypoints() function is called.

```python
def main(args=None):
    rclpy.init(args=args)
    try:
        waypoint = Waypoint()
        start = input("Press s to start: ")
        if start == "s":
            waypoint.get_waypoints()
    except KeyboardInterrupt:
        waypoint.destroy_node()
        rclpy.shutdown()
```

```python
def get_waypoints(self):
    n = num_of_waypoints
    while n != 0:
        inp = input("Enter input: ")
        if inp == "w":
            checkpt_id = int(input("Enter checkpoint: "))
            print("saving..")
            rclpy.spin_once(self)

            data = [self.px, self.py, self.ox, self.oy, self.oz]
            waypoints[checkpt_id].extend(data)
            print(waypoints)
            n -= 1
        elif inp == "s":
            print("saving...")
            with open('waypoints.pickle', 'wb') as handle:
                pickle.dump(waypoints, handle, protocol=pickle.HIGHEST_PROTOCOL)
            n -= 1
        else:
            print("Please enter 's' or 'w' only.")
```

The function get_waypoints() loops n times, where n is the number of waypoints set in the start of the code as a variable (in our case, our num_of_waypoints is 12).
If the user input is 'w', the user will be prompted again to enter the checkpoint number that they are currently collecting. (Checkpoints should be collected while moving the TurtleBot using rteleop as mentioned in Section 7.1.3.

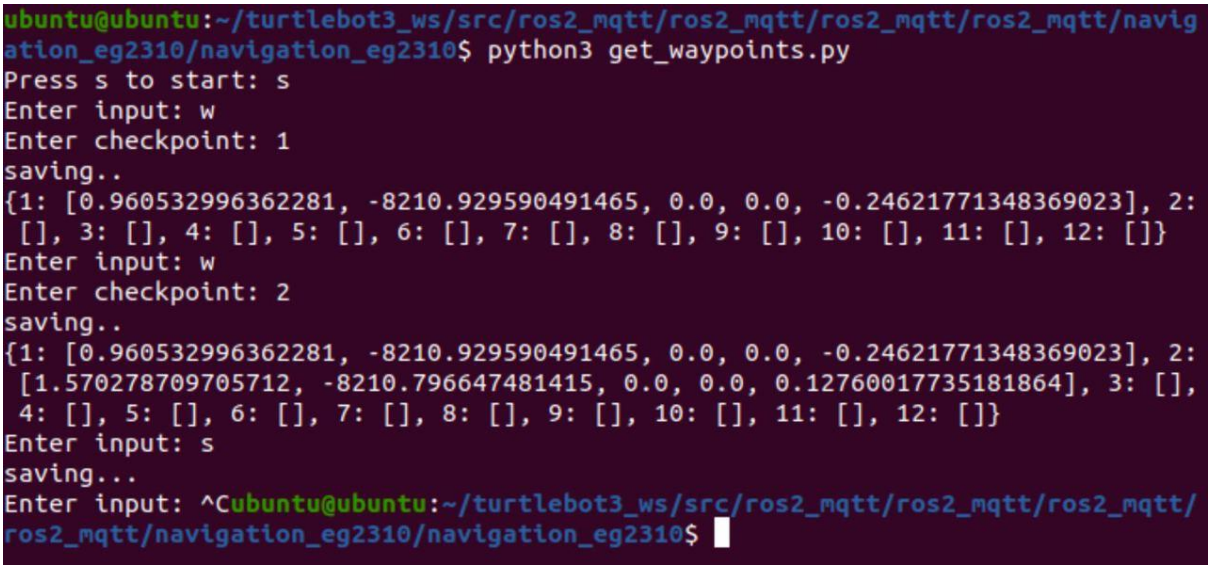
```
# code from https://automaticaddison.com/how-to-convert-a-quaternion-into-euler-angles-in-python/
def euler_from_quaternion(x, y, z, w):
    """
    Convert a quaternion into euler angles (roll, pitch, yaw)
    roll is rotation around x in radians (counterclockwise)
    pitch is rotation around y in radians (counterclockwise)
    yaw is rotation around z in radians (counterclockwise)
    """
    t0 = +2.0 * (w * x + y * z)
    t1 = +1.0 - 2.0 * (x * x + y * y)
    roll_x = math.atan2(t0, t1)

    t2 = +2.0 * (w * y - z * x)
    t2 = +1.0 if t2 > +1.0 else t2
    t2 = -1.0 if t2 < -1.0 else t2
    pitch_y = math.asin(t2)

    t3 = +2.0 * (w * z + x * y)
    t4 = +1.0 - 2.0 * (y * y + z * z)
    yaw_z = math.atan2(t3, t4)

    return roll_x, pitch_y, yaw_z # in radians
```

The function get_waypoints() is used to collect the Cartesian coordinates of the position of the TurtleBot (self.px and self.py), as well as the TurtleBot's current orientation in euler angles (self.ox, self.oy, self.oz) as published onto the /map2base topic.

When the user inputs 'w', they would be prompted to enter a checkpoint, and then the spin_once(self) function is called. This would call the callback function(s) of the self node, which for this node is m2b_callback, as shown below. It subscribes to the /map2base topic, saves the current x and y position, as well as the current orientation as euler angles.

```
def m2b_callback(self, msg):
    self.px = msg.position.x
    self.py = msg.position.y
    orien = msg.orientation
    self.ox, self.oy, self.oz = euler_from_quaternion(orien.x, orien.y, orien.z, orien.w)
```

The figure below shows the output of the script get_waypoints.py and the supposed output.



7.4. Navigation Software Design

In order to run the robot using the waypoints collected in Section 7.3, the script r2auto_nav_m2b.py is run.

Firstly, the main() function is called when running the script, which initialises an AutoNav() node, and calls the rclpy::spin_once function on this node. As mentioned above in Section 7.3, the callback functions are called. In this case, these are the m2b_callback and scan_callback functions. These will be further elaborated on below.

```
def main(args=None):
    rclpy.init(args=args)


    auto_nav = AutoNav()
    rclpy.spin_once(auto_nav)
    auto_nav.connect_to_mqtt()

    auto_nav.destroy_node()
    rclpy.shutdown()
```

The callback functions as mentioned are depicted in the figures below.

The m2b_callback function is identical to the one as mentioned in Section 7.3, which saves the position as well as orientation of the TurtleBot at its current position.

```
# callback function when a map2base message is received
def m2b_callback(self, msg):
    position = msg.position
    self.x = position.x
    self.y = position.y
    orientation_quat =  msg.orientation
    self.roll, self.pitch, self.yaw = euler_from_quaternion(
        orientation_quat.x, orientation_quat.y, orientation_quat.z, orientation_quat.w)
```

The scan_callback function is used to collect data from the LiDAR sensor at 360 degrees, saved in the script as a NumPy array of distances at each angle.

```
# callback function when a scan message is received
def scan_callback(self, msg):
    # create numpy array
    self.laser_range = np.array(msg.ranges)
    # print to file
    # np.savetxt(scanfile, self.laser_range)
    # replace 0's with nan
    self.laser_range[self.laser_range==0] = np.nan
```

The connect_mqtt() function is then called by the main, and connects the RPi to the MQTT. In this function, the MQTT loop begins.

```
# method to connect the RPi to the MQTT broker
def connect_to_mqtt(self):
    print('Connecting...')
    self.mqttclient.connect(self.broker_address)
    global run
    while True:
        self.mqttclient.loop()
```

Once the MQTT is connected, it will subscribe to the ‘table’ MQTT topic, and wait to receive a message.

```
# callback function when RPi is connected to MQTT broker
def on_connect(self, client, userdata, flags, rc):
    print('Connected with result code'  + str(rc))

    # RPi should subscripe to the table topic,
    # and will call the AutoNav::on_message method when it receives a message
    client.subscribe(self.TABLE_TOPIC)
```

Once a message is received, the script will get the route needed to be taken to go to the table and stop the MQTT loop. In the route function, the mover() function is called.

```python
# callback function when RPi receives a message from the MQTT broker (table topic)
def on_message(self, client, userdata, msg):
    table = msg.payload.decode('utf-8')
    print('Message received\nTopic: ' + msg.topic + '\nMessage: ' + table)

    # call the route method to get the path to the table
    self.route(table)

    # stop the MQTT broker, in order to subscribe to spin AutoNav node
    self.mqttclient.loop_stop()
```

While the can is not loaded, the TurtleBot will wait for the can to be loaded before reversing out of the dispenser. After travelling a short distance, the TurtleBot will line follow backwards before coming to a stop, and traversing the waypoints of the path retrieved from `route(table)`

```python
def mover(self):
    try:
        while bool(GPIO.input(21)):
            print(f'Can not loaded')
            time.sleep(0.001)
        twist = Twist()
        twist.linear.x = -0.2
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
        time.sleep(2)
        self.stopbot()
        self.line_following(-1)
        self.traverse_waypoints()

    except Exception as e:
        print(e)

    # Ctrl-c detected
    finally:
        # stop moving
        self.stopbot()
```

After traversing all the waypoints, the TurtleBot will either dock to the table or dock to the dispenser. When docking to the table, it will scan for the shortest distance direction and dock at that shortest distance, before returning to the dispenser.

```
def dock_to_table(self):
    print('docking..')
    rclpy.spin_once(self)
    time.sleep(1)
    while not bool(GPIO.input(21)):
        time.sleep(0.001)
    if (self.table == 1 or 3 or 5):
        self.pick_direction(0)
    elif (self.table == 4):
        self.pick_direction(1)
    elif (self.table == 6):
        self.go_to_table_6()

    self.return_home()
```

Going back to the dispenser, the TurtleBot will traverse the reverse order of checkpoints and line follow back into the dispenser. It will then connect back to the MQTT server to await for the next table number.

```
def return_home(self):
    self.going_back = True
    self.path = self.path * 100 + 1
    path_string = str(self.path)
    path_string = path_string[::-1]
    self.path = int(path_string)
    self.traverse_waypoints()
    self.line_following(1)
    self.connect_to_mqtt()
```

7.5. Dispenser software algorithm

To communicate with the RPi, MQTT will be set up by connecting to the same Wi-Fi connection. The onboard OLED screen will be used to display the status of the Wi-Fi

connection and connection to the MQTT broker. Once both have been connected, ESP32 is ready to communicate with the RPi.

```
void connect_MQTT(){
  Serial.print("Connecting to ");
  Serial.println(ssid);
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    displayWifiInformation(WiFi.status());
    Serial.print(".");
  }
  Serial.println("WiFi connected");
  Serial.print("IP address: ");
  Serial.println(WiFi.localIP());
  displayWifiInformation(WiFi.status());
  while (!client.connect(mqtt_server)){
    displayMQTTInformation(0);
    Serial.println("Connection to MQTT Broker failed…");
  }
  displayMQTTInformation(1);
  Serial.println("Connected to MQTT Broker!");
}
```

An analog pin connected to one of the ADC channels on the ESP32 will be used to measure the voltage level from the button interface to determine the table number that has been pressed. This is done by comparing the measured voltage level with certain threshold values measured beforehand as seen in figure below.

```
int findTable(int value) {
  if (value > 3300 && value < 3500) return 4;
  else if (value > 3100 && value < 3300 ) return 5;
  else if (value > 2850 && value < 3100) return 6;
  else if (value > 2500 && value < 2800) return 3;
  else if (value > 1900 && value < 2500) return 2;
  else if (value < 500) return 1;
  else return 0;
}
```

Whenever the TurtleBot returns to the dispenser, a message sent through MQTT will inform the dispenser, which will then release the can by turning a servo motor 180 degrees and then back. At the same time, the table number selected will be sent to the RPi via MQTT so that the TurtleBot knows where to deliver the can drink to.

```
void loop() {
  client.loop();
  int value = analogRead(ANALOGPIN);
  if (findTable(value)!=0){
    tableNumber=findTable(value);
  }
  displayInformation(tableNumber);
  Serial.println(tableNumber);
  if (releaseCount && tableNumber) {
    String msgPublish = String(tableNumber);
    client.publish("table", msgPublish.c_str());
    releaseCan();
    releaseCount=0;
    tableNumber=0;
  }

}
```

For the ESP32 to constantly listen for messages from the RPi, a callback function has to be initiated to allow the ESP32 to subscribe to the specific topic.

```
void callback(char* topic, byte* message, unsigned int length) {
  Serial.print("Message arrived on topic: ");
  Serial.print(topic);
  Serial.print(". Message: ");
  String msg;
  for (int i = 0; i < length; i++) {
    Serial.print((char)message[i]);
    msg += (char)message[i];
  }
  Serial.println();
  if (String(topic) == "dock") {
    if(msg == "Home"){
      releaseCount = 1;
      display.clearDisplay();
      display.setTextSize(2);
      display.setTextColor(WHITE);
      display.setCursor(0, 0);
      display.println("Home");
      display.display();
    }
  }
```

7.6. GitHub Repository

r2auto_nav: https://github.com/MoeySeanJean/r2auto_nav
navigation_eg2310: https://github.com/applepiofmyeye/navigation_eg2310

8. Before Your Run (Pre-Ops Check)

8.1. Cable Check

For proper wiring management, the team has measured and crimped the respective cable with the adequate pinhead and length, and they are arranged in a manner such that it will neither be too tense nor too loose. Also, it shouldn't be blocking or impeding the actuation path (the dispensing mechanism. On top of that, the pinheads are to be double-checked to ensure electrically insulated by heat shrink for safety concerns and to ensure a tight connection with the different pins to allow more consistent power/signal transmission.

8.2. Components Check

Objectives	Test Procedures	Expected Results
Power of dispenser	Connect ESP32 to wall power	LED lights
Power of robot	Switch on the OpenCR	LED lights
Connection of dispenser to wifi and MQTT broker	Wait for the OLED to display "Wifi Connected" and "MQTT Connected". Press "EN" on ESP32 if waiting is prolonged.	"Wifi Connected" and "MQTT Connected" are displayed on OLED
Button interface	Press any button on the interface	The corresponding number will be displayed by the OLED
Stability of dispenser and robot	Examine and shake the body	No visible damage and no rattling sound
Robot functions	Run rosbu and rteleop	LIDAR spins and motor moves

9. Getting Started

9.1. Wiring

Make sure your wiring follows sections 5.3 and 5.4.

9.2. Powering Up

Ensure the battery is connected to the OpenCR and flick the OpenCR on/off switch. Connect the ESP32 to wall power.

9.3. Connecting to the Turtlebot

Ensure that your laptop and the robot are on the same network as your laptop. To do this, you should create your wifi hotspot and connect your Turtlebot as well as laptop to it. Then, you can connect to the Turtlebot using SSH. Now, you can follow Section 7 to perform your task.

9.4. Instructions on dispenser

Follow section 6.3 for usage of the system.

10. Purchase List and Item Request List

Components	Used in	Quantity	Source	Price per unit	Total Price
Tutlebot3 Burger	Robot	1	From Electronic Lab		
Microswitch	Robot	1	From Electronic Lab		
IR Sensor Module	Robot	2	From Electronic Lab		
Acrylic Board 18''x24''x3mm	Robot and Dispenser	3	From Art Friend@Clementi	11.56	34.68
Jumper Wires	Robot and Dispenser		From Electronic Lab		
Screws and Nuts	Robot and Dispenser		From Electronic Lab		
Wooden Shaft 1/4''x36''	Dispenser	1	From Art Friend@Clementi	1.91	1.91
PVC Pipe	Dispenser	1	From Samroc Paints&Hardware Supply@Clementi	6.4	6.4
Protoboard	Dispenser	1	From Electronic Lab		
ESP32 Board	Dispenser	1	From Royston		
Servo Motor	Dispenser	1	From Electronic Lab		
OLED Screen	Dispenser	1	From Royston		
Push Button	Dispenser	6	From Electronic Lab		
				Total Spendin	42.99

11. Reference

Dev, A. (2019, March 23). *Communication between Arduino and Raspberry Pi using NRF24L01*. Medium. Retrieved from https://medium.com/@anujdev11/communication-between-arduino-and-raspberry-pi-using-nrf24l01-818687f7f363

Instructables.com. (n.d.). *Connecting Your Raspberry Pi to the Web*. Retrieved from https://www.instructables.com/Connecting-Your-Raspberry-Pi-to-the-Web/

PiMyLifeUp. (n.d.). *Raspberry Pi Pressure Pad*. Retrieved from https://pimylifeup.com/raspberry-pi-pressure-pad/

Raspberry Pi HQ. (n.d.). *Use a Push Button with Raspberry Pi GPIO*. Retrieved from https://raspberrypihq.com/use-a-push-button-with-raspberry-pi-gpio/

Robo India. (n.d.). *Raspberry Pi LDR Tutorial*. Retrieved from https://roboindia.com/tutorials/raspberry-ldr

Robotis. (n.d.). *TurtleBot3 LDS-01*. Retrieved from https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/

Robu, R. (n.d.). *Raspberry Pi Ultrasonic Sensor Interface Tutorial*. Retrieved from https://robu.in/raspberry-pi-ultrasonic-sensor-interface-tutorial/

Singleboardblog.com. (n.d.). *Install Nginx on Raspberry Pi*. Retrieved from https://singleboardblog.com/install-nginx-on-raspberry-pi/

Wikimedia Foundation. (2023, April 13). *MQTT*. Wikipedia. Retrieved April 23, 2023, from https://en.wikipedia.org/wiki/MQTT

Yong, A. (2020, October 5). *Maze Escape with Wall Following Algorithm*. Medium. Retrieved from https://andrewyong7338.medium.com/maze-escape-with-wall-following-algorithm-170c35b88e00#:~:text=The%20wall%20follower%20algorithm%20contains,extra%20memory%20to%20solve%20maze

YouTube. (2021, Apr 27). *Control Your Raspberry Pi Remotely Using Your Phone | RaspController.* [Video]. Retrieved from https://www.youtube.com/watch?v=lnHyVswZksM

Yu, J., Su, Y., & Liao, Y. (2020). The path planning of mobile robot by neural networks and hierarchical reinforcement learning. Frontiers in Neurorobotics, 14, 63.

Borenstein, Johann & Koren, Yoram. (1991). The Vector Field Histogram - Fast Obstacle Avoidance For Mobile Robots. Robotics and Automation, IEEE Transactions on. 7. 278 - 288. 10.1109/70.88137.