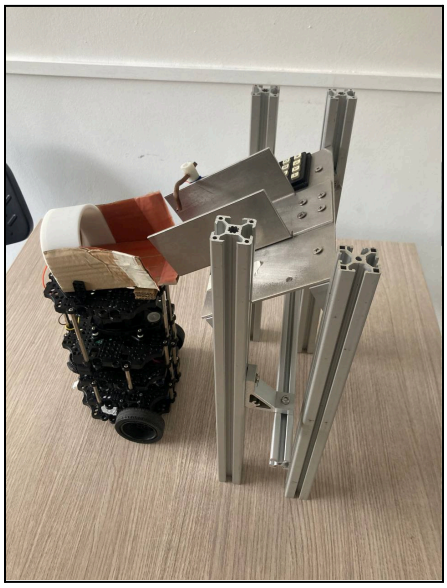
G2 REPORT
EG2310 Group 9



Aditi Joshi	A0265848L
Leow Kai Jie	A0254460J
Jing Ming Yuan	A0258733W
Rupan Kumar Manogar	A0259243B

Table of Contents

1. Introduction

1.1 Problem Definition

This document describes the system design process of the EG2310 project. The system was designed to achieve the following purpose:

> *To autonomously deliver a can that has been dispensed by a dispenser, to the desired table in a "restaurant" with 6 tables.*

1.2 Components Overview

The ecosystem of this mission is split into two major components, the delivery robot and the dispenser.

1.2.1 Delivery Robot

A Turtlebot3 Burger bot is to be used to deliver a can to the desired table number.

The three major considerations to ensure the smooth functioning of this component are divided into the following:

> Mechanical: The bot needs to be able to receive the can from the dispenser and hold the can in place while it navigates to the table.
>
> Electrical: In addition to the Raspberry Pi, OpenCR, and LIDAR, the bot needs to be able to detect the presence of the can in the bot.
>
> Software: The bot should autonomously navigate the restaurant to the desired table after receiving input of the desired table number.

1.2.2 Dispenser

The dispenser is responsible for taking in one can at a time from the TA and then dispensing it onto the Turtlebot3 for delivery. In addition to that, it also takes in the input for the desired table number for delivery.

The three major considerations to ensure the smooth functioning of this component are divided into the following:

Mechanical: The dispenser needs to deliver the can onto the turtlebot3 in a smooth and reliable manner.

Electrical: The dispenser has to take in the input of the table number, for example via a number pad. Additionally, components like a servo motor would be required for the dispensing of the can. A microcontroller would be required to tie in and control the various electronics used.

Software: The dispenser has to wirelessly communicate with the R-Pi on the Turtlebot3 to send the information of the chosen table number. It also has to control and operate the electronics on the dispenser, such as the servo motor and the number pad.

2. Literature Review

2.1 Modifications to Turtlebot3 to receive the payload

Option 1: Fifth Waffle Concept - In this concept, we are building upon the design features of the Turtlebot3 burger by adding another layer to it and using the extra layer to fit our needs. A custom-designed “can holder” will be secured on this fifth waffle for the soda can to sit in as well as to accommodate the loading of the can from the dispenser. This design will also be more stable because the consequences of the higher centre of gravity are not as high, as the can will be securely placed inside the cupholder.

Option 2: Piggyback Concept - This concept involves a cupholder attached to the back of the Turtlebot3. This ensures the can is well-secured and does not fall off the bot during motion. Moreover, the Lidar will be unobstructed. However, the con of this design is the uneven distribution of weight once the can is loaded, the bot may become unstable and trip over. It may also affect the navigational capabilities of the bot as the total area occupied by the bot has been increased and so may result in accidental collisions for example when the bot is turning near a table.

2.2 Can detection in the Turtlebot3

Option 1: Microswitch[1] - A simple microswitch can be used to detect the presence of the can in the dispenser. A small weight can trigger the switch when the lever is pressed down, which will then send a signal that can be interpreted and used to tell the bot whether the can is loaded or not.

Option 2: Weight sensor - Although the dispenser can also provide signals to the robot after the payload is pushed from the dispenser to indicate that loading is complete, there is a possibility that the payload did not load fully into the bot's can holder, owing to inaccuracy from the actuator. A weight sensor module, like a Load Cell with a HX711 amplifier [2], will be installed to double-check the loading procedure to make sure that the payload is correctly delivered to the holder on the bot. However, it is very complicated to set up because of the additional wiring and unnecessary weight added to the robot.

2.3 Autonomous Navigation

The Turtlebot3 has a 360-degree Laser Distance Sensor LDS-01, a 2D laser scanner that is capable of sensing 360 degrees with a detection distance of approximately 120 mm to 3,500 mm [3]. The data around the robot collected by the LDS-01 is used for SLAM (Simultaneous Localization and Mapping)[4] and Navigation.

Shortest distance algorithms such as Dijkstras' Algorithm, A* Algorithm[5], and D*[6] algorithm were some possible options. Dijkstra algorithm works on the basis that the map is unknown, resulting in an uninformed search where no information about the environment is known, and a brute force method is employed, resulting in greater time complexity than A* or D*. Also known as heuristic searches, A* and D* algorithm takes in other parameters such as cost to goal and cost taken from source in a partially or fully known map to produce much lesser time complexity than Dijkstra's algorithm. A* works when the entire environment is known while D* is "capable of planning paths in unknown, partially known, and changing environments in an efficient, optimal, and complete manner"
At face value, D* looks like it is better than A* which is better than Dijkstra's algorithm, however, the implementation of Dijkstra is easier as compared to A* which is easier than D*. This is because in A* and D*, we need to compare and choose the best heuristic from heuristics such as Euclidean distance or Manhattan distance to correctly implement the algorithm.

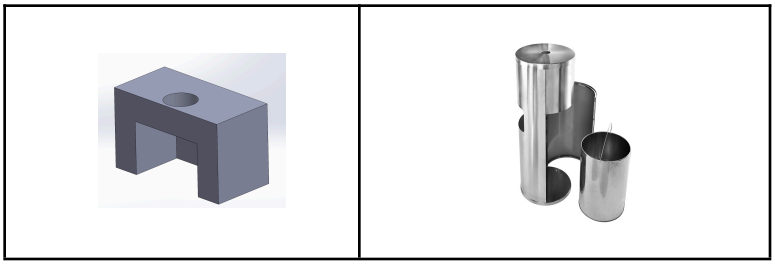
However, the above-mentioned algorithms might be excessive for our project. A method that might be best suited for us is the waypoints method. Utilising the coordinates on points in the map, we could navigate our robots through waypoints to reach the intended table. This is a simple method and it also makes use of coordinates data that we are able to obtain.

2.4 Mechanical functions to move the can from the dispenser to the Turtlebot3

Option 1: Trapdoor-based Dispenser - In this design, the dispenser will utilise gravity to facilitate the transfer of the can from the dispenser to the turtlebot.

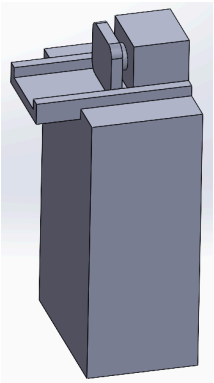
The first option is one where the dispenser will act as a dock or garage for the turtle bot where it can go under the dispenser. When the user has selected the table number, and the Turtlebot3 is in a position to receive the can, a trapdoor will open and allow the can to fall by gravity onto the bot. Although this method would be relatively simple to build and implement, there is a risk of it failing due to structural or material failure every time the can is dropped from a height, leading to uncertainty in the feasibility and longevity of this concept.

Hence a better option would be to slide the can down an inclined slope rather than simply dropping it from height. This would reduce the energy transferred during impact between the can and the can holder, thereby reducing the risk of failure or any breaking.



Option 2: Coil-based Dispenser - Similar to the coil-based systems used in most commercial dispensers, we can use a coil to achieve horizontal translation motion of the can. As the coil rotates in place, any object, such as a can, placed in the space between the coil will be pushed. Rails or walls along the sides will ensure that the can only moves in a straight line. A simple motor can be used to rotate the coil with a proper attachment between the motor and the coil. Since this mechanism is well established and already being used in most dispensers, this is a viable option. However, the coil needs to be perfectly uniform and shaped properly so that the can can fit. Another problem that may occur is that the coil spirals like a ferris wheel instead of rotating in place if the attachment between the motor and the coil is not perfectly in a straight line.

Option 3: Linear Actuator - A linear actuator that extends and pushes any object in front of it can be used. By using a similar mechanism to the linear actuator, a self-made linear actuator will be powered by a motor to rotate a lead screw and convert the angular motion to translational motion. To ensure the same distance of the linear actuator pushed every time, the number of rotations of the gear will be counted



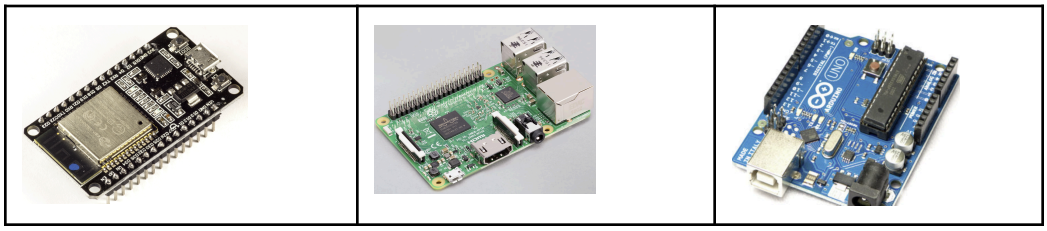
by the single board computer in the dispenser. Other supplementary designs to ensure the proper working for the linear actuator will be the stopper at the end of the track and the side boundaries of the track to ensure that the gear does not slide off from the track. We would likely be using the self-made linear actuator as off-the-shelf linear actuators are very expensive and over-specced for our needs.

Option 4: Conveyor Belt - Another way to move an object is through the use of a conveyor belt system. A motor drives a pulley with a belt to convert rotary motion to linear motion and move a load on its belt. Moreover, load weight sensors can be integrated to determine when the motor should start and stop. The drinks can be placed anywhere along the belt and once the bot is in place, the motor of the conveyor will activate until the can is off the belt. However, this technology can be both complex and expensive. Moreover, the conveyor belt cannot extend past the dispenser, so the can cannot be smoothly transferred to the Turtlebot3.

2.5 Digital Interface between the Turtlebot3 and Dispenser

2.5.1 Microcontroller

The dispenser needs a “brain” to receive incoming data. For example, to check whether the can is loaded in place during dispensing while also sending input table information to the Turtlebot3 for delivery. Some functionalities that we were looking for are programmable GPIO pins, wireless capabilities, and low cost. Several boards that could perform these tasks caught our eye during our research, such as ESP32, Raspberry Pi, and Arduino.



2.5.2 Number Pad

When an order is placed, the Turtlebot3 needs to deliver it to the correct table. A number pad will be used to receive input and send the desired table number to Turtlebot3. It must be part of the digital interactions between the Turtlebot3 and the dispenser.

2.6 Communication between Microcontrollers

Information must be sent between the dispenser's microcontroller and the bot's controller for functions to be performed seamlessly. We are opting for one of the 3 wireless options: HTTP, Bluetooth, or MQTT.

	Bluetooth	MQTT	HTTP
External requirements	Bluetooth module	Message Broker	Web server
Reliability	Designed to be resilient by using adaptive frequency hopping and sending data in fast, small packets [1]	Offers 3 Qualitiy of Service (QoS) levels to choose from to ensure the reliability of message being sent successfully	Proper functioning of the software and hardware involved in the transmission of data. Availability of web servers
Range	More than a kilometer	Internet Connection	Internet Connection

3. Concepts of Design and Operation

3.1 Overview of Phases

We have divided the mission into three phases, phase 1 is the dispensing of the soda can into the Turtlebot3 along with taking the input of the designated table number using the number pad on the dispenser. The second phase includes the detection of load in the turtle bot and the navigation toward the final destination. The third phase includes receiving the order from the customer and the return journey and the docking mechanism of the Turtlebot3 to the dispenser, where it will be ready for the next order.

3.2 Phase 1 - Dispensing Phase

1. **The objective of this phase:** To successfully dispense the can from the dispenser to the turtlebot for delivery and take the input of the desired table number from the user.

[1]https://www.bluetooth.com/blog/bluetooth-range-and-reliability-myth-vs-fact/

2. **Description of the phase:** First, the dispenser will be loaded with a can manually by the TA. Then, the table number is taken as input using the number pad placed on the dispenser. This triggers the servo motor and causes the trapdoor to open. The can is now free to slide down the slope of the dispenser and enter the can holder on the turtlebot. At the same time, the table number is sent from the dispenser's ESP32 to the bot's Raspberry Pi via HTTP communication.

3.3 Phase 2 - Search/Navigation Phase

1. **The objective of this phase:** To navigate to the desired table for delivery

2. **Description of the phase:**
   The map topic will tell us the position of our robot, in coordinates, relative to the map. We will first need to set the waypoints on the map. We will then calculate the Euclidean distance from our robot to a pre-set waypoint. We will then turn our robot to face the waypoint and walk the calculated distance toward the waypoint. This process of finding a new waypoint and walking towards it repeats until the robot reaches the table.

3.4 Phase 3 - Return Journey and Docking

1. **The objective of the phase:** This phase includes the return journey of the Turtlebot3 back to the dispenser for the next order.

2. **Description of the phase:** After the can has been taken out by the user, the micro switch on the can holder is released and the absence of the can causes the turtlebot to walk back to the dispenser for the next delivery. The bot walks back using the same waypoints that were used to navigate to the table initially, to reach a waypoint that is set in front of the dispenser. On reaching that waypoint, the bot will align itself to be ready for docking by measuring its distance from the wall on the right of the bot when it is facing the dispenser. The bot is now ready for its next delivery.

4. Preliminary Design

For each of the two major components, the following are the final concepts chosen and the rationales behind them:

4.1 Turtlebot

4.1.1 Mechanical

For the modifications on the Turtlebot3, the fifth waffle concept has been used. An additional waffle plate has been installed with the help of six 90mm tall plate supports which are made using twelve M3X45 plate supports, where 2 supports are fastened together using a set screw. The LIDAR will not detect the supports as they are too close to the LIDAR.

The can holder at the top of the fifth waffle has been designed to hold the can on its sides. It will be attached to the waffle plate using strong double-sided tape and will be held in place by 3 brackets mounted on the waffle plate using 2 rivets each.

OpenCR and Raspberry Pi are placed on the second and third waffle plates respectively.

Following are the mechanical components used for the modifications made on the Turtlebot3 for the mission:

S.No.	**Component**	**QTY**
1	Waffle plate	2
2	M3X45 plate support	12
3	M3X10 set screw	6
4	3D Printed Can Holder	1
5	Turtlebot3 Bracket	3
6	Black rivet	6
7	M3X8 cross screw	16
8	M3 nut	4

4.1.2 Electrical and Software components in the Turtlebot3

Raspberry-Pi 3B+: The Raspberry-Pi will subscribe to the data published by the LIDAR, handle wireless communication with the ESP32 that is mounted on the Dispenser and it is also used for autonomous navigation of the Turtlebot3 for delivery.

OpenCR: The OpenCR board on a TurtleBot3 plays a crucial role in the robot's control and communication systems, enabling it to perform tasks such as navigation, obstacle avoidance, and mapping. It is the main microcontroller present on board the Turtlebot3.

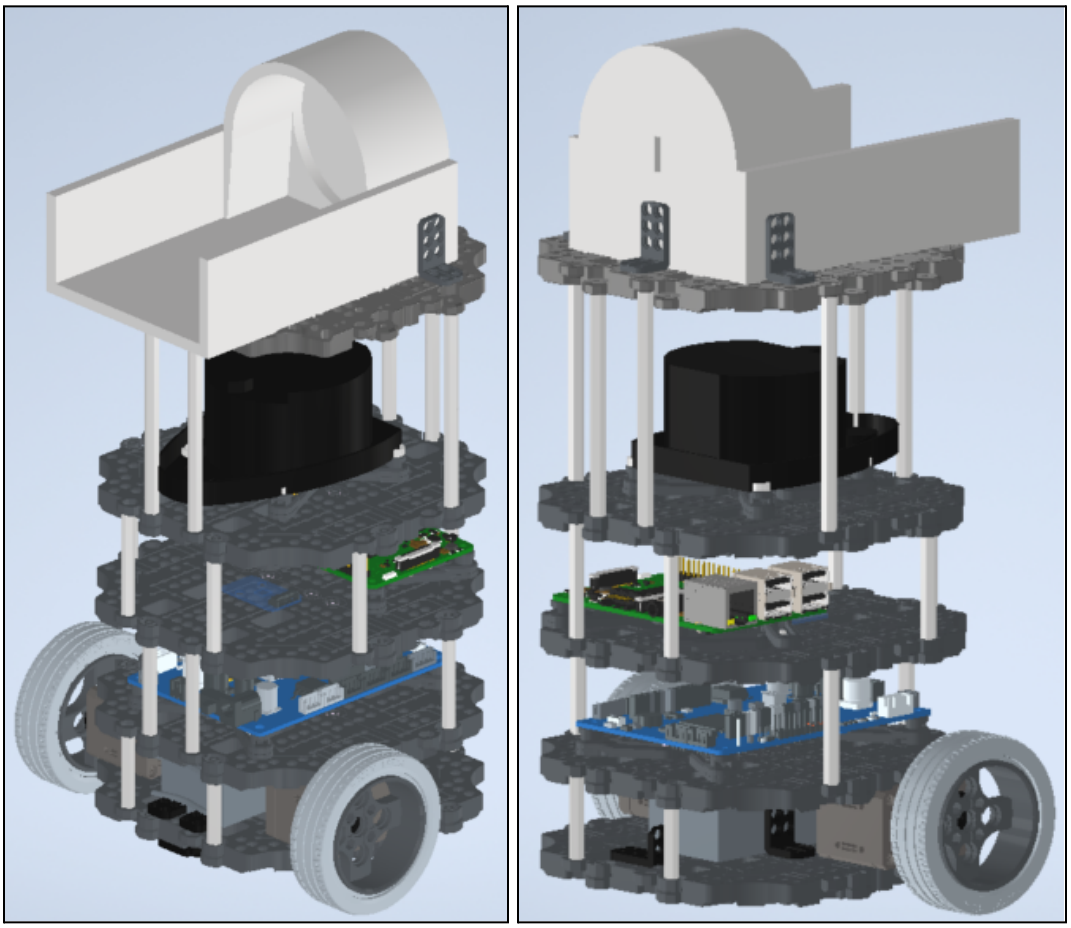
LIDAR (LDS-01): The role of LIDAR on the TurtleBot3 is to provide accurate distance measurements to help in mapping the surrounding environment and assist in obstacle detection and avoidance, enabling the robot to navigate through complex environments.

Li-Po (Lithium Polymer Battery): The Li-Po battery that has a voltage of 11.1V is the main power source to all the electrical components i.e. Raspberry Pi, OpenCR, LIDAR, and the dynamixel motors on the Turtlebot3.

Dynamixel: The Turtlebot3 consists of two dynamixel motors, one for each wheel, as the main actuators for its motion. The motors are used to control the speed and direction of the bot.

Microswitch: It detects the presence of the can by publishing the state of the switch when it is triggered. This information is useful because it informs the bot when dispensing is successful and when the can is successfully lifted from the can holder, aiding in the autonomous workflow of the bot.

4.1.3 CAD Mockup:



4.2 Dispenser:

4.2.1 Mechanical:

The base of the dispenser is a flat aluminium sheet that is angled at an incline. An aluminium U-Channel will be fastened onto this sheet to act as the channel for the can to be placed and held in. These will be supported by 4 aluminium profile bars with 4 brackets, where the profile bars are perpendicular to the floor while the brackets themselves are angled in order to accommodate the inclination of the sheet and U-channel parts.

The dispenser has been made using the following mechanical components:

S.No	Component	QTY
1	30 x 30 Aluminium Profile bars of height 265 mm	2

2	30 x 30 aluminium Profile bars of height 320 mm	2
3	30 x 30 brackets	4
4	220mm x 132mm aluminium sheet	1
5	172mm x 70mm x 57mm (L x W x H) U-Channel	1
6	M5x15 Cross screw	8
7	M6 Hex screw	10
8	M5 Nuts	4

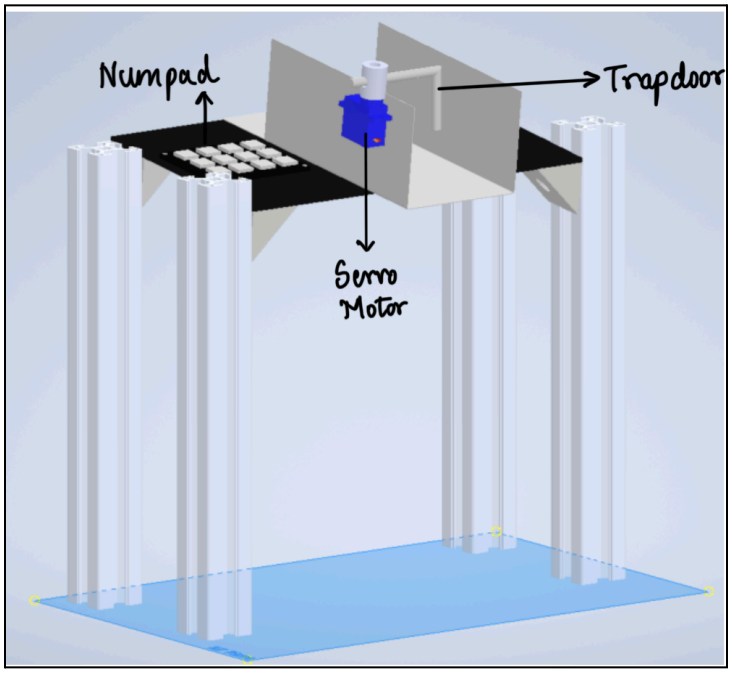
4.2.2 Electrical and Software Components:

ESP32: It is responsible for the communication between the dispenser and the Turtlebot3. HTTP is the protocol used for communication between ESP32 and Raspberry Pi, and it allows the ESP to send requests and RPI to receive responses. When TA presses a button on the number pad, the ESP32 sends a packet containing the corresponding table number to the RPI. The RPI receives and processes this packet and uses it to direct the turtlebot to the correct table.

Servo Motor: The trapdoor is used for controlling the dispensing of the can after it has been placed in the dispenser manually by the TA. The servo motor will connect to Vin and GND for power and GPIO12 for signal connection. When the button on the number pad is pressed, the servo motor will rotate to allow the can to slide into the turtlebot container.

Number Pad: Before the servo is initiated, the table number is taken as the input using the number pad placed on the dispenser. 7 pins on the number pad will be connected to the 7 GPIO pins on the ESP32. (from left to right, GPIO 4, GPIO 19, GPIO 16, GPIO 17, GPIO 15, GPIO 5, GPIO 18)

4.1.3 CAD Mockup:



5. Prototyping and Testing

5.1 Dispensing Mechanism

Our final mechanism involves the following components on the dispenser:

1. An inclined U-channel mounted on the dispenser which is responsible for holding the can.
2. A SG90 servo motor with a custom copper tube, which acts as the trapdoor, attached to it at the end. It is responsible for holding the can in place. This servo motor triggers only when a user inputs the table number. It then rotates the trapdoor away from the can, allowing the can to be dispensed.

During testing, we encountered some difficulties and inconsistencies. The following details these events as well as the solutions:

1) The can did not always slide smoothly down the U-channel
    a) We increased the angle of the incline by raising the back and lowering the front of the top platform with the U-channel.

    b) In order to reduce the friction between the can and the u-channel, mineral oil was used to lubricate the inner surfaces of the u-channel.
2) The four aluminium profile bars that were acting as the support legs of the dispenser did not make a perfect rectangle due to slight play in the screws as the holes drilled in the aluminium sheets were not threaded. Although this issue was very minor and did not cause any problems with the overall function of the dispenser, we added an extra profile bar joining the front two “legs” to ensure additional stability.
3) For the docking process, the Lidar on the turtlebot had difficulty detecting the dispenser, so a piece of cardboard was attached to the front of the dispenser.

5.2 Receiving Mechanism on Turtlebot

The Can Holder placed on the fifth waffle of the turtlebot is responsible for receiving and holding the can. We also encountered a few problems with the Can Holder during testing that led to making improvements such as:

1) The can sometimes did not fully enter the Can Holder
    a) Due to the angle of the can on the dispenser not matching the angle of the Can Holder on the bot, we made the Can Holder inclined as well by increasing the height of the front of the Can Holder by adding additional support below it.
    b) Due to friction between the can and the cardboard as the surface of the cardboard material is relatively coarse and hard, we used tape that had a very smooth plastic backing material and pasted it over the inner surfaces of the cardboard Can Holder.
2) On occasions, Bot Can One does not dock exactly in its right position and may have to receive the can from different angles. In order to accommodate for these inaccuracies, the entrance of the Can Holder is made wider by bending the front of the cardboard outwards, making a wider entrance.

5.3 Electrical Wiring

During the testing of the Servo motor and Number Pad, we attached each electrical component to the ESP32 individually using a breadboard and jumper wire headers. The use of a breadboard allowed for easy swapping of components, which was crucial in testing different components for functionality, especially testing the correct number output from the number pad. After verifying that the servo motor and number pad were functioning correctly individually, we connected all the components to the ESP32 to test the HTTP connection to the HTTP server on the laptop. Finally, the pi acted as the HTTP server to verify that the signal output from the ESP could be received by the RPi.

During the testing of the microswitch, we found that the connection between the jumper wire and the microswitch was extremely loose, which resulted in inconsistent and unreliable test results. We attempted to resolve the issue by soldering the wires to the normally closed (NC) and common (COM) pins on the switch, hoping to create a more secure and stable connection. However, upon testing the switch, we found that it was not outputting the correct signals as expected, and we were still encountering problems.

After further inspection and investigation, we discovered that the NC and COM pins were not suitable for our testing needs and that we needed to connect the wires to the normally open (NO) and common (COM) pins instead. (COM pin connect to GND on RPi and NO pin connect to GPIO18 on RPi) Once we made this change and retested the switch, we found that it worked perfectly and reliably, and we were able to proceed with our project testing without any further issues.

5.4 Navigation Algorithm

Our main concept will be to set waypoints to each table from the dispenser, i.e. the bot will walk from the dispenser to waypoint 1, waypoint 2, etc until it reaches the table. After it reaches the table, Bot Can One will wait for the can to be lifted before proceeding back to the dispenser again.

Several issues surfaced when we were testing out our algorithm for getting the current coordinates of Bot Can One and obtaining the new coordinates to move to. We had to make many adjustments:

1) Bot Can One's motors do not move with the same angular velocity due to hardware limitations. If we let the bot move from one point to another point just by facing that point and moving straight, the bot veers off course most of the time because of imperfect motors. What we did to overcome this is multiple recalibrations as the bot moves from one point to another. Bot Can One will periodically check if it is still facing the right direction, and if it is, it continues moving. If not, it adjusts to face the right direction and then moves.
   Another reason it might veer off course is that we turn the bot too quickly, so we overshoot the angle we are supposed to turn to. To mitigate this, we will slow the bot down when it is nearing the correct angle.
2) Due to hardware limitations and network latency, the coordinates that we obtain will fluctuate and are not precise. This will cause us to overshoot the target points sometimes. For example, if we are moving from (0,0) to the coordinate (1,0), we will check if we have reached very near (1,0) by checking the Euclidean distance between the bot's current position and the point (1,0) (*****we refer to the Euclidean distance between the bot's current position and a point as **"norm"**). However,

sometimes the coordinates of the bot do not update in time or are not precise, causing the bot to think that it has not reached the target point and will thus continue moving forward, overshooting the point. What we did to overcome this challenge was to check if the norm increased. By right, the norm should decrease steadily as the bot walks from one point to another but if the norm increases, it is evidence that the robot has overshot the target point and thus it will recalculate again and if necessary, it will turn around and walk towards the point again.

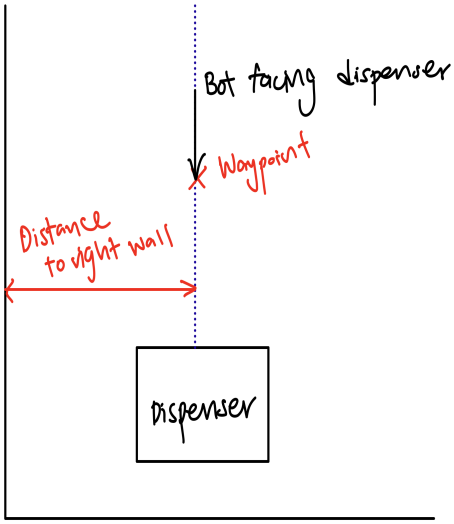
To further reduce the risk of overshooting, we will slow down the bot when we are reaching the destination.

3) Inconsistent coordinates even if we start up the bot in the same place are common and because we rely on pre-set waypoints, sometimes the waypoints will be “under” the bucket and the bot will crash into the bucket. We have set up an object detection system that will stop the bot once the lidar detects that there is an obstacle in front of it. It will then double-check if it is near the table. If yes, we will then exit the navigation phase and wait for the can to be lifted. If not, it will attempt to move to the table again.

   After the can is lifted, it will turn around and walk back to the dispenser, but this will mean that the bot will hit the obstacle as it turns. To overcome that, we will make the bot move back by a few centimetres first before turning.
4) To obtain the table number to move to, the script will collect the table number from a json file. This json file will update whenever a number is pressed on the keypad of the dispenser and sent to the raspberry pi via HTTP. The bot will wait for the file to be updated, and once there is a new entry, it will start the autonomous navigation towards that table.
5) For special table number 6, we will walk to the entrance of the “room” of the mystery bucket. When it is at the entrance, the bot will move forward slowly and try to detect an object on its left that is not the wall. Once it detects the bucket, it will turn and move slowly towards it. After which, it will backtrack to the entrance of the room and then back to the dispenser.

5.5 Docking Mechanism

Given our dispensing mechanism, the docking system had to be made precise in order to ensure smooth dispensing of the can. Incorrectly aligned docking will lead to unsuccessful dispensing causing the mission to fail at the very first step. At first, we tried to reach the waypoint in front of the dispenser, before turning to face the dispenser and then walking towards it. However, because of the inaccurate coordinates, we sometimes do not reach



the waypoint exactly. To combat this, we employ the technique of moving to the **waypoint** in front of the dispenser and then adjusting the **distance** of the bot from the wall to the right of the dispensing zone. The distance between the wall to the right of the dispenser and the desired point of parking was pre-calibrated before the mission.
When the bot reaches the waypoint, it will first turn to face the wall and then move forward/backward until the right distance before turning to face the dispenser and then walking straight into it. This gives us much more precise docking.

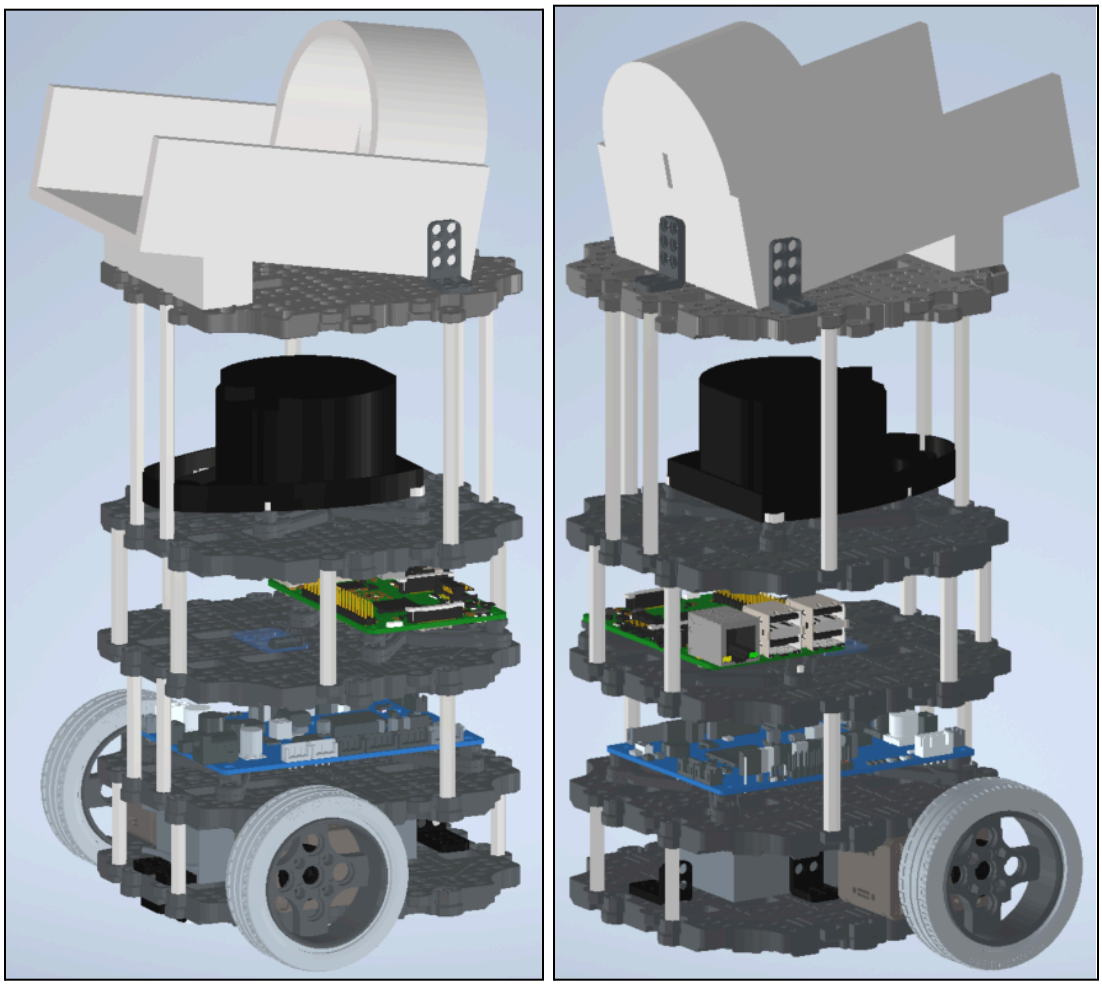
6. Final Design

6.1: Turtlebot3

6.1.1 Key Specifications

List	**Specifications**	**Note**
Size (mm)	180 x 178 x 298	Full assembly (Turtlebot3 with the can holder on the fifth waffle)
Weight	1.15 kg	
Drive Actuator	2 x L430-W250 Dynamixel	
Maximum Translational Speed	0.22 m/s	
Sensors on Board	LDS-01 Lidar Sensor	
Maximum Rotational Speed	2.84 rad/s	
Buttons and Switches	Microswitch	In addition to the pre-existing switches on the Turtblebot3
Battery and its capacity	LiPo Battery 11.1V 1,800 mAh	
Expected Operation Time	2.5h	
SBC/MCU	RPi3B+	

6.1.2 Final CAD



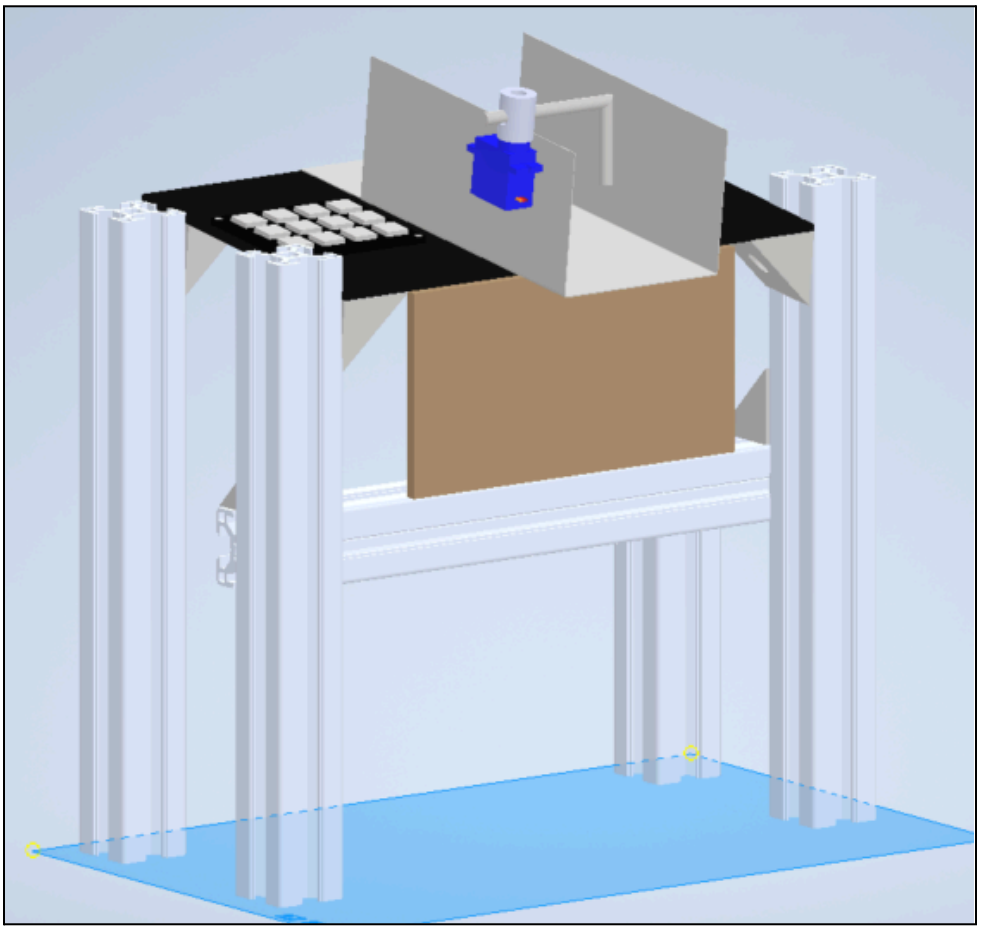
6.2 Dispenser

6.2.1 Key Specifications

List	Specifications	Note
Size (mm)	132 x 280 x 378	
Weight	2 kg	
Power Supply	12.5W Micro-USB wall plug	

SBC/MCU	ESP32	
U-Channel capacity	1 can	

6.2.2 Final CAD



6.3 Bill of Materials

We were provided with a Turtlebot3 Burger set, a mechanical toolbox as well as some miscellaneous electronic components to create the system. However, in order to source supplementary components, a budget of 100 SGD was provided. Following is the breakdown of the System's Finances.

No.	Part	QTY	Unit Cost	Total Cost

Turtlebot3 (Burger)				
1	Waffle Plate	8	Provided	Provided
2	Wheel	2		
3	Tire	2		
4	M2.5 Nuts (0.45P)	20		
5	M3 Nuts	16		
6	Spacer	4		
7	Rivet (Short)	14		
8	Rivet (Long)	2		
9	Plate Support M3x35	4		
10	Plate Support M3x45	10		
11	Adaptor Plate	1		
12	Adaptor Bracket	5		
13	PCB Support	12		
14	PH_M2x4mm K	8		

15	PH_M2x6mm K	4		
16	PH_M2.5x8mm K	16		
17	PH_T 2.6x12mm K	16		
18	PH_M 2.5x16mm K	4		
19	PH_M 3x8mm K	44		
20	Rear Ball Caster (w/ Ball)	1		
21	Li-Po battery	1	Provided	Provided
22	Li-Po battery charger	1		
23	USB Cable	2		
24	Dynamixel to OpenCR Cable	2		
25	Raspberry Pi 3 Power Cable	1		
26	Li-Po Battery Extension Cable	1		
27	SMPS	1		
28	AC-Cord	1		
29	Prototyping Board	1		

30	Dynamixel XL 430	2		
31	OpenCR 1.0	1	Provided	Provided
32	Raspberry Pi 3	1		
33	360 Laser Distance Sensor LDS-01	1		
34	USB2LDS	1		
35	Push Button	1		
36	M2x6 Bolt	2		
37	M3x8 Bolt	22		
38	M3x16 Bolt	4		
39	M3 Nut	24		
40	M2x10 Spacer	1		
41	M3x10 Spacer	4		
42	M3x40 Spacer	2		
<u>Additional Parts Required for Turtlebot</u>				
43	Micro Switch	1	Sourced from the Lab	

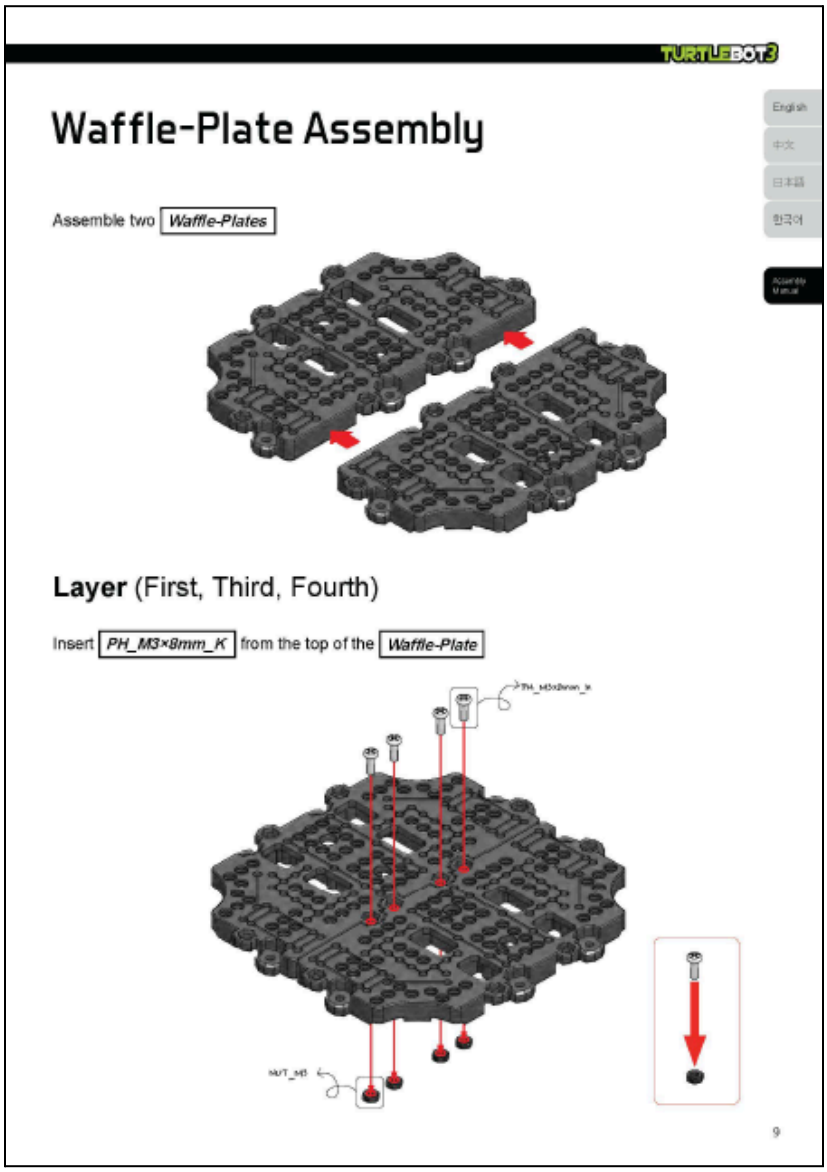
44	Can Holder	1	12	12
45	CardBoard U-Channel	1	Self-Sourced	
46	Fifth Waffle Plate	2	Sourced from the Lab	
47	M3 X 45 Plate Support	12	Sourced from the Lab	
48	M3X10 set screw	6	Sourced from the Lab	
49	Turtlebot3 Bracket	3	Sourced from the Lab	
50	Black Rivet	6	Sourced from the Lab	
51	M3 X 8 cross screw	16	Sourced from the Lab	
52	M3 Nut	4	Sourced from the Lab	
Dispenser				
1	30 x 30 Aluminium Profile bars of height 265 mm	3	Sourced from the iDP Lab and cut at the Central Workshop	
2	30 x 30 aluminium Profile bars of height 320 mm	2	Sourced from the iDP Lab and cut at the Central Workshop	
3	30 x 30 brackets	6	Sourced from the Lab	
4	220mm x 132mm aluminium sheet	1	Fabricated at the Central Workshop	
5	172mm x 70mm x 57mm (L x W x H) U-Channel	1	Fabricated at the Central Workshop	

6	M5x15 Cross screw	8	Sourced from the Lab	
7	M5 Nuts	4	Sourced from the Lab	
8	M6 Hex screw	10	Sourced from the Lab	
9	ESP32	1	Sourced from the Lab	
10	SG90 Servo Motor	1	Sourced from the Lab	
11	Number Pad	1	Sourced from the Lab	
12	Custom Copper tube-cum-Trapdoor	1	Sourced from the Lab	
13	3-D Printed Custom Servo Spline	4	3	12
Grand Total				24

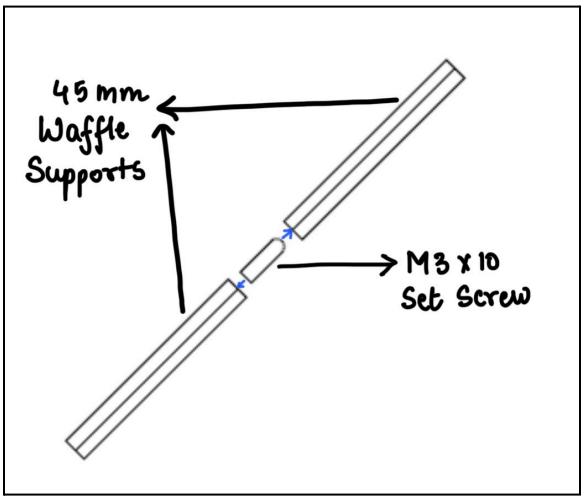
7. Assembly Instructions

7.1 Mechanical Assembly of Turtlebot3 with modifications:

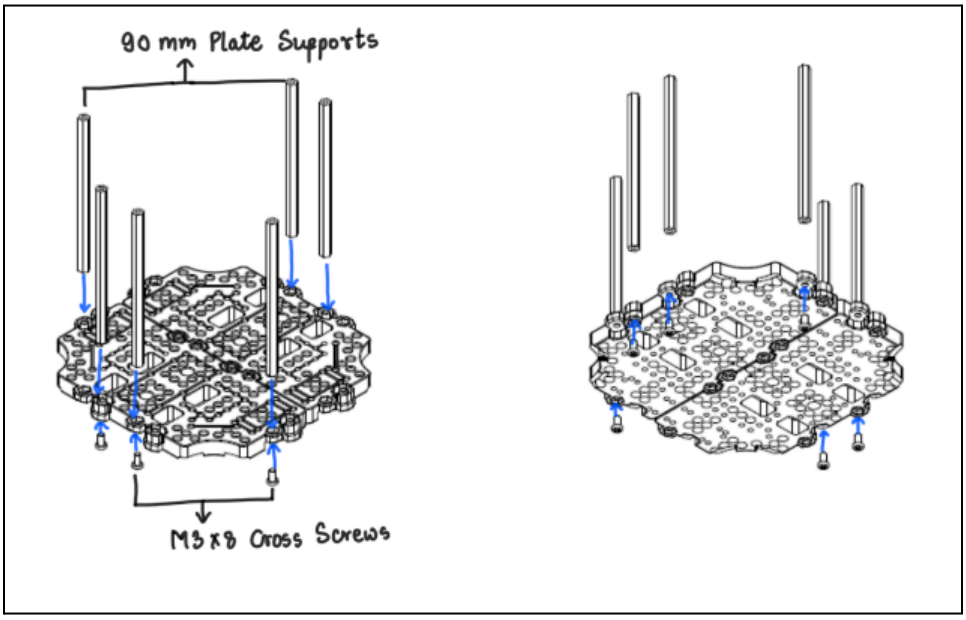
1. For assembly of base Turtlebot3 Burger, refer to the Turtlebot3 Burger Assembly Manual that can be found here.
2. Assemble the **fifth** waffle plate by following **page 9** of the Manual.



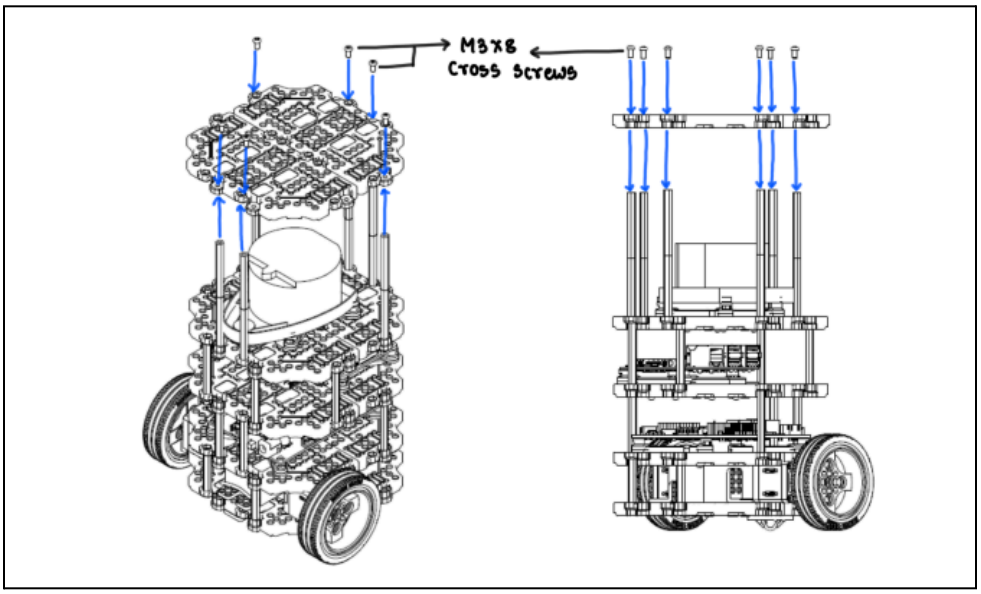
3. Join two 45mm waffle supports using an M3x10 set screw to make a 90mm waffle support. Repeat to make a total of **six** 90mm supports.



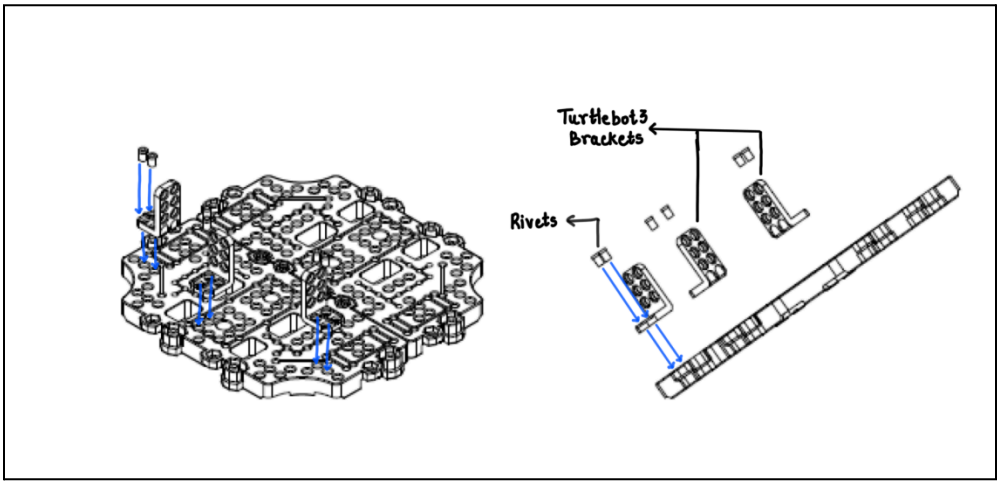
4. Secure the six 90mm supports to the **fourth** waffle plate on the Turtlebot using 6 M3x8 cross screws.



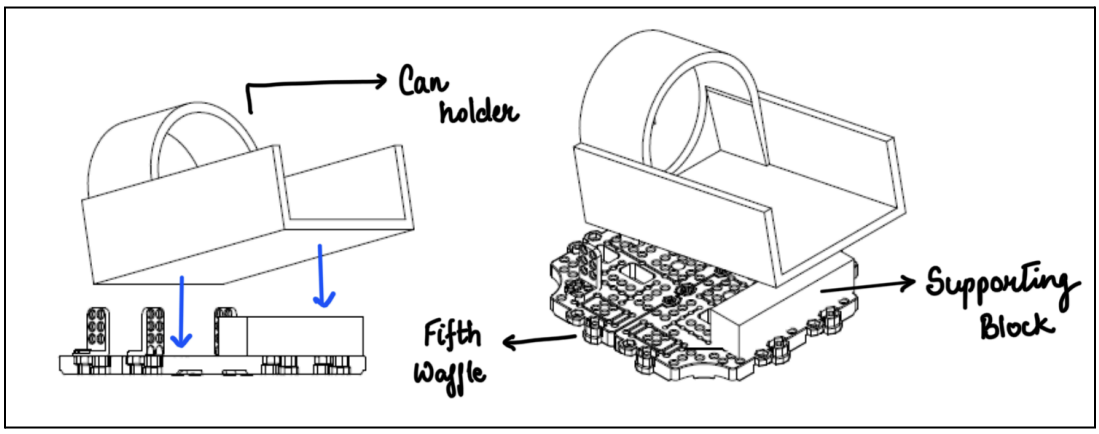
5. Secure the **fifth** waffle plate onto the Turtlebot using another 6 M3x8 cross screws.



6. Secure three Turtlebot Brackets onto the **fifth** waffle plate using two rivets per bracket as shown.

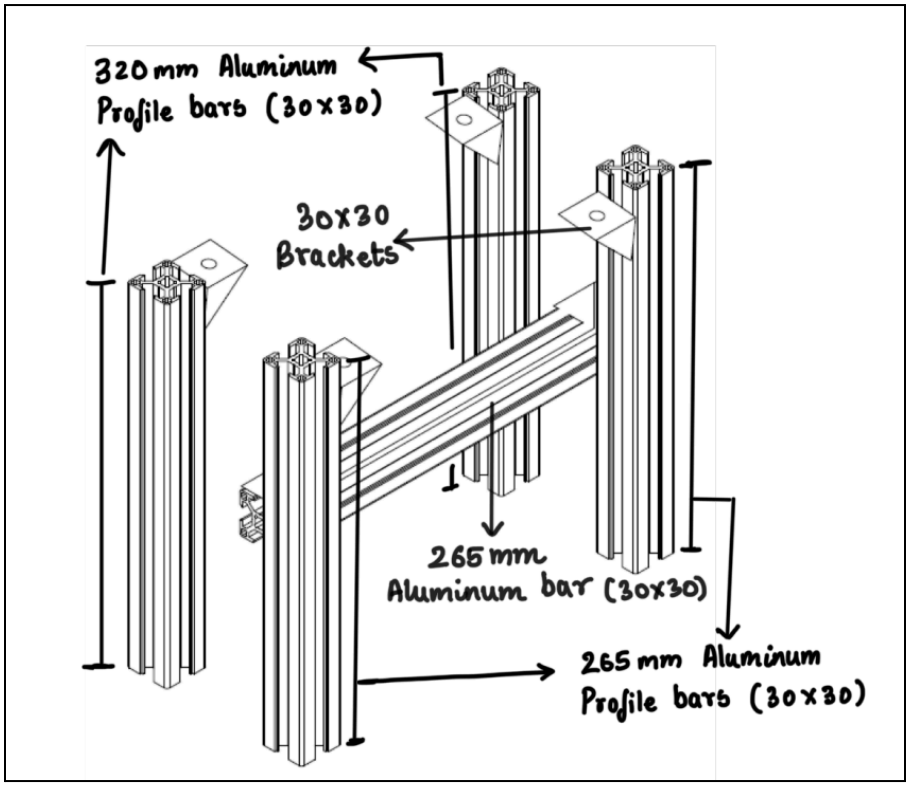


7. Place the supporting block onto the fifth waffle with the use of strong double-sided tape, and finally, place the Can Holder in position as shown.

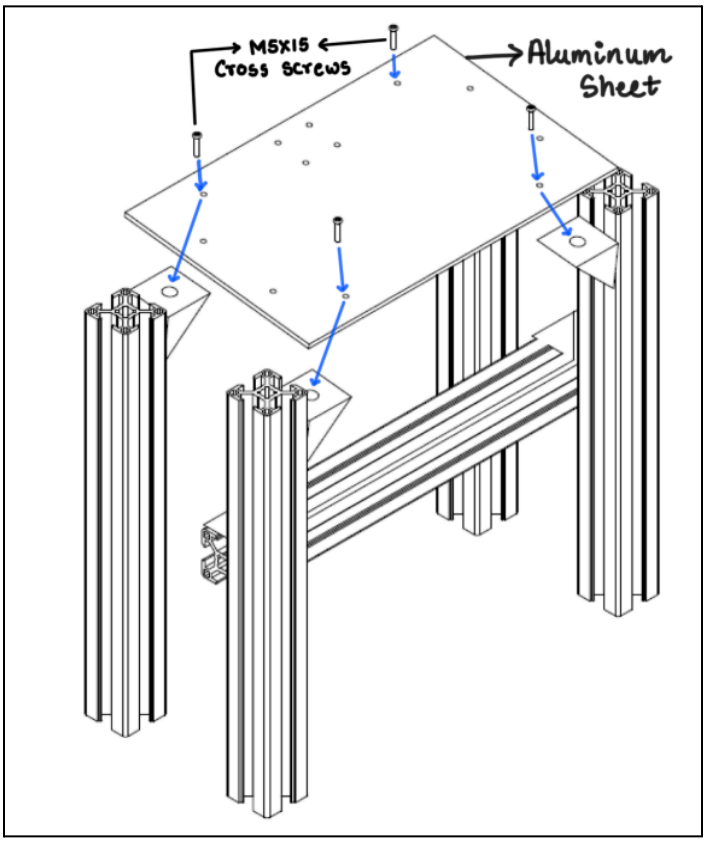


7.2 Mechanical Assembly of the Dispenser:

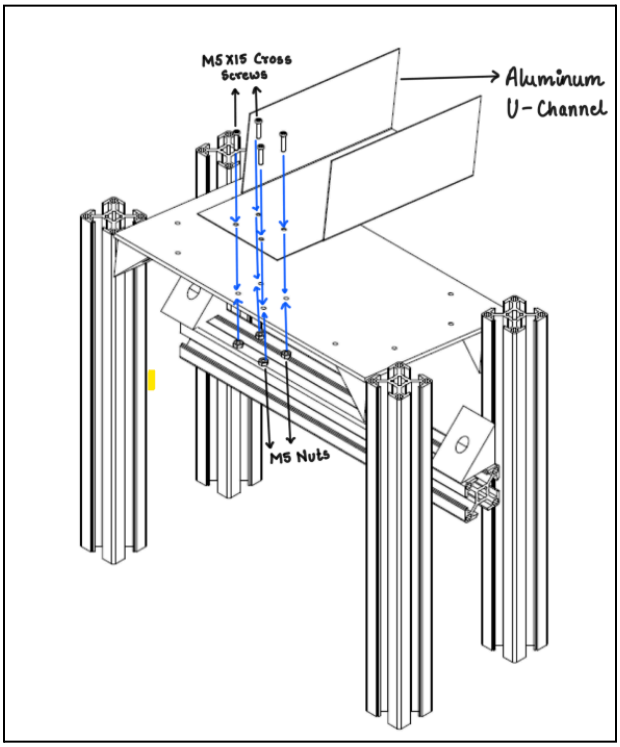
1. Construct the Aluminium Profile bars and brackets as follows.



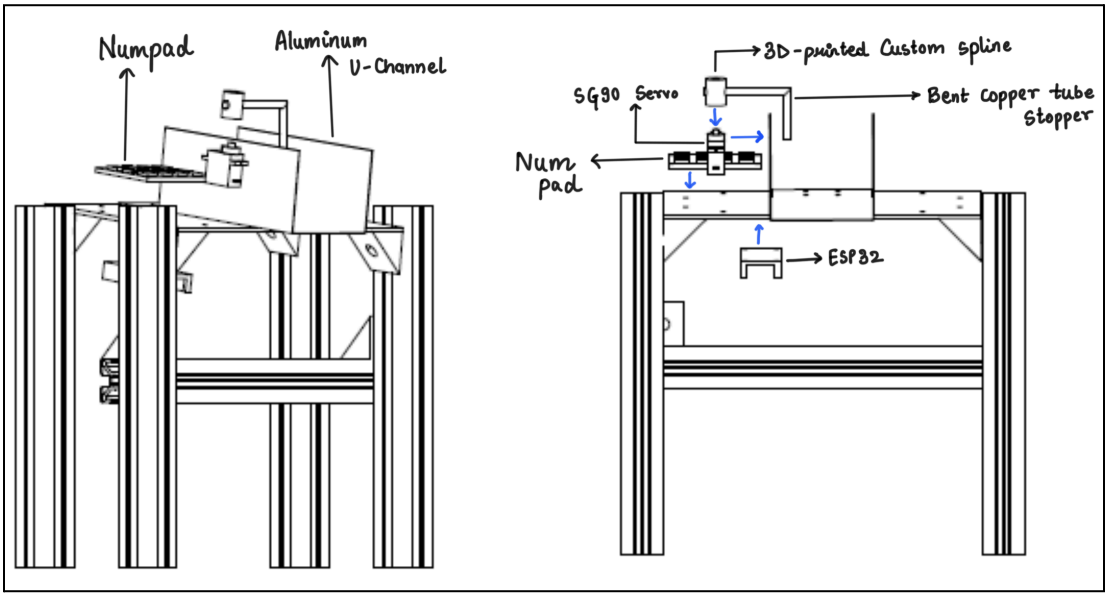
2. Secure the top aluminium sheet onto the frame using 4 M5x15 screws.



3. Secure the Aluminium U-Channel on top of the top sheet using 4 M5x15 screws and 4 M5 nuts.

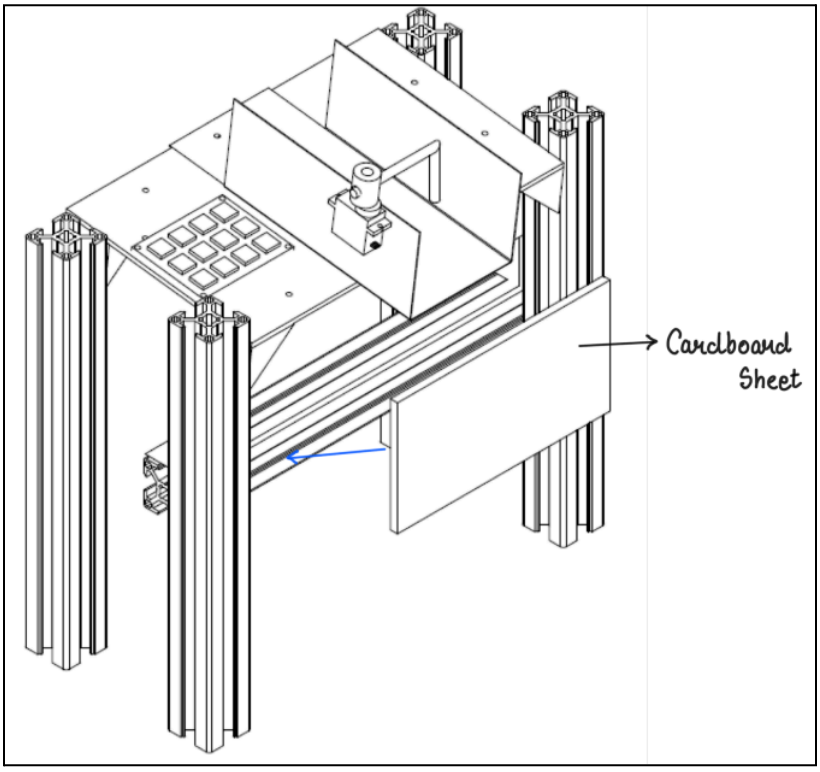


4. Tape the electronics to their respective positions using a strong double-sided tape: Numpad, SG90 Servo, ESP32.

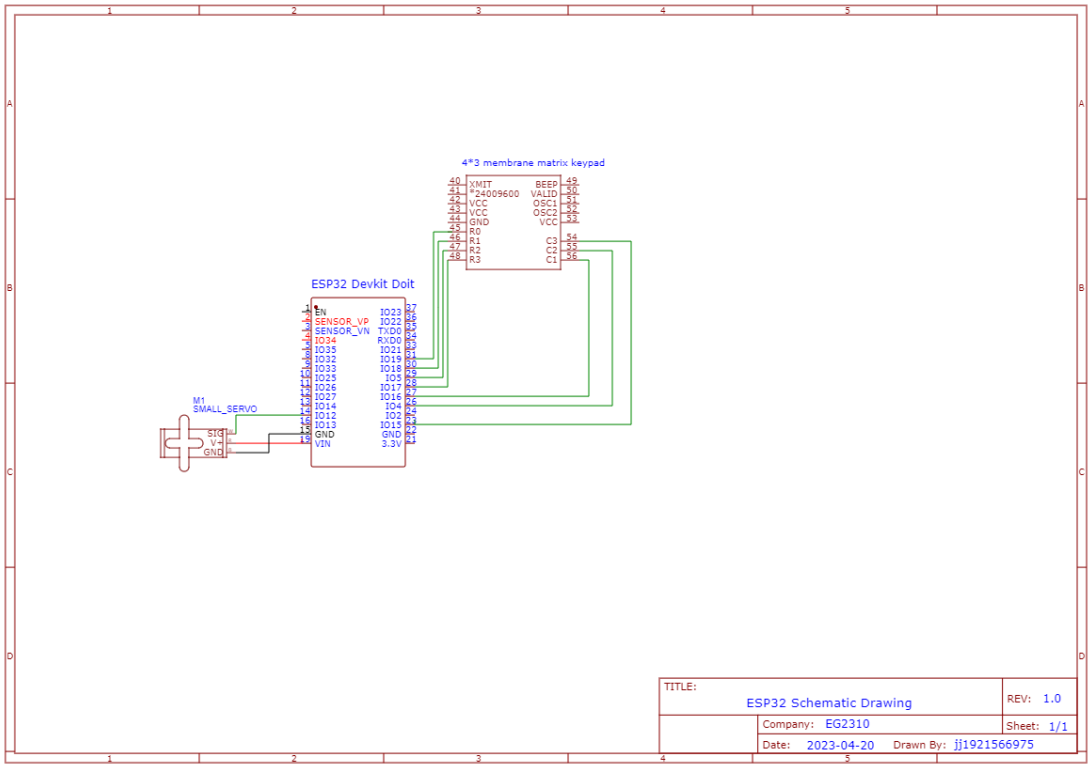


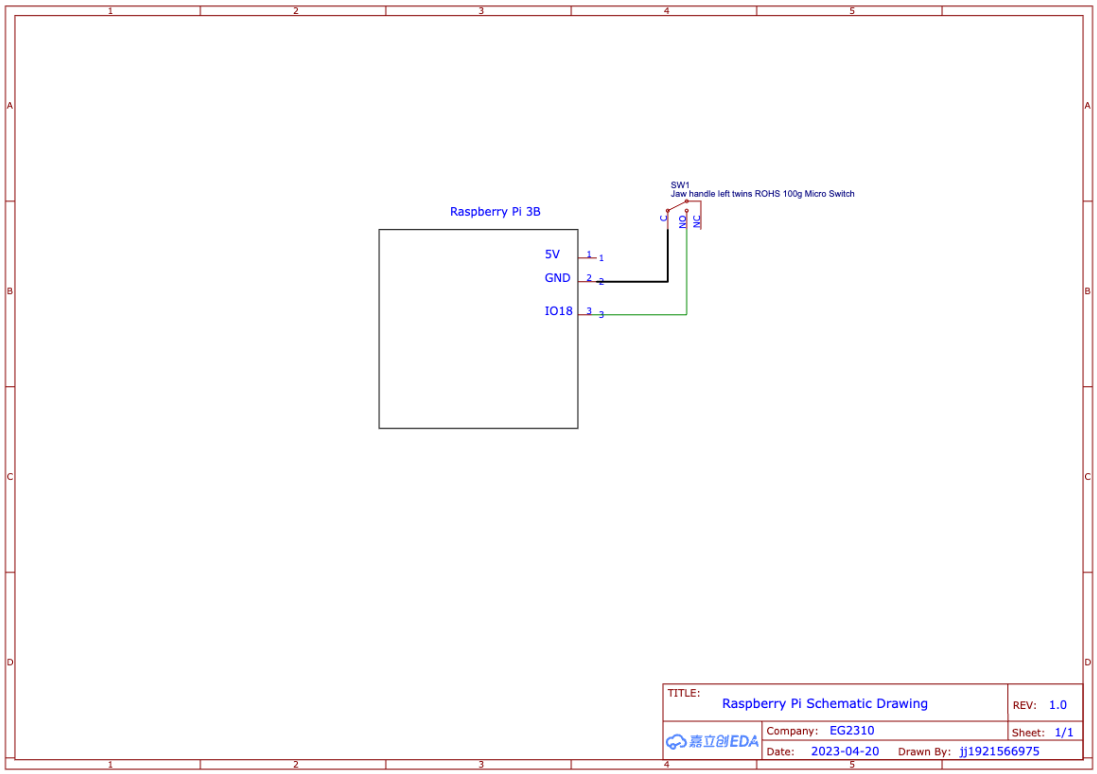
5. Wire the electronics appropriately as specified in the electrical diagram (Section 7.3).

6. Attach a cardboard of 20cm by 10cm to the front of the dispenser.



7.3 Electrical Diagram[7]:





7.4 Software Assembly:

7.4.1 Basic Installation

1. On the laptop, ensure that you have Ubuntu 20.04 and ROS 2 Foxy installed. Refer here for how to install the required software. Ensure that you are following the instructions under the "Foxy" tab.
2. Test that the ROS development environment is working by ensuring that a simple working publisher and a subscriber can be created using the instructions here.
3. Using Ubuntu, follow the instructions here, and burn the ROS 2 Foxy Image to the SD card onto the RPi on the Turtlebot3. Follow through the "Quick Start Guide" to get a working Turtlebot3.
4. Test that the ROS development environment is working by ensuring that a simple working publisher and a subscriber can be created using the instructions here.
5. Once the ROS development environment is working on both the remote laptop and the RPi, run the publisher from the RPi and the subscriber on the laptop, and ensure that the subscriber on the laptop replicates what is being produced by the publisher. Swap the device that the publisher and subscriber are publishing from to ensure that two-way communication between the RPi and the remote laptop can be established.
6. Add the following lines to .bashrc file in the root directory of the laptop.

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

7. Add the following lines to .bashrc file in the root directory of RPi.

```
export TURTLEBOT3_MODEL=burger
alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py'
```

8. Create an AWS ec2 instance by following the instructions here

7.4.2 Installing the program on remote laptop

1. Create a ROS 2 package on the remote laptop and move the file in the directory temporarily to the parent directory.

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

mv __init__.py ..
```

2. Clone the GitHub repository to the remote laptop. Make sure the period at the end is included.

```
git clone https://github.com/kaijie0102/turtlebot_ws.git .
```

3. Remove unnecessary folders since you have already created your own workspace

```
rm -rf build install log .git .gitattributes README.md
```

4. Build the package

```
cd ~/colcon_ws
colcon build
```

5. Add the following lines in .bashrc in the root directory of the laptop

```
alias sshrp='ssh ubuntu@`ssh aws cat rpi.txt`'
export TURTLEBOT3_MODEL=burger
alias rteleop='ros2 run turtlebot3_teleop teleop_keyboard'
alias rslam='ros2 launch turtlebot3_cartographer cartographer.launch.py'
alias autonav='cd ~/colcon_ws/src/auto_nav/auto_nav2'
alias setwp='autonav && python3 setwp_map2base.py'
alias map2base="autonav && python3 map2base.py"
```

6. Add the following lines in .bashrc in the root directory of raspberry pi

```
alias switch=’ros2 run autonav switch_motor’
```

7.4.3 Setting waypoints

1. Place your bot in front of the dispenser and perform the steps listed in **7.4.2** to start the program. Mark this position
2. Start teleoperation

```
setwp
```

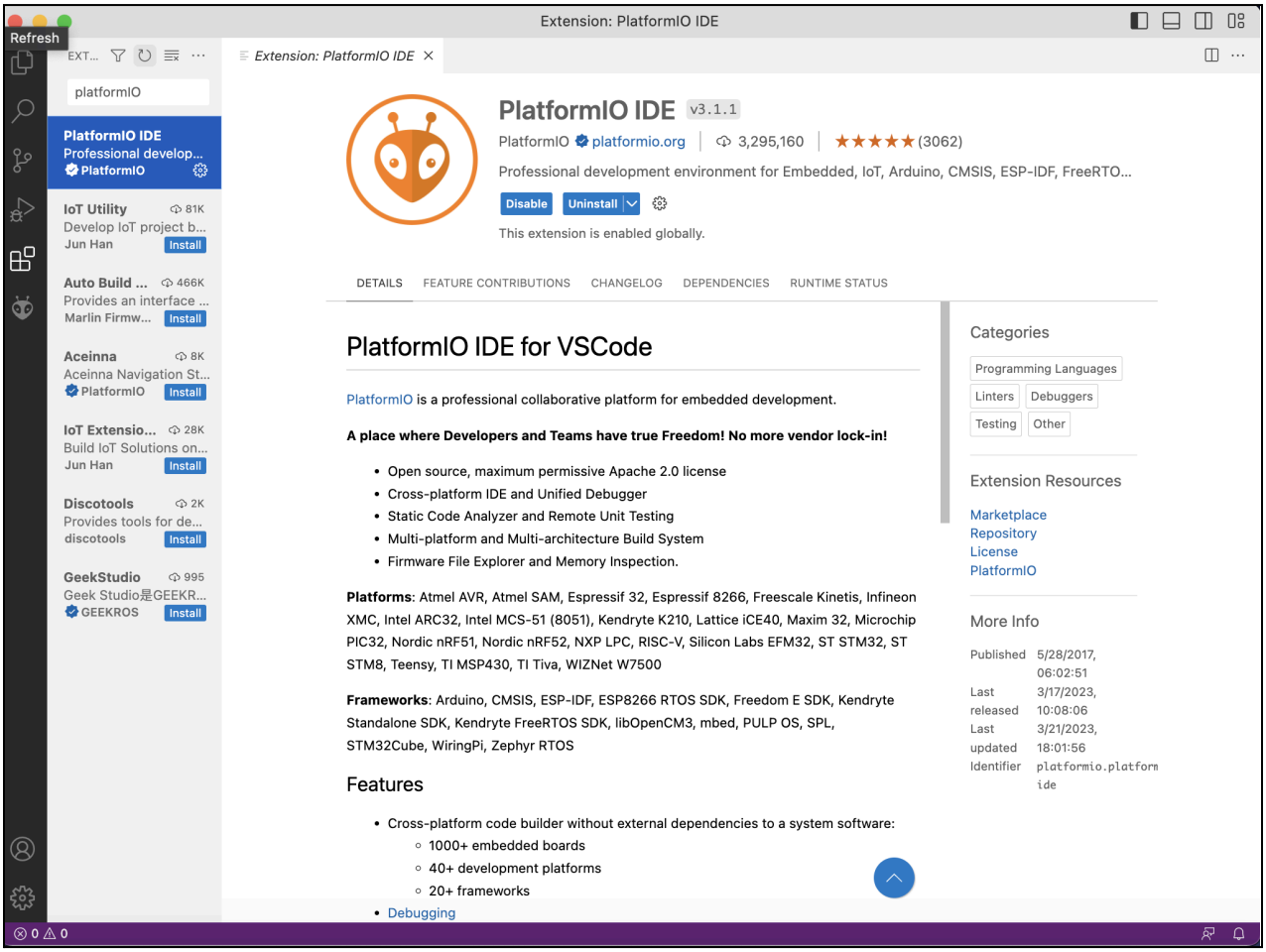
3. Move your bot to the desired waypoint using the controls shown and select ‘p’ and then input the table number. The table number should appear in a json file called waypoints.json

7.5 Code explanation

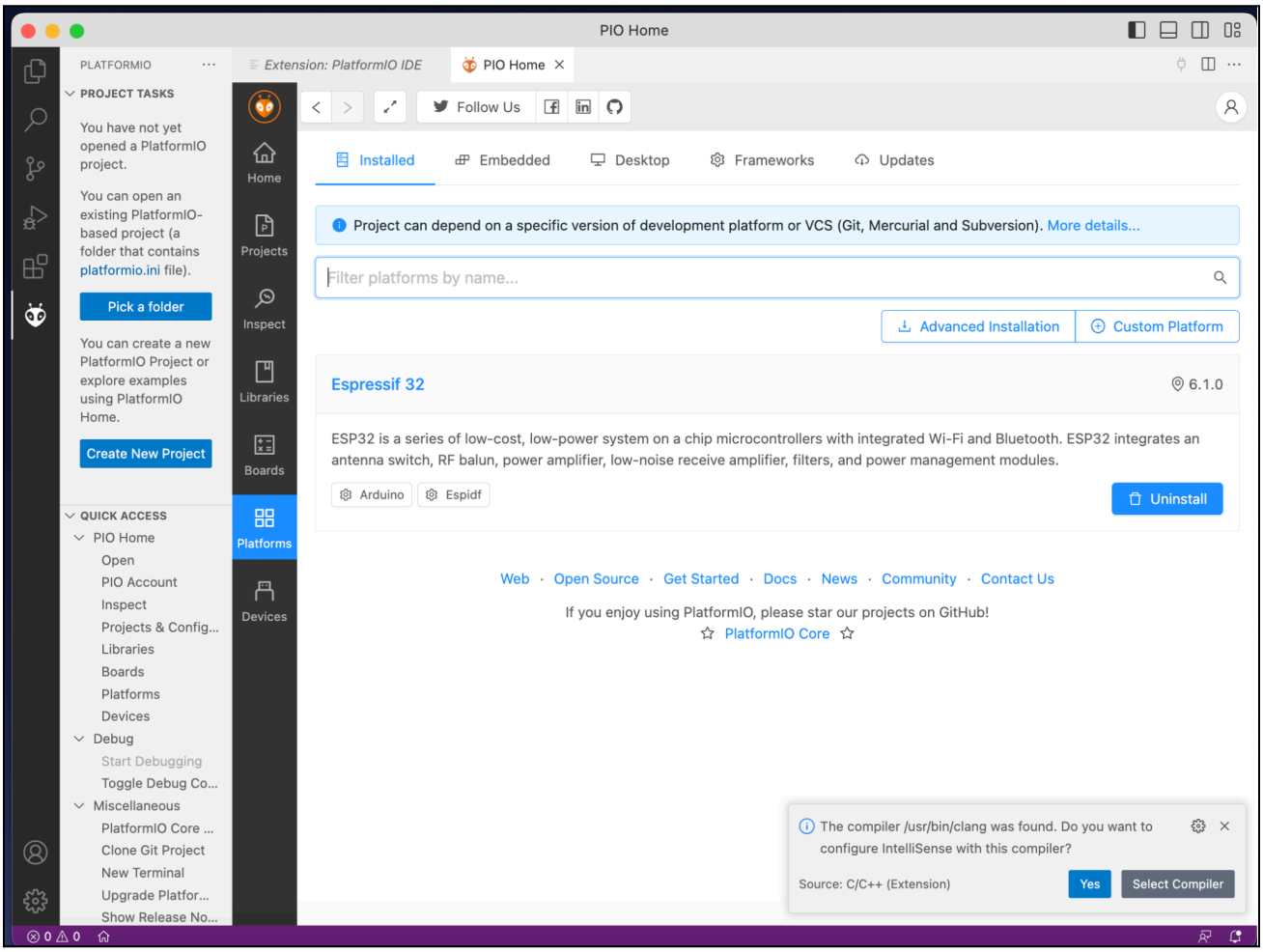
7.5.1 Code Explanation (ESP32[8] & HTTP communication[9])

PlatformIO[10] setup:

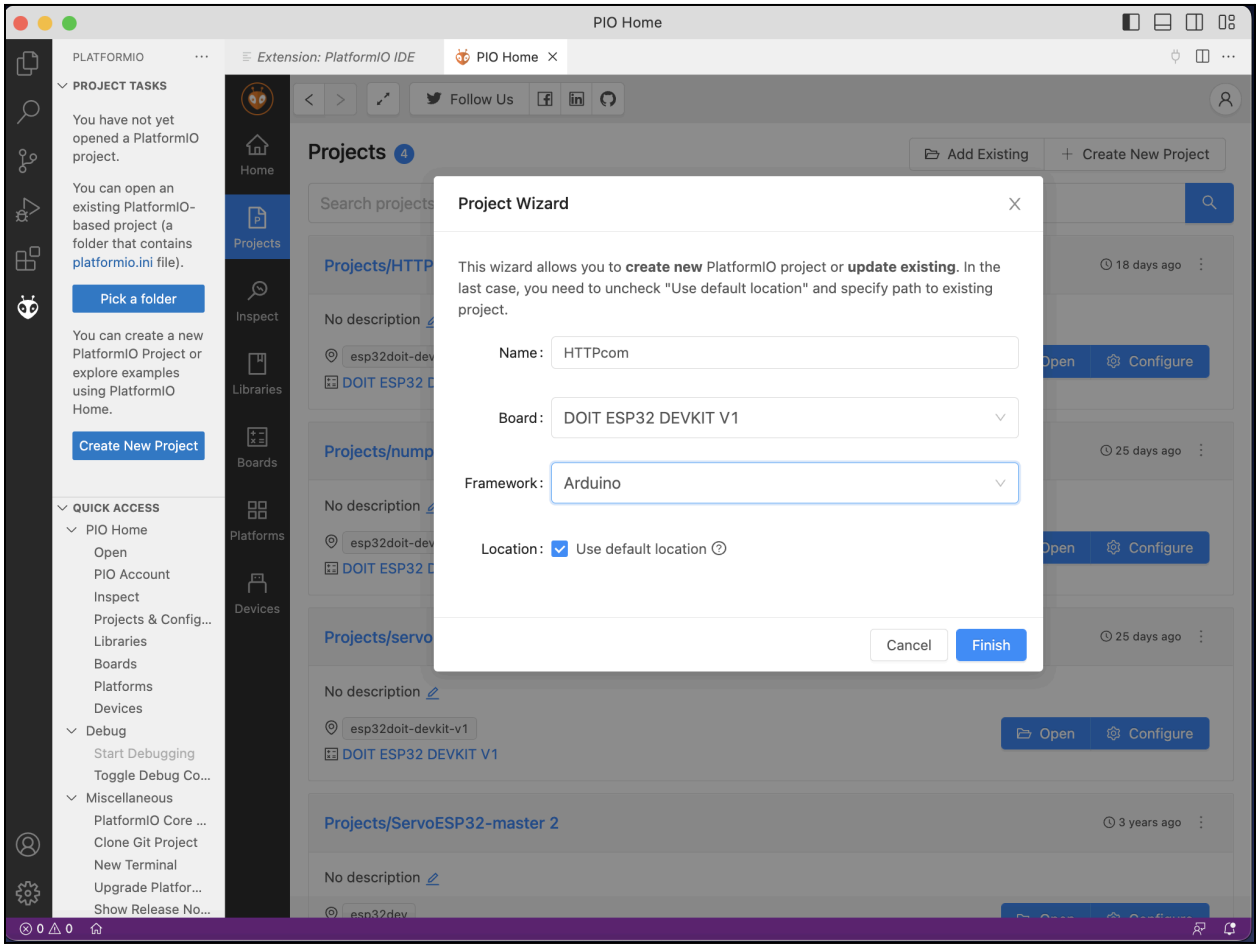
1. Search ‘PlatformIo in Extensions in VScode and install



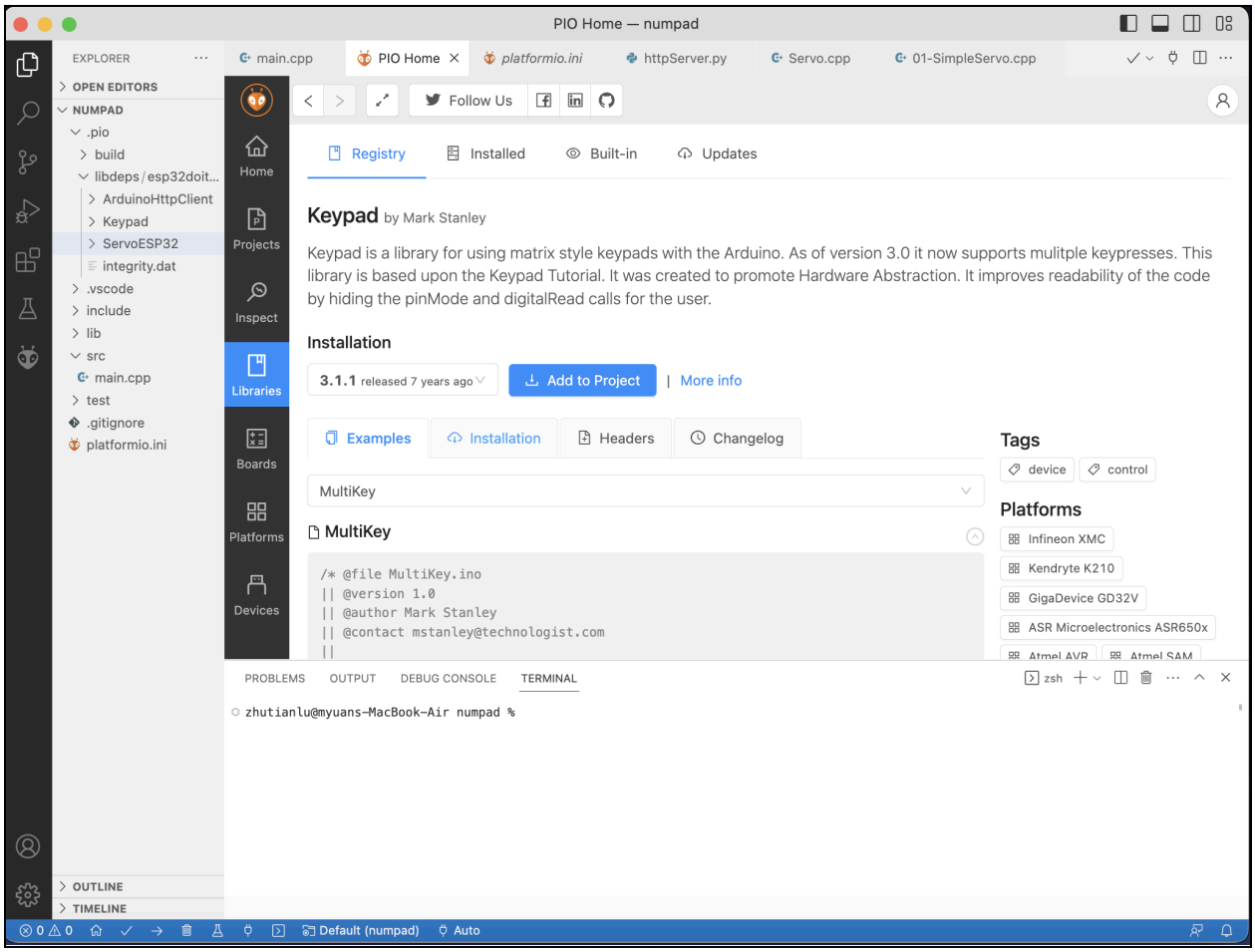
2. On the homepage of PlatformIO, search the ‘Espressif 32’ in the Platforms and install it.



3. Create a new project on the Homepage, search Board name ‘DOIT ESP32 DEVIT V1’ and choose the framework ‘Arduino’.



4. Search Servo motor, HTTP, Keypad, WIFI and String libraries from PlatformIO and import them to the current project.



5. In the file src/main.cpp
   Define the rows and columns of the matrix keypad.
   Define the correct IP address and Wireless network information.
   Define the GPIO Pin that the servo motor is connected to.

```cpp
#include <Arduino.h>
#include <WiFi.h>
#include <HTTPClient.h>
#include <string>
#include <Keypad.h>
#include <Servo.h>

#define ROW_NUM     4 // four rows
#define COLUMN_NUM  3 // three columns

const char* ssid = "myuan";
const char* password = "98765432";

//Your Domain name with URL path or IP address with path
// String serverName = "http://172.20.10.9:2401";
String serverName = "http://192.168.1.32:8000";
// create a Servo object
Servo servo1;
// set the pin number for the servo motor
const int servoPin = 12;
```

6. Setting up WIFI connections and initiating the servo motor.

```cpp

void setup() {
  HTTPClient http;
  Serial.begin(9600);
  WiFi.begin(ssid, password);
  Serial.println("Connecting");
  while(WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.print("Connected to WiFi network with IP Address: ");
  Serial.println(WiFi.localIP());
  // attach the servo to the specified pin
  servo1.attach(servoPin);
}
```

7. After ensuring the Network connection, setting up the keypad matrix to give a correct output number when the corresponding number button is pressed.

```
37
38  void loop() {
39    HTTPClient http;
40    // http.begin("172.20.10.9:2401",true);
41    http.begin(serverName,true);
42      //Check WiFi connection status
43    if(WiFi.status()== WL_CONNECTED){
44      String serverPath = serverName;
45      // Your Domain name with URL path or IP address with path
46      http.begin(serverPath.c_str());
47      // Send HTTP GET request
48      // int httpResponseCode = http.POST(&payload,sizeof(payload));
49      char keys[ROW_NUM][COLUMN_NUM] = {
50        {'1', '2', '3'},
51        {'4', '5', '6'},
52        {'7', '8', '9'},
53        {'*', '0', '#'}
54      };
55      byte pin_rows[ROW_NUM] = {19,18,5,17};
56      byte pin_column[COLUMN_NUM] = {16,4,15};
57      Keypad keypad = Keypad( makeKeymap(keys), pin_rows, pin_column, ROW_NUM, COLUMN_NUM );
58      // Serial.println(keypad.getKey());
59      char key;
60      key = 0;
```

```
61  ;       while (key == 0){
62          key = keypad.getKey();
63          Serial.println(String(key));
64        }
65        String payload = String(key);
66        Serial.println(payload);
67        Serial.println(String(key));
```

8. The program will wait until the variable key is not empty. This implies that a key has been pressed, and the signal of the “pressed key number” will be sent to Raspberry Pi via HTTP.POST. At the same time, the servo motor is activated and rotates by 90 degrees, and it will rotate back to its original position after 10 seconds.

```
68
69          if (String(key)!=""){
70            int httpResponseCode = http.POST(payload);
71            Serial.println(httpResponseCode);
72            // delay(200);
73
74            if (httpResponseCode > 0) {
75              Serial.println("HTTP Response code: " + String(httpResponseCode));
76              String response = http.getString();
77              Serial.println(response);
78              Serial.println(String(key));
79              if (String(key)!= ""){
80                servo1.write(0);
81                delay(10000);
82                servo1.write(90);
83              }
84
85            }
86            else {
87              Serial.print("Error code: ");
88              Serial.println(httpResponseCode);
89            }
90
91          }
92          // Free resources
93        http.end();
94        }
95        else {
96          Serial.println("WiFi Disconnected");
97        }
98      // lastTime = millis();
99      // }
100   }
```

9. HTTP Server code
   When the HTTP server is running on the RPi, it will allow ESP32 to communicate with RPi.

```python
from http.server import BaseHTTPRequestHandler, HTTPServer
import logging

class S(BaseHTTPRequestHandler):
    def _set_response(self):
        self.send_response(200)
        self.send_header('Content-type', 'text/html')
        self.end_headers()

    def do_GET(self):
        logging.info("GET request,\nPath: %s\nHeaders:\n%s\n", str(self.path), str(self.headers))
        self._set_response()
        self.wfile.write("GET request for {}".format(self.path).encode('utf-8'))

    def do_POST(self):
        content_length = int(self.headers['Content-Length']) # <--- Gets the size of data
        post_data = self.rfile.read(content_length) # <--- Gets the data itself
        logging.info("POST request,\nPath: %s\nHeaders:\n%s\n\nBody:\n%s\n",
                str(self.path), str(self.headers), post_data.decode('utf-8'))

        self._set_response()
        self.wfile.write("POST request for {}\n".format(self.path).encode('utf-8'))

def run(server_class=HTTPServer, handler_class=S, port=8000):
    logging.basicConfig(level=logging.INFO)
    server_address = ('', port)
    httpd = server_class(server_address, handler_class)
    logging.info('Starting httpd...\n')
    try:
        httpd.serve_forever()
    except KeyboardInterrupt:
        pass
    httpd.server_close()
    logging.info('Stopping httpd...\n')

if __name__ == '__main__':
    from sys import argv

    if len(argv) == 2:
        run(port=int(argv[1]))
    else:
        run()
```

7.5.2 Code Explanation (Main Script)

This is the explanation for the main script

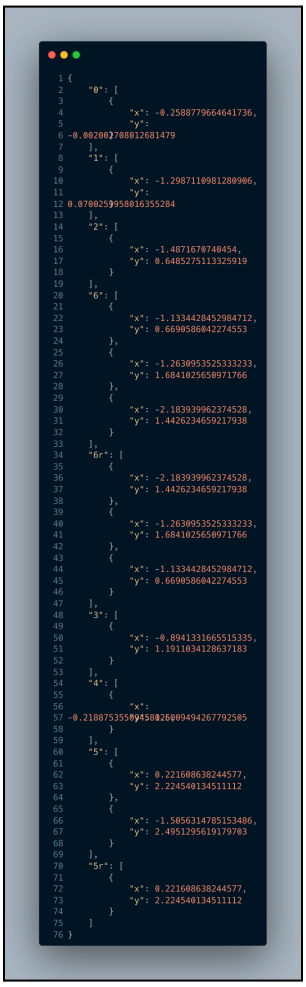
1. Initialization and waiting for user input. readKey() is the entry point to the script right after the main. It will take in user input and a function will be performed based on user input

```
def readKey(self):
        twist = Twist()
        try:
            while True:
                # get keyboard input
                rclpy.spin_once(self)
                cmd_char = str(input("Keys w/x/a/d -/+int s/p/auto:
"))

                # use our own function isnumber as isnumeric
                # does not handle negative numbers
                if isnumber(cmd_char):
                    # rotate by specified angle
                    self.rotatebot(int(cmd_char))
                else:
                    # print("callback is called: ")
                    # check which key was entered
                    if cmd_char == 's':
                        # stop moving
                        twist.linear.x = 0.0
                        twist.angular.z = 0.0
                    elif cmd_char == 'w':
                        # move forward
                        twist.linear.x += speedchange
                        twist.angular.z = 0.0
                    elif cmd_char == 'x':
                        # move backward
                        twist.linear.x -= speedchange
                        twist.angular.z = 0.0
                    elif cmd_char == 'a':
                        # turn counter-clockwise
                        twist.linear.x = 0.0
                        twist.angular.z += rotatechange
                    elif cmd_char == 'd':
                        # turn clockwise
                        twist.linear.x = 0.0
                        twist.angular.z -= rotatechange
                    elif cmd_char == 'p':
                        # set waypoint
                        self.set_waypoint()
                    elif cmd_char == 'auto':
                        self.auto_navigate()
                    elif cmd_char=='ff':
                        self.face_front()
                    elif cmd_char=='b':
                        self.backward()
                    elif cmd_char=='fs':
                        self.forward_stop()
                    elif cmd_char=='fb':
                        self.face_back()

                    # start the movement
                    self.publisher_.publish(twist)

        except Exception as e:
            print(e)

            # Ctrl-c detected
        finally:
            # stop moving
            twist.linear.x = 0.0
            twist.angular.z = 0.0
            self.publisher_.publish(twist)
```

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

    waypoint = Waypoint()
    waypoint.readKey()

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    waypoint.destroy_node()

    rclpy.shutdown()
```

2. Setting waypoints. The following function will save the waypoints into a json file

```python
def set_waypoint(self):

    # open json files containing waypoints and deserialise to
dict    wp_file = "waypoints.json"
    f = open(wp_file, 'r+')
    existing_waypoints = json.load(f)

    # print("existing waypoints", existing_waypoints)

    table_number = input("Enter table number: ")

    # check if key exist. Exist? Append:Create new entry
    if (table_number not in existing_waypoints):
        existing_waypoints[table_number] = [
            {"x": self.x_coord, "y": self.y_coord}]
    else:
        existing_waypoints[table_number].append(
            {"x": self.x_coord, "y": self.y_coord})

    # writing to json file
    f.seek(0)
    json.dump(existing_waypoints, f, indent=4)
    f.close()
```

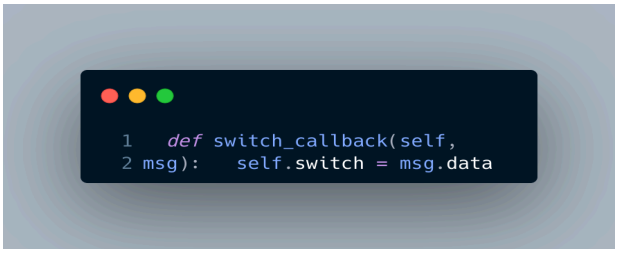



```
{
    "0": [
        {
            "x": -0.2588779664641736,
            "y":
-0.002002708012681479
        }
    ],
    "1": [
        {
            "x": -1.2987110981280906,
            "y":
0.07002599580163552 84
        }
    ],
    "2": [
        {
            "x": -1.4871670740454,
            "y": 0.6485275113325919
        }
    ],
    "6": [
        {
            "x": -1.1334428452984712,
            "y": 0.6690586042274553
        },
        {
            "x": -1.2630953525333233,
            "y": 1.6841025650971766
        },
        {
            "x": -2.183939962374528,
            "y": 1.4426234659217938
        }
    ],
    "6r": [
        {
            "x": -2.183939962374528,
            "y": 1.4426234659217938
        },
        {
            "x": -1.2630953525333233,
            "y": 1.6841025650971766
        },
        {
            "x": -1.1334428452984712,
            "y": 0.6690586042274553
        }
    ],
    "3": [
        {
            "x": -0.8941331665515335,
            "y": 1.1911034128637183
        }
    ],
    "4": [
        {
            "x":
-0.21887535589458826009494267792505
        }
    ],
    "5": [
        {
            "x": 0.221608638244577,
            "y": 2.224540134511112
        },
        {
            "x": -1.5056314785153486,
            "y": 2.4951295619179703
        }
    ],
    "5r": [
        {
            "x": 0.221608638244577,
            "y": 2.224540134511112
        }
    ]
}
```

3. Obstacle detection. The scan_callback function will take in data from the scan topic and check if the front of the bot has any obstacles, if yes, stop

```
def scan_callback(self, msg):

        # create numpy array
        self.laser_range = np.array(msg.ranges)
        # replace 0's with nan
        self.laser_range[self.laser_range==0] = np.nan

        # find index with minimum value
        lr2i = np.nanargmin(self.laser_range)


        self.left_laser = self.laser_range[90]

        if (self.special_table):
            if ( self.laser_range[0]<0.2 or self.laser_range[1]<0.2 or self.laser_range[359]<0.2 or
  self.laser_range[358]<0.2 or self.laser_range[5]<0.2 or self.laser_range[10]<0.2 or
  self.laser_range[355]<0.2 or self.laser_range[350]<0.2 or self.laser_range[270]<0.2 or
  self.laser_range[200]<0.2 or self.laser_range[290]<0.2 or self.laser_range[30]<0.2 or
  self.laser_range[20]<0.2 or self.laser_range[10]<0.2):
                self.get_logger().info('SPECIAL STOP!!')
                self.obstacle=True
                self.stop()

        # if the front of robot senses something in front, stop
        if (self.laser_range[0]<0.21 or self.laser_range[1]<0.210 or self.laser_range[359]<0.210 or
  self.laser_range[358]<0.210):
            if (not self.docked):
                self.get_logger().info('STOP!!')
                self.obstacle=True
                self.stop()

```

4. Detection of the can. Pi is subscribed to the /limit_switch topic which publishes “false” if the microswitch is released and “true” if the microswitch is pressed, signifying that a can is on it



```
def switch_callback(self,
msg):    self.switch = msg.data
```

5. Auto-navigation
    a. Bot will only begin navigation towards the table if the table number is sent to it and the can is on it. Ie, there will be a while loop to wait for the table number and another while loop to wait for the switch to be pressed.

```
post_file = "serverData.json"
        file_size = os.stat(post_file).st_size
        while True:
            self.p_dock=False
            self.slow = False
            self.get_logger().info("Please input table number into keypad: ")
            # getting table number by checking if there is a new entry in serverData.json
            while True:
                rclpy.spin_once(self)
                self.obstacle=False
                current_size = os.stat(post_file).st_size
                if current_size>file_size:
                    file_size=current_size
                    self.docked=True

                    # if there is new entry, break out of the while loop
                    break

            # UNCOMMENT!


            while (not self.switch):
                # print(self.laser_range[0])
                rclpy.spin_once(self)


            self.get_logger().info("Can detected!")
```

b. Obtaining new waypoint from json file and proceeding towards it

```
# open file and read the latest value
            f = open(post_file, 'r')
            existing_data = json.load(f)
            target_table = existing_data["postD"][-1]

            self.get_logger().info("Walking to table: %s"%target_table)

            self.backward()
            self.obstacle=False
            self.face_back()
            self.docked=False
            self.get_logger().info('Leaving dock...')

            current_yaw = math.degrees(self.yaw)
            self.get_logger().info('Starting auto-navigation: X:%f|Y:%f|Facing:%f ' %
                                   (self.x_coord, self.y_coord, current_yaw))

            # open json files containing waypoints and deserialise to dict
            wp_file = "waypoints.json"
            f = open(wp_file, 'r+')
            existing_waypoints = json.load(f)

            target_waypoints = existing_waypoints[str(target_table)]

            # target_waypoints is an array of dict
            count = 1
            for item in target_waypoints:
                # visiting every waypoint
                curr = (self.x_coord, self.y_coord)
                self.get_logger().info("curr x: %f curr y: %f" %
                                       (curr[0], curr[1]))
                # curr_x = self.x_coord
                # curr_y = self.y_coord
                new = [0, 0]
                for i in item.values():
                    # storing new x and y to travel to
                    # obtaining x first
                    if (count == 1):
                        count += 1
                        new[0] = i
                        self.get_logger().info("Obtaining new x: %f" % (new[0]))
                    else:
                        # after both x and y coordinate is obtained
                        count -= 1
                        new[1] = i
                        self.get_logger().info("Obtaining new y: %f. Facing: %f" %
   (new[1],current_yaw))
                        new = tuple(new)
                        self.move_to_point(curr,new)
```

c. Movement to waypoint - rotating to face the new endpoint. Sometimes the robot will be stuck while trying to rotate, so there is a self.rotate_stuck flag that attempts to help the robot unstuck

```
def move_to_point(self, curr, new):
        twist = Twist()
        current_yaw = math.degrees(self.yaw)
        norm = distance.euclidean(curr, new)
        self.get_logger().info("Norm is: %f" % (norm))
        target_angle = abs(np.arctan((new[1] - curr[1]) / (new[0] - curr[0])))

        # converting from radian to degree
        target_angle = target_angle * 180 / PI

         # calculating actual angle to turn, after taking into account the bot's bearings
         # make bot turn
        quadrant = self.checkQuadrant(curr, new)
        # self.get_logger().info("new point is in quadrant %d" % (quadrant))

        # switch case to check where new point is relative to old point
        if quadrant == 1:
            target_angle = 0-target_angle
        if quadrant == 2:
            target_angle = 0+target_angle
        if quadrant == 3:
            target_angle = 180-target_angle
        if quadrant == 4:
            target_angle = -180+target_angle

        # self.get_logger().info("Target Angle: %f" % (target_angle))
        final_angle = target_angle-current_yaw
        # self.get_logger().info("Pre Final Angle: %f" % (final_angle))
        if final_angle > 180 or final_angle < -180:  # to avoid turning the long
            final_angle = final_angle - 360


        self.rotatebot(final_angle)
        while (self.rotate_stuck==True):
            self.rotatebot(40)
            self.rotatebot(final_angle)
            self.rotate_stuck=False
```

d. The movement to the waypoint - moving towards the endpoint. When the Euclidean distance between the robot's current point and the endpoint is less than 0.04 (ln 18), the robot will stop. If the robot overshoots the point (ln 19) the robot will also stop. If there is an obstacle(ln 27), the robot will stop as well. If the robot is still far, the robot will move fast, else it will move slowly (ln2-5)

```
1  norm = distance.euclidean(curr, new)
2          if norm>0.7:
3              self.slow=False
4          else:
5              self.slow=True
6
7          # moving forward
8          print(self.slow)
9          if not self.slow:
10             self.forward()
11         else:
12             self.forward_slow()
13             self.slow = True
14
15         prev_norm = 999
16         recalib_counter=0
17
18         while (norm > 0.04 and
19 (norm<=prev_norm+0.02)):
20             prev_norm = norm
21
22             norm = distance.euclidean(curr, new)
23
24             rclpy.spin_once(self)
25             curr = (self.x_coord, self.y_coord)
26
27             if self.obstacle:
28                 # count=0
29                 self.stop()
30                 self.get_logger().info("oKOOok i
31 stop")
32                 # self.backward()
                   break
```

e. Recalibration. The robot will periodically recalibrate itself after 50 times in the while loop (ln3)

```
1 recalib_counter+=1
2             norm = distance.euclidean(curr, new)
3             if recalib_counter>50 and norm > 0.4 and not self.obstacle:
4                 self.get_logger().info("recalib, continue walking to x:%f y:%f"%
5 (new[0],new[1]))self.stop()
6                 self.move_to_point(curr,new)
7
```

f. Checking if the robot is actually at the endpoint when it exits the loop. If still too far away (ln 3). It is a false alarm that the robot has reached. Try to move towards the point again

```
1 norm = distance.euclidean(curr, new)
2         # if stop but norm too big, recallibrate
3         if (norm > 0.08 and not self.obstacle):
4             self.get_logger().info("norm:%f is too far!, continue walking to x:%f y:%f"%
  (norm,new[0],new[1]))
5             curr = (self.x_coord, self.y_coord)
6             self.stop()
7             self.move_to_point(curr,new)
8
9         current_yaw = math.degrees(self.yaw)
10        self.get_logger().info("MOVEMENT ENDS!")
11
12        self.stop()
```

g. Special Table number 6. The function to handle a special table is called. The bot tries to walk forward slowly when it is at the entrance of the special room and checks to its left (ln 5) to see if there is an object. Once the object is found, the bot faces the object (ln13) and walks toward it (ln 20). It will then wait for the can to be lifted (ln 27) before going back to the dispenser

```
1 # if special table 6 is called
2             if target_table=="6":
3                 self.get_logger().info("Starting
4 special")       self.face_back()
5                 # self.special_table=True
6                 self.handle_special()
7                 # after reaching entrance of table 6
8                 rclpy.spin_once(self)
```

```
1 def handle_special(self):
2         # detection of special table
3         self.get_logger().info('Starting detection of special table')
4         self.forward_slow()
5         while (self.og_left_laser - self.left_laser<0.5):
6             self.get_logger().info('DIFF: %f' %(self.og_left_laser - self.left_laser))
7             rclpy.spin_once(self)
8         self.get_logger().info('DIFF FINISH: %f' %(self.og_left_laser -
9 self.left_laser))      time.sleep(0.5)
10
11        self.get_logger().info('Located special table!')
12        self.stop()
13        self.rotatebot(90)
14        self.special_table=True
15        curr = (self.x_coord, self.y_coord)
16        new = (self.x_coord-0.1, self.y_coord-1.8)
17        self.get_logger().info("walking from: (%f,%f) to (%f,%f)" %
18                               (curr[0], curr[1],new[0],new[1]))
19        self.slow=False
20        self.move_to_point(curr,new)
21
22        self.docked=True
23        self.get_logger().info('Reached special table!')
24
25        while (self.switch):
26            rclpy.spin_once(self)
27        self.get_logger().info("Can lifted!")
28        time.sleep(3)
29
30        # go back to base by reverse waypoints
31        self.special_table=False
32        if self.obstacle:
33            self.handle_obstacle()
34        wp_file = "waypoints.json"
35        f = open(wp_file, 'r+')
36        existing_waypoints = json.load(f)
37        target_waypoints = existing_waypoints["6r"] # table 6 return journey
38
```

h. Docking. After the can is lifted at each table, the bot searches for the dispenser waypoint represented by the character “0” and moves back to the waypoint in front of the dispenser.

```
self.get_logger().info("Proceeding to dockk!")

            # walking back to dispenser
            dock_waypoints = existing_waypoints["0"]
            dock_count = 1

            for item in dock_waypoints:

                # obtaining new set of way point
                if (self.obstacle):
                    self.handle_obstacle()

                # visiting every waypoint
                curr = (self.x_coord, self.y_coord)
                self.get_logger().info("curr x: %f curr y: %f" %
                                       (curr[0], curr[1]))
                # curr_x = self.x_coord
                # curr_y = self.y_coord
                new = [0, 0]
                for i in item.values():
                    # storing new x and y to travel to
                    # obtaining x first
                    if (dock_count == 1):
                        dock_count += 1
                        new[0] = i
                    else:
                        # after both x and y coordinate is
obtained
                        dock_count -= 1
                        new[1] = i
                        new = tuple(new)
                        self.move_to_point(curr,new)
```

i. Precise docking. After the bot is at the waypoint of the dispenser, it will adjust according to the distance on its right to make sure it is directly in front of the dispenser, before finally moving straight into the dispenser.

```
    def precise_dock(self):
        self.face_front()
        self.rotatebot(-90)
        self.p_dock=True

        self.stop()
        while (self.laser_range[0]<0.514 or
self.laser_range[0]!=0.516 and self.laser_range[0])
            if (self.laser_range[0]<0.516):
                twist = Twist()
                twist.linear.x = -0.05
                twist.angular.z = 0.0
                self.publisher_.publish(twist)
            else:
                twist = Twist()
                twist.linear.x = 0.05
                twist.angular.z = 0.0
                self.publisher_.publish(twist)

            rclpy.spin_once(self)
        self.stop()
```

```
self.face_front()
self.forward_v_slow(
)
time.sleep(1)
self.face_front()
# time.sleep(0)
self.obstacle=False
self.forward_v_slow(
)
```

8. System Operation Manual

8.1 Charging the battery

1. Connect the Li-Po battery's charging head to the charger's port as shown in the picture below.

2. Connect the charger to a wall outlet using the provided black cable. If the red LED light turns on, the Li-Po battery is being charged. Once the battery is fully charged, the green LED light turns on.

8.2 Software Boot-up Commands and Terminals

8.2.1 Running the program

1. Make sure laptop and pi both connected to same network(hotspot)
2. Open 1st Terminal on laptop and start ros on the pi

```
sshrp # ssh into pi
rosbu # run this command after successful ssh into pi
```

3. Open 2nd Terminal to take in lidar data and create a map on rviz.

```
rslam
```

4. Open 3rd Terminal: to run a node that publishes the coordinates of the robot in the map. Map's origin is where rslam is called

```
map2base
```

5. Open 4th Terminal: to start the main script for auto navigation

```
setwp
```

6. Open 5th Terminal: to start HTTP server, awaits input from the keypad connected to the ESP32

```
autonav
python3 httpServer.py
```

7. Open 6th Terminal: runs a node that publishes the state of the microswitch, to check whether there is a "can" or not

```
sshrp
switch
```

8.2.2 Mission

1. Place your bot in front of the dispenser, the position that you have marked out in step 1 of 7.4.3, and perform the steps listed in **7.4.2** to start the program.
2. Start teleoperation

```
setwp
```

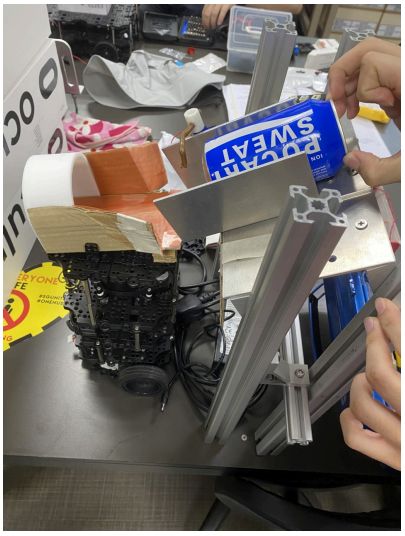
3. Start auto navigation

```
auto
```

4. Load the can and wait for the number to be pressed on the keypad

8.3 Loading of can and mission

1) User will lay the can **on its side**, with the top of the can facing the bot in the channel of the dispenser before the blade of the servo motor.



2) User will input the table number from 1 - 6 using the number pad mounted on the dispenser. It will trigger the servo arm to rotate and the can is dispensed into the Turtlebot3 for delivery.
3) When the bot stops at the desired table, take the can out from the bot.

9. Troubleshooting

9.1 Troubleshooting - Hardware and Electronics

1. Parts are loose or shaky - fasten all screws tightly and ensure taped parts are properly taped
2. Servo does not rotate as intended
    a. Ensure wires are fully connected to esp32 - female head fully fitted onto GPIO pins
    b. Ensure wires are connected to the correct GPIO pins
3. The can inside the turtlebot container is shaking and occasionally triggering the microswitch erroneously because of the cavity at the bottom of the can. - make sure that the can is loaded such that the top of the can (which is a flat surface) faces the bot.
4. Microswitch is not connected well by using jumper wires - solder the wires properly to the pins of microswitch.
5. The can might be stuck halfway in the track of the dispenser - ensure sufficient lubrication on the U-channel using mineral oil
6. The robot is unable to consistently stop at the exact same position while dispensing, which may result in the can not sliding successfully into the container every time - we designed the entrance of the can container in a funnel shape, which has increased the tolerance for dispensing the can. Increase the size of the “funnel” as required.

9.2 Troubleshooting - Software

1. The laptop unable to connect to Raspberry Pi
    a. Make sure that laptop and raspberry pi are on the same network. Make sure not to use networks that run on VPN like NUS’s wifi as it causes some connectivity issues.
2. No map received - restart rslam and rosbu, try rosbu first then rslam.
3. Connection Refused - check same hotspot(NOT the nus), restart hotspot
4. RViz not working well - Re-rosbu
5. RViz often performing badly

```
cd /opt/ros/foxy/share/turtlebot3_cartographer/config
```

```
vim turtlebot3_lds_2d.lua
# set use_odom to False
```

10. Future scope of expansion

Although the bot was able to complete the mission with an accuracy of 97%, there were some elements in the system that could have been modified to make it more efficient. There were some design considerations that were out of scope given the time and resource constraints. This section will analyse the flaws in all three components of the ecosystem i.e. Software, Electrical, and Mechanical along with some possible improvements.

10.1 Mechanical

10.1.1 Turtlebot

1. The Can Holder could be made of a sturdier and smoother material, that still retains the flexibility property so as to accommodate the bending of the front of the Can Holder. Possible materials include copper plates.
2. The fifth waffle was a little too high, thereby raising the bot's centre of gravity and hence having a negative impact on its overall stability. Perhaps custom waffle supports of height 60 or 70mm could be used instead.

10.1.2 Dispenser

1. The structure can be made more stable and sturdier using aluminium profiles of accurate lengths in order to prevent instability and misalignment.
2. Although the method of taping the electronics down with a strong double-sided tape worked well within the requirements and scope of this project, for better long-term stability these electronics should be fastened down with screws as well. Better provisions such as drilling appropriate holes in the aluminium parts could be done to facilitate this.
3. Overall, the use of cardboard and tape was purely due to time and resource constraints. So for the future of this product, sturdier materials and methods would be used to make it a more complete end-user product.

10.2 Electrical

10.2.1 Turtlebot

1. A NFC reader can be used as a safety net to double-check the accuracy of docking. It also can help the robot stop at the same position for each docking process, which will make dispensing more accurate.

10.2.2 Dispenser

1. The keypad should have a cancel and confirm button, as well as a LED screen to provide feedback[11] to users for a better user experience.
2. Instead of setting a long delay time(10s) for the trapdoor to close, the RPi can send a signal back to the ESP32 when the microswitch is triggered, and then the trapdoor will close after successful dispensing.

10.3 Software

10.3.1 TurtleBot

1. When checking for the distance of the bot to a wall, instead of using only one point, the 0th degree, we could additionally use 2 other points to form an isosceles triangle. For example, using 10th and 350th degree, when the value of 0 shows that the distance to the wall in front of it is x, we need to make sure that the 10th and 350th deg are equal to confirm that the bot is indeed facing a wall and not just any random object.
2. Instead of stopping every now and then to adjust its angle, it would be good if we could make the bot move and turn at the same time by giving it both linear and angular velocity. This will save a lot of time for us.
3. Sometimes, there will be random errors that occur causing the robot to go rogue, for example when the SLAM node goes down. A reboot script[12] could be written in times like this to automatically restart the scripts to make the robot more robust.

10.3.2 Dispenser

1. The keypad can be replaced by using a phone application, it may be more intuitive and elegant for users to choose the table number.

Citations

[1]Santos, S. (n.d.). *Arduino - Limit Switch: Arduino tutorial*. Arduino Getting Started. Retrieved April 23, 2023, from https://arduinogetstarted.com/tutorials/arduino-limit-switch

[2] Florita, Santos, S., Charlie, Doelling, R., Fringer, A., Sam, Duperly, H., Adrianna, Garrett, N., Max, Dutta, U., & Jenna. (2022, September 4). *Arduino with load cell and HX711 amplifier (Digital Scale)*. Random Nerd Tutorials. Retrieved April 23, 2023, from https://randomnerdtutorials.com/arduino-load-cell-hx711/

[3] Robotis e-Manual. LDS for TurtleBot3. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/

[4](SLAM) Navigating While Mapping — Navigation 2 1.0.0 documentation. (n.d.). Nav2. Retrieved April 23, 2023, from https://navigation.ros.org/tutorials/docs/navigation2_with_slam.html

[*5] A* Search Algorithm. (2023, March 8). GeeksforGeeks. Retrieved April 23, 2023, from https://www.geeksforgeeks.org/a-search-algorithm/*

[6] Mills, A. (n.d.). Robotic Motion Planning: A* and D* Search. Retrieved April 23, 2023, from https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf

[7]Raspberry pi 3 B+ pinout with GPIO functions, schematic and specs in detail. eTechnophiles. (2023, April 3). Retrieved April 23, 2023, from https://www.etechnophiles.com/raspberry-pi-3-b-pinout-with-gpio-functions-schematic-and-specs-in-detail/

[8]Renzo Mischianti. (2023, January 13). *Doit ESP32 Dev Kit V1 high resolution pinout and Specs*. Renzo Mischianti. Retrieved April 23, 2023, from https://www.mischianti.org/2021/02/17/doit-esp32-dev-kit-v1-high-resolution-pinout-and-specs/

[9] Python SimpleHTTPServer - Python HTTP Server. (2022, August 3). DigitalOcean. Retrieved April 23, 2023, from https://www.digitalocean.com/community/tutorials/python-simplehttpserver-http-server

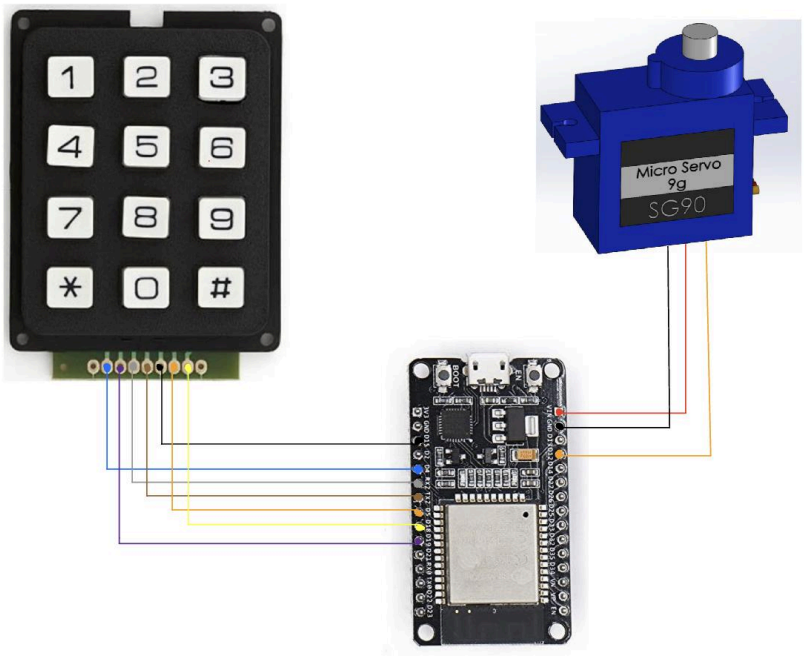
[10]PlatformIO. (n.d.). *Platformio is a professional collaborative platform for embedded development*. PlatformIO. Retrieved April 23, 2023, from https://platformio.org/install/ide?install=vscode

[11] What is the importance of feedback in user interface design? (2022, October 25). Axure. Retrieved April 23, 2023, from https://axureboutique.com/blogs/ui-ux-design/what-is-the-importance-of-feedback-in-user-interface-design

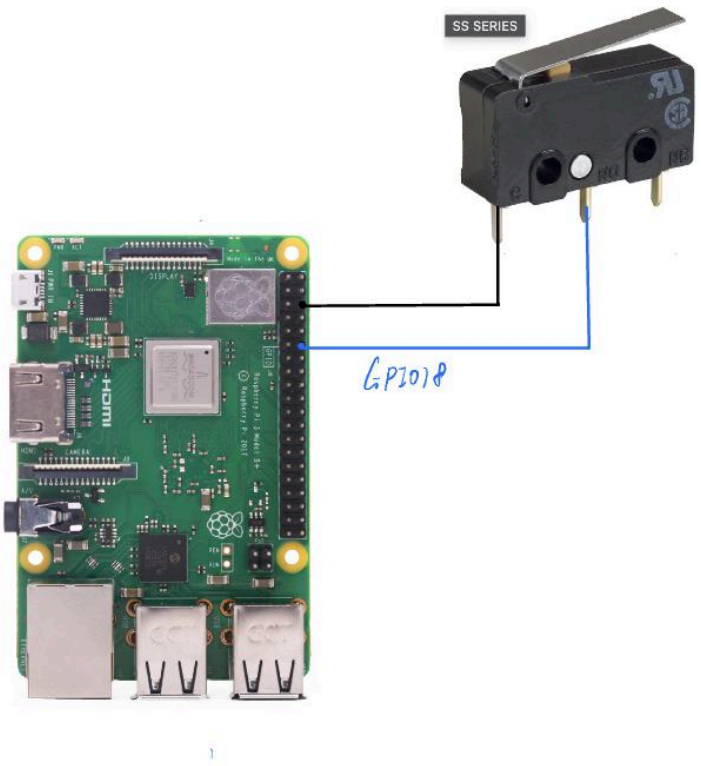
[12] Chouinard, J. (2022, October 9). Python Script Automation Using Task Scheduler (Windows). JC Chouinard. Retrieved April 23, 2023, from https://www.jcchouinard.com/python-automation-using-task-scheduler/

Annex

Annex A (Circuit Diagram for ESP32)



Circuit diagram for RPi

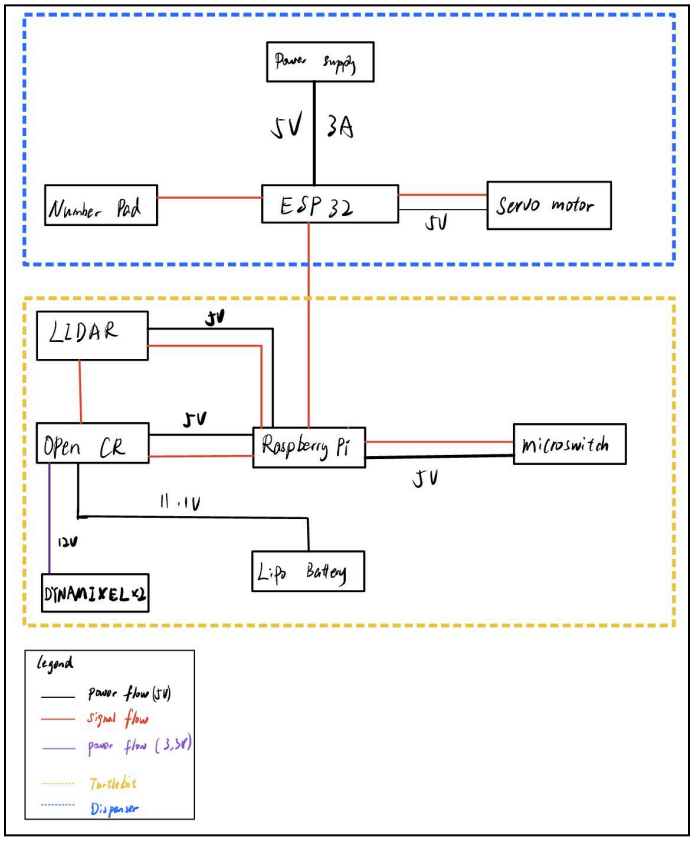


How It Works

Although The Limit Switch has 3 pins, a normal application usually uses only two pins: C pin and one of two remaining pins. Accordingly, there are four ways to use limit switch. The below is the wiring table for limit switch and the reading state on Arduino in all four ways:

C pin	NO pin	NC pin	Arduino Input Pin's State
1 **GND**	Arduino Input Pin (with pull-up)	not connected	**HIGH** when untouched, **LOW** when touched
2 **GND**	not connected	Arduino Input Pin (with pull-up)	**LOW** when untouched, **HIGH** when touched
3 **VCC**	Arduino Input Pin (with pull-down)	not connected	**LOW** when untouched, **HIGH** when touched
4 **VCC**	not connected	Arduino Input Pin (with pull-down)	**HIGH** when untouched, **LOW** when touched

Annex B (Electrical Architecture)



Annex C (Power Budget Table - Dispenser)

Components	Voltage	Current	Quantity	Power
ESP32	Typical Voltage: 3.3V	0.5A	1	1.65W
Membrane 3*4 Matrix Keypad	3.3V	less than 10mA	1	0.033W
SG90 servo Motor	5.0V	270mA	1	1.35W
Power plug	5.0V	2.5A	1	12.5W

Annex D (Power Budget Table - Turtlebot3)

Components		Voltage	Current	Quantity	Power
Turtlebot	Raspberry Pi	Standby: 12.105V Boot up: 12.105V Operation: 12.106V	Standby: 0.520A Boot up: 0.450A Operation: 0.637A	1	Standby: 6.52W Boot up: 5.44W Operation: 7.70W
	Lidar				
	OpenCR				
	Dyamixel				
NFC reader(PN532)		3.3V	100-200mA (in active state)	1	0.33-0.66W
microswitch		3.3V	Negligible	1	A few milliwatts

Annex E (Preliminary Monetary Budget)

No.	Item	Purpose	Cost + Link
1	Servo (SG-90 variants)	To operate the trapdoor	$6.60 (link)
2	Numpad	To take input of desired table number from the user.	$9.45 (link)
3	V-151-3A6 Microswitch - Solder terminal - 0.49N (50g) - 3.92N (400g) - datasheet:https://omronfs.omron.com/en_US/ecb/products/pdf/en-v.pdf	To detect the presence of can in the bot's cup holder	$2.08 (link)
4	ESP32	For digital communication between the Dispenser and the Bot	Self-source link
5	Fifth waffle - laser cut acrylic Total area needed: 276mm x 276mm	For placing the cup holder on the bot	Self-source/ Ask Miss Annie

	4mm thickness		
6	Cup holder - 3D printed with PLA	Storing the can in the bot	$5/h, 4h, $20
7	Aluminium sheet x3 One 132mmx66mm Two 132mmx40mm	For the rails on the dispenser	Self-source / Ask Miss Annie