EG2310: Fundamentals of Systems Design
Final Report
Group 11

Student Team Members

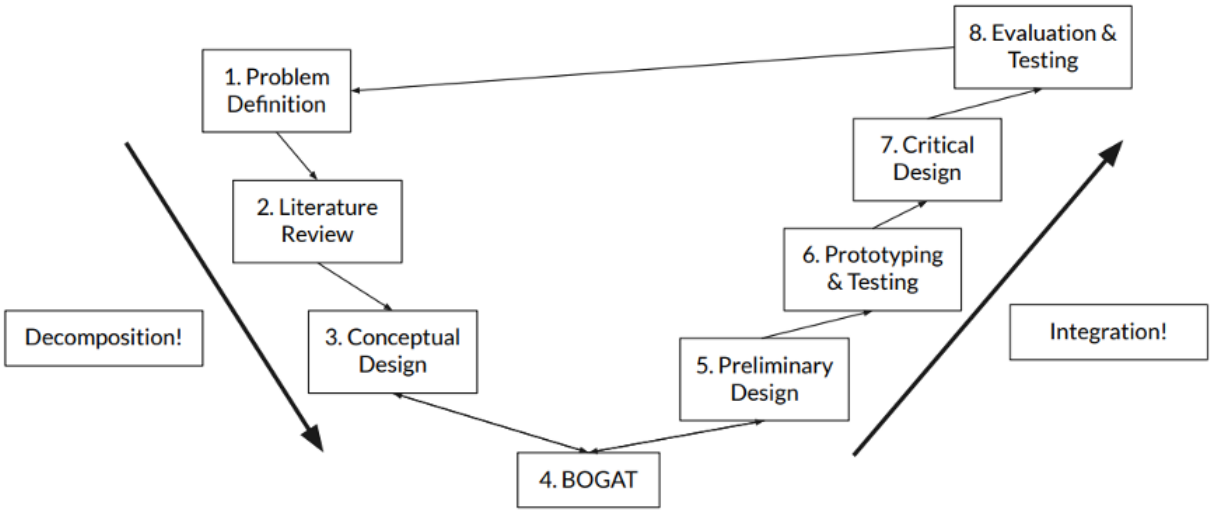
CHAE YEONGIN	A0264761Y
CHARLYN KWAN TING YU	A0258539M
CHU WEI RONG	A0262363H
KHOO KYE WEN	A0258789Y
MATHUR SAKSHAM	A0266609U

1. Introduction

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

> *The objective is to design and implement a TurtleBot system for automated delivery of canned drinks in a restaurant setting. The system must dispense a canned drink from a dispenser, have the TurtleBot navigate to a designated table based on user input, and prevent collisions with any part of the restaurant. Once the TurtleBot reaches the table, it must wait until the canned drink is collected before returning to the dispenser for further orders.*

The system design process utilised follows the modified EG2310 V-Model as shown below:



NUS EG2310 System Design V-Model

The following Sections detail each part of the system design process:

1. V-Model parts 1 - 3, 5 and 6 are described in their respective Sections from 2 - 6
2. BOGAT is subsumed under Section 4.6 Integration
3. Critical Design is described in detail over Sections 7 - 10
4. Evaluation & Testing is presented in Section 11 with a list of possible future works

2. Problem Definition

The project's ultimate objective is to design and construct a can dispenser and the TurtleBot burger into an autonomous waiter that delivers a canned drink to a table in a simulated restaurant within the given time limit of 26 minutes. The following are the subtasks that need to be accomplished:

Loading of can into dispenser

a) The TA can load the can onto a loading bay on the dispenser.
b) The dispenser stores the can securely until prompted to dispense it once the TurtleBot is docked and the table number is inputted.

Dispensing mechanism

a) The TA can press the designated table number on the keypad.
b) The stopper of the dispenser rotates horizontally to allow the can to drop.
c) A 'frontpack' is attached to the TurtleBot to store the can dropped from the dispenser.

Detection of can

a) The micro-switch in the 'frontpack' detects the presence of the can.
b) The TurtleBot starts its navigation immediately after a can is detected and a table number is selected prior.

Autonomous navigation

a) The TurtleBot autonomously navigates to the designated table.
b) The TurtleBot correctly identifies the designated table.
c) The TurtleBot waits near the table until the customer retrieves their canned drink, then navigates back to the dispenser.
d) The TurtleBot has sufficient power to autonomously start from and return to the dispenser to carry out deliveries within the restaurant.

This report highlights how these imperative subtasks can be tackled.

3. Literature Review

3.1 Structure of Dispenser

3.1.1 Capacity of dispenser

A dispenser may hold single or multiple cans. Holding multiple cans could allow the owner to load many drink orders at once, thus saving time of waiting and reducing inconvenience.

However, we decided to design a dispenser that only holds a single can. This simplifies the mechanics of our dispenser, making it easier to implement. The decrease in efficiency is negligible, as the objective is to deliver one can per trip.

3.1.2 Shape of dispenser

We considered using a slide to dispense the can. A stopper is placed at the end of this slide to prevent the can from dropping from a high position and at high speed, which could potentially damage the TurtleBot or the 'frontpack'. However, the can would not be dispensed vertically into the 'frontpack' in this case and might lead to misses.

Figure 1. Tube structure (source)

We then considered a Z-slide design which will straighten the drop of the can. However, it overcomplicates the structure by creating bends within the slide which may get the can stuck. Moreover, it might be difficult to find shapes that meet our requirements off the shelf; manufacturing it ourselves using methods such as 3D-printing would also be time-consuming and costly.

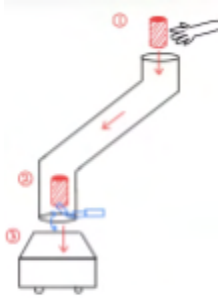


Figure 2. Diagram of Z-slide Dispenser

Integrating the dropping and stopper ideas, we decided to use an acrylic laser cut platform to support the MG995 servo motor. We added another platform on top that has a hole to stabilise the vertical standing of the can. The laser-cut stopper connected to the servo motor simply rotates horizontal to ground when dispensing the can into the 'frontpack'. This design is simple yet effective in reducing the shaking of the TurtleBot as the proximity between the stopper and the bottom of the 'frontpack' is low.

3.2 Dispenser Control Unit

3.2.1 Raspberry Pi 3B+

The Raspberry Pi 3B+ is a small single board computer that can control electronics while offering wireless communication. By being connected to the same network and using a publisher subscriber module, we can communicate the inputted table number to the TurtleBot.

3.2.3 ESP32

ESP32 is a microcontroller unit with integrated Wi-Fi and Bluetooth connectivity. Its built-in connectivity and processing power makes it ideal for projects requiring wireless communication. It

is recommended to be programmed in C++ and there are less resources available online on how to connect it to a Raspberry Pi which the TurtleBot uses.

3.2.4 Decision and Rationale

As both options have equally strong wifi and bluetooth capabilities, we have decided to use ESP32 which is the more economic option. Using the ESP32 fits our dispenser's functionality best as it is a microcontroller capable of performing the dispenser's functions while consuming less power than the RPi. RPi may be a too powerful microcomputer, hence redundant for our project.

3.3 Input System on Dispenser

3.3.1 Numeric Keypad

We can use a keypad with numeric input to indicate the destination table. Most common numeric keypads are membrane keypads, such as COM-08653, which are made up of top and bottom membranes. When a key is pressed, these conductive membrane traces come in contact with each other, thereby completing a circuit and sending a signal.

Figure 3. Keypad

3.3.2 Mobile App

We can use MIT App Inventor to create a mobile application which is more scalable for the long-run. Mobile apps also offer the flexibility of visualising the table location, enabling the TA to select the destination table more intuitively from diagrams as well as numbers. However, this method is relatively more difficult to set up and will introduce yet another device into the operation.

3.3.3 Decision and Rationale

The keypad will be used due to its convenience and simplicity. We will connect the keypad to the dispenser, then adopt the RPi to send the input signal to the robot. Using a mobile app can be helpful when restaurants perform multiple tasks and send different types of input but it may unnecessarily complicate simpler tasks such as this project.

3.4 Can Holding Structure on TurtleBot

3.4.1 Platform

Standoffs can be added to the top layer of the TurtleBot to create a platform to easily place the can. This would come at the expense of potentially (depending on thickness of bar) compromising data received by the LDS which is spinning continuously and mapping its surroundings. Moreover, the centre of gravity of the TurtleBot shifts further above the ground, making the motion of the TurtleBot less stable and increasing the risk of the bot toppling over during docking or in motion.

Figure 4. Example of Adding Top Platform

3.4.2 Trolley

Figure 5. Example of Trolley (source)

Another option is a trolley to be dragged behind, as the TurtleBot is not affected by the force or the orientation with which the can is loaded onto the robot. However, calculations need to be done as the path of the trolley needs to be smooth given that the TurtleBot makes sharp turns when circumventing boundary walls.

3.4.3 'Frontpack'

The TurtleBot 3 Burger comes with specially designed waffle plates to create its different levels. We can add another waffle plate to extend the bottom layer to create a platform for the 'frontpack' to sit. We can also add a castor ball under this new waffle plate so that the drop of the can onto the 'frontpack' does not cause the whole bot to tilt.

3.4.4 Decision and Rationale

While the trolley is easy to integrate with our TurtleBot, the uncertainty of the trolley not being able to evade possible obstacles during sharp turns outweighs its benefits. Then, adding a platform on top may cause the TurtleBot to be unstable as the centre of gravity rises after the can is loaded. On the other hand, it is much easier to add another waffle plate to the TurtleBot's bottom layer to create a base for the 'frontpack', as the plates are pre-designed to fit each other. Hence, we have decided to adopt the 'Frontpack' idea over the Trolley and Platform ideas. The reason for putting the backpack at the front of the TurtleBot instead of the back is to avoid difficulties during docking and to increase convenience of the customer as the drink will be presented in their direction.

3.5 Can Detection System on TurtleBot

3.5.1 Weight sensor

A weight sensor converts an input mechanical force, in this case the weight of the can, into an electrical output signal that can be measured to indicate that the can has been loaded.

3.5.2 Microswitch

When the weight of the can presses down the switch, the circuit will be closed, thus a signal will be sent to indicate the presence of the can. This is a simple yet effective way as the microwitch gives either a 1 or 0 (True or False) result.

3.5.5 Decision and Rationale

We decided to use the microswitch as it does not overcomplicate our system. Weight sensor reads a range of force values, which in the case of our project, is unnecessary as we only need to acquire information on the presence in the frontpack. A microswitch provides us with this functionality of an "on" and "off" value depending on whether or not the can is loaded, hence, it is simple and sufficient.

3.6 Path Planning and Obstacle Avoidance Algorithm

3.6.1 Hard-code

The first five tables are at fixed positions on the map, while the sixth table can be anywhere within a boundary. By hardcoding, we could calibrate the distance the robot moves and the angle it turns to reach each table. Using these fixed values, we will upload the code to the RPi to get the TurtleBot to move the exact same route to all the tables each time the code is executed. The same fixed code will be used for the TurtleBot to return back to the dispenser.

3.6.2 Waypoints

The TurtleBot will be navigating through a map. The TurtleBot will mark its starting coordinate as (0,0), which is the position right below the dispenser. To navigate to each table, different waypoints will be stored as checkpoints for the TurtleBot to arrive at, before moving to the next checkpoint. The resultant last waypoint stored will either be the table's position or the dispenser's position.

3.6.3 Decision and Rationale

Navigating through our maze requires very high accuracy for the TurtleBot to stop within 15cm from the table and exactly at the dispensing spot for the can to drop directly into the front-pack. Due to the high accuracy needed, we have decided to use waypoints as a way to navigate to and fro the dispenser and the tables. Hardcoding on the other hand will decrease the accuracy of our turtlebot's location as it only moves based on pre-fixed values, which is less accurate if we do not utilise LDS data. It will also be hard to readjust the stopping position of the robot if the layout of the restaurant changes as very accurate recalibration is required. Hence, in order to meet the end goal of being a restaurant delivery robot, we have decided on using waypoints.

3.8 Table Detection System

3.8.1 NFC Reader

One way that we can detect different tables is using NFC tags. It is similar to RFID as it is a two-way wireless communication between 2 devices. It would allow the robot to know the correct table to deliver the drink to. The drawback with this mechanism is that the TurtleBot needs to be extremely close to the NFC tag in order to read, which is not the most practical way of finishing the mission of this kind.

3.8.2 Number Recognition

A camera can be attached to detect physical numbers attached to the tables in the restaurant. This will make use of real-time computer vision and would allow the TurtleBot to recognise the destination table. This method requires the images from the camera to be clear enough to be read, which may not be possible given the robot's autonomous navigation around the restaurant.

3.8.3 Waypoint on Map

We will map the area out using cartographer and label the tables based on the map's coordinate system. The TurtleBot can estimate its initial position by overlaying the LDS data onto the recreated map. The TurtleBot can then navigate to the waypoint using the chosen navigation algorithm.

3.8.4 Decision and Rationale

We will be using a waypoint on map as it is the cheapest option available while being effective. The number recognition would require the number to be visible from various angles which might not be possible, it will also be computationally heavy to run a number recognition program. As for the NFC, if the robot is slightly too far or if the NFC is too weak, the robot might never detect the table. Hence, the waypoint is the most viable solution given that it complements our path planning algorithm and that it does not require any additional hardware to be set up.

3.9 Docking system

3.9.1 NFC Reader

One way that we can detect the alignment between the TurtleBot and the dispenser accurately is by using an NFC tag. The NFC reader would be onboard the TurtleBot and the tag would be attached right under the dispensing area, so everytime the bot made its way back to the dispenser, it would detect the tag that would indicate it to stop, hence being successfully 'docked'. The orientation with which the bot arrives at the dispenser would not be significant due to the shape of our 'frontpack' and the NFC would allow us to accurately stop right where it needs to.

3.9.2 Waypoints

One waypoint (A) can be set directly below the dispensing area and another one (B) can be set at a distance away from the dispensing area. When mapping and storing the waypoints based on the locations of all of the tables, we can instruct the bot to go back to B before going back to A. This is to ensure that docking is straight and in a desired manner every time. This method works in theory but would not work if the map data is inconsistent and if the map created by the bot drifts.

3.9.3 LIDAR (LDS)

This method would work by setting up a waypoint a certain distance away from where the bot needs to dock. The orientation of the bot would be fixed with the help of code, and then the bot could be instructed to go straight until the distance detected by LDS becomes a certain value. This method would work provided there is something solid on the dispenser which the LDS can detect, otherwise it will detect other objects and the docking will be unsuccessful.

3.9.4 Wall-following

This method would be a mixture of 2 mechanisms. This would work as we can set waypoints near walls and then allow the bot to do the wall-following all the way back to the dispenser. It would know when to stop based on the onboard NFC reader and tag mechanism that was explained earlier in this section.

3.9.5 Decision and rationale

We decided to use waypoints to initially test and then realised that this method was working well and with fine tuning, we would be able to make it reliable. On the other hand, using the wall-following would be an overcomplicated solution and the NFC idea, which we kept as our backup, heavily depended on the waypoints as the TurtleBot would have to land on top of the NFC tag in the first place and that would have depended on the waypoints being accurate. Therefore NFC was not used.

3.10 Intersystem communication

3.10.1 MicroROS

By flashing the dispenser microcontroller with MicroROS firmware, the microcontroller will be able to communicate using ROS topics which creates a more integrated system. If MicroROS was used, all the ROS topics would be shared with each other which could be useful in determining whether the TurtleBot has stopped to implement certain features like displaying the successful delivery on the dispenser display.

3.10.2 MQTT

MQTT is a lightweight, publish-subscribe, machine to machine network protocol for message queue/message queuing service. MQTT is well supported by microcontrollers using the pubsub client library on Arduino IDE which would be easy to implement. However, MQTT does require a device to serve as a broker to allow transmission of MQTT data.

3.10.3 HTTP Server

An HTTP server could be set up to send GET/POST requests to obtain the table number or send the table number. However, this could be far too complicated of a solution to integrate a HTTP server with the microcontroller as a HTTP server more capable of other functions. This makes the set up of the HTTP server more complicated than other choices.

3.10.4 Decision and Rationale

We will be using MQTT as the communication protocol due to its reliability in the microcontroller integration. MicroROS is not well supported by ROS2 Foxy which is the version used by the TurtleBot and the lack of resources and guides will make it more difficult. A HTTP server is more computationally heavy compared to the MQTT which could reduce the operating time of the TurtleBot, reducing the effectiveness of the TurtleBot as a can delivery robot.

4. Concept(s) Design

4.1 Autonomous navigation

For the autonomous navigation in this mission, two separate functions would be used. One for delivery to the table and one for return back to the dispenser. The waypoint system used would allow sharing of waypoints by the 2 functions. The general algorithm to navigate through the restaurant is to have preset waypoints for each table and allow the TurtleBot to follow the waypoints set for each table. This would assume that waypoints are set correctly with no obstacles between 2 intermediary waypoints as the TurtleBot will move forward towards the waypoint after rotating towards the next waypoint. There is a lack of obstacle detection algorithm as the TurtleBot will not be colliding with any obstacles if the waypoints are followed and the mission does not include any unexpected moving obstacles.

The waypoint information will be stored as a dictionary of lists of waypoints. The waypoints will use the x-y coordinate system to denote waypoints. These coordinates would come from one of the topics after running Cartographer. `/odom` was originally chosen as the topic but quickly replaced by `/map2base` due to reasons mentioned in the 4.5 Integration.

4.2 Microswitch

The microswitch used is D2F-01L. Its C and NC pins connected to the RPi on the TurtleBot, while connecting a pull-down resistor (10kΩ) and a current limit resistor (1kΩ) across it as well. The pull-down resistor is connected between the GPIO pin of the RPi and ground, and the microswitch is connected between the GPIO pin and the voltage supply (3V3 pin on the RPi). When the switch is open, the GPIO pin is connected directly to ground, and the pin reads a logic low. When the switch is closed, the GPIO pin is connected to the voltage supply through the pull-down resistor, and the pin reads a logic high. A current limit resistor is used to protect the GPIO pin from excessive current too. Once the microswitch is pressed by the can, data would be sent to the RPi. The small size of the microswitch allows for easy incorporation into the model of our ‘frontpack’ as well.

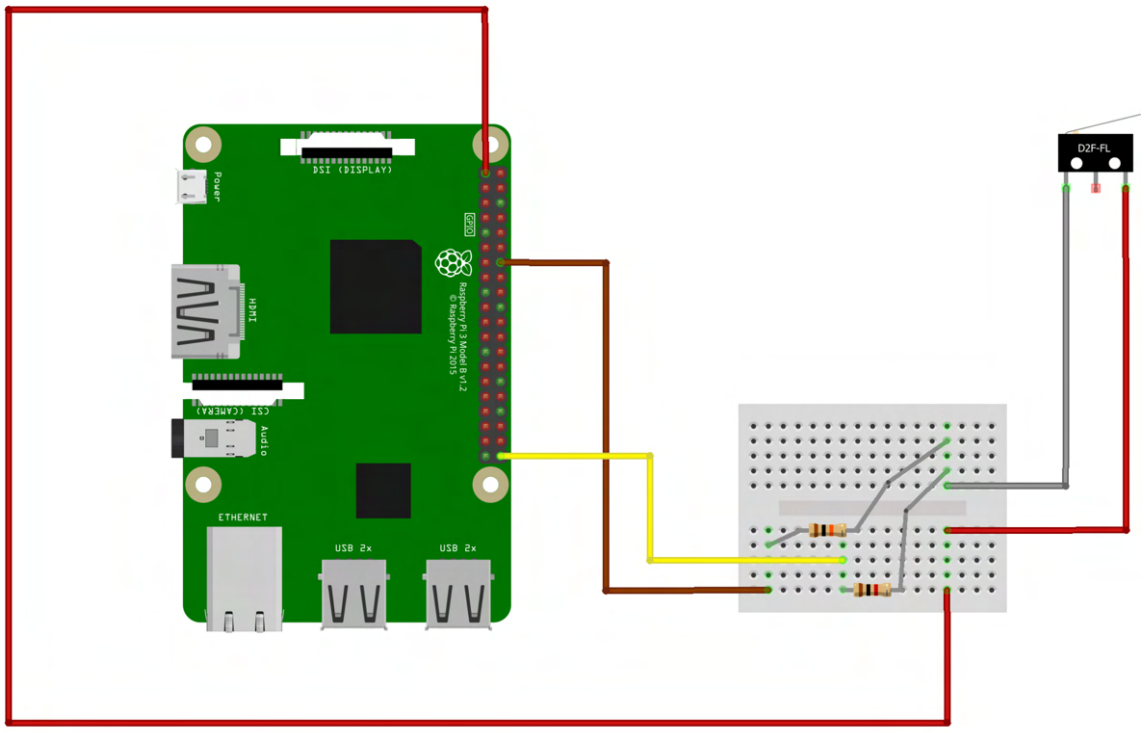


Figure 6. RPi connected to microswitch and resistors

4.3 Keypad and OLED

The keypad and OLED are used for the input and display system of the dispenser. The keypad would have its pins connected to the GPIO pins of the ESP32. For the OLED, the Vcc pin should be connected to the 3V3 pin of the ESP32, while the GND pin is connected to the GND pin of the ESP32. The SCL and SDA pins of the OLED are connected to the GPIO pins respectively. Once a number is pressed on the keypad, it would be shown on the OLED immediately. Once the enter (# key) is pressed, the data would be communicated to the RPi from the ESP32, indicating which table the TurtleBot should go to. If a wrong number is being pressed on the keypad, the * key could be pressed to delete it before inputting the new number followed by the # key.

4.4 Can dropping mechanism

As mentioned, we are doing a can dropping system, to favour the convenience of customers while retrieving their drink. The 3D- printed platform (stopper) attached to the servo (MG995) which is supporting the can would simply move away for it to drop.

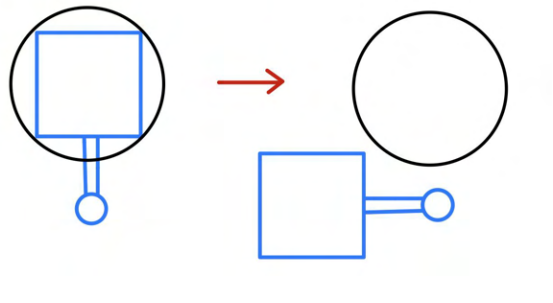
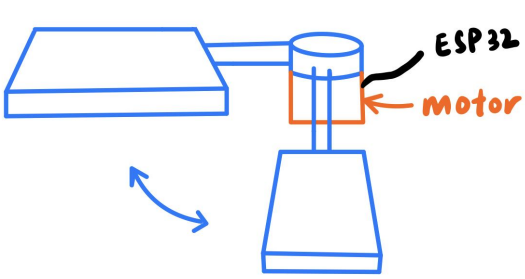
	

Figure 1. Stopper (in blue) & hole to insert can (circle in black) from the top view	Figure 2. Stopper with the motor connected to ESP32

Table 1. Stopper moving mechanism

The MG995 is hence also connected to the ESP32, with its Vcc pin connected to the voltage supply (5V5 pin) and the GND to the ground of ESP32. The signal pin of the servo is connected to the respective GPIO pin of the ESP32 as well. Once the # key is pressed on the keypad, data would be sent from the ESP32 to the servo to rotate for can to drop.

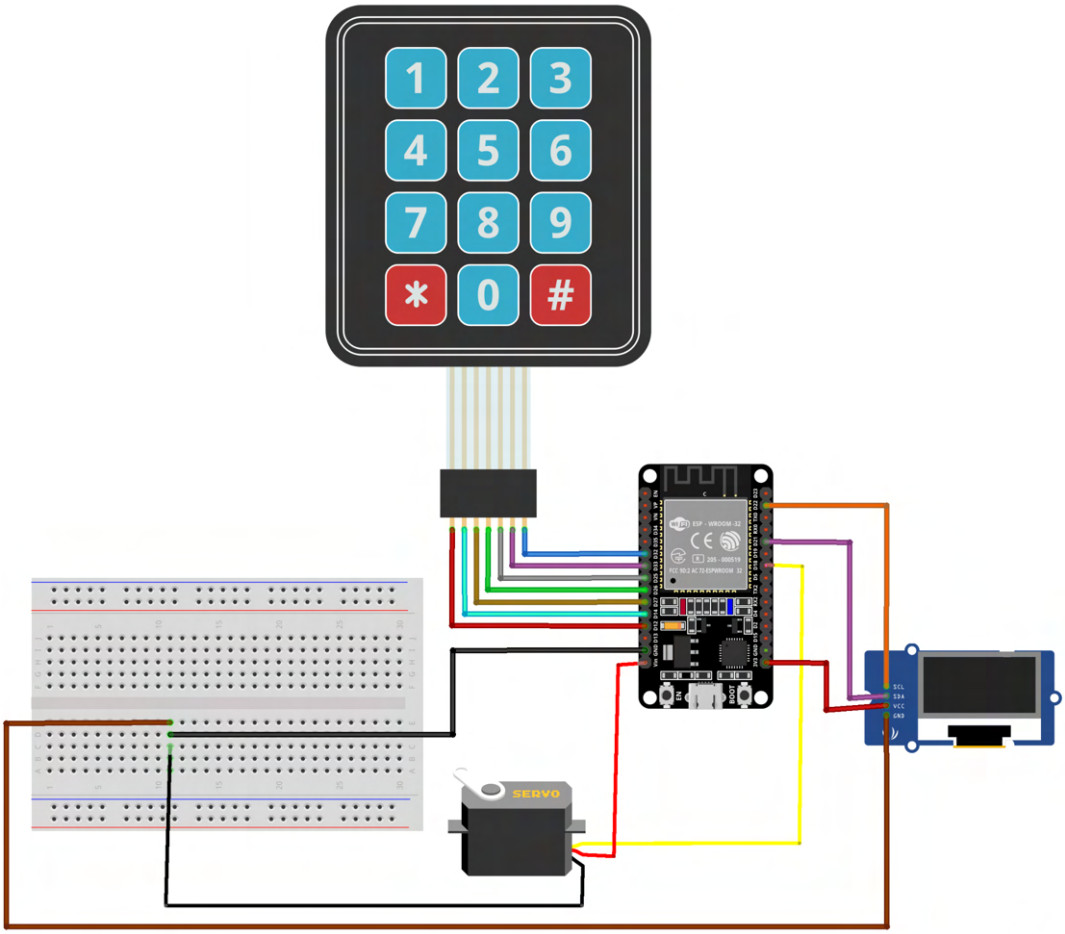


Figure 7. ESP32 connected to servo, OLED and keypad

4.5 Integration

Originally odometry was intended to be used to locate the TurtleBot. However, due to the existence of the ‘frontpack’ containing the can, the movement of TurtleBot was rough and could experience a slight bounce, especially during the receiving of the can. This slight bounce often caused the odometry data to drift, creating multiple layers of the maps in Cartographer. Instead of using odometry, we used `map2base`, a transformation of the transform data from Cartographer. This accompanied with the tweak to the Cartographer config file to disable odometry to be used resulted in a far more accurate data to locate the TurtleBot.

Due to multiple components connected to the ESP32 for the dispenser mechanism, wires would be used to connect the ground pin of the ESP32 to a separate row on a breadboard. This is to allow the components to be connected to a common ground due to the lack of GND pins on the ESP32.

For our ‘frontpack’ design, we decided to create a box made out of corrugated board which would capture the dropping can. The height of the ‘frontpack’ is taken into consideration to ensure that it does not block the LDS. The height of the middle and top plate was variable as we could just slide them across profile bars and adjust height accordingly. To support the weight of the can, the material, thickness and the shape of the stopper attached to the servo was taken into consideration. The stress analysis and calculations pertaining to friction during sliding was then conducted. The servo needs to be securely mounted onto our dispenser too which can be achieved with the help of the 3-D printed servo slot. The laser-cut stopper should be able to attach to the holes of the servo via screws. The height of the drop of can needs to be ideal too, so that the force of impact could be reduced, ideally should be the minimised how much ever it can. The keypad and OLED should be placed somewhere visible and easy to reach on the dispenser, with presentable and secure wire management and good cable management which would be imperative in the troubleshooting phase of the project. The design of our dispenser is stable, that is, the centre of gravity of it is low enough despite its potential height owing to the usage of 30x30 profile bars.

5. Preliminary Design

We divided the operation into 4 different phases.

1. Dispense phase: the owner loads the can into the dispenser, inputs the table number and dispenses the can into the ‘frontpack’
2. Delivery phase: the robot sends the can to the designated table while avoiding obstacles
3. Collection phase: the robot stays at the designated table until the can is collected.
4. Return phase: the robot navigates back to the dispenser and docks accurately for the next order

We will explain the objectives, systems, hardware and software involved.
(Refer to annex A for the general algorithm)
The functional block diagram for the overall system is as seen below:

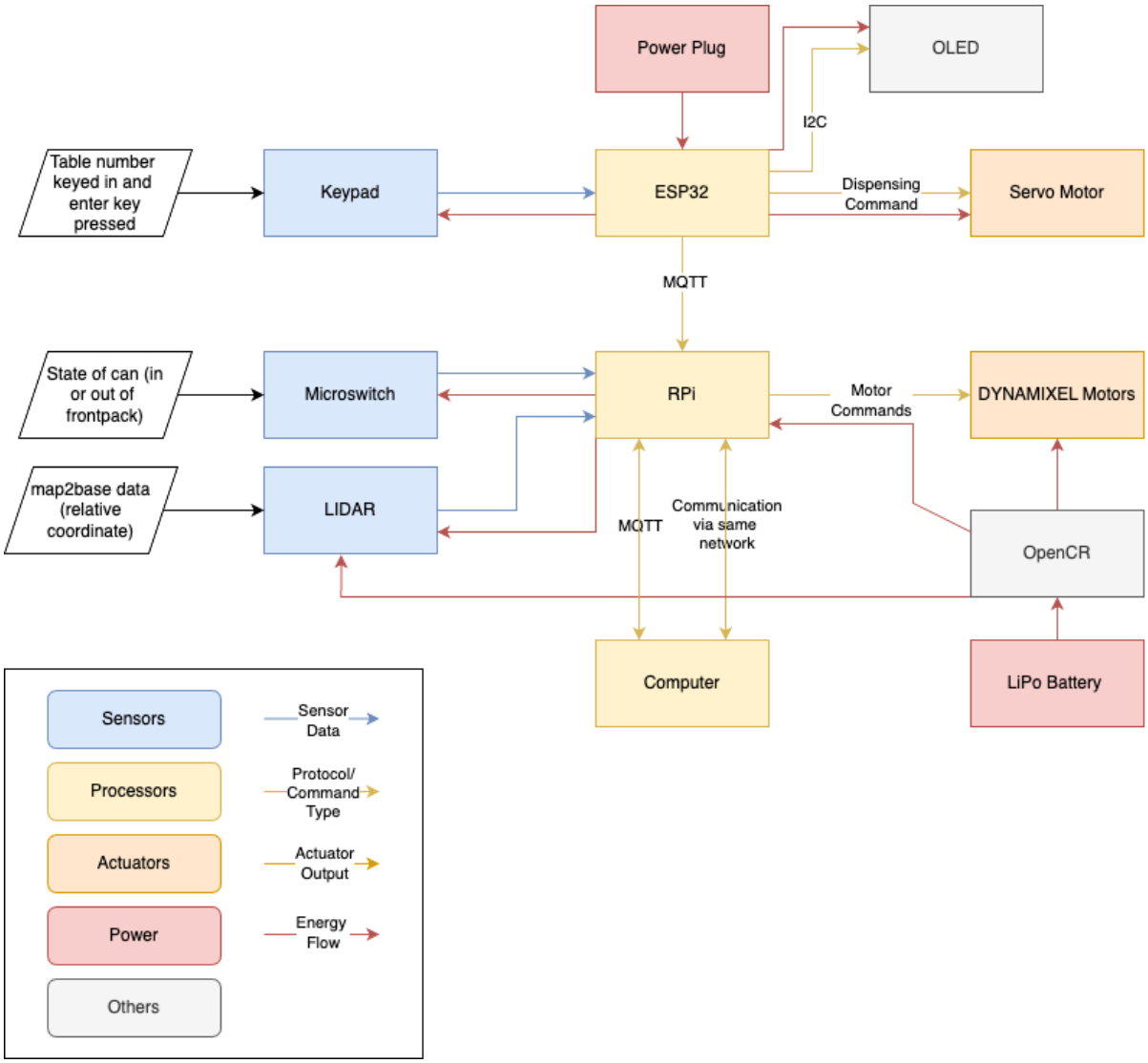


Figure 8. Functional Block Diagram

Overview of Systems

Phase	Dispense	Delivery	Collection	Return
Systems	Input system Dispensing system Can Detection System Communication System	Can Detection System Navigation System	Can Detection System	Navigation System
Hardware	**ESP32** 4 x 3 Matrix Keypad OLED MG995 Servo Motor **RPi** Microswitch	**RPi** OpenCR DYNAMIXEL Motors IMU LDS-01	**RPi** Microswitch	**RPi** OpenCR DYNAMIXEL Motors IMU LDS-01
Software	ESP32 program to 1. Display input, 2. Publish table number to TurtleBot 3. Actuate the Servo to drop the can Python program to 1. Detect activation of microswitch 2. Subscribe to the table number	Python program to 1. Read stored waypoints of the table 2. Navigate to the designated table	Python program to 1. Detect activation of microswitch	Python program to 1. Read stored waypoints of the table 2. Navigate to the designated table 3. Dock using LDS data

Table 2. Overview of systems in each phase

5.1 Phase 1 (Dispense)

Objective of this phase

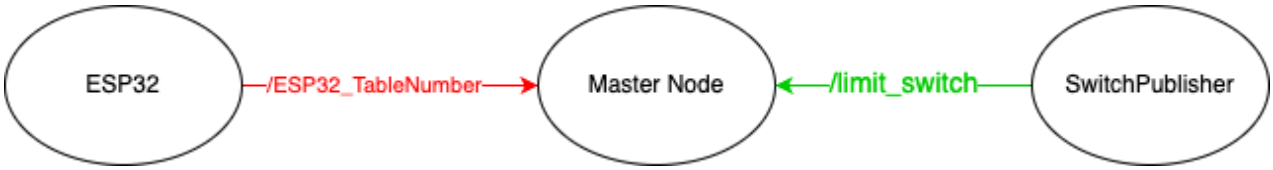
The TurtleBot is to receive the can and table number in this phase. It will proceed to phase 2 once a can is detected and the table number is received.

Description of operation

A can will be loaded into the dispenser and enter a table number for the can to be delivered. The OLED display on the dispenser will display the table number inputted by the user and will require the user to press the '#' key to confirm the table number. Alternatively, the user can press the '*' key to clear the table number inputted in case of wrong inputs.

Once the "#" key is pressed, the table number will then be published to the topic `/ESP32/TableNumber` and subscribed by TurtleBot using MQTT through WiFi connection. Parallely, the servo motor will turn the stopper 70 degrees and drop the can into the 'frontpack' of the TurtleBot. The TurtleBot will constantly be polling for the activation of the microswitch and the presence of table number in the python script.

The diagrams below show the communication, red line showing MQTT signal and green line showing ROS signal.



5.2 Phase 2 (Delivery)

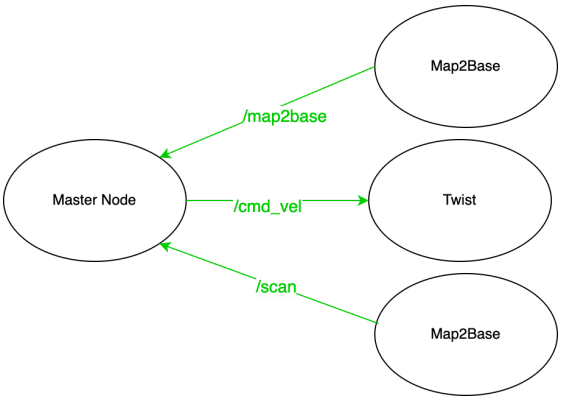
Objective of this phase

The TurtleBot is to autonomously navigate to the designated table in a fast and controlled manner while not damaging the can or making contact with any part of the restaurant. It will proceed to phase 3 upon arriving at the designated table.

Description of operation

The TurtleBot will read the list of stored waypoints for the designated table and use the coordinates data from the `/map2base` topic to navigate. This data from `/map2base` is generated from Cartographer and proves to be very accurate as it uses LDS and IMU to generate its location and coordinates. It will publish a Twist object through `/cmd_vel` to rotate towards the waypoint and move forward until it is within a distance threshold to the waypoint. This series of operations will be repeated for the rest of the set waypoints for this designated table and it will then proceed to stop and enter phase 3.

In the case of Table 6, upon arriving at the final waypoint and before entering phase 3, TurtleBot will use LDS data to detect Table 6 from the front of the TurtleBot as the TurtleBot will be facing the random zone. Once Table 6 is detected, the TurtleBot will rotate towards Table 6 using a Twist object and move forward until the front distance detected by the LDS is within a certain range.



5.3 Phase 3 (Collection)

Objective of this phase

The TurtleBot is to detect whether the customer has removed the can using the microswitch. It will then proceed to phase 4.

Description of operation

The microswitch signal will be constantly published by the `SwitchPublisher` node through the `/limit_switch` topic. The TurtleBot will be stuck in a loop until the message data from `/limit_switch` returns a False signal to denote that the can has been removed.

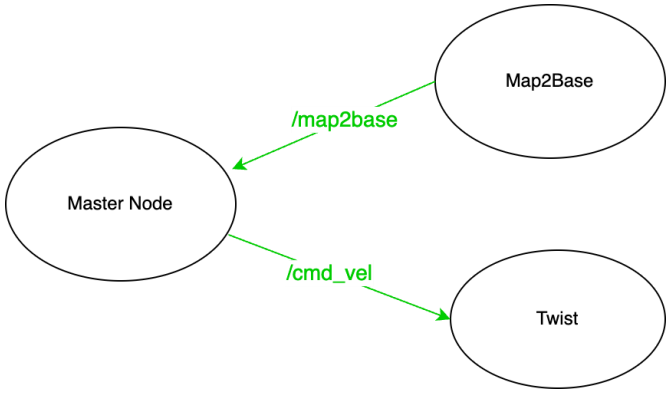
5.4 Phase 4 (Return)

Objective of this phase

The TurtleBot is to autonomously navigate back to the dispenser in a fast and controlled manner while not making contact with any part of the restaurant. The TurtleBot will then dock to allow receiving of another can. It will proceed back to phase 1 upon docking.

Description of operation

Similar to the delivery phase, the TurtleBot will reverse the list of stored waypoints for the designated table and use the coordinates data from the `/map2base` topic to navigate. It will publish a Twist object through `/cmd_vel` to rotate towards the waypoint and move forward until it is within a distance threshold to the waypoint. This series of operations will be repeated for the rest of the set waypoints. It will use the final waypoint to dock as the final waypoint will be set at the dispenser.



6. Prototyping & Testing

6.1 Navigation Algorithm

A waypoint navigation system was adopted for this specific mission. We faced 4 main problems during our initial navigation algorithm, namely sharing of waypoints, inaccuracy of turning, making large turns and lack of accuracy for docking.

Initially, a minimum number of waypoints were used as we were expecting to append the waypoints during the mission duration and adding additional waypoints would be time consuming therefore we opted for sharing of waypoints. However, after further testing with the `map2base` publisher, we realised that the waypoints could be stored beforehand as the `map2base` publisher would publish the same coordinates provided the TurtleBot starts at the same position each time. This meant that we were able to set specific waypoints for certain tables instead of tables sharing the same waypoints. This proved to be far more efficient as we were able to reduce the timing of table 2 from 1 minute 30 seconds to 1 minute, a reduction of 33%. We were able to achieve similar results for table 3, 4 and 5.

With regards to the inaccuracy of turning, we suffered from a small ripple effect where a small inaccuracy in angle turning could result in the TurtleBot being 5 centimetres off from the intended waypoint. As we could be deducted a mark for being more than 15 centimetres away from the table, accuracy of the TurtleBot cannot be compromised. To solve this problem, we have implemented a recalibration system where every certain distance travelled (set to be 55 centimetres in the mission) by the turtlebot, the TurtleBot will attempt to turn towards the waypoint again. This recalibration was very effective as the TurtleBot was never more than 3 cm from the set waypoint.

In the trial run, the TurtleBot faced a problem where it turned 350 degrees anticlockwise when the intended optimal turn was to turn 10 degrees clockwise. This was fixed by using complex numbers in the calculation of direction of turning.

6.2 Docking Algorithm

In our first iteration of the docking algorithm, we used a waypoint set at the dispenser as our form of docking. However, it was inaccurate as the TurtleBot was approaching the docking waypoint from different angles, making the can unable to be dispensed properly into the ‘frontpack’. To solve this problem, we made use of turning the TurtleBot to face the dispenser and using LDS to get the distance away from the dispenser to stop at. This proved to be more effective but it was still dependent on the accuracy of the `map2base` data.

6.3 ‘Frontpack’ Design

With limited understanding of the geometric dimensioning and tolerancing (GD&T), and the limitations of manufacturing processes, our first iteration of the ‘frontpack’ was very optimistic and unrealistic as rightly pointed out in the design review. It was an extremely complex geometry that 3-D printing would not be able to achieve. Moreover, the dimensions used were extremely accurate (to 2dp) and 3d printing would not be able to reach that level of precision which would then cause problems in mounting various components etc. As a result, this idea was quickly discarded.

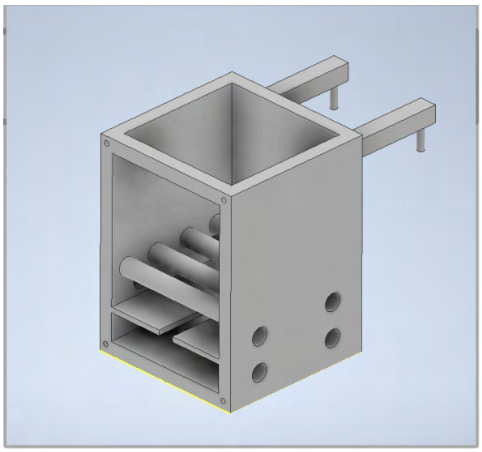


Figure 9. First iteration of ‘frontpack’

The second iteration of the ‘frontpack’ then came to life. The plan was that this would also be 3-D printed. The main idea behind this is that the arms would allow for the ‘frontpack’ to be secured onto the pre-existing holes within the second layer waffle plate. This was much simpler and looked like it could work. However, after doing stress analysis on CAD software, it came to light that the arms would not be able to support the weight of the ‘frontpack’ and the can. It would cause too much stress and the displacement would have resulted in the arms breaking. Taking this finding into consideration, we searched for other alternatives to secure the ‘frontpack’ onto the TurtleBot but we were unable to find any. The CAD model of this iteration of the ‘frontpack’ can be seen below

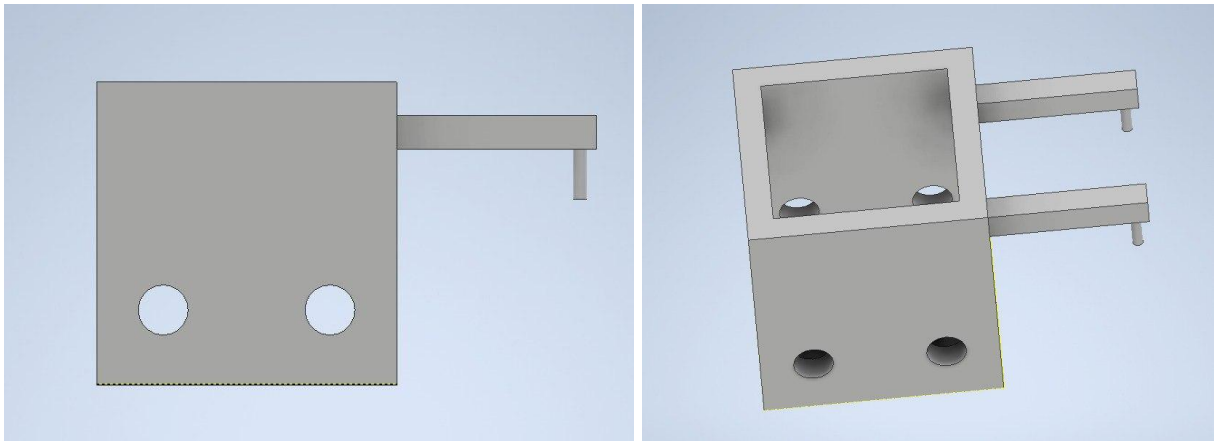


Figure 9. Second iteration of ‘frontpack’

It then struck us that we can use an extra waffle plate and take advantage of the TurtleBot's original design to secure the new waffle plate adjacent to the first level waffle plate. We used M3x35, M3x20 and M3x12 standoffs in order to secure the new waffle plate. The 'frontpack' itself could be made using corrugated cardboard available from the lab and secured onto this new waffle plate with the help of adhesives. This method was easier, cheaper and more convenient as we would not need to 3-D print this, so it wouldn't cut into the budget and printing this would have taken quite long.

Subsequently, minor changes that were made to the 'frontpack' included minor modifications of the shape from a cuboid to the top having small angled extrusions to resemble that of a funnel. This was done after rigorous testing and we found that the can did not always fall into the 'frontpack' once dispensed. Making these changes allowed us to circumvent this problem as the can was now able to fall into the 'frontpack' much more easily.

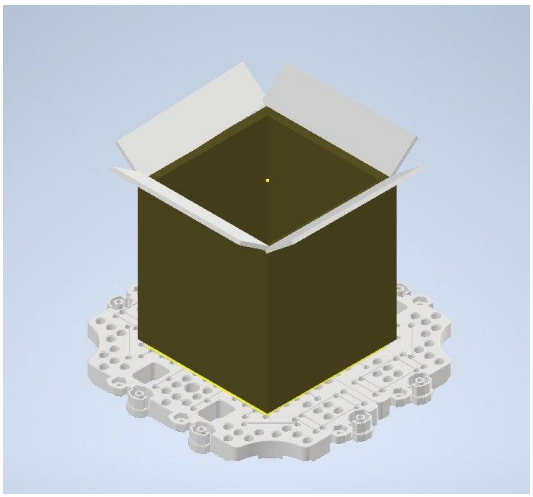


Figure 10. Final design of the 'frontpack'

6.4 Dispenser Design

Initially, the dispenser was designed in CAD as seen in the figure below and was planned to be 3-D printed. This design would require another laser cut flat piece that would be screwed onto the top of the dispenser which would support the can in place. There was also a hole in place dimensioned to meet that of MG995 so that it could be comfortably placed and the platform could be attached to it. The main mechanism is that the platform would slide to the side, allowing the can to drop directly below.

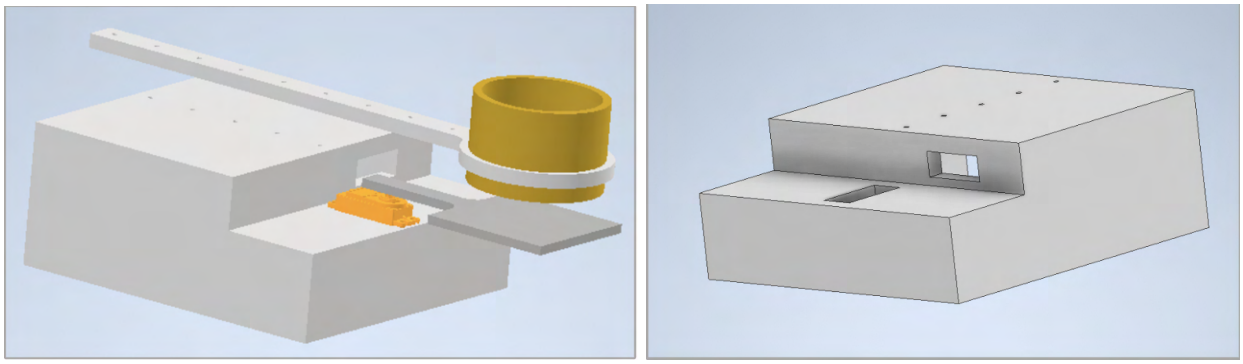


Figure 11. First iteration of Dispenser

As previously mentioned that 3-D printing a piece of this big size and to the accuracy to which it was designed is impractical. Moreover, another problem that was brought up is that the dispenser would not be stable and the vibrations from the servo would cause it to move out of place. As a result, we discarded the idea of 3-D printing and looked to using aluminium profile bars in order to build the frame of the dispenser. We could then have L corner brackets to screw on a top plate (for keypad, OLED and support for can) and a middle plate (with servo motor), both of which were laser cut from acrylic of 5mm. The underlying mechanism remained the same but the design was optimised. Another addition we made was a 3-D printed servo slot which would be screwed on to the middle plate.

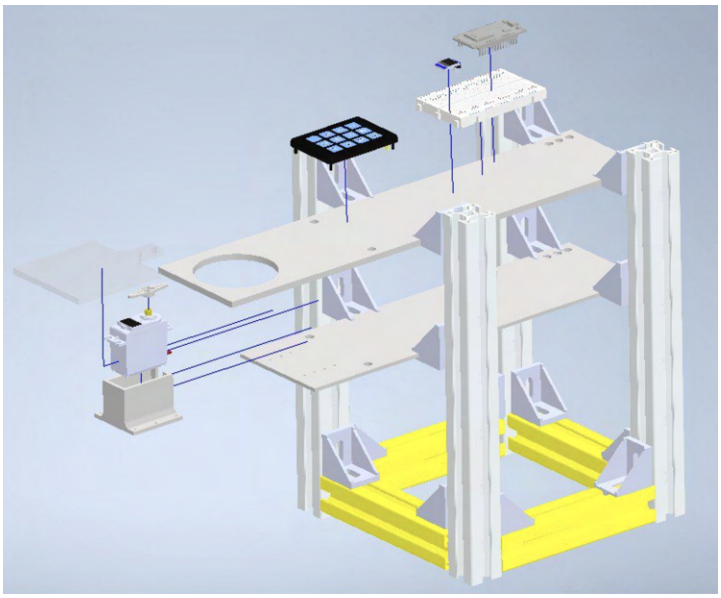


Figure 12. Final Design of Dispenser

6.5 Servo Slot

The servo slot was 3-D printed. During our testing, we realised that we had a minor design oversight. As the MG995 Servo Motor has a small extrusion from where the wires come out, it did not fit inside the 3-D printed slot, so a new slot with a slit was printed to accommodate for that part of the Servo Motor.

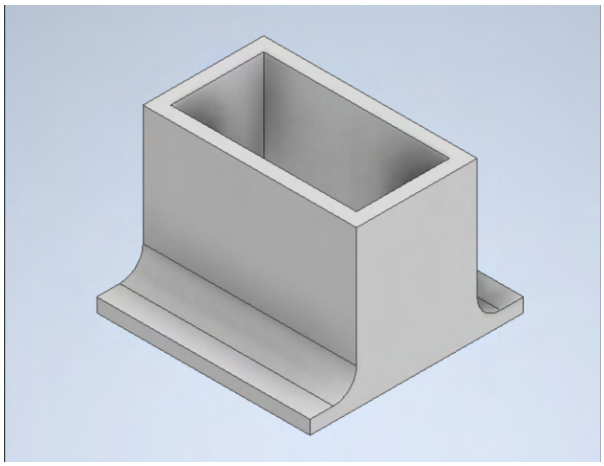


Figure 13. Initial design

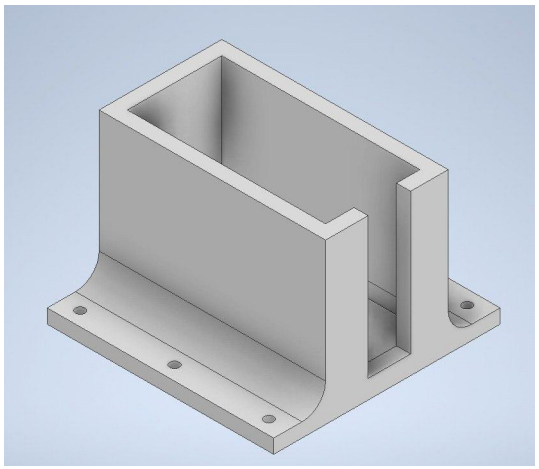


Figure 14. Final Design

6.6 Microswitch Location

Initially, we placed the microswitch on the bottom of the 'frontpack' meaning that everytime the can lands, it lands on the lever directly. During our testing phase, we realised that the microswitch could not handle the physical impact from repeated dropping of cans which could have ultimately led to it breaking. Therefore, to maintain the quality of the microswitch, we relocated the microswitch to the side of the 'frontpack' such that the force experienced by the lever is reduced and it lasts for a longer time while returning reliable readings. Furthermore, to prevent the microswitch from receiving the weight of the can dropping from the top, we attached a protective structure to the wall immediately on top of the microswitch as the following.

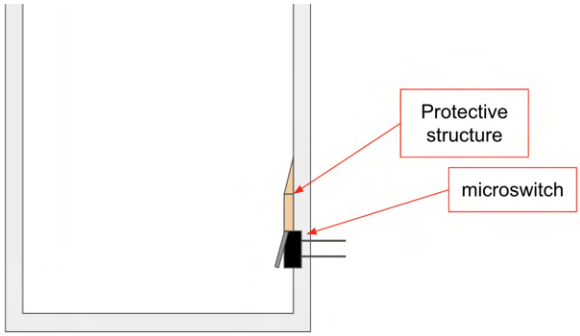


Figure 15. Diagram of microswitch placement

Figure 16. Photo of microswitch placement

Table 3. Microswitch mechanism

7. Final Design

7.1 Key Hardware Specifications

	List	Specifications	Notes
TurtleBot	Dimensions (mm) (LxWxH)	276mm x 178mm x 192mm	Full assembly (TurtleBot3 + Frontpack)
	Weight (kg)	Front Pack: 0.1296kg TurtleBot: 1kg Total weight: 1.1296kg	
	Wheel Base (mm)	65mm	From front of DYNAMIXEL to the back of ball caster
	FrontPack Dimensions (mm) (LxWxH)	80mm x 145mm x 80mm	
	Microswitch	1x D2F-01L	The microswitch should be attached to the interior side of the frontpack
Dispenser	Dimensions (mm) (LxWxH)	268mm x 210mm x 410mm	
	Weight (kg)	3.33kg	
	Servo Motor	1x MG995	
	Keypad	1x COM-08653	Input table number from 1-6 * = delete # = enter
	OLED screen	1x 0.96’’OLED	Reflect table number
	Micro-controller	1x ESP32	
	Power Plug	1x RPi power adaptor	Connect the ESP32 microcontroller to the wall socket for power
System	Battery Capacity	LiPo Battery 11.1V 1,800mAh	
	Expected Operating Time	2 hours 30 minutes	
	Communication Interface	I2C, MQTT, GPIO	

Table 4. Key Hardware specifications

7.2 System Finances

The bot was built using a TurtleBot3 Burger Set, created by ROBOTIS. The dispenser, on the other hand, was entirely built by the team. We had an allocated budget of S$100 which we used to buy a few additional components and to manufacture any specific parts that we had, all of which have been indicated below in the bill of materials.

Bill of Materials

No.	Part	Quantity	Unit Cost	Total Cost
TurtleBot3 Burger (Mechanical)				
1.	Waffle Plate	10	Provided	
2.	Wheel	2		
3.	Tire	2		
4.	Ball caster (w Ball)	2		
5.	PCB Support	12		
6.	Adapter Plate	1		
7.	L-Brackets	3		
8.	M2.5 Nuts	20		
9.	M3 Nuts	16		
10.	Spacer	4		
11.	Rivets (Short)	14		
12.	Rivets (Long)	2		
13.	M3x35 Plate Support	4		
14.	M3x45 Plate Support	10		
15.	M3x20 Standoff	1		
16.	M3x12 Standoff	1		
17.	PH_M2×4 mm K	8		
18.	PH_T2×6mm K	4		
19.	PH_M2.5x8mm K	16		
20.	PH_2.6×12mm K	16		

21.	PH_M2.5×16mm K	4		
22.	PH_M3×8mm_K	44		
23.	Zip Ties	2		
24.	Sponge	1	2.00	2.00
TurtleBot3 Burger (Electrical)				
25.	Li-Po Battery	1	Provided	
26.	Li-Po Battery Charger	1		
27.	USB Cable	2		
28.	DYNAMIXEL to OpenCR Cable	2		
29.	Raspberry Pi 3 Power Cable	1		
30.	Li-Po Battery Extension Cable	1		
31.	Dynamixel XL430	2		
TurtleBot3 Burger (Software)				
32.	OpenCR 1.0	1	Provided	
33.	Raspberry Pi 3	1		
34.	360 Laser Distance Sensor LDS-01	1		
35.	USB2LDS	1		
36.	D2F-01L Microswitch	1		
Dispenser (Mechanical)				
	Laser Cut Acrylic Plates			
1.	Middle Plate	1	Free of Cost	
2.	Top Plate	1		
3.	Can Platform	1	4.00	4.00
	3-D Printed Parts			
4.	Servo Slot	1	0.50	1.00
	Aluminium Profile Bars			
5.	30x30 Profile - 30cm	4	Free of Cost	

6.	30x30 Profile - 15cm	4		
7.	30x30 Profile - Corner brackets	16		
	Bolts/Nuts			
8.	M2x12 Bolts	6	Provided	
9.	M2 Hex Nuts	6		
10.	M6x20 Bolts	32		
11.	M6 Hex Nuts	32		
12.	M6 Rhombus Nuts	24		
13.	M3x8 Bolt	1		
Dispenser (Electrical)				
14.	Breadboard	1	Provided	
15.	MicroUSB Cable with Wall Plug	1		
Dispenser (Software)				
16.	MG995 Servo Motor	1	Provided	
17.	ESP32 Devkit V1	1		
18.	COM-08653 Numeric Keypad	1		
19.	0.96 inch OLED Display	1	4.05	4.05
Grand Total				11.05

Table 5. Bill of Materials

8. Assembly Instructions

8.1 Mechanical Assembly

8.1.1 TurtleBot3 Burger

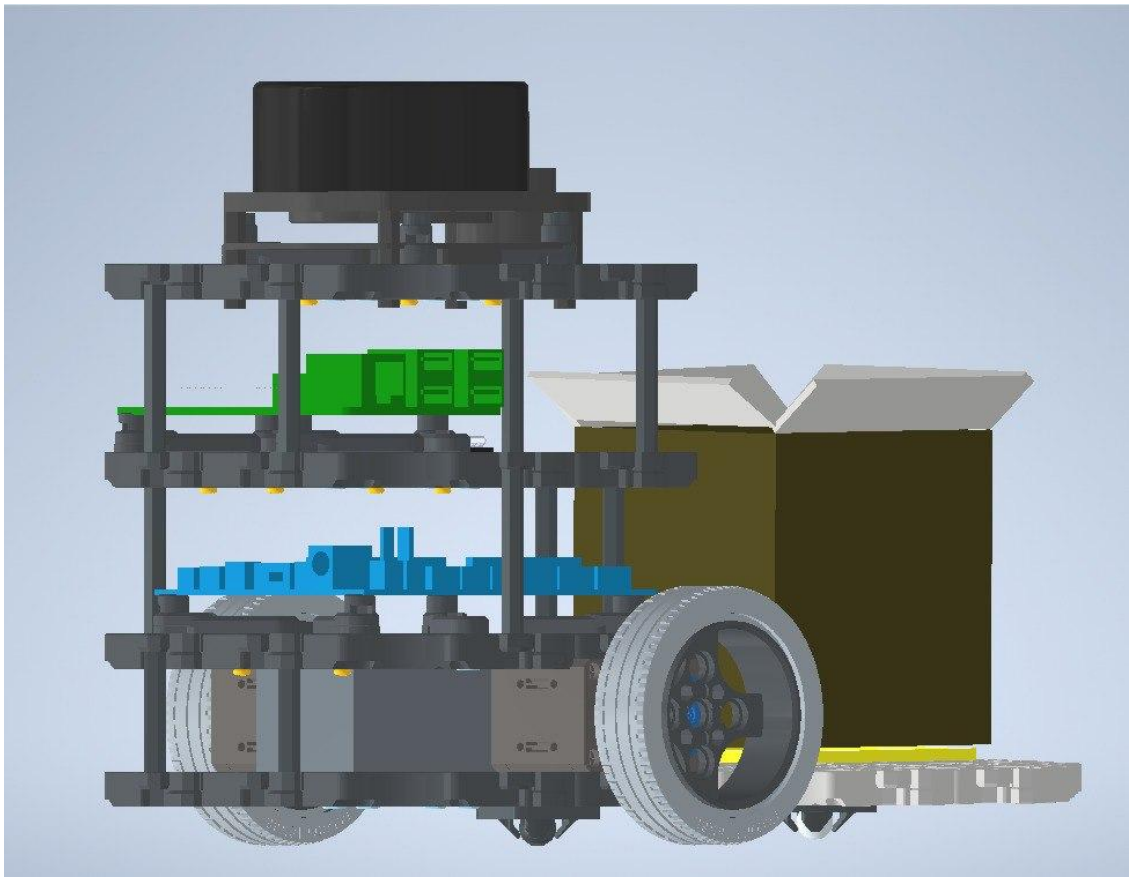


Figure 17. Overall Assembly of TurtleBot3 Burger

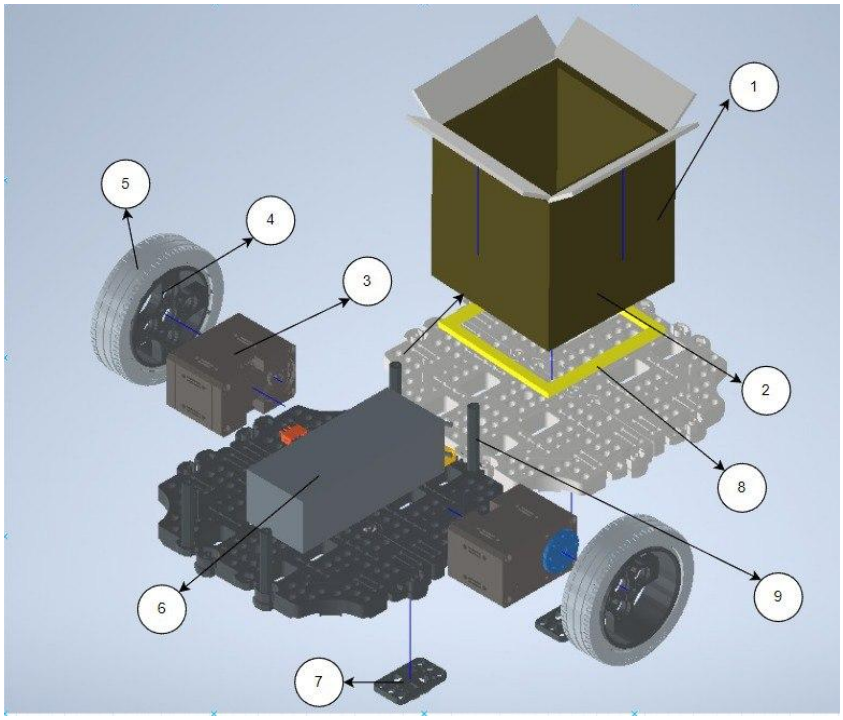


Figure 18. Layer 1 of TurtleBot3

No.	Part	Qty
1	D2F-01L	1
2	Frontpack	1
3	XL430 DYNAMIXEL	2
4	Wheel	2
5	Tire	2
6	Li-Po Battery	1
7	Ball Caster (w Ball)	2
8	Sponge	1
9	M3x35 Standoff	4

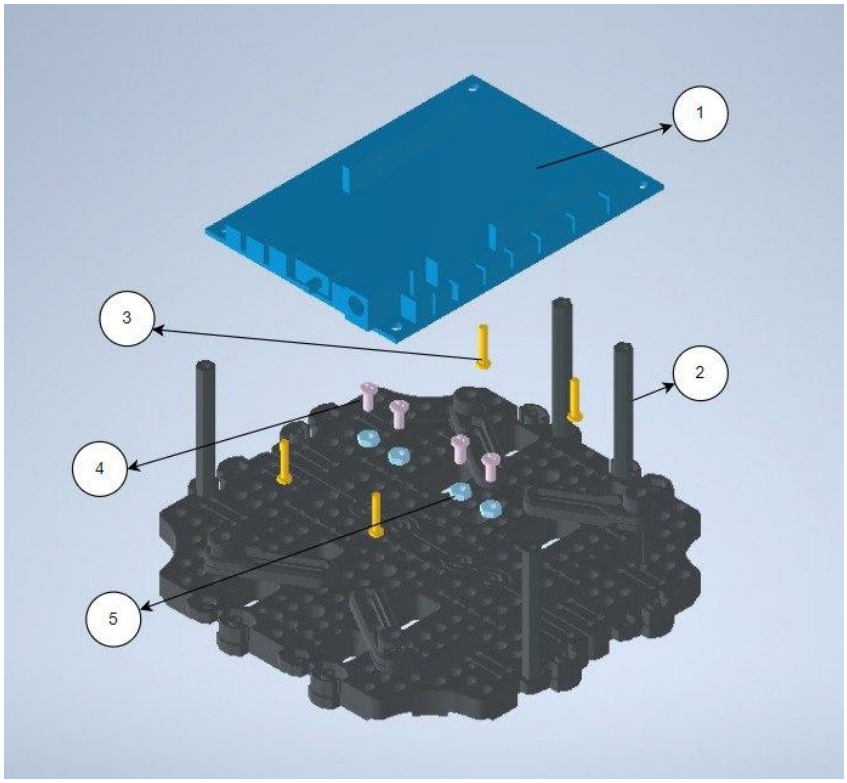


Figure 19. Layer 2 of TurtleBot3

No.	Part	Qty
1	OpenCR1.0	1
2	M3x45 Standoffs	4
3	PH_2.5x8_K	4
4	PH_3x8_K	4
5	M3 Hex Nuts	4

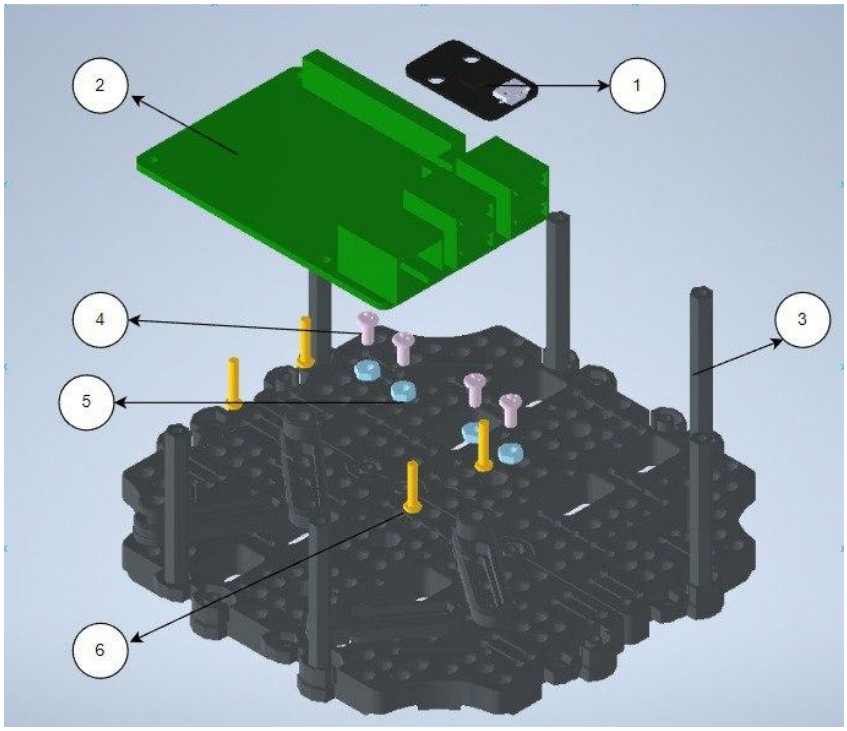


Figure 20. Layer 3 of TurtleBot3

No.	Part	Qty
1	USB2LDS	1
2	Raspberry Pi	1
3	M3x45 Standoffs	6
4	PH_3x8_K	4
5	M3 Hex Nuts	4
6	PH_2.5x8_K	4

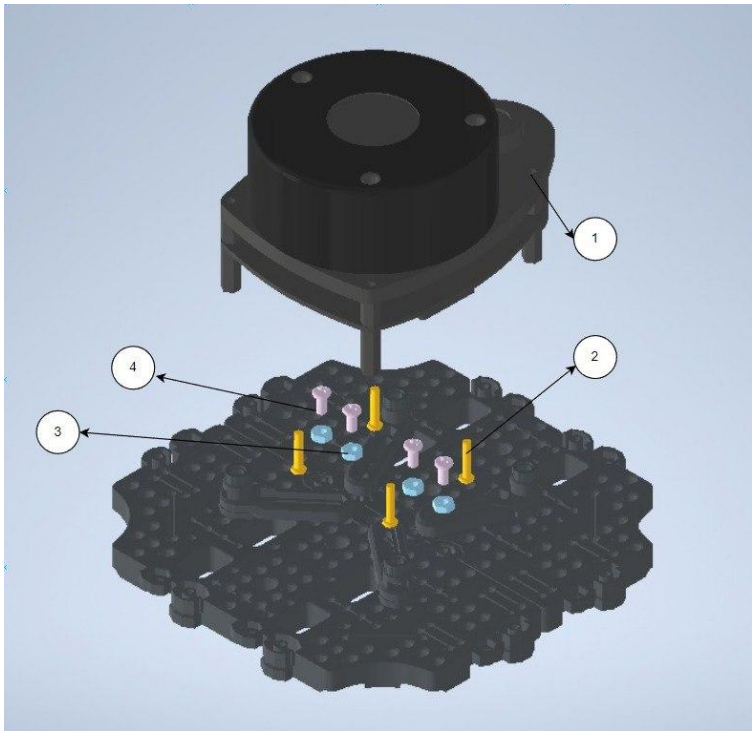


Figure 21. Layer 4 of TurtleBot3

No.	Part	Qty
1	LDS-01	1
2	PH_2.5x8_K	4
3	M3 Hex Nuts	4
4	PH_3x8_K	4

8.1.2 Dispenser

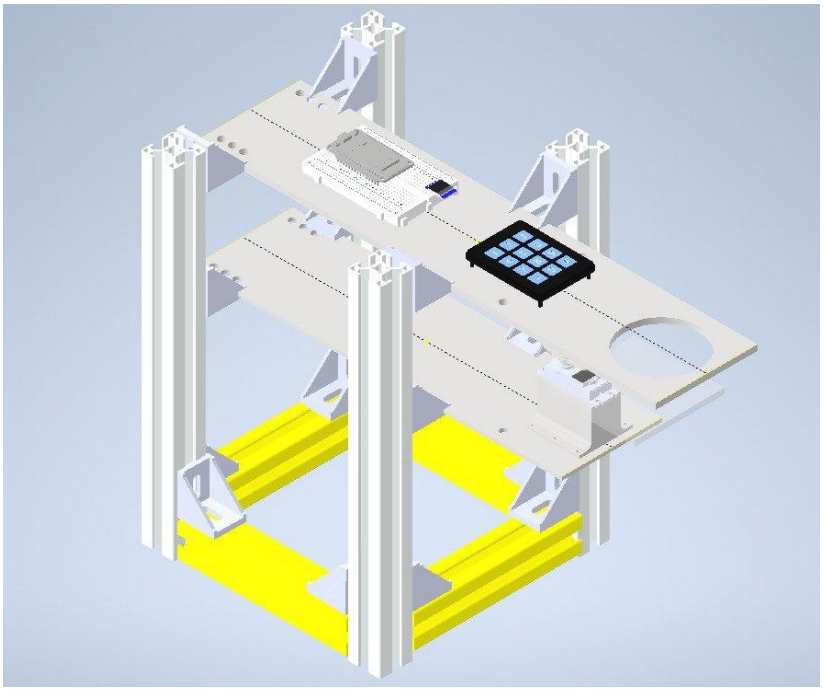


Figure 22. Overview Assembly of Dispenser

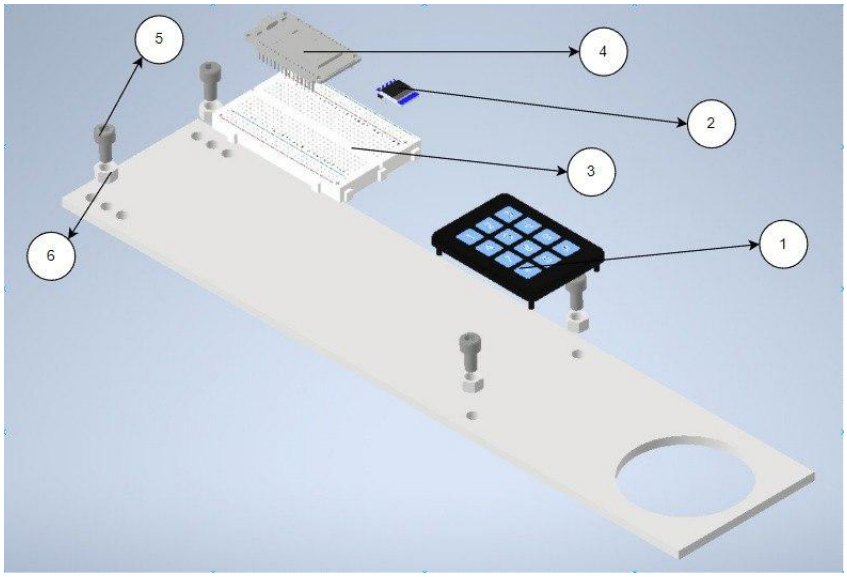
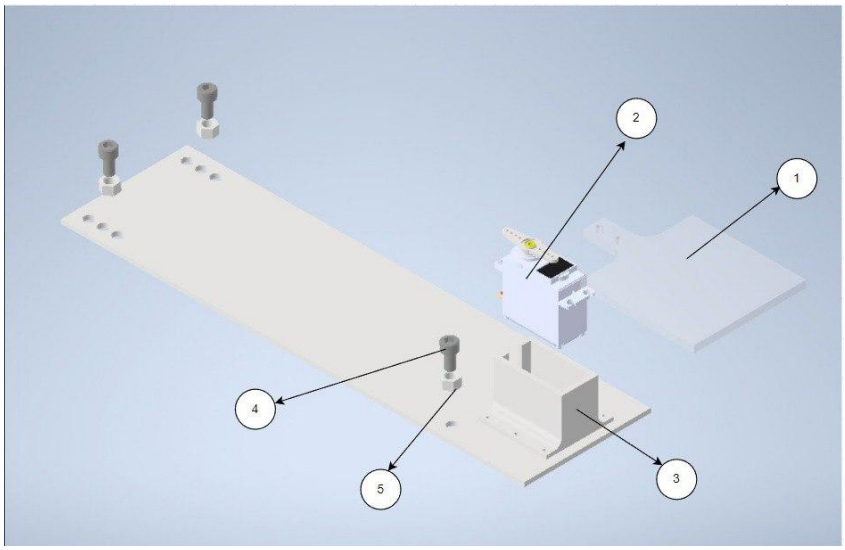


Figure 23. Top Plate of Dispenser

No.	Part	Qty
1	COM-08653	1
2	0.96 inch OLED	1
3	Breadboard	1
4	ESP32	1
5	M6x20	4
6	M6 Hex Nut	4



No.	Part	Qty
1	Platform	1
2	MG995 Servo Motor	1
3	Servo Slot	1
4	M6x20	4
5	M6 Hex Nuts	4

Figure 24. Middle Plate of Dispenser

8.2 Electronics Assembly

8.2.1 The Schematic diagram of the RPi and Microswitch

The diagram below shows the schematic of the RPi connected to the microswitch.

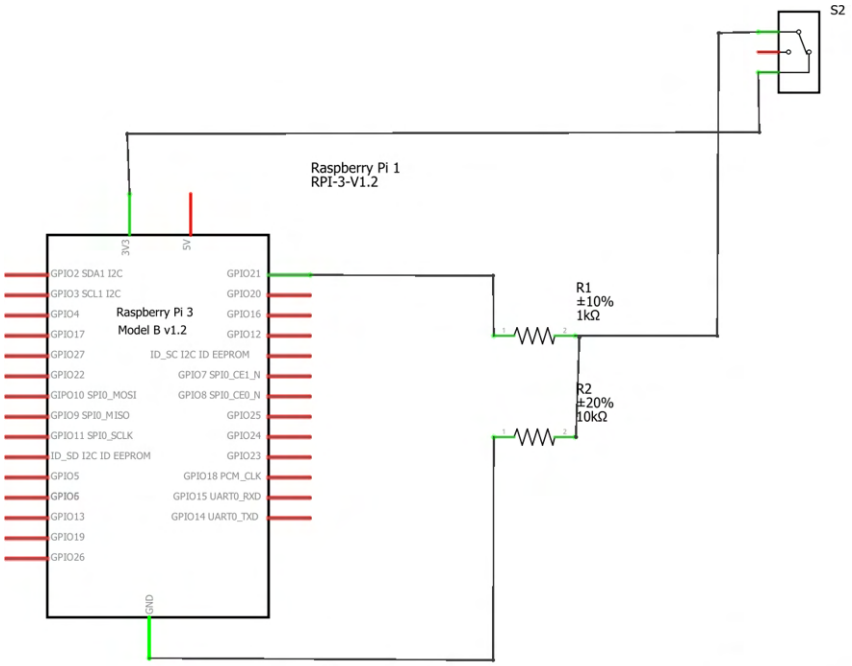


Figure 24. Schematic drawing of RPi and microswitch

The exact configuration of the switch connected to the GPIO pins is elaborated below, with the pin numbering convention used being the Broadcom SOC Channel (BCM).

The microswitch is connected to GPIO 21, with a current limit resistor of 1k ohm and a pull-down resistor of 10k ohm connected in parallel with the 3.3V pin of the RPi and the microswitch. The C pin of the microswitch is connected to the 3.3V pin of the RPi, while the NC pin is connected to the resistors in parallel. All the components are connected to the common ground on the RPi.

The OpenCR which is connected to the RPi provides the 5V and 3.3V that is supplied to the RPi and microswitch, and its ground is connected to the GND pin on the RPi.

8.2.2 The Schematic Diagram of the ESP32, OLED and servo

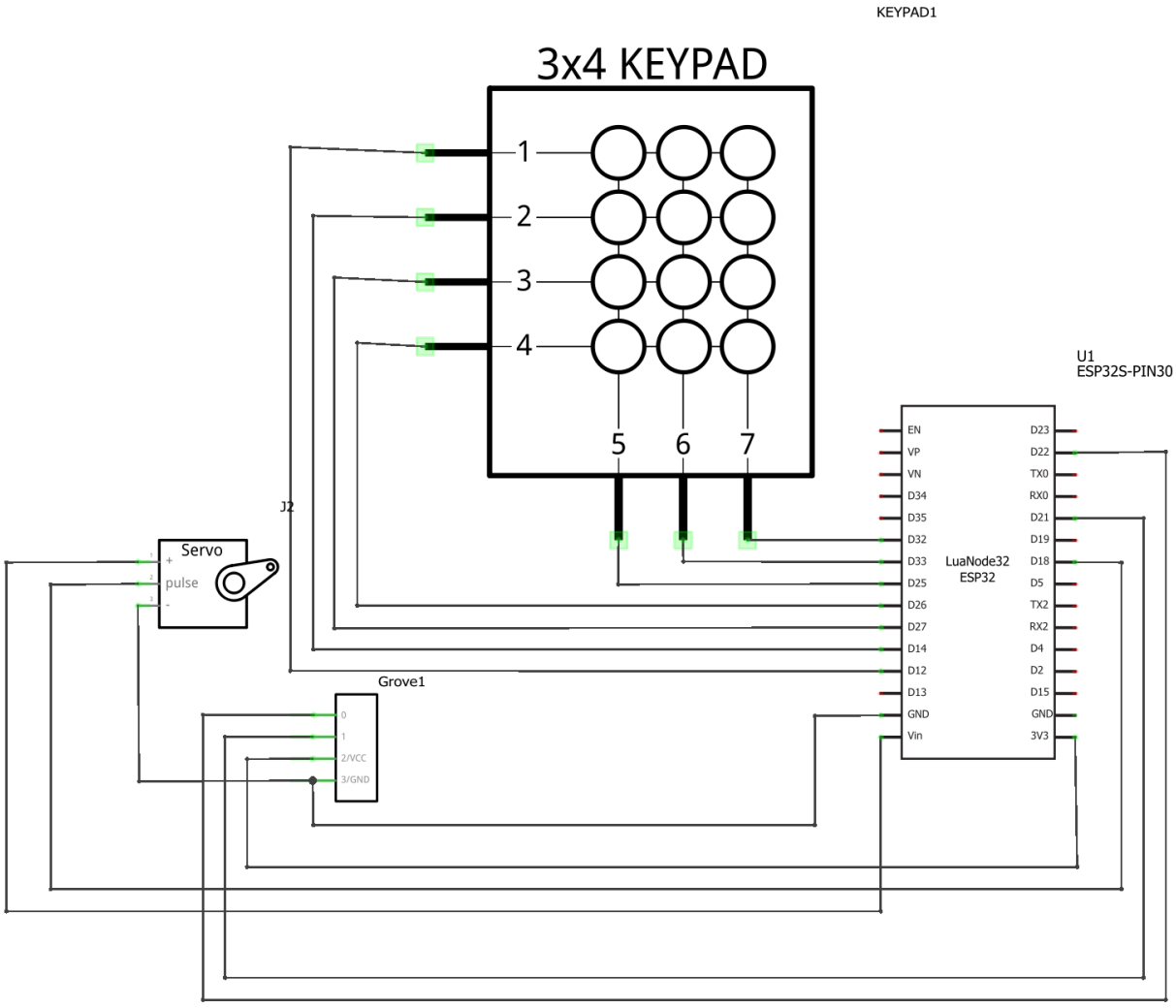


Figure 25. Schematic Drawing of connections in dispenser

The keypad is connected to the various GPIO pins of the ESP32, namely D32, D33, D25, D26, D27, D14 and D12. For the OLED, the Vcc pin is connected to the 3V3 pin of the ESP32 which has a 3.3V output. The SCL and SDA pins are connected to D21 and D22 of the ESP32 respectively. For the MG995 servo motor, the Vcc pin is connected to the Vin pin of the ESP32 which provides a 5.5V output, while the signal pin is connected to D18 of the ESP32. All the components are connected to the common ground, GND pin of the ESP32.

The ESP32 is powered by the RPi power adapter plugged into the wall via its micro USB connector.

8.3 Software Assembly

8.3.1 Setting up the Laptop

1. Prepare a laptop with Ubuntu 20.04 installed. Follow the following steps to install Ubuntu onto your laptop
    a. Follow this guide unless you are using a Macbook.
    b. Follow this guide if you are using a Macbook. Those with Intel Series Macbook can follow the guide too but must note the change in Ubuntu Image file that should be downloaded. (Non-ARM Ubuntu Image for Intel Series Macbook and ARM Ubuntu Image for M Series Macbooks).
        i. Ensure that you allocate at least 40-45 GB of storage to the Ubuntu Virtual Machine.

2. Install ROS2 Foxy by running the following commands in a terminal instance
    a. You can open a terminal instance by entering `Ctrl`+`Alt`+`T` in Ubuntu.

```
$ wget
https://raw.githubusercontent.com/ROBOTIS-GIT/robotis_tools/master/install_r
os2_foxy.sh
$ sudo chmod 755 ./install_ros2_foxy.sh
$ bash ./install_ros2_foxy.sh
```

3. Install Cartographer by running the following commands in a terminal instance.

```
$ sudo apt install ros-foxy-cartographer
$ sudo apt install ros-foxy-cartographer-ros
```

4. Install relevant TurtleBot3 packages by running the following commands in a terminal instance.

```
$ source ~/.bashrc
$ sudo apt install ros-foxy-dynamixel-sdk
$ sudo apt install ros-foxy-turtlebot3-msgs
$ sudo apt install ros-foxy-turtlebot3
```

5. Test the functionality of ROS2 Foxy on the laptop by running a simple publisher and subscriber node using this guide. You should be able to receive the results from the publisher node after running the subscriber node.

6. Edit the `~/.bashrc` file by running the command below and adding the 2nd code block into the `~/.bashrc` file.
    a. Replace {NUMBER} with a number that has not been used by the other ROS machines.This number should range from 0 to 101.

```
$ sudo nano ~/.bashrc
```

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

7. Edit the cartographer config.yaml file and change line 29 to the highlighted line.

```
$ cd /opt/ros/foxy/share/turtlebot3_cartographer/config/
$ sudo nano turtlebot3_lds_2d.lua
```

```
  provide_odom_frame = false,
  publish_frame_projected_to_2d = true,
  use_odometry = false,
  use_nav_sat = false,
  use_landmarks = false,
```

8.3.2 Setting up the RPi

1. Prepare a microSD Card and Reader (Only if your laptop does not have a microSD slot, please use a microSD card reader to burn the recovery image.)

2. Download the TurtleBot3 SBC Image here to be burned onto the microSD card.

3. Unzip the downloaded image file

4. Download Raspberry Pi Imager to allow burning of the image.

5. Open the Raspberry Pi Imager and burn the image
    a. Connect the microSD card to the laptop
    b. Choose OS > Select Custom > Select the extract TurtleBot3 .img file
    c. Press Write

6. Open a terminal instance and edit 50-cloud-init.yaml to edit the WiFi settings of the RPi
    a. Replace `WIFI_SSID` and `WIFI_PASSWORD` with the WiFi SSID and password.
    b. Save the file with `Ctrl`+`S` and `Ctrl`+`X`

```
$ cd /media/$USER/writable/etc/netplan
```

```
$ sudo nano 50-cloud-init.yaml
```

7. Access the RPi by connecting the RPi to a monitor and a keyboard or obtain the IP address of the RPi and entering the following command.

```
ssh ubuntu@{IP_ADDRESS}
```

8. Edit the `~/.bashrc` file similar to Step 6 of setting up the laptop and add the following lines.

```
export ROS_DOMAIN_ID={NUMBER}
export TURTLEBOT3_MODEL=burger
alias rosbu='ros2 launch turtlebot3_bringup robot.launch.py'
```

9. Test the functionality of ROS2 Foxy on the laptop by running a simple publisher and subscriber node using this guide. You should be able to receive the results from the publisher node after running the subscriber node.

10. Once both the RPi and the Ubuntu laptop are able to run the simple publisher and subscriber node, try running the publisher node on the RPi and run the subscriber node on the laptop. Ensure that the subscriber correctly receives the published information by the publisher node on the RPi. If any error is encountered, please go to the Troubleshooting section to resolve the issue.

11. Install MQTT on the RPi by running the following on an RPi terminal instance

```
sudo apt-add-repository ppa:mosquitto-dev/mosquitto-ppa
sudo apt-get update
sudo apt-get install mosquitto
sudo apt-get install mosquitto-clients
sudo apt clean
```

12. Edit the MQTT V2 Config File and add the first code block into the config file by following the second code block below.

```
# Add this to the config file
listener 1883
allow_anonymous true
```

```
# Run the following commands to edit
$ sudo cd /etc/mosquitto
$ sudo nano mosquitto.conf
```

8.3.3 Installing the Table Navigation program on the Laptop

1. Create a ROS2 package on the Ubuntu Laptop

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

2. Move the file in the directory temporarily to the parent directory.

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

3. Clone the GitHub repository to the ubuntu laptop. Ensure that the period at the end is included.

```
git clone git@github.com:wr1159/r2auto_nav.git .
```

4. Move the file __init__.py back.

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

5. Remove the unnecessary files in the workspace.

```
rm -rf switchpub.py
```

6. Build the ‘auto_nav’ and ‘custom_msgs' package on the laptop.

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

7. Add the following lines in the .bashrc file on the laptop for ease of use

```
alias tablenav='ros2 run auto_nav table_nav'
alias map2base='ros2 run auto_nav map2base'
alias r2waypoints='ros2 run auto_nav r2waypoints'
```

8.3.4 Installing the Table Navigation program on the RPi

1. Create a ROS 2 package on the RPi.

```
cd ~/turtlebot3_ws/src
ros2 pkg create --build-type ament_python hardware_bringup
cd hardware_bringup/hardware_bringup
```

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

```
git clone git@github.com:wr1159/r2auto_nav.git .
```

3. Remove all the files except for switchpub.py in the workspace.

```
rm -rf .git .gitattributes README.md ESP32 __init__.py package.xml
__pycache__ map2base.py setup.py r2table_nav.py r2waypoints.py
waypoints.json
```

4. Edit the package.xml using to insert the highlighted lines below

```
<export>
    <build_type>ament_python</build_type>
    <exec_depend>rclpy</exec_depend>
    <exec_depend>std_msgs</exec_depend>
</export>
```

5. Edit the setup.py to insert the highlighted lines below

```
entry_points={
        'console_scripts': [
                'switchpub = hardware_bringup.switchpub:main',
        ],
},
```

6. Build the 'hardware_bringup' package on the RPi.

```
cd ~/turtlebot3_ws && colcon build
```

7. Add the following line to the `~/.bashrc` file on the RPi.

```
alias switchpub='ros2 run hardware_bringup switchpub'
```

8.3.4 Dispenser Software Setup

1. Clone the GitHub repository to the laptop that is used to flash the ESP32. Ensure that the period at the end is included. **If the laptop used to flash the ESP32 is the same as the Ubuntu laptop, you may skip this step.**

```
git clone git@github.com:wr1159/r2auto_nav.git .
```

2. Install Arduino IDE

3. Install ESP32 Add-on into Arduino IDE
    a. Go to File > Preferences

    b. Enter the following into the "Additional Board Manager URLs" field and click OK.

```
https://raw.githubusercontent.com/espressif/arduino-esp32/gh-pages/package_e
sp32_index.json
```

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

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

4. Navigate to the cloned repository and open the ESP32_final_code inside ESP32 Folder either through a file explorer of your choice

5. Install the necessary libraries
    a. Adafruit GFX by Adafruit
    b. Adafruit SSD1306 by Adafruit
    c. ESP32Servo by Kevin Harrington, John K. Bennett
    d. Keypad by Mark Stanley, Alexander Brevig
    e. PubSubClient by Nick O’Leary

6. Edit the ssid, password variables to fit the the ssid and password of your WIFI or Hotspot that will be used in the restaurant.

7. Replace the broker_ip with the value shown by the RPi running the following command in the RPi. The RPi should be connected to a monitor and a keyboard for this to work. This broker_ip will not be changing if a new hotspot is solely used for the mission. A normal WiFi router should assign the same IP Address as well.

```
hostname -I
```

8. Flash the code into the ESP32 by connecting the ESP32 to the laptop with a micro usb cable and press the upload button located in the top left of Arduino IDE.

8.3.5 Calibration of Parameters

Starting from line 32 of r2table_nav.py, certain values could be tweaked to better fit the new restaurant.

```
# constants
DOCK_DISTANCE = 0.277 # distance until TurtleBot stops for docking
ROTATE_CHANGE = 0.4 # speed of rotation
SPEED_CHANGE = 0.175 # speed of movement
ANGLE_THRESHOLD = 0.8 # angle acceptance threshold
STOP_DISTANCE = 0.06 # distance stopping threshold
RECALIBRATE = 0.6 # distance travelled to recalibrate angle towards waypoint
FRONT_ANGLE = 23 # angles used in detecting table 6
```

r2table_nav.py	
Parameter	Description
`DOCK_DISTANCE`	The distance at which the LDS sensor detects directly in front of the turtlebot until turtlebot stops for docking. This could be modified to a lower value like 0.177 to allow replacement of can before the TurtleBot returns. However, in that case, the can must be present always when the TurtleBot returns or the TurtleBot will attempt to move forward even at the correct docking spot.
`ROTATE_CHANGE`	The speed at which the TurtleBot rotates. The higher the value, the faster and more inaccurate the turn will be. Suggested range of values are 0.2 - 0.6.
`SPEED_CHANGE`	The speed at which the TurtleBot moves The higher the value, the faster the TurtleBot will be moving at. Suggested range of values are 0.1 - 0.2.
`ANGLE_THRESHOLD`	The difference between the intended angle and the current TurtleBot yaw value. The angle threshold at which the TurtleBot will be able to accept before stopping the rotation. Suggested range of values are 0.8 - 1.5.
`STOP_DISTANCE`	The distance between the TurtleBot and the set waypoint at which the TurtleBot will stop. Lower values would lead to the TurtleBot being closer to the set waypoint. If the value is too low, it could result in the TurtleBot being unable to reach a distance to the set

	waypoint that is less than the set `STOP_DISTANCE`. Suggested range of values are 0.05 - 0.08
`RECALIBRATE`	The distance at which the TurtleBot travels until it rotates towards the target waypoint again. This is to help the TurtleBot be more accurate at reaching the waypoint. The lower the value, the more frequently the TurtleBot will recalibrate. Suggested range of values are 0.4 - 0.8
`FRONT_ANGLE`	The angle at which the TurtleBot uses to detect table 6, the table in the random zone. It will use the first `FRONT_ANGLE` values clockwise and anticlockwise starting from the front of the TurtleBot. This value should only be changed if the waypoint for table 6 has been altered. Suggested values are 20 - 40.

Table 6. Parameters and descriptions

8.4 Algorithm Overview

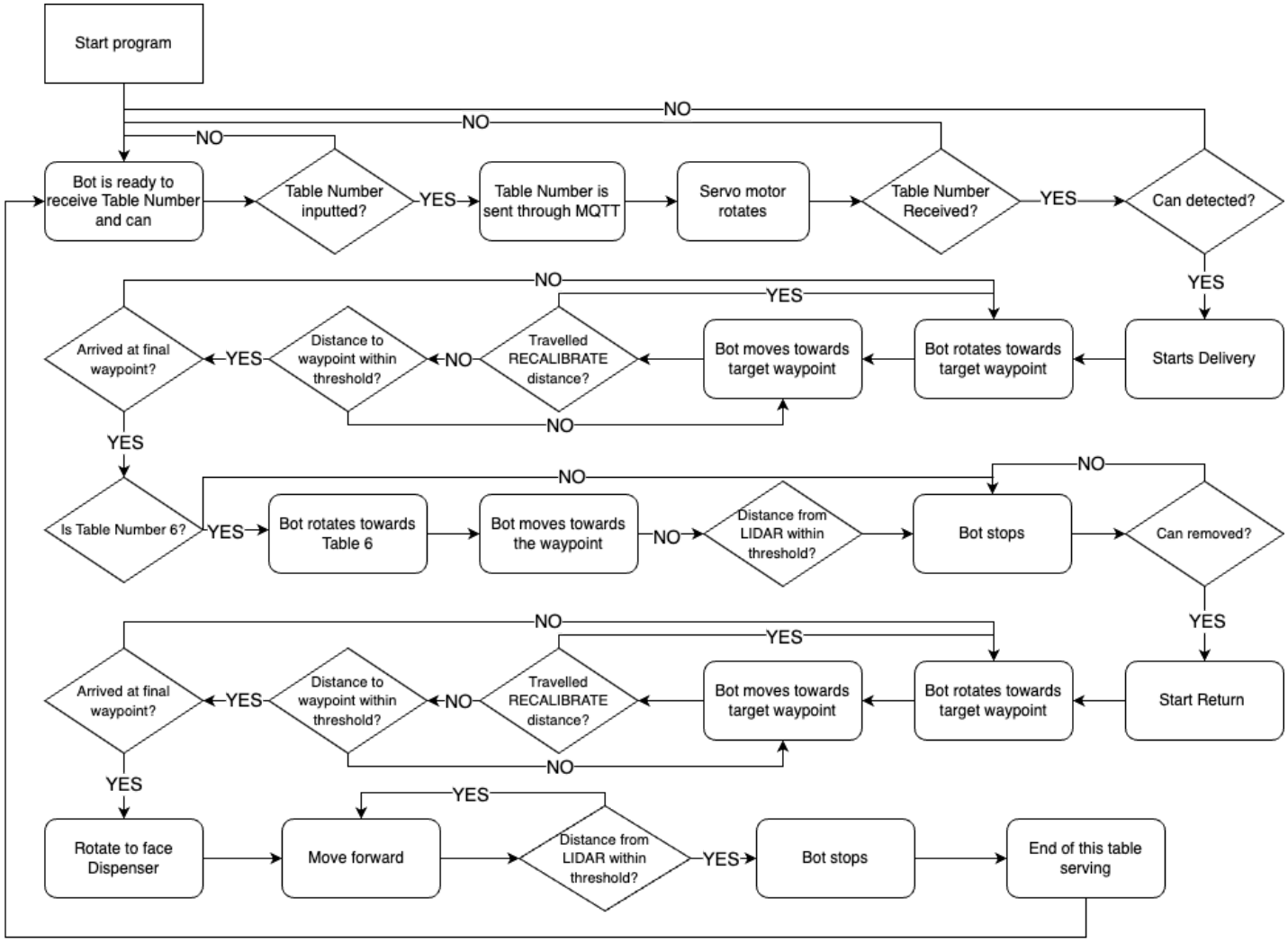


Figure 26. General Algorithm Flowchart

The algorithm flowchart shows the overview of the TableNav algorithm used. For the mission, the TurtleBot will start under the dispenser and be ready to receive the TableNumber through MQTT. Upon inputting the Table Number on the dispenser, it will be sent to the TurtleBot. The Servo will be activated and will rotate to drop the can. Once both the can is detected by the limit switch and the table number is received, the TurtleBot will start its delivery phase.

To deliver, the TurtleBot will first rotate towards the first of the many waypoints set for the certain table. It will then move towards the target waypoint, if at any moment, it has travelled a set distance of `RECALIBRATE`, it will rotate towards the target waypoint again to have better accuracy. Once the distance between the TurtleBot and the waypoint is within the threshold, it will repeat the process for the next waypoints. If table 6 was selected, the TurtleBot will turn towards table 6 by taking the shortest distance from the front 46 degrees. It will turn towards the shortest distance

in the front 46 degrees (23 degrees clockwise and 23 degrees anticlockwise from the front) and move until the detected distance in front by the LDS is within the threshold.

Upon arriving at the final waypoint, the TurtleBot will stop and will wait until the can has been removed which will be detected by the limit switch. Once the can has been removed, the TurtleBot will start its return back to the dispenser.

To return to the dispenser, the TurtleBot will navigate through the reversed list of waypoints set for that certain table. It will then rotate and move towards the waypoints similar to the navigation phase. After reaching the final waypoint, the TurtleBot will proceed to dock more accurately by turning to face the 0th degree, where it will then face the dispenser. It will then proceed to use the LDS values and move forward until the detected distance directly in front is less than the `DOCK_DISTANCE` value.

After returning to the dispenser, the TurtleBot is now available to receive a new can and table number.

8.4.1 System Communication

The system uses a publisher subscriber model for transmission and receiving of data. This comes in the forms of MQTT and ROS2 Topics. The diagram below shows the nodes and topics used by the `TableNav` program. The arrows represent the topics and show the direction of data transmission, from a publisher node to a subscriber. The `TableNav` node is the master node that subscribes to both ROS2 Topics and MQTT Topics.

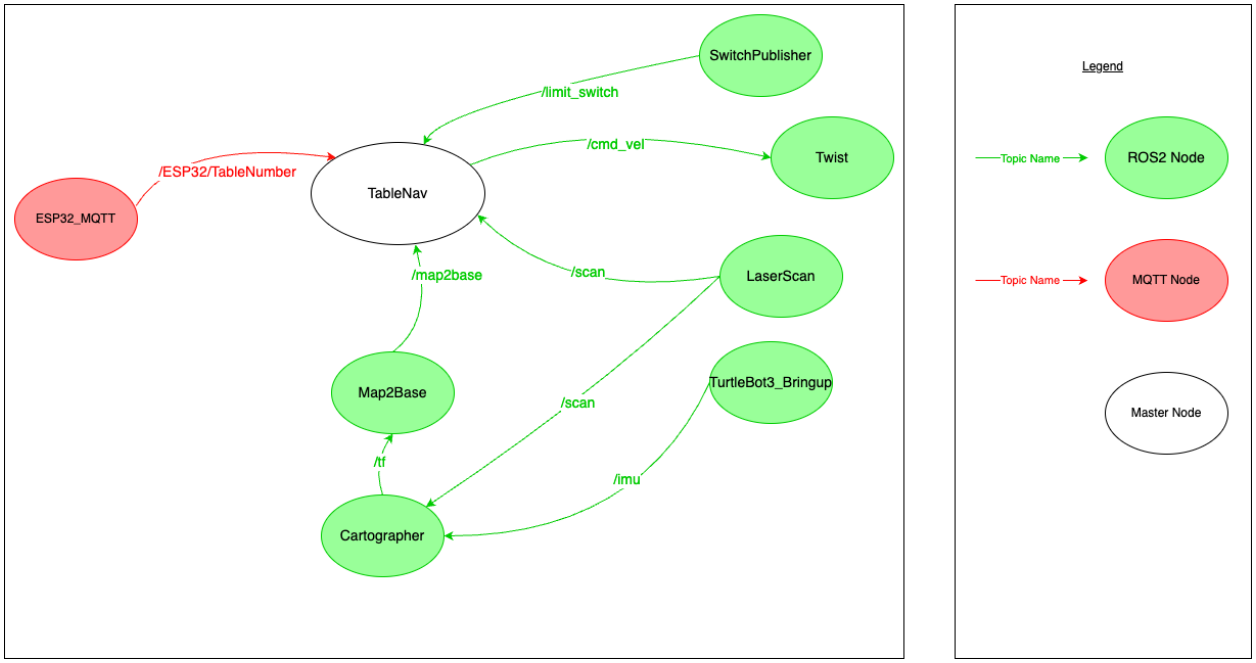


Figure 27. System Communication

The `ESP32_MQTT` node on the left runs on the dispenser's ESP32.

The `TableNav`, `Map2Base` and `Cartographer` nodes in the middle run on the laptop. `SwitchPublisher`, `Twist`, `LaserScan`, `TurtleBot3_Bringup` nodes on the right run on the TurtleBot's RPi.

To receive the table number, the `ESP32_MQTT` node publishes a string message in the topic `/ESP32/TableNumber` to TableNav through MQTT. This is the only MQTT topic used.

To dock and detect table 6, the data generated by the LDS is transmitted over as the topic /scan to both `TableNav`.

To navigate through waypoints, `/map2base` pose data will be used to locate the TurtleBot relative to the starting point. To obtain the `/map2base` pose data, `Map2Base` subscribes to cartographer for the TurtleBot transform data in the form of `/tf` topic. To obtain the `/tf` data, the Cartographer subscribes to both `/scan` from the `LaserScan` and `/imu`, data from the Inertial Measurement Unit published by `TurtleBot3_BringUp` node. Note that /odom has been excluded for the creation of `/tf` to obtain more consistent results.

To detect the presence of the can, the `SwitchPublisher` will publish the activation of the Switch as topic `/limit_switch` to `TableNav`.

All the code used in the system can be found on this GitHub repository.

8.4.2 Detailed Breakdown of Program on the Dispenser

ESP32.ino

```
#define Servo_PWM 18

const char* ssid       = "Saksham";
const char* password   = "saksham3004";
const char* broker_ip = "172.20.10.5";

/* instatiate wifi, mqtt clients */
WiFiClient wifi_client;
PubSubClient mqtt_client(wifi_client);
Servo MG995_Servo;

/*Setting up the oled*/
#define SCREEN_WIDTH 128 // OLED display width, in pixels
#define SCREEN_HEIGHT 64 // OLED display height, in pixels
#define OLED_RESET     -1 // Reset pin

/* Setting up the keypad */
const byte ROWS = 4; //four rows
const byte COLS = 3; //four columns

String input_num; //for input

char keys[ROWS][COLS] = {
  {'1', '2', '3'},
  {'4', '5', '6'},
  {'7', '8', '9'},
  {'*', '0', '#'}
};

/* For ESP32 Microcontroller */
byte pin_rows[ROWS] = {14, 32, 33, 26};
byte pin_column[COLS] = {27, 12, 25};

/*initialise oled*/
Adafruit_SSD1306 display = Adafruit_SSD1306(128, 64, &Wire, -1);

/*initialise keypad*/
Keypad keypad = Keypad(makeKeymap(keys), pin_rows, pin_column, ROWS, COLS); //
instantiating a keypad object

```

At the start of the program, the pin numbers and relevant constants such as WiFi SSID, password and Broker IP address are declared. The clients like the OLED, keypad, wifi client, MQTT client and the servo to be used are also initialised.

```
/* handle a reconnect to the broker if the connection is dropped */
void mqtt_reconnect(void){
    while(!mqtt_client.connected()){
        Serial.printf("Connecting to broker '%s'...", broker_ip);
        if(mqtt_client.connect("R&D #")){
            Serial.println("CONNECTED");
        }
        else{
            Serial.println("FAILED:");
            Serial.print(mqtt_client.state());
            Serial.println("\r\nRetrying in 5 seconds.");
            delay(5000);
        }
    }

}
```

The above snippet of code is the function to connect to the MQTT broker. It will attempt to connect to the broker until it is successful.

```
void Display(char key){
  Serial.println(key);
  display.setTextSize(3);
  display.setTextColor(WHITE);
  display.setCursor(0, 10);
  display.println(key);
  display.display();
}

void clearDisplay(){
  display.clearDisplay();
  display.setCursor(0,0);
  display.display();
}
```

The above snippet of code represents functions to display the inputted key onto the OLED and the function to clear the display respectively. This is where the appearance of the OLED could be tweaked to better suit the restaurant owner.

```
void setup() {
  Serial.begin(115200);

  Serial.println("OLED FeatherWing test");
  /* SSD1306_SWITCHCAPVCC = generate display voltage from 3.3V internally */
  display.begin(SSD1306_SWITCHCAPVCC, 0x3C); // Address 0x3C for 128x32

  Serial.println("OLED begun");

  /* Show image buffer on the display hardware. */
  /* Since the buffer is intialized with an Adafruit splashscreen */
  /* internally, this will display the splashscreen. */
  display.display();
  delay(1000);

  /* Clear the buffer. */
  display.clearDisplay();
  display.display();

  Serial.println("IO test");

  /* text display tests */
  display.setTextSize(4);
  display.setTextColor(SSD1306_WHITE);
  display.setCursor(0,0);
  display.display(); // actually display all of the above
```

The setup function is called by the ESP32 only once at the very beginning. In the above snippet, it is displaying the FeatherWing test and an Input/Output test to ensure that the OLED is indeed working.

```
// connect to WiFi
  Serial.printf("Connecting to '%s'...", ssid);

  WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED) {
        delay(500);
        Serial.print(".");
    }
    Serial.println(" CONNECTED");

  /* initialize (but don't connect yet) to the MQTT broker */
  mqtt_client.setServer(broker_ip, 1883);

  MG995_Servo.attach(Servo_PWM, 771, 2470);
  MG995_Servo.write(30);

}
```

The above snippet is a continuation of the setup function, this part of code completes the connection to the WiFi network using the WiFi client setup earlier. It will attempt to connect to the WiFi network until it successfully connects to the WiFi.

The `mqtt_client` will then be initialised with the Broker IP address and the port number. The Servo motor will be initialised using the relevant pin number, min and max value in microseconds. Those values are set specifically for MG995.

```
void loop() {
  char key = keypad.getKey(); //get table number
  Display(key);

  if(!mqtt_client.connected()){
        mqtt_reconnect();
  }

  /* handle/maintain the MQTT connection, including any new messages */
  mqtt_client.loop();
```

The loop function will be executed repeatedly after the setup function has been completed. The above snippet attempts to get the key from the keypad and displays it on the OLED using the display function explained earlier. It will then attempt to connect to the MQTT broker until it is successful. Once connected to the broker, it will maintain the connection to publish new messages.

```
  while (key){
    if (key == '*'){
      clearDisplay();
      input_num = ""; //clear table number
    }
    else if (key == '#'){
      Serial.print("Input number: ");
      Serial.println(input_num);

      //Sending topic to RPi
      mqtt_client.publish("ESP32/tableNumber", input_num.c_str());
      MG995_Servo.write(100);
      delay(5000);
      MG995_Servo.write(30);
      input_num = "";
      clearDisplay();
    }
    else{
      input_num = key;
    }
    key = keypad.getKey();
    Display(key);
  }

}
```

The above snippet is the logic for displaying the key, publishing the table number to the RPi and turning the servo motor. If the clear button (`'*'`), the previous table number will be cleared. If the confirm button (`'#'`) is pressed, the table number stored will be published as an MQTT message to the RPi. It will then rotate the servo 70 degrees and reset to its original position after 5 seconds, clearing the previous table number and the display.

8.4.3 Detailed Breakdown of Program on the Ubuntu Laptop

`map2base.py`

```python
import rclpy
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from geometry_msgs.msg import Pose
import numpy as np
from tf2_ros import TransformException

# Stores known frames and offers frame graph requests
from tf2_ros.buffer import Buffer

# Easy way to request and receive coordinate frame transform information
from tf2_ros.transform_listener import TransformListener
```

The above snippet imports the relevant libraries and message type to allow the frame transformation used later.

```python
class Map2Base(Node):

    def __init__(self):
        super().__init__('map2base')
        self.declare_parameter('target_frame', 'base_footprint')
        self.target_frame = self.get_parameter(
        'target_frame').get_parameter_value().string_value

        self.tf_buffer = Buffer()
        self.tf_listener = TransformListener(self.tf_buffer,self,
spin_thread=True)
        self.mapbase = None
        self.map2base = self.create_publisher(Pose, '/map2base', 10)
        timer_period = 0.05
        self.timer = self.create_timer(timer_period, self.timer_callback)
```

In the above snippet, the `Map2Base` node is defined. The publisher is created with the Pose message type and the topic of `/map2base`. The timer period is defined and every 0.05 seconds, the message will be published by the callback function explained below. Other relevant Transform objects are defined to be used in the callback function later. This is far more reliable than the `/map` topic which is published every 1 second.

```
16      def timer_callback(self):
17          # create numpy array
18          msg = Pose()
19          from_frame_rel = self.target_frame
20          to_frame_rel = 'map'
21          now = rclpy.time.Time()
22          try:
23              # while not self.tf_buffer.can_transform(to_frame_rel, from_frame_rel,
    now, timeout = Duration(seconds=1.0)):
24              self.mapbase = self.tf_buffer.lookup_transform(
25                          to_frame_rel,
26                          from_frame_rel,
27                          now)
28                          # ,
29                          # timeout = Duration(seconds=1.0))
30          except TransformException as ex:
31              self.get_logger().info(
32                  f'Could not transform {to_frame_rel} to {from_frame_rel}: {ex}')
33              return
34          msg.position.x = self.mapbase.transform.translation.x
35          msg.position.y = self.mapbase.transform.translation.y
36          msg.orientation = self.mapbase.transform.rotation
37
38          self.map2base.publish(msg)
39          self.get_logger().info('Publishing: "%s"' % msg)
40
```

This callback function publishes the pose data as a message after converting the Transform data from `/tf` topic into a pose data. In the case where it is unable to receive any Transform data, it will print a log message and return `None`. The pose message will contain the x and y coordinates of the TurtleBot alongside the orientation.

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

    map2base = Map2Base()

    rclpy.spin(map2base)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    map2base.destroy_node()
    rclpy.shutdown()


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

The above snippet is a driver code to initialise the ROS Client Library and to instantiate the `Map2Base` node. It will then start the `map2base` publisher using the spin function.

`r2waypoints.py`

```python
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from rclpy.qos import qos_profile_sensor_data
from geometry_msgs.msg import Pose
from nav_msgs.msg import OccupancyGrid
import math
import cmath
import numpy as np
import json

# constants
rotatechange = 0.1
speedchange = 0.05
```

At the start of the program, the relevant libraries are imported and speed constants are defined.

```python
# 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

```

This code converts the quaternion values which the TurtleBot orientation data provides into euler angles in python.

```python
# function to check if keyboard input is a number as
# isnumeric does not handle negative numbers
def isnumber(value):
    try:
        int(value)
        return True
    except ValueError:
        return False

```

A simple function to whether a value is a number. It is used later on in rotating the TurtleBot by a certain angle.

```python
# class for moving and rotating robot
class Mover(Node):
    def __init__(self):
        super().__init__('moverotate')
        self.publisher_ = self.create_publisher(Twist,'cmd_vel',10)
        # self.get_logger().info('Created publisher')

        self.map2base_sub = self.create_subscription(
            Pose,
            'map2base',
            self.map2base_callback,
            1)
        # self.get_logger().info('Created subscriber')
        self.map2base_sub # prevent unused variable warning
        # initialize variables
        self.roll = 0
        self.pitch = 0
        self.yaw = 0
        self.mapbase = []

    def map2base_callback(self, msg):
        # self.get_logger().info('In map2basecallback')

        self.mapbase = msg.position
        self.roll, self.pitch, self.yaw = euler_from_quaternion(msg.orientation.x,
msg.orientation.y, msg.orientation.z, msg.orientation.w)
```

In the above snippet, the moverotate node used for setting the waypoints is defined. It publishes a `Twist` object through the topic `/cmd_vel` to move the TurtleBot and subscribes to `/map2base` for the Pose of the TurtleBot. It initialises variables which will be overwritten in the callback function as it reassigns the variables like mapbase with the updated values of the TurtleBot.

```python
    def jsonAdd(self):
        wp_file = "waypoints.json"
        f = open(wp_file, 'r+')
        existing_waypoints = json.load(f)

        # print("existing waypoints", existing_waypoints)

        all_table_number = int(input("Enter table number: "))
        while (all_table_number > 0) :
            # check if key exist. Exist? Append:Create new entry
            table_number = str(all_table_number % 10)
            if (table_number not in existing_waypoints):
                print('not in existing_waypoints')
                existing_waypoints[table_number] = [{"x":self.mapbase.x, "y":self.mapbase.y}]
            else:
                existing_waypoints[table_number].append({"x":self.mapbase.x, "y":self.mapbase.y})
            all_table_number = all_table_number // 10

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

The `jsonAdd` function adds the current x and y coordinates of the TurtleBot into a json file called `waypoints.json`. It requires an input of table number and allows for inputs of multiple table numbers like `"123456"` as it will iterate through each digit to create a new table object with the current coordinates or append the coordinates to the assigned table. This will create a dictionary with the table number as key and list of coordinates as the value.

```python
# function to rotate the TurtleBot
    def rotatebot(self, rot_angle):
        # self.get_logger().info('In rotatebot')
        # create Twist object
        twist = Twist()

        # get current yaw angle
        current_yaw = self.yaw
        # log the info
        self.get_logger().info('Current: %f' % math.degrees(current_yaw))
        # we are going to use complex numbers to avoid problems when the angles go from
        # 360 to 0, or from -180 to 180
        c_yaw = complex(math.cos(current_yaw),math.sin(current_yaw))
        # calculate desired yaw
        target_yaw = current_yaw + math.radians(rot_angle)
        # convert to complex notation
        c_target_yaw = complex(math.cos(target_yaw),math.sin(target_yaw))
        self.get_logger().info('Desired: %f' % math.degrees(cmath.phase(c_target_yaw)))
        # divide the two complex numbers to get the change in direction
        c_change = c_target_yaw / c_yaw
        # get the sign of the imaginary component to figure out which way we have to turn
        c_change_dir = np.sign(c_change.imag)
        # set linear speed to zero so the TurtleBot rotates on the spot
        twist.linear.x = 0.0
        # set the direction to rotate
        twist.angular.z = c_change_dir * speedchange
        # start rotation
        self.publisher_.publish(twist)

        # we will use the c_dir_diff variable to see if we can stop rotating
        c_dir_diff = c_change_dir
        # self.get_logger().info('c_change_dir: %f c_dir_diff: %f' % (c_change_dir, c_dir_diff))
        # if the rotation direction was 1.0, then we will want to stop when the c_dir_diff
        # becomes -1.0, and vice versa
        while(c_change_dir * c_dir_diff > 0):
            # allow the callback functions to run
            rclpy.spin_once(self)
            current_yaw = self.yaw
            # convert the current yaw to complex form
            c_yaw = complex(math.cos(current_yaw),math.sin(current_yaw))
            self.get_logger().info('Current Yaw: %f' % math.degrees(current_yaw))
            # get difference in angle between current and target
            c_change = c_target_yaw / c_yaw
            # get the sign to see if we can stop
            c_dir_diff = np.sign(c_change.imag)
            # self.get_logger().info('c_change_dir: %f c_dir_diff: %f' % (c_change_dir, c_dir_diff))

        self.get_logger().info('End Yaw: %f' % math.degrees(current_yaw))
        # set the rotation speed to 0
        twist.angular.z = 0.0
        # stop the rotation
        self.publisher_.publish(twist)
```

The `rotatebot` function rotates the TurtleBot by the input angle in degrees. It uses complex numbers in deciding the direction of rotation. It will continue to calculate the direction of rotation towards the target goal, the moment the direction of rotation is now opposite, for example from clockwise to anti-clockwise, the TurtleBot will come to a stop.

```python
    def readKey(self):
        twist = Twist()
        try:
            while True:
                # use loop if using map topic
                # no need loop if using map2base
                rclpy.spin_once(self)
                # get keyboard input
                cmd_char = str(input("Keys w/x/a/d/p -/+int s: "))

                # 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:
                    # check which key was entered
                    # rclpy.spin_once(self)
                    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
                    elif cmd_char == 'x':
                        # move backward
                        twist.linear.x -= speedchange
                    elif cmd_char == 'a':
                        # turn counter-clockwise
                        twist.angular.z += rotatechange
                    elif cmd_char == 'd':
                        # turn clockwise
                        twist.angular.z -= rotatechange
                    elif cmd_char == 'p':
                        twist.linear.x = 0.0
                        twist.angular.z = 0.0
                        rclpy.spin_once(self)
                        self.jsonAdd()
                    # 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)
```

The `readKey` function will allow inputs of `"wasdx"` to move the TurtleBot similar to the `rteleop` command. It also allows the TurtleBot to be rotated by a certain angle using the rotatebot function. When the `'p'` is given, it will call the `jsonAdd` function and prompt the user to store the current table number to store the waypoint for. When the program is terminated, the TurtleBot will come to a stop.

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

    mover = Mover()
    mover.readKey()

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

    rclpy.shutdown()


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

The above snippet is a driver code to initialise the ROS Client Library and to instantiate the `SwitchPublisher` node. It will then start the publisher using the spin function.

`r2table_nav.py`

```python
import rclpy
from rclpy.node import Node
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
from geometry_msgs.msg import Pose
from std_msgs.msg import String, Bool
from rclpy.qos import qos_profile_sensor_data
from sensor_msgs.msg import LaserScan
from nav_msgs.msg import OccupancyGrid
import paho.mqtt.client as mqtt

import numpy as np
import math
import cmath
import time
import json

# constants
DOCK_DISTANCE = 0.277 # distance until TurtleBot stops for docking
ROTATE_CHANGE = 0.4 # speed of rotation
SPEED_CHANGE = 0.175 # speed of movement
ANGLE_THRESHOLD = 0.8 # angle acceptance threshold
STOP_DISTANCE = 0.06 # distance stopping threshold
RECALIBRATE = 0.6 # distance travelled to recalibrate angle towards waypoint
FRONT_ANGLE = 23 # angles used in detecting table 6, first 23 and last 23 degrees from the front of the TurtleBot
FRONT_ANGLES = range(-FRONT_ANGLE,FRONT_ANGLE+1,1) # first 23 degrees and last 23 degrees

WP_FILE = "waypoints.json"
F = open(WP_FILE, 'r+')
EXISTING_WAYPOINTS = json.load(F)
IP_ADDRESS = "172.20.10.5"


table_number = -1
```

At the start of the program, the relevant libraries are imported and the constants are set for the. The `waypoints.json` is also opened and extracted into a python dictionary for later use. The `table_number` variable that will be reassigned to the actual table number is defined here first.

```python
def euler_from_quaternion(x, y, z, w):
    ...
```

Same `euler_from_quaternion` function used in `r2waypoints.py`.

```python
class TableNav(Node):

    def __init__(self):
        super().__init__('table_nav')

        # create publisher for moving TurtleBot
        self.publisher_ = self.create_publisher(Twist,'cmd_vel',10)
        # self.get_logger().info('Created publisher')

        # initialize variables

        self.limitswitch_sub = self.create_subscription(
            Bool,
            'limit_switch',
            self.switch_callback,
            qos_profile_sensor_data)
        self.switch = False

        # create subscription for map2base
        self.map2base_sub = self.create_subscription(
            Pose,
            'map2base',
            self.map2base_callback,
            1)
        self.map2base_sub # prevent unused variable warning
        self.mapbase = {}
        self.roll = 0
        self.pitch = 0
        self.yaw = 0

        # create subscription to track lidar
        self.scan_subscription = self.create_subscription(
            LaserScan,
            'scan',
            self.scan_callback,
            qos_profile_sensor_data)
        self.scan_subscription  # prevent unused variable warning
        self.laser_range = np.array([])
```

In the above snippet, the `TableNav` node is defined. It publishes a `Twist` object through the topic `/cmd_vel` to move the TurtleBot and subscribes to `/map2base` for the `Pose` of the TurtleBot, `/limit_switch` for the activation of the limit_switch and `/scan` for the LDS data. It initialises variables which will be overwritten in the callback functions as they will reassign the variables like mapbase with the updated values of the TurtleBot.

```
40      def switch_callback(self, msg):
41          self.switch = msg.data
42
43      def scan_callback(self, msg):
44          # self.get_logger().info('In scan_callback')
45          # create numpy array
46          self.laser_range = np.array(msg.ranges)
47          # print to file
48          # np.savetxt(scanfile, self.laser_range)
49          # replace 0's with nan
50          self.laser_range[self.laser_range==0] = np.nan
51
52      def map2base_callback(self, msg):
53          # self.get_logger().info('In map2basecallback')
54
55          self.mapbase = msg.position
56          self.roll, self.pitch, self.yaw =
    euler_from_quaternion(msg.orientation.x, msg.orientation.y,
    msg.orientation.z, msg.orientation.w)
```

The callback functions are called by the subscribers to update the variables with the updated values from the `/limit_switch`, `/scan`, `/map2base` topics.

```
1  def rotatebot(self, rot_angle):
2      ...
```

Same `rotatebot` function used in `r2waypoints.py`.

```python
 # distance_to returns the distance between the turtlebot and the waypoint parameter using pythagoras theorem.
    def distance_to(self, goal):
        rclpy.spin_once(self)
        x_diff = goal['x'] - self.mapbase.x
        y_diff = goal['y'] - self.mapbase.y
        distance = math.sqrt(x_diff ** 2 + y_diff ** 2)
        self.get_logger().info('goal[x]: %f' % goal['x'])
        self.get_logger().info('mapbase.x: %f' % self.mapbase.x)
        self.get_logger().info('goal[y]: %f' % goal['y'])
        self.get_logger().info('mapbase.y: %f' % self.mapbase.y)
        self.get_logger().info('x_diff: %f' % x_diff)
        self.get_logger().info('y_diff: %f' % y_diff)
        self.get_logger().info('Current distance away: %f' % distance)
        if distance > 5:
            return self.distance_to(goal)
        return distance
```

The above `distance_to` function calculates the distance between the TurtleBot and the waypoint using the pythagoras theorem. In the case the distance calculated is > 5, it will call itself again as it is often a result of wrong `/map2base` data which happens occasionally.

```python
# angle_to calculuates the angle the turtlebot has to rotate to face the waypoint paramater
    def angle_to(self, goal):
        rclpy.spin_once(self)
        x_diff = goal['x'] - self.mapbase.x
        y_diff = goal['y'] - self.mapbase.y
        angle_diff = math.degrees(math.atan2(y_diff, x_diff))
        self.get_logger().info('Current angle: %f' % math.degrees(self.yaw))
        self.get_logger().info('Angle difference: %f' % angle_diff)
        return angle_diff
```

The above `angle_to` function calculates the angle the TurtleBot needs to rotate to be facing the waypoint using trigonometry, specifically the arctan function. A right angled triangle is created using the coordinates, which is why the arctan function can be used.

```python
# rotate_to_goal rotates towards the waypoint parameter
    def rotate_to_goal(self, goal):
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        self.publisher_.publish(twist)

        curr_angle_diff = self.angle_to(goal)
        diff = curr_angle_diff - math.degrees(self.yaw)
        if diff > 180:
            diff -= 360
        elif diff <= -180:
            diff += 360

        self.get_logger().info('curr diff : %f' % diff)

        if (diff > 0):
            twist.angular.z += ROTATE_CHANGE
        else:
            twist.angular.z -= ROTATE_CHANGE

        self.publisher_.publish(twist)
        min_angle = abs(math.degrees(self.yaw) - curr_angle_diff)
        break_flag = False
        while (abs(math.degrees(self.yaw) - curr_angle_diff) > ANGLE_THRESHOLD):
            self.get_logger().info('curr diff : %f' %  abs(math.degrees(self.yaw) - curr_angle_diff))
            if (abs(math.degrees(self.yaw) - curr_angle_diff) < 10 and abs(twist.angular.z) == ROTATE_CHANGE):
                twist.angular.z = twist.angular.z * 0.25
            min_angle = min(abs(math.degrees(self.yaw) - curr_angle_diff), min_angle)
            if (min_angle < abs(math.degrees(self.yaw) - curr_angle_diff)):
                break_flag = True
                break
            self.publisher_.publish(twist)
            rclpy.spin_once(self)
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
        if (break_flag):
            self.rotate_to_goal(goal)
```

The `rotate_to_goal` function takes in a target waypoint and will rotate towards the target waypoint. It first gets the difference ein the angle, the angle the TurtleBot needs to rotate to be facing the target waypoint and reduces it by 360 or increase the value by 360 to ensure the TurtleBot does not rotate by more than 360 degrees in the actual turn.

It will then determine the direction of turn based on the difference. It stores the minimum angle difference to the target waypoint as the function should aim to minimise the angle facing the minimum angle. While the difference in angle towards the target waypoint is greater than the acceptable `ANGLE_THRESHOLD`, it will check if the angle difference is less than 10 degrees. If it is, it will slow down to a quarter of the `ROTATE_CHANGE` rotate speed and update the minimum angle difference. If in any case, the `minimum_angle_difference` variable is not decreasing anymore, the function will call itself to get a more precise turn towards the waypoint.

This readjusting recursive call of the function will ensure an accurate direction towards the waypoint.

```python
# rotate_to is similar to rotatebot but with higher accuracy which makes it slower
    def rotate_to(self, degree):
        twist = Twist()
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
        degree = float(degree)
        desired_position = math.degrees(self.yaw) + degree
        if (desired_position > 180):
            desired_position -= 360
        elif (desired_position < -180):
            desired_position += 360

        # self.get_logger().info('Current angle diff: %f' % curr_angle_diff)

        if (degree > 0):
            twist.angular.z += ROTATE_CHANGE
        else:
            twist.angular.z -= ROTATE_CHANGE
        self.publisher_.publish(twist)

        self.get_logger().info('degree: %s' % str(degree))
        while (abs(desired_position - math.degrees(self.yaw)) > ANGLE_THRESHOLD):
            if(abs(desired_position - math.degrees(self.yaw)) <= 10 and twist.angular.z == ROTATE_CHANGE):
                twist.angular.z = 0.25 * twist.angular.z
                self.publisher_.publish(twist)
            self.get_logger().info('desired - yaw: %s' % str(abs(desired_position - math.degrees(self.yaw))))
            rclpy.spin_once(self)

```

The `rotate_to` function takes in a degree instead of a waypoint and it is similar to the `rotate_to_goal` function where. It performs the same 360 decrease or increase to determine the turning direction. Similarly, it will also reduce its rotation speed when the angle difference is less than 10 to achieve a more accurate turn

```
    def moveToTable(self, table_number):
        twist = Twist()
        table_number = str(table_number)
        twist.linear.x = -SPEED_CHANGE
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
        time.sleep(0.3)
        twist.linear.x = 0.0
        self.publisher_.publish(twist)
```

The `moveToTable` function is the main function for phase 2 of `delivery` to the table. It will take in a table number as an input parameter. It will move backwards for 0.3 seconds to allow for smoother turning later on.

```
        for index, waypoint in enumerate(EXISTING_WAYPOINTS[table_number][1::]):
            rclpy.spin_once(self)
            self.get_logger().info('Current waypoint target: %d' % index)
            self.rotate_to_goal(waypoint)
            rclpy.spin_once(self)
            twist.linear.x = SPEED_CHANGE
            twist.angular.z = 0.0
            self.publisher_.publish(twist)
            self.waypointDistance = self.distance_to(waypoint)
            print('published twist')

            travelled = 0
            while (self.waypointDistance > STOP_DISTANCE) :
                # Slow down when near waypoint
                if(self.waypointDistance <= 2.5 * STOP_DISTANCE and twist.linear.x == SPEED_CHANGE):
                    twist.linear.x = 0.25 * SPEED_CHANGE
                twist.angular.z = 0.0
                self.publisher_.publish(twist)
                distance = self.distance_to(waypoint)
                diff = abs(self.waypointDistance - distance)
                self.waypointDistance = distance
                travelled += diff
                self.get_logger().info('distance: %s' % str(self.waypointDistance))
                self.get_logger().info('travelled: %s' % str(travelled))
                if (travelled > RECALIBRATE):
                    self.rotatebot(self.angle_to(waypoint) - math.degrees(self.yaw))
                    travelled = 0

```

It will then iterate through the list of waypoints for that certain table and perform the following operations. First, it will rotate towards the waypoint. Next, it will proceed to move towards the target waypoint. It will continue to move towards the waypoint until the distance between the TurtleBot and the waypoint is less than the `STOP_DISTANCE` threshold. If at any moment the distance between the TurtleBot and the waypoint is less than 2.5 times of `STOP_DISTANCE`, the TurtleBot will slow down itself to ensure it does not overshoot the waypoint.

There is a travelled variable which will add the difference in the previous distance between and the new distance between after a moment. If at any moment the distance travelled exceeds `RECALIBRATE`, the TurtleBot will turn towards the waypoint using the `rotatebot` function instead of `rotate_to_goal` as `rotate_to_goal` is more accurate but slower.

Upon reaching the current waypoint, it will repeat the same operations for the next waypoint in the list until it reaches the last waypoint in the list.

```
        if(table_number == "6"):
            self.rotate_to(270 - math.degrees(self.yaw))
            self.get_logger().info('initialising table 6')
            min_distance= 10000;
            min_degree = 360
            for angle in FRONT_ANGLES:
                self.get_logger().info('%d, %f' % (angle, self.laser_range[angle]))
                if(self.laser_range[angle] < min_distance):
                    min_degree = angle
                    min_distance = self.laser_range[angle]
            self.rotate_to(min_degree)
            twist.linear.x = SPEED_CHANGE
            twist.angular.z = 0.0
            self.publisher_.publish(twist)
            while (self.laser_range[0] > DOCK_DISTANCE + 0.02 or math.isnan(self.laser_range[0])):
                rclpy.spin_once(self)
                if(self.laser_range[0] <= 2.5 * DOCK_DISTANCE and twist.linear.x == SPEED_CHANGE):
                    twist.linear.x = 0.25 * SPEED_CHANGE
                    self.publisher_.publish(twist)
                self.get_logger().info('laser_range[0]: %s' % str(self.laser_range[0]))
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
```

The above snippet is the operation for table 6. It will first use the `rotate_to` function to rotate towards the centre of the random zone. It will then loop through the `FRONT_ANGLES` (the clockwise 23 angles and anticlockwise 23 angles from the front) to get the minimum distance. After it has obtained the minimum distance, the rotate_to function will be called to rotate towards table 6.

The TurtleBot will move towards table 6 until the front distance detected by the LDS is less than the `DOCK_DISTANCE` + 0.02. The `math.isnan` function is to ensure that LDS is not returning a `NaN` value which it might occasionally. If the front distance detected by the LDS is less than 2.5 times of the `DOCK_DISTANCE`, the TurtleBot will slow down.

Once the LDS returns a value less than `DOCK_DISTANCE` + 0.02, the TurtleBot will come to a stop.

```
    def returnFromTable(self, table_number):
        twist = Twist()
        table_number = str(table_number)
        global STOP_DISTANCE
        for index, waypoint in enumerate(EXISTING_WAYPOINTS[table_number][-2::-1]):

            rclpy.spin_once(self)
            self.get_logger().info('Current waypoint target: %d' % index)
            self.rotatebot(self.angle_to(waypoint) - math.degrees(self.yaw))

            #self.rotate_to_goal(waypoint)
            rclpy.spin_once(self)
            self.waypointDistance = self.distance_to(waypoint)
            twist.linear.x = SPEED_CHANGE
            twist.angular.z = 0.0
            self.publisher_.publish(twist)
            print('published twist')
                        travelled = 0

            while (self.waypointDistance > STOP_DISTANCE) :
                # Slow down when near waypoint
                if(self.waypointDistance <= 2.5 * STOP_DISTANCE and twist.linear.x == SPEED_CHANGE):
                    twist.linear.x = 0.25 * SPEED_CHANGE
                twist.angular.z = 0.0
                self.publisher_.publish(twist)
                distance = self.distance_to(waypoint)
                diff = abs(self.waypointDistance - distance)
                self.waypointDistance = distance
                travelled += diff
                self.get_logger().info('travelled: %s' % str(travelled))
                if (travelled > RECALIBRATE):

                    self.rotatebot(self.angle_to(waypoint) - math.degrees(self.yaw))
                    #self.rotate_to_goal(waypoint)
                    travelled = 0
```

The `returnFromTable` function is the main function for phase 4 of `return`. It will take in a table number as an input parameter. Similar to the `moveToTable` function, it will iterate through the list of the waypoints for that certain table. However, in this case, the original final waypoint is skipped and the list of waypoints is reversed.

It will rotate towards the waypoint and move towards the waypoint. It will perform the same slow down operation when the distance to waypoint is less than 2.5 times of `STOP_DISTANCE` and will also perform the recalibration rotation every time travelled exceeds `RECALIBRATE`.

```
        # set orentation to face 0 degrees and move forward until laser_range[0] hits dockdistance
        self.rotate_to(-math.degrees(self.yaw))
        twist.linear.x = 0.022
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
        dock_count = 0
        while (dock_count < 10):
            if(self.laser_range[0] >= DOCK_DISTANCE or math.isnan(self.laser_range[0])):

                self.get_logger().info('docking')
                rclpy.spin_once(self)
            else:
                self.get_logger().info('laser_range[0]: %s' % str(self.laser_range[0]))
                self.get_logger().info('dock count: %s' % str(dock_count))
                dock_count += 1
        twist.linear.x = 0.0
        twist.angular.z = 0.0
        self.publisher_.publish(twist)
```

Upon reaching the final waypoint of the reversed list, (or the original first waypoint set), the TurtleBot will rotate to face the 0 degree and move forward until the front distance detected by the LDS is less than the `DOCK_DISTANCE`.

There is an implementation of `dock_count` in case of inconsistent spikes of LDS distances where the TurtleBot will incorrectly stop even when the front distance detected by LDS actually exceeds the `DOCK_DISTANCE`. This will make the docking far more consistent.

```python
    def move(self):
        twist = Twist()
        try:
            while True:
                rclpy.spin_once(self)
                '''
                #isolated navigation testing without microswitch and mqtt
                table_number = input("Enter table number: ")
                self.moveToTable(table_number)
                self.returnFromTable(table_number)
                '''
                global table_number

                self.get_logger().info("self.switch state: %s" % str(self.switch))
                self.get_logger().info("table_number: %s" % str(table_number))
                if (table_number != -1 and self.switch):
                    self.get_logger().info("Met conditions")
                    # move to table
                    self.get_logger().info("table_number: %s" % str(table_number))
                    self.moveToTable(table_number)
                    # wait until switch detects off
                    while(self.switch):
                        self.get_logger().info("wating for can to be removed")
                        rclpy.spin_once(self)
                    # return to dispenser
                    self.returnFromTable(table_number)
                    # reset table_num to -1
                    table_number = -1
        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)
```

The move function combines the 4 phases into one function. First, it will start phase 1 of "`Dispense`" by constantly polling for table number and the state of the switch. It will only start phase 2 of "`Delivery`" when a table number exists and when the switch returns true, of which then it will run the `moveToTable` function. After the end of `moveToTable` function, it will start phase 3 of "`Collection`", stopping at the table until the can is removed using the limit switch state. After the can is collected, it will start final phase 4 of "`Return`" using the `returnFromTable` function. It will then reset the table number and return to phase 1.

In case of exceptions, it will print the error and stop the bot immediately.

```python
def on_table_num(client, userdata, msg):
    global table_number
    if (msg.payload.decode('utf-8') != ""):
        val = int(msg.payload.decode('utf-8'))
        if(val >= 1 and val <= 6):
            table_number = val
    print(table_number)
```

The `on_table_num` function is a MQTT callback function that will reassign the `table_number` value with the table number sent through MQTT.

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

    client = mqtt.Client("Turtlebot")
    client.message_callback_add('ESP32/tableNumber', on_table_num)
    client.connect(IP_ADDRESS, 1883)
    client.loop_start()
    client.subscribe("ESP32/tableNumber", qos=2)
    table_nav = TableNav()
    table_nav.move()

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


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

This is the main driver function where it initialises the ROS Client Library and instantiates the MQTT client and `TableNav` ROS node. It sets up the MQTT client with the `IP_ADDRESS` and the port number and subscribes to the `/ESP32/tableNumber` topic. The move function of the `TableNav` node is then called to start the process.

8.4.4 Detailed Breakdown of Program on the RPi

`switchpub.py`

```python
import rclpy
from rclpy.node import Node
from std_msgs.msg import Bool
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BCM)
LimitSwitchPin = 21
GPIO.setup(LimitSwitchPin, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

At the start of the program, the relevant libraries are imported and the GPIO pin for the limit switch is defined.

```python
class SwitchPublisher(Node):

    def __init__(self):
        super().__init__('SwitchPublisher')
        self.publisher_ = self.create_publisher(Bool, 'limit_switch', 10)
        timer_period = 0.1  # seconds
        self.timer = self.create_timer(timer_period, self.timer_callback)

```

In the above snippet, the SwitchPublisher node is defined. The publisher is created with the Bool message type and the topic of `/limit_switch`. The timer period is defined and every 0.1 seconds, the message will be published by the callback function explained below.

```python
    def timer_callback(self):
        msg = Bool()
        if GPIO.input(LimitSwitchPin) == 0:
            msg.data = True
        else:
            msg.data = False
        self.get_logger().info(str(msg.data))
        self.publisher_.publish(msg)
```

In the callback function, it will read the value of the limit switch and publish the data as a Bool message through the topic `/limit_switch`. It will be True when the limit switch detects an object, and False when the limit switch does not detect an object

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

    minimal_publisher = SwitchPublisher()

    rclpy.spin(minimal_publisher)

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


if __name__ == '__main__':
    main()

```

The above snippet is a driver code to initialise the ROS Client Library and to instantiate the `SwitchPublisher` node. It will then start the publisher using the spin function.

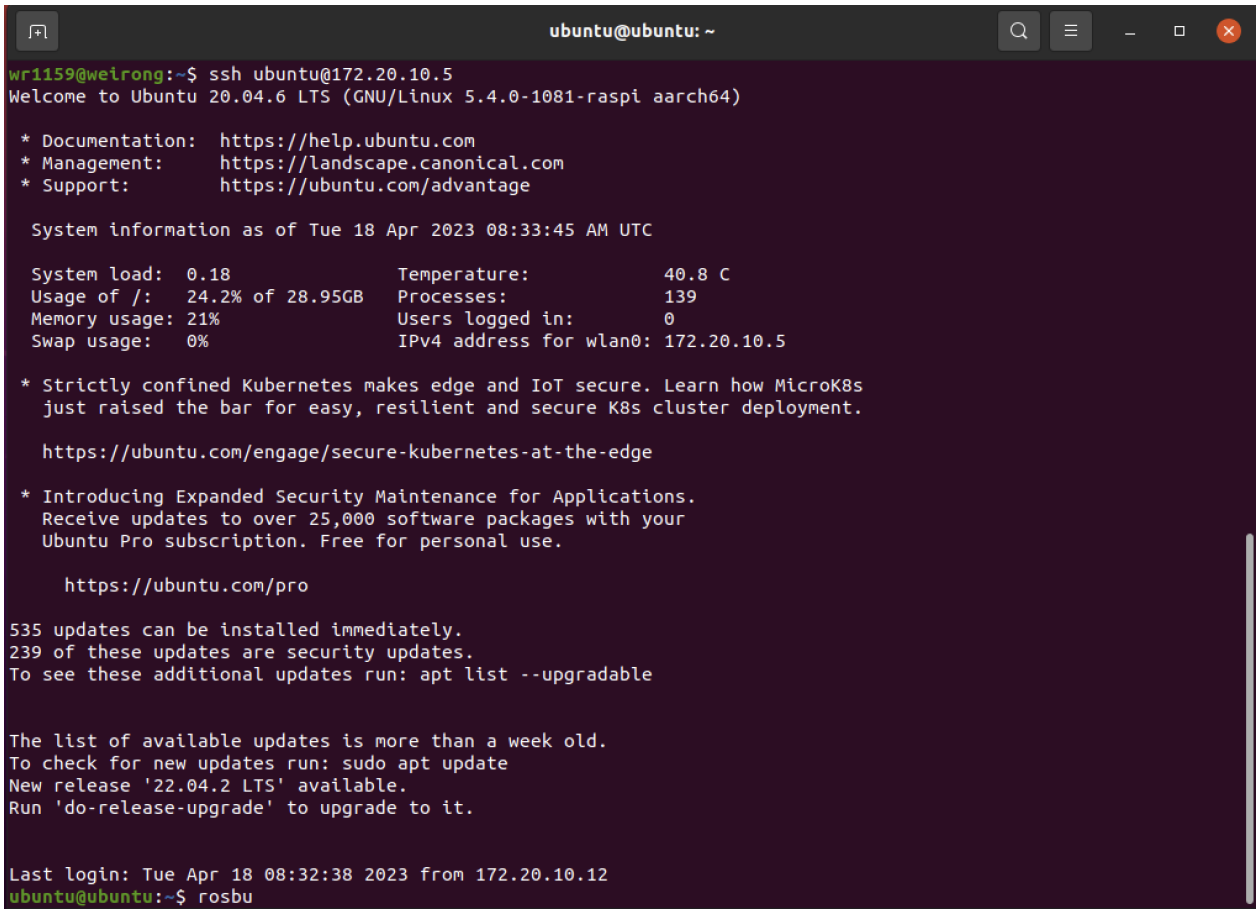
9. System Operation Manual

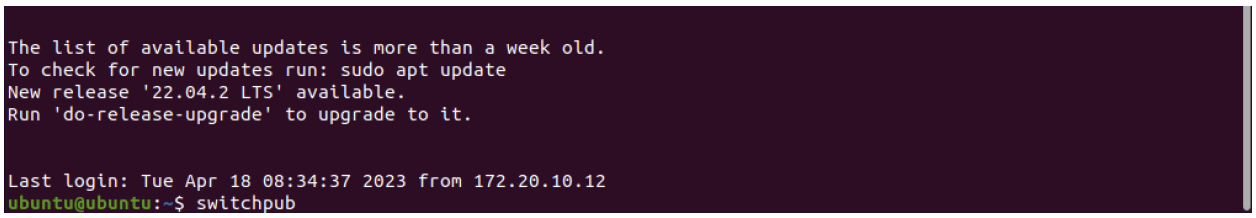
9.1 Software Boot-up Instructions

1. Ensure both the dispenser and TurtleBot are on, the ESP32 must have a red light flashing and both the OpenCR should have a constant green light and the RPi should have a blinking green light.
2. Open up 5 terminal instances in the form of tabs or windows.
3. In 2 of the terminal instances, ssh into the RPi using the command

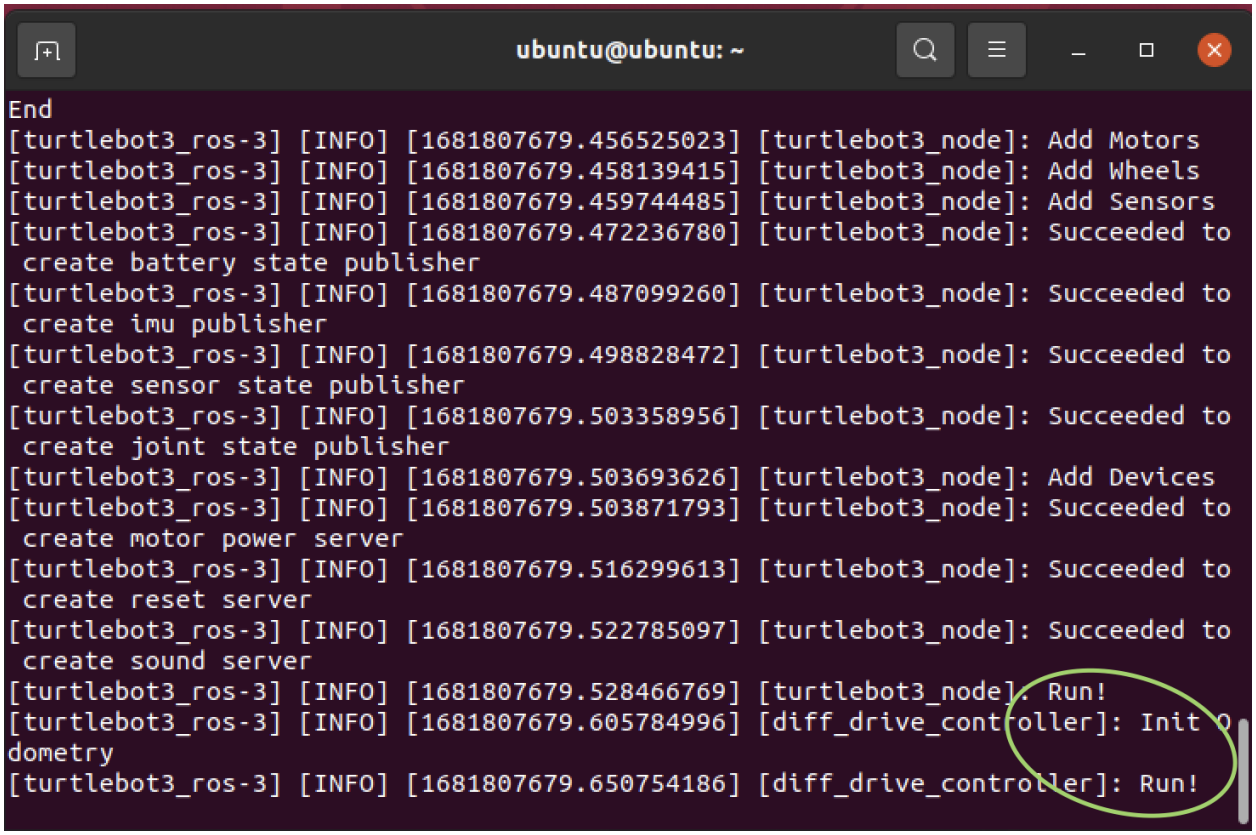
   ```
   ssh ubuntu@{IP_ADDRESS}
   ```

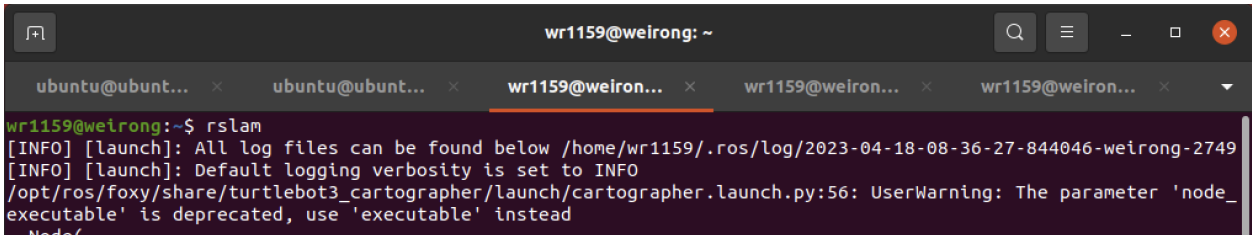
4. Once successfully ssh-ed into the RPi, run the commands `rosbu` and `switchpub` on the 2 terminal instances as shown in the pictures below.

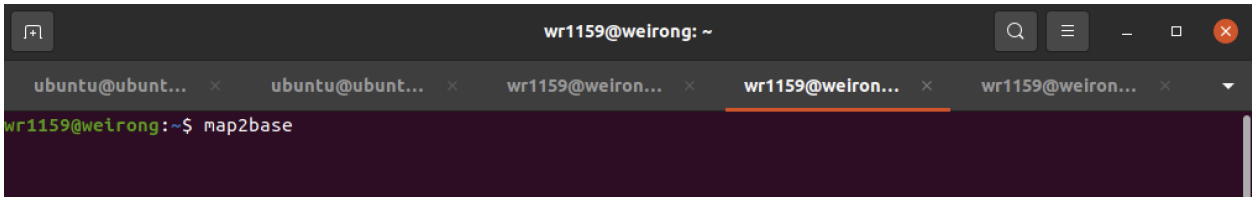


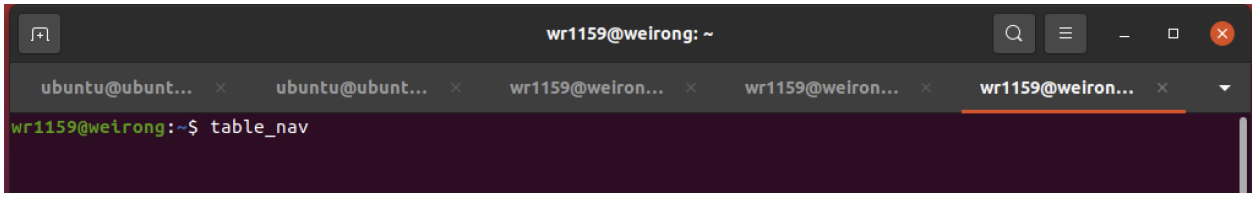


5. Once `rosbu` has successfully been completed, as denoted by the 2 “Run!”s shown below, run the commands rslam, `map2base` and `table_nav` in the remaining 3 instances in this exact order.









6. Once the TurtleBot has completed its deliveries, it would wait for the next can and repeat the deliveries until the terminals instances are closed.

9.2 TurtleBot and Dispenser Positioning Instruction

Dispenser

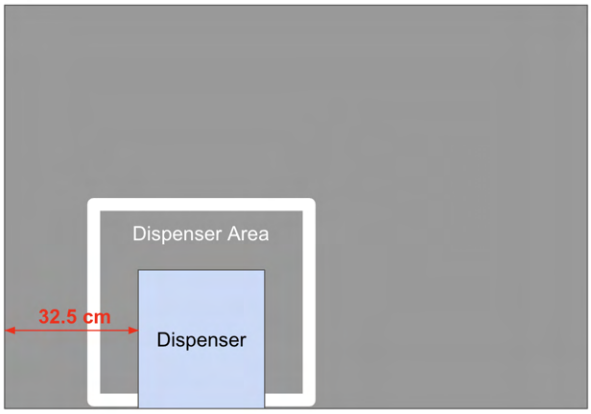


Figure #. Diagram of Dispenser Positioning

- Place the dispenser to face Table 1
- 32.5 cm apart from the left wall of the restaurant

TurtleBot

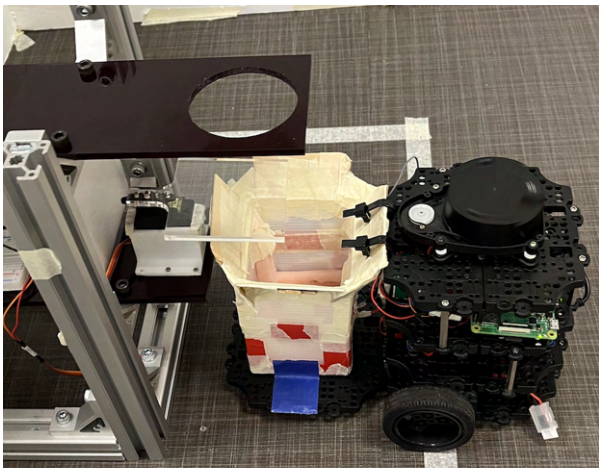
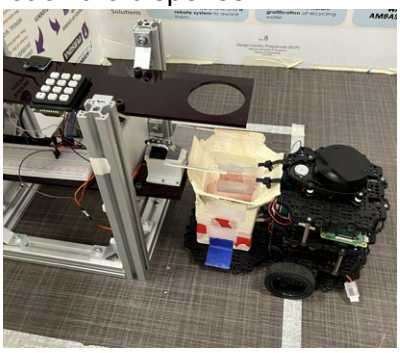
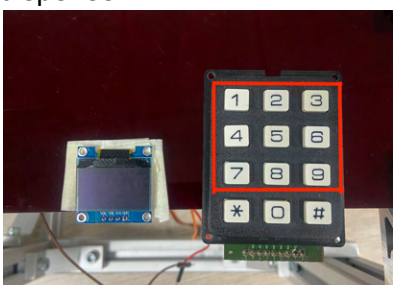
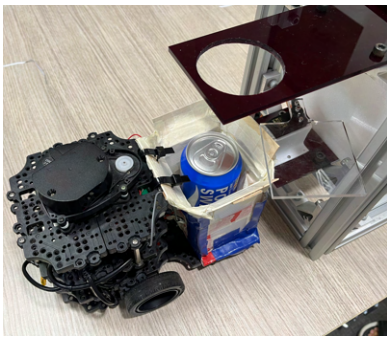


Figure #. Diagram of TurtleBot Positioning

- Initially set up so that the TurtleBot is facing parallel to the dispenser

9.3 Dispenser Operation Instructions

1. Wait for the TurtleBot to reach the dispenser 	2. Place the can on the platform through the hole 	3. Key in the table number into the keypad only when the robot has docked at the dispenser. 
4. If wrong number is keyed in, press '*' to backspace 	5. Press '#' to confirm the table number 	6. Wait for the stopper to turn and dispense the can into the TurtleBot. 

9.4 Instructions to Set Waypoints for New Layout

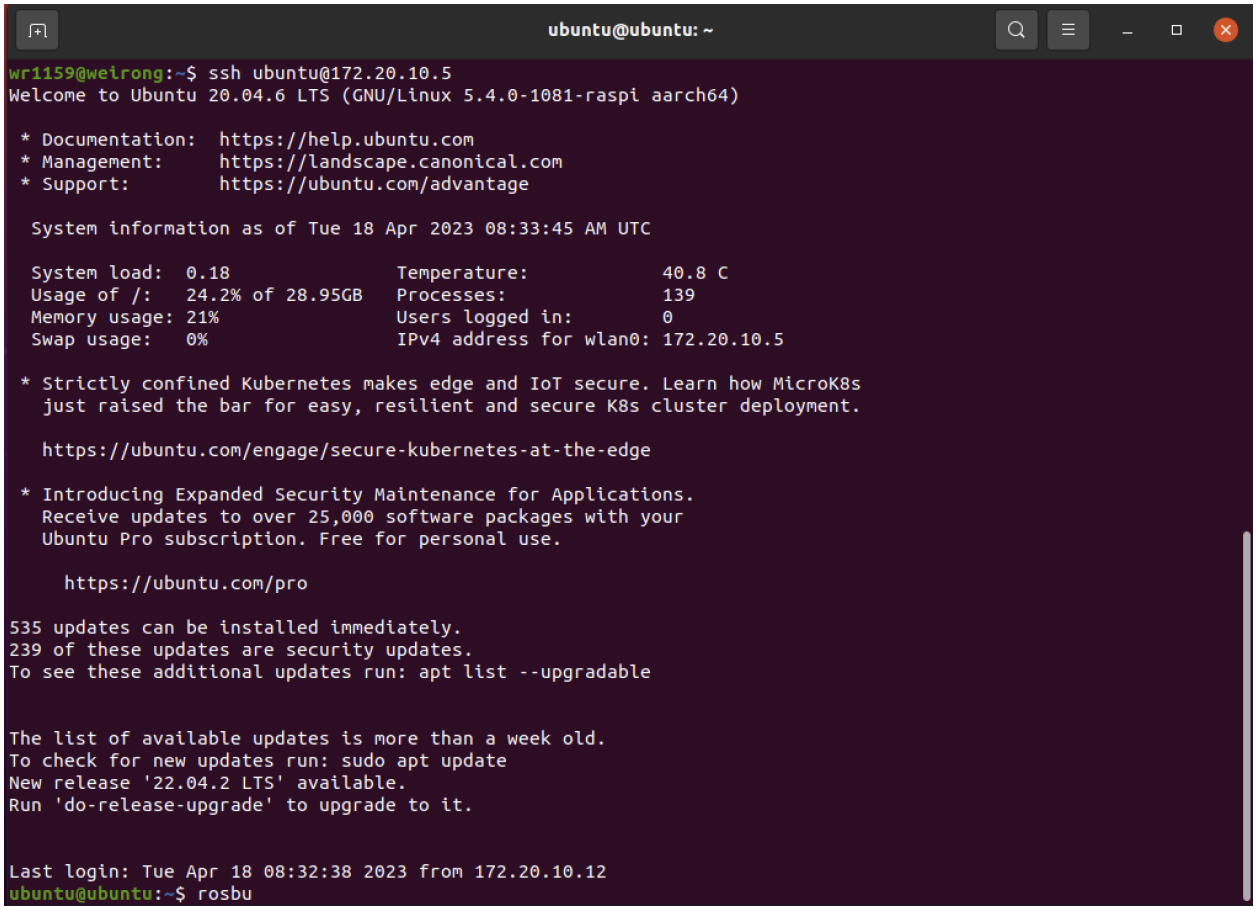
1. Ensure both the dispenser and TurtleBot are on, the ESP32 must have a red light flashing and both the OpenCR should have a constant green light and the RPi should have a blinking green light.

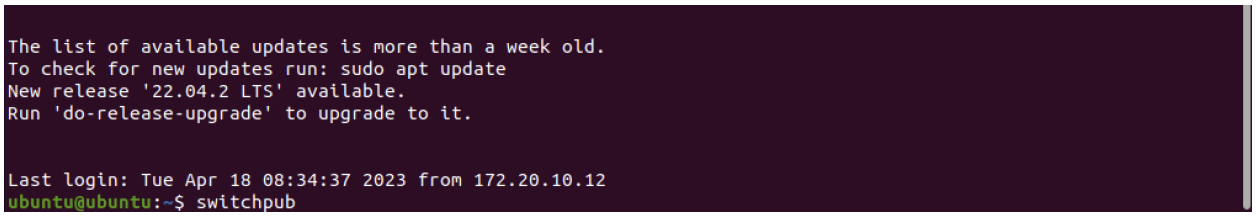
2. Open up 5 terminal instances in the form of tabs or windows

3. In 2 of the terminal instances, ssh into the RPi using the command of

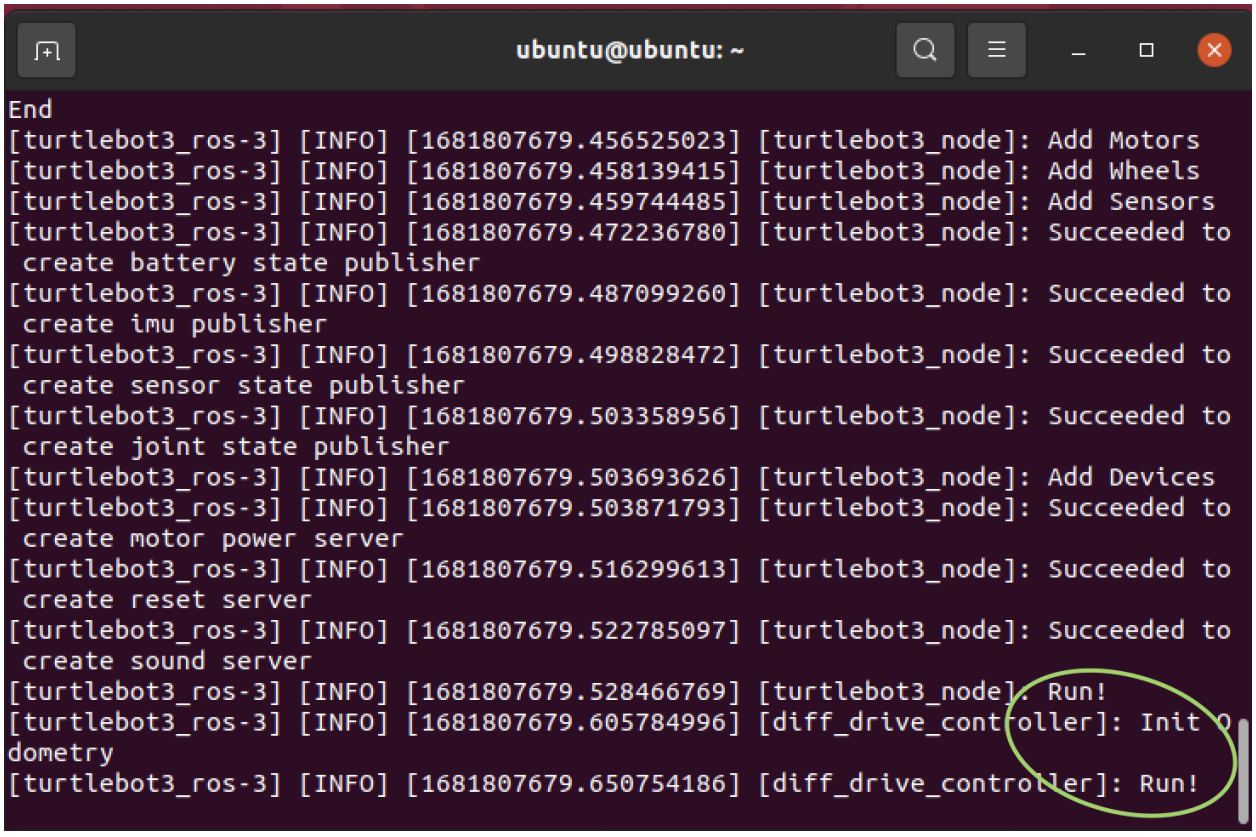
   ```
   ssh ubuntu@{IP_ADDRESS}
   ```

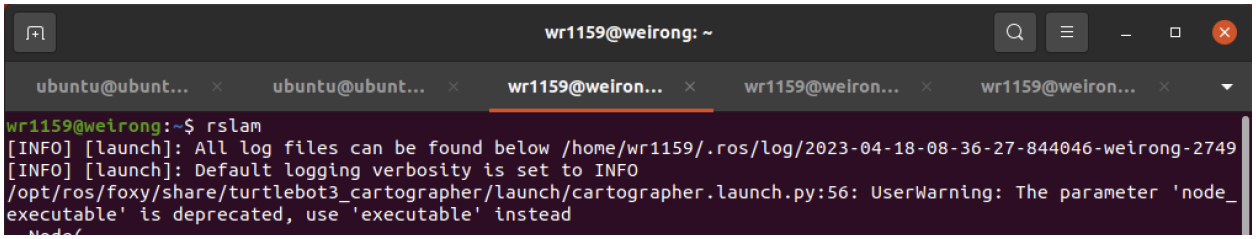
4. Once successfully ssh-ed into the RPi, run the commands `rosbu` and `switchpub` on the 2 terminal instances as shown in the pictures below.

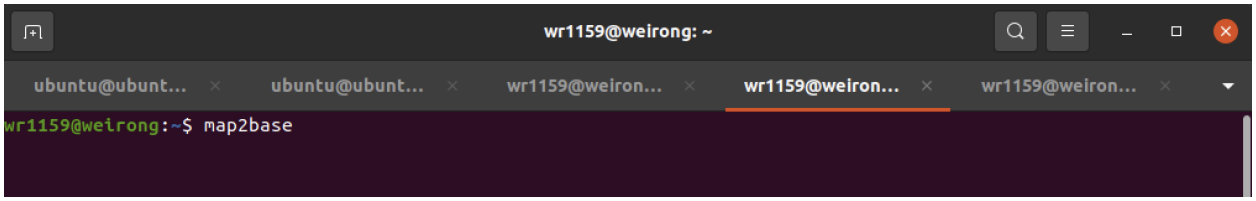




5. Once `rosbu` has successfully been completed, as denoted by the 2 “Run!”s shown below, run the commands rslam, `map2base` and `r2waypoints` in the remaining 3 instances in this exact order.







6. Start the TurtleBot with the front facing the Dispenser originally.
    a. There are 2 points that must be set for all tables regardless of their location, that is the starting point and the docking aid point a few distance away form the dispenser
        i. Set that as a point for all the remaining tables by inputting “p” followed by ENTER and the total table numbers like “123456”.
        ii. Input “x” followed by enter and the total table numbers like “123456”.
        iii. Now you can navigate to other tables and map the waypoints for the other tables. Make sure you set waypoints as a straight route from the other waypoint. There must not be any obstacles in between the route.
    b. Follow this rteleop guide for a detailed guide of moving the TurtleBot using wasdx keys. The controls are the same with r2waypoints.
    c. You can turn the TurtleBot by a certain angle by inputting the desired number.
    d. Save that waypoint by inputting “p” followed by enter and the table number which requires that point.

10. Troubleshooting

10.1 Troubleshooting - Hardware

1. High bending of dispenser's stopper
    a. Strengthen screwing between servo motor and motor holding case
2. Can does not drop from the dispenser's stopper
    a. Ensure that software commands to fully rotate the stopper are in place
    b. Ensure that the height of the 'frontpack' does not block the stopper from rotating
3. Can gets stuck midway of the TurtleBot's 'frontpack'
    a. Increase funnel inclination to guarantee vertical drop of the can
    b. Strengthen walls of 'frontpack' walls and funnel to prevent deformation
    c. Ensure that the 'frontpack' is held tightly to the TurtleBot's main body through zip ties
4. Can has not been detected by the 'frontpack' when the can is inside it
    a. Ensure that the microswitch is in place, embedded in the wall of the 'frontpack', and not pushed outside
    b. Ensure that the protection on top of the microswitch is enough to prevent physical harm on the microswitch due to repeated can drops
    c. Add sponges to the inner walls of the 'frontpack' to guarantee that the can comes in contact with the microswitch when placed inside
    d. Check the microswitch for defects
5. TurtleBot's wheels (castor, tire wheels) do not roll smoothly
    a. Ensure strengthened screwing of castor balls to waffle plates
    b. Smoothen out the tires and ensure there are not foreign materials stuck on the tires
    c. Check the floor for bumps and scratches; take out any tapes or other materials making the floor uneven

10.2 Troubleshooting - Software

1. RPi is unable to connect to the WiFI
    a. Ensure that the WiFi is 2.4 GHz compatible as the RPi used, RPi 3B+ cannot connect to 5GHz WiFi.
    b. Ensure that the WiFi SSID and Password has been added to the RPi by doing the following
        i. Connect the RPi to a monitor and keyboard
        ii. Enter the following commands

```
cd etc/netplan
sudo nano 50-cloud-init.yaml
```

        iii. In the yaml file, edit the SSID and Password so that it matches the WiFi and hotspot used in the restaurant.

2. Laptop is unable to connect to the RPi
    a. Ensure that the laptop is connected to the same WiFi as the RPi.
    b. Ensure that the WiFi is 2.4 GHz compatible as the RPi used, RPi 3B+ cannot connect to 5GHz WiFi.
3. ROS2 topic list on laptop is not synchronised with RPi
    a. Ensure that `ROS_DOMAIN_ID` in `~/.bashrc` of both devices are the same
    b. Ensure that WiFi/hotspot used is 2.4GHz and does not have any internal firewall
    c. Rerun the install command
4. `rslam` crashes when it opens
    a. Run `rslam` before running `rosbu`
    b. Run `bash ./install_ros2_foxy.sh`
    c. Reinstall cartographer using

```
sudo apt --reinstall install ros-foxy-cartographer
sudo apt --reinstall install ros-foxy-cartographer-ros
```

5. ImportError, no module named “***”.
    a. Install the module using the following command:

```
sudo pip install <module_name>
```

    b. If pip is not found, follow the following steps

```
sudo apt update
sudo apt install python3-pip
```

6. “mapbase has no x value.” error after running `tablenav`
    a. Ensure `map2base` is running properly
    b. Restart `map2base` by typing `Ctrl`+`C` twice and re-running the `map2base` command
    c. Restart the `tablenav` instance by typing `Ctrl`+`C` and re-running the `tablenav` command
7. Keypad number does not show up on the OLED
    a. Ensure that the wires of the OLED and Keypad are secured
    b. Press the ESP32 reset button
    c. Connect the ESP32 to the laptop using a micro usb cable
        i. Open up Arduino IDE and view the serial monitor
        ii. 

        If the serial monitor shows “Connecting to ‘WIFI_NAME’.....” with many dots, restart the wifi or hotspot.
        iii. Ensure that the ssid and password variables have the correct ssid and password

8. Connecting to broker failed in serial monitor

```
17:08:02.247 ->
17:08:02.247 -> Connecting to broker '172.20.10.5'...FAILED:
17:08:05.255 -> -2
17:08:05.255 -> Retrying in 5 seconds.
17:08:10.258 -> Connecting to broker '172.20.10.5'...FAILED:
```

   a. Ensure that the broker is on, in this case the broker is the turtlebot.
   b. Ensure that the broker_ip variable has the correct ip address.
   c. Ensure MQTT is running on the broker by typing

```
mqtt -v
```

   d. Ensure that the MQTT config file has been setup properly by editing the MQTT config file and add the first code block into the config file by following the second code block below.

```
# Add this to the config file
listener 1883
allow_anonymous true
```

```
# Run the following commands to edit
$ sudo cd /etc/mosquitto
$ sudo nano mosquitto.conf
```

11. Future Scope

11.1 Mechanical

11.1.1 Integration of Frontpack

The current iteration of the TurtleBot3 Burger makes use of adhesives such as tape and zip ties in order to secure the ‘frontpack’ onto the extra waffle plate. It could be better to manufacture a specific design of the ‘frontpack’ through 3-D printing and then using standoffs in order to mount it rather than simply attaching it. For the purpose of the project, the rigid structural stability provided by screws and standoffs was not necessary but integrating this new mounting mechanism could bring the following benefits:

1. No need to continuously keep replacing the adhesives after a set amount of time that the TurtleBot has done its function.
2. Easier to disassemble to troubleshoot
3. Reduced risk of components shaking and vibrating when the TurtleBot is in motion.
4. Future iterations could include more sophisticated components and the use of standoffs would allow for better heat dissipation.
5. Increased reliability of the ‘frontpack’ staying in place.

11.1.2 Number of Cans that Dispenser Holds

The current project presented to us only required us to dispense one can at a time, which is why we chose the underlying mechanism. However, having a multiple can holding system can be time and labour efficient for restaurant owners. It could also accommodate customer tables that order multiple drinks together.

11.2 Electronics

11.2.1 Queue System

In the future, we will be implementing a queue system to allow more than 1 order to be taken while the turtlebot is out for delivery. This will make the restaurant’s service more efficient as the waiter can immediately place a drink in the dispenser when an order is received, not needing to wait for the turtlebot to be back before entering a single order. For this to work, we will need to implement an additional input system in the arduino code uploaded to our ESP32, and also to the queue system implemented on our computer which sends a command to the RPi.

11.2.2 Compact Design

The wire management of our dispenser will definitely be improved where we will use PCBs to avoid tangling wires and resistors.

11.2.3 OLED Display

We will improve our OLED display to display exactly what the user keys in, where a backspace will cancel the last keyed in number and numbers keyed in will appear next to the previously

keyed in number. This allows for the restaurant's expansion when the number of tables they have goes up to the 10s or 100s.

11.2.4 Keypad

Currently, the enter button and backspace button on our keyboard is not that intuitive where a user manual must be read before understanding how to use the keypad. To improve simplicity, we will change the backspace and enter buttons on the keypad to words instead of symbols.

11.3 Software

11.3.1 Ease of Use

To better convenient the user of the can delivery robot system, there should be an option to allow input of the next table number even while the TurtleBot is on delivery. This could be implemented with a next table number variable which would be assigned to the table number variable when TurtleBot returns to the dispenser instead of assigning the table number variable to -1.

The above next table number feature could be implemented alongside 2 way communication between the Ubuntu laptop and the dispenser. The laptop should publish a message to the dispenser to denote when the TurtleBot is ready to receive a can. The dispenser can then subscribe to this message of `/ready` topic and only dispense when it receives a valid value from that topic. This implementation of 2 way communication will allow the dispenser to autonomously dispense for the next table once the TurtleBot returns, allowing for further ease of use.

The mapping of waypoints could be more user friendly by allowing users to set the waypoints on RViz, the GUI for Cartographer. This could be as simple as adding the points using the mouse instead of operating the TurtleBot with the CLI `r2waypoints` script.

11.3.2 Navigation

Currently, the TurtleBot will have to stop to recalibrate and turn towards the target waypoint. This makes the TurtleBot slower than it is, it should be able to self correct its route on the way towards the target waypoint. This could be done by applying PID control to the navigation algorithm, allowing the TurtleBot to self correct its yaw value. PID control is a control loop mechanism to readjust values.

With PID control, the TurtleBot will be able to navigate between waypoints seamlessly. It will not need to stop after arriving at the intermediary waypoint and rotate towards the next waypoint. Instead, it could start turning towards the next waypoint while it is near the waypoint, decreasing the time the customer will have to wait for their canned drink.

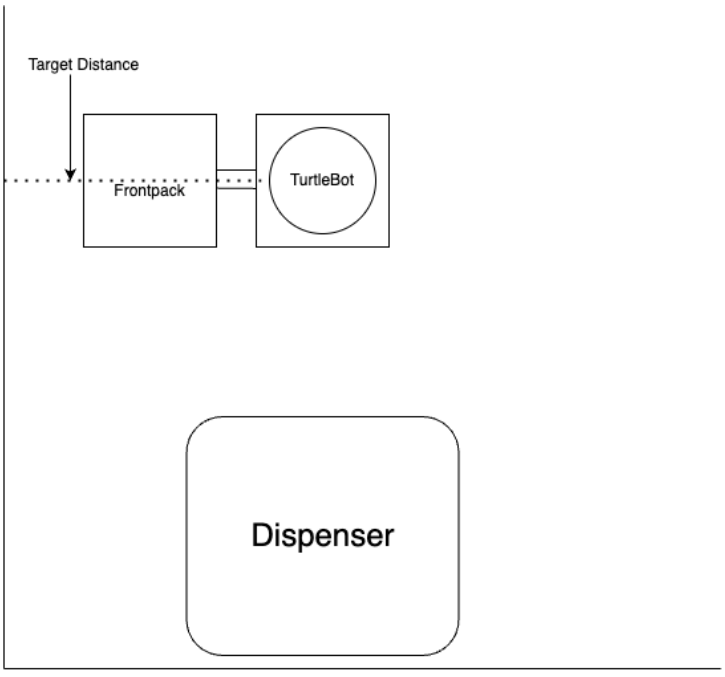
In the final mission run, the TurtleBot made unexpected turns and routes not following the waypoints set. This is likely a result of inaccurate data from `/map2base` (`/map2base` value is not updated every timer period, but stuck at the same value), as the connection between the TurtleBot and the laptop was not stable enough. This could be amended using a router instead of

a mobile phone hotspot. Alternatively, the TurtleBot should have a `map2base` data validation to ensure that the `map2base` value is not repeated.

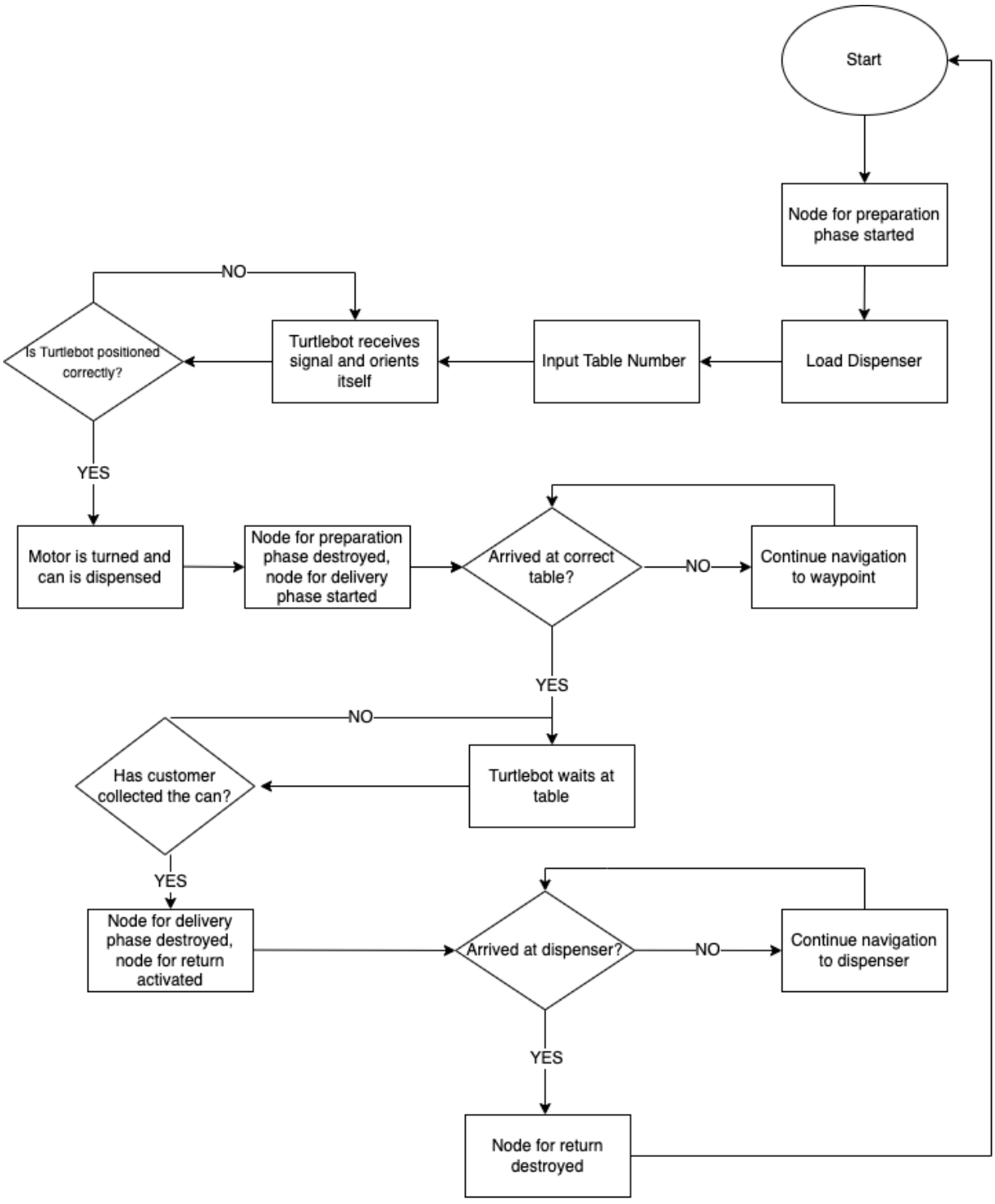
To better fit the use case of a restaurant delivery robot, the TurtleBot should include obstacle avoidance to avoid bumping into customers who could be walking around the restaurant. This could be done using the LDS data from the `/scan` topic.

11.3.3 Docking

Changes must be made to the docking system as it was only able to successfully dock 3/6 times despite the successful docks in the test runs. The failure in docking is likely caused by the overdependence of `map2base` data as a successful dock required the TurtleBot to perfectly align with the dispenser. A solution is to use the LDS to help align with the dispenser by facing the wall and moving forwards if the actual distance is less than the target distance and moving backwards if the actual distance is greater than the target distance. A diagram below illustrates the idea. Once it is within the Target Distance, it will rotate to face the Dispenser and continue the docking until the LDS detects a value less than the `DOCK_DISTANCE`.



Annex A - Preliminary Algorithm Flowchart



Annex B - Power Budgeting Tables

TurtleBot

Component	Voltage	Current	Power
Running off LiPo battery (11.1V 2250mAh/**24.975Wh**)			
TurtleBot (During operation)	11.1V	0.70A	7.77W
Micro Switch	3.3V	0.16A	0.53W
Total Power Consumption			8.3W
Total Running time (Assuming battery efficiency of 85%)			2.5h

Dispenser

Component	Voltage	Current	Power
ESP32	3.3V	250mA	0.825W
OLED	3.3V	20mA	0.08W
Keypad	3.3V	10mA	0.033W
MG995 Servo Motor	5V	1.2A	6W
RPi power adapter	5.1V	2.5A	12.75W

Total power required by components = 6.938W
Total power supplied by adapter = 12.75W

Annex C - Factory Acceptance Tests

Component	To be checked	Observation
OpenCR	Able to be powered by the LiPo Battery	Green LED lights up when connected to a power source, boot up tune being played
RPi	Able to turn on the RPi when connected to OpenCR	Red light turns on while green light flashes
	Can be connected to from the remote laptop	Terminal returns “Welcome to Ubuntu…” when ssh-ing into the RPi
LDS-01	Able to spin and collect data consistently	Environment will be mapped on Rviz
ESP32	Able to be powered by wall plug adaptor	Constant red light on ESP32
MG995	Able to rotate the stopper to allow can to drop, then back to allow a new can to be loaded	Can drops down vertically when the stopper attached to the servo moves 60 degrees horizontally. Servo then rotates the stopper back to its original position.
Keypad	Able to detect when it is pressed	Terminal switches from blank to the table number that was pressed
OLED	Able to display the Adafruit Logo	OLED turns Blue when ESP32 is booted up
Wheels	Able to move the bot in all directions freely	Bot can be controlled properly when running `rteleop`
Ball caster	Able to roll in all directions freely	Bot able to move around in all direction smoothly with ball caster attached
Microswitch	Able to detect when it is pressed	Terminal instance correctly prints state of switch when switchpub is run on the RPi
Structural stability	Structural platforms and components installed correctly	Shake Turtlebot to verify all components are mounted securely
	Verify all fasteners installed and tightened	Verify fastener count are consistent with assembly document