



NUS
National University
of Singapore

EG2310 Group 3 G2 Report



Student Team Members:

AGARWAL ANANYA	A0246120W
HANG JIN GUANG	A0254475X
LOH YIN HENG	A0259282W
SIM YU, JEANETTE	A0254736X
WANG BO	A0252200A

Table of Contents

1. Problem Definition.....	3
2. Organisation of Documentation.....	4
3. Final Design.....	4
3.1. Operation Overview.....	4
3.2. Object Transfer Mechanism.....	5
3.2.1. Dispenser.....	5
3.2.2. Robot Carrier.....	6
3.2.3. Docking.....	7
3.3. Communication Protocols.....	7
3.3.1. Electrical Architecture.....	7
3.3.1.1. Dispenser.....	8
3.3.1.2. Delivery Robot.....	10
3.3.2. Inter-System Communication.....	13
3.3.2.1. Docking: (Dispenser ↔ Delivery Robot).....	13
3.3.2.2. Table Number: (Dispenser → Delivery Robot).....	14
3.4. Navigation.....	15
3.4.1. Navigation Flowchart.....	15
3.4.2. Navigation Paths.....	16
3.4.3. Table 1 to 5 Navigation.....	17
3.4.3.1. Angle Calibration.....	19
3.4.3.2. Distance Calibration.....	22
3.4.4. Table 6 Navigation.....	24
3.5. Operational Setup.....	27
3.5.1. Preparation.....	27
3.5.1.1. Installation.....	27
3.5.1.2. Setup.....	28
3.5.2. System Check.....	29
3.5.3. Operation Parameters.....	30
4. Final Design Fabrication.....	32
4.1. Dispenser.....	32
4.1.1. Mechanical Fabrication.....	32
4.2. Delivery Robot.....	35
4.2.1. Mechanical Fabrication.....	35
4.3. Bill of Materials.....	37
5. Design and Development (Concept → Preliminary Design).....	38
5.1. Object Transfer Mechanism.....	38
5.1.1. Dispenser.....	38
5.1.2. Robot Carrier.....	40
5.1.3. Docking.....	43
5.2. Communication Protocols.....	44
5.2.1. Electrical Architecture.....	44
5.2.2. Inter-system Communication.....	47

5.3. Navigation.....	48
6. Testing & Validation (Preliminary → Critical Design).....	50
6.1. Dispenser Operation.....	50
6.1.1. ESP32.....	50
6.1.2. Keypad & Limit Switch Activation.....	51
6.1.3. Servo Motor.....	52
6.1.4. Integrated Testing.....	53
6.2. Can dispensing, Robot Teleop and Docking.....	54
6.2.1. Mechanical Testing.....	54
6.2.2. Manual Docking.....	55
6.2.3. Autonomous Docking.....	56
6.3. Robot Navigation.....	57
7. Evaluation and Testing (Final Run).....	58
7.1. Factory Acceptance Test.....	58
7.2. Evaluation Setting.....	58
7.3. Evaluation Results.....	60
8. Conclusion & Areas for Improvement.....	60
Appendix 1: Wiring Connections of Hardware Components.....	61
Appendix 2: Literature Review.....	63
1. Object Transfer Mechanism.....	63
1.1. Dispenser System.....	63
1.2. Delivery System.....	64
2. Communication Protocols.....	64
2.1. Delivery System.....	64
2.2. Inter-system Communication.....	65
3. Navigation.....	67
3.1. Mapping.....	67
3.2. Path Planning.....	68
3.3. Table Identification.....	68
References.....	69
Appendix 3: Concept Design (G1 Report - Before Week 6).....	71
1.1. Concept Design of Dispenser system.....	71
1.1.1. Mechanical System.....	71
1.1.2. Electrical System.....	71
1.1.3. Software System.....	72
1.2. Concept Design of Delivery Robot.....	73
1.2.1. Mechanical System.....	73
1.2.2. Electrical System.....	73
1.2.3. Software System.....	74
Appendix 4: Preliminary Design (Week 7 Design Review).....	75

1. Problem Definition

The ultimate goal of this project is to design and construct a payload module for food delivery in a restaurant environment. The module is composed of **two primary systems**: a **Dispenser** and a **Delivery Robot**. The dispenser system receives orders from the kitchen (a TA in this case) and communicates them to the delivery robot, which would be a modified TurtleBot3 Burger. The delivery robot then navigates through the restaurant to deliver the order to the designated table accurately. After delivery, the robot returns to the dispenser to receive the next order, and the process repeats. The payload in this case would be a single soda can at a time.

In accordance with the requirements outlined by the stakeholders and the Mission Master, the following table presents a comprehensive list of the project's deliverables.

Stakeholder's Requirements	Project Deliverables
Food receiving on Dispenser	Dispenser can hold and know when food is received at any time.
Choose which table to deliver food to on the Dispenser itself	Dispenser can acquire input data of designated table numbers & communicate that with the delivery robot.
Food is transferred from Dispenser to Delivery Robot once robot is present	Dispenser has an efficient mechanism to transfer food to Delivery Robot
Dispenser is in a fixed location	Delivery Robot has to move to/from the dispenser, not the other way round.
Delivery Robot navigates towards the designated table	Delivery Robot can navigate through a complex environment safely with the food while avoiding obstacles (walls/tables) automatically.
Delivery Robot waits at the table till the food is delivered over	Delivery Robot has to stop and wait for the food to be collected manually before moving again automatically.
Delivery Robot returns to the dispenser & waits to receive a new order.	Delivery Robot has to know when to return back and wait for the next food delivery after communicating with the dispenser.
Overall Power, Maintenance & Scalability is stable.	<p>Delivery Robot must have a reliable power supply & efficient power management to ensure it can operate for extended periods without interruption.</p> <p>Easy to maintain and repair, with simple and accessible components.</p> <p>Able to handle a large number of requests, and be able to adapt to different restaurant environments and layouts.</p>

The problem is systematically categorised into three distinct areas of focus: **Object Transfer Mechanism**; **Communication Protocols**; and **Navigation**. **Object Transfer Mechanism** encompasses the mechanical aspects of the two modules, specifically focusing on the design and functionality of the dispenser and delivery robot. **Communication Protocols** emphasise on the electrical architecture of interactions and connections between the human, dispenser, and delivery robot, with a specific emphasis on the procedural flow of the entire operation. Lastly, **Navigation** concentrates on the software aspects of the delivery robot, specifically focusing on the development of efficient and safe navigation strategies to the chosen table.

2. Organisation of Documentation

To ensure readability and organisation, we have split our documentation into the following **two main documents**, which are accessible on our **Github** page (https://github.com/yinheng996/r2table_nav):

1. Project Report (this document)
2. End User Document and Technical Log

Additionally, the following material may also be obtained from our **Github**:

1. Delivery Robot navigation code in Python
2. Delivery Robot publisher code in Python
3. Dispenser function code in C++
4. DXF/STL files for laser-cut and 3D printed components (under /Hardware)
5. Final Assessment Rubrics (under /Documentations)

This **Project Report** is designed to list the most important information (such as final designs, fabrication details, testing and validation etc.) first, for readers to understand the motives, design and usage of our system. Additional information, such as literature reviews and concept/preliminary designs, are relegated to the back of the report for interested readers to find out more.

The **End User Document & Technical Log** is a 5-page document containing a quick summary of system specifications, usage instructions, assembly instructions, acceptable defect log, factory acceptance tests and a parts replacement/maintenance log. This document is to be printed for and will accompany each Robot-Dispenser set throughout its service life.

3. Final Design

The Final Design presented in this section details the system used in the Final Run in Week 13. This is the result of the refinements made during the building & testing phase of the project from Week 7 to Week 13), which would be detailed under Design Considerations & Development (Chapter 5), and Testing & Validation (Chapter 6).

3.1. Operation Overview

The main operation flow between the Dispenser and Delivery Robot during the mission is outlined in the accompanying flowchart. This provides an overview of the various stages and steps involved in the system's operation.

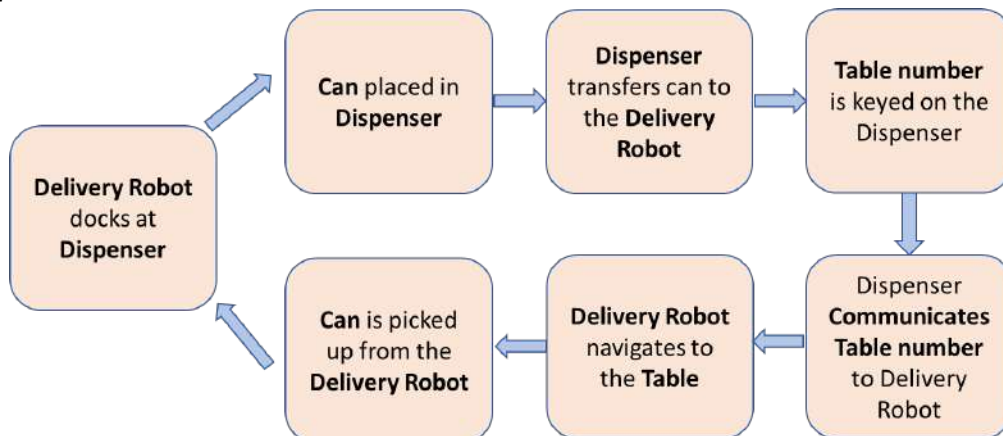


Figure 3.1: Overview of Mission Flow

3.2. Object Transfer Mechanism

The transfer of soda cans from the dispenser to the delivery robot is a crucial component of the system's overall functionality. For this, there is a tilting mechanism within the dispenser carrier that drops the can into the robot's carrier from a height. The following image depicts the entire system, including both subsystems and their respective carriers, once they have been docked with each other.

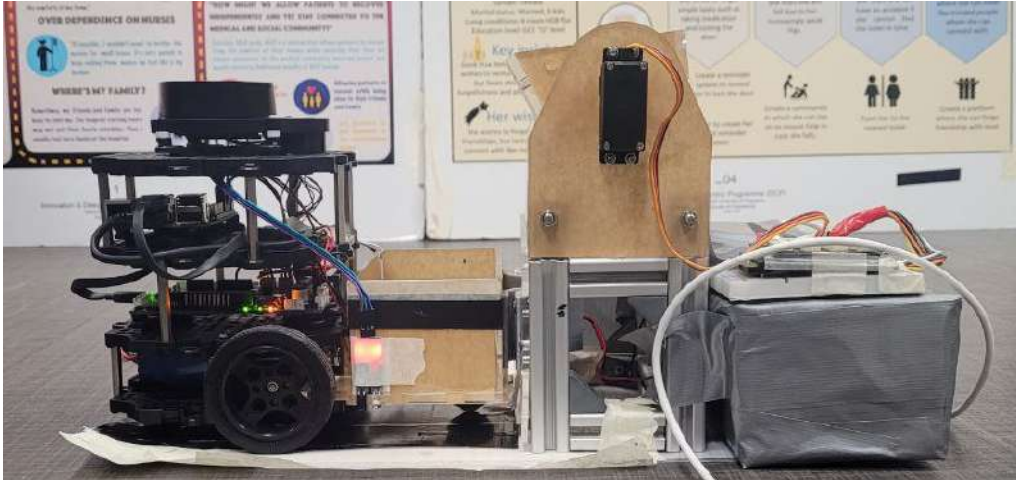


Figure 3.2: Full Integrated System when Docked.

3.2.1. Dispenser

The Dispenser has 4 main assemblies:

1. Dispenser Carrier
2. Supports
3. Profile Bar Base
4. Docking Assembly

The Dispenser Carrier is rotated by a servo motor and a rotary bearing, with one support piece for each. The **Servo-side Support** secures the servo and circular attachment, while the **Bearing-side Support** houses the bearing and constrains it with a washer and M10 bolt. The bearing allows for smooth rotation of the carrier, enabling accurate can transfer to the delivery robot. **The Profile Bar Base** elevates the Carrier and Supports to the right height and features a **Docking Assembly**, including an acrylic plate and a Docking Angle. The Docking Angle aligns the robot with the dispenser, and a limit switch mounted to it detects successful docking.

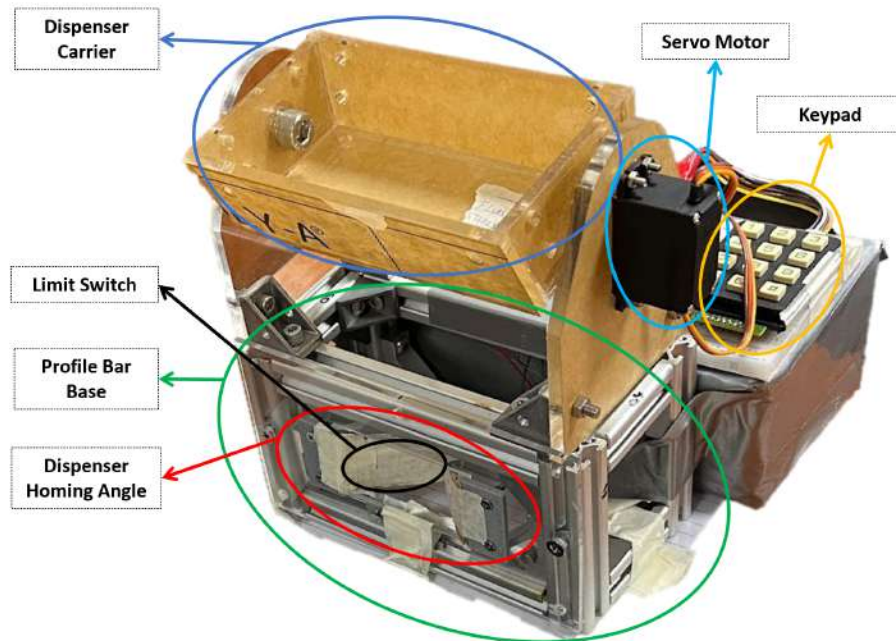


Figure 3.3: Dispenser with Dispenser Carrier & Profile Bar Base.

3.2.2. Robot Carrier

The Robot Carrier is an acrylic assembly attached to the bottom waffle plate on the Turtlebot. It includes a slot for the limit switch and a ball caster wheel to prevent the robot from tilting forwards when the can is loaded. The Robot Homing Angle complements the Dispenser's Homing Angle, with similar geometry, and is located at the front of the carrier.

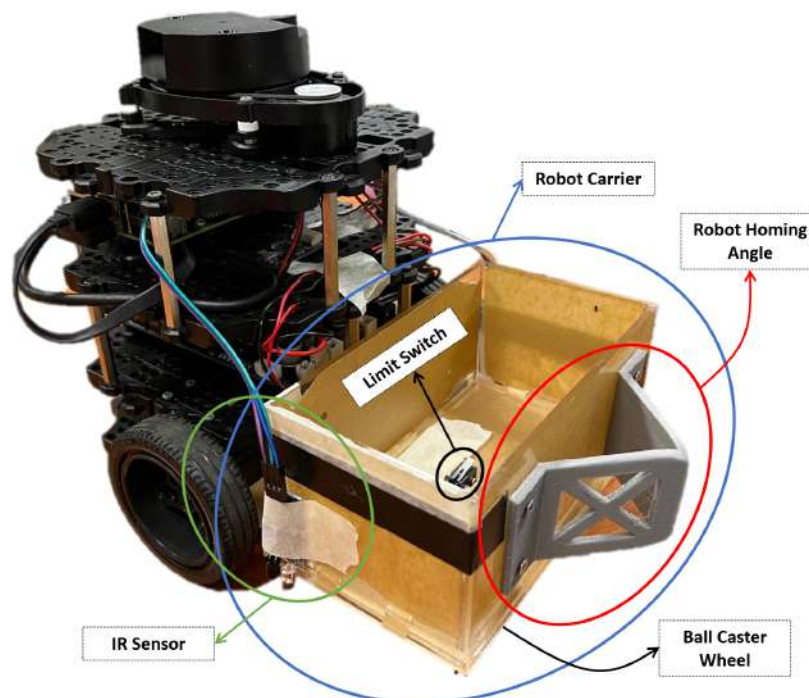


Figure 3.4: Labelled Assembly Image of Robot Carrier with TurtleBot3.

3.2.3. Docking

Successful soda can transfer from Dispenser to Delivery Robot requires docking every time the robot returns to the dispenser, ensuring both systems are ready for delivering to the next table. A **black tape pathway** on the dispensing area floor guides the robot to the dispenser, with **Infrared Sensors** on both sides to aid alignment. The **Robot Docking Angle** and **Dispenser Docking Angle** align as the robot approaches, and the Robot Homing Angle presses the limit switch to confirm successful docking.

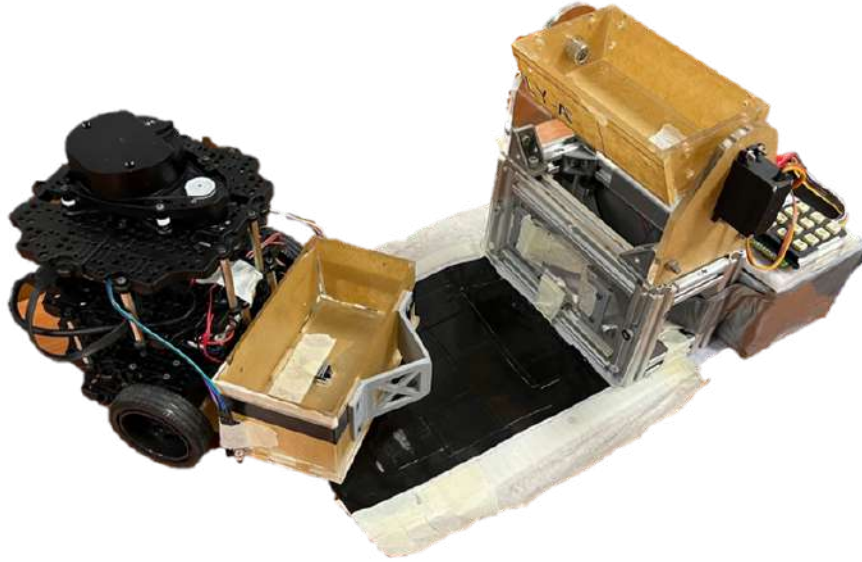


Figure 3.5: Dispenser & Robot Homing Angle Structure.

3.3. Communication Protocols

3.3.1. Electrical Architecture

The electrical architecture provides information about essential hardware components used by the dispenser and delivery robot. Each system includes a **Breadboard Diagram**, **Schematics Diagram**, and **Power Calculation**. Precise power calculation is critical for powering multiple components and avoiding issues like overheating or component failure. A stable power supply optimises energy consumption, improving efficiency and reducing battery replacements or recharging needs.

3.3.1.1. Dispenser

For the dispenser, the following table shows the hardware components required with their specific uses.

Type of Component	Use
1. Microcontroller (ESP32) - Espressif ESP32 WROOM 32	- Main brain for program flow between these components. - Main power for the keypad & limit switch.
2. Limit Switch - D2F-01L	- Located at Dispenser Homing Angle as part of the docking assembly for the dispenser to detect the arrival of the delivery robot at the dispensing station. - Activated when the robot has been docked to the dispenser, indicating that the robot is in the right alignment is ready to receive the loaded can.
3. Keypad - 3x4 Membrane Matrix Keypad	- For the TA to choose which table number for the delivery robot to navigate towards and deliver the soda can accordingly.
4. Servo Motor - MG996R	- Connected to the dispenser carrier and tilts the loaded carrier for the can to be dropped on the robot's carrier once the robot has arrived.

Figure 3.6 displays hardware component interconnections for the dispenser, and their corresponding power sources are labelled with their voltages. A set of **four 1.5V AA** batteries in a battery holder powers the servo motor, while the ESP32 powers the keypad and limit switch. The ESP32 is powered via a **MicroUSB data cable** when connected to a laptop, which also acts as a program flow display for the user during the mission. **Appendix 4** includes pinout diagrams, connection details, and datasheets for the servo motor, keypad, and limit switch.

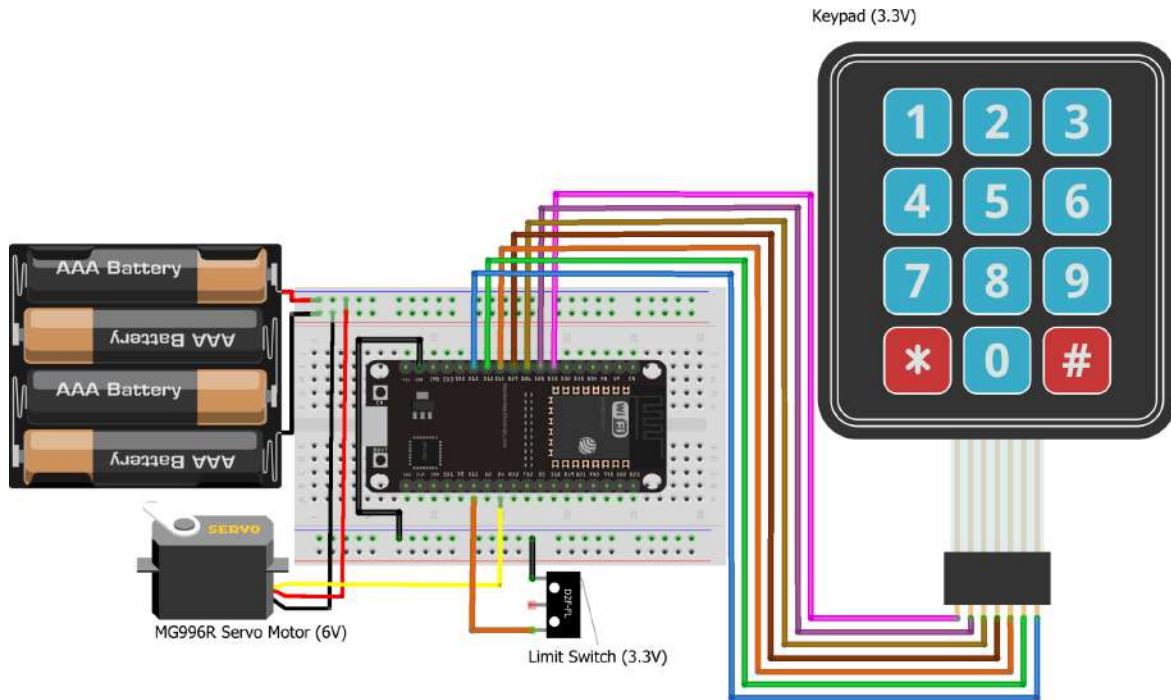


Figure 3.6: Electrical Architecture of Dispenser System (Breadboard)

In **Figure 3.7**, the wiring to the pins of the ESP32 and its components are schematically presented, providing a clearer visual representation of the connections and circuitry within the system.

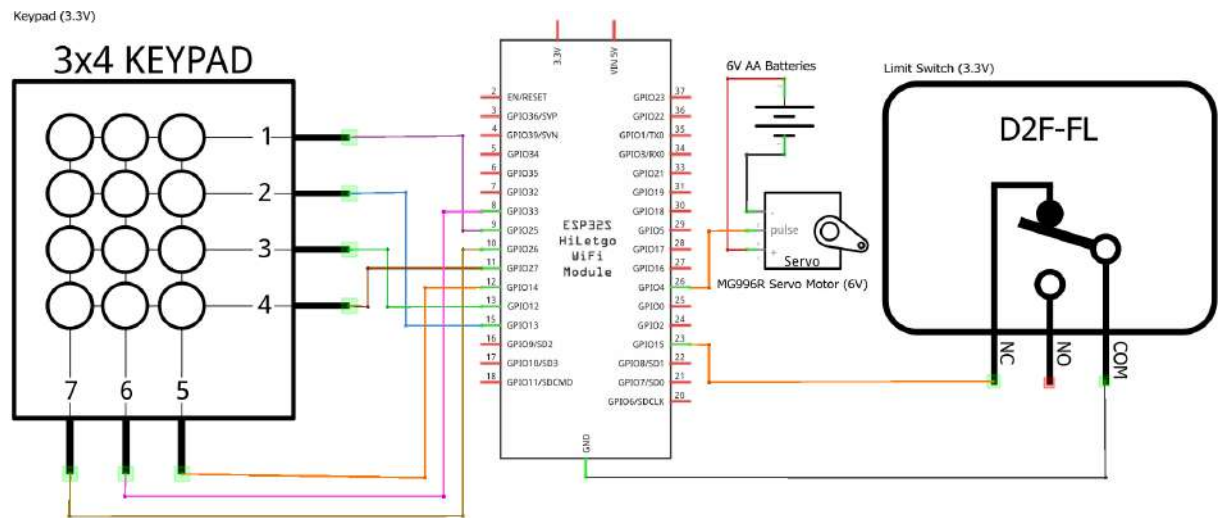


Figure 3.7: Electrical Architecture of Dispenser System (Schematics)

Figure 3.8 presents a **power budgeting table** that outlines the power requirements and distribution for hardware components in the dispenser system. The two primary power sources are the **laptop** and a set of **four AA batteries**, with the former powering the ESP32 and the latter the servo motor. The servo motor requires a higher voltage than the ESP32, which necessitated the use of batteries. Four AA batteries provide a combined voltage of 6V (1.5V each) to ensure each component receives the appropriate voltage for correct operation.

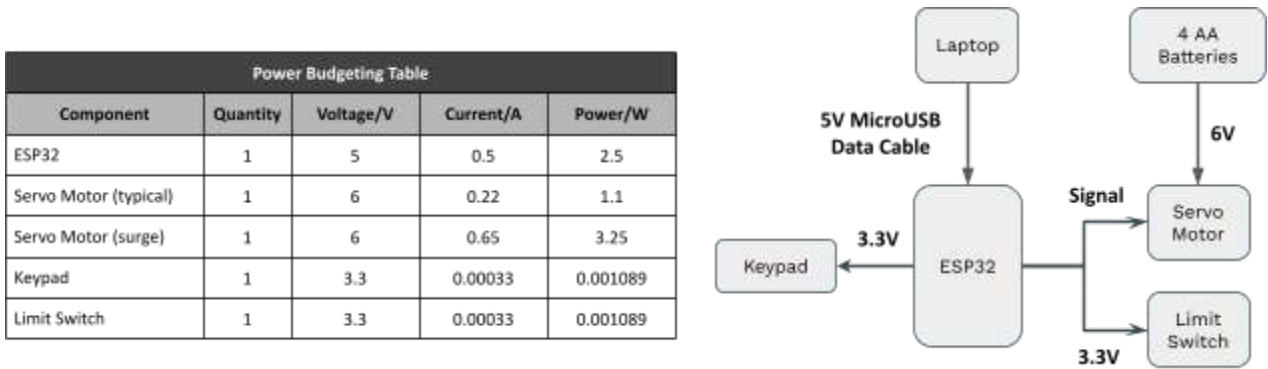


Figure 3.8: Power Budgeting Table with Power Distribution of Dispenser System

3.3.1.2. Delivery Robot

For the delivery robot, the following table shows the additional hardware components required with their specific uses.

Type of Component	Use
1. Microcontroller (Raspberry Pi 3)	<ul style="list-style-type: none">- Main brain for program flow between these components.- Main power for the limit switch and IR sensor.
2. Limit Switch - D2F-01L	<ul style="list-style-type: none">- Located at the bottom in the robot's carrier to detect the arrival of the loaded can from the dispenser.- Activated when the can has dropped into the carrier, indicating that the can has been transferred successfully from the dispenser to the robot.
3. IR Sensor	<ul style="list-style-type: none">- Located at both sides of the robot's carrier to detect the presence of a black-coloured pathway, which helps the robot move in the correct alignment towards the Dispenser Homing Angle.

Figure 3.9 illustrates the interconnections among the electronic components for the delivery robot, along with the corresponding voltage supplied by their respective power sources. The three components are all powered by the **Raspberry Pi 3 (R-Pi)**, which is ultimately powered by a **Li-Po Battery**. The pinout diagrams and connection details for the limit switch and IR sensors are included in **Appendix 4**, along with their datasheets as well. Meanwhile, the schematics of the whole connection is presented in **Figure 3.10**.

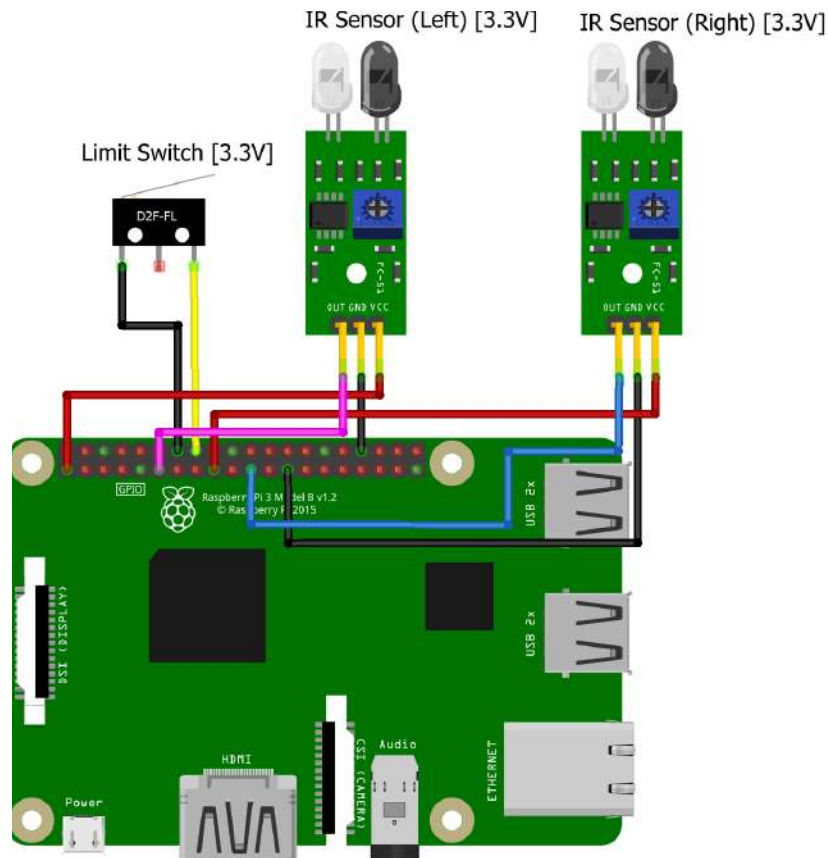


Figure 3.9: Electrical Architecture of Dispenser System (Breadboard)

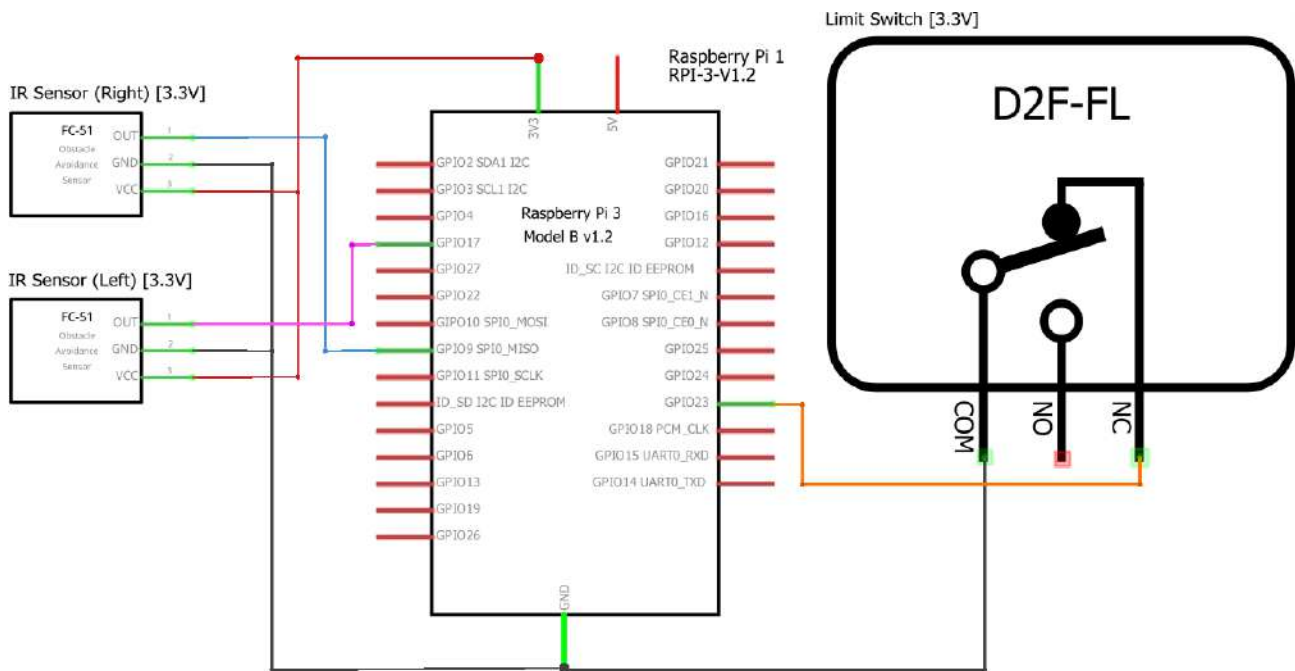
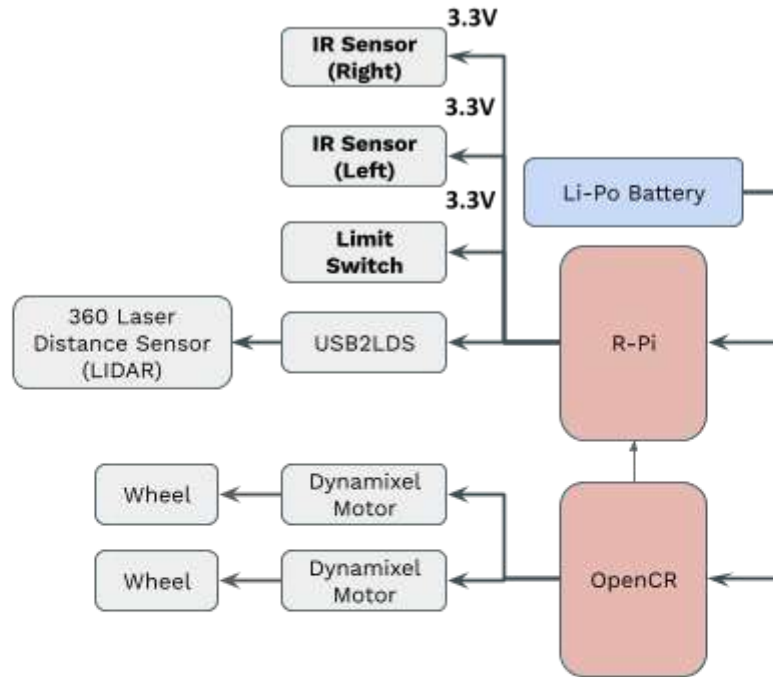


Figure 3.10: Electrical Architecture of Delivery Robot System (Breadboard + Schematics)

Figure 3.11 shows the **power budgeting table** for the **delivery robot system** as a whole, and the power distribution figure for all of the hardware components with only the known voltages acquired being labelled. The primary source of power for the hardware components is the **Li-Po Battery** provided for the TurtleBot and it has a power rating of **1800mAh**. It is responsible for **powering up the R-Pi and OpenCR** in the TurtleBot, which then power up the other required components for the robot itself.



Power Budgeting Table						
Component	Quantity	Voltage/V	Current/A	Power/W	Runtime/s	Energy Required/Wh
TurtleBot (bootup)	1	11.2	0.7	7.84	180	0.392
TurtleBot (standby)	1	11.2	0.59	6.61	600	1.101
TurtleBot (during operation)	1	11.2	0.72	8.06	1500	3.360
Limit Switch	1	3.3	0.00033	0.001089	1500	0.000454
IR sensor	2	3.3	0.02	0.066	1500	0.055

Figure 3.11: Power Budgeting Table with Power Distribution of Delivery Robot System

The following are then the calculations performed to assess the ability of the battery's capacity at full charge to power the TurtleBot system throughout the entire mission.

$$\text{Total Power Required} \times \text{Operation Time} = \text{Total Energy Consumed}$$

$$\text{Energy Required: } 0.392Wh + 1.101Wh + 3.360Wh + 0.000454Wh + 0.055Wh = 4.909Wh$$

$$\text{Energy Provided by Battery: } 11.1V \times 1800mAh = 19980mWh = 19.98Wh$$

$$\text{Considering Safety Factor of 60\%: } 4.909Wh \times 1.6 = 7.85Wh < 19.98Wh$$

Even after accounting for a safety factor, the energy demand of the TurtleBot during the mission is found to be lower than the battery capacity, which suggests that the battery chosen is adequate to support the TurtleBot's operation.

3.3.2. Inter-System Communication

The two **main inter-system communications** required for our system are for the **docking** between the dispenser and the delivery robot and conveying the **table number** from the dispenser to the delivery robot. The main communicating protocol selected for the communication between both systems of both the docking status and table number data is MQTT, a Message Queuing Telemetry Transport protocol which involves the use of publishing the necessary messages into a topic and having the required clients to subscribe to the topic in order to receive the data.

This involves three essential components: **Publisher (ESP32)**, **Subscriber (R-Pi)**, and **Broker (MQTT)**. The ESP32 creates two topics: "docking" to publish docking status and "table_num" to publish table number input. The R-Pi reads input data by subscribing to both topics. The **MQTT broker** collects and stores messages from all publishers and distributes them to correct subscribers. The desktop app "MQTTX". on the laptop connected to the ESP32, manages the broker and displays necessary data during the mission. All devices must be connected to the same WiFi network for seamless communication. Figure 3.12 illustrates the entire MQTT system and its connections.

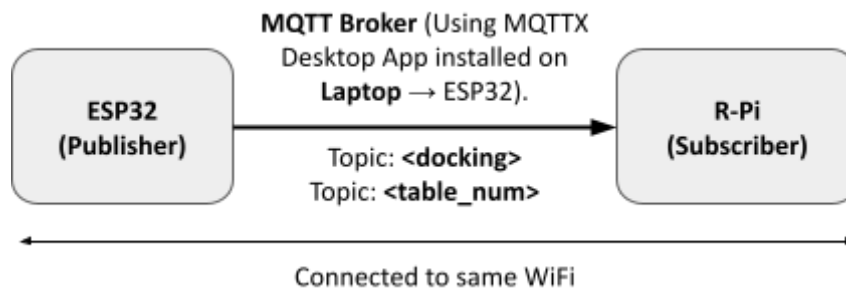


Figure 3.12: Overview of MQTT System

3.3.2.1. Docking: (Dispenser ↔ Delivery Robot)

➤ Mechanical

The Dispenser Docking Angle guides the delivery robot to the dispenser centre, even with minor misalignments. When the limit switch is activated, the robot has docked so it sends a signal to the Delivery Robot to stop moving. After confirming the docking status to the delivery robot, it waits for the can transfer to begin.

➤ Software (Navigation)

During the mission setup, the centre of the dispenser is positioned 50 cm away from the wall, and the robot verifies its position using the Distance Calibration method in Section 3.4.1. An experimental margin of error of $\pm 1\text{cm}$ was determined, which can be corrected by the docking structure. This guarantees accurate docking of the robot with the dispenser for successful can transfer.

➤ Software (Arduino/MQTT)

The ESP32 code has a variable "docking" storing a boolean for the Dispenser Homing Angle's limit switch status. It's vital for the robot to know when to stop, so the ESP32 publishes this information via MQTT on the "docking" topic. The R-Pi subscribes to this topic to get the updates. '1' represents the limit switch being activated and '0' if not. The status is published every 2 seconds to keep the robot informed until it reaches '1' and stops.

3.3.2.2. Table Number: (Dispenser → Delivery Robot)

➤ Mechanical

The keypad allows the user to enter the table number for delivery during the mission. The numbers 1 to 6 on the keypad are used, and the code only accepts input within this range. After the robot reaches the dispensing station and confirms its arrival using the limit switch, the ESP32 waits for input from the keypad. The table number is then sent from the ESP32 to the robot, enabling it to navigate to the designated table.

➤ Software (Arduino/MQTT)

After the user selects the table number on the keypad, the value is stored in a variable called "table_num" in the Arduino code. The ESP32 then publishes this data to the MQTT broker under the topic of the same name. The R-Pi on the delivery robot is subscribed to this topic and receives the message containing the selected table number, which must be within the range of 1 to 6. The dispenser carrier with the loaded can then tilts and drops the soda can with the help of the servo motor once the table number has been published. Once both conditions are met, the robot will leave the dispenser and navigate towards the designated table. The dispenser will then wait for the robot's arrival again for the next can delivery.

3.4. Navigation

Based on the map provided, we have carefully determined the directional movements required to reach each table. No mapping or computational path planning for navigation were involved in our Final Design.

3.4.1. Navigation Flowchart

Below is the flow of our navigation script. It is repeatable for all 6 tables.

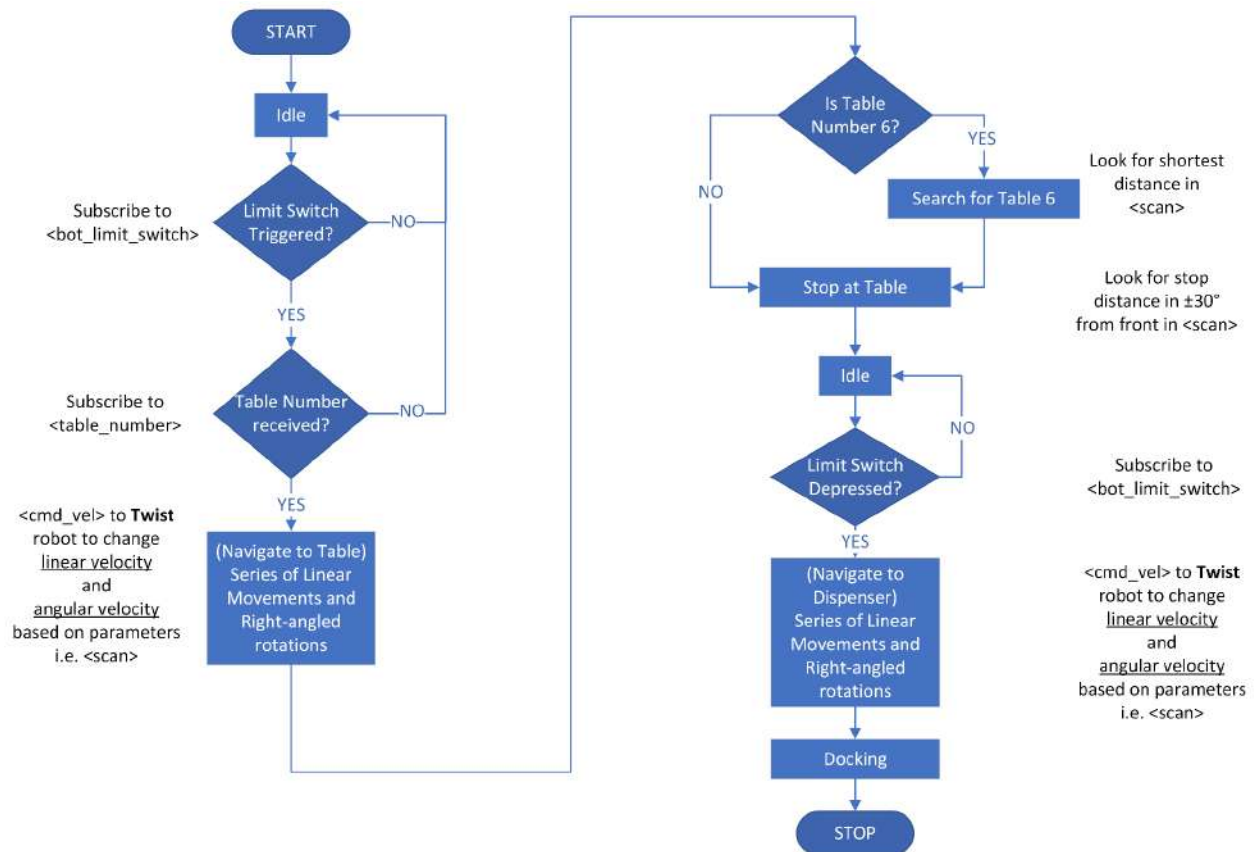


Figure 3.13: Flowchart for Navigation to all 6 tables

3.4.2. Navigation Paths

To navigate through the restaurant, our robot follows a carefully planned path that includes a Main Axis and two Reference Lines. The Main Axis serves as the primary route, with the robot moving straight along it to reach Table 1. To reach Table 5, the robot breaks out into Reference Line 2. For Tables 2, 3, 4, and 6, the robot uses Reference Line 1. The robot returns to the dispenser similarly. By using Main Axis and Reference Lines 1 and 2 for reference, we can systematically minimise errors and increase the efficiency of our navigation system.

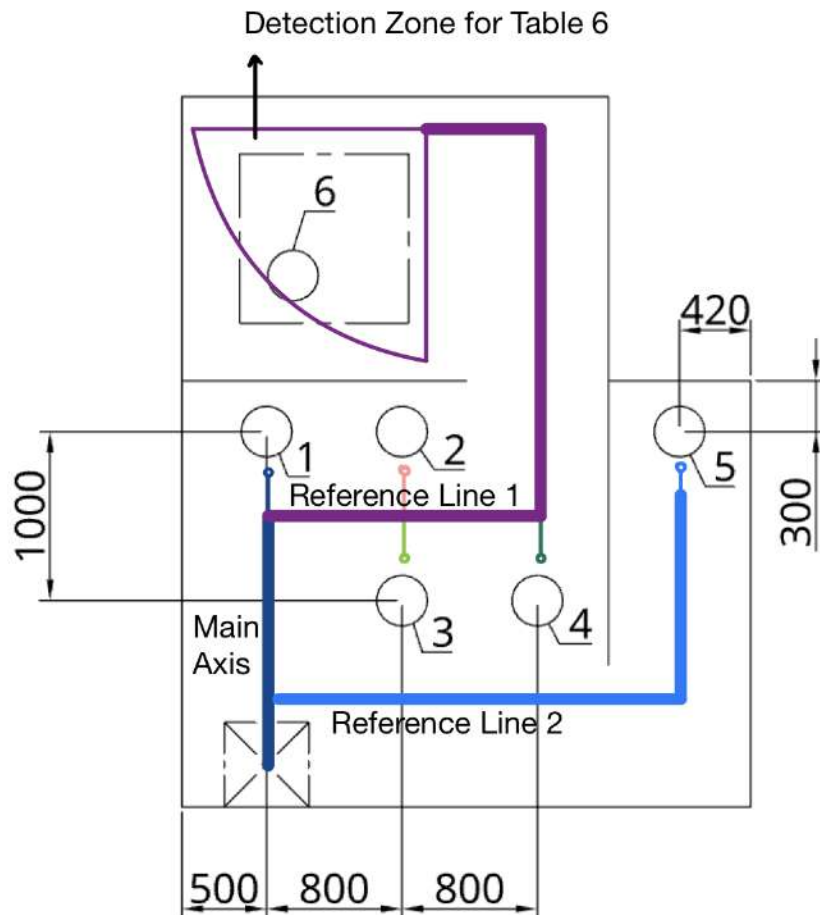


Figure 3.14: Navigation Paths for Robot

3.4.3. Table 1 to 5 Navigation

Table 1 to 5 Navigations are mainly based on a series of linear movements and right-angled rotations as outlined above.

```
# all-in-one function for linear movements
# first input == direction of movement (forward or backward)
# second input == angle to check (0, 90, 180, 270)
# third input == check if more or less than the input distance (more or less)
# fourth input == distance to check
# fifth input == point of reference (lidar or front, back, centre of rotation
from front, centre of rotation from back or lidar)

def move_til(self, direction, angle, more_less, dist, pt_of_ref):

    self.get_logger().info('Moving %s until distance at %s degrees is %s than %s
from %s' % (direction, angle, more_less, dist, pt_of_ref))

    move_dict = {'forward': 1, 'backward': -1, 'more': True, 'less': False}

    # create Twist object, publish movement
    twist = Twist()
    twist.linear.x = move_dict[direction] * moving_speed
    twist.angular.z = 0.0
    time.sleep(1)
    self.publisher_.publish(twist)

    # convert distance to float
    dist = float(dist)

    #rebasng the distance based on the point of reference
    if pt_of_ref == 'front':
        dist += move_dist_f
    elif pt_of_ref == 'back':
        dist += move_dist_b
    elif pt_of_ref == 'centre of rotation from front':
        dist += move_dist_corf
    elif pt_of_ref == 'centre of rotation from back':
        dist += move_dist_corb

    print('Distance to check: %s' % dist)

    self.get_logger().info('Linear movement initiated')

    # create parameter to check distance
    check_dist = self.laser_range[angle]

    while math.isnan(check_dist) or (((not math.isnan(check_dist)) and
(check_dist < dist)) == move_dict[more_less]):
        #allow the callback functions to run
        rclpy.spin_once(self)
        check_dist = self.laser_range[angle]

    # log the info
    self.get_logger().info('Distance at %s degrees: %f' % (angle,
check_dist))

    # stop moving
    self.stopbot()
```

Figure 3.15: Code snippet of our `move_til()` function for robot linear movements in [r2checkpt_nav.py](#)

```

def rotatebot(self, rot_angle):

    if rot_angle > 5:
        rotation_speed = rotation_speed_fast
    else:
        rotation_speed = rotation_speed_slow

    # 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 * rotation_speed
    # 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)

```

Figure 3.16: Code snippet of our `rotatebot()` function for robot rotations in [r2checkpoint_nav.py](#)

However, upon careful examination of our robot's linear movements and rotations, we observed significant drifts. To address this issue, we conducted several experiments utilising the [r2calib_lidar.py](#) and [r2calib_odom.py](#) files available in our GitHub repository. These experiments allowed us to evaluate the distance in a specified angle and rotation of a specified degree, respectively.

Our findings revealed that the observed drifts were due to random errors, rendering it impossible to determine a consistent drift offset. To overcome this challenge, we incorporated angle and distance calibration to correct for the drift offsets and ensure that our robot can move under more ideal conditions.

3.4.3.1. Angle Calibration

To address the issue of inconsistent rotations due to Odometry-based measurements, we have implemented a triangulation-based method to determine the robot's alignment with the wall. This method involves determining the location of the robot with respect to the walls by measuring the angles between the robot and two points on the wall.

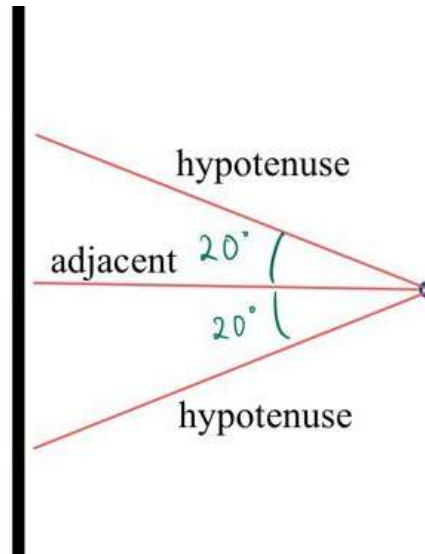


Figure 3.17: Triangular Method for Calibration

Since

$$\frac{1}{\cos(20^\circ)} = \frac{\text{hypotenuse}}{\text{adjacent}} = 1.064$$

We check for both hypotenuses:

$$\frac{\text{hypotenuse}}{1.064} - \text{adjacent} \leq 0.01(\text{aligned})$$

If the robot is found to be misaligned, we instruct it to make a small correction of 0.5 degrees in the determined direction. While these 0.5 degree rotations are still Odometry-based, the error is negligible due to the small angle of rotation.

To ensure that the robot's alignment with the wall is always accurate, we continuously check the alignment with the wall, applying the necessary correction whenever a misalignment is detected. However, to prevent infinite recursion, we have set a limit of 50 calibrations. We have found that this limit is more than sufficient in our experiments.

We execute this angle calibration function before and after every turn to ensure that the robot's orientation is correct. In the unlikely event that the robot cannot align with the wall within 50 tries in one spot, the angle calibrations that we apply along the way would be able to correct any misalignment. Thus, we can effectively account for any angle drift.

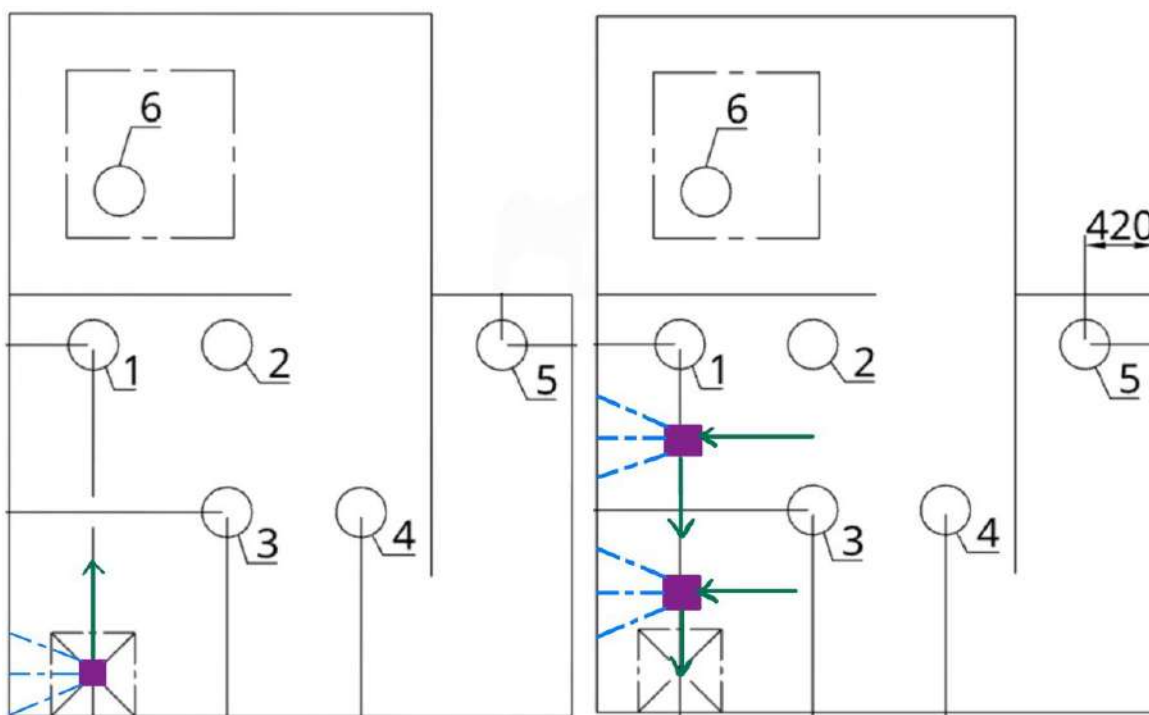


Figure 3.18: Multiple calibrations along navigation path to ensure consistency.

It is worth noting that this method assumes that the walls are straight and perpendicular around corners. However, in a real restaurant setting, this is usually not a concern. Nevertheless, we performed extensive checks on the walls to ensure that they are reliable for the robot to calibrate to.


```

# calibration using triangulation
def calibrate(self, direction):
    max_attempts = 50 # to avoid infinite recursion

    calib_dict = {'R': [250, 270, 290, 'right'], \
                  'L': [70, 90, 110, 'left'], \
                  'F': [340, 0, 20, 'front'], \
                  'B': [160, 180, 200, 'back']}

    for attempt in range(max_attempts):
        print(f"Aligning with {calib_dict[direction][3]} wall...")
        if (abs((self.laser_range[calib_dict[direction][0]]/1.064) -\
self.laser_range[calib_dict[direction][1]])) <= 0.01 and \
(abs((self.laser_range[calib_dict[direction][2]]/1.064) -\
self.laser_range[calib_dict[direction][1]])) <= 0.01:
            print("Aligned")
            break

        else:
            if self.laser_range[calib_dict[direction][0]]/1.064 >\
self.laser_range[calib_dict[direction][1]] or \
self.laser_range[calib_dict[direction][1]] >\
self.laser_range[calib_dict[direction][2]]/1.064:
                print("Tilting left")
                self.rotatebot(0.5)
                rclpy.spin_once(self)

            elif self.laser_range[calib_dict[direction][2]]/1.064 >\
self.laser_range[calib_dict[direction][1]] or \
self.laser_range[calib_dict[direction][1]] >\
self.laser_range[calib_dict[direction][0]]/1.064:
                print("Tilting right")
                self.rotatebot(-0.5)
                rclpy.spin_once(self)

        else:
            print("Failed to align after {} attempts".format(max_attempts))

```

Figure 3.19: Code snippet of our `calibrate()` function for angle calibrations in [r2checkpoint_nav.py](#)

3.4.3.2. Distance Calibration

To address the issue of inconsistent distance measurement while on the run due to lags in LiDAR logging speed, we included a simple distance calibration function where the robot would move at minimum speed to ensure minimal lag from data logging to reaction. The function is particularly valuable in preventing collisions when making turns or docking. All distance checked is with respect to the centre of rotation of the robot for convenience.

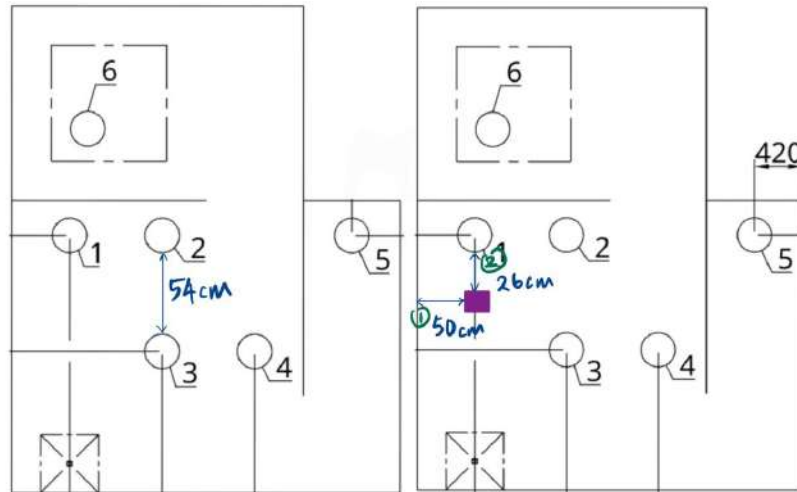


Figure 3.20: Distance of the gap between Table 2 & 3 and our Distance Calibration Details before going through the challenging gap.

For example, the space in between tables 2 and 3 is measured to be 54 cm in the actual arena. Given the robot's width of 17.8 cm, there is only a slim 18.1 cm clearance on each side. To address this, we deployed our distance calibration method. First, we positioned the robot in front of table 1, ensuring a distance of 50 cm from the wall to align the centre of rotation with the table's centre. Next, we calibrated our position with the left wall using angle calibration, then carefully positioned the robot 26 cm away from table 1 to pass precisely through the centre of the path between tables 2 and 3.

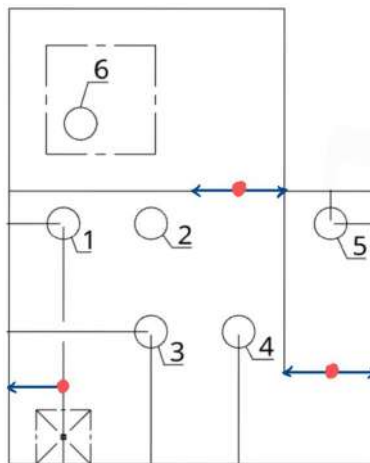


Figure 3.21: High-risk spots that Distance Calibration is particular useful in

The distance calibration has also been noticeably useful for navigation through the smaller gaps on the way to table 5 and table 6 as well as to dock at the dispenser accurately. We determined experimentally that the margin or error for this specific setup is $\pm 1\text{cm}$, which is acceptable in our case both in terms of docking and navigation.

```

def check_dist(self, limit):

    # ensure distance at 0 degrees is limit
    while rclpy.ok():
        rclpy.spin_once(self)
        self.get_logger().info(str(self.laser_range[0] -\
lidar_offset + centre_of_rotation_f_offset))

        # create Twist object, publish movement
        twist = Twist()
        if round((self.laser_range[0] - lidar_offset +\
centre_of_rotation_f_offset),3) > (limit + 0.01):
            twist.linear.x, twist.angular.z = 0.01, 0.0
        elif math.isnan(self.laser_range[0]):
            twist.linear.x, twist.angular.z = 0.0, 0.0
        elif round((self.laser_range[0] - lidar_offset +\
centre_of_rotation_f_offset),3) < (limit - 0.01):
            twist.linear.x, twist.angular.z = -0.01, 0.0
        else:
            twist.linear.x, twist.angular.z = 0.0, 0.0
            break

        time.sleep(1)
        self.publisher_.publish(twist)
        rclpy.spin_once(self)

```

Figure 3.22: Code snippet of our `check_dist()` function for distance calibrations in [r2checkpt_nav.py](#)

3.4.4. Table 6 Navigation

Utilising the methodology delineated in [Section 3.4.2](#), the robot navigates to the upper right corner of the Table 6 Zone and positions the Table 6 Zone within its upper left quadrant.

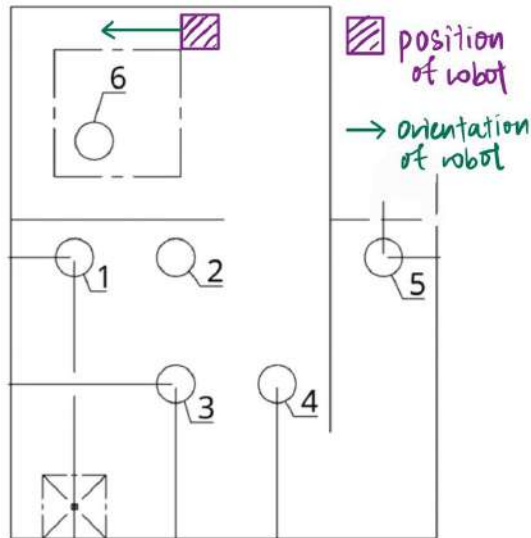


Figure 3.23: Illustration showing Position and Orientation of Robot to search for Table 6

Subsequently, to search for Table 6, the robot scans the upper left quadrant to identify the nearest obstacle. It can be proven through calculations that the robot can't mistake the wall for Table 6 in this spot, even if Table 6 is located at the farthest corner from the robot. Furthermore, using our Distance Calibration method outlined in [Section 3.4.1.2](#), we are able to position the robot very accurately. Hence, the obstacle closest to the robot in its upper left quadrant will always be Table 6.

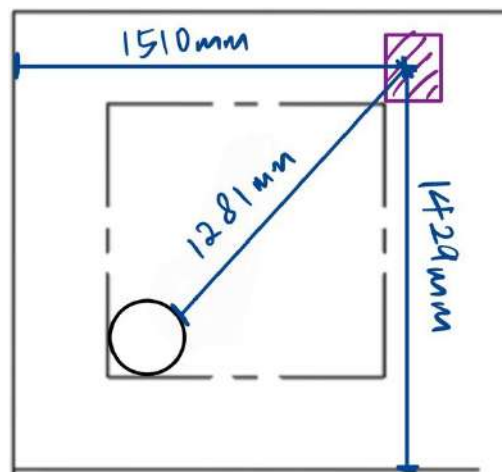


Figure 3.24: Illustration showing Table 6 will be closest obstacle detected for the robot

Calculations to prove that shortest distance in upper quadrant of robot will only be Table 6:

Since the centre of LiDAR is where the distance is 0, we take it as our point of reference for the robot.

$$\text{Distance of centre of LiDAR from front} = 170mm$$

$$\text{Distance of centre of LiDAR from side} = 89mm$$

$$\text{Distance of centre of LiDAR from corner of Table 6 Zone} = \sqrt{170^2 + 89^2} \approx 192mm$$

(Rounded up for more buffer)

Calculating Maximum Distance of Table 6 from Robot:

$$\text{Minimum Radius of Table} = 270/2 = 135mm$$

$$\text{Length of Table 6 Zone} = 1000m$$

$$\text{Distance of Centre of Table from corner of Table 6 Zone} = \sqrt{2 * 135^2}$$

$$\text{Maximum Distance of Table from corner of Table 6 Zone (Rounded up for more buffer)} = \sqrt{2 * 1000^2} - \sqrt{2 * 135^2} - 135 \approx 1089mm$$

$$\text{Maximum Distance of Table 6 from Robot} = 1089 + 192 = 1281mm$$

Calculating Maximum Distance of front and side from Robot:

$$\text{Distance of wall from corner of Table 6 Zone} = 1340mm$$

$$\text{Distance of wall in front of Robot} = 1340 + 170 = 1510mm$$

$$\text{Distance of wall to the left of Robot} = 1340 + 89 = 1429mm$$

Hence, shortest distance in the upper quadrant of the robot will always be Table 6 since:

$$1281mm < 1429mm < 1510mm$$

After detecting Table 6, the robot then rotates towards it and moves closer until it's within 15 cm of the table. Upon retrieval of the can, the robot rotates the same angle but in the opposite direction and uses the same navigation method outlined in [Section 3.4.2](#) to return to the dispenser.

```

# function to locate table 6
# table 6 is located at the top left quadrant of the robot
# robot navigates to it by finding the shortest distance in the quadrant and
rotating to that angle, then proceeds towards it
# robot then waits for the limit switch to be depressed
# then returns to its study position to its starting position
def locate_table6(self, starting_angle, ending_angle):
    angle = np.nanargmin(self.laser_range[starting_angle:ending_angle])

    hypotenuse = self.laser_range[angle]

    self.get_logger().info('Table located: %d %f m' % (angle, hypotenuse))

    self.rotatebot(angle)

    self.stop_at_table('front')
    self.stopbot()

    rclpy.spin_once(self)
    while self.bot_limit == 1:
        print(self.bot_limit)
        self.get_logger().info('Waiting for can to be picked up')

        rclpy.spin_once(self)

    self.rotatebot(-angle)

```

Figure 3.25: Code snippet of our `locate_table6()` function for searching Table 6 in [r2checkpoint_nav.py](#)

3.5. Operational Setup

The setup for the mission run will be discussed in this section. For more details, refer to this [Github link](#) for all the required files and more details.

3.5.1. Preparation

3.5.1.1. Installation

1. Follow the instructions here to setup ROS 2 Foxy on laptop and TurtleBot.
(You may stop once you are able to teleoperate the TurtleBot from your computer.)
<https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/>
2. Follow the instructions here to download, install and setup MQTT X on your laptop.
(We are using a cloud-based MQTT desktop client for this project (MQTT X) for better consistency.)
<https://github.com/emqx/MQTTX/blob/main/docs/manual.md>
Refer to our project configurations [here](#)
3. Follow the instructions here to setup ESP32 software.
<https://esp32io.com/tutorials/esp32-software-installization>
4. Follow the instructions here to setup Arduino IDE on your laptop.
<https://www.arduino.cc/en/software>
Install the follow libraries on your Arduino IDE:
 - PubSubClient by Nick O'Leary
 - EspMQTTClient by Patrick
 - Keypad by Mark
 - ESP32Servo by Kevin
 - ezButton by ArduinoGetStarted.com
5. Follow the instructions here to setup Paho-MQTT on your laptop.
<https://pypi.org/project/paho-mqtt/>
6. Refer to our [/Hardware/Electrical Circuits](#) folder in our GitHub repository for the schematics of the circuits that we use to replicate our robot system.

Important: Please note that our project has been developed using Ubuntu 20.04.4, ROS2 Foxy, Arduino 2.1.0 and Python 3.6, as well as a DOIT ESP32 Devkit V1 Board and a Robotis Co. TurtleBot3 Burger with Raspberry Pi 3B+. As such, modifications may be necessary when using other software systems or hardware platforms.

3.5.1.2. Setup

To setup your Ubuntu Machine, clone our GitHub repository into your Home directory. Compile the workspace.

```
cd colcon_ws/src/auto_nav
git clone https://github.com/yinheng996/r2table_nav.git
cd colcon_ws
colcon build
```

To setup your TurtleBot, copy the `/py_pubsub` folder onto the Raspberry Pi. Compile the workspace.

```
ssh ubuntu@<RPi IP address>
scp -r <path to r2table_nav directory>/py_pubsub ubuntu@<RPi IP address>:~/turtlebot_ws/src
cd turtlebot3_ws
colcon build
```

To setup the ESP32, we will use the Arduino IDE.

- i. Open the code for the ESP32 in the Arduino IDE. This will be located in the repository that you cloned in step 1. In the code, make any necessary changes or modifications.
- ii. Connect the ESP32 to your computer using a USB cable.
- iii. In the Arduino IDE, select the correct board and serial port. To do this, navigate to `Tools -> Board` and select `ESP32 Dev Module`. Then, navigate to `Tools -> Port` and select the correct serial port.
- iv. To upload the code onto the ESP32, click on the `Upload` button in the Arduino IDE and hold down BOOT button on esp32, release the button once u see "connecting" in the serial monitor.
- v. Once the upload is complete, disconnect the ESP32 from your computer and power it using an external power source. The code should now be running on the ESP32.

To setup MQTT X, launch the MQTT X Desktop Client.

- vi. After the connection to ESP32 is successful, click the `New Subscription` button in the lower left corner to add New Topics.
- vii. Add the topics `docking` and `table_num`

3.5.2. System Check

Before you start using the robot, make sure everything is running properly by using `r2factory_test.py`. Follow these instructions to check the system:

For your TurtleBot

In one terminal

```
ssh ubuntu@<RPi IP address>
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

In another terminal

```
ssh ubuntu@<RPi IP address>
cd <path to r2table_nav directory>/py_pubsub
python3 limit_switch.py
```

The terminal should start publishing the limit switch state, press and release the limit switch and check if the publisher updates the limit switch state. If not, reboot your TurtleBot and repeat the TurtleBot system check.

For your Laptop

In one terminal

```
cd <path to r2table_nav directory>/table_nav/table_nav
python3 r2factory_test.py
```

Follow the instructions printed on your terminal, and if everything works out fine, it means the system is ready to go.

For your ESP32

- Launch the ESP32 factory test scripts in our `/ESP32_test` folder on our GitHub repository onto your Arduino IDE, then upload and run on your ESP32 module.
- Do this individually for each script.
- If your ESP32 module passes all the tests, it is ready to go.

For your MQTT X

- Launch the MQTT desktop client.
- Check for messages received from the ESP32 Publisher. If nothing is received, refresh the desktop client.

3.5.3. Operation Parameters

From lines 15 to 39 of /table_nav/table_nav/r2tcheckpoint_nav.py

The explanation of all the parameters are as commented below. These are the values which we found work the best for us in our use case. Please change these values if you require a different behaviour of the robot.

```
# constants
rotation_speed_fast = 0.5 # speed of fast rotation, mainly for regular
turns
rotation_speed_slow = 0.2 # speed of slow rotation, mainly for calibrations
moving_speed = 0.18 # speed of linear movements
lidar_offset = 0.193 # distance from lidar to front of robot
lidar_offset_b = 0.094 # distance from lidar to back of robot
disp_from_wall = 0.50 # distance from centre of dispenser to wall
stop_distance = 0.1 # distance to stop from Table with front facing table
stop_distance += lidar_offset
stop_distance_b = 0.1 # distance to stop from Table with back facing table
stop_distance_b += lidar_offset_b

centre_of_rotation_f_offset = 0.152 # distance from centre of rotation to
front of robot
centre_of_rotation_b_offset = 0.13 # distance from centre of rotation to
back of robot

move_dist_f = lidar_offset + 0.14 + 0.04 + 0.03 # movement offset at speed
0.18 from front
move_dist_corf = move_dist_f - centre_of_rotation_f_offset # movement
offset at speed 0.18 from centre of rotation from front
move_dist_b = lidar_offset_b + 0.13 + 0.04 + 0.04 + 0.03 # movement offset
at speed -0.18 from back
move_dist_corb = move_dist_b - centre_of_rotation_b_offset # movement
offset at speed -0.18 from centre of rotation from back

angle_sweep = 30 # angle to sweep for Table
front_angles = range(-angle_sweep, angle_sweep+1,1)
back_angles = range(180-angle_sweep, 180+angle_sweep+1,1)
scanfile = 'lidar.txt' # file to store lidar data logged
mapfile = 'map.txt' # file to store map data logged
```

From lines 41 to 47 of /table_nav/table_nav/r2tcheckpoint_nav.py

Please input your MQTT configurations, especially your username and password.

```
# MQTT variables
mqtt_broker = "broker.emqx.io"
mqtt_port = 1883
mqtt_username = "idpgrp3"      #username for mqtt broker
mqtt_password = "turtlebot"    #password for mqtt broker
mqtt_topic_tablenum = "table_num" # topic to subscribe to for table number
mqtt_topic_docking = "docking" # topic to subscribe to for docking
```

Running the Code

A total of 3 terminals would be required to run the code, in addition to the MQTT X desktop client. Ensure that Wifi connection is established and the ESP32, Laptop and TurtleBot are all in the same Wifi connection.

In Terminal 1: To bring up TurtleBot3

```
ssh ubuntu@<RPi IP address>
roslaunch turtlebot3_bringup turtlebot3_robot.launch
```

In Terminal 2: To host Bot Limit Switch Publisher

```
ssh ubuntu@<RPi IP address>
cd <path to r2table_nav directory>/py_pubsub_robot
python3 limit_switch.py
```

In Terminal 3: To run navigation code

```
cd <path to r2table_nav directory>/table_nav/table_nav
python3 r2tcheckpoint_nav.py
```

In MQTT X:

Launch MQTT X and select `connect`.

The system is ready to go. Press a number from 1-6 on the keypad for Food Delivery.

4. Final Design Fabrication

4.1. Dispenser

4.1.1. Mechanical Fabrication

After finalising the design, the next step would be to fabricate the dispenser using appropriate materials and machining techniques. **Figure 4.1** shows the CAD drawing of the dispenser assembly. The necessary components for fabrication are the Dispenser Carrier, Profile Bar Base, and Docking Assembly, which are depicted in subsequent figures with their corresponding parts.

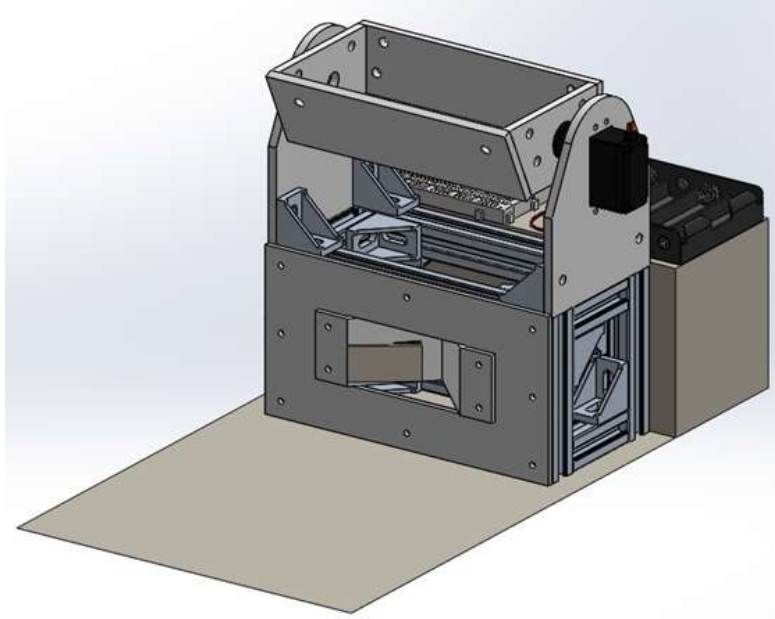


Figure 4.1: Overall Dispenser Assembly

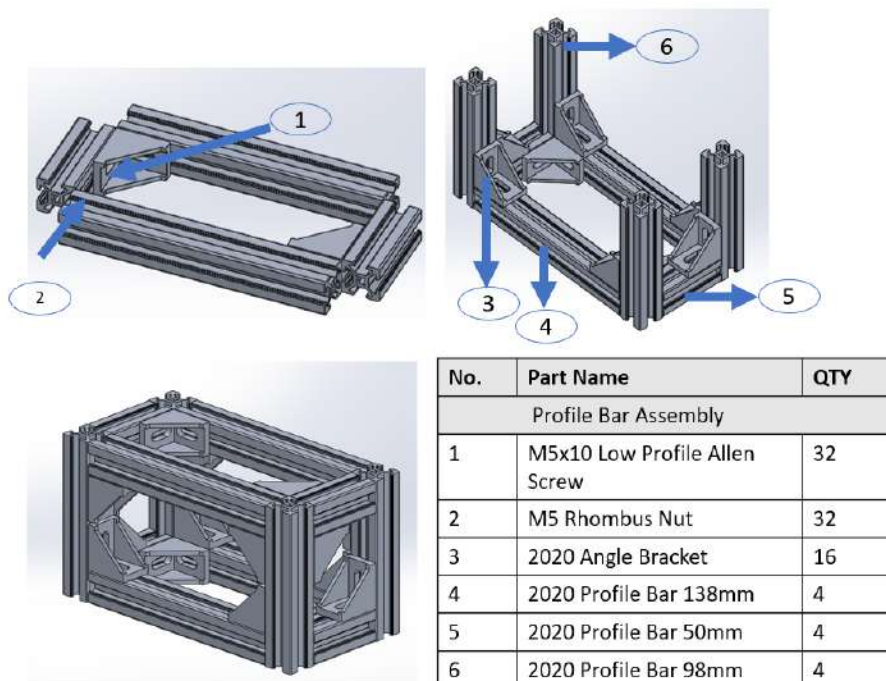


Figure 4.2: Profile Bar Base Assembly

The Dispenser Carrier plates are laser-cut from 5mm thick acrylic and assembled using glue. The MG996R servo motor is attached to the carrier using M1x16 screws for the tilting mechanism.

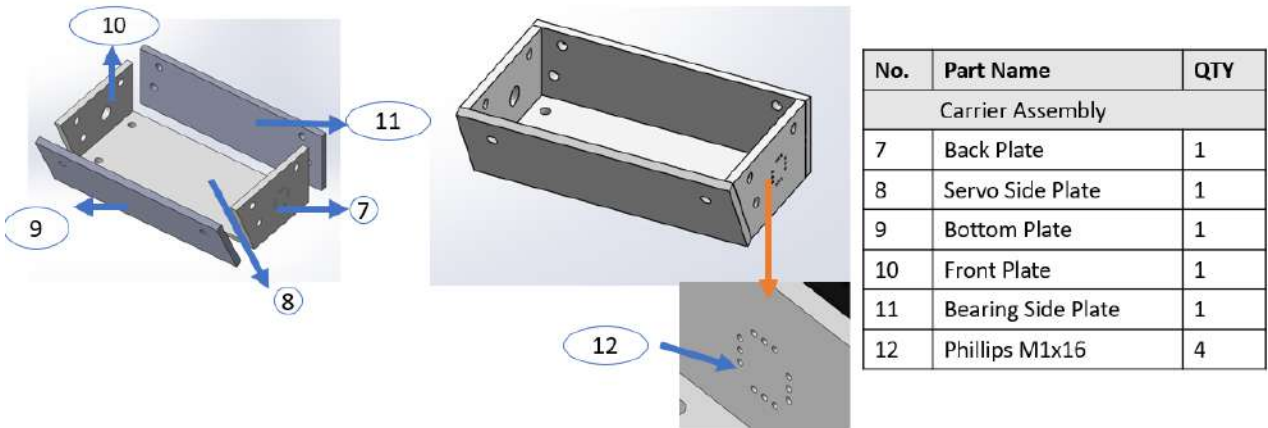


Figure 4.3: Dispenser Carrier Assembly

The Docking plate is made of 3mm acrylic, while the Docking Angle and Limit Switch mount are 3D printed. To enhance the limit switch sensitivity to the Delivery Robot's docking, a Homing Adapter made of cardboard is taped in front of it.

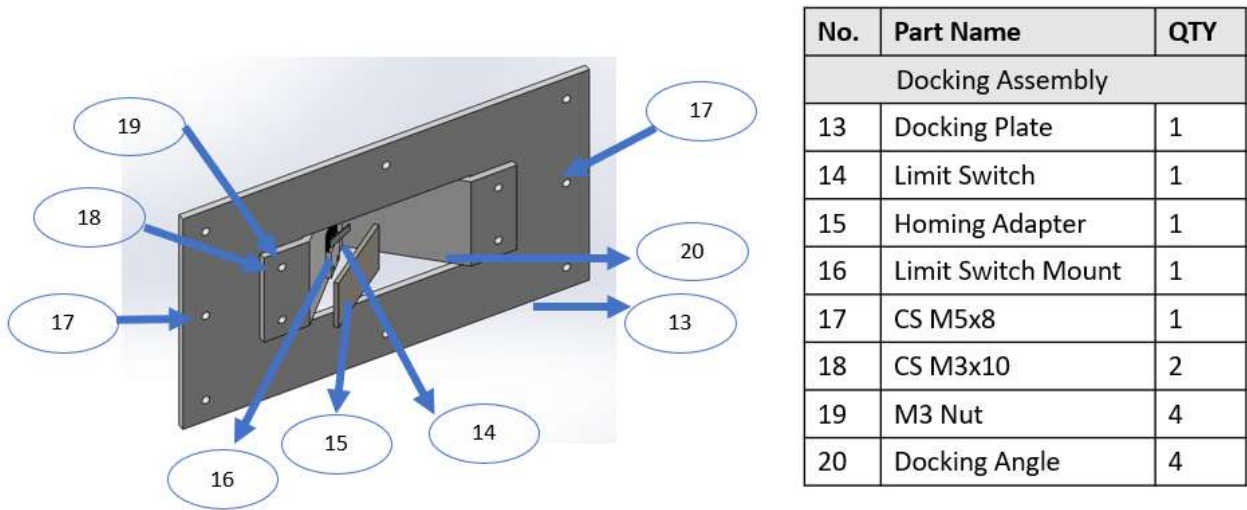


Figure 4.4: Docking Assembly

Through laser cutting as well, the two side supports are cut using a 5mm thick acrylic piece, whereas the bearing washer is from a 3mm acrylic piece.

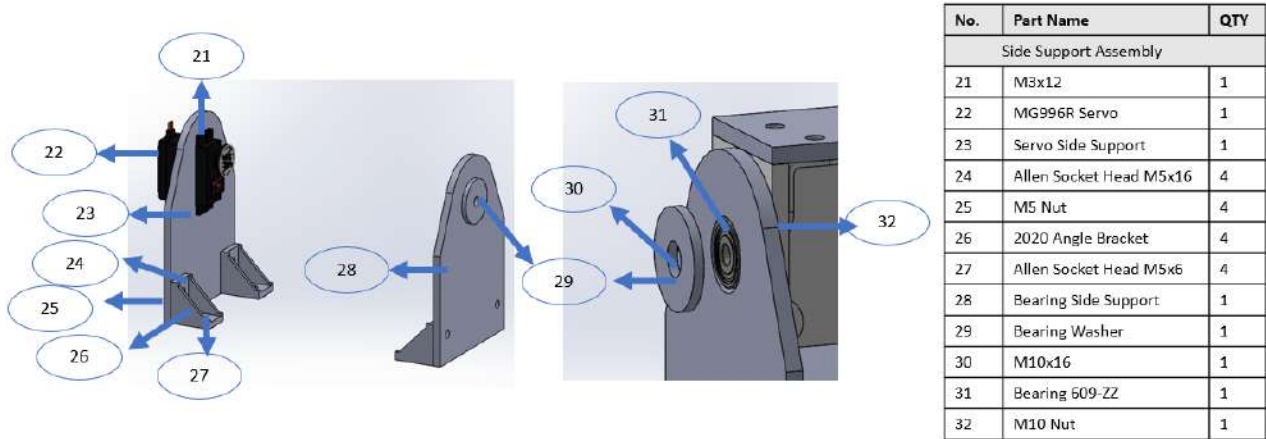


Figure 4.5: Side Supports Assembly

The electronics platform is made out of a cardboard box. Electronic components are then taped down onto the platform, which is then taped to the profile bar base assembly.

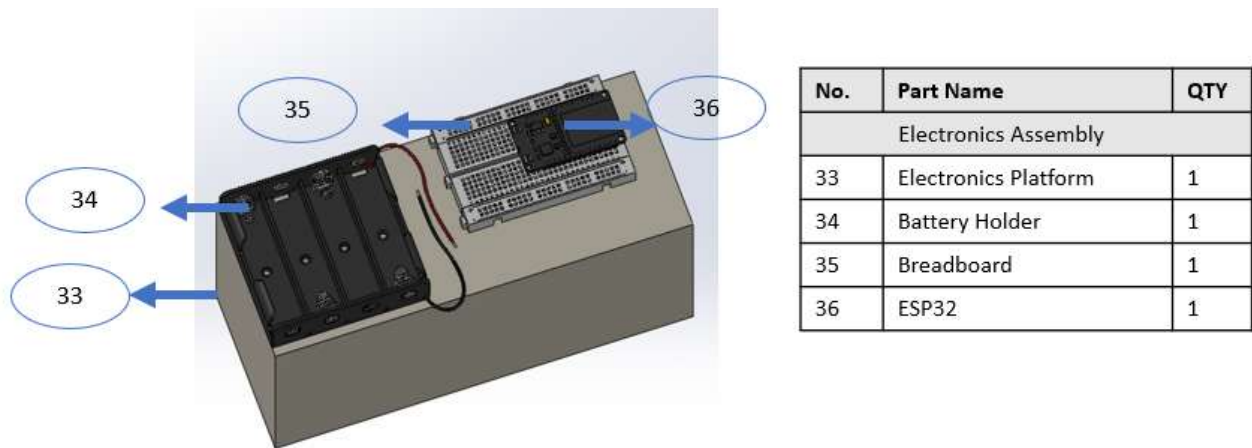


Figure 4.6: Electronics Assembly

Additional Information about Fabrication:

1. All laser cut (DXF Files) and 3D printed (STL Files) components are available on our [Github](#) for download.
2. All other components are commercially sourced. Please refer to our BOM for more details.

4.2. Delivery Robot

4.2.1. Mechanical Fabrication

The **Delivery Robot** is built using the **Turtlebot 3 Burger** as a base. The manufacturer should assemble a working Turtlebot3 Burger in accordance with the **Turtlebot3 Burger Assembly Manual**¹ before proceeding to assemble and install the custom made **Can Carrier Assembly** onto the robot.

The **Can Carrier Assembly** is an acrylic box mounted to the front of the robot. The base of the box is screwed to the lowest-level waffle plate on the Turtlebot using **8x M3** screws. A **ball caster wheel** is secured to the base using **2x M3** screws. A **limit switch** is secured at the base of the box via a **Limit Switch mount** using **4x M2** screws. The **V-shape Homing Angle** (Docking Angle) is mounted on the side of the acrylic box facing forwards, using **4x M3 screws**. Lastly, the **Infrared Sensor Modules** are fixed to the sides of the box using duct/masking tape.

The **Exploded View** on the **next page** shows the Delivery Robot, with Turtlebot 3 Burger Main components labelled in **red**, and components belonging to the Can Carrier Assembly marked in **blue**. The locations of screws and fasteners to be used on the Can Carrier Assembly are marked in **green**.

Additional Information about Fabrication:

1. The Acrylic Can Carrier consists of 5 laser-cut acrylic pieces, fixed together at the joints using acrylic glue. The DXF files are available on our [Github](#) for download.
2. The Limit Switch Mount and V-shape Homing Angle are 3D printed plastic components. The STL files are available on our Github for download.
3. The Ball Caster may result in the Robot tyres being lifted slightly above the ground, impeding movement of the robot. Should that occur, add 1-2 layers of hex nuts below the acrylic can carrier (At the screw holes #14) to prop the carrier up such that the ball caster no longer lifts up the tyres.
4. All other components are commercially sourced. Please refer to our BOM for more details.

¹ https://emanual.robotis.com/docs/en/platform/turtlebot3/hardware_setup/

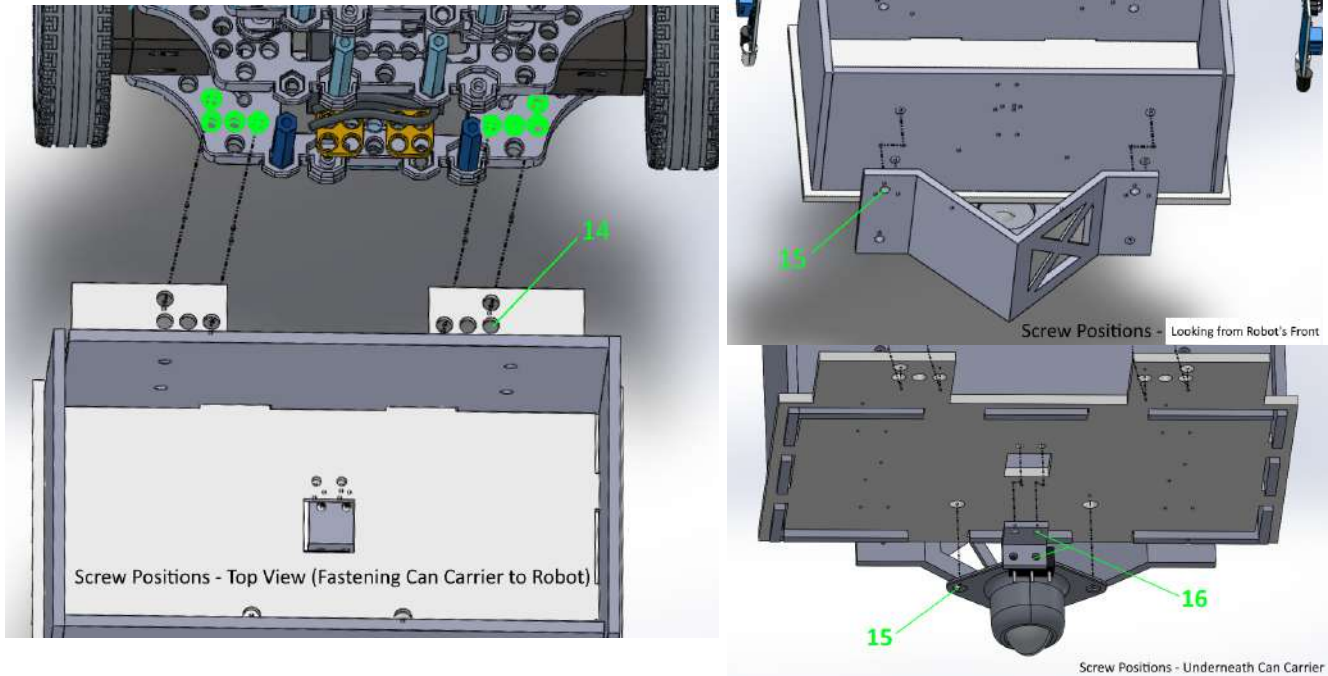
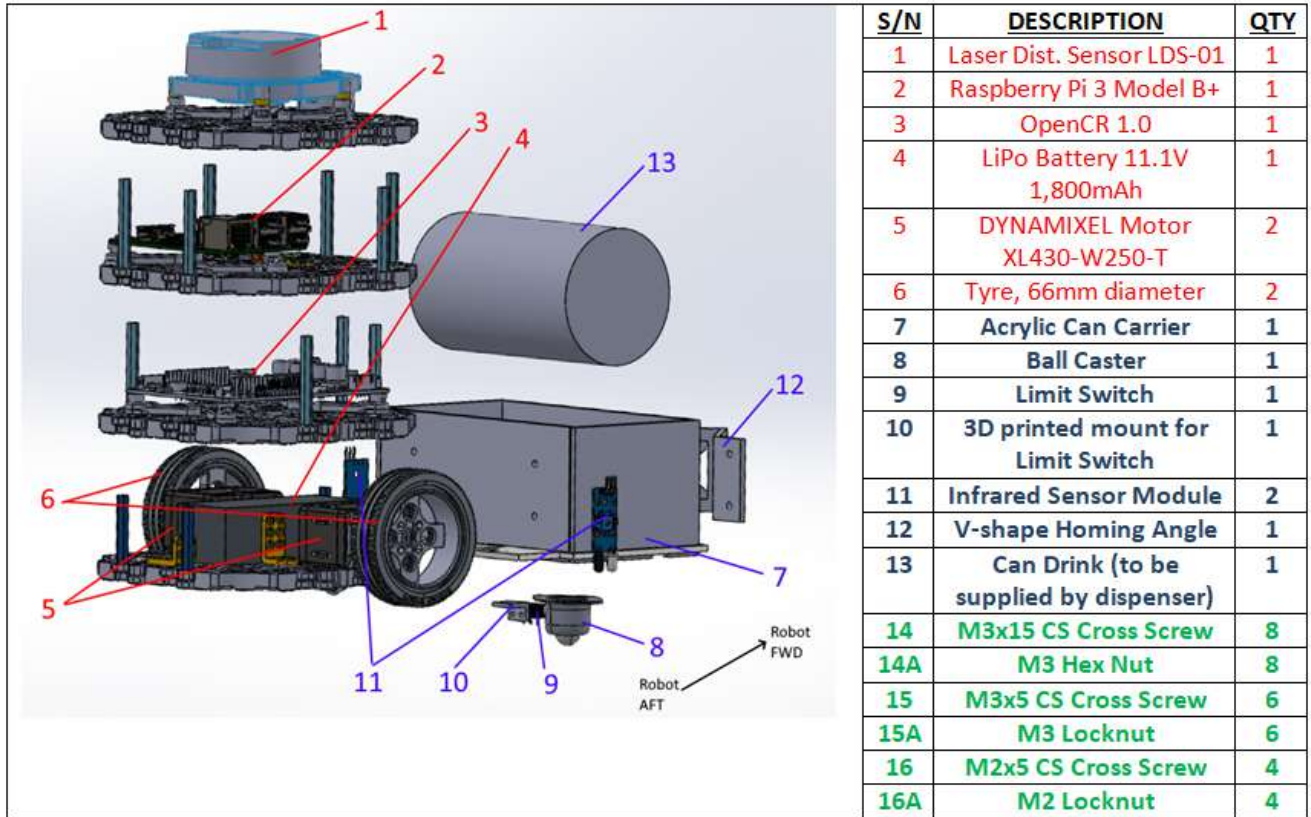


Fig 4.7. Exploded View of Delivery Robot and Illustration of Screw Positions

4.3. Bill of Materials

For our project, some hardware components were provided to us, such as a complete Turtlebot set (including RPi, OpenCR, Lidar etc.), and other miscellaneous hardware available in the Electronics Lab such as screws, nuts, washers and bolts. Such components are not included in our list.

There were also components that we were supposed to purchase for our Can Carrier and Dispenser, but were available in the Electronics Lab. These components are listed in the table, together with their market price, however the total cost to our team for these components will be zero.

Note all costs of fabrication (laser cutting, 3D printing, etc) are not included in this list.

S/N	Type of Component	Model Name	Qty	Unit Price /\$	Subtotal /\$	Supplier	Provided by NUS? (Y/N)	Total Cost to our Team
1	Microcontroller	Espressif ESP32 WROOM 32	1	5.51	5.51	Shopee	Y	0
2	Servo Motor	Mg996-R	1	4.88	4.88	Shopee	Y	0
3	Limit Switch	D2F-01L	2	0.65	1.30	Shopee	Y	0
4	Numeric Keypad	-	1	6	6	Shopee	Y	0
5	Ball Bearings	626 ZZ	3	0.96	2.88	Shopee	Y	0
6	Profile Bar Fasteners	20x20 L brackets & T slot nuts	2 sets	18	36	Shopee	Y	0
7	Angle Brackets	20x20 Angle Brackets	2 sets	2	4	Shopee	Y	0
8	Profile Bars	20x20 Profile Bars	2m	6.47 (per 400 mm)	32.85	Shopee	Y	0
9	Ball Caster	RBT-02194	1	1.2	1.2	Sgbotic	Y	0
10	IR Sensor	Flying Fish	2	1.3	2.60	Shopee	Y	0
11	AA Batteries	-	4	0.59	2.36	Value\$	N	2.36
TOTAL					99.58			2.36

5. Design and Development (Concept → Preliminary Design)

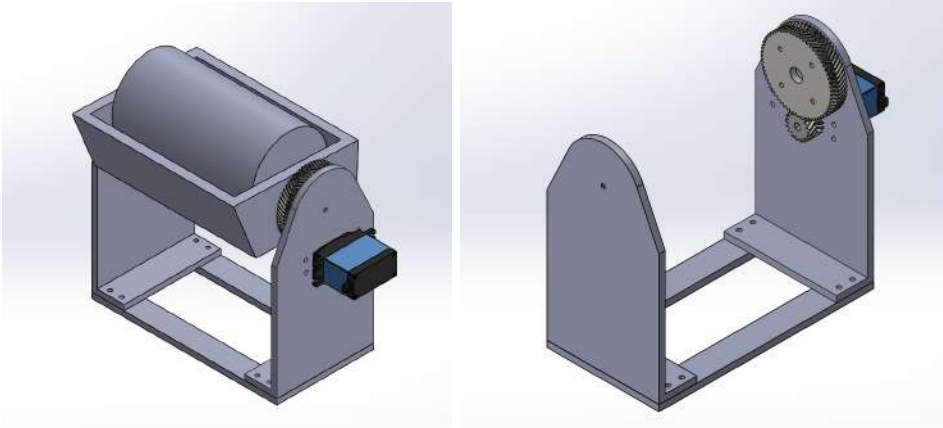
The project design went through three phases: **Concept**, **Preliminary**, and **Critical**. Initial design concepts were developed during the **Concept Design** phase and detailed in the **G1 report**, which were refined further during the **Preliminary Design** phase, taking into account practical considerations of building the system tight during lectures and TA consultations. The **Critical Design** phase resulted in the final design used in the project's final mission. **In this section, we will focus on the Concept and Preliminary Design** phases, while the next section will cover Testing and Validation and detail the development leading up to the Critical Design. The Appendix contains both full designs.

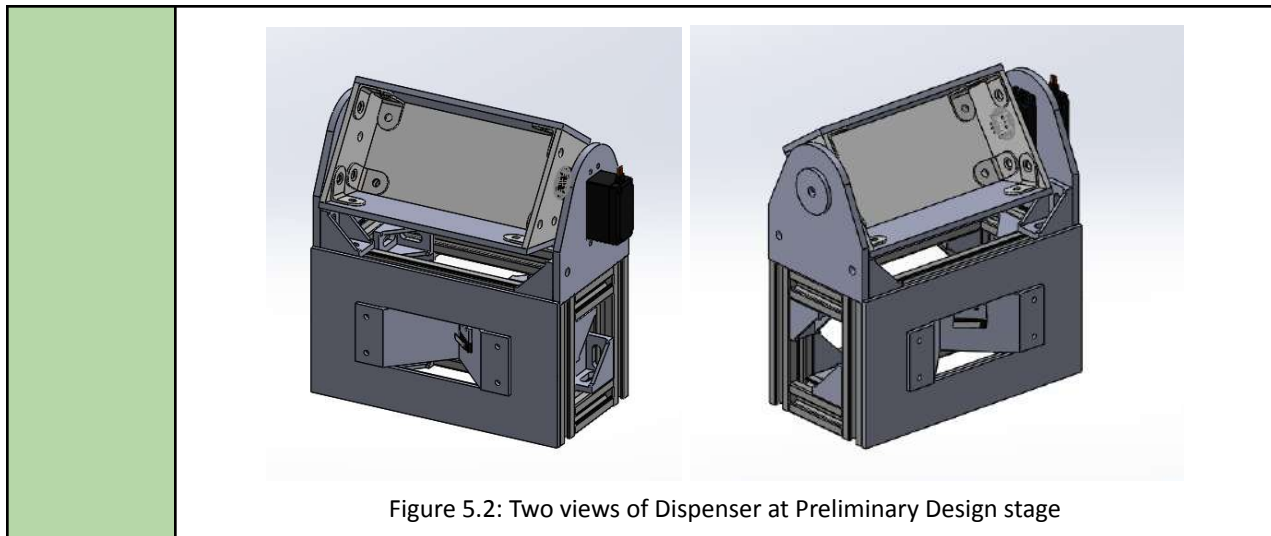
5.1. Object Transfer Mechanism

5.1.1. Dispenser

With reference from Literature Review in Appendix 1, the following table includes three possible methods of can transferring mechanism with a final judgement on why the Tilting Bucket Dropping mechanism is chosen at Concept Design stage. The flow of design development across the Concept and Preliminary design phases is then presented in a table format below.

	Gate (Rolling Mechanism)	Tilting Bucket (Dropping Mechanism)	Pusher (Pushing Mechanism)
Pros	<ul style="list-style-type: none">- Relies on gravity so it's not very hard to execute	<ul style="list-style-type: none">- More control over the falling of the can.- Space efficient due to compact design.	<ul style="list-style-type: none">- Easily scalable to many cans by changing length of pushing rack gear.- Low load on the servo as it relies on cans rolling.
Cons	<ul style="list-style-type: none">- Not space efficient as it requires a channel to allow the can to roll down.- Possible misalignment of rack and pinion system due to weight of the cans on the gate.- Not easy to scale for many cans as it would need precise and fast opening and closing of the gate.	<ul style="list-style-type: none">- Not simple to upscale for multiple cans because the bucket must push up the extra cans while it drops one can.- Heavier load on servo as the servo needs to transfer sufficient torque to turn the bucket and drop the can.	<ul style="list-style-type: none">- Less control of cans and possible accidental rolling of cans.- Not space efficient because the rack must extend at least the length of a diameter of the can forward and then backwards in each cycle of pushing the can.
Final Judgement	<p>Tilting Bucket idea is selected. It was adapted from the Pan & Tilt camera because it was the most space efficient and reliable method. It does not require a big model to be effective and there is less chance of an accidental dropping of the can since the servo must turn to drop it.</p> <p>The can of soda is about 350g. To solve the issue of load on the servo, the chosen motor should be able to support this weight along with a safety factor to consider for the energy loss in torque transfer between the gears and bucket.</p>		

	Object Transfer Mechanism		Can Carrier (Dispenser)
Concept Design	Using spur gears to attach to servo motor for bucket tilting	3D printed side supports & carrier	Carrier is propped high enough by using long side supports .
	 <p>Figure 5.1: Two views of Dispenser at Concept Design stage</p>		
Reason for change	The load of the can is not too heavy and gears are not needed to turn the loaded carrier of the dispenser. A servo motor with sufficient load capacity would work.	Given the large size of the carrier and side supports, the estimated print time was over 10 hours per part. This is inefficient.	Higher load and shear force on the L-shape Side supports which can cause flexing of the side supports which could cause them to break under the weight of the cans after a couple of uses.
Preliminary Design	Gears are removed. Calculated torque required to turn the carrier, with a safety factor of 2, was 0.14Nm, which is well below the maximum torque of 11 kg-cm (1.078 Nm) at 6V provided by the chosen servo, MG966R . Bearing is also chosen according to load conditions see calculation in Appendix 3 .	Side support made of laser-cut acrylic and supported by angle brackets. Carrier made using acrylic pieces joined together with L brackets.	Profile bar base to raise up the dispenser and reduce the shear force on the side supports.



5.1.2. Robot Carrier

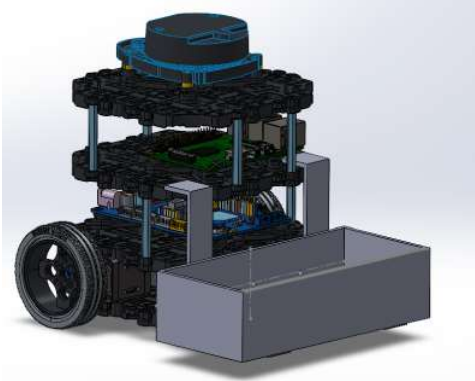
The following table includes the possible orientation of the can as it is dropped into the robot's carrier, and the final judgement on why the Lying-down Orientation is chosen at Concept Design stage. This affects the sizing of the carrier to be fabricated and thus, the mounting methods of it to the robot itself.

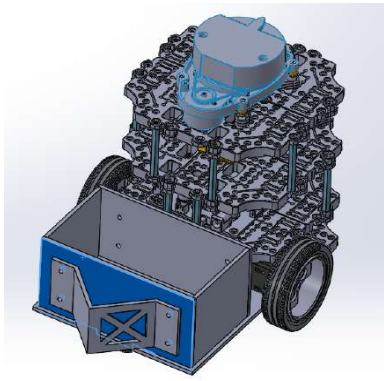
	Lying Down	Upright
Pros	- Easier for the can to drop into the holder from the dispenser.	- More compact design. - Easier to pick up the can from the tray.
Cons	- Slightly harder (though not impossible) to pick the can up from the tray.	- Difficult to drop the can accurately into the holder. - Higher chance of failure.
Final Judgement	Lying-down orientation was chosen, as it offered an easier method for dispensers to drop the can into the robot with the lowest chance of failure.	

Orientation of Can	
Concept Design	Lying-down Orientation
Change needed?	No - No problems surfaced with orientation of the can when dropped at a height.
Preliminary Design	Lying-down Orientation

The following table includes the possible mounting position of the carrier itself to the TurtleBot, and the final judgement on why the bottom position is chosen at the Concept Design stage.

	Aft (bottom)	Above LIDAR	Between LIDAR & RPI
Pros	<ul style="list-style-type: none"> - Will not interfere with LIDAR operation. - The CG of the robot (with can) is lower, reducing likelihood of robot toppling. 	<ul style="list-style-type: none"> - Easy to reach by the TA as it is a higher position. 	<ul style="list-style-type: none"> - Will not interfere with LIDAR operation
Cons	<ul style="list-style-type: none"> - May block certain I/O ports on RasPi. - Extends slightly the length of robot rearwards 	<ul style="list-style-type: none"> - The CG of the robot (with can) is high, making the robot more unstable. 	<ul style="list-style-type: none"> - Need to disassemble/heavily modify the robot to install the holder. - Tray difficult to access.
Final Judgement	<p>The holder has been selected to be mounted at aft and low on the robot, as no modifications to the main body was required. A modular mounting design allows the easy removal of the holder to access any blocked I/O ports during maintenance.</p>		

Can Carrier (Robot)	
Concept Design	<p>The Concept Design called for a Z shape mount to be fixed aft of the robot, and a box that would be secured to the mount using a few screws, such that the box could be removed or installed as required.</p>
	 <p>Figure 5.3: Can Carrier for Robot at Concept Design stage</p>
Reason for Change	<ul style="list-style-type: none"> - Fabrication of the Z shape mount was deemed complex, requiring either acrylic bending (if laser cut) or 3D printing, where there were concerns over load-bearing. <u>Fastening the base (made out of a single piece of material) directly to the robot</u> was judged to be better capable of bearing the loads induced by the can drink. - By installing the box at the front of the robot instead, it would not block any critical components on the Turtlebot, and thus would not require frequent removal or installation.

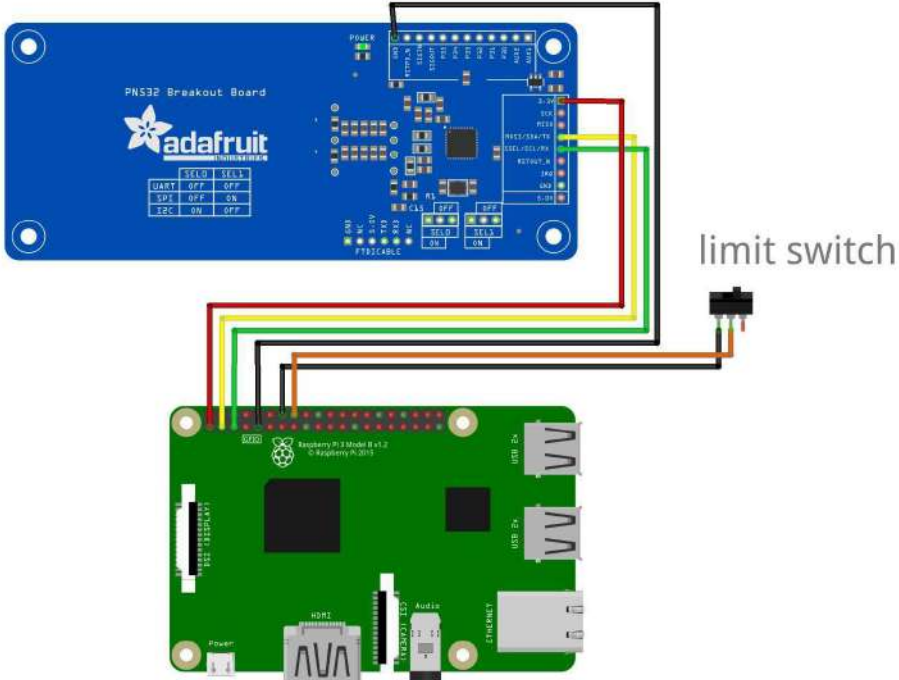
Preliminary Design	<p>We hence decided on a simpler design - a <u>large acrylic plate with holes to secure the plate to the lowest-level waffle plate</u> on the Turtlebot, with walls fixed onto the plate to create the box for the can drink. This would be installed on the front of the robot.</p> <p>*Note: Dimensions of the box was changed when the Mission Master provided updated dimensions of the can drink used in the Graded Run.</p>
	 <p>Figure 5.4: Can Carrier for Robot at Concept Design stage</p>

The following table includes the possible sensors to detect the presence of can in the robot's carrier, and the final judgement on why the limit switch is chosen at the Concept Design stage. This sensor serves as an indication of successful can transfer from the dispenser to the delivery robot for the program to flow.

	Electromechanical Sensor - Limit Switch	Weight/Load Sensor	IR Sensor
Pros	- Low cost and simple to use.	- Less likely to face interference from environmental factors	- Low cost
Cons	- Switch may become stuck in engaged position, causing false detection of can presence	- Generally, more expensive than other options (especially sensors that are highly sensitive to detect loads of 200-400g)	- May not work under bright lighting conditions. - Minimum distance required between can and sensor
Final Judgement	Limit Switch was chosen due to its low cost and simplicity, with a relatively low likelihood of creating errors on the software end.		

Sensor for Can Presence	
Concept Design	Limit Switch
Change needed?	<p>No</p> <ul style="list-style-type: none"> - No problems surfaced with the usage of limit switch as a sensor.
Preliminary Design	Limit Switch

5.1.3. Docking

Docking Verification	
<div>Concept Design</div>	<p>Docking via NFC Tag (Tag on Dispenser; Reader on Delivery Robot)</p> <p>PN532 Breakout Board NFC Reader</p>  <p>Figure 5.5: NFC Reader for Robot at Concept Design stage</p>
	<p>Reason for Change</p> <p>NFC tag cannot ensure that the Robot's carrier is aligned to the Dispenser' Carrier. It only confirms whether or not the robot has arrived. The robot may never reach the NFC tag accurately, or it may accidentally activate it in the wrong position.</p>
	<p>Preliminary Design</p> <p>Docking via Mechanical Means - Docking Homing Structure</p>

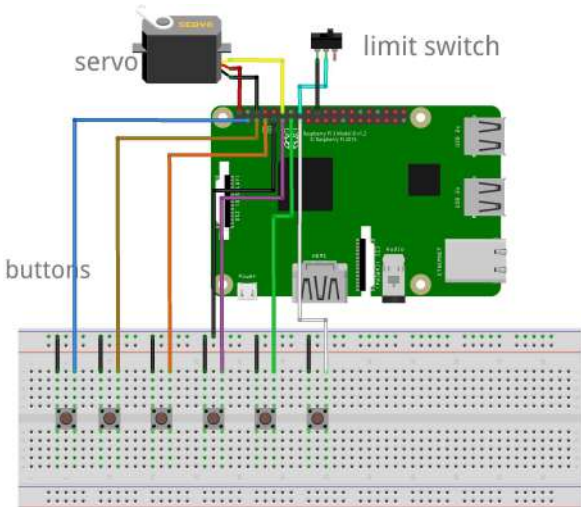
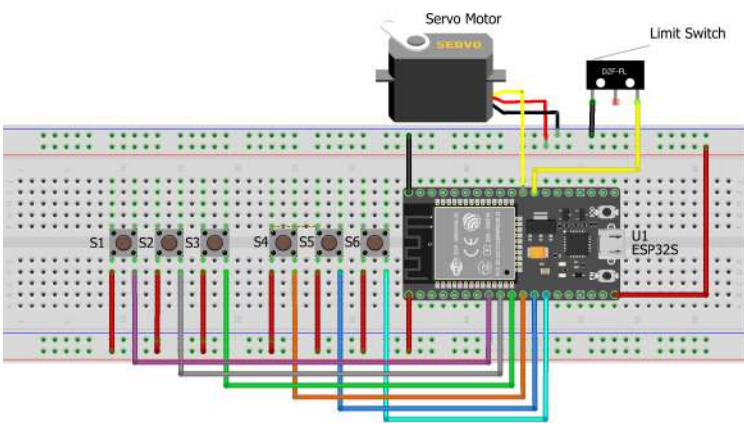
	<p>Hard-coded movement of the Robot to calibrate against the wall and move into the Dispenser's Docking Angle. Limit switch is used to validate whether the Robot has arrived as it is easier to integrate in the electrical architecture and does not need additional equipment such as an NFC scanner.</p> <p>Addition of Homing Angle on Robot's Carrier (see Figure 5.3 above) and Dispenser's Profile Bar Base for the robot to align itself to the dispenser. Limit switch is mounted on the Dispenser's Homing Angle, which allows the limit switch to be activated only when the robot is properly docked and aligned.</p>
--	---

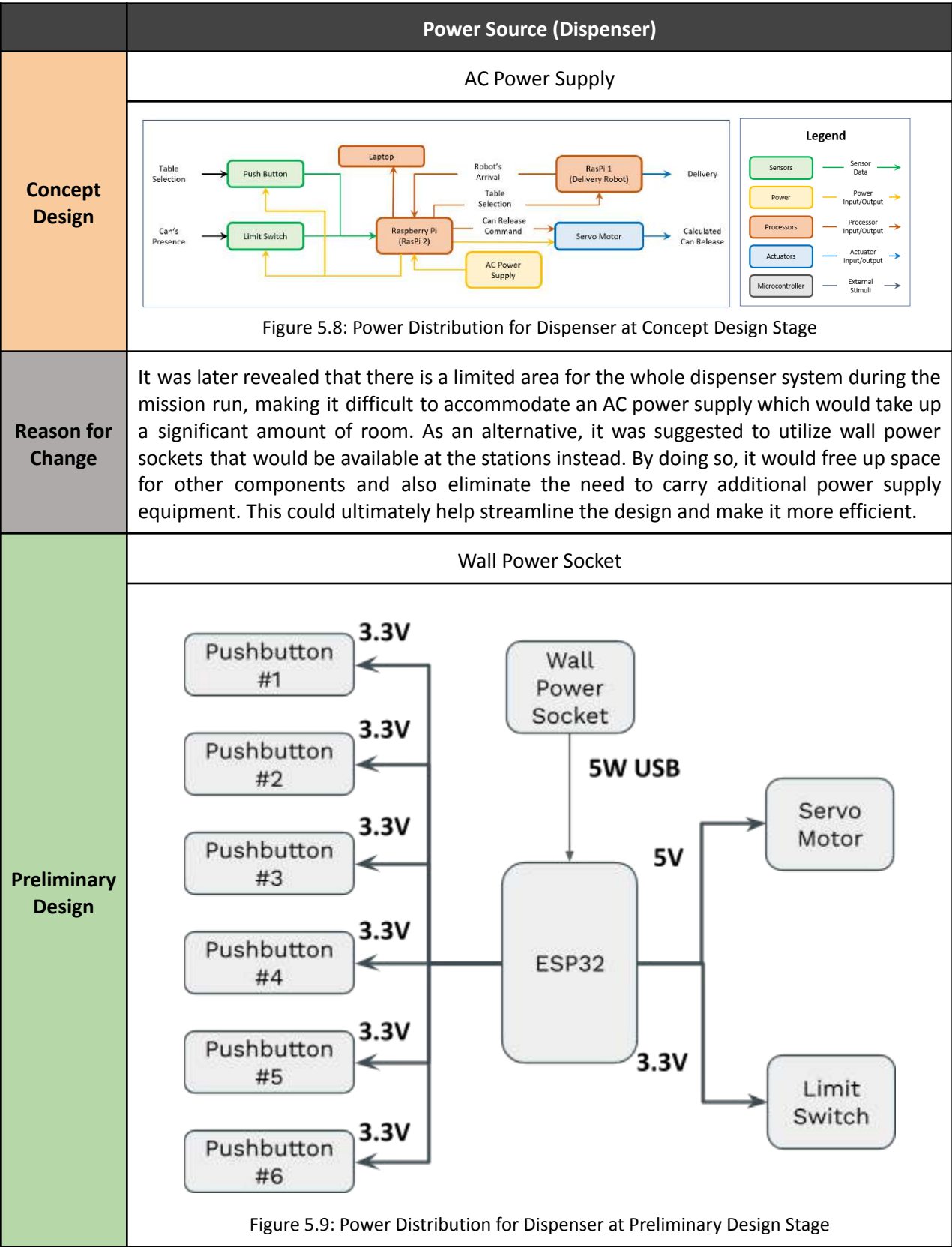
5.2. Communication Protocols

5.2.1. Electrical Architecture

The following table includes the possible microcontroller boards to control the electrical components on the dispenser, and the final judgement on why ESP32 is chosen at the Concept Design stage.

	Arduino MKR WiFi 1010	Arduino Nano 33 IoT	ESP32
Pros	<ul style="list-style-type: none"> - Has a built-in WiFi module and antenna, making it easy to connect to WiFi networks. - Small and compact form factor. - Runs on a low-power ARM Cortex-M0+ processor, making it ideal for battery-powered projects. - Compatible with Arduino IDE. - Uses a simplified version of C++. 		<ul style="list-style-type: none"> - Dual-core processor, higher computing power, lower power consumption. - Has a wider range of peripherals and interfaces, including I2C, SPI, UART, ADC, and PWM. - Compatible with a wide range of programming languages i.e. Python, C++, Java. - Has built-in WiFi and Bluetooth modules. - A large number of GPIO pins (up to 34). - Compatible with the Arduino IDE. - Offers a variety of development boards with different features and form factors. - Support for more complex communication protocols (e.g., Ethernet, CAN).
Cons	<ul style="list-style-type: none"> - More expensive compared to other Arduino boards. - Limited number of pins (22), making it difficult to use for larger projects. 	<ul style="list-style-type: none"> - Limited number of pins (21). - Slightly more expensive compared to other Arduino boards. 	<ul style="list-style-type: none"> - More complex programming compared to Arduino boards.
Final Judgement	<p>ESP32 would be most suitable due to its cheap cost and easy accessibility. Because of its dual-core processor, it is faster in multitasking multiple tasks simultaneously as well, which could be an advantage if other command tasks are required in the dispenser in future.</p>		

	Microcontroller (Dispenser)
Concept Design	Raspberry Pi
	 <p>Figure 5.6: Raspberry Pi for Dispenser at Concept Design stage</p>
Reason for Change	<p>During the planning phase of the project, it was initially assumed that Raspberry Pis would be the primary microcontrollers used for the dispenser and delivery robot systems. However, as the project progressed, we realised that the cost of an additional Raspberry Pi would be a significant factor in the overall budget of the project. Furthermore, the backup Raspberry Pi that was available and planned to be used before was intended solely for the use of the delivery robot. Given these factors, the team decided to use Arduino microcontrollers would be a more economical option that would also be easier to set up and program. This decision allowed the team to save on costs while still maintaining the necessary functionality and performance of the dispenser and delivery robot systems.</p>
Preliminary Design	ESP32 WROOM 32
	 <p>Figure 5.7: ESP32 for Dispenser at Preliminary Design stage</p>



5.2.2. Inter-system Communication

The following table includes the possible inter-system communication modes between the dispenser and the delivery robot, and the final judgement on why MQTT is chosen at the Concept Design stage.

ESP32 & RasPi	Serial Communication	Wireless Communication	
		MQTT	Shared Memory
Pros	<ul style="list-style-type: none"> - Less complicated to program. 	<ul style="list-style-type: none"> - Do not need physical wiring since both boards are on separate systems. - Useful for small amounts of data so that conflicts need not occur. - Easy to implement. 	<ul style="list-style-type: none"> - Do not need physical wiring since both boards are on separate systems. - Deemed as faster for high volume of data passing than Message Queuing. - Low overhead.
Cons	<ul style="list-style-type: none"> - Need physical wiring connection. 	<ul style="list-style-type: none"> - Low volume of data. - Time consuming to set up communication links or connections between the processes. 	<ul style="list-style-type: none"> - If both processes try to write to the shared memory region at the same time, the result would be unpredictable and could lead to errors in one or both processes.
Final Judgement	<p>Based on the chosen microcontroller board (ESP32), it allows both WiFi and Bluetooth communication. Moreover, the two boards have to be away from one another since the RasPi (robot) has to be moving around the area. Since a WiFi network has already been setup on the RasPi, WiFi connection would be used as the communication tool between these boards.</p> <p>As our communication does not require a high volume of data to be transferred in addition to us being more familiar with the Message Queue protocol as compared to Shared Memory and the rest, Message Queuing would be used as the communication protocol between these boards.</p>		

Inter-System Communication Protocol	
Concept Design	MQTT
Change Needed?	<p>No</p> <ul style="list-style-type: none"> - No tests have been done yet.
Preliminary Design	<p>MQTT</p> <p>Using the ESP32 on the Dispenser as the publisher and the R-Pi on the Delivery Robot as the subscriber to communicate with each other through the MQTT broker (Mosquitto) on the R-Pi.</p>

5.3. Navigation

The following table includes the possible mapping algorithm to generate a map of the environment, and the final judgement on why cartographer is chosen at the Concept Design stage.

	GMapping	Cartographer
Pros	<ul style="list-style-type: none"> - Less computationally intensive. - Can handle partial observability using grid based FastSLAM algorithm. - Easier to configure. 	<ul style="list-style-type: none"> - Higher accuracy using real-time loop closure and scan matching. - Designed to handle highly dynamic environments. - Can handle multiple sensor inputs.
Cons	<ul style="list-style-type: none"> - Assumes the environment is static. 	<ul style="list-style-type: none"> - More sensitive to noise and error.
Final Judgement	<p>Cartographer is better suited for our mission due to its higher accuracy and ability to handle highly dynamic environments. While GMapping's ability to handle partial observability is useful, a higher mapping accuracy from Cartographer using real-time loop closure, scan matching and multiple sensor inputs is preferred. With potentially moving obstacles, GMapping's assumption of a static environment could lead to more uncertainties, whereas Cartographer would have little difficulties mapping it as part of a highly dynamic environment.</p>	

The following table includes the possible path planning algorithm, and the final judgement on why RRT-Connect is chosen at the Concept Design stage.

	A*Algorithm	RRT-Connect Algorithm
Pros	<ul style="list-style-type: none"> - Guarantees the shortest path. 	<ul style="list-style-type: none"> - Guarantees a working solution. - Can handle partially known environments. - Easier to configure.
Cons	<ul style="list-style-type: none"> - Requires a heuristic function. - Computationally expensive. - Could be trapped in local minima if the initial condition is bad. 	<ul style="list-style-type: none"> - Takes a long time to find a solution.
Final Judgement	<p>RRT-Connect is better-suited for our mission as we would need consistently workable solutions and flexibility in navigating dynamic environments. Since time is not a criteria in the mission, a working solution using RRT-Connect is more useful than an optimal solution from A* that might not work. RRT-Connect's ability to navigate in partially known environments also stood out as moving obstacles could create a dynamic environment for delivery.</p>	

The following table includes the possible table identification methods and the final judgement on why NFC tag is chosen at the Concept Design stage.

	ArUco marker	NFC Tag
Pros	<ul style="list-style-type: none"> - Able to scan multiple tables from far. - Predefined OpenCV library and existing ROS support. 	<ul style="list-style-type: none"> - Relatively cheaper. - Simpler in concept.
Cons	<ul style="list-style-type: none"> - Requires camera hardware which may be expensive. - High occlusion sensitivity (corners cannot be blocked). 	<ul style="list-style-type: none"> - Requires the robot to physically go near the tables to scan the NFC tags.
Final Judgement	NFC tag is decided as the table identifier, as its workflow is less foreign to us, unlike the ArUco Marker which requires new knowledge on OpenCV. This method will then double up as an indicator when the delivery robot is back in position at the dispenser.	

Robot Navigation Methodology	
Concept Design	Cartographer + RRT-Connect + NFC Tag
Reason for Change	Table 6 will only be placed in the map after setting up, which does not allow modifications from us such as placing of NFC tags onto the tables. We have decided that manually giving the robot instructions for movement is sufficient to navigate through the map as the exact dimensions of the map are given to us. Thus there is no need for a mapping and path planning algorithm which requires much more time and effort for us to execute, given that we do not have members who are very proficient in programming in our team.
Preliminary Design	Navigation by coded moving instructions for each respective table, with calibration to the walls and tables implemented for accuracy of movement within the test area.

6. Testing & Validation (Preliminary → Critical Design)

In this Section, we will detail the decisions made during the refinement process from **Preliminary Design to Critical Design** - specifically, the tests we conducted, our observations, and improvements made which resulted in the Final Design that was sent for the Final Run.

6.1. Dispenser Operation

6.1.1. ESP32

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Upload a simple existing test program (eg: Blink onboard LED) - Test if ESP32 board works	Onboard LED should blink when program is run	Onboard LED did blink when program is run	Pass	No Change
Test if MQTT connection works with broker software 'Mosquitto' installed on R-Pi	Broker can be setup ESP32 should connect to the broker	Broker cannot be setup on R-Pi, too complicated. ESP32 could not connect to the broker	Fail Fail	Change broker to one with more user-friendly interface: Desktop App 'MQTT X'
Test if MQTT communication works between ESP32, MQTT X broker & R-Pi (publish & subscribe simple messages)	Messages are published onto the broker successfully from ESP32 with R-Pi receiving the same message	Messages cannot be received on the broker from the ESP32 after a certain period of time as broker would disconnect from the client when no message is received for certain time	Fail	Code debugging for ESP32 - Add a reconnect() function before the publishing code (allow the ESP32 to reconnect to the mqtt broker each time it publishes.

6.1.2. Keypad & Limit Switch Activation

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Upload a simple existing limit switch test program - Test if the limit switch works	Serial.print "High" when activated, "Low" when not activated	Serial.printed "High" when activated, "Low" when not activated	Pass	No Change
Test if the robot is able to activate the servo motor while docking	Serial.print "Docked" when activated, print nothing when not activated	Constantly printing nothing even when limit switch is activated	Fail	Added limit switch.loop() into the code in every function for the status of the limit switch to be checked at all times.
Upload a simple existing keypad test program - Test if the keypad works	Serial.print the corresponding number keyed in on keypad	Serial.printed the corresponding number keyed in on keypad	Pass	No Change

6.1.3. Servo Motor

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Upload a simple existing servo motor sweeping continuously test program - Test if servo motor is working	Servo motor should turn continuously in a loop according to setup code	Constant twitching with occasional pauses was observed	Fail	Changed from using the 5V pin of the ESP32 to power up the servo to 4xAA batteries supplying 6V in total.
Test if servo motor is working after using external power supply	Servo motor should turn continuously in a loop according to setup code	Constant twitching with occasional pauses was observed	Fail	Changed ground connections between ESP32, Servo & Power Supply (all connecting to common ground)
Test if servo motor is working with proper connections	Servo motor should turn continuously in a loop according to setup code	Did not turn smoothly	Fail	For-loop code is removed, as it caused the servo to turn incrementally to the chosen degree, slowing down the motor which made it twitch continuously
Test if servo motor is working with proper code	Servo motor should turn continuously in a loop according to setup code	Did not turn smoothly	Fail	Changed the Pulse Width parameter - <code>servo.attach(SERVO_PIN, int_min, int_max)</code>


All motors have their own Pulse Width Parameters of how long should the High logic stay at high in terms of duty cycles. The defined parameter for “int min” and “int max” in the ESP32Servo.h library installed in the arduino IDE did not correspond to the correct minimum and maximum pulse width for the MG996R servo motor. Therefore, by using the values mentioned in this [article](#) describing the min and max parameters for the particular model we were using, After adjusting the values in the code, the servo motor starts to turn smoothly.

6.1.4. Integrated Testing

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Can placed on assembled dispenser where carrier is assembled without L brackets (only acrylic glue)	Acrylic glue would be strong enough to hold carrier assembly	No flexing or breakage was observed on the faces. Carrier holds can as expected.	Pass	L bracket not installed, only acrylic glue used
Dispenser's limit switch mounted to Dispenser's Docking Angle Robot's V-shape Homing plate mated with Dispenser's Docking Angle to verify it is able to activate the limit switch.	Limit switch activated when V-shape homing plate is mated to Docking Angle	Limit Switch activated	Pass	No change

6.2. Can dispensing, Robot Teleop and Docking

6.2.1. Mechanical Testing

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Drop test: Drop a drink can into the Robot's can carrier from a height of 10cm above the can carrier base. Repeat 50 times.	<ul style="list-style-type: none"> - No damage to can - No bending, cracks, signs of damage to the can carrier box. - Glued portions of box do not come apart 	<ul style="list-style-type: none"> - No damage to the can nor the box. - All glued portions intact. 	Pass	1 round of tape around the walls of the box added as a safeguard in case the glue wears out.
Movement test: Place can drink into the can carrier and <i>teleop</i> the robot in all directions (forward, back, rotate)	<p>Robot should move in a straight line when powered forward/back.</p> <p>Robot should be able to move unimpeded even when can is loaded.</p>	<p>1st try: Robot movement was impeded by ball caster which lifted the robot tyres slightly above ground.</p> <p>2nd try: Robot able to move unimpeded and relatively straight (negligible left drift)</p>	<p>Fail</p> <p>Pass</p>	<p>Entire Can carrier was lifted slightly up by adding hex nuts below the can carrier when it is installed on the robot.</p> <p>No further action required.</p>
Dispensing test: Manually dock the robot, place the drink can in the dispenser, and activate the dispenser to dispense the drink.	Drink can should fall into the can carrier and be fully seated inside the carrier, without human intervention.	<p>Noted certain positions of can drink in the dispenser, where the can would not fall properly into the can carrier when dispensed (i.e. rim of the can stuck on the edges)</p>  <p>Figure 6.1: Rim of can stuck on edge of can carrier (Re-enactment)</p>	Conditional Pass	<p>Sticker added onto dispenser carrier assembly to demarcate the correct position of can, in order for can drink to be dispensed properly.</p> <p>A new base with dimensions increased slightly was fabricated to create more room for the can to drop into the box (and not get stuck on the edges)</p>

6.2.2. Manual Docking

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Press the limit switch.	Dispenser Serial Monitor should display 'Docking Status: 1'	Dispenser Serial Monitor did display 'Docking Status: 1'	Pass	NA
Position the robot <u>straight in front</u> of the dispenser and <i>rteleop</i> the robot forward till the homing angle enters the dock.	Robot should be able to fully dock, i.e. Dispenser Serial Monitor should display 'Docking Status: 1'	1st try: Robot was able to fully dock but <u>did not engage</u> limit switch	Fail	Cardboard homing adaptor added to bridge the gap between homing angle and docking angle (allowing robot to engage the switch)
		2nd try: Robot able to engage limit switch	Pass	
Position the robot <u>slightly off centre</u> of the dispenser and <i>rteleop</i> the robot forward till the homing angle enters the dock.	Robot should be able to fully dock by sliding along docking angle into place, i.e. Dispenser Serial Monitor should display 'Docking Status: 1'	1st try: Robot was stuck when entering docking angle (due to obstruction caused by homing adapter)	Fail	Cardboard homing adapter extended to cover the entire docking angle, such that the robot's homing angle will not get stuck when entering the dock.
		2nd try: Robot able to dock/ engage limit switch	Pass	

6.2.3. Autonomous Docking

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Test that the Robot stops at an appropriate distance from the Dispenser	Robot stops at a distance where limit switch is activated but the Dispenser is not moved	<p>The robot would often crash into the dispenser and push it out of position slightly.</p> <p>The dispenser cannot be aligned to the wall behind it because it will</p>	Fail	<p>A cardboard box is added behind the Profile Bar base and a piece of paper is taped below the Dispenser. The cardboard box acts as an electronics platform, where the breadboard, keypad and battery holder is mounted.</p> <p>The paper ensures that, when the Robot moves on the paper, it is not able to push the Dispenser since it is taped to the paper.</p>
Test the docking sequence for all 6 tables	Robot is accurately aligned to the Carrier using the Docking Angle	The Docking was not reliable despite the calibration step	Fail	Additional calibration step closer to the Dispenser where Robot aligns itself to the wall beside the dispenser such that its centre of gravity is 50cm from the wall (the centre of the docking angle is also 50cm from the wall).
Test the docking sequence is reliable after additional calibration steps	Robot is accurately aligned to the Carrier using the Docking Angle	<p>Due to the slight inaccuracies of the LiDAR, the Robot would still get stuck in front of the Dispenser as the turning angle was not always accurate.</p> <p>The homing adapter may get stuck outside the homing structure of the dispenser as the adapter is exactly the same size as the opening on the dispenser, requiring high vertical accuracy when moving in.</p>	Fail	<p>Infrared Sensor Module is added. One on each side of the Robot's carrier. The paper at the bottom of the dispenser is black in the middle and white on the sides so that the IR Module can be used to align to the Robot.</p> <p>The homing adapter on the robot was chamfered to give more room of error so the robot can slide into the homing structure even if there is a slight height difference between the homing adapter and the opening on the dispenser during docking.</p>

6.3. Robot Navigation

Test Performed	Expected Outcome	Observations	Pass/Fail	Improvements Made
Test if the robot can correctly navigate to and stop near table 6	The robot finds table 6 and moves towards it, eventually stops within 15 cm from table 6.	The robot sometimes cannot stop in time and crash into table 6 when the table is too near the position where the robot searches for table 6, as the robot needs a certain distance to react to the table and performs its stopping function.	Fail	Changed the robot's position to look for table 6 and allow more room for the robot to move when moving towards table 6 after finding it.
Test if the robot can navigate to respective tables without crashing into other tables along the way.	The robot moves through a designated path between the tables and stops at the desired table.	The robot sometimes crashes into tables along the way, especially when passing through table 2 and 3 as the gap was quite narrow.	Fail	Added distance checking with respect to walls and tables such that the robot ensures its position is at the centre of the narrow gap before passing through the gap.

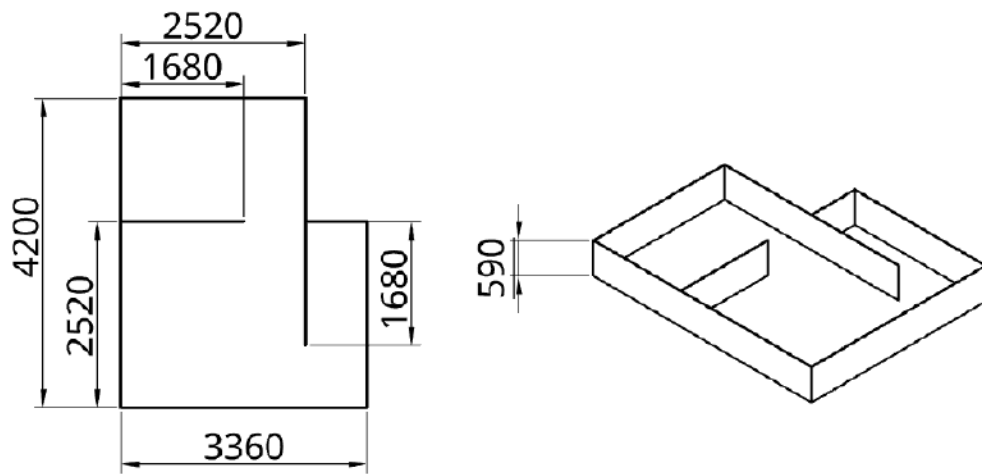
7. Evaluation and Testing (Final Run)

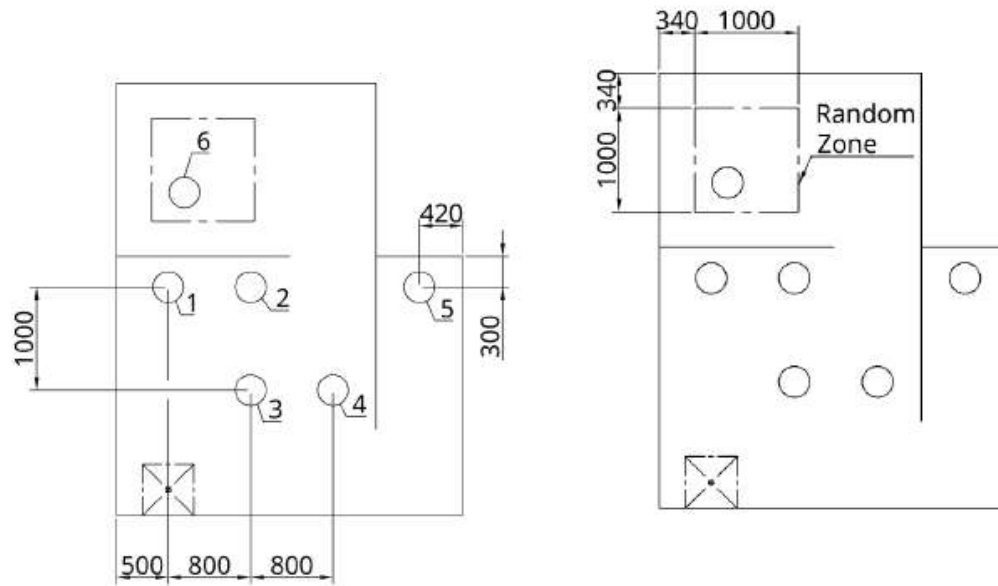
7.1. Factory Acceptance Test

A Factory Acceptance Test (FAT) was carried out and passed prior to the Final Run in accordance with the FAT checklist in the End User Documentation. These checks should be carried out prior to every mission to ensure that hardware, electrical and software components/systems on the Delivery Robot and Dispenser are serviceable and functioning correctly. Faults can also be more easily isolated for rectification to occur, should the need arise.

7.2. Evaluation Setting

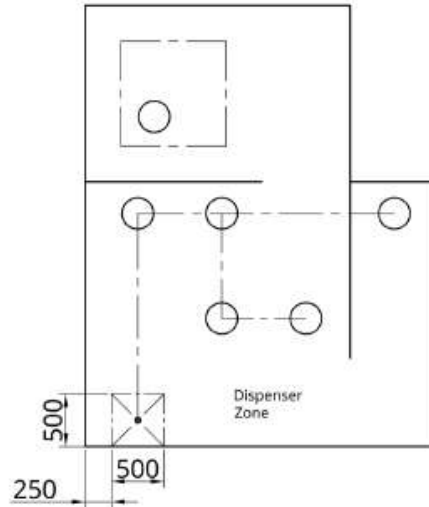
In accordance with the Final Assessment Rubrics created by the Mission Master, our Delivery Robot and Dispenser were tested in the following mock-up of a restaurant (dimensions in mm):





*Drawing 2: Position of the tables 1 to 5 (left);
Position of the Random Zone (right)*

The dispenser must be placed inside the dispenser zone (50x50cm).



Drawing 3: Position of the Dispenser Zone

Figure 7.1. Floorplan of the Restaurant Mock-up (Source: Final Assessment Rubrics)

The restaurant includes a dispenser zone (where the dispenser is located), 4 fixed tables in the main dining hall, 1 fixed table in a separate room (Table 5), and 1 table to be randomly placed in a 1x1m square in another separate room (Table 6).

7.3. Evaluation Results

According to the Mission Regulations, groups were given 26 minutes to set up their dispenser and robot, clear their End User Documentation with the teaching team, and have the robot deliver drinks to all 6 tables.

The Dispenser was able to dispense the drink accurately every single time, and for every single table, the delivery robot was able to transport the drink to the selected table, wait for the TA to collect the drink, and navigate back to the dispenser without colliding with the restaurant walls/tables.

However, the robot was unable to correctly dock when returning from Table 5, due to misalignment of the robot with respect to the dispenser during the docking process (resulting in a 1 mark deduction). The docking process was accurate and successful for all other tables.

With an overall score of **59/60** according to the rubrics and completing the entire graded run with **1 min 58 seconds** left on the clock, the graded run can be described as an overall success, apart from the docking issue on Table 5 which needed to be investigated.

8. Conclusion & Areas for Improvement

Despite the overall success of the Delivery Robot and Dispenser System, some areas of improvement were identified by the team.

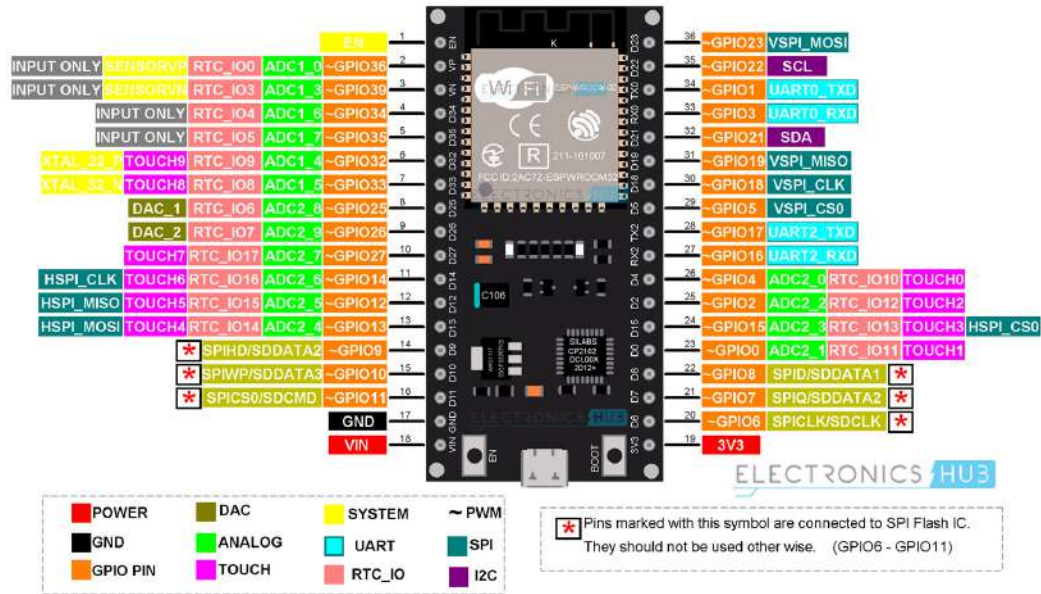
An investigation was carried out into the 'Failure to dock correctly' incident when executing Table 5. It was found that the robot was misaligned too far left of the dispenser, such that when making entry into the docking angle, the robot's homing angle was stuck on the outside of the docking angle and thus unable to proceed further and dock correctly. Attempts to replicate the error were unsuccessful, signifying that the incident resulted from random errors in distance/angle measurements made by the robot when returning from Table 5 and when aligning with the dispenser. This could have been avoided if line-tracing was serviceable.

Our Team was unable to use the IR sensor for our line-following algorithm at the last minute. As the IR sensor added was not properly calibrated to the lighting conditions of the test venue at the final run timing, the reading became inaccurate and detected the dark coloured floor as "black coloured", leading to unusable line-tracing, causing our effort of adding the line-following algorithm to go to waste. Our team made the difficult decision of disabling the line-following portion on the spot in order to proceed with the Graded Run (which led to inaccurate docking for Table 5). For future missions, we could improve by conducting more thorough calibrations before the mission so that our sensors are tuned to work under the specific lighting conditions of the mission.

Appendix 1: Wiring Connections of Hardware Components

Dispenser:

- ESPRESSIF ESP32 WROOM 32 [Datasheet](#)



Appendix 1.1: ESP32 Pinout Diagram

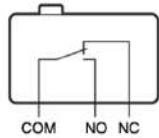
How the ESP32 is connected:

GPIO4 (D4)	Servo Motor
GPIO15 (D15)	Limit Switch
GPIO12-14; 25-27; 33	Keypad

- D2F-01L Limit Switch [Datasheet](#)

Contact Form

●SPDT



C pin	NO pin	NC pin	ESP32 Input Pin's State
1 GND	ESP32 Input Pin (with pull-up)	not connected	HIGH when untouched, LOW when touched
2 GND	not connected	ESP32 Input Pin (with pull-up)	LOW when untouched, HIGH when touched
3 VCC	ESP32 Input Pin (with pull-down)	not connected	LOW when untouched, HIGH when touched
4 VCC	not connected	ESP32 Input Pin (with pull-down)	HIGH when untouched, LOW when touched

Appendix 1.2: Wiring Connections for D2F-01L Limit Switch with ESP32.

How the limit switch is connected:

COM pin	NO pin	NC pin
ESP32 (GND)	-	ESP32 GPIO15 (D15)

- 3x4 Membrane Matrix Keypad [Datasheet](#)



Appendix 1.3: Keypad Pinout Diagram

How the keypad is connected:

COL2	ROW1	COL1	ROW4	COL3	ROW3	ROW2
GPIO33	GPIO25	GPIO26	GPIO27	GPIO14	GPIO12	GPIO13

- MG996R Servo Motor [Datasheet](#)

PWM=Orange (⏏)
Vcc = Red (+)
Ground=Brown (-)



Appendix 1.4: Servo Motor Pinout Diagram

How the servo motor is connected:

ORANGE pin	RED pin	BLACK pin
ESP32 (GPIO4)	AA Batteries (+ve)	AA Batteries (-ve)

- IR Sensor [Datasheet](#)

How the IR Sensor is connected:




	VCC pin	OUT pin	GND pin
Left IR Sensor	3.3V	R-Pi (GPIO17)	R-Pi (GND)
Right IR Sensor	3.3V	R-Pi (GPIO9)	R-Pi (GND)



Appendix 2: Literature Review

1. Object Transfer Mechanism

1.1. Dispenser System

The main consideration in the dispenser system is the object transfer mechanism, which is the type of method in transferring the loaded soda can from the dispenser to the delivery robot. Several possible mechanisms have been identified to be potentially suitable for implementation of the dispenser system.

Type of System	Working Principle	Image
Pusher Mechanism	<p>In some manually operated vending machines, a spring-loaded pusher shelf is used (Figure 1.1).</p> <p>When the tray is full, the spring is tensioned.</p> <p>When a can is removed, the spring releases tension and pushes the next can forward.</p> <p>In our case, we could replace the spring-loaded pushing with a Rack-and-Pinion Pusher (Figure 1.2) instead.</p>	 <p>Figure 1.1: Spring-loaded Pusher used in Vending Machines [1]</p>  <p>Figure 1.2: Rack-&-Pinion Pusher using Servo Motor [2]</p>
Pan & Tilt Camera System (Dropping Mechanism)	<p>Using dropping mechanism of can for the delivery system to catch, a similar mechanism in pan & tilt actuators for camera can be used (Figure 1.3) [3].</p> <p>Works using a stepper motor that drives a small gear, which transfers torque to a larger gear that tilts the camera holder.</p> <p>A linear bearing is utilised on the opposing side to support the rotational movement of the holder.</p> <p>In our case, the camera holder would be replaced by a can holder, where the turning of holder would drop the can into the delivery robot.</p>	 <p>Figure 1.3: Pan & Tilt Camera</p>

<p>Gate with Ramp System (Rolling Mechanism)</p>	<p>Using rolling mechanism, a ramp that opens with a gate at the entrance, like a sluice gate (Figure 1.4).</p> <p>Sluice gates use linear actuators or manual wheels to open and close them.</p> <p>However, a rack-and-pinion can be used instead to pull the gate up and down.</p> <p>With the rack against the inside wall of the gate, the load from the can should not cause misalignment.</p>	 <p>Figure 1.4: Sluice Gate [4]</p>  <p>Figure 1.5: Rack-and-pinion operated door to adapt sluice gate [5]</p>
--	--	--

1.2. Delivery System

As for the delivery system, the object transfer mechanism here is the receiving end from the dispenser. Two main components to consider would be the design of the receiving tray on the delivery robot, as well as the mounting of tray on it.

Design of Can Holder on Delivery Robot:

The soda can may be carried along either in an upright position or lying on its side. The specific dimensions of the can were provided to be a diameter of 66mm, height of 115mm, and the weight of 350g. The holder should be able to meet the above dimensions and support that weight. One material to be considered for the holder would be a laser-cut acrylic as it is sufficiently lightweight to support a load of 500g.

Mounting Position of Can Holder on Delivery Robot:

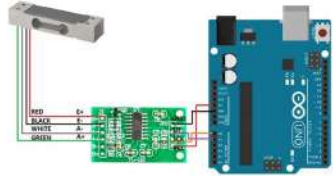
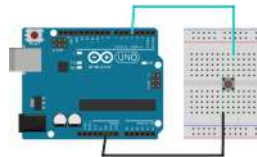

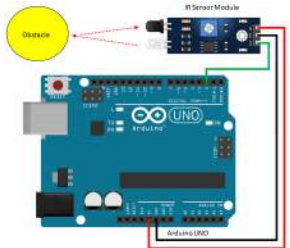
The holder may be mounted on either the back, front, sides or the top of the robot. Preferably, it should be mounted such that it does not present weight and balance issues to the robot (i.e. it should be mounted lower for a lower centre of gravity (CG)). It should also be modular to allow easy removal and installation (R&I) from the robot body during maintenance.

2. Communication Protocols

2.1. Delivery System

Detection of Can in Carrier:

The robot must detect if the can has been transferred to the robot, and if the can has been collected by the TA after arrival to the table. Some sensors available are as shown below.

Type of Sensors			Working Principle	Image
Interaction with Objects using Physical Contact	Pressure	Weight/Load Sensor	Measures the weight of contents in the holder - load of >200g indicates presence of can.	 <p>Figure 1.6: Weight Sensor in Arduino</p>
	Touch	Pushbutton	Detect a push contact to change from HIGH to LOW or vice versa.	 <p>Figure 1.7: Pushbutton in Arduino</p>
		Electro-mechanical Sensor - Limit Switch	Uses an actuator to make or break an electrical connection that performs an action. Switch is depressed/engaged when can is present, and disengages when can is removed.	 <p>Figure 1.8: Limit Switch in Arduino</p>
Interaction with Object using Proximity	Infrared Radiation (IR)	IR Sensor Module	Uses infrared light to determine whether or not the can is present. Measures heat of object as well as motion of objects. Digital Sensor → Output 1 if object is present; Output 0 if object is absent.	 <p>Figure 1.9: IR Sensor Module in Arduino</p>

2.2. Inter-system Communication

Within the TurtleBot3 robot itself, there are two types of boards that act as the interface and connection between the hardware and software components of the robot: OpenCR & Raspberry-Pi (R-Pi) boards. The deliverables for this project include one OpenCR and R-Pi board each, delivery system having both boards while another microcontroller is required on the dispensing system for full communication in the integrated system. OpenCR is an Open-source Control module designed to be integrated for Robot Operating System (ROS) which contains a microcontroller, power management, and various interfaces like USB, UART (Universal Asynchronous Receiver/Transmitter), and I2C (Inter-Integrated Circuit). The board also includes firmware that enables communication with ROS, allowing for easy integration with existing ROS packages and tools. Meanwhile, the R-Pi is a single board computer that simplifies the process of adding a brain to the system, allowing the programmer to determine how the robot will perform its tasks.

In our application, the Dynamixel motors for the robot's wheels that were setup during robot assembly are particularly driven by the OpenCR. However when necessary, external sensors on the robot and software coding on the RasPi would be used as a means of communication between the dispenser and robot. The USB cable has been used to link the OpenCR and RasPi, and the USB communication protocols have already been set.

This project requires communication between different interfaces using the following various types of protocols.

Connection between	Communication Protocols
Microcontroller & R-Pi [Dispenser ↔ Delivery Robot]	<p><u>Serial Communication:</u></p> <ol style="list-style-type: none"> 1. UART: Using UART Port on both boards, they exchange data in a simple, reliable, and cost-effective way. UART interface has two data lines: TX (transmitter), RX (receiver) and uses a common ground. 2. I2C: A synchronous serial communication protocol. Using a single bus, typically with two wires (SDA & SCL) and a common ground. Often used for connecting multiple devices to the RasPi. 3. SPI (Serial Peripheral Interface): Another synchronous serial communication protocol. Using four wires (MOSI, MISO, SCK, and SS) and a common ground. Often used for high-speed communication and supports full-duplex communication.
	<p><u>Wireless Communication:</u></p> <ol style="list-style-type: none"> 1. Bluetooth: Both boards have built-in Bluetooth modules, which can be used to establish a wireless connection between them to exchange data & commands. Bluetooth technology provides a low-power, short-range wireless connection that can transmit data at a moderate speed. 2. Wi-Fi: RasPi has built-in Wi-Fi connectivity, which can be used to connect to a wireless network or to create a wireless access point. Microcontroller requires additional Wi-Fi module which can be connected to the same wireless network, and both can then communicate with each other through that network. 3. Radio Frequency (RF): Using radio waves. Can be achieved using modules (e.g. NRF24L01 or RFM69), which can be connected to both boards. The microcontroller and Raspberry Pi can send and receive data wirelessly using these modules. <p><u>Wireless Protocols:</u></p> <p>MQTT (Message Queuing Telemetry Transport): Lightweight publish-subscribe messaging protocol that is widely used for IoT applications. It can be used to enable communication between both boards by allowing them to exchange messages with each other over a network.</p> <p>Shared Memory: Allows two or more processes to share a region of memory that can be accessed by all processes. Typically used in high-performance applications that require fast communication between processes (e.g. video processing, real-time control systems).</p>

Microcontroller Choices on Dispenser System

1) Arduino MKR WiFi 1010

It features an Atmel SAM D21, a 32-bit Cortex-M0+ low power ARM microcontroller and a Nina W102 uBlox module for Wi-Fi and Bluetooth connectivity. It runs at 48 MHz and features 256 KB of flash memory and 32 KB of SRAM, but no EEPROM.

2) Arduino Nano 33 IoT

It features an Atmel SAM D21, a 32-bit Cortex-M0+ low power ARM microcontroller and a Nina W102 uBlox module for Wi-Fi and Bluetooth connectivity. It runs at 48 MHz and features 1 MB of flash memory and 256 KB of SRAM, but no EEPROM.

3) ESP32

It features a Xtensa dual-core (or single-core) 32-bit LX6 microprocessor. Aside from WiFi and Bluetooth connectivity, ESP32 has additional features such as dual-mode Bluetooth, low-power BLE, and support for various high-speed communication protocols. It runs at 240 MHz and features 4 MB of flash memory, 520 KB of SRAM, and 64 KB of EEPROM.

3. Navigation

Navigation is important for the delivery robot to operate about the restaurant entirely autonomously without much human interference. The following shows a variety of methods of how the TurtleBot3 robot can locate itself.

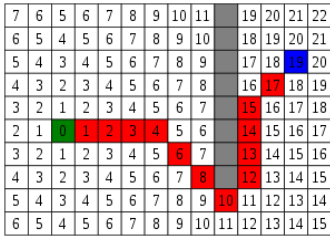
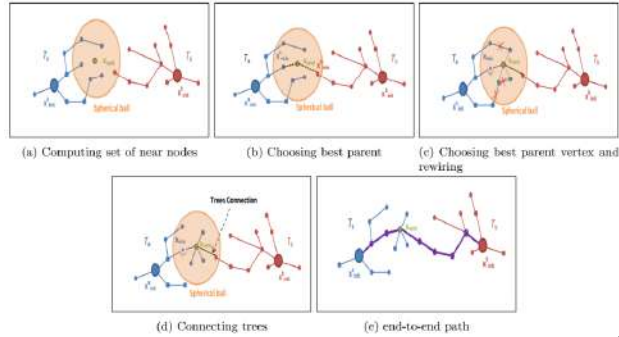
3.1. Mapping

Mapping in robots is a form of a data model for it to know the surrounding layout and store them in memory. Simultaneous Localisation and Mapping (SLAM) algorithm will be implemented using Light detection and ranging (LiDAR) data from LDS-01 and odometry data from Inertial Measurement Unit (IMU) on OpenCR. The LiDAR sensor produces an accurate map of their constantly changing surroundings for safe navigation by gathering and measuring millions of information points in instantaneously. This gives them a high-resolution 3D representation of their surroundings. Meanwhile, the odometry data refers to those of the motion sensors, which would be the Dynamixel motors for the wheels, to compute the position of the object in relation to a starting point. Some possible SLAM algorithms are presented in the following table.

#1: Gmapping	#2: Cartographer
GMapping is based on the grid-based FastSLAM algorithm and uses Rao-Blackwellized particle filters (RBPFs) to predict the state transition function. It is designed to handle dynamic environments, partial observability, and loop closures. Gmapping is well-suited for small-scale mapping and localization (7).	Cartographer is based on an extended Kalman filter (EKF) and is designed to handle large-scale, highly dynamic environments and multiple sensor inputs. It uses real-time loop closure detection to correct for drift and improve the overall accuracy of the map. It also uses scan matching, which aligns laser scans of the environment to improve the accuracy of the robot's estimated pose. Cartographer is well-suited for highly dynamic environments (8).

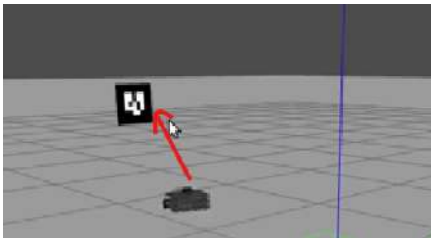
3.2. Path Planning



Upon receipt of data on table selection for delivery, the robot should determine potential paths for navigation to the designated table, utilising both pre-inputted maps and maps generated through SLAM. To accomplish this, a suitable path planning algorithm is required, some possible algorithms are shown in the following table.

#1: A* Algorithm	#2: RRT-Connect Algorithm
 <p>Figure 2.10: Graphic Representation of A* Algorithm</p> <p>A graph search algorithm that combines the strengths of Dijkstra's algorithm and a heuristic function (9). It uses a heuristic function to guide the search towards the goal and is guaranteed to find the optimal solution if the heuristic function is admissible (never overestimates the distance to the goal) and consistent (satisfies the triangle inequality).</p>	 <p>Figure 2.11: Bidirectional Trees Representation of RRT-Connect Algorithm</p> <p>A variation of the Rapidly exploring Random Trees (RRT) algorithm that is designed to find a path between two points more efficiently by growing two trees simultaneously, one from the start point and one from the goal point, and then connecting the two trees when they come close enough (10).</p>

3.3. Table Identification

Once the robot has received the user's input data of which table to deliver the can to, it has to know how to identify the tables at different locations. Some possible ways to identify are shown in the following table.

#1: OpenCV Detection of ArUco Marker	#2: NFC Tag
 <p>Figure 2.12: Simulation of ArUco Marker Tracking by TurtleBot3 Waffle Robot.</p> <p>ArUco markers are binary square fiducial markers used for camera pose estimation. This simulation displays a TurtleBot3 tracking a ArUco marker. The camera mounted on the robot is constantly detecting images of</p>	<p>An NFC reader module will be installed and connected to the RasPi, allowing the robot to be able to read NFC tags placed at the tables (12).</p> <p>NFC reader activates and transmits radio waves to induce electric current in the NFC tag, a passive device which does not need any power source, but instead built to activate and begin transmitting NFC signal when exposed to a changing magnetic field (13). The reader will have an electric current running through the tag and when they are put in close proximity, the reader will prompt the tag to begin transmitting while it reads</p>

the surrounding environment and then moves towards the closest ArUco marker (11).	the signal, then the tag will stop transmitting when it's moved away.
<p>Table Identification Process (Aruco Marker)</p> <pre> graph TD A[Received information regarding the table number from dispenser] --> B[Scan the surrounding and look for different Aruco markers on the 6 tables] B --> C[Identify the specific marker that denotes the desired table number] C --> D[Move towards the identified Aruco marker] </pre>	<p>Table Identification Process (NFC Tag)</p> <pre> graph TD A[Received information regarding the table number from dispenser] --> B[Move around the area to the different tables and scan the NFC tags at the table] B --> C[Pick the correct table once the NFC tag information match the information from the dispenser] </pre>
<p>Hardware required: Raspberry Pi Camera</p>  <p>Figure 1.13: Raspberry Pi Camera (14)</p>	<p>Hardware required: PN532 NFC Module</p>  <p>Figure 1.14: PN532 NFC Module</p>

References

[1] "Spring-Loaded Pusherfeed and Shelf Dividers | Hangzhou Novday," Plastic Point-of-Sales | Plastic shelf label profile strips | Shelf pushfeed system Supply.
<https://nova-day.com/product-category/shelf-management-system/shelf-pushers-shelf-dividers/page/2/> (accessed Jan. 25, 2023).

[2] DIY Linear Servo Actuator, 3D Printed. Accessed: Jan. 25, 2023. [Online Video]. Available:
<https://www.youtube.com/watch?v=2vAoOYF3m8U>

[3] 3D Printed DSLR Camera Pan Tilt Mount (Arduino/Stepper Driven) 2020. Accessed: Jan. 25, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=uJO7mv4-OPY>

[4] chutchinson, "Heavy Duty Tite Seal Sluice Gate - Plasti fab," Plasti-fab.
<https://www.plasti-fab.com/products/heavy-duty-tite-seal-sluice-gate/> (accessed Jan. 25, 2023).

- [5] How to make Automatic Sliding Door/ Easy Tutorial. Accessed: Jan. 25, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=3hEipRDH3_o
- [6] "Standard Soda Can Dimensions and Guidelines - MeasuringKnowHow," Jan. 16, 2022. <https://www.measuringknowhow.com/soda-can-dimensions/> (accessed Jan. 25, 2023).
- [7] "Different flavors of Coke® have different densities," Web Physics UCSB. <https://web.physics.ucsb.edu/~lecturedemonstrations/Composer/Pages/36.34.html#:~:text=Each%20can%20of%20regular%20Coke,enough%20to%20sink%20the%20can>
- [8] WatElectronics, "Inter-Process Communication : Types, Working, Differences & Applications," WatElectronics.com, Jul. 07, 2022. <https://www.watelectronics.com/inter-process-communication/> (accessed Jan. 25, 2023).
- [9] Z. Liu, Z. Cui, Y. Li, and W. Wang, "Parameter Optimization Analysis of Gmapping Algorithm Based on Improved RBPF Particle Filter," J. Phys.: Conf. Ser., vol. 1646, no. 1, p. 012004, Sep. 2020, doi: 10.1088/1742-6596/1646/1/012004.
- [10] K. Xing, X. Zhang, Y. Lin, W. Ci, and W. Dong, "Simultaneous Localization and Mapping Algorithm Based on the Asynchronous Fusion of Laser and Vision Sensors," Frontiers in Neurorobotics, vol. 16, 2022, Accessed: Jan. 25, 2023. [Online]. Available: <https://www.frontiersin.org/articles/10.3389/fnbot.2022.866294>
- [11] N. Swift, "Easy A* (star) Pathfinding," Medium, May 29, 2020. <https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2> (accessed Jan. 25, 2023).
- [12] A. H. Qureshi and Y. Ayaz, "Intelligent bidirectional rapidly-exploring random trees for optimal motion planning in complex cluttered environments," Robotics and Autonomous Systems, vol. 68, pp. 1–11, Jun. 2015, doi: 10.1016/j.robot.2015.02.007.
- [13] Arcuo markers tracking : turtlebot waffle. Accessed: Jan. 25, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=fj_FiS5P9t4
- [14] Writing and Reading RFID Tags and NFC Cards on Raspberry Pi with PN532 Module. Accessed: Jan. 25, 2023. [Online Video]. Available: <https://www.youtube.com/watch?v=kpaQAqhv4R0>
- [15] "How to Use an NFC Reader." <https://www.getkisi.com/academy/lessons/how-to-use-an-nfc-reader> (accessed Jan. 25, 2023).
- [16] Y. Name, "ROBOTIS e-Manual," ROBOTIS e-Manual. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_raspi_cam/ (accessed Jan. 25, 2023).

Appendix 3: Concept Design (G1 Report - Before Week 6)

There were major changes between Concept and Preliminary Designs, as stated in Section 5 of the main body of our report. Below contains our Concept Design as submitted in the G1 report, for reference.

1.1. Concept Design of Dispenser system

1.1.1. Mechanical System

The chosen mechanism for the dispenser is adapted from the pan and tilt camera idea. Though the original idea used a stepper motor with a hall sensor, in our case we chose to use a servo motor since it already allows position control and removes the need for a hall sensor. Additionally, it's more suitable for our lower mechanical load requirements compared to the DSLR camera. The servo turns a smaller spur gear which turns a large spur gear attached to the 'bucket' where the can is placed. This idea supports one can to be placed at a time. The following Figure 3.1 shows a CAD model of the final dispenser system.

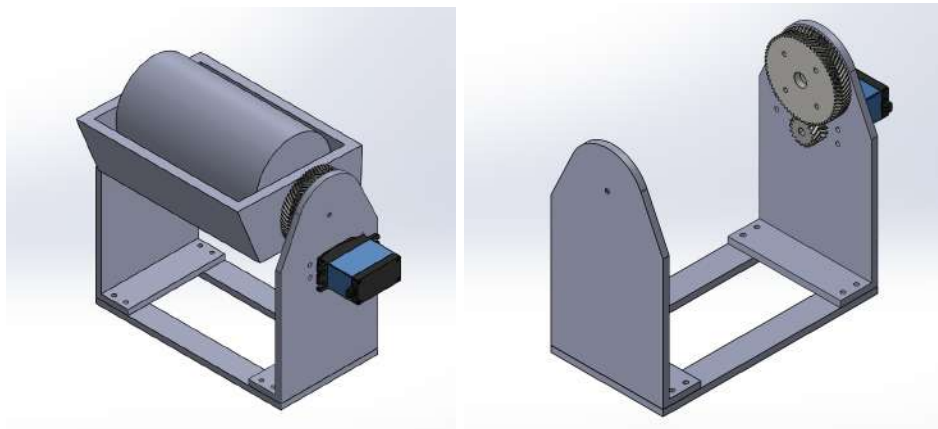


Figure 3.1: CAD Model of Dispenser System

If we were to upscale the design for more cans, we would introduce a barrel above the bucket to load in more cans such that when one can is being dropped, the other cans are blocked by the back of the bucket. Reloading would occur when the bucket is back in its starting position. However, we would only be exploring this iteration once we are satisfied with the stability of the single can system.

1.1.2. Electrical System

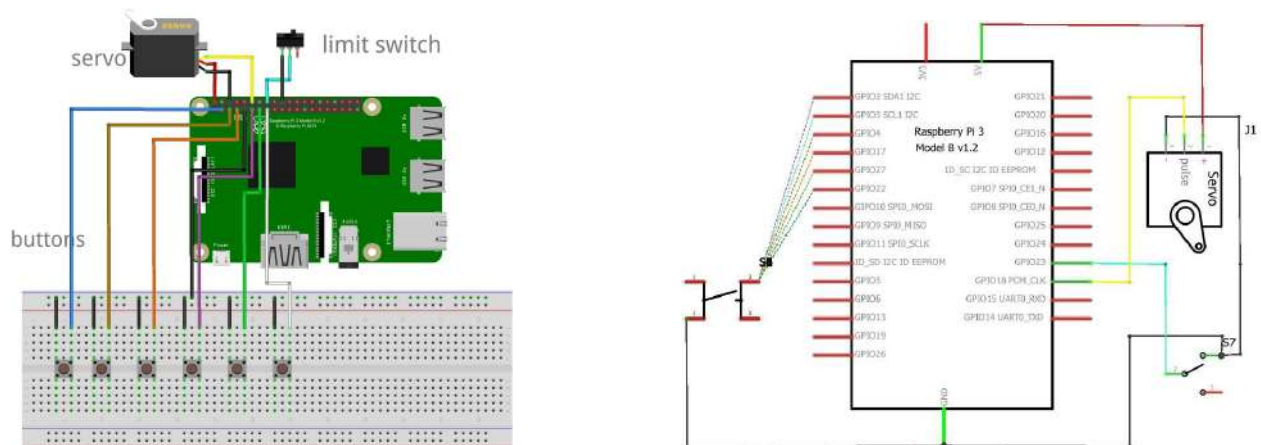


Figure 3.2: Electrical Architecture of Dispenser System

The dispenser uses RasPi 2 to control the servo, the limit switch, the button to enter the choice of table and the communication with the delivery robot. The purpose of the limit switch is to detect whether the can has been placed in the bucket or not. The system will make use of the GPIO pins on the RasPi 2 to control the servos. The components will be interfaced using a protoboard.

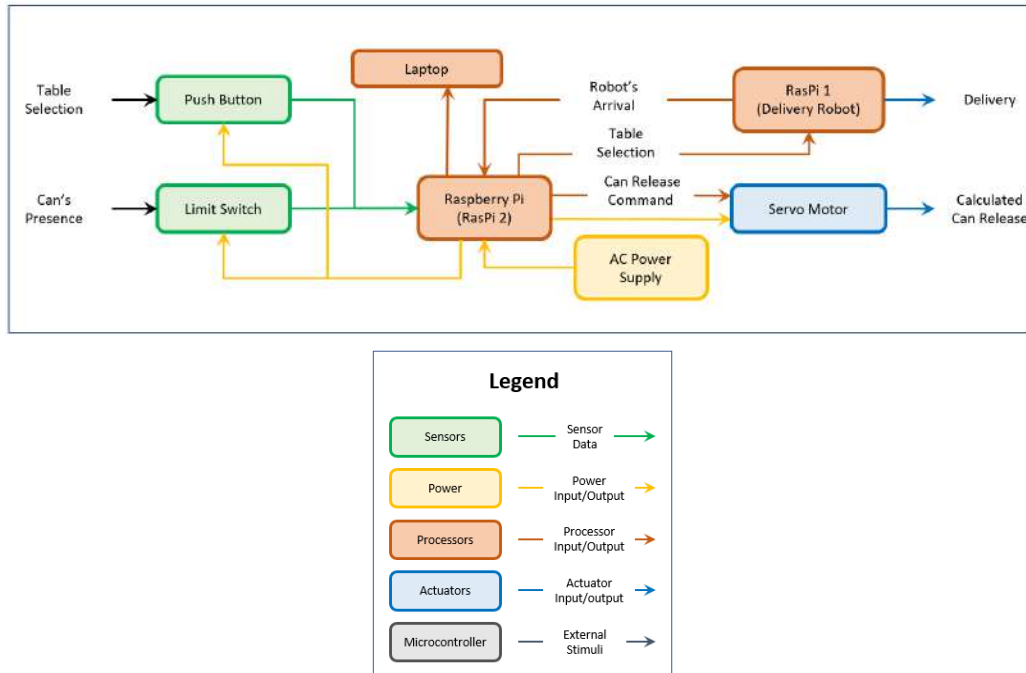


Figure 3.3: Functional Block Diagram of Dispenser

1.1.3. Software System

When the RasPi 2 receives data from the limit switch that the can has been placed, it will inform the servo to turn the bucket. Once the limit switch indicates that there is no longer a can in the bucket and it has received information from the buttons on the delivery bot that there is no can, RasPi 2 will publish a message to RasPi 1 on the Delivery Robot that the can has been dispensed and which table the can should go to using Message Queuing Protocol. When the delivery bot returns, RasPi 2 will receive communication that the delivery bot has returned and will repeat the cycle again.

1.2. Concept Design of Delivery Robot

1.2.1. Mechanical System

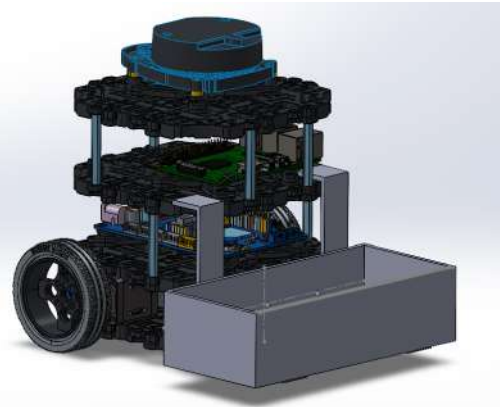


Figure 3.4: CAD Drawing of Delivery Robot

The can is transported lying down in a can holder mounted low and to the rear of the Turtlebot. The can holder is mounted to a Z-shape mount that is connected with screws to the main body. The Z-shape mount allows access to critical I/O ports on the Rpi and OpenCR with the box removed. The holder is made from laser-cut acrylic, with the dimensions 160 x 70 x 45 mm to accommodate a drink can lying on its side.

Electromechanical sensor (limit switch) is installed at the base of the holder to determine the presence of the can (switch depressed/engaged) before the robot travels to the table, and determine that the can is removed (switch disengaged) before returning to the dispenser.

1.2.2. Electrical System

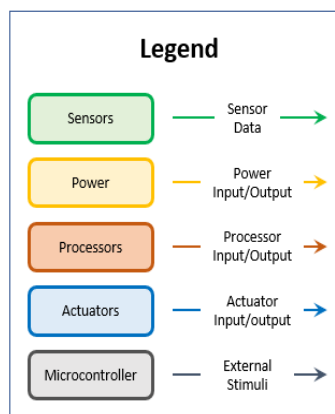
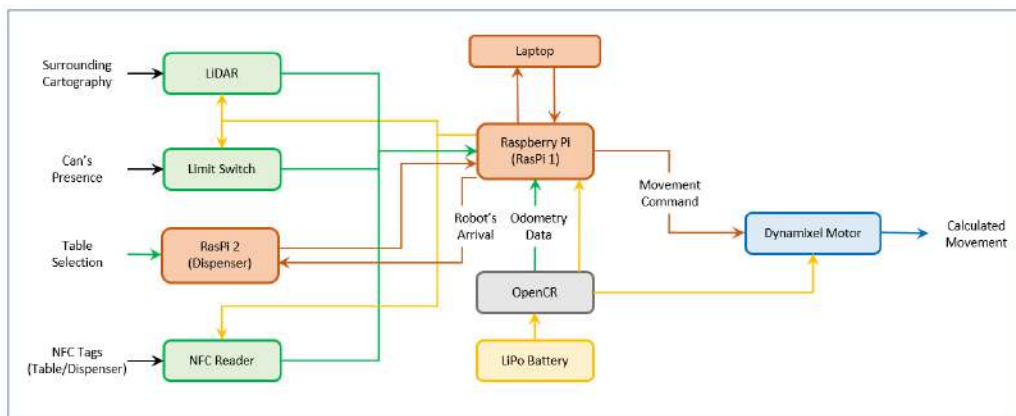


Figure 3.5: Functional Block Diagram of Delivery Robot

Appendix 4: Preliminary Design (Week 7 Design Review)

There were only minor changes made between the Preliminary Design and the Final Design, as described under Section 6 'Testing and Validation' in the main body of our report. Please refer to our Design Review slides, available at this link:

https://github.com/yinheng996/r2table_nav/blob/1dda3ea3f326f36e6e8cc90856f0b17b45e27098/Documentations/EG2310%20Design%20Review%20Presentation%20-%20Group%203.pdf

A Summary is provided below:

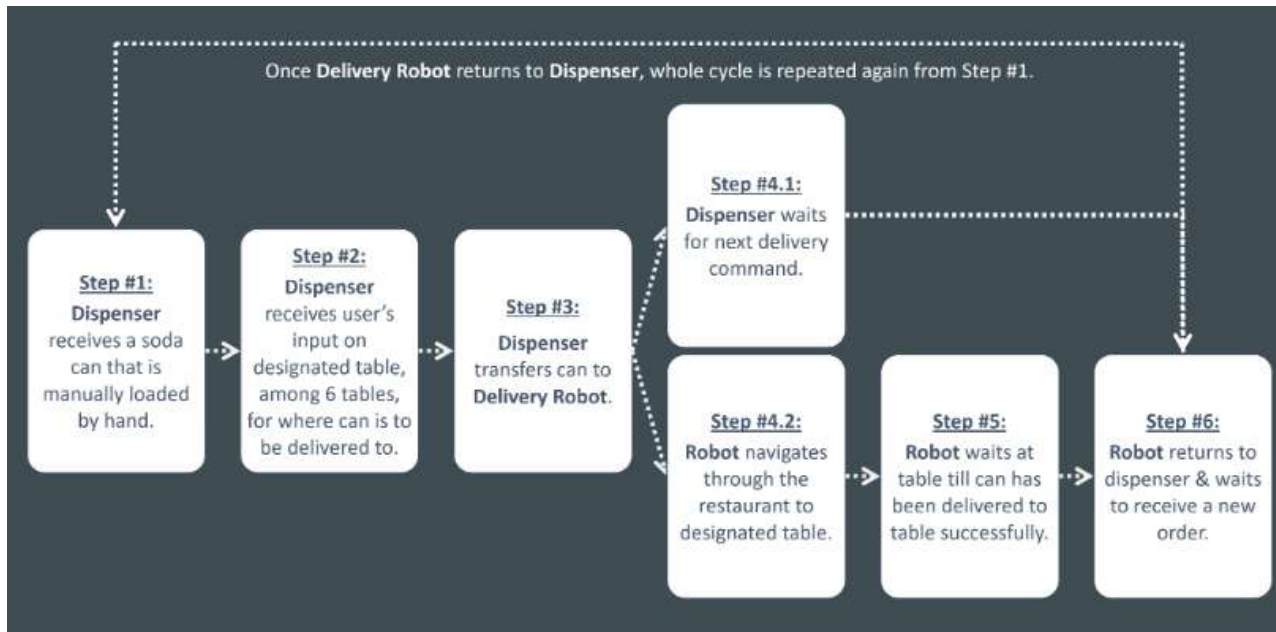


Fig 1.1 Overview of Mission Flow

There was no change to the Mission Flow between Preliminary and Final Design.

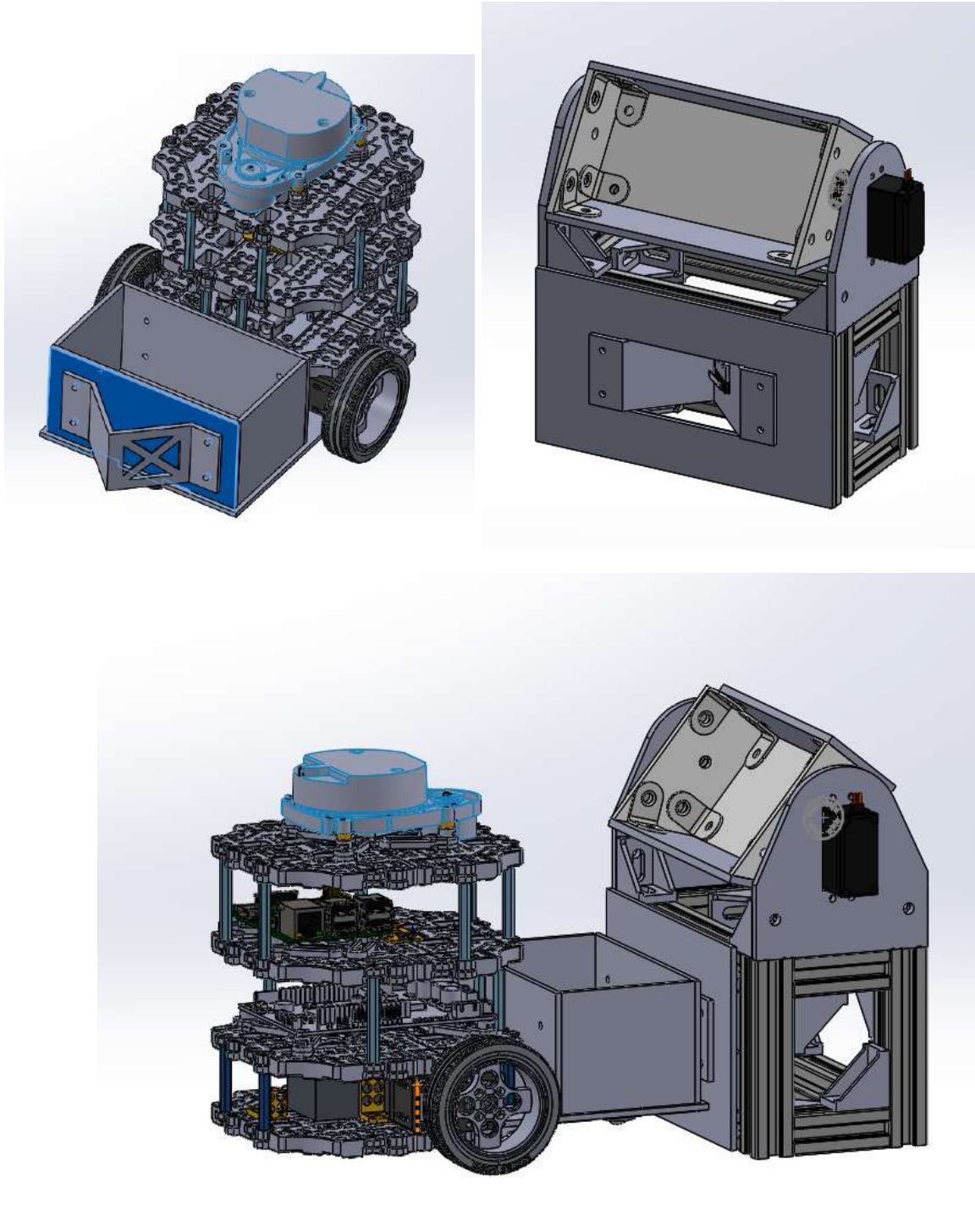
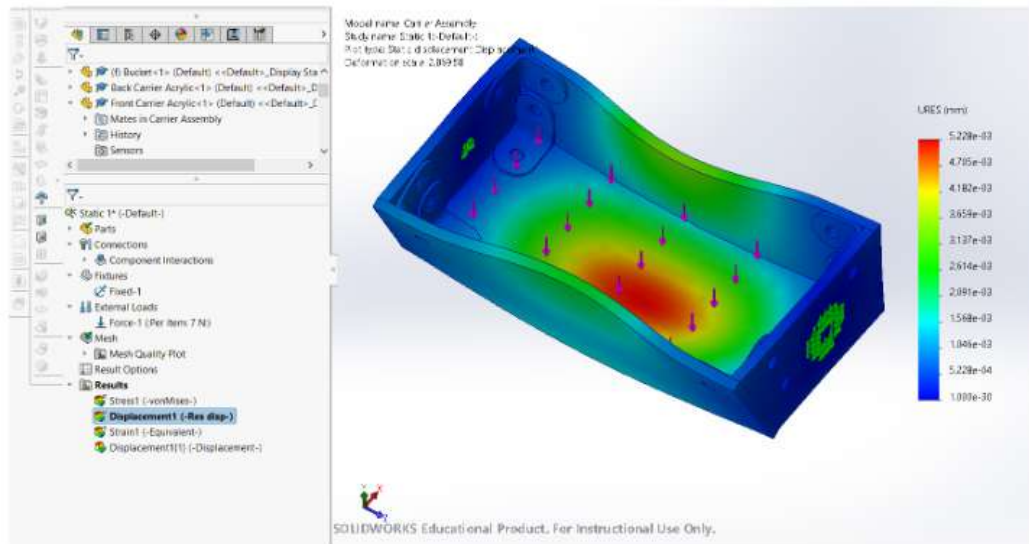


Fig 1.2. CAD of Dispenser and Delivery Robot

Major changes made after Preliminary Design include the addition of IR sensors to the robot for line following, and removal of L-brackets within the Dispenser's Can Carrier box.

FEA for the carrier- Displacement



Maximum Displacement is $5.2 \cdot 10^{-3}$ mm

Figure 1.3 : Finite Element Analysis of the Dispenser Carrier shows an acceptable maximum displacement

FEA- von Mises Stress

The tensile strength of the material AISI 304 is **between $4.85 \cdot 10^8$ Rm N/mm²**
For acrylic sheet it is **$7.5 \cdot 10^7$ Rm N/mm²**

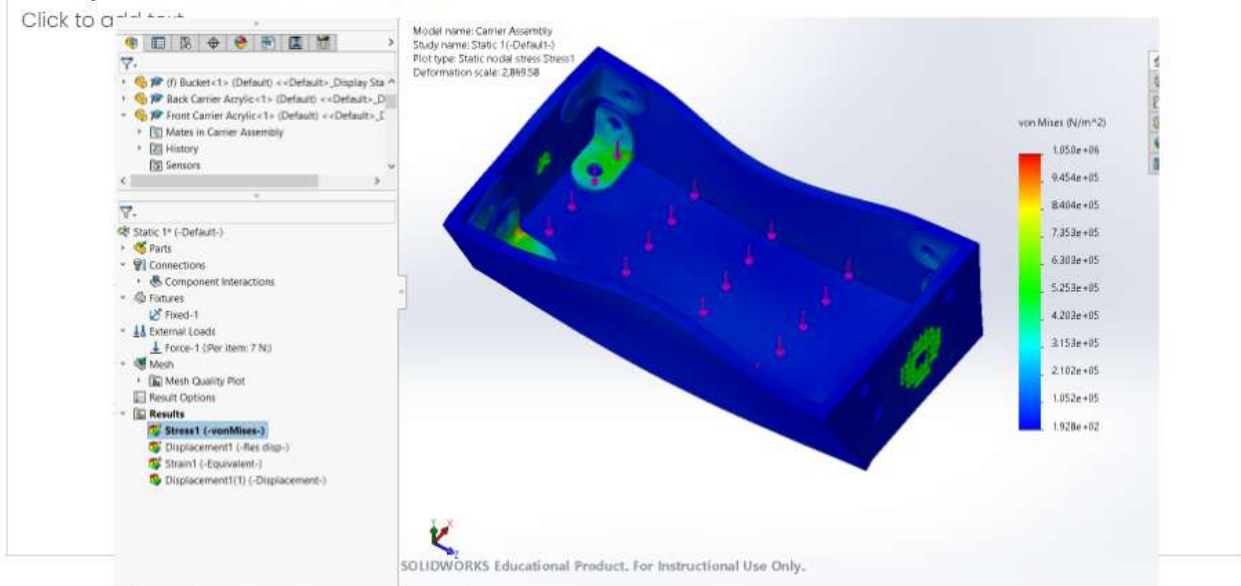


Figure 1.4 : Finite Element Analysis of the Dispenser Carrier shows an acceptable von Mises Stress

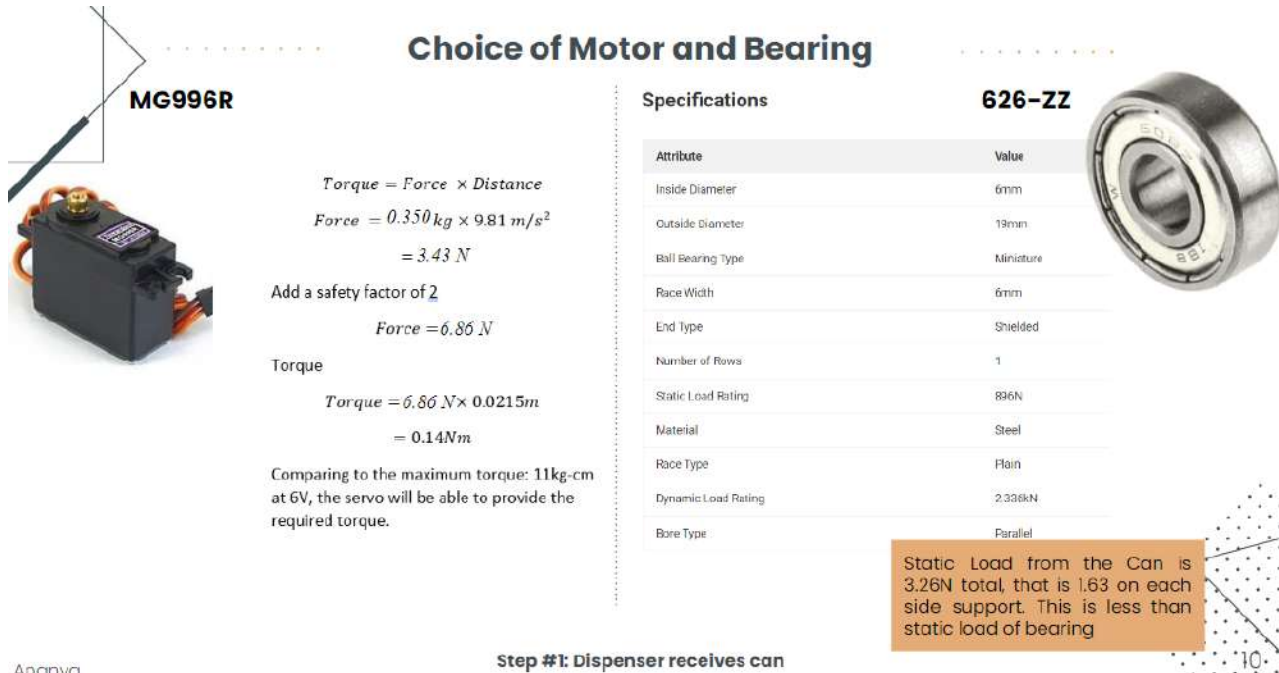


Fig 1.5. Calculations for the servo and bearing to ensure they are able to sustain the load conditions

Legend

- Table 1 —
- Table 2 —
- Table 3 —
- Table 4 —
- Table 5 —
- Table 6 —

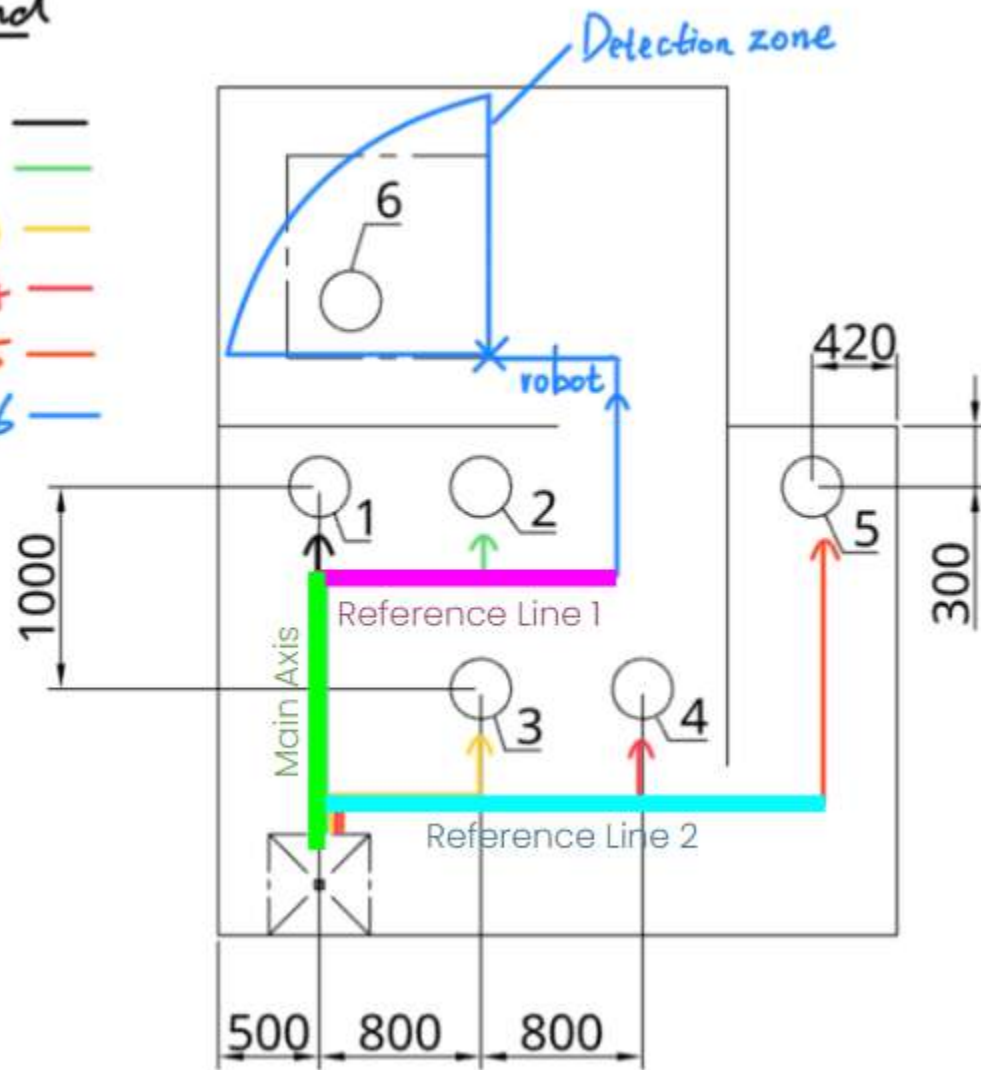


Fig 1.6. Navigation Plan for Robot

Major changes made after Preliminary Design include calibration against the wall to ensure accuracy whilst navigating to/from tables, as well as changing the location in which the Robot starts the detection process for Table 6 (from bottom right to the top right corner of the Table 6 zone).

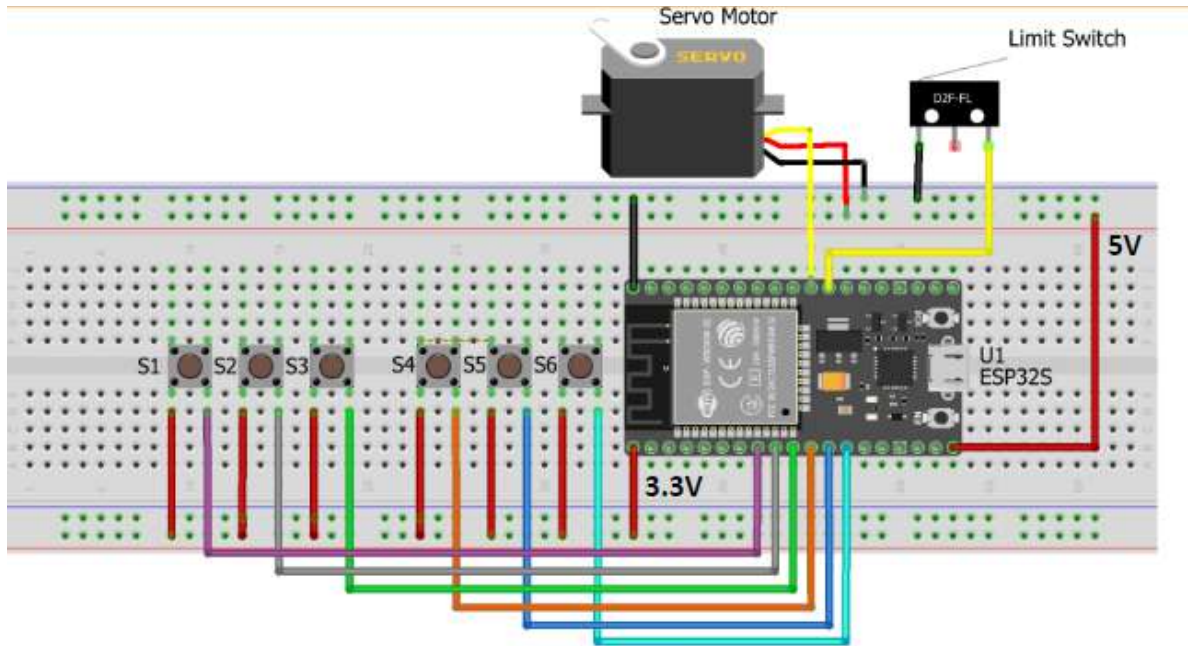


Fig 1.7 Electrical Wiring for Dispenser

The Preliminary Design also called for 6x Push Buttons for table selection, this was changed to a numerical keypad as it was available in the Electronics Lab.

It was originally planned to power the ESP32 using a 5V phone charger plug, and the Servo Motor from the 5V pin of the ESP32, however it was changed to 4xAA batteries for the Servo Motor due to insufficient voltage supply for the Servo Motor, and the USB outlet of the laptop for the ESP32 so that the Serial Monitor output on the Arduino IDE app can be monitored while the dispenser is in use.