CG2111A Engineering Principle and Practice
Semester 2 2023/2024

"Alex to the Rescue"
Final Report
Team: B04-4B

NAME	STUDENT	SUB-TEAM	ROLE
MARK NEO QI HAO	A0281484X	Hardware	Circuitry, Color, Controller
POH YU WEN	A0271715E	Software	TCP, Lidar, Movement
SAW SHENG JIE, ISAAC	A0272400W	Firmware	Ultrasonic, SPI, Serial Comms
NGUYEN DUC PHONG	A0276035E	Software	ROS, BareMetal

Table of Contents

Introduction

Alex, our robotic vehicle, is specifically designed to address the challenges of locating and identifying victims trapped in disaster-stricken or obstacle-laden environments.

To simulate scenarios resembling real-world crises, Alex is deployed in a controlled space filled with various obstructions, simulating debris, or collapsed structures typically found in disaster sites. The primary mission for Alex is to conduct a 'search and rescue' operation by navigating through these impediments within a stringent six-minute window to identify victims marked by distinct color codes: "green" for healthy but trapped individuals, and "red" for those who are injured, and lastly generating an accurate map of its environment.

To achieve its objectives, Alex is equipped with sophisticated navigation and control systems. It operates under teleoperation, where commands are issued from a remote operator via a laptop interfaced with a master control program running on a Raspberry Pi. This setup converts commands into actionable movements controlled via the Arduino Mega 2560 board, ensuring precise maneuvering and interaction within the simulated environment, all while utilizing RPLIDAR data to minimize collisions with objects and ensuring the precision of the environment map it produces. These functionalities allow Alex to not only identify the victims with high accuracy but also map the entire environment meticulously. This mapping is crucial for post-operation review and planning future rescue missions in similar settings.

Review of State of the Art

Boston Dynamics "Spot" Robot

This system integrates advanced hardware and software components for tele-operated search and rescue missions. It has a rugged chassis equipped with high-resolution cameras, LIDAR sensors, and manipulator arms. The software component comprises real-time communication protocols, robust navigation algorithms, & intuitive user interfaces. (Boston Dynamics, 2024).


[1]

Strengths

- High-resolution cameras and LIDAR sensors provide detailed environmental data for accurate mapping and navigation of its surroundings.
- Strong attachable arm can lift weights of 20 pounds and drag up to 50 pounds, allowing it to move objects in its environment when needed.

Weaknesses

- Dependence on tele-operation limits autonomy, especially in communication-challenged environments such as disaster sites.
- Struggles with extreme terrain conditions such as steep relief, slippery surfaces, and undulating surfaces. This is due to the design of its legs being optimized for level-ground movement, and the traction may be insufficient elsewhere.

DJI Matrice 300 RTK

This system emphasizes mobility and adaptability in search and rescue missions. It typically features a modular design with interchangeable sensors and payloads, supported by a robust communication network. The hardware includes ruggedized wheels or tracks, along with a variety of sensors such as cameras, thermal imaging devices, and gas detectors. Software components encompass intelligent navigation algorithms, machine learning for object recognition, and tele-operation interfaces. (DJI, n.d.)


[2]

Strengths

- Diverse sensor suite provides comprehensive situational awareness for effective search and rescue operations.
- Intelligent navigation algorithms optimize path planning and obstacle avoidance.
- Machine learning enhances its object recognition and autonomous decision-making.

Weaknesses

- Limited tele-operation range in remote or communication-challenged areas.
- Initial setup and calibration of sensors and software modules can be time-consuming.

Learning Points

Drawing inspiration from our case studies, it's imperative that our Alex exhibits navigation expertise and stability akin to the DJI3000. The crux lies in ensuring the precision of our environment mapping via LIDAR and SLAM technologies, coupled with intuitive movement controls to maximize speed during navigation and exploration.

Additionally, we aspire to emulate the robustness and resilience of the Spot Robot in our hardware and firmware design. This entails integrating more durable batteries, streamlining our electrical circuits to mitigate the risk of accidental disconnections or protrusions, and fortifying the integrity of our onboard components using methods like secure taping, zip ties, or blu-tack applications.

System Architecture

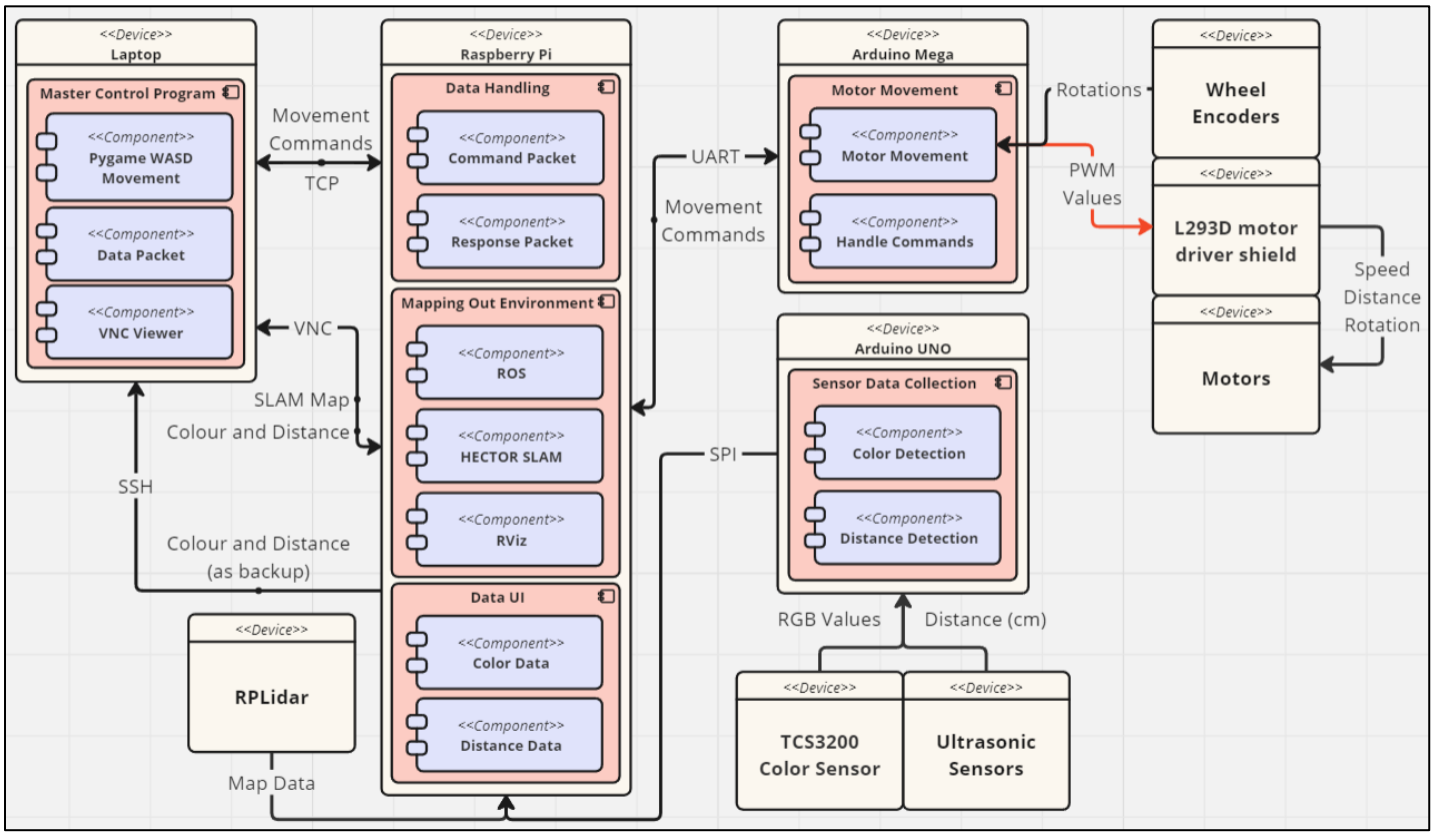
The Alex robot is an intricate system comprising essential components such as: RPLidar, Arduino Mega, Arduino Uno, Raspberry Pi 4, color and ultrasonic sensors, and four motors. These components communicate with each other through various channels, including UART, SPI, Wi-Fi, and wired connections.

To facilitate movement, commands are initiated via traditional keyboard movement inputs commonly utilized in gaming, employing the WASD keys for navigation and interaction, which are then transmitted remotely via VNC from a laptop. It is crucial for these components to synchronize seamlessly to ensure accurate processing of commands.

Furthermore, the Lidar readings play a pivotal role in mapping the maze environment. These readings are relayed back to the RPi for us to visualize the complete maze layout.

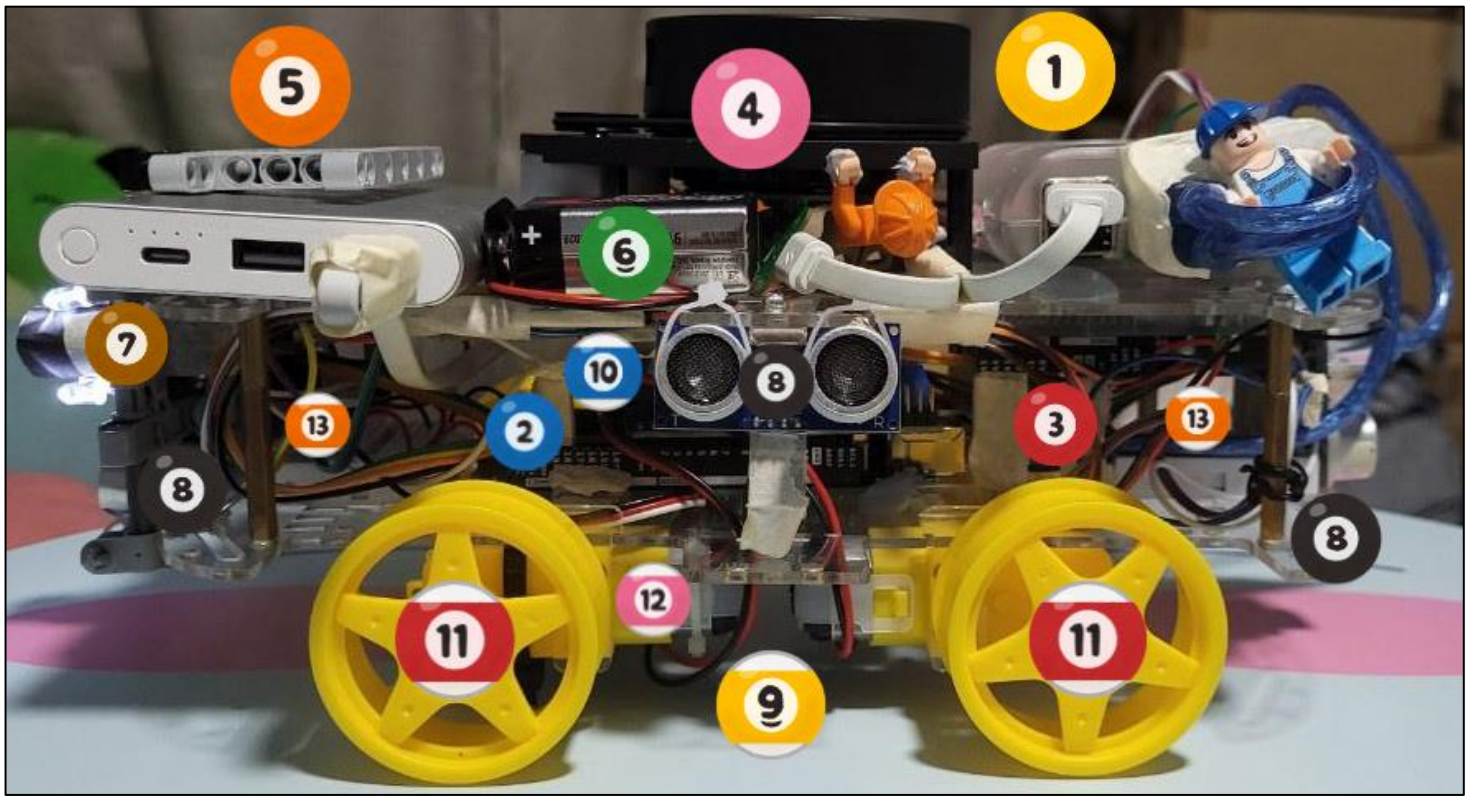
The overall algorithm governing the interaction of these components within the Alex robot is paramount. It orchestrates the collaborative effort of each component, ensuring efficient communication, accurate processing of commands, and effective utilization of sensory data for navigation and mapping purposes.

An UML deployment diagram for Alex is provided below. This highlights the connections between the various devices; notably the 4 key devices being the laptop, the RPi, the Mega and the Uno, and the flow of data between the various modules and components.



UML Diagram of the Alex system

Hardware Design

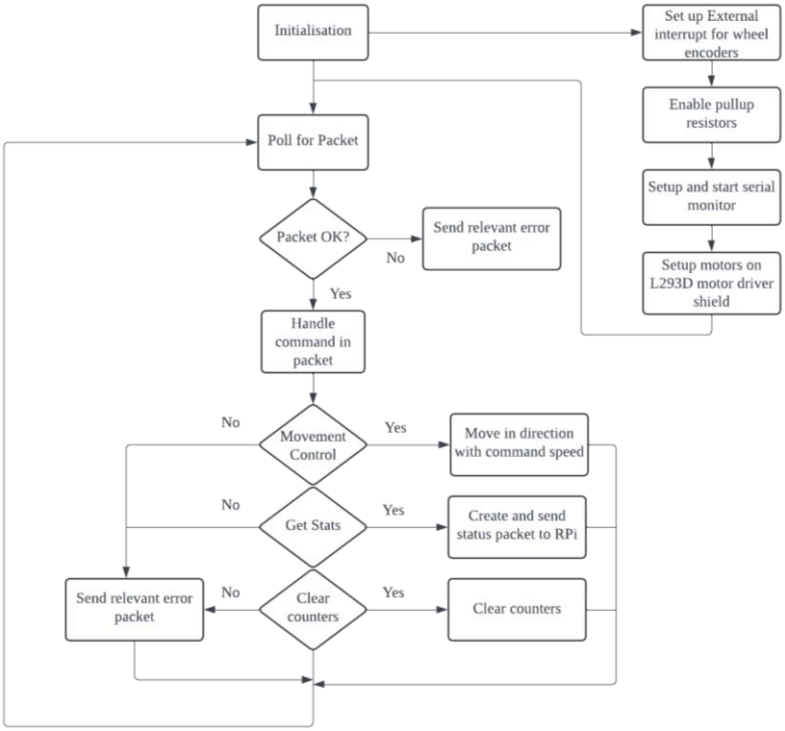


Left View of Alex

No.	Component	Purpose	Considerations
1	**Raspberry Pi**	The main controller of Alex. Receives commands from laptop via TCP and instructs the Mega2560 via UART. Receives map data from RPLidar and uses it to visually map out the surroundings on Rviz using Hector SLAM. Receives sensor data from the UNO via SPI to be transmitted and shown to us.	It is connected to the Mega and UNO via a USB-B adaptor cable, and to the RPLidar via a microUSB cable. The cables are then taped up to prevent it from protruding out of the Alex a lot.
2	**Arduino Mega2560**	Receives commands from the RPi and controls the L293D motor driver.	Receives power from RPi through the USB-B adaptor cable.
3	**Arduino UNO**	Connected to color and ultrasonic sensors. Sends the data obtained from the sensors to the RPi via SPI.	Receives power from RPi through the USB-B adaptor cable.
4	**RPLidar**	Performs 360° environment scans within a 6-meter range and sends the obtained data back to the RPi to map out the obstacles.	Lidar is placed on the axis of rotation such that there is minimal translation of the module during turning, allowing for greater ease and accuracy of mapping.
5	**Power Bank**	Provides power to the RPi, which then powers the Mega, UNO and Lidar.	Built a Lego holder to keep it in place to prevent it from dislodging when moving. Same Lego structure helps to fixate the front color and ultrasonic sensor together.

6	**9V Battery**	Provides power to the motors to facilitate movement of the Alex.	The 9V is much lighter than the 4 AA Batteries + battery holder, and takes much less space, but it runs out faster.
7	**TCS3200 Color Sensor**	Placed at the front of the Alex. It can differentiate between the red and green colored victims by obtaining their RGB frequencies and going through a color detection algorithm. The data is sent to the UNO which sends it to RPi via SPI.	Included skirting made of black paper placed around the photodiodes in the color sensor, to minimize light by the surrounding LEDs from affecting it and maximize its color detecting accuracy.
8	**Ultrasonic Sensors**	Capable of returning the distance between it and the nearest object in front of it. We use the data received in conjunction with the Lidar map to facilitate accurate movement of the Alex and prevent any collisions with obstacles. The data is sent to the UNO which sends it to RPi via SPI.	We placed a sensor at the front, back, left, and right of the Alex, to ensure we covered all sides. Its Zip-tied to the Alex to prevent any excessive movement to maintain accurate readings.
9	**Motors**	Controlled by the Mega and motor driver, it receives power from the 9V battery to move Alex. Configured to be able to rotate in both directions at variable speed.	Motors are placed pointed inwards to facilitate simpler and more compact wiring as they are nearer to each other.
10	**L293D motor driver shield**	Helps Mega control motor movement. Vcc is powered via the 9V battery.	The initial <AFMotor.h> library dependency was eventually replaced by our own bare-metal code.
11	**Wheels**	Allows Alex to move. 4 wheels, one at every corner, each rotated by the motors.	Removed the tires because its somehow easier to move and turn Alex without it.
12	**Wheel Encoders**	The 2 front motors are equipped with an encoder each that measures the number of ticks during rotation, letting us estimate the distance each motor has traveled.	This helps to provide a basic form of odometry, but is rather unreliable, hence we did not rely on this as much.
13	**Breadboard and Wires**	2 mini breadboards at the front and back of the Alex, each providing connection between components and the Mega/UNO.	Used for organization of the wires, which are then taped together for neatness and to help prevent them from accidental disconnection due to movement.

Firmware Design



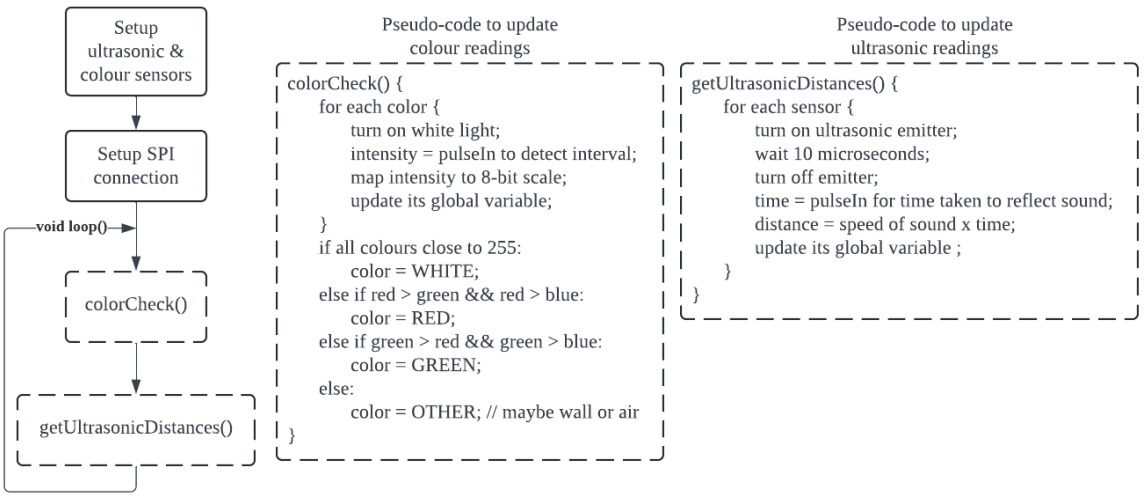
High-level algorithm on the Arduino Mega2560

The Arduino Mega2560, upon setup, constantly polls for a packet via the connection with the Raspberry Pi. Serial communication between the Mega and RPi is a serial connection of baud rate 9600, and of frame format 8N1. The serialized messages sent are in the following format:

```
typedef struct
{
  char packetType;
  char command;
  char dummy[2]; // Padding to make up 4 bytes
  char data[MAX_STR_LEN]; // String data
  uint32_t params[16];
} TPacket;
```

The structure is similar on both platforms for correct serializing and deserializing.

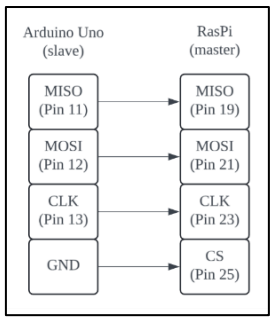
Serial Peripheral Interface (SPI) for the Arduino Uno



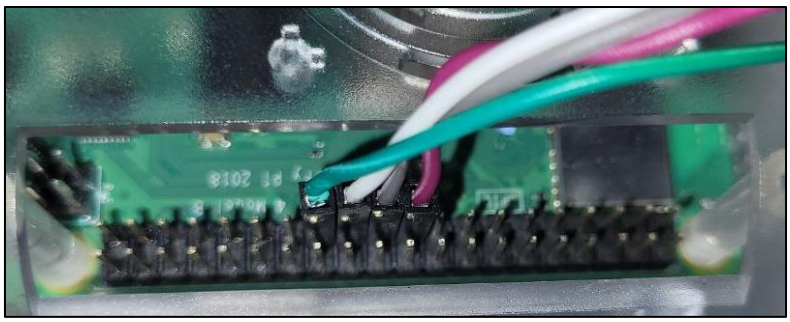
Algorithm used on the Arduino Uno

An uncommon approach that we used to obtain our color sensor and ultrasonic distance readings was using an additional Arduino Uno board. We felt that this method was necessary in the interest of hardware and code simplicity. The blocking delays used in the color detection algorithm interfered with movement serial transmissions when we attached the color sensor onto the Arduino Mega.

As such, the communication protocol of choice between the Arduino Uno and RPi was via Serial Peripheral Interface (SPI). This method can be used for accurate transmission of data over distances <10m, and supports higher clock speeds than I^2C (Brand, 2019). Our main consideration in using SPI was in not clogging the serial ports of the RPi. Numerous attempts in setting up 2 separate serial connections from the RPi to the Mega and Uno proved unsuccessful, with the "bad magic number" problem hinting that serial connections were suboptimal.



Pin mappings of SPI



RPi GPIO port used for SPI

As such, we opted for the hardware-based approach, which was to make use of the RPi's GPIO to connect to the Uno via SPI. This fulfilled our intention to send the ultrasonic and color readings over to the RPi. The SPI communication is an interrupt-based approach, whereby the RPi (master) will send a command over to the Uno (slave), and the Uno will return a byte based on the command, using a switch case. This cycle loops continuously while both the RPi and UNO are on and serially connected.

Color detection algorithm on Arduino Uno

We calibrated our color sensor to work optimally in a range of 5-10cm from the object directly in front of Alex. This was done with multiple considerations in mind. Firstly, we wanted Alex to be able to detect color accurately without running the risk of collisions with walls or victims. Next, we realized that the pulseIn readings (color frequencies) did not scale linearly with the distance from a certain colored object. To mitigate this inaccuracy, we mapped the RGB colors to an 8-bit scale (0 – 255), with the lowest end corresponding to no object in front, and highest end corresponding to a white paper placed 5cm in front of Alex. We also set a threshold value (~30) for the difference between the RGB values to verify the color white since their values will be very similar. This value is pre-defined by us based on tests that we ran in various lighting conditions. This helps us to correctly identify other colors, namely red and green, instead of white when the object in front is very close and reflecting a lot of light into the sensor.

A noteworthy point in the color detection algorithm is that it was only accurate at distances in our optimum range (5-10cm from the object). Too close and any object would appear to be white; too far and the object would not be detected at all. Furthermore, the walls revealed in the trail run were turquoise, which showed up as green on our color detection. To tackle this issue, additional human observation was done by our esteemed teammate Isaac during the runs to affirm the color detection. He was able to identify the color of the targets, based off the raw RGB values from the Uno, after countless hours of practice (The walls read high values of R compared to the green victim, who reads almost 0 R value, even though both reads high value of G). This ensured our color calls during the final run were accurate and correct, not accidentally mistaking any walls as victims.

Similarly, the ultrasonic distance readings were closely monitored to avert a collision with obstacles or targets in the maze. These readings were of exceptional importance when Alex was extremely close to the walls, where the RViz map lacked precision in collision detection. This proved to be paramount during the process of parking Alex, where our data-reading copilot announced the distances to our meticulous and capable driver, Mark.

Serial communication for motor controlling and bare metal code for motor driver

To control four motors, we utilized a motor driver shield connected to the Arduino Mega2560, which receives commands from a Raspberry Pi via the USART communication protocol. Upon receiving a packet from the Raspberry Pi, the packet is deserialized to determine its type. If it's an error packet, a corresponding response is sent back to the Raspberry Pi based on the error type. For command packets, they are processed using the handleCommand() function.

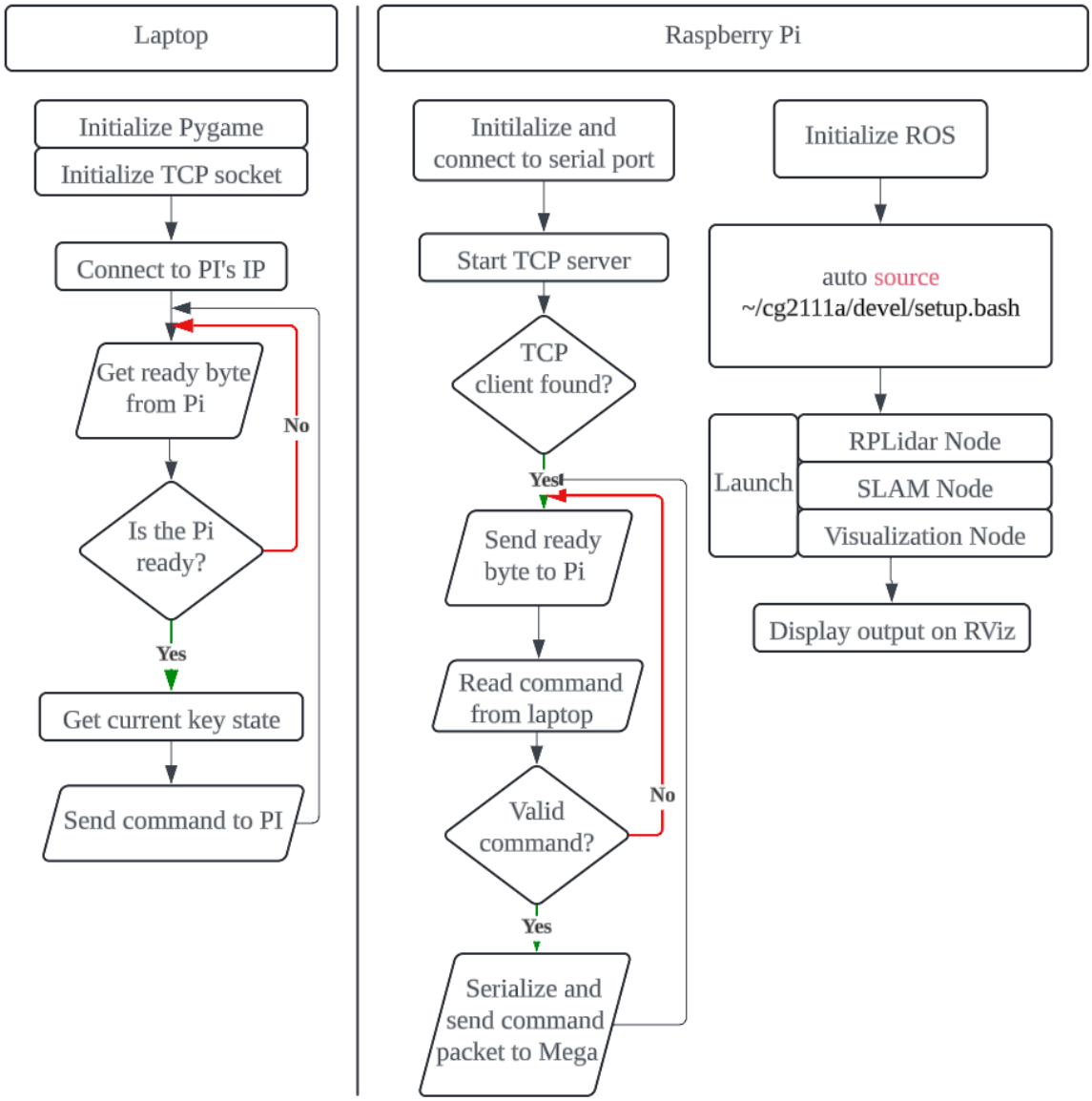
In terms of movement commands, unlike the older Alex code for the Mega2560 that requires specifying distance and speed, we simplified this by setting a fixed distance and speed for every movement. The only input needed is the direction of movement, which is sent from our control laptop through a TCP connection to the Raspberry Pi. The Raspberry Pi then sends a movement command with the predetermined distance and speed to the Mega2560, which in turn drives the motor.

For controlling motor drivers in bare metal code, we reference the Adafruit Motor Shield library. Motor speed is regulated by controlling the PWM signals of four counters: OC1A, OC3C, OC4A, and OC3A, corresponding to port PB5 (pin 11), PE5 (pin 3), PH3 (pin 6), and PE3 (pin 5), respectively.

To control motor direction, the motor shield uses a third integrated circuit that receives commands from the Mega2560. We maintain an 8-bit latch state variable for the eight input ports (A and B) of two motor drivers, with each A and B pair controlling one motor. To move a motor forward, we set port A to 1 and port B to 0. For reverse movement, these values are reversed, and setting both ports to 0 stops the motor. After updating the latch state variable, the output latch state is sent to the shift register using the latch_tx function. The transmission begins by setting the latch port to 0, along with the clock port. We then transmit the 8 bits from the latch state variable, starting from bit 7 to bit 0. We end the transmission by setting the latch port to high, thereby securing the transmission line.

Software Design

Algorithm on the Raspberry Pi/ Operator Laptop



Flowchart for Laptop and RPi Communication

Teleoperation of Alex

The teleoperation of Alex is one of the most crucial parts of the whole project. As we are required to navigate and map out the environment as fast as possible without any collisions, the two main considerations for teleoperation are the **speed** and **accuracy** of Alex. With this in mind, we decided to implement continuous movement for Alex. Continuous movement on Alex allows us to move freely at any distance and rotate at any angle without the need to key in commands in the command line manually. As a result, the time lag between issuing commands is greatly reduced, thereby increasing the efficiency of the mapping process.

We further enhanced the continuous movement by integrating ultrasonic sensors on all four sides of Alex. This setup enables our navigator, Mark, to skillfully maneuver through the maze, easily avoiding obstacles and efficiently navigating through tight spaces.

```
# Set keyboard mappings
KEY_MAP = {
    pygame.K_w: 'f',
    pygame.K_a: 'l',
    pygame.K_s: 'b',
    pygame.K_d: 'r'
}
```

Key mappings for movement control

Continuous movement is implemented using the Pygame library to capture the current state of movement keys —'w', 'a', 's', and 'd' for forward, left, backwards, and right movements, respectively. These inputs are then transformed into commands—'f', 'l', 'b', 'r'—that are transmitted to the Raspberry Pi and subsequently to the Arduino Mega for execution. A stop command is issued when no keys are pressed, halting Alex's movement immediately.

During the development of this feature, we encountered a "BAD MAGIC NUMBER" error, which we diagnosed as an overflow issue in the Arduino Mega's buffer due to the continuous stream of commands. This overflow disrupted the reading of packet data sent over the serial port, causing subsequent command packets to be affected, which prevented Alex from executing any movements. After exploring some potential solutions, including increasing the buffer size or altering the packet structure, we opted to implement a 0.2-second delay between commands. This simple adjustment not only resolved the buffer overflow issue but also maintained the responsiveness of Alex's movements, striking a balance between functionality and complexity.

ROS, LIDAR, SLAM and RViz

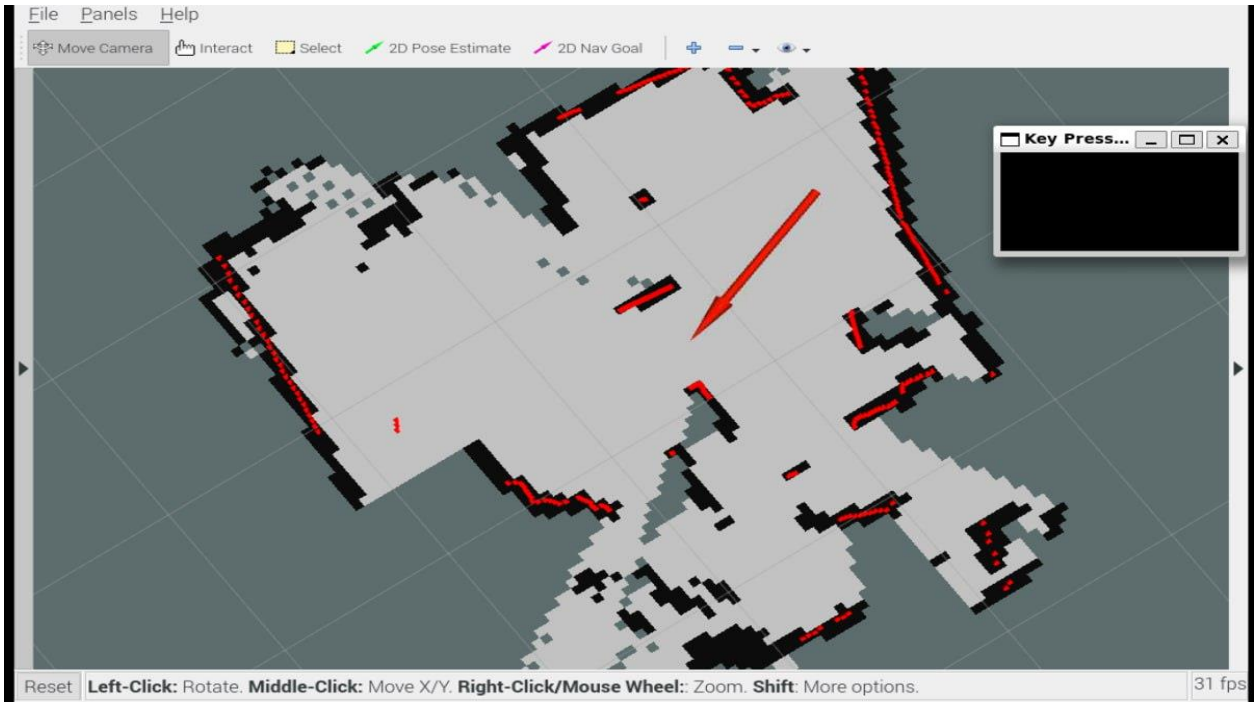
In our project, we opted to utilize ROS (Robot Operating System) to control the LiDAR for the Alex robot. We adopted the pre-built PurplePet image from the tutorial, which meets our needs and saves time on manual installation and setup of ROS. We configured ROS to handle three nodes: data collection, processing, and visual rendering. Data is initially captured by an RPLIDAR node that broadcasts raw data. A SLAM node then processes this data, applying the SLAM algorithm to generate a preliminary map sketch. Finally, the map is rendered and displayed through a Visualization node.

To explain the SLAM algorithm simply, it selects several random data points from a scan, calculates the best fit line for these points, and checks if sufficient data points fall within a predefined threshold distance from this line. If so, these points are recognized as part of an object, like a wall or an obstacle. The selected data points are then removed, the object is outlined, and its information is forwarded to the Visualization node to be represented on the map. This process repeats until the program is terminated or there are too few data points remaining to identify new objects.

The SLAM algorithm also assists in determining the robot's position on the map, which generally requires environmental data from LIDAR and odometry data, such as readings from hall sensors. However, the specific SLAM algorithm we use, Hector SLAM, does not require

odometry data. It estimates the robot's location by comparing previous and current maps to match them. This method allows us to operate independently of the wheel encoder readings but can lead to inaccuracies if Alex moves too quickly, potentially causing map distortions. Therefore, we typically maintain a slow speed to ensure that HECTOR SLAM can accurately estimate our location.

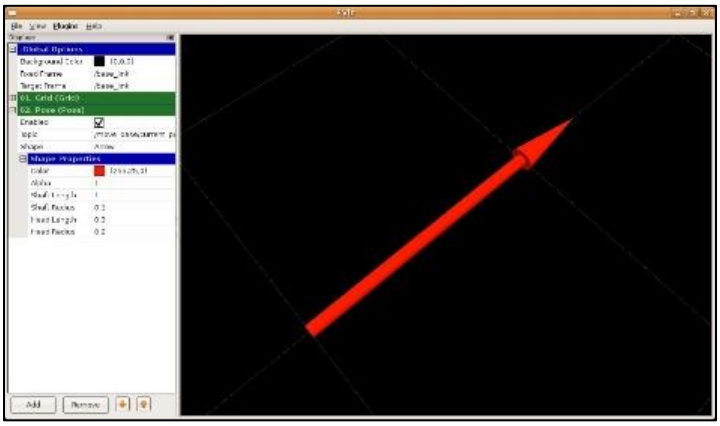
For map visualization, we use RViz (ROS Visualization), a 3D visualization tool included in ROS. After Hector SLAM identifies an object, it sends this data to RViz, which then displays the object, such as a wall represented by a black line, on the map. The location of Alex is indicated by an arrow, as calculated by Hector SLAM.



One of our sample run's Rviz map

Rviz Maneuvering Nuances

To further aid the navigation of Alex in tight spaces, we modified the pose arrow within RViz to represent Alex's dimensions more accurately. Specifically, we altered the arrow's head radius to 0.2 and its shaft length to 0.9. With these modifications and with sufficient practice with tight spaces with our custom maze, our controller can be more confident in navigation, resulting in shorter overall exploration timings.



Arrowhead represents Alex's dimensions in RViz

Lessons Learnt - Conclusion

Lessons learnt in this project:

1. We recognized the critical role of rehearsals in preparing for significant events. By repeatedly practicing the workflow of the run, we were able to pinpoint potential issues that might arise. For instance, during the initial trial run, we discovered that the SPI was malfunctioning, preventing us from accessing the color sensor data when Alex entered the maze. This early detection allowed us to refine our workflow, ensuring we only activated the LIDAR Hector SLAM algorithm when Alex was in the maze. As a result, our final run proceeded much more smoothly.

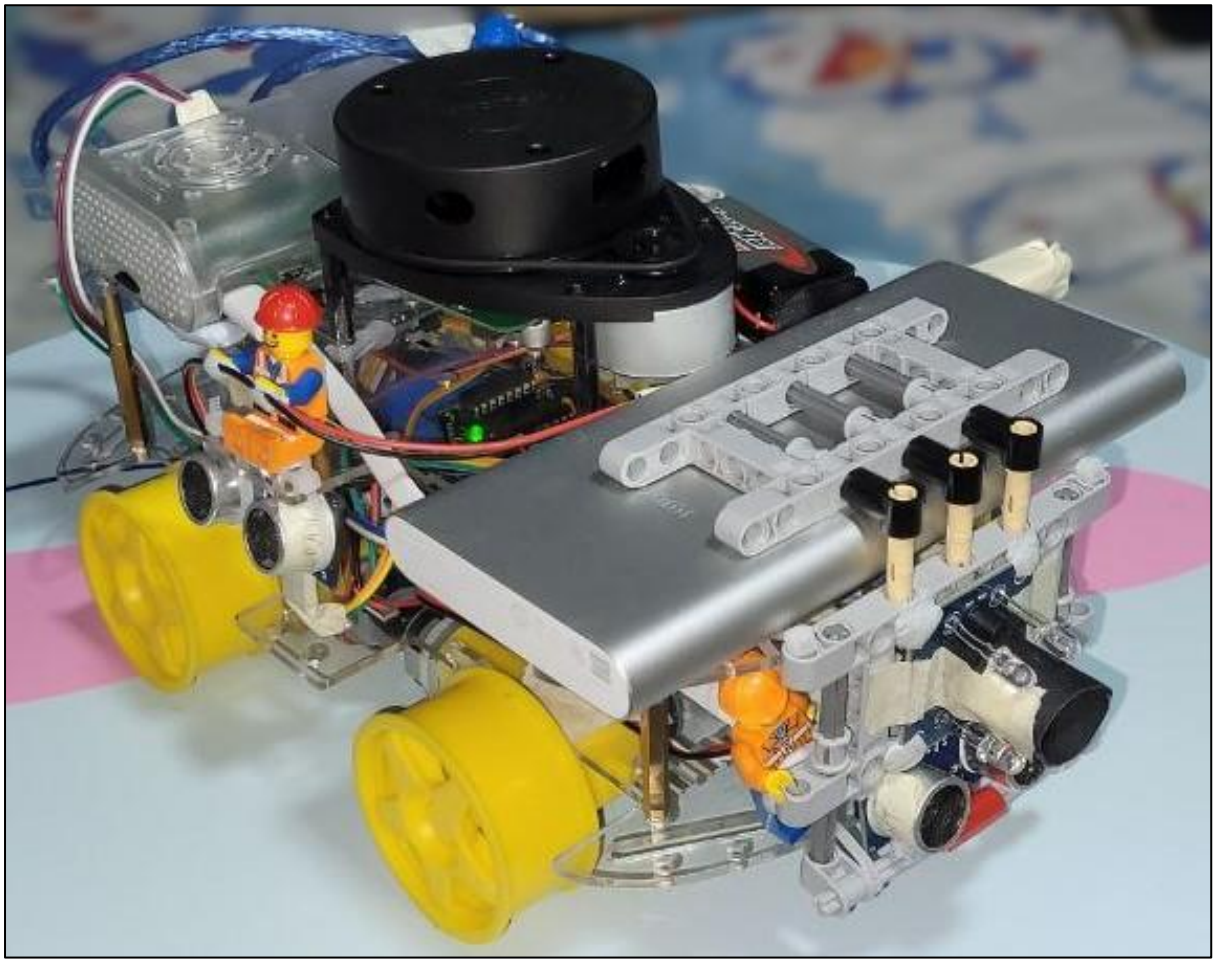
2. We discovered that by networking with other teams, our productivity significantly increased. Collaborating allowed us to save time by learning from each other's mistakes, such as avoiding problematic code that leads to magic number errors. We also gained valuable insights from the diverse approaches of different teams. For instance, we were inspired to offload sensor processing to a separate Arduino Uno and to adjust the size of the arrow in ROS to better represent Alex's orientation and movement. These collaborative experiences not only enhanced our project but also enriched our understanding and approach to complex problems.

Mistakes made as a group:

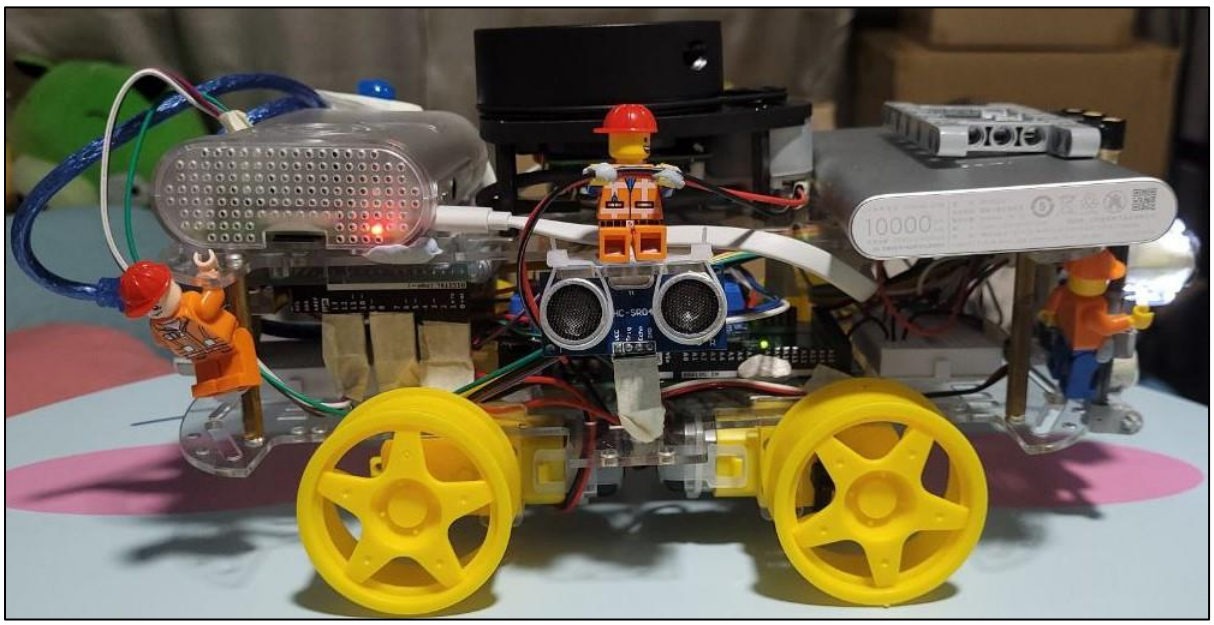
1. Our enthusiasm to implement our systems prematurely, without a thorough understanding of their mechanisms, led to multiple complications with the TCP and SPI interfaces. We attempted to troubleshoot these issues without sufficient background knowledge, which resulted in significant time wasted and frustration during the debugging process.

2. In retrospect, we overlooked several potentially simpler solutions to our problems encountered. Our struggle with the SPI connection was a prime example, which consistently produced garbage values. Rather than persisting with SPI for transmitting ultrasonic and color sensor readings, we should have considered alternative methods sooner. We eventually discovered that using a CLI-based serial monitor to view Arduino Uno's serial output—executed via the command `sudo screen <port name> <baud rate>`—allowed us to monitor the device remotely on our laptop. This approach was not only simpler but would have significantly reduced the time spent on debugging SPI-related hardware and software issues.

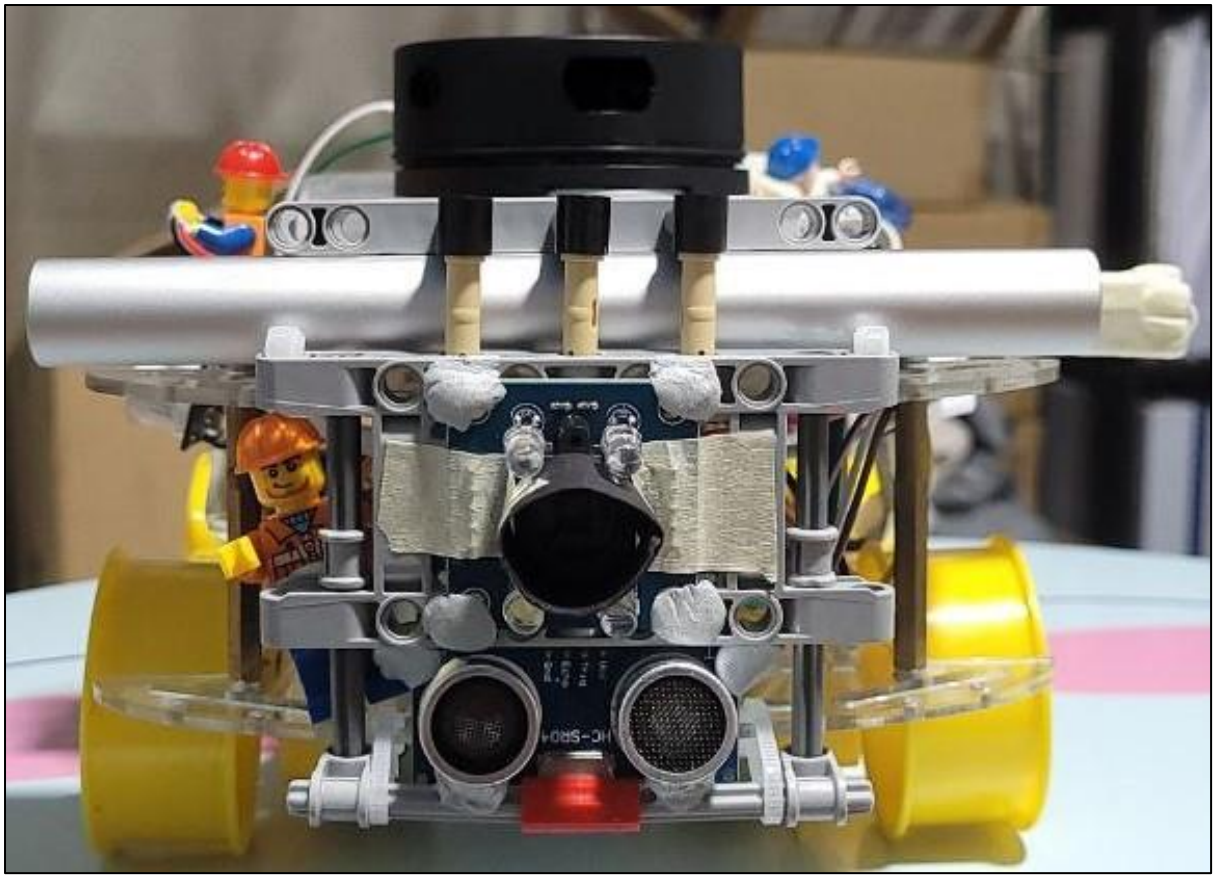
Appendix



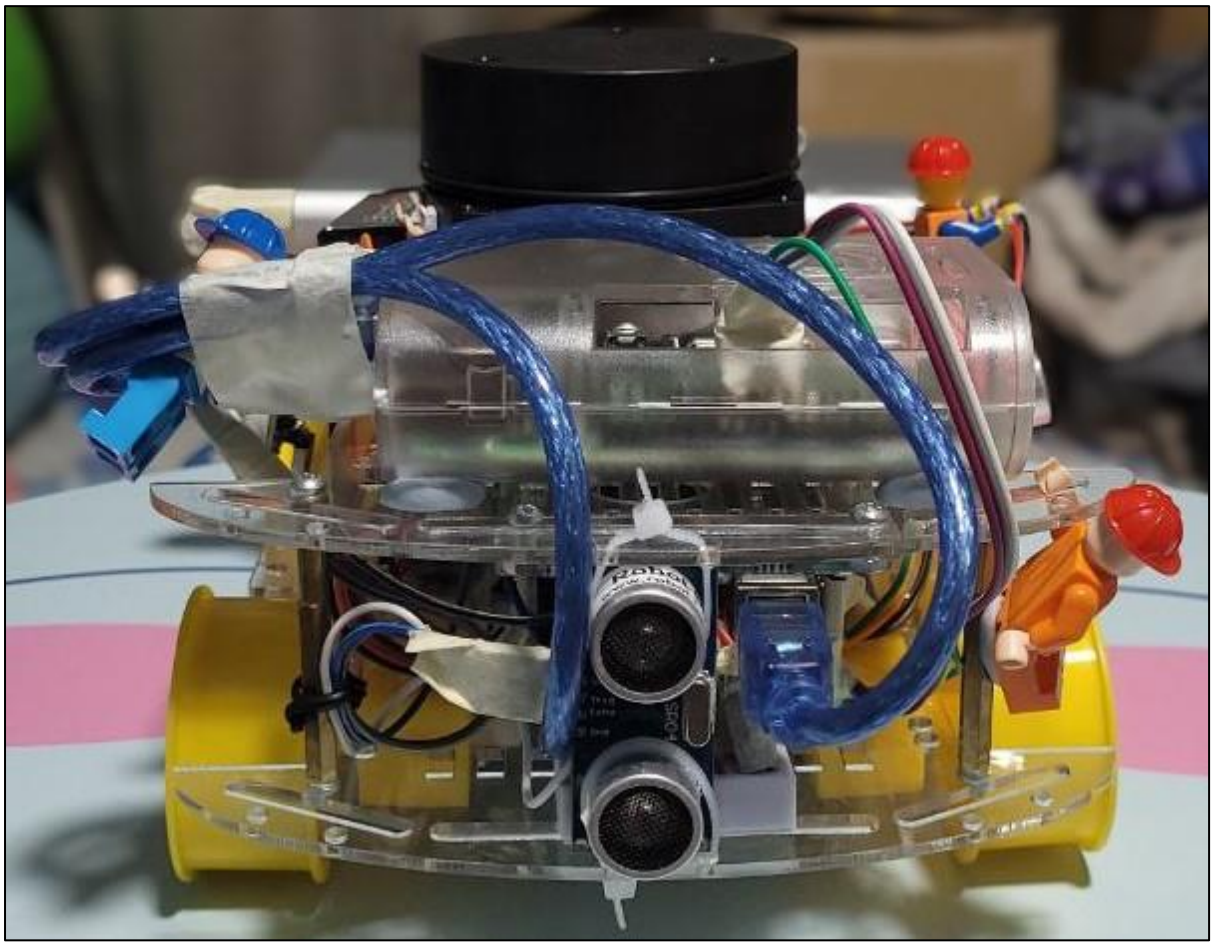
Axonometric View of Alex



Right View of Alex

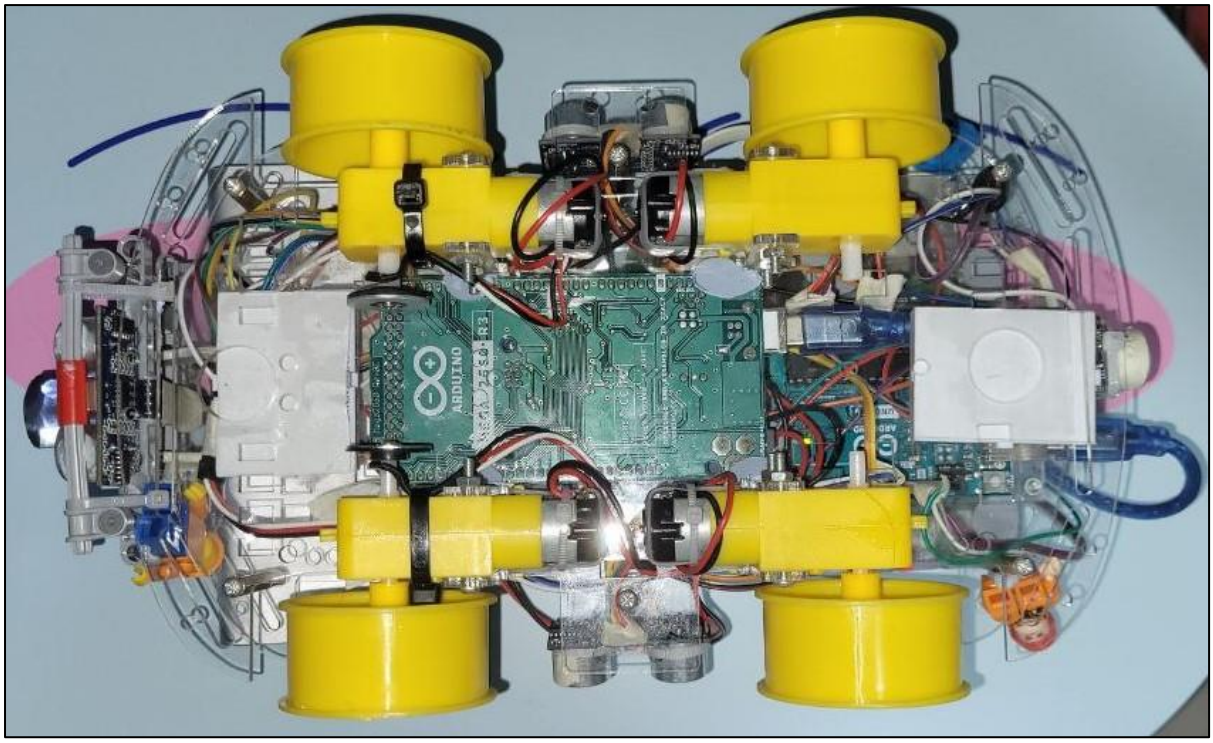


Front View of Alex

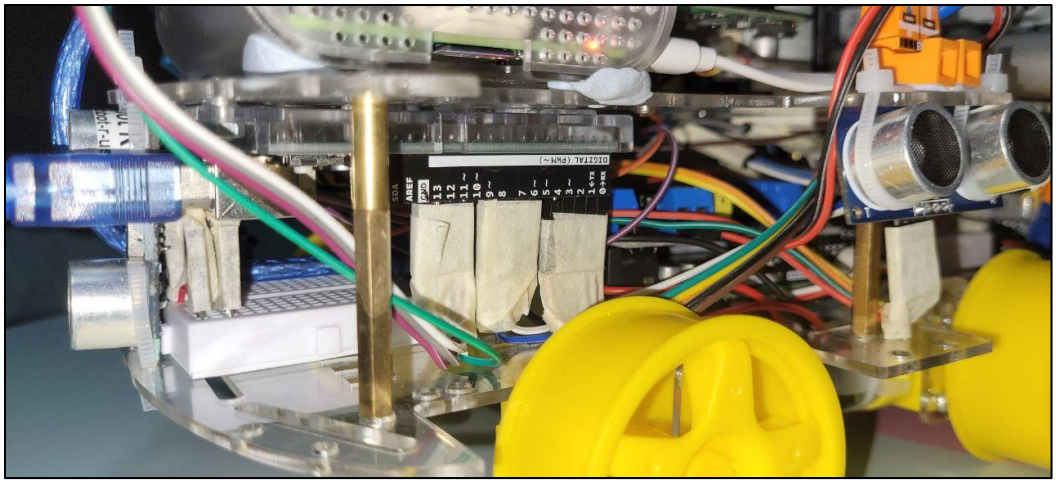


Back View of Alex

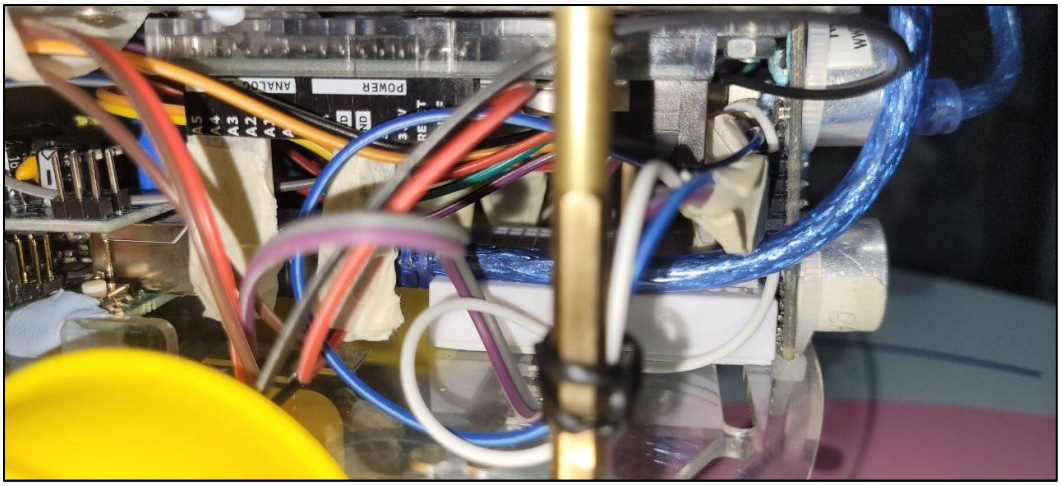
Top View of Alex



Bottom View of Alex



Right View of Arduino UNO



Left View of Arduino UNO

One of our custom mazes designed to challenge the Navigator

Afterword

To close this off, Alex has been a significant eye-opener in our journey into the Internet of Things (IoT). It has provided us with invaluable experience and insights that will undoubtedly shape our future. We're glad that our countless hours of pouring blood, sweat, and tears into Alex bore fruits of success in the final run, where we managed to complete the course in the short time of 3 min 43 sec, successfully identifying both victims without any collisions! We would like to dedicate our special thanks to our TA Justin, who has gone out of his way to guide us throughout our project, answering all our annoying questions; As well as Prof. Henry Tan, who always succinctly answers our late-night emails regarding the project, encouraging us to push ourselves! Working on Alex has been not just a project for us, but rather a journey, a steppingstone for us towards becoming better engineers. 😊

Final Group Photo with our Alex

References

Our GitHub Repository
https://github.com/Markneoneo/ALEX-Project.git

Images
[1] Goode, L. (2020, October 26). *Boston Dynamics' Robots won't take our jobs ... yet*. https://www.wired.com/story/get-wired-podcast-14-boston-dynamics/

[2] Avetics. (n.d.-a). *DJI Enterprise Store*. https://www.avetics.com/dji-enterprise-store/dji-m300

Sources
Spot to the Rescue. (2024, March 26). Boston Dynamics. https://bostondynamics.com/blog/spot-to-the-rescue/

Matrice 300 RTK - Industrial grade mapping inspection drones - DJI Enterprise. (n.d.). DJI. https://enterprise.dji.com/matrice-300

Brand, T. (2019, July 17). *Isolated SPI communication made easy.* Analog Devices. https://www.analog.com/en/resources/technical-articles/isolated-spi-communication-made-easy.html#:~:text=While%20the%20SPI%20communication%20method,line%20resistance%20of%20long%20cables.