CG2111A Engineering Principle and Practice
Semester 2 2024/2025

"Alex to the Rescue"

Final Report

Team: B04-6B

Name	Student #	Sub-Team	Role
SIM CHIN KAI ALSON	A0307409E	SOFTWARE	TLS + ROS
LIM SWEE HOW GABRIEL	A0307733E	HARDWARE	Circuitry
SHAH KUSHAL HITESH	A0308033R	FIRMWARE	Sensors + RPI
VANSH PURI	A0307563A	SOFTWARE	Arduino Code

Content

Section 1 Introduction

The “Alex to the Rescue” challenge is a simulated disaster-response scenario in which a tele-operated robot must remotely navigate an unfamiliar terrain to identify victims and map out the environment. Our robot, Alex, is engineered to fulfill this mission by operating under secure remote control, relaying real-time sensor feedback, and executing precision movements based on user commands.

The system is expected to:

- Be controlled remotely through a laptop, sending and receiving secure TLS-based command and sensor data.
- Execute directional commands such as moving forward, backward, and turning with reliable accuracy using PWM-driven motors and wheel encoders.
- Detect color-coded astronauts using a TCS3200 color sensor: red objects require evacuation, green ones require medpaks drop-off.
- Map the entire environment using a 360° LiDAR sensor integrated with ROS and Hector SLAM for localization and navigation.

Alex is equipped with a range of hardware, including an Arduino, Raspberry Pi 4, LiDAR, camera, color sensors, motors, ultrasonic sensors, servo motors and wheel encoders, and software components such as ROS, TLS protocols, and serialization mechanisms to ensure robust communication between the Raspberry Pi and Arduino. The integration of these modules allows Alex to deliver a functional and semi-autonomous search-and-rescue platform suitable for operating in disaster-stricken environments.

Section 2 Review of State of the Art

2.1 KARO Mobile Robot

The KARO mobile robot (see Appendix Figure 8) is a modular platform developed for disaster-response operations, particularly in inaccessible or hazardous urban environments. It is equipped with advanced sensors such as LiDAR for spatial mapping, gas sensors for chemical detection, and acoustic sensors for locating survivors. The robot is capable of both tele-operation and limited autonomous navigation, using a tracked wheel design for enhanced mobility over rubble and uneven terrain. KARO has been successfully deployed in simulated disaster drills, such as post-earthquake urban collapse scenarios, where its ability to relay real-time environmental data proved invaluable. The platform supports interchangeable sensor modules, allowing it to be adapted for specific missions, such as radiation mapping or fire zone inspection. Its strengths include rugged terrain traversal, adaptability for different payloads, and remote operation capability. However, it suffers from limitations in payload

capacity, short battery life during prolonged missions, and a requirement for trained personnel to manage its sensor suite effectively.

2.2 Spot by Boston Dynamics

Spot (see Appendix Figure 9) is a quadruped robotic platform engineered for a wide range of applications, including inspection, surveillance, and search-and-rescue. Powered by the Robot Operating System (ROS), Spot uses LiDAR, stereo cameras, and IMUs to navigate autonomously or be remotely piloted in complex environments. Spot's unique legged design allows it to traverse stairs, slopes, and debris, making it ideal for collapsed-building scenarios or forested disaster zones. In a recent earthquake drill, Spot was used to explore partially collapsed structures and deliver real-time video and environmental data to responders. Its modularity allows it to be equipped with thermal cameras, radiation sensors, or even communication relays for use in connectivity-dead zones. Spot's primary advantages lie in its mobility and customizability. However, it has limitations in battery life (approximately 90 minutes), high procurement cost, and a limited payload capacity, which constrains its ability to carry large equipment or extract victims.

Section 3 System Architecture

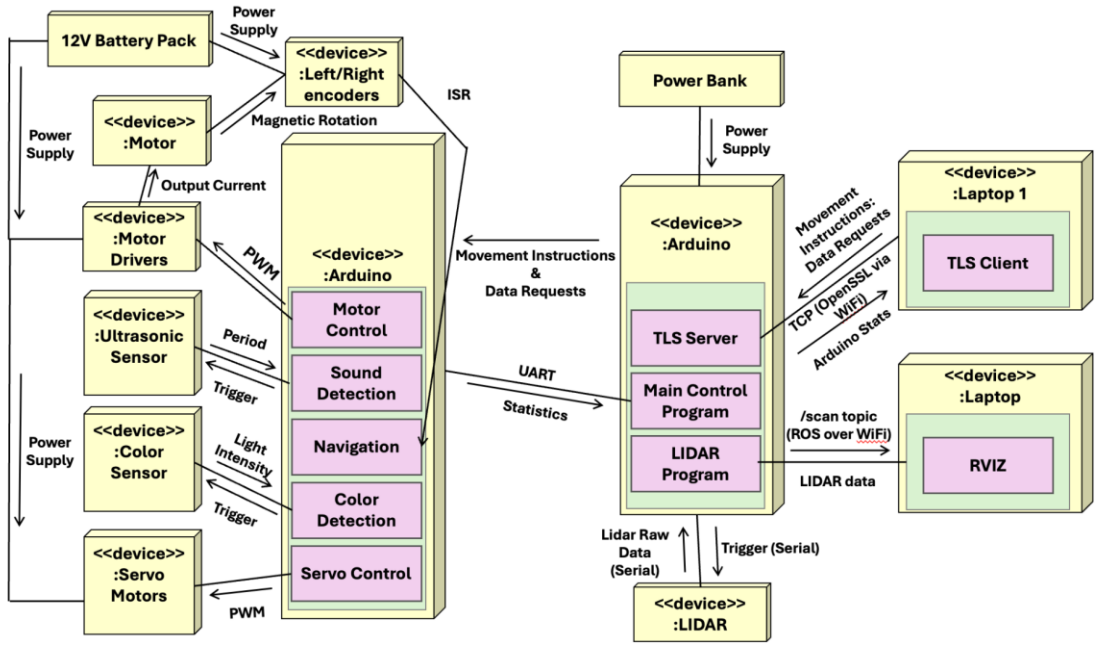

Figure 1: UML Deployment Diagram

Section 4 Hardware Design

4.1 Robot Design

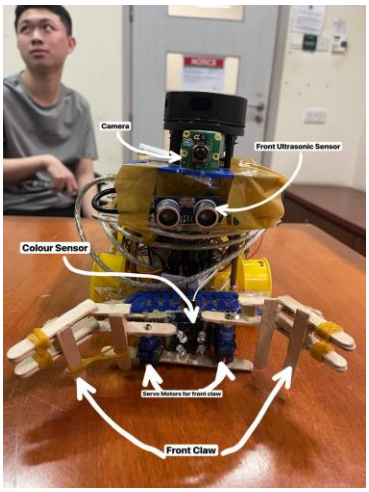

Fig 2: Front view of Alex

Fig 3: Back view of Alex

Fig 4: Right Top Side view of Alex

Fig 5: Left Top Side view of Alex

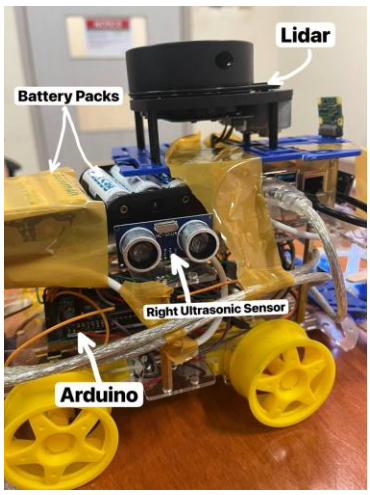

Fig 6: Close right view of Claw

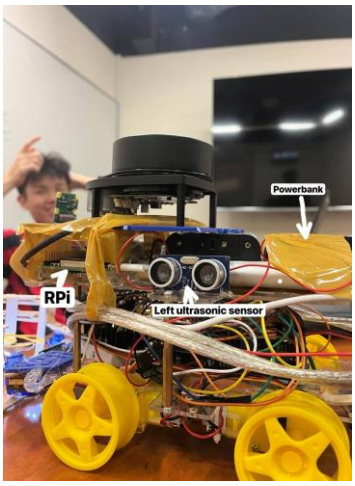

Fig 7: Close Left view of Alex

4.2 Mechanical Construction

Alex is built on a compact, layered chassis for robustness and modularity. The base supports four DC motors with Hall-effect encoders, enabling differential steering with real-time tick tracking. On the upper levels, the Raspberry Pi and Arduino are securely mounted using tape and screws, allowing for easy reconfiguration while keeping wiring organized and accessible.

The chassis layout is optimized for maneuverability within tight environments, while providing ample mounting points for all hardware components: LiDAR, color sensor, ultrasonic sensors, dual claws, and power systems. Cable routing and component spacing were planned to reduce signal interference and minimize hardware instability during motion.

4.3 Custom and Integrated Hardware Components

4.3.1 Dual Claw Mechanism for Simultaneous Object Handling

A key challenge was designing Alex to perform two critical tasks at once: transporting a medpak and rescuing red astronauts. Since the robot starts every mission carrying the med-pack, it needs to be able to secure the package while remaining free to interact with red astronauts immediately upon detection.

To meet this need, we implemented a dual claw system:

- The front claw, made from two servo motors and reinforced ice cream sticks, was used to align astronaut dummies for color sensing. Because the TCS3200 sensor requires precise positioning, the claw opens to center the object before a color reading is taken. If a red astronaut is detected, Alex can immediately proceed to drag the object from the front, while continuing to carry the med-pack at the rear.

- The rear claw securely holds the med-pack throughout the mission, freeing up the front claw for other tasks. Positioned at the back, it ensures the TCS3200 color sensor remains unobstructed for accurate color detection. The simple, servo-driven mechanism allows quick deployment of the med-pak when needed, enabling flexible decision-making during navigation.

This system enabled Alex to handle both objectives of rescuing and transporting in parallel without compromising detection accuracy or mechanical stability.

4.3.2 Ultrasonic Sensors

Three ultrasonic sensors—mounted at the front, left, and right—provide Alex with environmental awareness. Connected to the Arduino and read using pulseIn(), these sensors serve several purposes:

- Preventing collisions by alerting the robot to nearby obstacles.

- Helping align astronauts with the color sensor during detection and allowing us to know when the astronaut is close enough to the robot to be picked up by our claw

- Enhancing user control by offering real-time distance feedback during manual navigation.

4.3.3 Wheel Encoders and Tick-Based Motion Control

Each DC motor is equipped with a Hall-effect sensor and encoder that counts rotational ticks. These ticks are central to Alex's movement system. Rather than relying on time-based motor commands, the robot moves using tick-based control: when a movement or turning instruction is issued, the Arduino calculates the exact number of ticks required and drives the motors until the tick count is reached.

Using Interrupt Service Routines (ISRs), the Arduino tracks the encoder feedback in real time, ensuring consistent and accurate motion without polling. This system provides high precision for both linear and angular movement, improving command repeatability and obstacle avoidance.

4.3.4 Colour Sensor Mounting and Detection

The TCS3200 color sensor is mounted just behind the front claw, positioned at a fixed height and angle for consistent RGB detection. The front claw plays a crucial role in placing astronaut dummies within the sensor's optimal reading range. RGB values are collected via the pulseIn() function and analyzed using a least-error algorithm based on pre-calibrated red and green color profiles. This setup ensures reliable differentiation between red and green astronauts during live operation.

4.3.5 Modular Battery Architecture

Power delivery is split across two systems for reliability and layout optimization:

- The motor and Arduino subsystems are powered by two 4x1.5V AA battery packs wired in series. This was a deliberate upgrade from the original 7xAA pack, which was found to be bulky, heavy, and mechanically awkward. The twin-pack setup allows for better weight distribution and easier integration within the robot's limited chassis space.

- The Raspberry Pi is powered separately using a USB power bank, ensuring that voltage dips from motor activity do not interfere with ROS operations or communication protocols.

This modular power design minimizes electrical noise and improves both system stability and battery efficiency.

Section 5 Firmware Design

5.1 High-Level Arduino Algorithm

5.1.1 setup():

Firstly, the setup function will set all the Arduino pins used to the correct input/output mode and to high/low output if necessary. It also sets up the registers used for serial communication, enables interrupts for the hall sensors, and sets the frequency scaling of colour sensor input to 20%.

5.1.1 loop():

Next, in the main loop function, just 2 things are done every cycle. **First,** the readPacket() function is called, where the UDR0 buffer is checked to see if there is any new command sent by the RPi. If so, then the new command is read and executed. If not, then the **second** thing done is to check if Alex is currently tasked with moving or turning. If so, then checks are made to see if Alex has moved or turned the required amoun,t and to stop Alex if that amount has been reached. These 2 checks are done repeatedly throughout the entire operation of Alex, unless in the middle of executing a received command.

There are 13 different commands that the Arduino has been programmed to accept from the RPi. They are all defined in the TCommandType enum (see Appendix Figure 11).
They do the following:

- triggers the colour sensor and returns the detected colour
- triggers ultrasonic sensors and returns the measured distances
- move Alex forwards and backwards amounts at a preset power
- make Alex turn left and right amounts at a preset power
- Triggers the servo motors to open and close the arms to hold/drop the astronaut and the medpak bottle

The forward, backward, left, and right movement commands all have preset power values. This is because manually inputting power values (e.g. F 25 90) would be extremely tedious, slow, and repetitive. Hence, by merging those parameters into a single input, with the power always set at 100, it becomes much faster to move Alex. For example, pressing F10 on the keyboard would instantly get Arduino to interpret it as F10 100, and the robot would move the distance of 10 at 100% power.

The get_color command will call the colour sensor function [match_colour();], in which 2 other functions were called [measure_colour(); identifyColour()], which will be described in detail in section 5.3.1. The get_dist command will trigger the ultrasonic sensors [read_US();]. Both of these functions will then update a global variable that will hold the obtained sensor data (variables frontD, leftD, rightD for the ultrasonic command and variable color for the colour sensor command). Then, the sendStatus() function is called which creates a new TPacket struct called statusPacket and fills up its params[] array with the sensor data from the above mentioned global variables. Then, that packet of data is sent via the sendResponse()

function to the RPi, which will send it to the laptop to be displayed to the user.

5.2 Communication Protocol

5.2.1 Format Of Messages:

Messages to be sent are contained within the TPacket struct, as defined below.

```
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;
```

5.2.2 Reading Of Messages:

1. Creation of a struct variable recvPacket of type TPacket [TPacket recvPacket;]
    a. Holds any command that is sent by the RPi.
2. readPacket() function is called [TResult result = readPacket(&recvPacket);]
    a. with the created packet variable (recvPacket) as its input.
    b. Then the readSerial() function is called [len=readSerial(buffer);],
        i. the char buffer array with the data sent by RPi is filled in, which is accessed through the UDR0 register
        ii. while (UCSR0A >> 7) buffer[count++] = UDR0;
3. Next, deserialize() function is called [deserialize(buffer, len, packet);]
    a. it deserializes the char buffer array and saves the received data into recvPacket
4. Depending on the packet type received, different actions are taken:
    a. PACKET_OK
        i. handlePacket() function is called [if(result == PACKET_OK) handlePacket(&recvPacket);],
        ii. calls the handleCommand() function [handleCommand(packet);]. In this function, the command parameter of the recvPacket struct is read to see which of the 13 commands has been received. Firstly, the sendOK() function is called [sendOK();], which sends a packet back to the RPi of command RESP_OK. Then, the appropriate action is taken, based on the command parameter.
        iii. code returns back to void loop().
    b. PACKET_BAD or PACKET_CHECKSUM_BAD,
        i. a packet is sent back to the RPi of command RESP_BAD_PACKET or RESP_BAD_CHECKSUM [sendBadPacket(); or sendBadChecksum();].
        ii. Then the code returns back to void loop().

5.2.3 Sending Of Messages

Messages are sent to the RPi via the sendResponse() function [sendResponse(packet);]. Here, the serialize() function is called [len = serialize(buffer, packet, sizeof(TPacket));], which will fill a char buffer array with the serialised bytes of the packet to be sent. Then, the writeSerial() function is called [writeSerial(buffer, len);] which will repeatedly fill up the UDR0 register until every byte of data from the buffer has been sent to the RPi.

```
for (int i = 0; i < len; i++) {
   while ((UCSR0A & 0b00100000) == 0);
   UDR0 = buffer[i];
}
```

5.3 Additional Noteworthy Software

5.3.1 Colour Detection Algorithm

The TCS3200 colour sensor uses an array of photodiodes, each filtered to detect different colours: red, green, blue, and clear. By shining white LEDs on the object being detected, the sensor measures the intensity of the red, green, and blue light reflected back. It does so by generating a square wave output, where a higher intensity of a given colour results in a higher frequency.

The Arduino uses the pulseIn() function to measure the duration of each pulse in the square wave output from the TCS3200. This duration corresponds to the inverse of the frequency, meaning that a shorter pulse duration corresponds to a higher intensity of that colour, and a longer pulse duration indicates lower intensity. This behavior is key to understanding how the sensor's readings can be interpreted. In our specific case, we needed to detect two colours: red and green. We ignored the blue readings because they were inconsistent and not necessary for distinguishing between red and green.

```
if (red < green - redThreshold) {
  return RED
} else if (green < red - greenThreshold) {
  return GREEN
}
```

How the Code Works (the Arduino code for the algorithm is in Appendix Figure 12):
1)Measure color intensity:
The TCS3200 sensor generates a square wave for each colour (red, green, and blue). The Arduino measures the pulse duration using pulseIn(), which gives a value that corresponds to the time the signal stays high. A shorter pulse duration means a higher intensity of that colour, while a longer pulse duration indicates lower intensity.

2) Threshold Comparison:
The measure_color() function calls readFrequency() three times to measure red, green, and blue intensity. readFrequency() sets the S2 and S3 control lines to filter for the desired photodiode group, waits briefly, and then takes multiple pulse readings to average out noise. The final red, green, and blue values represent the average pulse duration for each respective colour channel.

3) Thresholds for Stability:
To determine the most dominant colour, the code compares the red, green, and blue values. Since a lower pulse duration means higher intensity, the colour with the lowest value is considered dominant. Thresholds are used to ensure a meaningful difference between values. Red is only returned if it is significantly lower than Green (by more than redThreshold), and green is returned only if it's significantly lower than red (by more than greenThreshold).

4) Handling Sensor Variability and Ignoring Blue:
Blue was excluded from final decisions due to inconsistent readings observed during testing. As a result, the system only compares red and green values for classification. Thresholds help stabilize detection despite minor fluctuations in pulse readings caused by lighting.

5) Final Classification:
The function identifyColor() returns either RED, GREEN, or -1 (for unknown) based on which colour has the strongest intensity. If neither red nor green meets the threshold conditions, the result is considered inconclusive.

5.3.2 Ultrasonic code

To measure the distance between Alex and surrounding objects, the read_US() function sends a 10 µs trigger pulse to each ultrasonic sensor (front, left, right) by briefly setting the respective pin high (see Appendix Figure 13). The Arduino then uses pulseIn() to measure how long the echo pin stays high, which represents the time taken for the ultrasonic wave to travel to the object and back. To obtain the one-way travel time, the result is divided by 2. The speed of sound is approximated as 0.0345 cm/µs, so the final distance in centimeters is calculated using the formula: Distance = (duration in µs × 0.0345) / 2

Section 6 Software Design

6.1 High Level Pi Algorithm

6.1.1 Teleoperation

6.1.1.1 Network and TLS

To operate our Alex, our two laptops and RPi were all connected to the hotspot of Alson's iPhone. Connecting them to the same network allows them to transfer information across to one another. To increase network security, TLS was used by generating certs, public and private keys for the server and client using OpenSSL, as per the TLS studio.
Firstly, to set up the TLS server:

1. Both laptop and RPi are connected to the same network
2. TLS server is started on the RPi [./tls-alex-server]
3. TLS client is started on the laptop [./tls-alex-client 172.20.10.2 5001]
4. RPi sends the laptop its certificate (alex.crt)
5. Laptop uses the CA authority's public key to check alex.crt's validity
6. Laptop sends its certificate (laptop.crt) to RPi
7. RPi uses the CA authority's public key to check laptop.crt's validity
8. The handshake is complete and the laptop and RPi will now talk to each other

6.1.1.2 Raspberry Pi server and Laptop client

We used the RPi to host a server that could receive commands sent over the network. The code for the server was created using the one provided in the studios. When the MCP was started, the program would start a listener thread and once a client is connected to the server, a new worker thread would be created. The worker thread managed the communication between the client, the server and the Arduino by deserializing the packets sent over network and serialising the commands and data into packets to be sent to the Arduino. On the laptop, we create a client program which would connect to the RPi server and act as an interface between the operator and the main control program. The client would continuously poll for user input and subsequently serialise the data into a network packet that is sent to the server.

When a command is entered on the laptop side, it is stored in a 10 byte buffer array, and this buffer is what is sent over to the RPi through the TLS server.

```
char buffer[10];
int32_t params[2];
```

buffer[0] holds the value corresponding to the nature of the packet; since we are sending a command, it will be NET_COMMAND_PACKET.

Then, the RPi separates the received data (contained within the buffer) into its components, and stores them into a TPacket struct (commandPacket), as seen below. This TPacket struct will then be serialized [int len = serialize(buffer, packet, sizeof(TPacket))] and sent over to the Arduino via serial communications [serialWrite(buffer, len)]. Finally, the Arduino will deserialize the received packet and execute the command. (See Appendix Table 1 for the full list of commands)

```
char cmd = buffer[1];
uint32_t cmdParam[2];
memcpy(cmdParam, &buffer[2], sizeof(cmdParam));
TPacket commandPacket;
commandPacket.packetType = PACKET_TYPE_COMMAND;
commandPacket.params[0] = cmdParam[0];
commandPacket.params[1] = cmdParam[1];
```

6.1.1.3 Servo Motors

To control the servo, we will enter the keys ‘n’, ‘m’, ‘k’, 'j’ to the laptop, which will send the command packets respectively from the laptop to the RPi to the Arduino. Then, the Arduino will trigger the front 2 servo motors to open and close based on the COMMAND_OPEN (m) and COMMAND_CLOSE (n) commands respectively, while it will trigger the back servo motor to open and close based on the COMMAND_SLIDE (J) and COMMAND_PLANE (K) commands respectively.

6.1.1.4 RPLIDAR

A broad overview of how the RPLIDAR is being used in Alex (See Appendix Figure 15):

1. Operator looks at the map of the surroundings generated by the RPLIDAR unit
    a. roslaunch rplidar_ros view_slam.launch
    b. The map is made by RVIZ using Hector SLAM on a ROS node on the RPi, which is then shown on the laptop screen via the RealVNC Viewer app

2. Input is sent as a command packet to the RPi through the TLS server.
3. RPi relays the command to the Arduino via serial communication.
4. Finally, the Arduino executes this command.

6.1.1.5 Camera

A broad overview of how the Pi Camera is used to support remote operations of ALEX:

1. Operator runs AlexCameraStreamServer.py on the Pi and initiates the video feed by opening a web browser and navigating to: http://172.20.10.2:8000
2. Streaming is initiated by pressing the "Start Stream" button on the web page, with each request limited to 10 seconds. The server begins sending a highly processed, black-and-white outline video stream.
3. Based on the video and map data from RPLIDAR, the operator sends a command packet via the secure TLS server.
4. RPi relays the command to the Arduino via serial communication.
5. Finally, the Arduino executes this command.

6.1.2 Colour Detection

To detect colour, we will enter the key ‘z’ to the laptop, which will send the COMMAND_GET_COLOR command packet from the laptop to the RPi to the Arduino. Then, the Arduino will trigger the colour sensor and figure out what colour is present in front of Alex. The Arduino responds by calling the measure_color() function to read the red, green, and blue frequencies, which are then passed to the identifyColor() function to determine the detected color using simple thresholds. It stores the result as an integer value in params[0] of the status packet: 82 for Red ('R'), 71 for Green ('G'), and -1 for Error ('X'). (See Appendix Figure 14)This status packet is sent back to the Raspberry Pi, which forwards it to the laptop. The result will then be displayed on the laptop via the handleStatus() function. On the laptop, the integer value in params[0] is also typecast to a character ('R', 'G', or 'X') for the user to see the detected color or an error message, so as to allow a clearer and easier interpretation for the user.

```
printf("Detected Colour:\t\t%c\n", static_cast<char>(data[0]));
```

Section 7 Lessons Learnt - Conclusion

7.1 Lessons Learned

One lesson that we learnt was not testing our robot early enough within the actual maze environment. Much of our attention was focused on getting individual components to work in isolation. It was only when everything was assembled and run together that we realized a major design flaw. Using only a front claw to carry the medpak obstructed the colour sensor mounted just behind it, which meant we couldn’t detect the astronaut’s colour accurately. Moreover, if we found the red astronaut before the green one, we had no way to pick it up because the claw was already holding the med-pack. Fortunately, we managed to implement a rear claw before the final run. This allowed the med-pack to be held at the back, leaving the front claw free and the color sensor unobstructed. It can now detect the astronaut’s colour,

deploy the medpak if it was green, or pick up the red astronaut and continue searching, eliminating the need to traverse the maze twice. But this taught us that it is important to get the robot running in the maze early so that we are able to identify issues that we would have never thought of initially and would have enough time to solve them before any graded runs.

The second lesson we learned was the importance of keeping our code clean, consistent and well-organized, especially when working across multiple files and systems for the first time. Initially, interfacing with numerous Digital I/O pins led to a cluttered and hard-to-follow codebase. To address this, we identified the Data Direction Registers associated with each I/O pin and defined their bit numbers clearly at the beginning of our code. We then used bit shifting to manage their values throughout the program. This approach made our code significantly easier to debug, as everything was logically structured and easy to trace. Having a clean code is crucial for efficient troubleshooting and makes it significantly easier for other group members to read and understand, even if they weren't the ones who originally wrote it.

7.2 Mistakes Made

One mistake we made was not practising navigation with Alex enough before the trial run. Before the trial run, we spent too much time troubleshooting issues with the RPi and Arduino, focusing on fixing problems and improving Alex. This caused us to neglect the importance in practising using it and learning how to navigate using the SLAM produced map. Due to the lack of time and practice, our movement and decision making during the trial run were considered slow as we ran out of time, which resulted in us only managing to find the green and red astronaut. Learning from this mistake, we invested more time into conducting practice runs before our final graded run. We created a checklist and conducted multiple tests, such as measuring ultrasonic sensor readings. For example, we considered values of 14 or less as too close to obstacles at the sides or at the front. Hence, with more test runs, we became more efficient, as our judgement largely improved.

The second greatest mistake we made was not finalizing Alex's design early in the project. As a result, we repeatedly had to modify the build by adding components like servo motors, which meant constantly disassembling and reassembling the top chassis of Alex. This quickly became a major hassle and led to frequent hardware problems. A recurring issue was that every time we removed the top chassis and screwed it back on, some wires connected to the Arduino would get dislodged. To fix this, we have to take the top off again, reconnect the wires and screw everything back together which was something we repeated multiple times. Moreover, our initial design is to make Alex more compact, which requires us to push the top chassis against the many wires on the Arduino. The main issue is that the connections were highly affected, causing some components to stop functioning frequently. For example, when the back servo is able to function, the front 2 servos stopped working and the motors were not working properly. This forced us to troubleshoot extensively. To solve this, we elevated the top chassis with connectors, creating space and preventing wire compression to ensure their connections remain intact. Hence, we learnt to put the functionalities of Alex as top priority, even if it requires us to compromise its compactness.

References

Habibian, S., Dadvar, M., Peykari, B. *et al.* (2021). Design and implementation of a maxi-sized mobile robot (Karo) for rescue missions. *Robomech J* 8, 1. https://robomechjournal.springeropen.com/articles/10.1186/s40648-020-00188-9#Sec3

SPOT TO THE RESCUE : Boston Dynamics. (n.d.). Spot to the Rescue. *Boston Dynamics Blog*. https://bostondynamics.com/blog/spot-to-the-rescue/

Appendix

Figure 8: KARO Mobile Robot

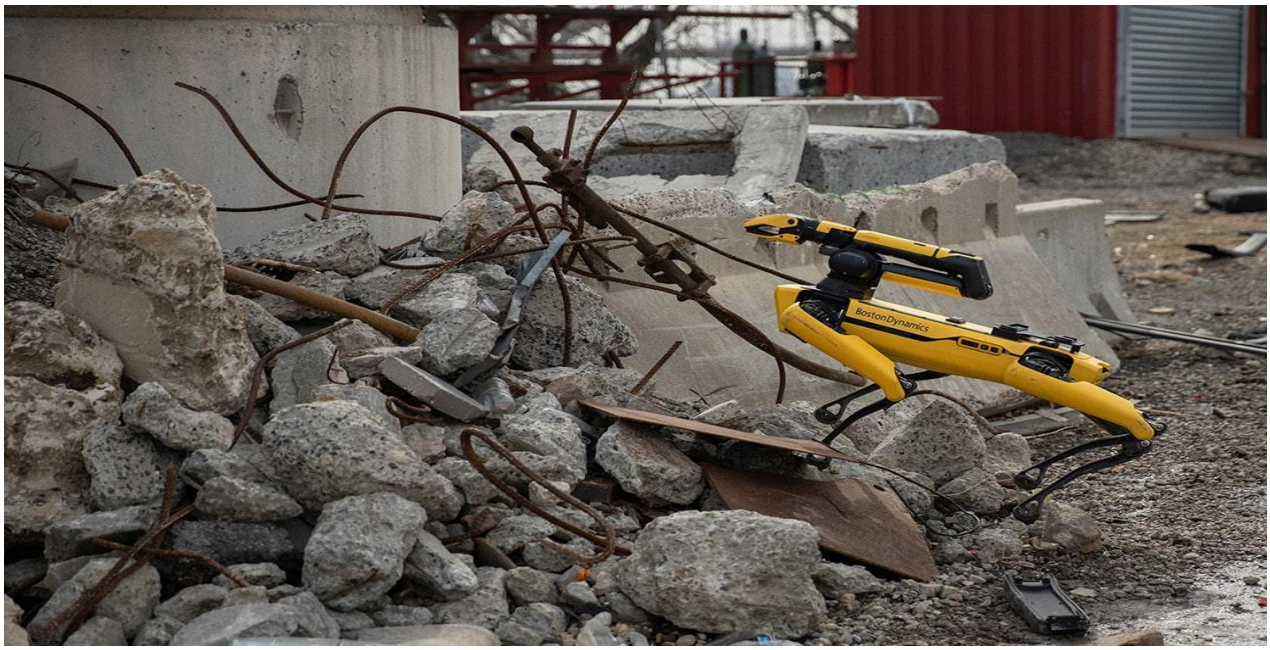

Figure 9: Spot

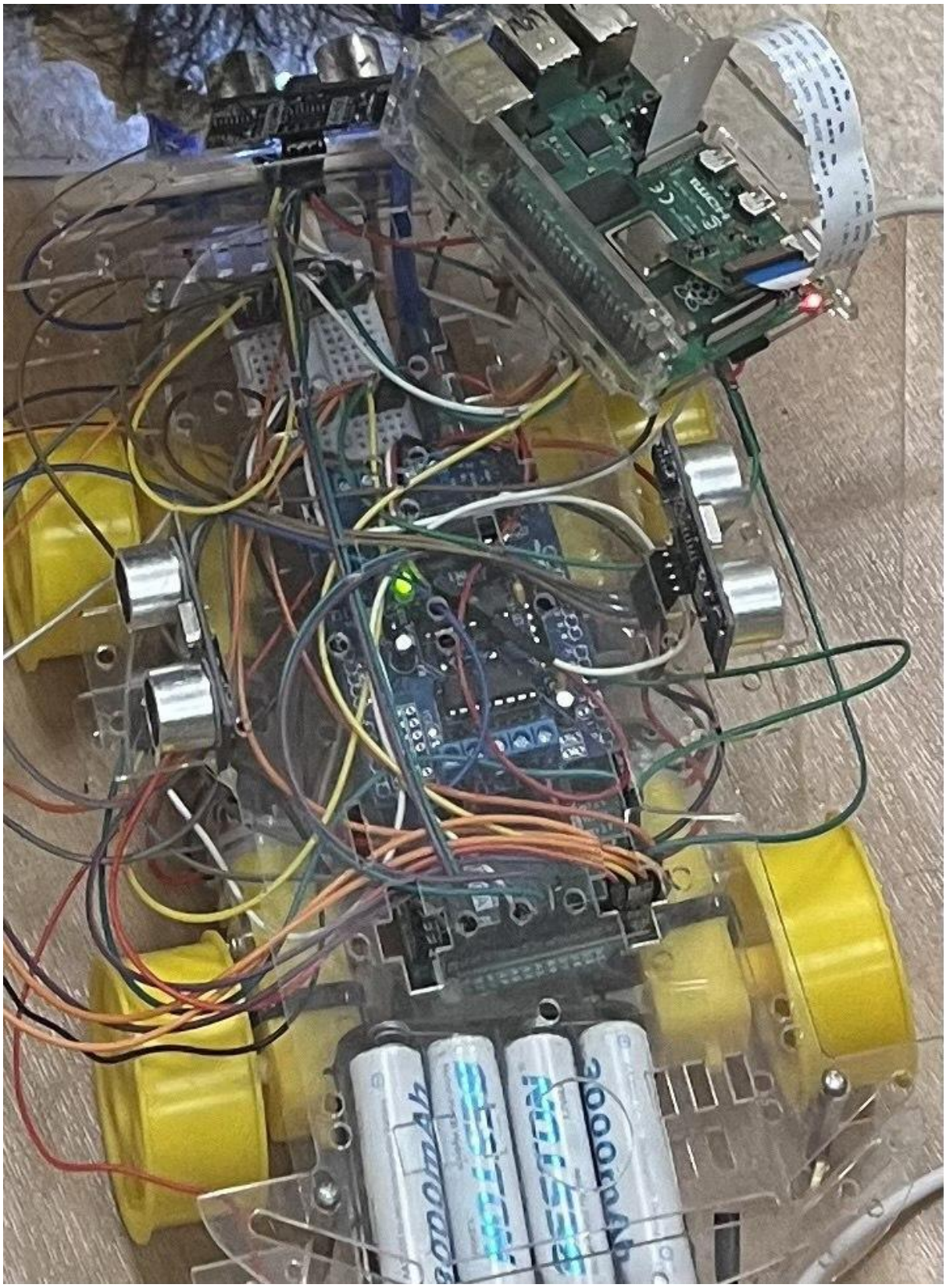

Figure 10: Top View of Alex to expose the hardware and wires

```
typedef enum {
  COMMAND_FORWARD = 0,
  COMMAND_REVERSE = 1,
  COMMAND_TURN_LEFT = 2,
  COMMAND_TURN_RIGHT = 3,
  COMMAND_STOP = 4,
  COMMAND_GET_STATS = 5,
  COMMAND_CLEAR_STATS = 6,
  COMMAND_GET_COLOR = 7,
  COMMAND_GET_DIST = 8,
  COMMAND_OPEN = 9,
  COMMAND_CLOSE = 10,
  COMMAND_SLIDE = 11,
  COMMAND_PLANE = 12
} TCommandType;
```

Figure 11: All 13 commands recognised by Alex

```
void measure_color(int &red, int &green, int &blue) {
  red = readFrequency(0, 0);
  green = readFrequency(1, 1);
  blue = readFrequency(0, 1);
}

int readFrequency(int s2Val, int s3Val) {
  digitalWrite(S2, s2Val);
  digitalWrite(S3, s3Val);
  delay(colorDelay);
  int total = 0;
  for (int i = 0; i < colorSampleCount; i++) {
    total += pulseIn(sensorOut, LOW, 50000);
    delay(colorDelay);
  }
  return total / colorSampleCount;
}

uint32_t identifyColor(int red, int green, int blue) {
  // Simple ratio-based detection
  int redThreshold = 30;
  int greenThreshold = 30;

  if (red < green - redThreshold) {
    return RED;
  } else if (green < red - greenThreshold) {
    return GREEN;
  }
  return -1; // UNKNOWN
}

void match_colour() {
  int r, g, b;
  measure_color(r, g, b);
  color = identifyColor(r, g, b);
}
```

Figure 12: Colour Detection Algorithm

```
unsigned long f_dist() {
  PORTC |= (1 << trigF);
  delayMicroseconds(10);
  PORTC &= ~(1 << trigF);
  int microsecs = pulseIn(echoFPin, HIGH);
  frontD = (microsecs * SPEED_OF_SOUND)/2;
}

void read_US() {
  PORTC |= (1 << trigF);
  delayMicroseconds(10);
  PORTC &= ~(1 << trigF);
  int microsecs = pulseIn(echoFPin, HIGH);
  frontD = (microsecs * SPEED_OF_SOUND)/2;

  PORTB |= (1 << trigL);
  delayMicroseconds(10);
  PORTB &= ~(1 << trigL);
  microsecs = pulseIn(echoLPin, HIGH);
  leftD = (microsecs * SPEED_OF_SOUND)/2;

  PORTA |= (1 << trigR);
  delayMicroseconds(10);
  PORTA &= ~(1 << trigR);
  microsecs = pulseIn(echoRPin, HIGH);
  rightD = (microsecs * SPEED_OF_SOUND)/2;
}
```

Figure 13: Ultrasonic code

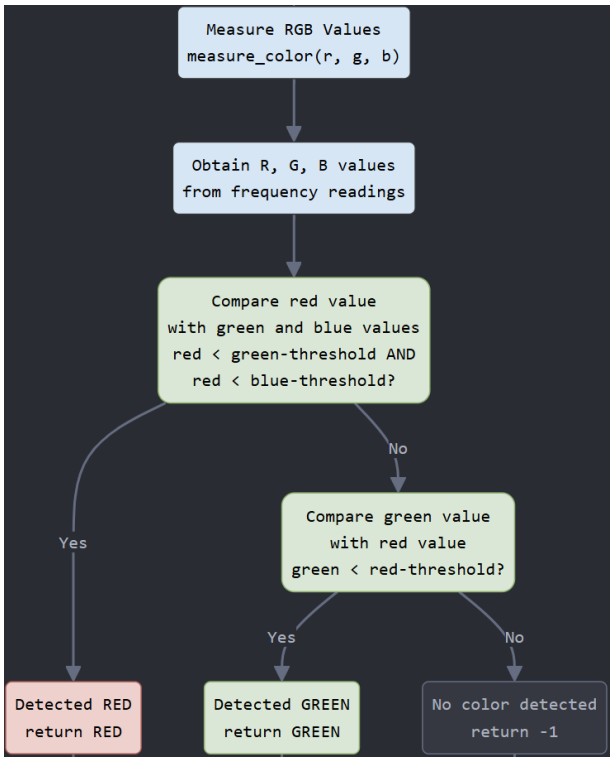

Figure 14: Flowchart for colour sensor algorithm

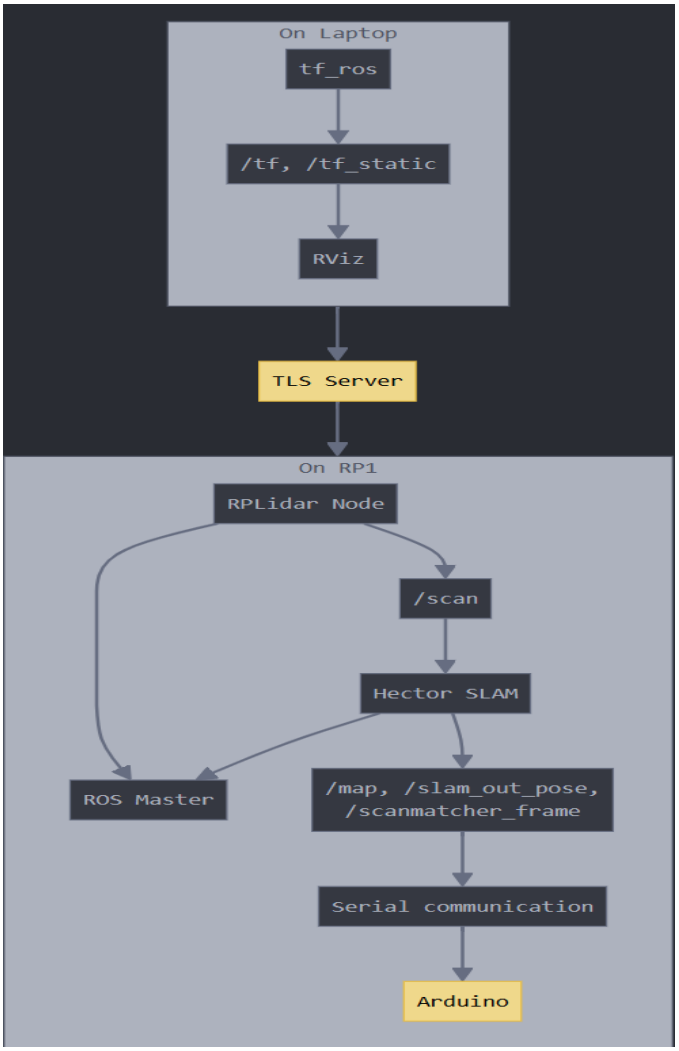

Figure 15: Flowchart for RPLIDAR

Command	Key	Description
FORWARD	F / f	Takes in the params[0] as distance in cm. Sends a move forward command to the RPi, which will then send it to the Arduino.
REVERSE	B / b	Takes in the params[0] as distance in cm. Sends a move backward command to the RPi which will then send it to the Arduino.
TURN LEFT	L / l	Take in the params[0] as angles in degrees. Sends a turn left command to the RPi which will then send it to the Arduino.
TURN RIGHT	R / r	Take in the params[0] as angles in degrees. Sends a turn right command to the RPi which will then send it to the Arduino.
GET COLOR	Z / z	Sends to the RPi a colour sensor data request, which will then send it to theArduino. Thereafter, it will print the detected colour on the Laptop.
GET DIST	D / d	Sends to the RPi ultrasonic data request, which will then send it to the Arduino. Thereafter, it will print the 3 measured distances from the 3 ultrasonic sensors on the Laptop.
CLEAR STATS	C / c	Resets all the stats back to 0
STOP	S / s	Sends a stop command to the RPi which will then send it to the Arduino. This is to tell Alex to stop moving.
OPEN	N /n	Sends to the RPi a servo request, which will then send it to the Arduino. This is to set the front 2 servo motors to its open position.
CLOSE	M /m	Sends to the RPi a servo request, which will then send it to the Arduino. This is to set the front 2 servo motors to its closed position.
PLANE	K / k	Sends to the RPi a servo request, which will then send it to the Arduino. This is to set the back servo motor to its close position.
SLIDE	J / j	Sends to the RPi a servo request, which will then send it to the Arduino. This is to set the back servo motor to its open position.

Table 1: List of commands

```
void setupSerial()
{
  // To replace later with bare-metal.
  //Serial.begin(9600);
  UBRR0L = 103;
  UBRR0H = 0;
  UCSR0C = 0b00000110;
  UCSR0A = 0;
  // Change Serial to Serial2/Serial3/Serial4 in later labs when using
the other UARTs
}

// Start the serial connection. For now we are using
// Arduino wiring and this function is empty. We will
// replace this later with bare-metal code.

void startSerial()
{
  UCSR0B = 0b00011000;
  // Empty for now. To be replaced with bare-metal code
  // later on.

}

// Read the serial port. Returns the read character in
// ch if available. Also returns TRUE if ch is valid.
// This will be replaced later with bare-metal code.

int readSerial(char *buffer)
{

  int count=0;

  // Change Serial to Serial2/Serial3/Serial4 in later labs when using
other UARTs

  while((UCSR0A >> 7))
    buffer[count++] = UDR0;

  return count;
}

// Write to the serial port. Replaced later with
```

```
// bare-metal code

void writeSerial(const char *buffer, int len) {
  for (int i = 0; i < len; i++) {
    while ((UCSR0A & 0b00100000) == 0);
    UDR0 = buffer[i];// Change Serial to Serial2/Serial3/Serial4 in
later labs when using other UARTs
  }
}
```

Figure 16: Bare-metal UART Code

```
case COMMAND_OPEN:
        sendOK();
        servoLeft.write(180);     // Open position (adjust angle as
needed)
        servoRight.write(0);    // Open position (adjust angle as
needed)
      break;
    case COMMAND_CLOSE:
        sendOK();
        servoLeft.write(90);     // Open position (adjust angle as
needed)
        servoRight.write(90);    // Open position (adjust angle as
needed)
      break;

    case COMMAND_SLIDE:
        sendOK();
        servoBack.write(90);     // open position (adjust angle as
needed
      break;

     case COMMAND_PLANE:
        sendOK();
        servoBack.write(170);     // close position (adjust angle as
needed
      break;
```

Figure 17: Servo Code

```
void setup() {
  // put your setup code here, to run once:

  alexDiagonal = sqrt((ALEX_LENGTH * ALEX_LENGTH) + (ALEX_BREADTH *
ALEX_BREADTH));
  alexCirc = PI * alexDiagonal;
  servoLeft.attach(SERVO_LEFT_PIN);
  servoRight.attach(SERVO_RIGHT_PIN);
  servoBack.attach(SERVO_BACK_PIN);

// Set to default (closed)
  servoLeft.write(180);   // Closed position
  servoRight.write(0); // Closed position
  servoBack.write(170); // Closed position

  DDRA |= (1 << trigR);
  DDRA &= ~(1 << echoR);
  DDRB |= (1 << trigL);
  DDRB &= ~(1 << echoL);
  DDRC |= (1 << trigF); // | (1 << S3) | (1 << S1);
  DDRD &= ~(1 << echoF);
  //DDRG &= ~(1 << sensorOut);

  //setup color sensor
  //set S0,S1,S2,S3 as output
  DDRA |= (1 << PA0) | (1 << PA1) | (1 << PA2) | (1 << PA3);
  //set sensorOut as input
  DDRK &= ~(1 << PK0);
  DDRG &= ~(1 << PG1);
  PORTK &= ~(1 << PK0);
  PORTG &= ~(1 << PG1);
  //set S0 as HIGH and S1 as LOW
  PORTA |= (1 << PA0);
  PORTA &= ~(1 << PA1);

  cli();
  setupEINT();
  setupSerial();
  startSerial();
  enablePullups();
  initializeState();
  sei();
}
```

Figure 18: void setup() Code

```
void handleCommand(void *conn, const char *buffer)
{
	// The first byte contains the command
	char cmd = buffer[1];
	uint32_t cmdParam[2];

	// Copy over the parameters.
	memcpy(cmdParam, &buffer[2], sizeof(cmdParam));

	TPacket commandPacket;

	commandPacket.packetType = PACKET_TYPE_COMMAND;
	commandPacket.params[0] = cmdParam[0];
	commandPacket.params[1] = cmdParam[1];

	printf("COMMAND RECEIVED: %c %d %d\n", cmd, cmdParam[0], cmdParam[1]);

	switch(cmd)
	{
		case 'f':
		case 'F':
			commandPacket.command = COMMAND_FORWARD;
			uartSendPacket(&commandPacket);
			break;

		case 'b':
		case 'B':
			commandPacket.command = COMMAND_REVERSE;
			uartSendPacket(&commandPacket);
			break;

		case 'l':
		case 'L':
			commandPacket.command = COMMAND_TURN_LEFT;
			uartSendPacket(&commandPacket);
			break;

		case 'r':
		case 'R':
			commandPacket.command = COMMAND_TURN_RIGHT;
			uartSendPacket(&commandPacket);
			break;

		case 's':
		case 'S':
			commandPacket.command = COMMAND_STOP;
			uartSendPacket(&commandPacket);
			break;

		case 'c':
		case 'C':
			commandPacket.command = COMMAND_CLEAR_STATS;
```

```
                commandPacket.params[0] = 0;
                uartSendPacket(&commandPacket);
                break;

        case 'g':
        case 'G':
                commandPacket.command = COMMAND_GET_STATS;
                uartSendPacket(&commandPacket);
                break;

        case 'z':
        case 'Z':
                commandPacket.command = COMMAND_GET_COLOR;
                uartSendPacket(&commandPacket);
                break;

        case 'd':
        case 'D':
                commandPacket.command = COMMAND_GET_DIST;
                uartSendPacket(&commandPacket);
                break;

        case 'n':
        case 'N':
                commandPacket.command = COMMAND_OPEN;
                uartSendPacket(&commandPacket);
                break;

        case 'm':
        case 'M':
                commandPacket.command = COMMAND_CLOSE;
                uartSendPacket(&commandPacket);
                break;

        case 'j':
        case 'J':
              commandPacket.command = COMMAND_SLIDE;
              uartSendPacket(&commandPacket);
              break;

    case 'k':
        case 'K':
                commandPacket.command = COMMAND_PLANE;
                uartSendPacket(&commandPacket);
                break;

        default:
                printf("Bad command\n");

    }
}
```

Figure 19: void handleCommand() function in Server Code

```
void handleStatus(const char *buffer)
{
	int32_t data[16];
	memcpy(data, &buffer[1], sizeof(data));

	printf("\n ------- ALEX STATUS REPORT ------- \n\n");

	printf("Detected Colour:\t\t%c\n", static_cast<char>(data[0]));
	printf("Color Value: \%d\n", data[0]);
	printf("Front US Distance:\t\t%d\n", data[1]);
	printf("Left US Distance:\t\t%d\n", data[2]);
	printf("Right US Distance:\t\t%d\n", data[3]);
	//printf("Left Forward Ticks:\t\t%d\n", data[4]); //Uncomment to determine on turning
radius / movement distance
	//printf("Right Forward Ticks:\t\t%d\n", data[5]);
	//printf("Left Reverse Ticks:\t\t%d\n", data[6]);
	//printf("Right Reverse Ticks:\t\t%d\n", data[7]);
	//printf("Left Forward Ticks Turns:\t%d\n", data[8]);
	//printf("Right Forward Ticks Turns:\t%d\n", data[9]);
	//printf("Left Reverse Ticks Turns:\t%d\n", data[10]);
	//printf("Right Reverse Ticks Turns:\t%d\n", data[11]);
	printf("Forward Distance:\t\t%d\n", data[12]);
	printf("Reverse Distance:\t\t%d\n", data[13]);
	printf("\n---------------------------------------\n\n");
}

void getParams(int32_t *params)
{
	printf("Enter distance/angle in cm/degrees (e.g. 50) and power in %% (e.g. 75)
separated by space.\n");
	printf("E.g. 50 75 means go at 50 cm at 75%% power for forward/backward, or 50
degrees left or right turn at 75%%  power\n");

	scanf("%d", &params[0]);
	params[1] = 100;
	//scanf("%d %d", &params[0], &params[1]);
	flushInput();
}

void *writerThread(void *conn)
{
	int quit=0;

	while(!quit)
	{
		char ch;
		printf("Command (f=forward, b=reverse, l=turn left, r=turn right, s=stop,
c=clear stats, g=get stats q=exit)\n");
		scanf("%c", &ch);

		// Purge extraneous characters from input stream
		flushInput();
```

```
        char buffer[10];
        int32_t params[2];

        buffer[0] = NET_COMMAND_PACKET;
        switch(ch)
        {
                case 'f':
                case 'F':
                case 'b':
                case 'B':
                case 'l':
                case 'L':
                case 'r':
                case 'R':
                                        getParams(params);
                                        buffer[1] = ch;
                                        memcpy(&buffer[2], params, sizeof(params));
                                        sendData(conn, buffer, sizeof(buffer));
                                        break;
                case 'M':
                case 'm':
                case 'j':
                case 'J':
                case 'k':
                case 'K':
                case 'N':
                case 'n':
                case 's':
                case 'S':
                case 'c':
                case 'C':
                case 'g':
                case 'G':
                case 'z':
                case 'Z':
                case 'd':
                case 'D':
                                params[0]=0;
                                params[1]=0;
                                memcpy(&buffer[2], params, sizeof(params));
                                buffer[1] = ch;
                                sendData(conn, buffer, sizeof(buffer));
                                break;
                case 'q':
                case 'Q':
                        quit=1;
                        break;
                default:
                        printf("BAD COMMAND\n");
        }
}
```

```
        printf("Exiting keyboard thread\n");

    /* TODO: Stop the client loop and call EXIT_THREAD */

        stopClient();
    EXIT_THREAD(conn);

    /* END TODO */

    return NULL;
}
```

Figure 20: void handleCommand() function in Client Code