CG2111A Engineering Principles and Practice II

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B04-5A

Name	Student #	Main Role
Brandon Kang	A0272866N	Software Lead
Roderick Kong Zhang	A0286550Y	Software Engineer
Sean Ter Yong Jun	A0272977J	Hardware Lead
Pavithra Srinivasan	A0282405J	Hardware Engineer
Ser Jun Wei Kingsley	A0271886M	Hardware Engineer

Table of Contents

Section 1 | Introduction

Alex is a robotic vehicle that is tasked to perform search-and-rescue missions. In this prototype phase, Alex will be placed in a simulated environment where it is easier to navigate and detect victims. It is tasked to navigate a room filled with various man-made obstacles such as boxes, map out its surrounding environment, and identify victims. In this test, victims are either "green" – healthy but trapped, or "red" – injured and trapped. Moreover, Alex must be teleoperated so that operators are not in contact with the disaster site. The goal for the teleoperator is to locate two victims and correctly identify their statuses, map out the disaster site, and park Alex in a designated spot – all within a 6-minute window, with minimal collisions.

To this end, Alex is designed with the following specifications and functionalities. A Raspberry Pi is installed on Alex to allow for communication between the operator's laptop and the Arduino controlling its components. The LiDAR sensor, in conjunction with Hector SLAM, allows Alex to detect obstacles and objects while simultaneously identifying its position on the map. To differentiate victims, healthy or injured, from obstacles, a colour sensor is installed on Alex. If the detected colour is not red or green, Alex will identify the object as an obstacle instead of a victim. Additionally, ultrasonic sensors are placed at the front, left and right sides on Alex to detect and avoid nearby obstacles. Alex also features a loud buzzer, used to alert the teleoperator and nearby pedestrians when a victim has been detected.

Section 2 | Review of State of the Art

2.1 | Boston Dynamics – Atlas

The Atlas is a humanoid robot designed to assist emergency management teams in handling natural and man-made catastrophes. It can perform tasks such as flipping switches, shutting off valves, opening doors, and running power equipment. These functions are possible through the use of components such as lightweight hydraulics and 3D-printed appendages. It also uses LiDAR and stereo vision to effectively navigate through rough terrains.

Strengths:

- 3D printing is used to manufacture components to save weight and space, resulting in a compact robot with a high strength-to-weight ratio
- Advanced control algorithms enable the robot to plan and perform actions based on the environment it analyses

Weaknesses:

- Too large to fit in tight spaces, inefficient in spotting and saving casualties in such spaces
- Ineffective in other terrains, such as water and air, hence only useful for land operations
- Uses a lot of energy, which may not be sustainable for long operations

2.2 | Pilant Energy – Velox

The Velox is a robot equipped with fins that give it amphibious capabilities, allowing for rescue missions on both land and water. It is able to reverse and make quick turns instantaneously, allowing it to rapidly manoeuvre and look around and between objects.

Strengths:

- Able to traverse through a variety of terrains, including land and water
- Lower environmental impact compared to other robots with spinning propeller-thrusters
- Resistant to entanglement in plants and other aquatic debris

Weaknesses:

- Slow travelling speed, hence inefficient in finding casualties
- Ineffective in overcoming large solid obstacles

Section 3 | System Architecture

Alex comprises multiple devices working and communicating together to complete its search-and-rescue mission. *Figure 1* illustrates Alex's system architecture, detailing how its various components are connected, and how they communicate with one another.

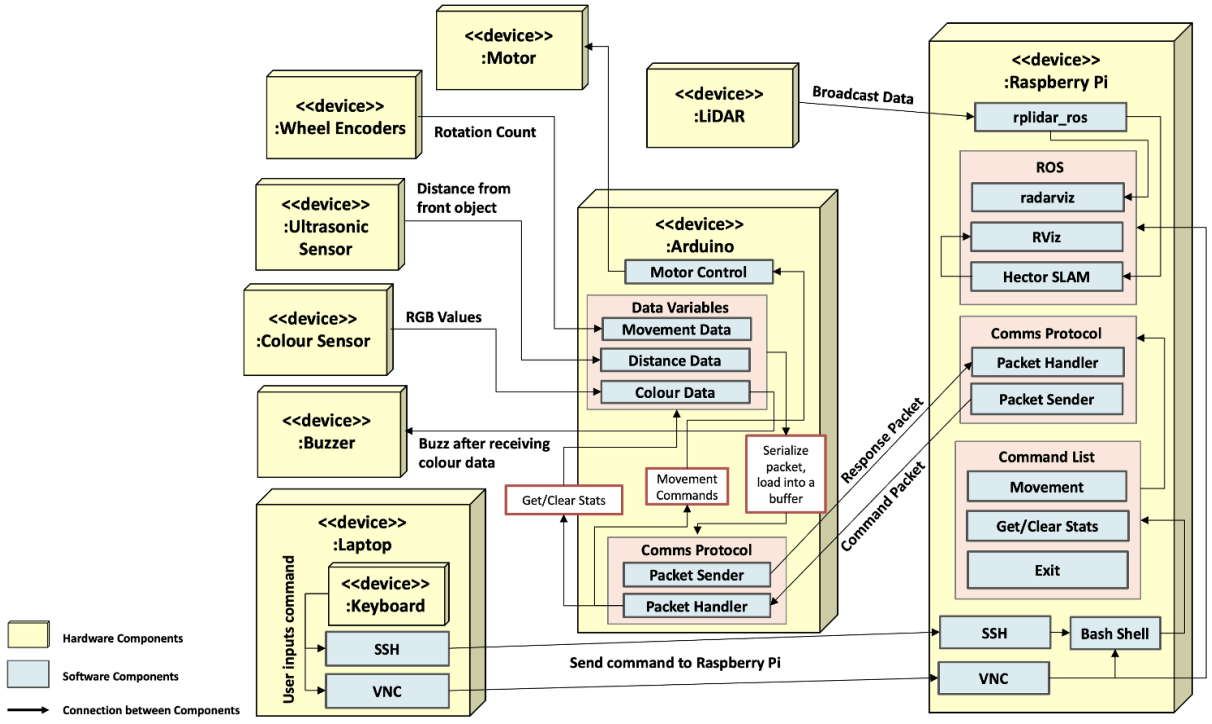

Figure 1. System Architecture Design

Section 4 | Hardware Design

4.1 | Photos of Hardware Components

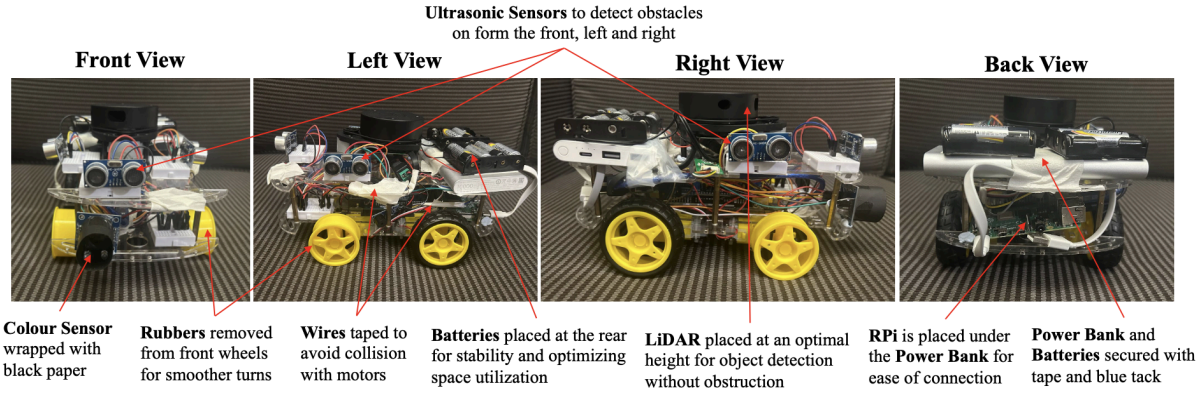

Figure 2. Photos of Alex

4.2 | Non-Standard hardware components

4.2.1 | Active Buzzer Module

We chose to add an active buzzer as one of Alex's additional functionalities. In a real-world context, when Alex detects victims, it is beneficial for nearby pedestrians around the area to be alerted by the buzzer whenever a victim has been detected, for them to seek help to rescue victims. In our robot implementation, we programmed the buzzer to sound wherever the colour sensor and ultrasonic sensors have been activated, alerting us that the detection system has been triggered successfully. Additionally, the buzzer module was used as a debugger in our building process as well. By setting the input of the buzzer as "high" whenever the ultrasonic sensor detects a distance below a certain threshold, we are able to determine when the ultrasonic sensors are not working as intended.

4.2.2 | HC-SR04 Ultrasonic Sensor

We added three ultrasonic sensors to the front, left and right sides of the robot. This helps us better avoid obstacles from all directions except the back. As the colour sensor is inaccurate at longer distances, having an ultrasonic sensor at the front allows us to better position Alex for it to detect the colour of the object accurately. This will be further elaborated on in **Section 6.1.2**. From our experiments, we determined that the optimal distance to detect the colour of objects is 5 cm. In a real-world context, Alex will be able to leverage on the 3 ultrasonic sensors to avoid collision with the surrounding rubble.

4.3 | Additional Hardware Features

4.3.1 | Black Paper Skirting

Applying black paper around the colour sensor gives more consistent readings by blocking ambient light, helping us to determine colours of victims more accurately. In a real-world context, it would be reprehensible to misidentify a live victim as dead and not send help.

4.3.2 | Rubber Covers

Removing the rubber from the front wheels reduces friction between the wheels and the ground. We noticed that this made our turns smoother and more consistent, resulting in improved traversal around the maze.

4.3.3 | Soldering of Battery Holders

In order to provide our motors with 9V, we had to use 6 AA batteries. As 1 battery holder is only able to hold 4 AA batteries, we had the 2 battery holders provided soldered together to fit the 6 AA batteries used to power our motors.

Section 5 | Firmware Design

5.1 | High Level Algorithm on the Arduino Mega

During serial communication with the RPi, the Arduino continuously polls for an incoming packet and executes the algorithm shown in *Figure 3*.

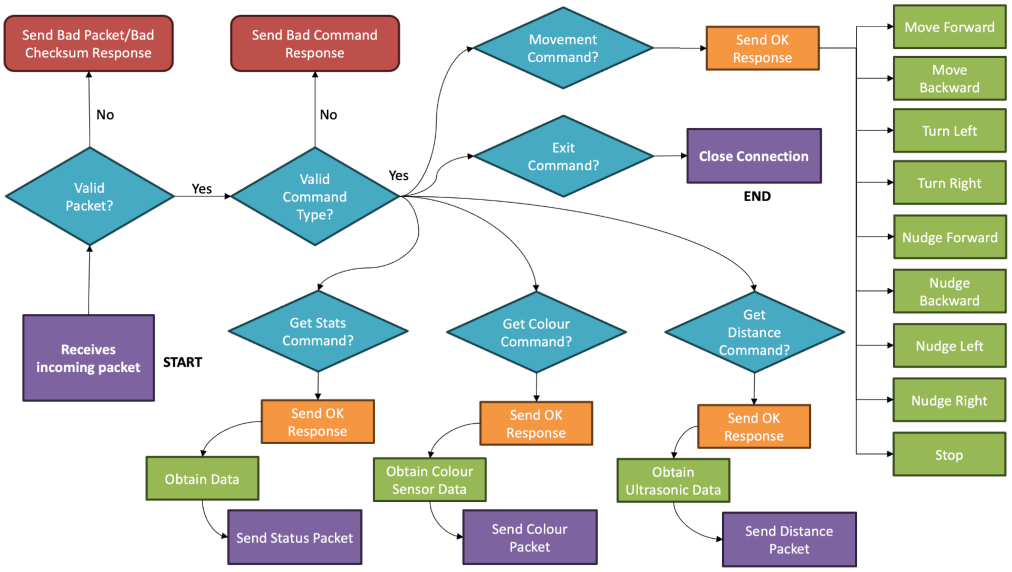

Figure 3. Flowchart of High Level Algorithm on the Arduino Mega

5.2 | Communication Protocol

The Arduino and RPi communicate via UART with a baud rate of 9600 bps and a frame format of 8N1. We begin serial communication by enabling TXEN0 and RXEN0 in the UCSR0B register. This setting is configured in bare metal as shown in *Figure 4* below.

```
void setupSerial()
{
  //Setting baud rate
  UBRR0L = 103;
  UBRR0H = 0;
  //Set Asynchronous USART, disable parity, set 1 stop bit, set 8-bit frame
  UCSR0C = 0b110;
  //Disable double-speed mode and multiprocessor mode
  UCSR0A = 0;
}

void startSerial()
{
  //Start by enabling RXEN0 and TXEN0
  UCSR0B = 0b11000;
}
```

Figure 4. Bare metal code for setting up serial communication

After the setup, the Arduino is ready to send and receive data in polling mode. To receive data, the Arduino polls RXC0 in UCSR0A until it has been set, then places the bytes in UDR0 into a buffer for it to be deserialized. To send data, a packet is serialized into a buffer, and the Arduino polls for UDR0 to be empty by checking that UDRE0 in UCSR0A has been cleared. The buffer is then written to UDR0 to be sent out to the RPi. *Figure 5* below shows the code implemented in bare metal.

```
int readSerial(char *buffer)
{
  int count=0;
  while ((UCSR0A & (1 << RXC0)) == 0); //Poll until data has been received
  buffer[count++] = UDR0; // Assign received byte to buffer array
  return 1;
}

void writeSerial(const char *buffer, int len)
{
  for (int i = 0; i < len; i++) {
    while ((UCSR0A & (1 << UDRE0)) == 0); //Poll until UDRE0 is set
    UDR0 = buffer[i];
  }
}
```

Figure 5. Bare metal code for sending and receiving in polling mode

Data is sent in the form of packets. Each TPacket is 100 bytes long and is constructed as shown in *Figure 6* below.

```
#define MAX_STR_LEN   32
// This packet has 1 + 1 + 2 + 32 + 16 * 4 = 100 bytes
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;
```

Figure 6. TPacket struct

The full details of the packet contents in the packetType and command field can be found in **Appendix A**.

5.3 | Additional Firmware Features

Bare metal programming was used when implementing our buzzer, ultrasonic sensor, and colour sensor. The full bare metal code can be found in **Appendix B**.

Section 6 | Software Design

6.1 | High Level Algorithm on the RPi

Figure 7 below shows an overview of the algorithm executed by the RPi during serial communication with the Arduino.

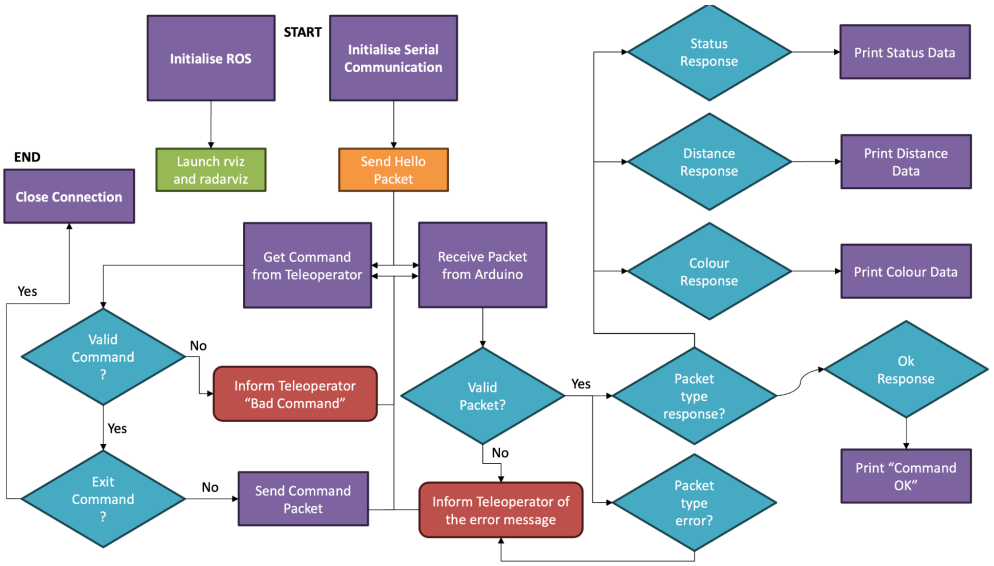

Figure 7. Flowchart of High Level Algorithm on the RPi

6.1.1 | Teleoperation

Alex's teleoperation controls are designed to be simple and intuitive. Command inputs by the teleoperator only consist of three main categories – movement commands, query commands and other commands.

Movement Commands

Movement commands are used to control the movements of Alex, and are implemented using the standard "WASD" keys, commonly used in video games. This allows for simple and intuitive controls that provide more ease to the teleoperator when controlling Alex. In addition to the standard movements for moving forward, backward, left and right, we have also implemented *nudging* for Alex to make small adjustments to avoid collision, and for better control in tight spaces. These commands use "IJKL" keys for intuitiveness, as they are in a similar layout as "WASD" on the keyboard. More details of this feature will be discussed in **Section 6.2.1**.

The terminal waits for a specific character to be pressed on the keyboard of the teleoperator, which then executes the command that is mapped to that specific character input. The teleoperator can enter a character without pressing the "ENTER" key to execute the command, providing ease of control. This feature is further discussed in **Section 6.2.2**. An overview of all movement commands are shown in the table below.

Command	Description
Standard Movement Commands	
W	Commands Alex to move forward
A	Commands Alex to move turn anticlockwise
S	Commands Alex to move backwards
D	Commands Alex to move turn clockwise

Spacebar	Commands Alex to stop movement
Nudging Commands	
I	Commands Alex to move forward for 0.4s (*Nudge* forward)
J	Commands Alex to move turn anticlockwise for 0.4s (*Nudge* left)
K	Commands Alex to move backwards for 0.4s (*Nudge* backward)
L	Commands Alex to move turn clockwise for 0.4s (*Nudge* right)

Figure 8. Overview of all Movement Commands

Query Commands

Query commands are used to obtain data collected by the wheel encoder, ultrasonic sensors and colour sensor. Data collected from these devices are stored in data variables, which are then used in response packets to send information back to the teleoperator. The table below shows an overview of all query commands.

Command	Description
G	Obtains odometry data collected from the wheel encoders which are stored in global variables on the Arduino Mega. These data consist of the number of ticks of the wheel encoder for each of the 4 movement directions, as well as the total distance travelled by Alex. A response packet from the Arduino containing these data will be sent to the RPi and printed out on the terminal for the teleoperator.
Z	Activate the colour sensor and obtain the colour data collected by it, storing the data in global variables in the Arduino Mega. These data consist of red, green and blue frequency values. Before sending a response packet, distance data from the front ultrasonic sensor is also collected which will be sent together with the colour data in the same response packet. The response packet will then be sent to the RPi and printed out on the terminal for the teleoperator. More details about colour detection can be found in **Section 6.1.2**.
U	Activate all 3 ultrasonic sensors and obtain distance data collected by all 3 of them, storing the data in global variables in the Arduino Mega. These data consist of the distances of an object from the front, left and right of Alex. A response packet from the Arduino containing these data will be sent to the RPi and printed out on the terminal for the teleoperator.

Figure 9. Overview of all Query Commands

Other Commands

An overview of the remaining commands are shown in the table below.

Command	Description
C	Set all global variables containing odometry data to 0.
Q	Terminate serial communication and close connection with the Arduino

Figure 10. Overview of all Other Commands

RViz

Figure 11 below shows a side-by-side comparison of the RViz output and the actual surroundings of Alex. The red arrow from the */slam_out_pose* topic is configured such that the arrow head approximately represents the size of Alex's body, which can be seen in the configuration in *Figure 11* on the left of the map. We also used the measure tool in RViz to plot a line to estimate the distance between two points in Alex's surroundings. The information circled in red in *Figure 11* demonstrates the use of the measure tool, where in this case, the distance between the left and right walls is about 2.04m.

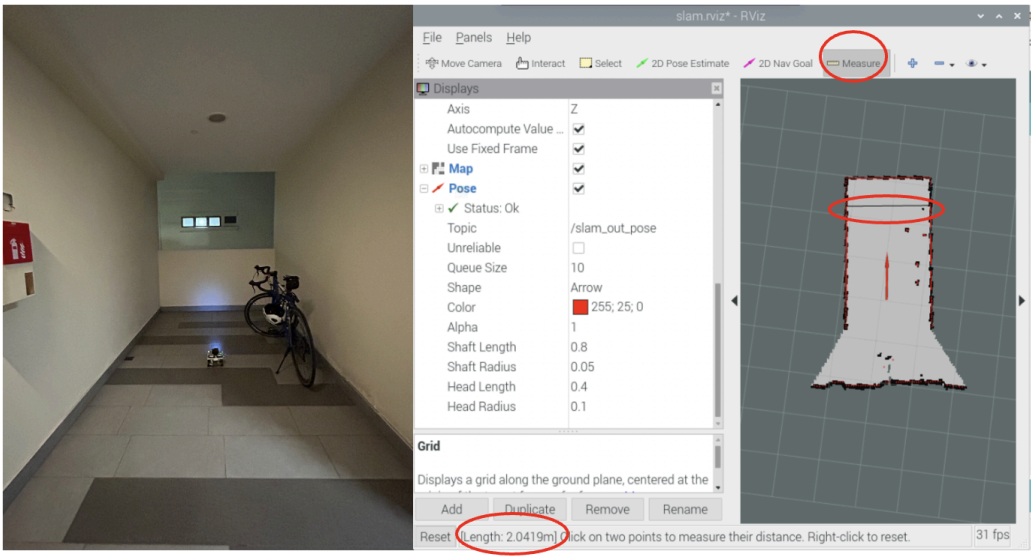

Figure 11. Comparison of Alex's surroundings (left) with RViz output (right)

6.1.2 | Colour Detection

Colours of objects scanned by the colour sensor can be distinguished using the frequency values obtained by the colour sensor. Since the victims were either red, green or white, we would need to differentiate between these 3 colours. This can be achieved by determining which of the frequency readings for red, green and blue is the lowest. For example, if the frequency reading for red is the lowest compared to the rest, then the object detected is red. For white, all frequency readings will be low and relatively close to one another. The full code for colour detection can be found in **Appendix B**.

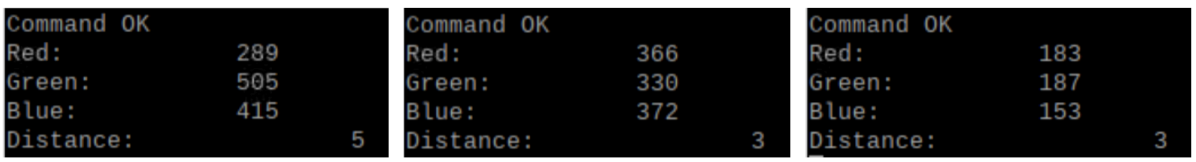

Figure 12. Sample data for red, green and white objects respectively (left to right)

However, we noticed that as the object gets further away from the colour sensor, the frequency readings become harder to differentiate from one another, since more ambient light is picked up by the sensor. We fixed a threshold of 5cm where the readings will be the most accurate. This is illustrated in *Figure 13* below.

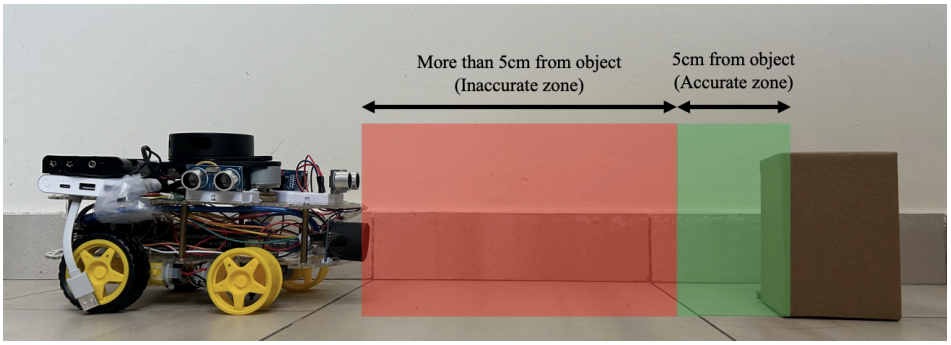

Figure 13. Accuracy zone for colour sensor readings

As such, we implemented our colour-sensing code such that the front ultrasonic sensor located above the colour sensor will also be activated when a query command to obtain colour data is requested by the teleoperator. This distance reading is also included in the response packet that is sent back to the RPi. This informs the teleoperator of the accuracy of the readings based on the distance the reading was taken from.

```
void sendColour() {
  unsigned long distance = getDistance(echoPinF); //Obtain the distance of the object from the front of Alex

  TPacket colourPacket;
  colourPacket.packetType = PACKET_TYPE_RESPONSE;
  colourPacket.command = RESP_COLOUR;
  colourPacket.params[0] = redFreq;
  colourPacket.params[1] = greenFreq;
  colourPacket.params[2] = blueFreq;
  colourPacket.params[3] = distance;
  sendResponse(&colourPacket);
}
```

Figure 14. Code for sending colour response packet to the RPi

The following flowchart shows the procedure to be followed by the teleoperator when detecting the colour of a nearby object.

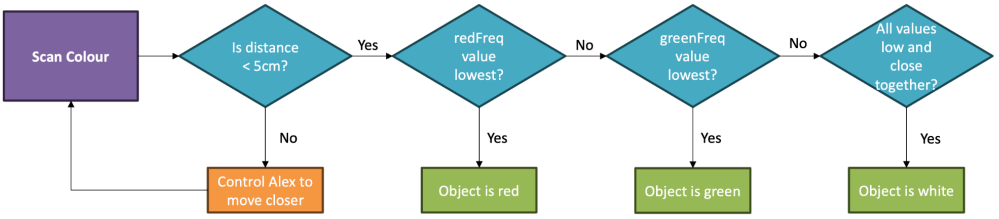

Figure 15. Flowchart of the procedures taken by the teleoperator

6.2 | Additional Software Features

6.2.1 | Nudging

This feature allows the teleoperator to make small adjustments to allow Alex to navigate through tight spaces, thus providing better control in Alex's movements. This feature was also implemented to avoid *bad magic number errors*, which is further explained in **Section 7.1.1**. We defined NUDGE_DELAY to be 400, which we used to move Alex in a certain

direction for 0.4 seconds, giving the nudging effect. We found that 0.4 seconds is the optimal duration for Alex to make the small movement adjustments we needed, without it moving too little which requires multiple nudges and additional time, or moving too much which causes Alex to collide with nearby obstacles before stopping. Refer to **Appendix C** for the full code.

6.2.2 | getch()

The function getch() from the <*conio.h*> c++ header file used in the *alex-pi* script enables the use of quick intuitive teleoperation commands to control Alex. getch() reads single characters keyed into the standard output, allowing single-character commands to be keyed and registered by the program without pressing the "enter" key. With this function, swift and precise manoeuvres could be executed by entering "WASD" commands in quick succession. getch() was chosen over getchar() as the former does not use a buffer and immediately returns the inputted character without waiting for the "enter" key (Karunakar, 2023). Moreover, we had to use a manual implementation of getch() (see **Appendix D**) as the <conio.h> header file, used primarily by MS-DOS compilers (Karunakar, 2023), was incompatible with Linux on the Raspberry Pi.

6.2.3 | radarviz

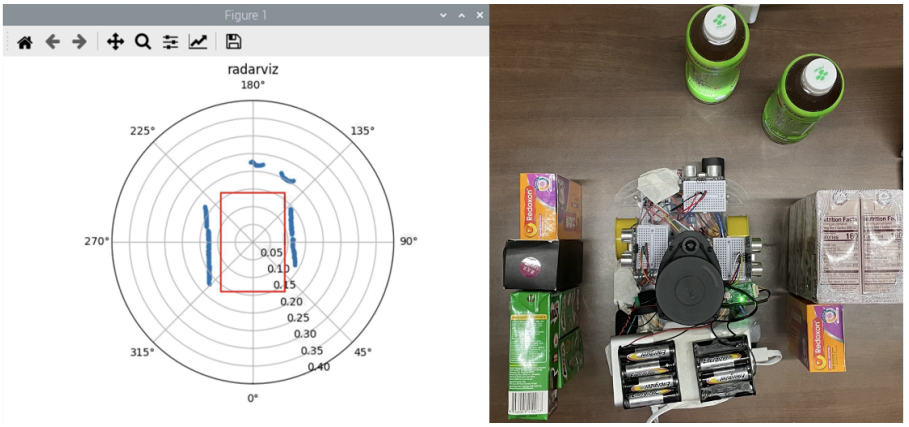

Figure 16. Comparison of radarviz output (left) with Alex's surroundings (right)

We implemented a software, named *radarviz*, that maps Alex's immediate surroundings on a radar map. This was born out of the desire for a more precise mapping solution to provide the teleoperator with a clearer view (than the *RViz*) when parking. However, it also proved to be extremely useful in distinguishing victims from nearby obstacles. It is also dynamic such that it can be configured to Alex's dimensions easily when changes are made, and to the preferred zoom level of the operator. The parameters used specific to our Alex are: ALEX_LENGTH = 0.28, ALEX_BREADTH = 0.18, and MAX_LIDAR_DISTANCE = 0.4.

radarviz was created based on Python code from the CG2111A ROS Networking Tutorial (Chen & Anderson, 2024), modified to run locally on the Raspberry Pi by shortening the subprocess ROS command (see *Figure 17*).

```
cmd = "source ~/cg2111a/devel/setup.bash && " \
      "rosrun rplidar_ros rplidarNodeClient"
```

Figure 17. Shortened command to retrieve LiDAR data from ROS locally

The angle values retrieved from *rplidar_ros* range from -180° to 180°. Hence, in order for *radarviz* to accurately represent obstacles surrounding Alex on the map, the regular expression has to be edited to account for negative numbers (see *Figure 18*), and the angles have to be normalised to 0° to 360° to be represented correctly on the map (see *Figure 19)*.

```
def parse_data_line(line):
    """Retrieve data from ROS Terminal"""
    match = re.search(r"\[ INFO\] \[\d+\.\d+\]: : \[(-?\d+\.\d+), (\d+\.\d+|inf)\]", line)
```

Figure 18. Updated regular expression to match negative numbers

```
angle_rad = np.radians((angle + 180) % 360)
```

Figure 19. Updated angle formula

To plot Alex's body on the polar radar map, Alex's dimensions have to be converted into polar coordinates (see *Figure 20*). These coordinates are then inputted into the *matplotlib* plot() function for the rectangle representing Alex's body to be drawn (see *Figure 21*).

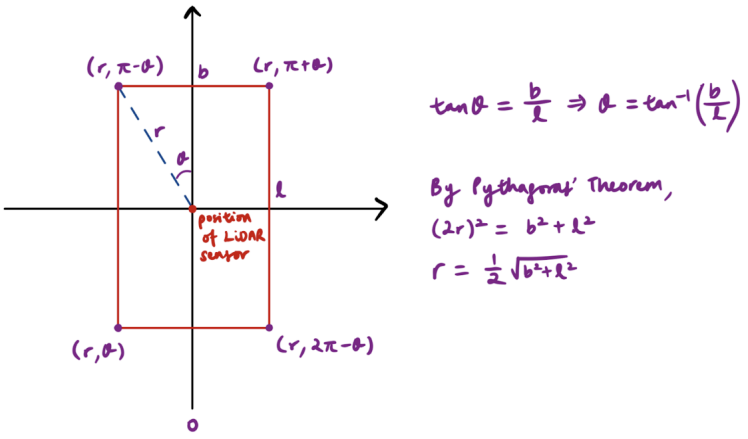

Figure 20. Derivation of Polar Coordinates

```
alex.plot([alex_angle, np.pi - alex_angle, np.pi + alex_angle, 2 * np.pi - alex_angle, alex_angle],
          [alex_hypotenuse] * 5, 'r')
```

Figure 21. Python *matplotlib* code to plot rectangle representing Alex's body

Section 7 | Lessons Learnt – Conclusion

7.1 | Mistakes Made

7.1.1 | Bad Magic Number Errors

```
Command OK
Command OK
Command OK
Command OK
Front Distance:         82
Left Distance:          39
Right Distance:         65
Arduino received bad magic number
```

We did not foresee that we will encounter *bad magic number errors* after implementing the getch() function. As the getch() function allows us to transmit commands to Alex without pressing the "enter" key, when the teleoperator inputs commands too quickly, it would cause the Arduino to receive incomplete packets, resulting in bad magic number errors. After implementing the getch() function, there were numerous times where we encountered the *bad magic number error* (often due to pressing the keys too quickly to position Alex for parking), which forced us to restart our connection with the RPi, resulting in time wasted. To this end, we decided to

implement the *nudging* feature (see **Section 6.2.1**) to allow Alex to rotate over a short distance when needed, reducing the possibility of encountering the *bad magic number error*.

7.1.2 | Planning Ahead for Design

Our group failed to plan early for the design of Alex as we were too focused on ensuring that the functionalities of Alex were working. When we started to assemble the components, we had difficulty figuring out where to place each component as there were issues with weight distribution which caused problems for our turning. We also did not want the wires to be too close to the motors and risk them getting caught by the motors. The LiDAR also had to be positioned high enough to avoid the front ultrasonic sensor but low enough to detect the walls and victims. As such, we wasted a considerable amount of time figuring out Alex's layout, which could have been spent more effectively on testing and practising controlling Alex in the maze. This could have been avoided if we carefully planned and assembled the robot right after its basic movements had been set up at the start. This way, more time would have been available for testing and improving Alex's functionalities.

7.2 | Lessons Learnt

7.2.1 | Power Considerations

As Alex was having difficulties with turning, we decided to increase the voltage supplied to Alex from 6V to 9V. As such, we had to make a decision between using six 1.5V AA batteries or one 9V battery. One main advantage of using one 9V battery is that it takes up less space on Alex as compared to 2 battery holders for the AA batteries. However, after some experimentation, we realised that the battery life of the 9V battery is significantly shorter than that of the AA batteries. From our trial runs, the average battery life of a 9V battery was around 15 minutes before it could no longer power the motors to turn. Thus, even though the 2 battery holders are heavier than the 9V battery, we decided to opt for the AA batteries to leverage on the longer battery life of AA batteries as we wanted Alex to run for as long as possible and detect all the victims. We soldered 2 AA battery holders together and mounted them onto the back of Alex. The average battery life of six 1.5V AA batteries was around 4 hours, significantly longer than that of the 9V battery.

In the future, depending on the specification of the project, we can make a more informed decision on how to power our robot. If the robot is lightweight and only needs to be powered on for a short duration, a single 9V battery would suffice. Else, if the robot weight is not a major concern or if it needs to be powered on for a longer duration, we should use the AA batteries instead.

7.2.2 | Code Backups and Testing

We also learnt the importance of implementing changes iteratively and creating backups of our code. Throughout the entire journey of our project, we would periodically back up our code if major changes were to be made. By running the code after every major change, we are able to test the functionalities of Alex and make sure everything is working as intended. In the event that there are bugs in our code which causes our programme to crash, we would be able to tell which version to revert our code back to by using the latest backup of our code, which acts as a safety net. In future projects, we can mitigate risks more effectively with prior experience in implementing this iterative process.

References

Boston Dynamics. (n.d.). Atlas. Retrieved from https://bostondynamics.com/atlas/

Chen, G., & Anderson, B. (2024). Tutorial: Robot operating system for EPP2. Retrieved from https://www.comp.nus.edu.sg/~guoyi/tutorial/cg2111a/ros-network/

Karunakar, V. (2023). Difference between getc(), getchar(), getch() and getche(). Retrieved from https://www.geeksforgeeks.org/difference-getchar-getch-getc-getche/

Pliant Energy. (n.d.). Robotics. Retrieved from https://www.pliantenergy.com/robotics

Wevolver. (n.d.). Atlas Robot. Retrieved from https://www.wevolver.com/specs/atlas.robot

Appendices

Appendix A | Packet Contents

```
// Packet types
typedef enum
{
  PACKET_TYPE_COMMAND = 0,
  PACKET_TYPE_RESPONSE = 1,
  PACKET_TYPE_ERROR = 2,
  PACKET_TYPE_MESSAGE = 3,
  PACKET_TYPE_HELLO = 4
} TPacketType;

// Response types. This goes into the command field
typedef enum
{
  RESP_OK = 0,
  RESP_STATUS=1,
  RESP_BAD_PACKET = 2,
  RESP_BAD_CHECKSUM = 3,
  RESP_BAD_COMMAND = 4,
  RESP_BAD_RESPONSE = 5,
  RESP_COLOUR = 6,
  RESP_DIST = 7
} TResponseType;


// Commands
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_COLOUR = 7,
  COMMAND_DIST = 8,
  COMMAND_NUDGE_FORWARD = 9,
  COMMAND_NUDGE_BACKWARD = 10,
  COMMAND_NUDGE_LEFT = 11,
  COMMAND_NUDGE_RIGHT = 12
} TCommandType;
```

Appendix B | Bare Metal Programming for Hardware Components

Buzzer

```
void setupBuzzer() {
  //Set buzzer as OUTPUT and turn off buzzer
  DDRB |= (1 << buzzer);
  PORTB |= (1 << buzzer);
}

void buzz() {
  //Turn on buzzer
  PORTB &= ~(1 << buzzer);
  delay(500);

  //Turn off buzzer
  PORTB |= (1 << buzzer);
  delay(500);
}
```

Ultrasonic Sensor

```
void setupUltrasonic() {
  //Set trigPin as OUTPUT and echoPinF, echoPinL, echoPinR as INPUT
  DDRG |= (1 << trigPin);
  DDRG &= ~(1 << echoPinL_mapped);
  DDRL &= ~(1 << echoPinR_mapped);
  DDRB &= ~(1 << echoPinF_mapped);
}

unsigned long getDistance(int echoPin) {
  PORTG &= ~(1 << trigPin);
  delayMicroseconds(2);
  PORTG |= (1 << trigPin);
  delayMicroseconds(10);
  PORTG &= ~(1 << trigPin);

  unsigned long duration = pulseIn(echoPin, HIGH);
  return duration * SPEED_OF_SOUND / 20000.0; //returns the distance in cm
}

void sendDistance() {
  distFront = getDistance(echoPinF);
  distLeft = getDistance(echoPinL);
  distRight = getDistance(echoPinR);

  TPacket distancePacket;
  distancePacket.packetType = PACKET_TYPE_RESPONSE;
  distancePacket.command = RESP_DIST;
  distancePacket.params[0] = distFront;
  distancePacket.params[1] = distLeft;
  distancePacket.params[2] = distRight;
  sendResponse(&distancePacket);
}
```

Colour Sensor

```
void colourSetup() {
  DDRA |= ((1 << S0) | (1 << S1) | (1 << S2) | (1 << S3)); //Set S0, S1, S2, S3 to OUTPUT
  DDRA &= ~(1 << sensorOut_mapped); //Set sensorOut to INPUT

  //Setting frequency scaling to 20%
  PORTA |= (1 << S0);
  PORTA &= ~(1 << S1);
}

int avgFreq() {
  int reading;
  int total = 0;

  for (int i = 0; i < 5; i++) {
    reading = pulseIn(sensorOut, LOW);
    total += reading;
    delay(colorSensorDelay);
  }

  return total / 5;
}
void findColor() {
  // Setting RED filtered photodiodes to be read
  PORTA &= ~((1 << S2) | (1 << S3));
  delay(colorSensorDelay);

  // Reading the output frequency for RED
  redFreq = avgFreq();
  delay(colorSensorDelay);

  // Setting GREEN filtered photodiodes to be read
  PORTA |= ((1 << S2) | (1 << S3));
  delay(colorSensorDelay);

  // Reading the output frequency for GREEN
  greenFreq = avgFreq();
  delay(colorSensorDelay);

  // Setting BLUE filtered photodiodes to be read
  PORTA &= ~(1 << S2);
  PORTA |= (1 << S3);
  delay(colorSensorDelay);

  // Reading the output frequency for BLUE
  blueFreq = avgFreq();
  delay(colorSensorDelay);
}

void sendColour() {
  unsigned long distance = getDistance(echoPinF);

  TPacket colourPacket;
  colourPacket.packetType = PACKET_TYPE_RESPONSE;
  colourPacket.command = RESP_COLOUR;
  colourPacket.params[0] = redFreq;
  colourPacket.params[1] = greenFreq;
  colourPacket.params[2] = blueFreq;
  colourPacket.params[3] = distance;
  sendResponse(&colourPacket);
}
```

Appendix C | Code Snippet for Nudging

```
case COMMAND_NUDGE_FORWARD:
    sendOK();
    dir = (TDirection) FORWARD;
    move(MOVEMENT_SPEED, FORWARD);
    delay(NUDGE_DELAY);
    stop();
  break;

case COMMAND_NUDGE_BACKWARD:
    sendOK();
    dir = (TDirection) BACKWARD;
    move(MOVEMENT_SPEED, BACKWARD);
    delay(NUDGE_DELAY);
    stop();
  break;

case COMMAND_NUDGE_LEFT:
    sendOK();
    dir = (TDirection) LEFT;
    move(MOVEMENT_SPEED, CCW);
    delay(NUDGE_DELAY);
    stop();
  break;

case COMMAND_NUDGE_RIGHT:
    sendOK();
    dir = (TDirection) RIGHT;
    move(MOVEMENT_SPEED, CW);
    delay(NUDGE_DELAY);
    stop();
  break;
```

Appendix D | Source Code for Manual Implementation of getch()

```
/*****************************************************************************
*    Title: Manual Implementation of getch()
*    Author: Mole, A. & Aman
*    Date: 2023
*    Code version: 1.0
*    Availability: https://stackoverflow.com/questions/41407242/declaring-the-getch-function/77131663#77131663
*
*****************************************************************************/



char getch (void)
{
    /*
        This function requires
        #include <stdio.h>
        #include <termios.h>
        #include <unistd.h>
    */

    fflush(stdout);
    struct termios raw;
    tcgetattr(STDIN_FILENO, &raw);
    struct termios aa = raw;
    aa.c_lflag &= ~(ECHO | ICANON);
    tcsetattr(STDIN_FILENO, TCSAFLUSH, & aa);

    char ch;
    read(STDIN_FILENO, &ch, 1);
    tcsetattr(STDIN_FILENO, TCSAFLUSH, & raw);
    return ch;
}
```