CG2111A Engineering Principle and Practice
Semester 2 2023/2024

"Alex to the Rescue"
Final Report
Team: B05-5A

Name	Student #	Sub-Team	Role
Tong Kian Kiat	A0271934Y	Software	Report and Software Editor
Tan Ye Xin	A0283199M	Hardware	Report Editor and Hardware Lead
Nigel Yeo Tong Wei	A0271565Y	Firmware	Report Editor and Firmware Implementation
Sugumar Danusan	A0284724X	Software	Report and Software Editor

Section 1 Introduction **3**
Section 2 Review of State of the Art **3**
2.1 Battlefield Extraction-Assist Robot (BEAR) 3
2.2 ANYmal 4
Section 3 System Architecture **4**
3.1 Simplified UML Diagram 4
3.2 Electrical Schematic Diagram 5
3.2.1 HW-130, Arduino Mega 2560 and motor connections 5
3.2.2 Electrical components and Arduino Mega 2560 connections 5
3.2.3 Raspberry Pi 4B connections 6
Section 4 Hardware Design **6**
4.1 Alex's Design Framework 7
4.2 Alex's Hardware Functionalities 7
4.2.1 HR-SR04 Ultrasonic Sensor 7
4.2.2 TCS-3200 Colour Sensor 7
4.2.3 Removal of tire around wheel 8
4.2.4 Securing of LiDAR to Alex 8
4.3 Alex's Additional Hardware Functionalities 8
4.3.1 Speaker Module 8
Section 5 Firmware Design **8**
5.1 Main algorithm running on Arduino Mega 2560 9
5.2 Communication Protocol 9
5.3 Motor Control 9
5.4 Speaker Module 10
5.5 Power Efficiency 10
Section 6 Software Design **10**
6.1 High Level Algorithm for the Raspberry Pi 4B and Client 10
6.2 Teleoperation of Alex 10
6.3 Colour Detection 12
6.4 Ultrasonic Sensor 12
6.5 Hector SLAM 13
6.6 RViz Setup 13
Section 7 Lessons Learnt - Conclusion **14**
7.1 Mistake 1: Error in Bare-Metal Programming 14
7.2 Mistake 2: Planning Alex's layout earlier before testing other functionalities 14
7.3 Lesson 1: Version Control and Backup 14
7.4 Lesson 2: Efficient and Effective Debugging 14
Section 8 Appendix **16**

Section 1 Introduction

Alex is a rescue robot designed to navigate the unknown territory of the unknown rescue site. Alex will provide rescuers with data such as the layout of the rescue site as well as identify the state of the victims that are inside. This information is used by the rescuers to decide the next course of action and more efficiently navigate and rescue the victims.

Alex will need to return the environmental data of the surroundings constantly back to the client. This will allow Alex to be easily teleoperated by the client so that it can efficiently do its job of providing the necessary information while leaving the decision making to the user operating him. Due to the nature of the rescue site having much uneven ground and obstacles, Alex has to be able to navigate inside carefully. Thus, Alex has to be equipped with a robust movement system that allows him to manoeuvre through even the toughest of routes to explore the entire site easily.

Alex must also be able to identify the state of the victims inside the site of rescue. In the simulation that Alex will be put through, the state of the victims will be colour coded and Alex has to identify the colours and send the state of these victims back to the client. This is similar to how Alex might need to identify the state of the victims in real life through identification of certain signs of fatality such as the presence of blood, etc.

Section 2 Review of State of the Art

This section showcases the 2 tele-operated search and rescue robots platforms.

2.1 Battlefield Extraction-Assist Robot (BEAR)

Developed by Vecna Robotics, BEAR is a humanoid robot that can locate victims in mineshafts, battlefields, toxic spills, or earthquake-damaged structures. BEAR has sensors and cameras implemented that allows it to detect obstacles in its path and avoid them, and is able to operate autonomously, allowing it to accomplish different tasks without constant need for supervision, and can be teleoperated as well for specific task completion.

BEAR's ability to navigate quickly using its equipped sensors and cameras, as well as its capability to carry heavy objects and people make it a great search and rescue operator. However, its size might make it difficult for it to fit in certain narrow spaces without breaking obstacles nearby, which could put the lives of survivors in greater danger. Comparatively, Alex has a tiny and compact frame, together with its robust movement control, it will be able to move through tight angles and spots without causing disturbance to the environment.

Figure 1: BEAR

Figure 2: ANYmal

2.2 ANYmal

ANYmal is a quadrupedal robot developed by ANYbotics for autonomous operation in challenging environments. ANYmal is equipped with laser scanners and contact sensors in order to properly map out its environment, and moves accordingly to avoid obstacles and traverse terrain. It also has a thermal camera and ultrasonic microphone to detect and identify survivors in its surroundings. ANYmal establishes a teleoperation relationship with a human operator to complete its tasks, and is able to send feedback and communication during search and rescue operations. A warning light is also equipped on it to alert nearby operators of any dangers.

ANYmal's contact sensors allow it to adapt to any changes in the environment. Its high mobility allows it to move around quickly in search and rescue operations, while being able to traverse vertically and horizontally efficiently. However, ANYmal does not have any way to pass through or clear debris blocking its path, which might limit its usability in disaster situations. It also has a low battery run-time of 90 minutes. Comparatively to Alex, Alex has a high efficiency system, keeping its power output to a minimum, allowing him to run for longer hours during an arduously long process of search and rescue.

2.3 Summary

Upon analysing these 2 search and rescue robots, we concur that a search and rescue robot should consist of the following functionalities: smooth teleoperation connectivity, a precise and accurate mapping function, as well as an alert system to inform nearby rescuers if victims are located. Alex is implemented with wireless capability using the Raspberry-Pi so that it can be controlled wirelessly, a LiDAR that we have calibrated extensively to produce a proper SLAM map reading that we can interpret in order to give us the best chances of identifying and rescuing victims in a search and rescue situation. We have also implemented a speaker module to alert any nearby rescuers on the physical state of the victims located.

Section 3 System Architecture

The following shows the overall view of the system architecture and gives a look into the low level connections within each subsystem that make up Alex entirely.

3.1 Simplified UML Diagram

The UML diagram is designed to give an overview of how the different subsystems in Alex communicate with one another. The diagram provides information as to which components are drawing data and which components are providing them.

The yellow boxes represent the hardware components, the light blue boxes represent the software components within the hardware and the red boxes represent the components supplying power.

For example, a typical cycle starts from the client laptop, which will give user input through the TLS server to the client handler on the Raspberry Pi 4B. The handler will process the data and communicate with the Arduino Mega 2560 via UART. Depending on the type of command packet received, the Arduino will execute the command accordingly.

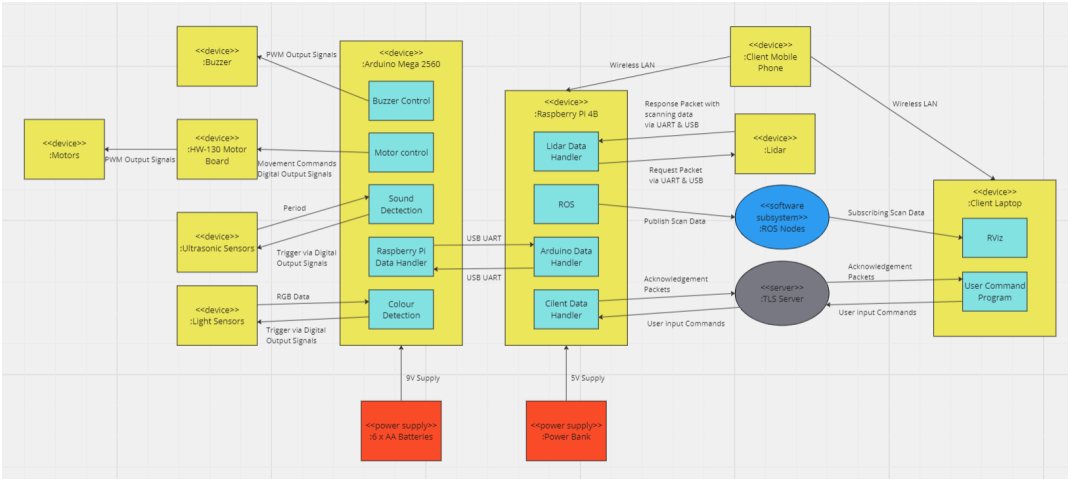


Figure 3: Simplified UML Diagram

3.2 Electrical Schematic Diagram

The Electrical Schematic Diagrams depicts the connections between the Arduino Mega, Raspberry Pi, and the relevant hardware components

3.2.1 HW-130, Arduino Mega 2560 and motor connections

The following shows the electrical schematic of the connections within the Arduino Mega 2560 and HW-130 Motor Board as well as between them. Additionally, there are 4 motors namely M1, M2, M3 and M4 connected to the HW-130 Motor Board as shown below.

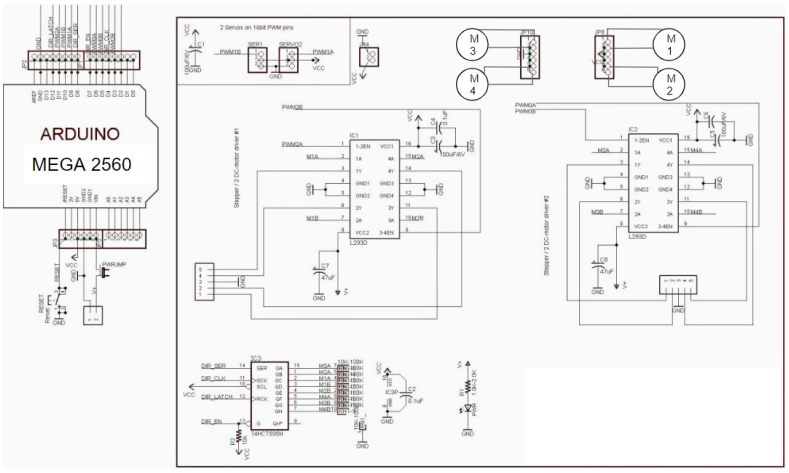


Figure 4: Arduino Mega 2560 connections with HW-130 and Driver Motors

3.2.2 Electrical components and Arduino Mega 2560 connections

The following shows the electrical schematic of the connections between the electrical components with the Arduino Mega 2560. The main electrical components are the ultrasonic sensors, colour sensor (TCS-3200) and speaker.

The speaker is connected to the PWM pins of the Arduino Mega 2560 while the others are connected to the available digital pins. Note that most of the pins are taken up by the HW-130 Motor Board as seen in 3.2.1 above.

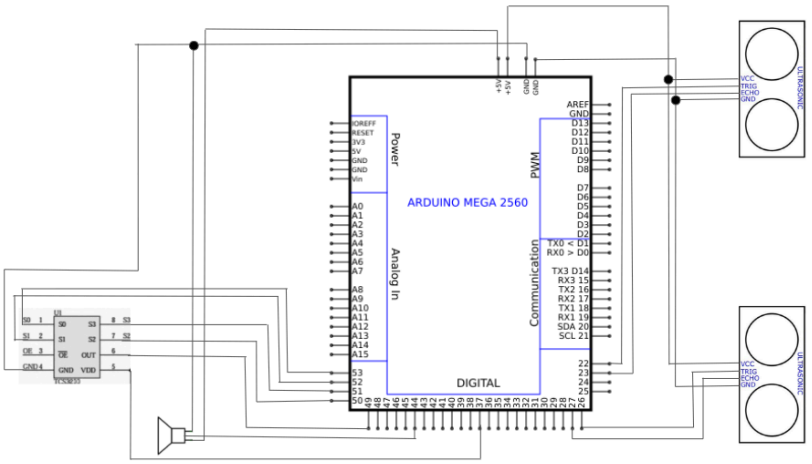


Figure 5: Arduino Mega 2560 connections with Ultrasonic Sensors, TCS-3200 Colour Sensor and Speaker Module

3.2.3 Raspberry Pi 4B connections

The Raspberry Pi 4B is used mainly as a source of connection between Alex and the client over the TLS server. The information is then communicated to the Arduino Mega 2560 via UART. None of the GPIO pins on the Raspberry Pi 4B are being used. The Raspberry Pi 4B is also connected to the Lidar through a signal adapter.

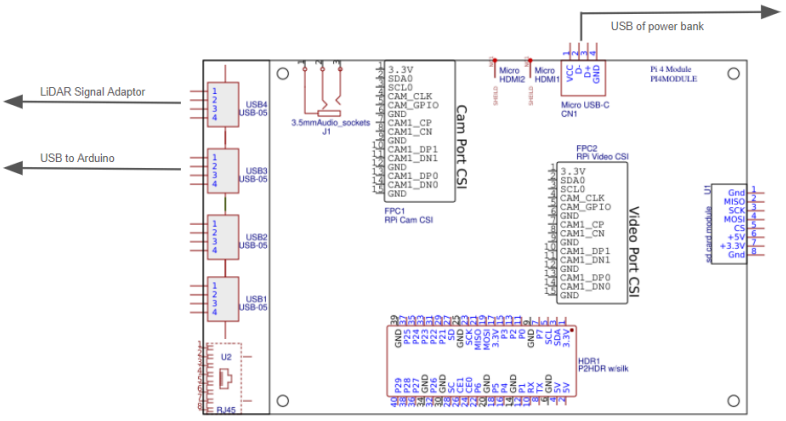


Figure 6:Raspberry Pi 4B Connections with Speaker Module, Arduino Mega 2560 and LiDAR

Section 4 Hardware Design

This section showcases the design considerations that we have done on Alex.

4.1 Alex's Design Framework

Alex consists of numerous hardware components which are required for him to perform its necessary capabilities for search and rescue. Figure 6 shows the side view with all the components that are mounted on Alex and their relative placements. Figures 7 and 8 show the front and top view of Alex respectively to give a clearer image of how certain components are placed.

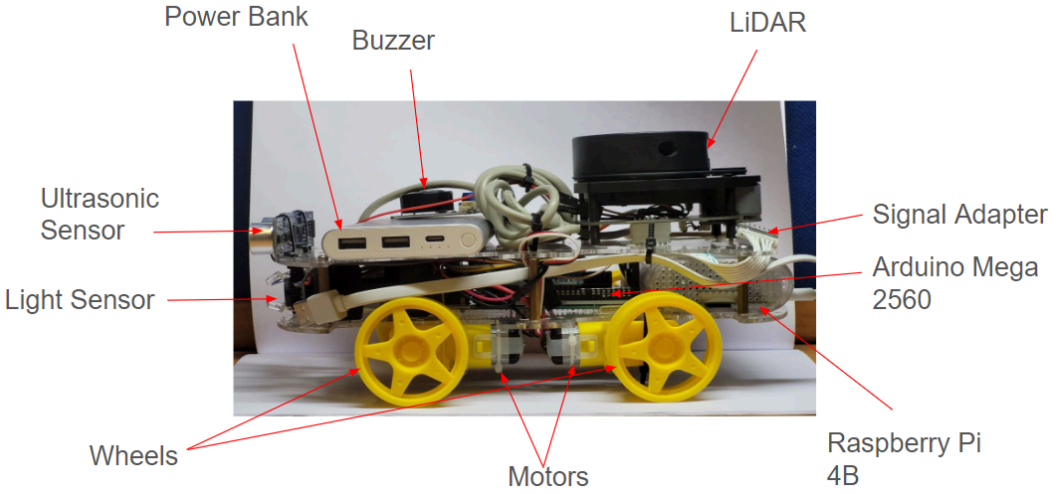


Figure 7: Side View of Alex with Annotations

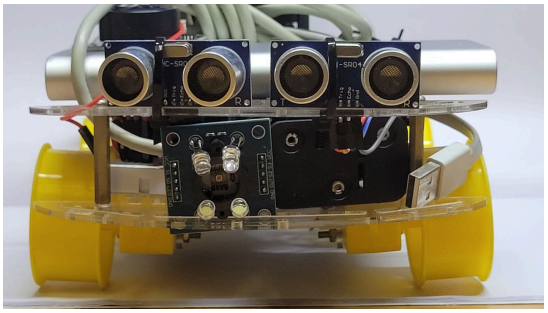


Figure 8: Front View of Alex

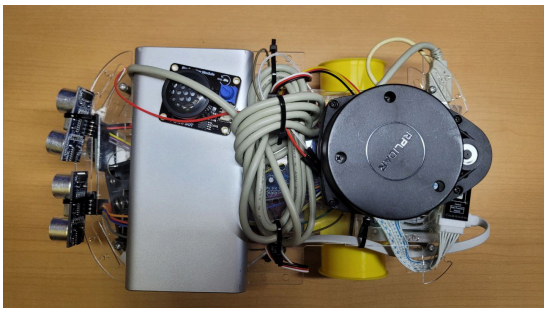


Figure 9: Top View of Alex

4.2 Alex's Hardware Functionalities

In this section, we will highlight the different types of hardware mounted onto Alex, explain their functionality and how they help Alex to navigate and perform search and rescue.

4.2.1 HR-SR04 Ultrasonic Sensor

Additional ultrasonic sensors are mounted onto the front of Alex as shown above. The client obtains real time environment data to control Alex to avoid obstacles. Though the movement commands of Alex are already robust enough, to factor in the possibility of human or equipment errors that causes the client to send Alex straight into an obstacle, Alex has been equipped with these sensors as an additional redundancy to stop itself before collision and alert the client about the obstacle ahead.

4.2.2 TCS-3200 Colour Sensor

Alex is also fitted with a colour sensor at its front in order to identify the colour of the victim (red/green) or object (white). The colour sensor works by having the reflected light passing through a red, green, blue and clear filter onto photodiodes, hence determining the intensity

of the light rays reflected from the object when the LEDs around the colour sensor is turned on for each RGB ray. The sensor will then output a squarewave, with the frequency being dependent on the detected intensity for each RGB ray. We calibrated the colour sensor by taking in multiple readings of the desired colours, and determining the pattern of RGB values for each of the relevant colours. An algorithm is then written to send the detected colour to the user when the command for colour detection is sent to Alex.

4.2.3 Removal of tire around wheel

During our testing, we realised that the Alex had difficulty turning on the floor. After further experimentation, we discovered that Alex was able to turn more effectively without the tires attached to the wheel, hence prompting us to proceed without the tires.

4.2.4 Securing of LiDAR to Alex

In order to ensure accurate LiDAR readings and clear SLAM mapping, we made precise measurements to drill fitting holes for the legs of the LiDAR such that the LiDAR can mount snugly onto the chassis of Alex without any obstruction blocking the LiDAR light emitter and receiver. This allows the LiDAR to remain stable and give environmental data with high accuracy.

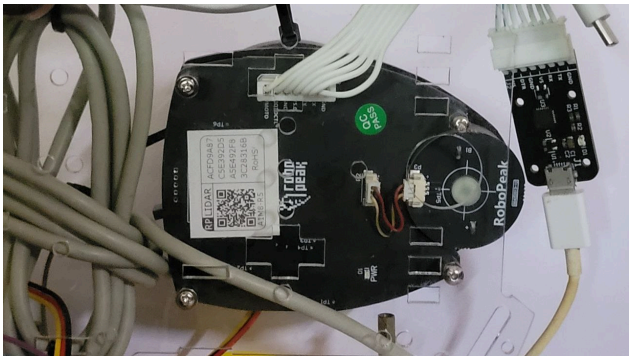


Figure 10: Mounting of LiDAR onto the chassis

4.3 Alex's Additional Hardware Functionalities

Alex is also fitted with additional hardware functionalities to further aid in search and rescue operations.

4.3.1 Speaker Module

Additionally, a speaker module is also mounted onto Alex as shown above. A different tune will be played according to the state of the victim. This allows Alex to verbalise the state of the victims that he has discovered, allowing the rescuers nearby to be alerted, and can find and respond swiftly, while at the same time knowing the severity of the state of the victim.

Section 5 Firmware Design

For the firmware of Alex, Alex communicates with the Raspberry Pi 4B via USB, it handles the packets that it receives from the Raspberry Pi 4B to instruct the different hardware components to perform the command that was sent to him.

5.1 Main algorithm running on Arduino Mega 2560

1. Arduino is powered on together with Raspberry Pi 4B and a hello packet is exchanged between them to establish a working connection
2. Alex polls for a command packet that it will receive from the Raspberry Pi 4B
3. Upon receiving the command packet, it checks for the validity of the packet
4. The packet is deserialized to obtain the packet type and packet command
5. One of the 7 commands possible commands will be executed according to the one Alex receives (Refer to Section 6)
6. Repeat step 2-6

5.2 Communication Protocol

The communication between the Raspberry Pi 4B and the Arduino Mega 2560 is done via serial connection through USB UART, communicating at 9600 bps, 8N1 format. Each command is sent as a structure TPacket which will serialise and deserialize each time it is sent and received. Each data packet is 100 bytes long, and contains information such as the packet type, command and other essential information as shown 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;
```

Once the Arudino receives the command packet from the Raspberry Pi 4B, it will check for its validity via readPacket() function and if the packet is valid it will proceed to handle it, else it will send a bad packet response back to the client.

Packet Handling is done by first identifying which packet type is being received. For this segment, we look at only user command packets as the rest is used for error handling and connection establishment at the start. When a user command packet is received, handleCommand() is called. The command variable is then taken from the packet to identify and carry out the client's instructions accordingly.

```
void handleCommand(TPacket *command)
{
  switch (command->command)
  {
    case COMMAND_FORWARD:
      sendOK();
      // carry out instructions
      break;
    case COMMAND_REVERSE:
      sendOK();
      // carry out instructions
      break;
    ...
  }
}
```

5.3 Motor Control

The motors on Alex are controlled by the HW-130 motor board and <AFMotor.h> library is used to aid the movement of Alex. The HW-130 Motor Board is connected to PWM pins on the Arduino Mega 2560. Based on commands of the client, the H-bridge on the motor board

is used to change the directions of the motors while the PWM pins duty cycle will be set from 0 to 100 to regulate the speed of Alex.

5.4 Speaker Module

When a colour is detected by the colour sensor, this will trigger the speaker to play a tune based on whether the colour detected is red or green. A red victim will play an ambulance tune, while the green victim will play a short verse from the "Super Mario Bros" theme song. Each note is denoted by a specific frequency, and the notes to be played are stored in an array, which is iterated through with a delay in between each note. The tone() function plays the note through the speaker based on the duration and note.

5.5 Power Efficiency

To increase the duration of usage of Alex in a single session, only the necessary components and parts will be powered on while the others will be disconnected from the power source until required. An example would be the light sensors which would be powered on with the 4 LED lights lit up at all times. To reduce power usage from the sensor, it will thus be connected to the arduino but not drawing any power as a switch pin will be written to LOW until Alex needs to detect colour and the pin will be pulled to HIGH and turning on the colour sensor.

Section 6 Software Design

The main software functionalities of Alex would include the teleoperation controls which allows the user to give inputs to manoeuvre Alex though all types of situations.

6.1 High Level Algorithm for the Raspberry Pi 4B and Client

1. Client establishes a VNC connection with the Raspberry Pi 4B on his laptop
2. Client runs main alex-pi script on the the Raspberry Pi 4B which also starts up the TLS server
3. Client establishes a TLS connection between his laptop and Alex after running tls-client script and doing key exchange
4. Client sets up necessary ROS packages to start up LiDAR, SLAM and ROS networking on the Raspberry Pi 4B
5. Client establishes ROS connection between his laptop and Alex and subscribes to the LiDAR data
6. Client runs RViz on his laptop to visualise the environment Alex is in
7. Client sends commands over to Alex continuously via the TLS server concurrently with RViz until the mission is complete. Commands are serialised and sent over to the Arduino via serial communication as explained in section 5

6.2 Teleoperation of Alex

There are mainly 7 types of commands that the interface allows the user to input:

key	'f'	'b'	'l'	'r'	's'	'g'	'c'
command	Move forward	Move backward	Turn left	Turn right	Emergency stop	Semi-auto forward movement	Colour detection

The first 4 commands are the four essential basic movement commands that allow Alex to move in all directions. Each of these commands is set to have fixed durations of movements measured using a timer onboard the Arduino which is started using the startTimer() function within each movement function.

We configured timer 5 using bare metal to produce an interrupt after 0.5 seconds. The interrupt stops the motor and the timer. This timer is used to fix the moving duration of Alex so that he can easily move through tight areas and give the user a higher amount of control. The code for the setupTimer() and startTimer() functions are shown below.

```
void setupTimer() {
  TCCR5A = 0;
  OCR5AH = 0b01111010;
  OCR5AL = 0b00010010;
  TIMSK5 = 0b00000010;
}

void startTimer() {
  //clear counter as a good practice
  TCNT5H = 0;
  TCNT5L = 0;
  TCCR5B = 0b00001101;
}
```

The interval has been calibrated such that the forward and backward movements are fixed to inching 5cm and 3cm in their respective directions. For forward movement, to decrease the chances of collision even further, a check on whether there is an obstacle in front of Alex is also done via the checkUS() function which returns true when the distance between Alex and an object in front of him is 5 cm or less, stopping Alex and the timer immediately. Alex then sends a message to the client to alert him that there is an obstacle as seen from the code below. The check is done in the loop() by the code shown below.

```
if (dir == FORWARD || dir == FORWARD_AUTO) {
   if (checkUS()) {
     stop();
     stopTimer();
     sendMessage("Obstacle");
   }
 }
```

Turning commands takes in an additional input parameter from the user of 0 to 100. This number scales the power that is used to turn the motors. As the turn is in place, there is no risk of collision and this allows the user to turn Alex quickly to any direction that the client desires before moving forward again.

Due to the slower nature of inching forward from the usual forward command, an additional semi-auto movement command is implemented to speed up the process of moving Alex forward when the area in front of Alex is clear as time is of the essence during search and rescue operations. When the user sends this command, Alex will continuously move forward while doing checkUS() and only stops when checkUS() returns true.

The emergency stop command gives the user additional control to stop Alex in the event that the user deems the need to. The colour detection command will be further discussed in the following paragraph.

6.3 Colour Detection

When the colour detection command is called by the user, the TCS-3200 will send data back to Alex. The data includes 3 different values, taken from the pulseIn() of 3 squarewaves of different frequencies generated corresponding to the red, green and blue colour frequencies emitted from the sensor. The 3 values are then used to differentiate between red, green and white colours.

After much experimentation, the two main factors affecting the reading of the values of the colour sensors are mainly ambient lighting and distance between Alex and the object to be detected. The algorithm is tested at fixed distances with various different lighting conditions and it was found that the distinct differentiating factor between the colours red and green was in the red and blue value readings. The general trend is that the colour red has a higher blue value reading than red value reading and vice versa for the colour green. The colour white generally has all 3 readings significantly lower than the readings for the colours red and green.

However, the readings of any colour will increase exponentially as the distance increases between Alex and the object to detect. To counter this, the client will always align Alex to the object to detect before moving towards it until the checkUS() function mentioned above is triggered. This allows a relatively constant distance between Alex and the object and more consistent readings for the different colours. An experiment of 30 data sets were taken, each with distances 4, 5 and 6 cm between Alex and the object. Disregarding anomalies, the lowest red readings for red and green colour is averaging 180 and the highest red reading for white colour is averaging 160, thus we set the condition where any values under 170 for red value readings will identify the object as white.

```
const char *detectColourdiff() {  // keep colour sensing distance between 5
to 10 cm
    //dprintf("detecting colour\n");
    if (redPW <= 170) {
      return "wht";
    } else if (redPW >= bluePW) {
      return "grn";
    } else if (redPW <= bluePW) {
      return "red";
    }
  }
```

6.4 Ultrasonic Sensor

The ultrasonic sensor is used for crash prevention and aids in the consistency of colour sensor readings as mentioned above. The sensor returns the distance value every time the checkUS() function is called. The checkUS() function writes to the trigger pin and reads the pulseIn() of the echo pin. This gives the duration from when the wave is sent and received back by the sensor. Simple calculation is then done to convert this value to the distance between Alex and the object in front of him.

The limitation of this sensor is that it requires the object to be rather perpendicular to Alex for detection to occur. If the angle is too wide, the waves are reflected off the object in front and do not return back to the sensor. To counter this we have added 2 ultrasonic sensors at the front of Alex to increase the range of the angles that Alex is able to detect an obstacle at,

allowing it to detect objects and stop himself from crashing in the range of +/- 30 degrees, taking forward as 0 degrees. As the sensor is used to increase consistency of colour sensor reading distance, the client will have to ensure that Alex is perpendicular to the object of detection before moving Alex forward.

6.5 Hector SLAM

Alex uses the SLAMTEC RPLidar A1 (A1M8) to scan its environment. The Lidar will spin continuously, taking in scan data based on the distances of the nearest object to the Lidar at every angle. The LiDAR is connected to a signal adapter and to the Raspberry Pi 4B via USB where information of the scan data will be transmitted through.

The client connects to the same local network as Alex and through ROS Networking, the client subscribes to the node with the LiDAR scan data that Alex is publishing to. A visualisation tool such as RViz is then used on the client's laptop to produce the SLAM. Localisation and making use of known landmarks allows the SLAM algorithm to map out the environment around Alex. By utilising closed loop detection, this allows Alex to identify when it has already visited a location, allowing SLAM to then minimise and fix any mapping errors.

Due to the nature of hector SLAM, it causes drawbacks such as having a limit as to how fast Alex can move. This is due to the usage of landmarks which requires time for the algorithm to recognise and moving too fast will cause the map to become distorted as it could not identify the landmarks fast enough. This becomes especially sensitive when it comes to turning Alex. Thus, for our implementation of movements, our forward and backward movements are short enough of a distance to account for this and the maximum turning speed is also tuned to be lower than the limit allowed.

6.6 RViz Setup

To help the client drive Alex around its surroundings, certain configurations have been made to the visualisation tool, RViz. After running Rviz, under display, for the 'Map' configurations, the topic is changed to /scan instead of /map. This instead causes the map on RViz to turn according to the orientation of Alex when Alex turns. This reduces the client's need to visualise which direction Alex needs to turn relative to Alex's orientation in the map, easing driving of Alex. Alex will also be fixed onto the centre grid of the map with this configuration, as Alex is not simply a point and it is hard to visualise the size of Alex on the map, we have also set the offset of the x-axis to -0.2. This allows the point of the centre grid to represent the head of Alex which helps the client get a better estimate of how close Alex is to any object nearby.

Section 7 Lessons Learnt - Conclusion

7.1 Mistake 1: Error in Bare-Metal Programming

When implementing the colour sensor, we did our testing on a separate .ino file using bare-metal implementation before importing it over to our main Alex.ino code. However, when we added the colour-sensor code, all our motors suddenly stopped spinning, and Alex refused to move despite ensuring that the connection between our client and server side was not broken or disconnected. After a few hours of debugging, we realised that the bare-metal programming for our colour sensor was not done using masking, but implemented with direct assignment using (=) rather than (|=). This caused our motors to not work, as our motor pins made use of the same PORTx pins as the colour sensor, and we turned them off when we used assignment rather than masking, which would have kept the other bits at their original values. By changing our PORTx implementation to masking, we managed to get Alex's motors to start working again.

7.2 Mistake 2: Planning Alex's layout earlier before testing other functionalities

When the project commenced, we were initially tunnel-visioned on getting our various functionalities, such as the motors, colour sensors, and ultrasonic sensors to work. As a result, we did not plan how the various components would be placed on Alex until later on when we got our hardware and software functionalities to work. This caused us to spend an additional few hours during the week before our trial run to remove and replug in all the hardware components on Alex to ensure that the weight would be distributed evenly, and nothing would be blocking our LiDAR.

7.3 Lesson 1: Version Control and Backup

When we were editing our code in order to add more commands, we encountered a fatal bug in our code, which caused our motors to stop working completely. Luckily, we had the original copy of all our codes saved, which allowed us to revert back and start again from scratch. This taught us that we should always save our code and back it up regularly, especially when working on larger-scale projects. If we had encountered such bugs closer to our trial run, or when our robot was fitted with all our additional functionalities, reverting to the original skeleton code would cause us to lose way more progress and time.

7.4 Lesson 2: Efficient and Effective Debugging

We encountered another fatal bug when we tried to implement our colour sensor together with the rest of our functionalities, where Alex's motors refused to turn. Initially, we tried to revert our code back to the skeleton code, but that did not solve our problem. We then commented out all the additional functionalities that we added, and traced out the logic of our code, before proceeding to uncomment line by line and running the code again and again until we got our motors to start spinning. This taught us to perform debugging systematically and to also take a step back to trace out the logic of our code, and pick out specific lines that we think might cause the problem, and perform testing from there.

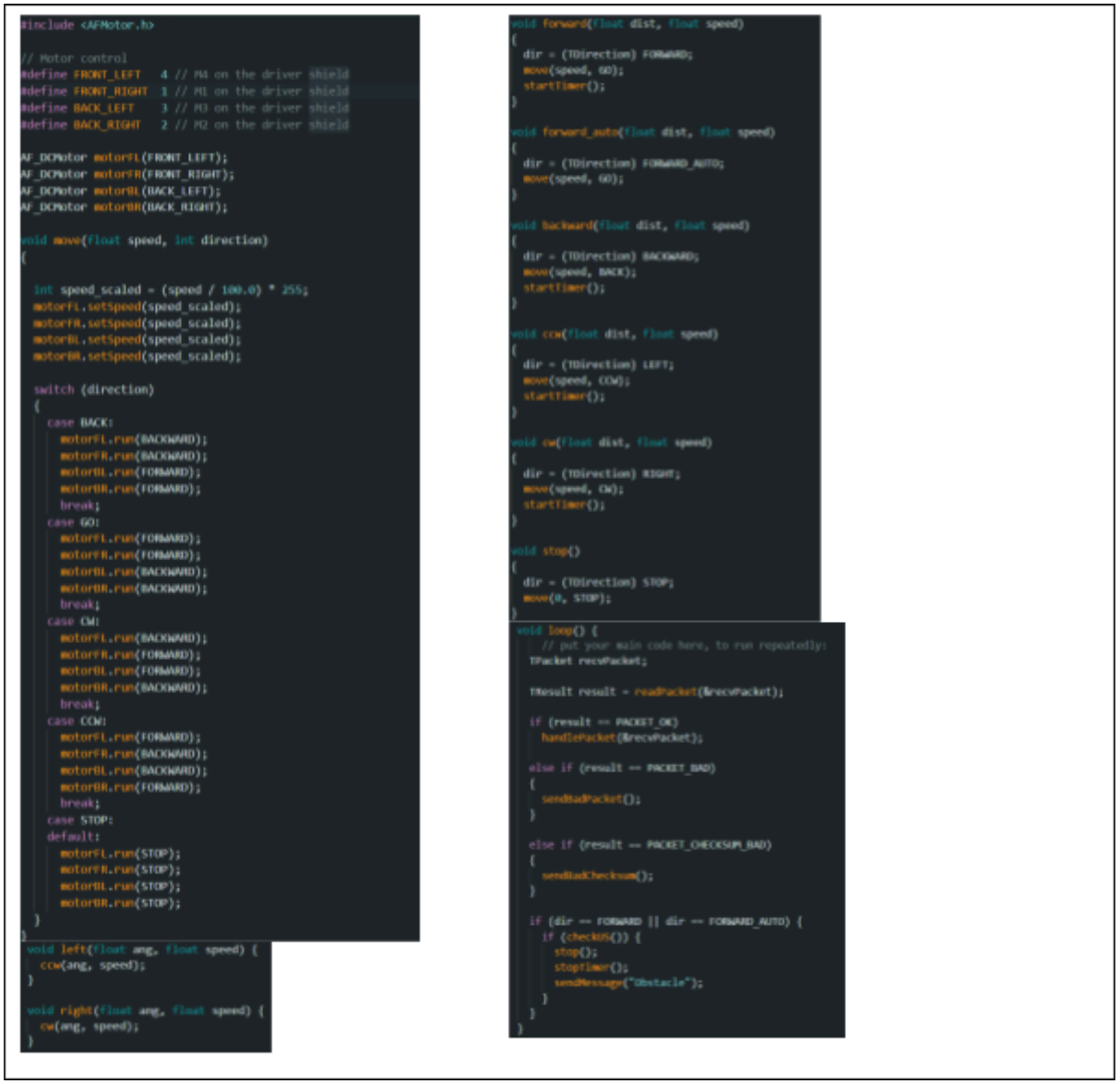
References

https://robotsguide.com/robots/bear?video=2
https://www.cnet.com/science/bear-robot-roars-to-the-rescue/
https://www.linkedin.com/pulse/battlefield-extraction-assist-robot-bear-kipkirui-patrick-limo
https://rsl.ethz.ch/robots-media/anymal.html
https://grobotronics.com/images/companies/1/Datasheet%20for%20ARD2006.pdf?1685516551589
https://easyeda.com/modules/RASPBERRY-PI-4-MODEL-B-SCHEMATIC-1_1673095f168c45af848b2d34b65b3e58
https://www.comp.nus.edu.sg/~guoyi/tutorial/cg2111a/ros-slam/

Section 8 Appendix

The codes for Alex's functions are stored here.

Appendix A. Motor Code



```
#include <AFMotor.h>

// Motor control
#define FRONT_LEFT   4 // M4 on the driver shield
#define FRONT_RIGHT  1 // M1 on the driver shield
#define BACK_LEFT    3 // M3 on the driver shield
#define BACK_RIGHT   2 // M2 on the driver shield

AF_DCMotor motorFL(FRONT_LEFT);
AF_DCMotor motorFR(FRONT_RIGHT);
AF_DCMotor motorBL(BACK_LEFT);
AF_DCMotor motorBR(BACK_RIGHT);

void move(float speed, int direction)
{

  int speed_scaled = (speed / 100.0) * 255;
  motorFL.setSpeed(speed_scaled);
  motorFR.setSpeed(speed_scaled);
  motorBL.setSpeed(speed_scaled);
  motorBR.setSpeed(speed_scaled);

  switch (direction)
  {
    case BACK:
      motorFL.run(BACKWARD);
      motorFR.run(BACKWARD);
      motorBL.run(FORWARD);
      motorBR.run(FORWARD);
      break;
    case GO:
      motorFL.run(FORWARD);
      motorFR.run(FORWARD);
      motorBL.run(BACKWARD);
      motorBR.run(BACKWARD);
      break;
    case CW:
      motorFL.run(BACKWARD);
      motorFR.run(FORWARD);
      motorBL.run(FORWARD);
      motorBR.run(BACKWARD);
      break;
    case CCW:
      motorFL.run(FORWARD);
      motorFR.run(BACKWARD);
      motorBL.run(BACKWARD);
      motorBR.run(FORWARD);
      break;
    case STOP:
    default:
      motorFL.run(STOP);
      motorFR.run(STOP);
      motorBL.run(STOP);
      motorBR.run(STOP);
  }
}
void left(float ang, float speed) {
  ccw(ang, speed);
}

void right(float ang, float speed) {
  cw(ang, speed);
}
```

```
void forward(float dist, float speed)
{
  dir = (TDirection) FORWARD;
  move(speed, GO);
  startTimer();
}

void forward_auto(float dist, float speed)
{
  dir = (TDirection) FORWARD_AUTO;
  move(speed, GO);
}

void backward(float dist, float speed)
{
  dir = (TDirection) BACKWARD;
  move(speed, BACK);
  startTimer();
}

void ccw(float dist, float speed)
{
  dir = (TDirection) LEFT;
  move(speed, CCW);
  startTimer();
}

void cw(float dist, float speed)
{
  dir = (TDirection) RIGHT;
  move(speed, CW);
  startTimer();
}

void stop()
{
  dir = (TDirection) STOP;
  move(0, STOP);
}
```

```
void loop() {
  // put your main code here, to run repeatedly:
  TPacket recvPacket;

  TResult result = readPacket(&recvPacket);

  if (result == PACKET_OK)
    handlePacket(&recvPacket);

  else if (result == PACKET_BAD)
  {
    sendBadPacket();
  }

  else if (result == PACKET_CHECKSUM_BAD)
  {
    sendBadChecksum();
  }

  if (dir == FORWARD || dir == FORWARD_AUTO) {
    if (checkUS()) {
      stop();
      stopTimer();
      sendMessage("Obstacle");
    }
  }
}
```

Appendix B. Colour Sensor Code

```
// Variables for colour detection
int redPW = 0;
int greenPW = 0;
int bluePW = 0;
bool detected = 0;

// Function to read Red Pulse Widths
int getRedPW() {
  // Set sensor to read Red only
  PORTB &= 0b11110011;
  delay(100);
  // Define integer to represent Pulse Width
  int PW;
  // Read the output Pulse Width
  PW = pulseIn(sensorOut, LOW);
  // Return the value
  return PW;
}

// Function to read Green Pulse Widths
int getGreenPW() {
  // Set sensor to read Green only
  PORTB |= 0b00001100;
  delay(100);
  // Define integer to represent Pulse Width
  int PW;
  // Read the output Pulse Width
  PW = pulseIn(sensorOut, LOW);
  // Return the value
  return PW;
}

// Function to read Blue Pulse Widths
int getBluePW() {
  // Set sensor to read Blue only
  PORTB &= 0b11111011;
  PORTB |= 0b00001000;
  delay(100);
  // Define integer to represent Pulse Width
  int PW;
  // Read the output Pulse Width
  PW = pulseIn(sensorOut, LOW);
  // Return the value
  return PW;
}
```

```
case COMMAND_GET_COLOUR:
  sendOK();
  PORTC |= 0b00000001;
  // Set Pulse Width scaling to 20%
  PORTB |= 0b00000001;
  PORTB &= 0b11111101;
  while (!detected) {
    // Read Red Pulse Width
    redPW = getRedPW();
    // Delay to stabilize sensor
    delay(200);

    // Read Green Pulse Width
    greenPW = getGreenPW();
    // Delay to stabilize sensor
    delay(200);

    // Read Blue Pulse Width
    bluePW = getBluePW();
    // Delay to stabilize sensor
    delay(200);
    detected = true;
  }
  PORTC &= 0b11111110;
  PORTB &= 0b11111100;

  if (detectColourdiff() == "red") {
    sendMessage(detectColourdiff());
    sendMessage(String(redPW).c_str());
    sendMessage(String(bluePW).c_str());
    sendMessage(String(greenPW).c_str());
    red_music();
  } else if (detectColourdiff() == "wht") {
    sendMessage(detectColourdiff());
    sendMessage(String(redPW).c_str());
    sendMessage(String(bluePW).c_str());
    sendMessage(String(greenPW).c_str());
  } else if (detectColourdiff() == "grn") {
    sendMessage(detectColourdiff());
    sendMessage(String(redPW).c_str());
    sendMessage(String(bluePW).c_str());
    sendMessage(String(greenPW).c_str());
    green_music();
  }
  detected = false;
  break;
```

```
const char *detectColourdiff() {
  //dprintf("detecting colour\n");
  if (redPW <= 170) {
    return "wht";
  } else if (redPW >= bluePW) {
    return "grn";
  } else if (redPW <= bluePW) {
    return "red";
  }
}
```

Appendix C. Speaker Module Code

```
// Alex Speaker
#define NOTE_C4 262
#define NOTE_D4 294
#define NOTE_E4 330
#define NOTE_F4 349
#define NOTE_G4 392
#define NOTE_A4 440
#define NOTE_B4 494
#define NOTE_C5 523
#define NOTE_G3 196
#define SPEAKER_PIN 44

int red_melody[] = {
  NOTE_C4, NOTE_D4, NOTE_E4, NOTE_F4, NOTE_G4, NOTE_A4, NOTE_B4, NOTE_
  NOTE_B4, NOTE_A4, NOTE_G4, NOTE_F4, NOTE_E4, NOTE_D4
};

int green_melody[] = {
  NOTE_E4, NOTE_E4, NOTE_E4, NOTE_C4, NOTE_E4, NOTE_G4, NOTE_G3
};

int red_noteDurations[] = {
  4, 4, 4, 4, 4, 4, 4, 4,
  4, 4, 4, 4, 4, 4
};

int green_noteDurations[] = {
  8, 4, 4, 8, 4, 2, 2
};

void red_music() {
  for (int thisNote = 0; thisNote < 14; thisNote++) {
    int red_noteDuration = 1000 / red_noteDurations[thisNote];
    tone(SPEAKER_PIN, red_melody[thisNote], red_noteDuration);
    int pauseBetweenNotes = red_noteDuration * 1.30;
    delay(pauseBetweenNotes);
    // stop the tone playing:
    noTone(SPEAKER_PIN);
  }
}

void green_music() {
  for (int thisNote = 0; thisNote < 7; thisNote++) {
    int green_noteDuration = 800 / green_noteDurations[thisNote];
    tone(SPEAKER_PIN, green_melody[thisNote], green_noteDuration);
    int pauseBetweenNotes = green_noteDuration * 1.30;
    delay(pauseBetweenNotes);
    // stop the tone playing:
    noTone(SPEAKER_PIN);
  }
}
```

```
if (detectColourdiff() == "red") {
  sendMessage(detectColourdiff());
  sendMessage(String(redPW).c_str());
  sendMessage(String(bluePW).c_str());
  sendMessage(String(greenPW).c_str());
  red_music();
} else if (detectColourdiff() == "wht") {
  sendMessage(detectColourdiff());
  sendMessage(String(redPW).c_str());
  sendMessage(String(bluePW).c_str());
  sendMessage(String(greenPW).c_str());
} else if (detectColourdiff() == "grn") {
  sendMessage(detectColourdiff());
  sendMessage(String(redPW).c_str());
  sendMessage(String(bluePW).c_str());
  sendMessage(String(greenPW).c_str());
  green_music();
}
detected = false;
break;
```