CG2111A Engineering Principles and Practice

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B04-1B

Name	Student #	Main Role
Amir Husaini Bin Musa	**A0271792W**	Hardware
Nguyen Trung Hieu	**A0277107A**	Software
Seah Tze Yang, Kendrick	**A0282805B**	Hardware
Shen Jiaming	**A0282057A**	Firmware

Section 1 Introduction

In this project, our team is developing a robot called Alex for search and rescue operations in a simulated disaster environment. Alex will be controlled remotely from a laptop, allowing the user to guide it through obstacles and locate victims efficiently. Our main focus is on enabling Alex to map its surroundings in real-time using LiDar, providing the user with information for manual navigation, which is essential for optimising rescue efforts and minimising time taken.

In addition to mapping, we are adding two functionalities to Alex. First, Alex will autonomously identify objects of interest, including victims (red-coloured paper) and non-victim (green-coloured paper) objects using a colour sensor. This feature will help distinguish between victims and other obstacles, improving rescue efficiency. Second, we will add ultrasonic sensors to prevent Alex from crashing into obstacles. Our team will continue to explore creative ways to make Alex more effective and versatile in search and rescue scenarios.

Section 2 Review of State of the Art

The RAPOSA robot is designed for tele-operated detection of potential survivors in outdoor environments hostile to human presence. Its mechanical structure consists of a main body and a front body with tracked wheels. The front body features sensors - including thermal cameras, temperature, and humidity sensors. The robot utilises wireless communications, facilitating docking and undocking remotely by the operator with the help of a camera inside the robot.

Strengths of RAPOSA:
- Rapid inspection of hazardous environments.
- Significant reduction in inspection time compared to specialised firefighter teams.

Weaknesses of RAPOSA:
- Wireless communication issues related to antennae location on the robot body.
- Interference with other wireless networks.

Fig. 1: RAPOSA robot

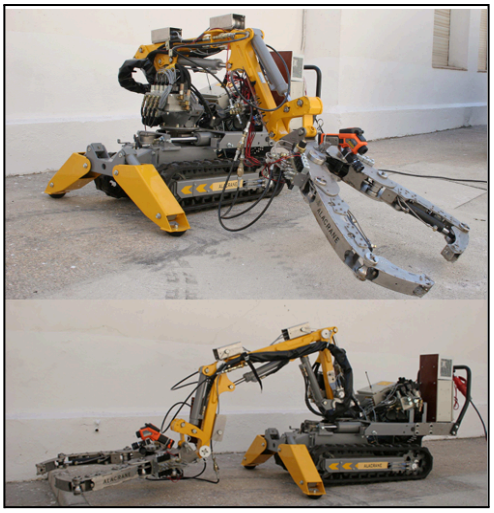


Fig. 2: ALACRANE robot

The ALACRANE robot is a fully hydraulic system developed from a modified small demolition machine, featuring a mobile base, main arm, and LR-Arms dual manipulator. The mobile base utilises tracked skid-steer traction controlled by hydraulic motors. The main arm boasts a payload of 120kg when fully extended. Electronic systems include three onboard PC computers for low-level control, and high-level perception, with sensors such as CCD cameras, thermal cameras, and a 3D laser scanner aiding navigation and target detection.

Strengths of ALACRANE:

- Capable of carrying heavy rubble.
- Equipped with CCD and IR cameras, a 3D laser scanner, and force/torque sensors for comprehensive environment perception and victim detection capabilities.

Weaknesses of ALACRANE:

- Complex system design and control architecture.
- Requires thorough understanding and expertise for operation and maintenance.

Section 3 System Architecture

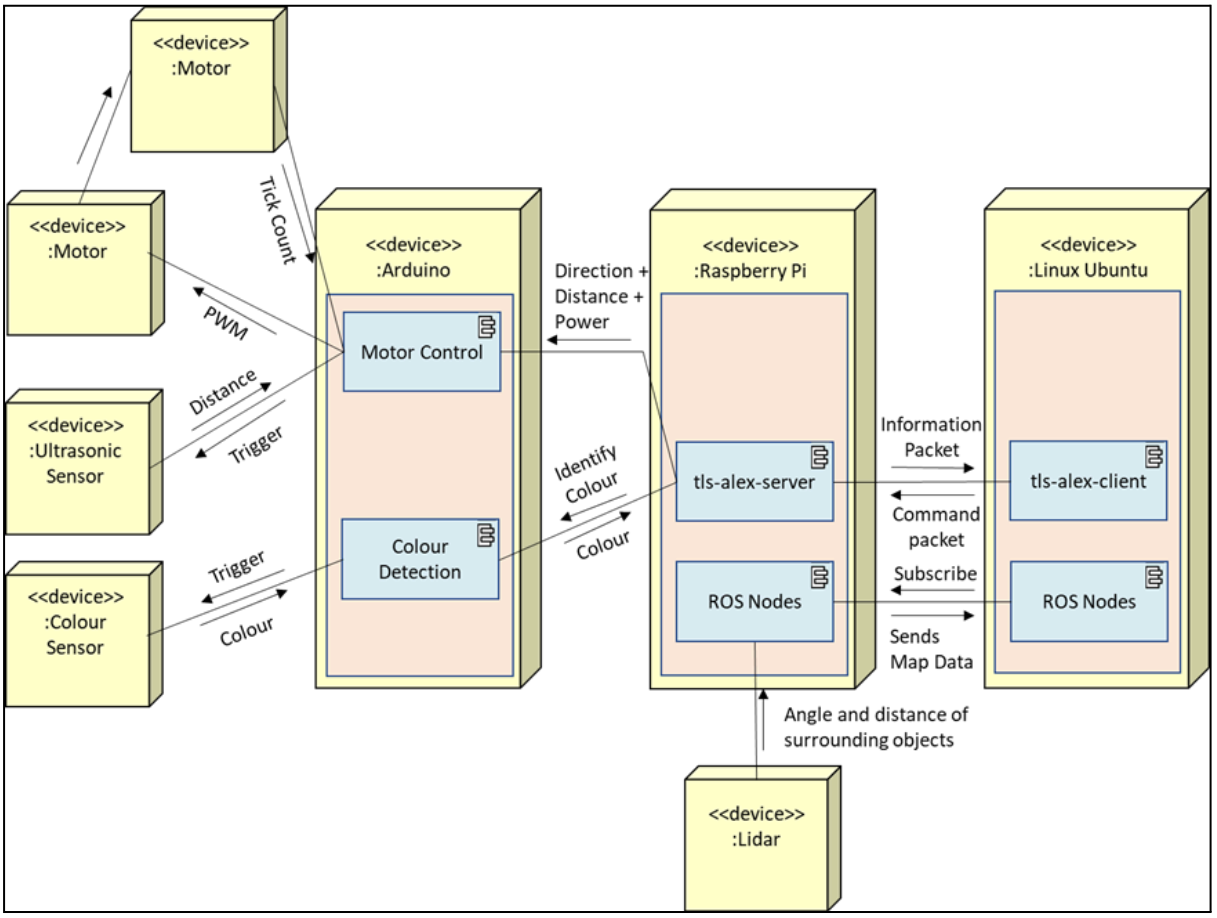


Fig. 3: System Architecture Diagram

Hardware Components: Raspberry Pi, Arduino Uno, RPLIDAR, 2 Motors, 2 Wheel Encoders, UltraSonic Sensor, and Colour Sensor

The RPi and Arduino/LiDar are connected to each other through the UART port.

UART packets are sent from the RPi to the Arduino Uno containing commands to be executed by the Arduino Uno to control Alex's movements and in return the RPi receives information from Arduino Uno on Tick Statuses, Object's Colour and Distance from Obstacles. The wheel encoders, motors, colour sensors, and ultrasonic sensors are connected physically using wires to the Arduino Uno's pins, which allows for control of the physical components of Alex.

The laptop is the master machine, while the RPI is the slave machine on ROS. Data from RPLIDAR will be processed by the Hector SLAM Algorithm. The environment map will be shown on RVIZ on the laptop. The operator uses this map to control the movement of Alex.

The RPi and the laptop communicate through TCP/IP packets. Commands are sent over from the laptop to RPi using TLS. The commands will then be converted from TCP/IP to UART packets to be transferred over to the Arduino Uno for execution, and vice versa for the packets sent from Arduino.

Section 4 Hardware Design

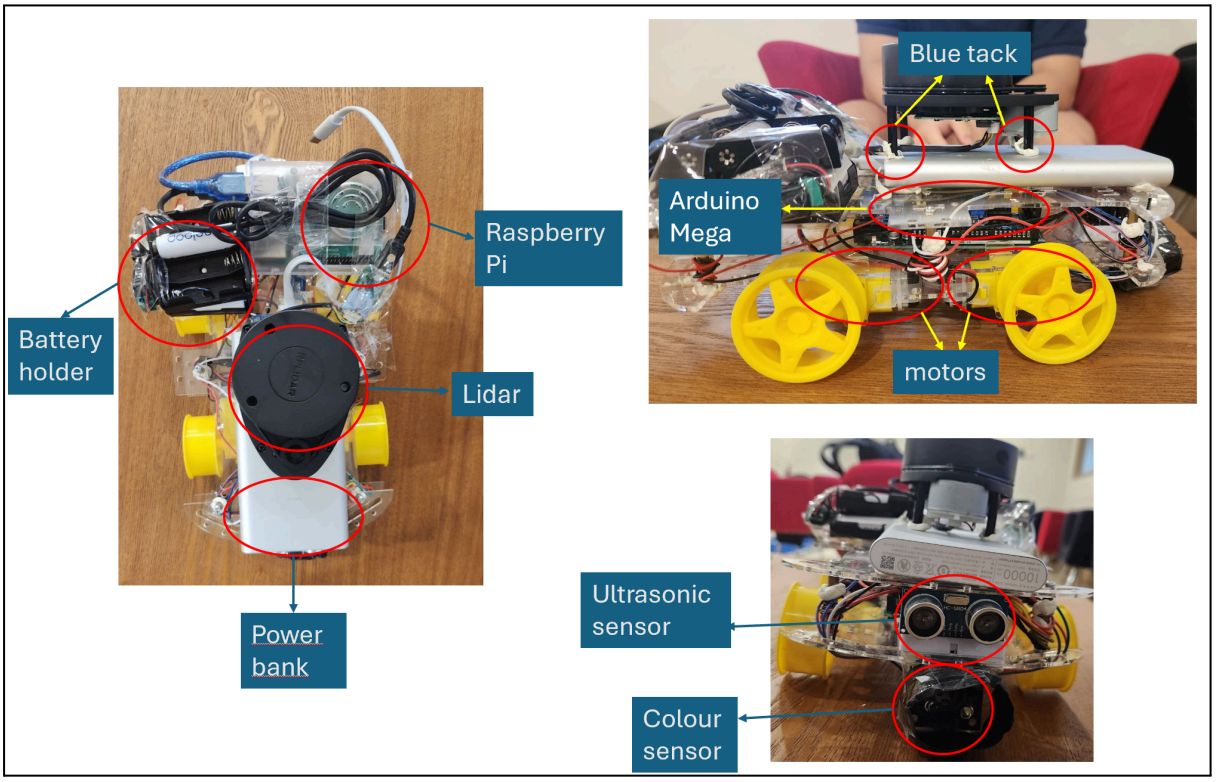


Fig. 4: Photograph of the final form of Alex

Alex is a tele-operated robot that consists of 2 chassis secured on top of one another: a top chassis and a bottom chassis. The arrangement for the top chassis is such that the LiDar is secured on top of the power bank; and behind them are the Raspberry Pi and 9V battery holder (that can hold six 1.5V batteries) placed side by side. This arrangement is so that the LiDar is the highest component in the robot's arrangement and can hence map Alex's surroundings without interference from the other components on the top chassis.

Below, the bottom chassis is arranged such that the Arduino Mega is in the centre so that it can be connected to the 4 motors attached below the bottom chassis, and to the Raspberry Pi on the top chassis. The 4 motors have wheels attached, with all the rubber tyres taken off them to reduce friction, in order for Alex to turn easily. Further, the front 2 motors have wheel encoders attached to them. Other than that, there are 2 ultrasonic sensors (each attached to a breadboard) at the front and back of the bottom chassis. Besides the 4 motors below the bottom chassis, we also placed a colour sensor right under the front ultrasonic sensor, with a black shielding made of construction paper.

For the non-standard hardware components our team used for Alex, the 9V battery holder was modified using soldering to accommodate 6 (instead of 8) 1.5V batteries, which allowed us not to exceed the 9V limit stated in the project requirements. Additionally, we used tape and blue tack to secure and organise the wires and unscrewed components such as the LiDar and Raspberry Pi within Alex. Further, we borrowed two ultrasonic sensors from the DSA lab and placed them on the bottom chassis to prevent collisions when Alex is moving forward and backward. Lastly, the colour sensor had a black shielding made of construction paper to prevent external light from affecting the colour sensing algorithm.

Section 5 Firmware Design

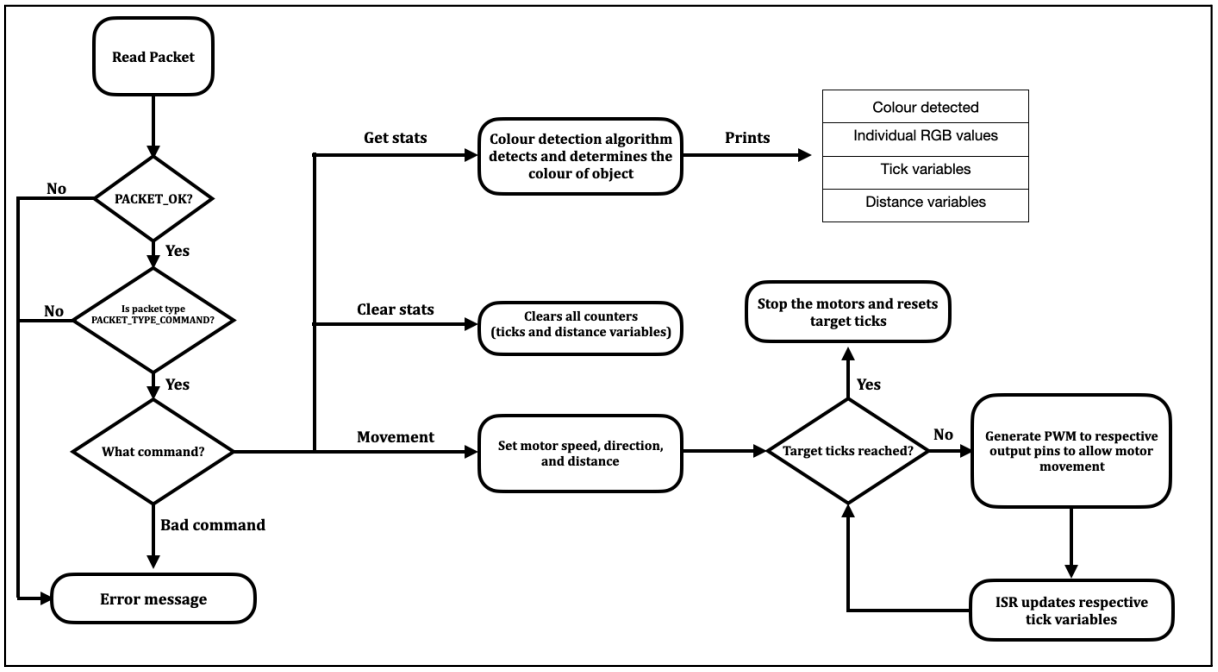


Fig. 5: Firmware Design

Communication Protocol

The communication between the Arduino and RPi will utilise UART protocols, operating at a baud rate of 9600 and following an 8N1 framework. The initialization of serial communication is handled at the beginning of the setup() section through the setupSerial() and startSerial() functions. Data reading and writing tasks are managed by the readSerial and writeSerial functions, respectively, both employing a polling method.

Packet Types

TPacket

Type of data	No. bytes	Contents
Packet type	1	PACKET_TYPE_COMMAND
Command	1	COMMAND_FORWARD COMMAND_REVERSE COMMAND_TURN_LEFT COMMAND_TURN_RIGHT COMMAND_STOP COMMAND_GET_STATS COMMAND_CLEAR_STATS
Padding	2	-
String data	32	Stores string data
Parameters	64	Stores parameters

Colour sensor and Ultrasonic sensor algorithm

Upon identifying a potential victim, we would operate the robot to align with the target and move forward. The ultrasonic sensor plays a key role in this process, halting the robot precisely at a distance of 7 cm from the victim. This distance has been determined through practical testing to yield consistent and reliable results in our colour detection algorithm.

At this designated distance, we will trigger the 'g' shortcut, which activates the colour sensor to capture the RGB values of the object in front of it. These RGB values are then mapped to a range of 0 to 255, ensuring standardised data processing.

The algorithm utilises this colour data to determine and display the detected colour, providing valuable information about the status of the victim. Additionally, to enhance the reliability of our colour detection system, the algorithm also prints out the individual RGB values detected by the colour sensor so as to verify and ascertain the colour ourselves.

Movement and Distance

To facilitate quicker movement and speed up the mapping process, we implemented additional movement controls that better suit the situation that Alex was in.

Shortcut	Direction	Distance/Angle	Power
w/W	forward	10 cm	70
s/S	backward	10 cm	70
a/A	left	90 °	70
d/D	right	90 °	70
j/J	left	input	70
l/L	right	input	70

Instead of manually inputting the distance/angle and power to achieve a specific movement, we implemented new commands to cut down on time spent. We create new packets with pre-assigned values for distance/angle and power that correspond to their shortcut inputs. For cases where smaller adjustments were called for, we could still manually input the smaller turns via the ‘j’ / ’J’ and ‘l’ / ’L’ commands. Note that forward and backward do not require customisable input parameters due to the ultrasonic sensor preventing any collisions in those two directions.

Section 6 Software Design

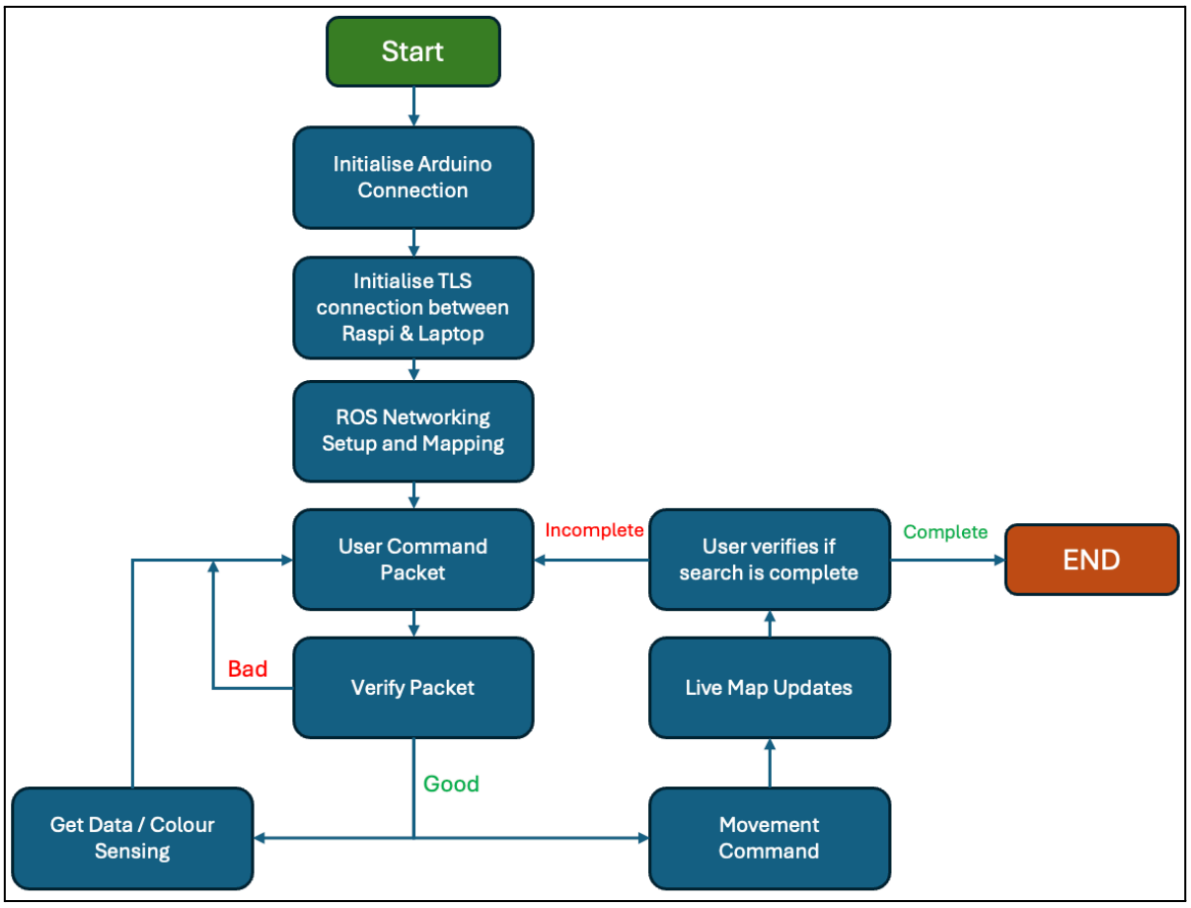


Fig. 6: Teleoperation Architecture

1. Initialise Arduino Connection to RPi

- Run tls-alex-server on RPi
- tls-alex-server will initialise Serial Connection between RPi and Arduino at specified port up to 5 attempts. Once connected, the program waits 3 seconds for Arduino to reboot.
- After creating the TLS server, RPi sends a Hello Packet to the Arduino. If the Hello Packet is successfully received by Arduino, Arduino will send back to RPi a PACKET_OK packet to indicate the successful connection.

```
pi@cg2111a:~/keys2send/tls-server-lib $ ./tls-alex-server

ALEX REMOTE SUBSYSTEM

Opening Serial Port
ATTEMPTING TO CONNECT TO SERIAL. ATTEMPT # 1 of 5.
Done. Waiting 3 seconds for Arduino to reboot
DONE. Starting Serial Listener
Starting Alex Server

** Spawning TLS Server **
```

Fig. 7: Arduino connecting to the RPi

2. Initialise TLS connection between RPi and The Laptop

- Server Role: The RPi runs the tls-alex-server program, establishing it as a TLS server.
- Client Initiation: The laptop runs the tls-alex-client program, initiating a TLS connection request to the RPi server.
- Certificate-Based Authentication: During the TLS handshake, the laptop (client) presents its certificate (laptop.crt), signed by a mutually trusted Certificate Authority (signing.pem), to the RPi (server) for verification. Once verified, the TLS connection is secured, and the laptop is considered authenticated.

3. ROS Networking and Mapping

- ROS Master: Launch the ROS Master on the RPi. This is the central hub for communication.
- RPLidar Publisher Node:
    - Create a ROS publisher node to handle RPLidar data acquisition.
    - Ensure it publishes the Lidar readings to a topic named /lidar/scan
- Hector SLAM Subscriber Node:
    - Create a ROS subscriber node configured to listen to the /lidar/scan topic.
    - Make sure this node implements the Hector SLAM algorithm for mapping.
    - This node will publish map data on a topic /map and pose information on a topic /pose.
- RViz Visualization:
    - Launch RViz and configure it to subscribe to:
        - The /map topic (for the generated map data).
        - The /pose topic (to display the LiDAR/robot's position within the map).

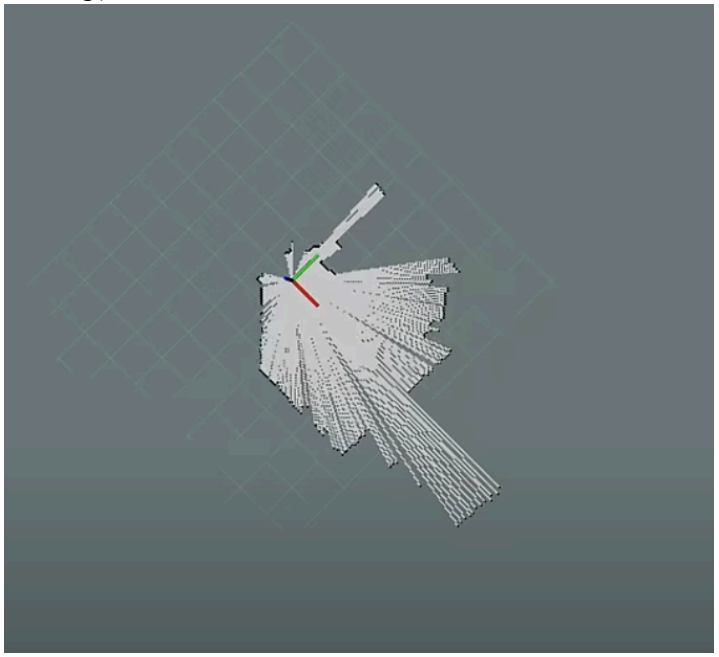


Fig. 8: Rviz Map

- Data Flow:
    - The ROS Master routes the LiDAR data from the publisher node to both Hector SLAM and RViz subscriber nodes.
    - Hector SLAM processes the data, generates a map, and publishes it along with pose information.
    - RViz receives and visualises the map, displaying the environment sensed by the LiDAR and the LiDAR/robot's position.

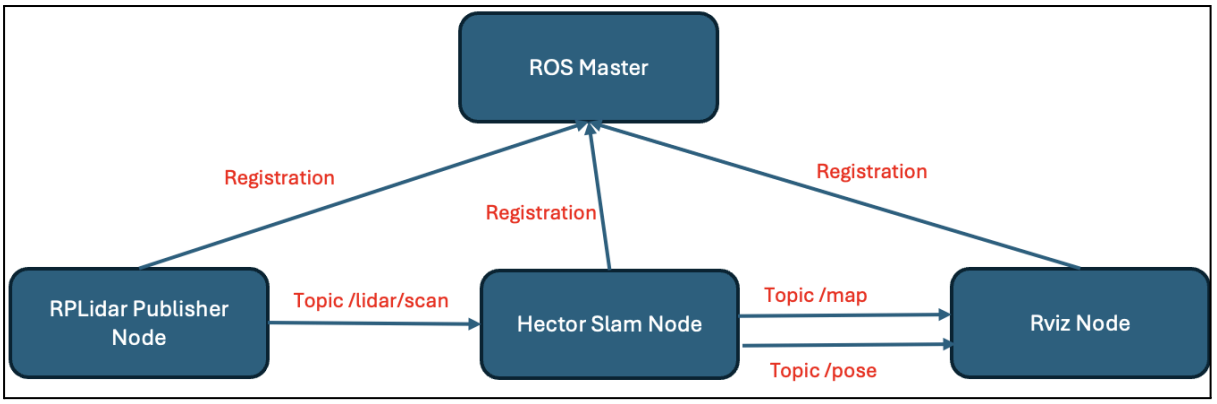


Fig. 9: ROS Networking

4. User Command & Packet Verification

- User Command Initiation
  - The user executes the tls-alex-client program on their laptop.
  - Using this client, the user generates a command packet containing either color sensing and get stats gathering instructions, or movement instructions.
  - This command packet is secured via TLS encryption and transmitted to the RPi.
- Command Relay and Verification (RPi)
  - The RPi, running the tls-alex-server program, receives and decrypts the TLS-secured command packet.
  - The RPi establishes a serial (UART) connection with the Arduino and forwards the command packet.
  - The RPi awaits a response packet from the Arduino.
- Arduino Response
  - Success: If the Arduino receives the command packet correctly, it executes the specified instructions. It then transmits a PACKET_OK response to the RPi.
  - Failure: If the Arduino encounters errors in receiving or parsing the command packet, it transmits a PACKET_BAD response to the RPi, indicating a failure to execute.
- Communication Back to User (RPi + Laptop)
  - The RPi relays the Arduino's response packet (either PACKET_OK or PACKET_BAD) back to the laptop client via the secure TLS connection.
  - The tls-alex-client on the laptop displays the outcome of the command execution to the user.

Fig. 10: Example for good and bad packets

5. Teleoperation Completion

- After the search operation concludes, the user terminates the tls-alex-client program, ending the teleoperation session and the TLS connection.

Section 7 Lessons Learnt - Conclusion

The lab sessions on TCP/IP communication setup were extremely useful in helping us be aware of the different things to consider like how to establish secure communications between the Raspberry Pi and Arduino, endianness, and how all these factors would influence the behaviour of our robot.

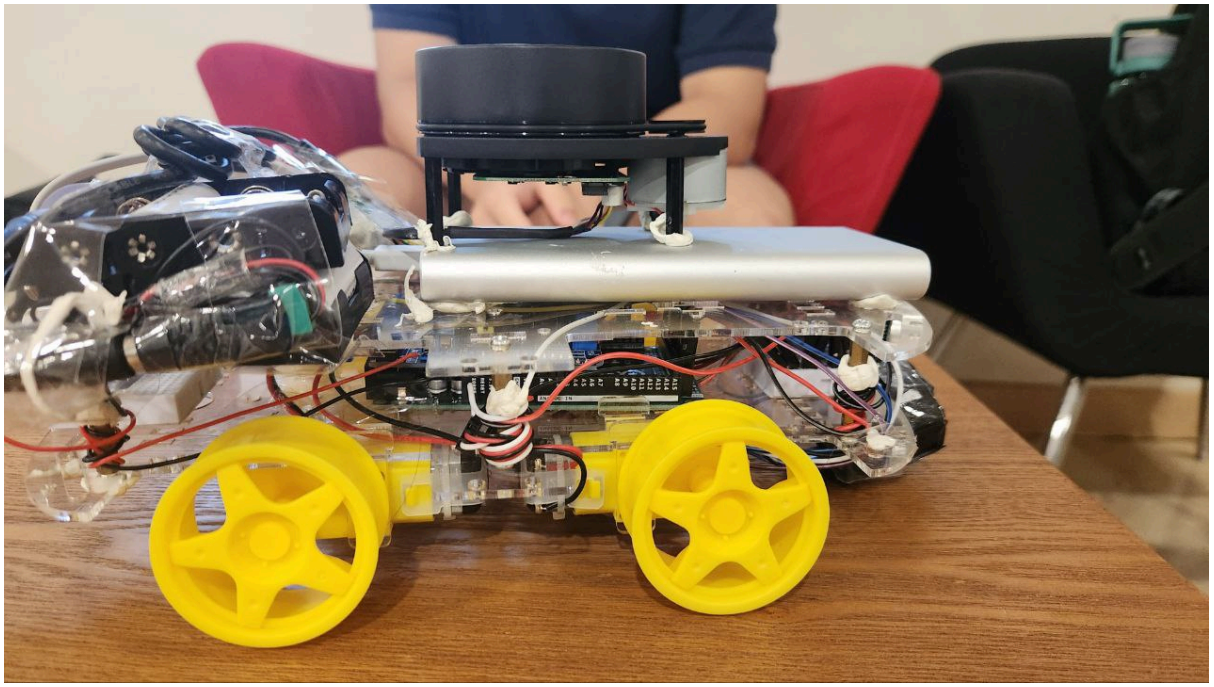
Learning how to use the LIDAR, ROS and Rviz mapping well was also critical to our project in helping us navigate through the maze even when we were not physically there in the maze. In particular, we learnt the importance of working well with all the different aspects of our robot, and striking a balance between different functionalities to allow our robot to function optimally as per what was required in the project. For instance, we initially wanted our turning speed to be higher so that we can navigate through the maze within a shorter time. However, we realised turning above a certain threshold speed would cause our LIDAR to go haywire and mess up our mapping of the maze. Hence, we settled for a lower turning speed so as to achieve a more accurate mapping, which definitely served us well when navigating unknown obstacles.

We made the mistake of putting code in the void loop() in our Alex.ino code. The void loop() was constantly checking for packets and calculating the delta distance. As such, when we put other code into the loop, it frequently disrupted the serial communications and would lead to a “bad magic number” error.

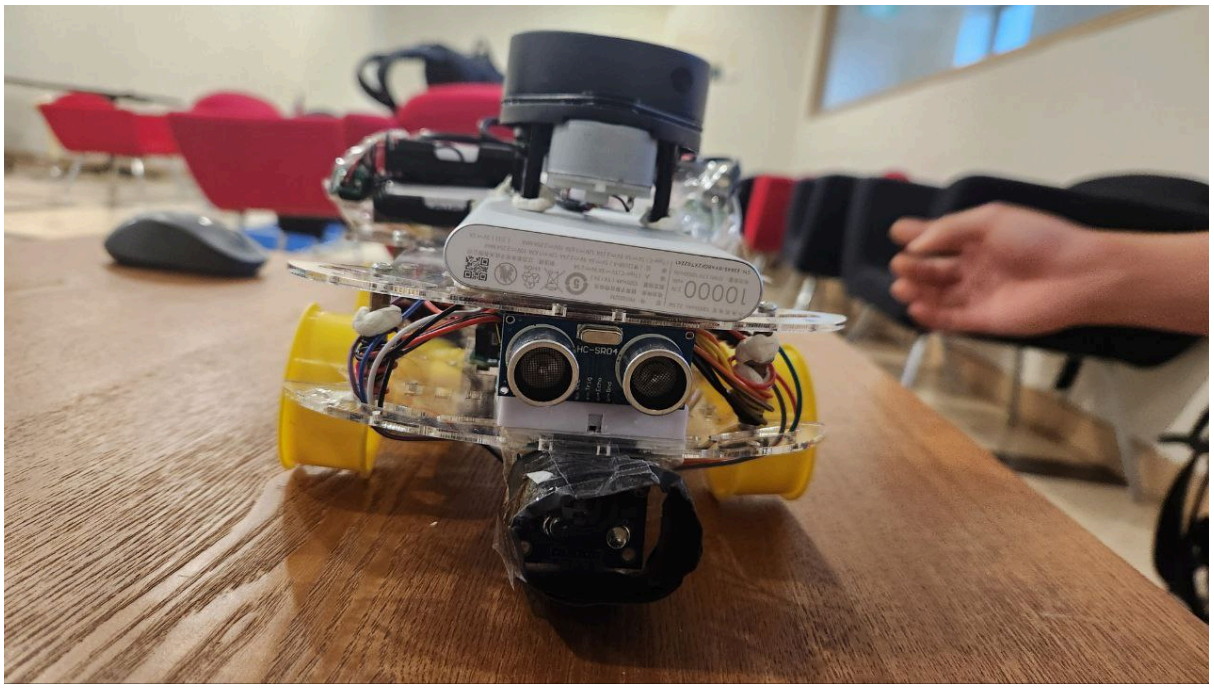
Even though we were unable to get a reliable and insightful mapping during the graded run, we did not restart the ROS and Rviz mapping as we felt that doing so would take away precious time during the run. As such, all we could do was try our best to make a good guess on the different obstacles in the run. Looking back, this was wasteful of us as we did not manage to make full use of our mapping system despite spending a lot of time on it beforehand. In our practice runs, our mapping system was serving us well and we were confident that we configured it well. Even if restarting the ROS and Rviz mapping took up a significant amount of time, it would have at least enabled us to make the best out of the situation during the graded run by trying to score well in the accuracy of our hand-drawn map. Due to us letting the pressure of the time constraint get to us, we were unable to detect any object and collided into the wall 4 times.

Appendix

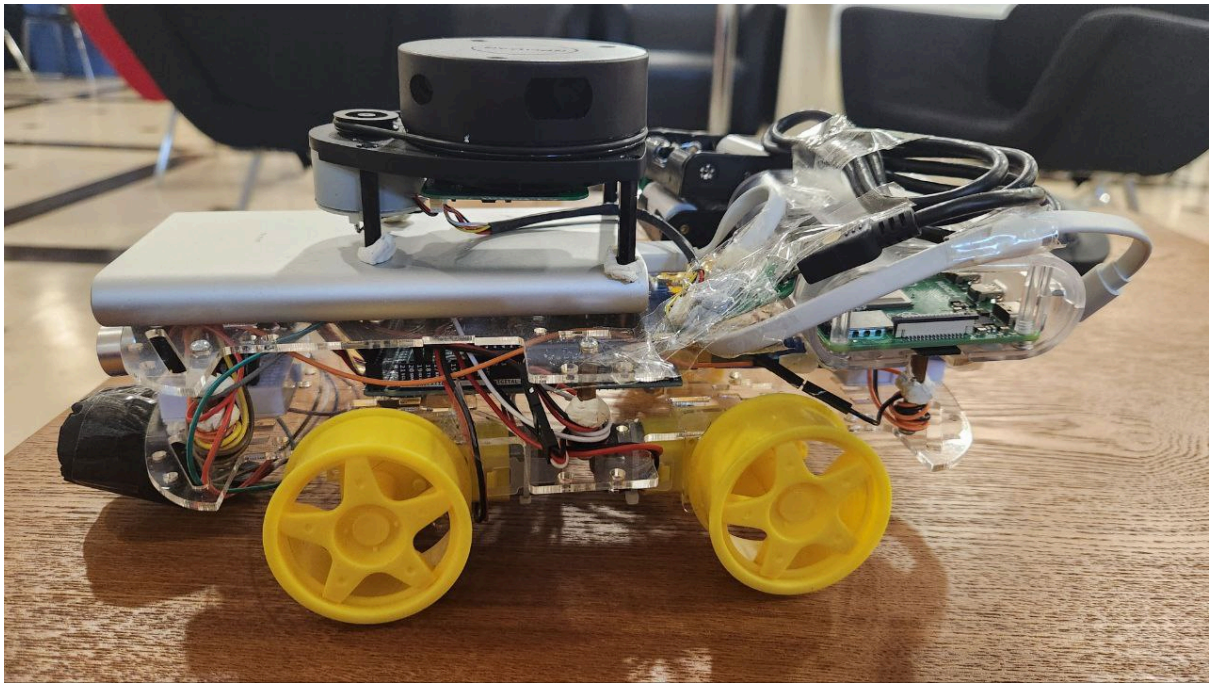
Photos of Alex



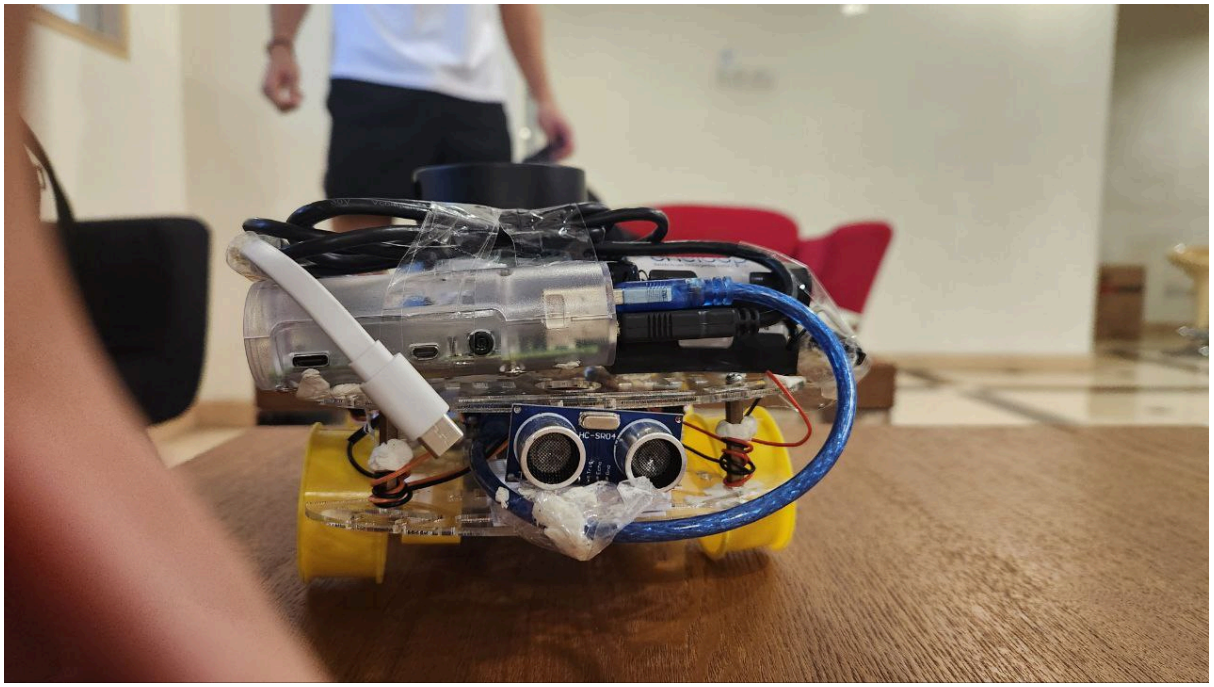
Right side of Alex



Front of Alex



Left side of Alex



Back of Alex

Top View of Alex

Code from Alex.ino

```
#include <stdarg.h>
#include <math.h>
#include <serialize.h>
#include "packet.h"
#include "constants.h"

#define PI            3.141592654
#define ALEX_LENGTH   16
#define ALEX_BREADTH  6
//#define PACKET_SIZE 32

float alexDiagonal = 0.0;
float alexCirc = 0.0;

volatile TDirection dir;

/*
 * Alex's configuration constants
 */

// Number of ticks per revolution from the
// wheel encoder.

#define COUNTS_PER_REV      4

// Wheel circumference in cm.
// We will use this to calculate forward/backward distance traveled
// by taking revs * WHEEL_CIRC

#define WHEEL_CIRC          20.4

/*
 *    Alex's State Variables
 */

// Store the ticks from Alex's left and
// right encoders.
volatile unsigned long leftForwardTicks;
volatile unsigned long rightForwardTicks;
volatile unsigned long leftReverseTicks;
volatile unsigned long rightReverseTicks;

//left and right encoder ticks for turning
volatile unsigned long leftForwardTicksTurns;
volatile unsigned long rightForwardTicksTurns;
volatile unsigned long leftReverseTicksTurns;
volatile unsigned long rightReverseTicksTurns;

// Store the revolutions on Alex's left
// and right wheels
volatile unsigned long leftRevs;
volatile unsigned long rightRevs;

// Forward and backward distance traveled
volatile unsigned long forwardDist;
volatile unsigned long reverseDist;
```

```
unsigned long deltaDist;
unsigned long newDist;

unsigned long deltaTicks;
unsigned long targetTicks;

// TCS230 or TCS3200 pins wiring to Arduino
#define S0 47
#define S1 45
#define S2 43
#define S3 41
#define sensorOut 39
#define RED 0
#define GREEN 1
#define WHITE 2

// Stores frequency read by the photodiodes
int redFrequency = 0;
int greenFrequency = 0;
int blueFrequency = 0;

// Stores the red. green and blue RGB values
int rgb[3] = {0};

// final colour.
static unsigned long colourDetected = 0;

const int TRIGFRONT = 25;
const int ECHOFRONT = 23;
const int TRIGBACK = 27;
const int ECHOBACK = 29;
#define SPEED_OF_SOUND 0.0345

double getUltra(const int TRIGPIN, const int ECHOPIN) {
  digitalWrite(TRIGPIN, HIGH);
  delayMicroseconds(10);
  digitalWrite(TRIGPIN, LOW);
  int microsecs = pulseIn(ECHOPIN,HIGH);
  double cms = microsecs*SPEED_OF_SOUND/2;

  return cms;
}

void colourDetect () {
    // Setting RED (R) filtered photodiodes to be read
  digitalWrite(S2,LOW);
  digitalWrite(S3,LOW);

  // Reading the output frequency
  redFrequency = pulseIn(sensorOut, LOW);
  // Remaping the value of the RED (R) frequency from 0 to 255
  rgb[0] = map(redFrequency, 3350, 2050, 0, 255);
  delay(100);

  // Setting GREEN (G) filtered photodiodes to be read
  digitalWrite(S2,HIGH);
  digitalWrite(S3,HIGH);

  // Reading the output frequency
  greenFrequency = pulseIn(sensorOut, LOW);
```

```
  // Remaping the value of the GREEN (G) frequency from 0 to 255. (low to high freq first)
  rgb[1] = map(greenFrequency, 4600, 3500, 0, 255);
  delay(100);

  // Setting BLUE (B) filtered photodiodes to be read
  digitalWrite(S2,LOW);
  digitalWrite(S3,HIGH);

  // Reading the output frequency
  blueFrequency = pulseIn(sensorOut, LOW);

  // Remaping the value of the BLUE (B) frequency from 0 to 255
  rgb[2] = map(blueFrequency, 4300, 2300, 0, 255);
  delay(100);


  // Checks the current detected colour. red = 0, green = 1, white = 2.
  if(rgb[1] > rgb[0] && rgb[1] > rgb[2]) {
    colourDetected = GREEN;
  }
  else if(rgb[0] > rgb[1] && rgb[0] > rgb[2]){
    colourDetected = RED;
  }
  else {
    colourDetected =  WHITE;
  }
}

unsigned long computeDeltaTicks(float ang) {
    unsigned long ticks = (unsigned long)((ang * alexCirc * COUNTS_PER_REV) / (360.0 *
WHEEL_CIRC));
  return ticks;
}
void left(float ang, float speed) {

  if (ang == 0)
    deltaTicks=99999999;

  else
    deltaTicks=computeDeltaTicks(ang);

  targetTicks = leftReverseTicksTurns + deltaTicks;
  //dbprintf("Initial Left: %d\n", leftReverseTicksTurns);
  //dbprintf("deltaTicks: %d \n", deltaTicks);
  cw(ang, speed);
}

void right(float ang, float speed) {
  if (ang == 0)
    deltaTicks=99999999;

  else
    deltaTicks=computeDeltaTicks(ang);

  targetTicks = rightReverseTicksTurns + deltaTicks;
  //dbprintf("Initial Right: %d \n", rightReverseTicksTurns);
  //dbprintf("deltaTicks: %d \n", deltaTicks);
  ccw(ang, speed);
}
```

```
/*
 *
 * Alex Communication Routines.
 *
 */

TResult readPacket(TPacket *packet)
{
    // Reads in data from the serial port and
    // deserializes it.Returns deserialized
    // data in "packet".

    char buffer[PACKET_SIZE];
    int len;

    len = readSerial(buffer);

    if(len == 0)
      return PACKET_INCOMPLETE;
    else
      return deserialize(buffer, len, packet);
}

void sendStatus()
{
  // Implement code to send back a packet containing key
  // information like leftTicks, rightTicks, leftRevs, rightRevs
  // forwardDist and reverseDist
  // Use the params array to store this information, and set the
  // packetType and command files accordingly, then use sendResponse
  // to send out the packet. See sendMessage on how to use sendResponse.
  //
  TPacket statusPacket;
  statusPacket.packetType = PACKET_TYPE_RESPONSE;
  statusPacket.command = RESP_STATUS;

  statusPacket.params[0] = colourDetected;
  statusPacket.params[1] = rgb[0];
  statusPacket.params[2] = rgb[1];
  statusPacket.params[3] = rgb[2];
  statusPacket.params[4] = leftForwardTicksTurns;
  statusPacket.params[5] = rightForwardTicksTurns;
  statusPacket.params[6] = leftReverseTicksTurns;
  statusPacket.params[7] = rightReverseTicksTurns;
  statusPacket.params[8] = forwardDist;
  statusPacket.params[9] = reverseDist;
  //statusPacket.params[10] = colourDetected;
  sendResponse(&statusPacket);
}

void sendMessage(const char *message)
{
  // Sends text messages back to the Pi. Useful
  // for debugging.

  TPacket messagePacket;
  messagePacket.packetType=PACKET_TYPE_MESSAGE;
  strncpy(messagePacket.data, message, MAX_STR_LEN);
```

```
  sendResponse(&messagePacket);
}

void dbprintf(char *format, ...) {
     va_list args;
     char buffer[128];
     va_start(args, format);
     vsprintf(buffer, format, args);
     sendMessage(buffer);
}

void sendBadPacket()
{
  // Tell the Pi that it sent us a packet with a bad
  // magic number.

  TPacket badPacket;
  badPacket.packetType = PACKET_TYPE_ERROR;
  badPacket.command = RESP_BAD_PACKET;
  sendResponse(&badPacket);
}

void sendBadChecksum()
{
  // Tell the Pi that it sent us a packet with a bad
  // checksum.

  TPacket badChecksum;
  badChecksum.packetType = PACKET_TYPE_ERROR;
  badChecksum.command = RESP_BAD_CHECKSUM;
  sendResponse(&badChecksum);
}

void sendBadCommand()
{
  // Tell the Pi that we don't understand its
  // command sent to us.

  TPacket badCommand;
  badCommand.packetType=PACKET_TYPE_ERROR;
  badCommand.command=RESP_BAD_COMMAND;
  sendResponse(&badCommand);
}

void sendBadResponse()
{
  TPacket badResponse;
  badResponse.packetType = PACKET_TYPE_ERROR;
  badResponse.command = RESP_BAD_RESPONSE;
  sendResponse(&badResponse);
}

void sendOK()
{
  TPacket okPacket;
  okPacket.packetType = PACKET_TYPE_RESPONSE;
  okPacket.command = RESP_OK;
  sendResponse(&okPacket);
}
```

```
void sendResponse(TPacket *packet)
{
  // Takes a packet, serializes it then sends it out
  // over the serial port.
  char buffer[PACKET_SIZE];
  int len;

  len = serialize(buffer, packet, sizeof(TPacket));
  writeSerial(buffer, len);
}



/*
 * Setup and start codes for external interrupts and
 * pullup resistors.
 *
 */
// Enable pull up resistors on pins 18 and 19
void enablePullups()
{
  // Use bare-metal to enable the pull-up resistors on pins
  // 19 and 18. These are pins PD2 and PD3 respectively.
  // We set bits 2 and 3 in DDRD to 0 to make them inputs.
  DDRD &= 0b11110011;
  PORTD |= 0b00001100;
}



// Functions to be called by INT2 and INT3 ISRs.
void leftISR()
{
  if (dir == 1) {
    leftForwardTicks ++;
    forwardDist = (unsigned long) ((float) leftForwardTicks / COUNTS_PER_REV * WHEEL_CIRC);
  }
  if (dir == 2) {
    leftReverseTicks ++;
    reverseDist = (unsigned long) ((float) leftReverseTicks / COUNTS_PER_REV * WHEEL_CIRC);
  }
  if (dir == 3) {
    leftReverseTicksTurns ++;
    //dbprintf("LTick %d \n", leftReverseTicksTurns);
  }
  if (dir == 4) {
    leftForwardTicksTurns ++;
  }
}

void rightISR()
{
  if (dir == 1) {
    rightForwardTicks ++;
  }
  if (dir == 2) {
    rightReverseTicks ++;
  }
  if (dir == 3) {
    rightForwardTicksTurns ++;
  }
```

```
  if (dir == 4) {
    rightReverseTicksTurns ++;
    //dbprintf("RTicks %d \n", rightReverseTicksTurns);
  }
}

// Set up the external interrupt pins INT2 and INT3
// for falling edge triggered. Use bare-metal.
void setupEINT()
{
  // Use bare-metal to configure pins 18 and 19 to be
  // falling edge triggered. Remember to enable
  // the INT2 and INT3 interrupts.
  // Hint: Check pages 110 and 111 in the ATmega2560 Datasheet.
  EIMSK = 0b00001100;
  EICRA = 0b10100000;
}

// Implement the external interrupt ISRs below.
// INT3 ISR should call leftISR while INT2 ISR
// should call rightISR.
ISR(INT3_vect)
{
  leftISR();
}

ISR(INT2_vect)
{
  rightISR();
}


// Implement INT2 and INT3 ISRs above.

/*
 * Setup and start codes for serial communications
 *
 */
// Set up the serial connection. For now we are using
// Arduino Wiring, you will replace this later
// with bare-metal code.
void setupSerial()
{
  // To replace later with bare-metal.
  Serial.begin(9600);
  // 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()
{
  // 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(Serial.available())
    buffer[count++] = Serial.read();

  return count;
}

// Write to the serial port. Replaced later with
// bare-metal code

void writeSerial(const char *buffer, int len)
{
  Serial.write(buffer, len);
  // Change Serial to Serial2/Serial3/Serial4 in later labs when using other UARTs
}

/*
 * Alex's setup and run codes
 *
 */

// Clears all our counters
void clearCounters()
{
  leftForwardTicks=0;
  rightForwardTicks=0;
  leftForwardTicksTurns=0;
  rightForwardTicksTurns=0;
  leftReverseTicks=0;
  rightReverseTicks=0;
  leftReverseTicksTurns=0;
  rightReverseTicksTurns=0;
  leftRevs=0;
  rightRevs=0;
  forwardDist=0;
  reverseDist=0;
}

// Clears one particular counter
void clearOneCounter(int which)
{
  clearCounters();
}
// Intialize Alex's internal states

void initializeState()
{
  clearCounters();
}

void handleCommand(TPacket *command)
```

```
{
  switch(command->command)
  {
    // For movement commands, param[0] = distance, param[1] = speed.
    case COMMAND_FORWARD:
        sendOK();
        forward((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_REVERSE:
        sendOK();
        backward((double) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_TURN_LEFT:
        sendOK();
        left((float) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_TURN_RIGHT:
        sendOK();
        right((float) command->params[0], (float) command->params[1]);
      break;
    case COMMAND_GET_STATS:
    {
        sendOK();
        colourDetect();
        sendStatus();
      break;
    }
    case COMMAND_CLEAR_STATS:
        sendOK();
        clearOneCounter(command->params[0]); //clear all counters / params
      break;
    case COMMAND_STOP:
        sendOK();
        stop();
    /*
     * Implement code for other commands here.
     *
     */

    default:
      sendBadCommand();
  }
}

void waitForHello()
{
  int exit=0;

  while(!exit)
  {
    TPacket hello;
    TResult result;

    do
    {
      result = readPacket(&hello);
    } while (result == PACKET_INCOMPLETE);

    if(result == PACKET_OK)
    {
```

```
      if(hello.packetType == PACKET_TYPE_HELLO)
      {


        sendOK();
        exit=1;
      }
      else
        sendBadResponse();
    }
    else
      if(result == PACKET_BAD)
      {
        sendBadPacket();
      }
      else
        if(result == PACKET_CHECKSUM_BAD)
          sendBadChecksum();
  } // !exit
}

void setup() {
  // put your setup code here, to run once:
  alexDiagonal = sqrt((ALEX_LENGTH * ALEX_LENGTH) + (ALEX_BREADTH * ALEX_BREADTH));
  alexCirc = PI * alexDiagonal;
  cli();
  setupEINT();
  setupSerial();
  startSerial();
  enablePullups();
  initializeState();
  sei();
  //set up ultrasonic sensors
  pinMode(TRIGFRONT, OUTPUT);
  pinMode(ECHOFRONT, INPUT);
  pinMode(TRIGBACK, OUTPUT);
  pinMode(ECHOBACK, INPUT);
  // set up colour sensor
  pinMode(S0, OUTPUT);
  pinMode(S1, OUTPUT);
  pinMode(S2, OUTPUT);
  pinMode(S3, OUTPUT);
  pinMode(sensorOut, INPUT);

  // Setting colour sensor frequency scaling to 20%
  digitalWrite(S0,HIGH);
  digitalWrite(S1,LOW);
}

void handlePacket(TPacket *packet)
{
  switch(packet->packetType)
  {
    case PACKET_TYPE_COMMAND:
      handleCommand(packet);
      break;

    case PACKET_TYPE_RESPONSE:
      break;
```

```
    case PACKET_TYPE_ERROR:
      break;

    case PACKET_TYPE_MESSAGE:
      break;

    case PACKET_TYPE_HELLO:
      break;
  }
}
void loop() {

  TPacket recvPacket; // This holds commands from the Pi

  TResult result = readPacket(&recvPacket);

  if(result == PACKET_OK)
    handlePacket(&recvPacket);
  else
    if(result == PACKET_BAD)
    {
      sendBadPacket();
    }
    else
      if(result == PACKET_CHECKSUM_BAD)
      {
        sendBadChecksum();
      }

  if (deltaDist > 0) {
    if (dir == 1) {
      if (forwardDist > newDist || getUltra(TRIGFRONT, ECHOFRONT) < 9.0) {
        deltaDist = 0;
        newDist = 0;
        stop();
        clearCounters();
      }
    }
    else {
      if (dir == 2) {
        //dbprintf("Backward Command Executing");
        if (reverseDist > newDist) {
          deltaDist = 0;
          newDist = 0;
          stop();
          clearCounters();
        }
      }
      else {}
    }
    if (dir == STOPPED) {
      deltaDist = 0;
      newDist = 0;
      stop();
    }
  }
  if (deltaTicks > 0) {
    if (dir == 3) {
      //dbprintf("Left Ticks: %d \n", leftReverseTicksTurns);
      //dbprintf("Left,Target: %d \n", targetTicks);
```

```
      if (leftReverseTicksTurns >= targetTicks) {
        deltaTicks = 0;
        targetTicks = 0;
        stop();
        clearCounters();
      }
    }
    else if (dir == 4) {
      //dbprintf("Right Ticks: %d \n", rightReverseTicksTurns);
      //dbprintf("Target: %d \n", targetTicks);
      //dbprintf("Target: %d \n", targetTicks);
      //dbprintf("Target: %d \n", targetTicks);

      if (rightReverseTicksTurns >= targetTicks) {
        deltaTicks = 0;
        targetTicks = 0;
        stop();
        clearCounters();
      }
    }
    else if (dir == 5) {
      deltaTicks = 0;
      targetTicks = 0;
      stop();
    }
  }

}
```

References

Marques, C., Cristóvão, J., Lima, P., Frazão, J., Ribeiro, I., & Ventura, R. (Year). RAPOSA: Semi-Autonomous Robot for Rescue Operations. IdMind-Engenharia de Sistemas, Lda, Pólo Tecnológico de Lisboa, Lote1, Estrada do Paço do Lumiar 1600-546 Lisboa, PORTUGAL. Institute for Systems and Robotics Instituto Superior Técnico, Av. Rovisco Pais, 1 1049-001 Lisboa, PORTUGAL. https://irsgroup.isr.tecnico.ulisboa.pt/raposa/

A. Garcia-Cerezo et al., "Development of ALACRANE: A Mobile Robotic Assistance for Exploration and Rescue Missions," 2007 IEEE International Workshop on Safety, Security and Rescue Robotics, Rome, Italy, 2007, pp. 1-6, doi: 10.1109/SSRR.2007.4381269. keywords: {Mobile robots;Vehicles;Manipulators;Wrist;Hydraulic actuators;Charge coupled devices;Robot vision systems;Charge-coupled image sensors;Cameras;Infrared detectors;Mobile robots;Search and rescue;Mobile manipulators}. https://ieeexplore.ieee.org/abstract/document/4381269