CG2111A Engineering Principle and Practice

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B02-3A

Name	Student #	Main Role
Anderson Lim Weiheng	A0269757E	Hardware
Cheng Tze Yong	A0272343J	Hardware
Han Zixuan Nancy	A0281489M	Firmware
Jensen Kuok Thai Hock	A0281767N	Hardware/Firmware
Lee Hao Zhe	A0272689J	Software

CONTENTS

Section 1 Introduction

We are tasked to build a robotic vehicle, Alex, with search and rescue functionalities. We have to navigate our way in a room filled with various objects. They consist of obstacles, “victims” which are of either red or green, as well as one other “dummy victims” of colour white, all of which need to be identified by Alex. This constitutes a similar environment in the aftermath of a disaster. There will be at least one clear path for Alex to navigate from the starting room to the last room. During the evaluation, we have to manually draw the environment mapped out by Alex, which will be submitted 1 minute after the end of our evaluation run. Evaluation stops after Alex has completely explored and mapped out the entire arena and finished parking in the parking area, or after the time limit of 6 minutes.

This “search and rescue” problem is successfully accomplished by the project group’s ability to fulfil three main objectives:

1. Demonstrating good knowledge with embedded platforms, especially in the areas of GPIO programming, Timers, UART and more.
2. Ensuring error-free communication between different platforms, allowing the smooth transmission and receiving of critical commands and sensory data to successfully complete the “search and rescue task”.
3. Perform localization and mapping using LiDAR to safely navigate Alex and complete its mission.

Section 2 Review of State of the Art

Our group has researched and summarised two search and rescue robot implementations, as listed below.

Platform 1: RAPOSA

RAPOSA was designed to navigate through collapsed buildings, and detect potential survivors, whose information is then transmitted to a human operator remotely. It contains hardware components such as : a thermal camera, two web cameras, humidity, gas and temperature sensors, microphone, light diodes and a loudspeaker.[1] RAPOSA operates on wireless communications, but tethered operations are also available.It runs on an agent-based software architecture, enabling information integration between sensors and collaboration between robots, all controlled via a GUI.[2] Strengths: RAPOSA is able to reach inaccessible disaster locations, climb up and down the stairs, traverse through sewage pipes and even travel upside down without any issues. It also reduces inspection time by 75%. Weaknesses: Wireless connections are weakened depending on antennas location and interference with other wireless networks. [3]

Platform 2: Robbie

Robbie is a search and rescue robot developed by UAS Technikum Wien, aiming to provide a precise area evaluation, essential for S&R operations. The hardware setup and specifications as well the GUI are shown in the diagrams below:

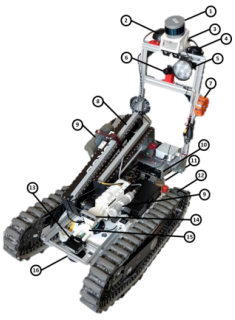


Fig. 1. Robbie hardware setup
(1) Velodyne PUCK-VLP16, (2) Wifi/ 4G Antenna, (3) Rear-Facing Intel Realsense D435, (4) Garmin GPS Module, (5) LED Headlight, (6) Ueye UI-3240LE Camera with Camera Mount, (7) Operation Indication Light, (8) Robotic Arm, (9) SSM1+ Radiatiometer with mounted Probe, (10) Embedded PC, (11) Rear-Facing Internal Camera and SICK TIM-551-2050001, (12) taurob Tracker, (13) Front-facing TIM-551-2050001, (14) Intel Realsense D435 mounted on EEF, (15) EEF, (16) Front-Facing Internal Camera and Internal Headlights

Diagram 2.1: Hardware specifications of Robbie

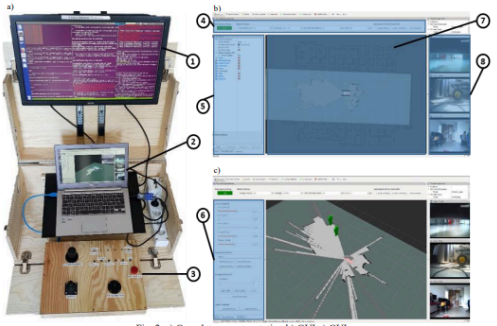


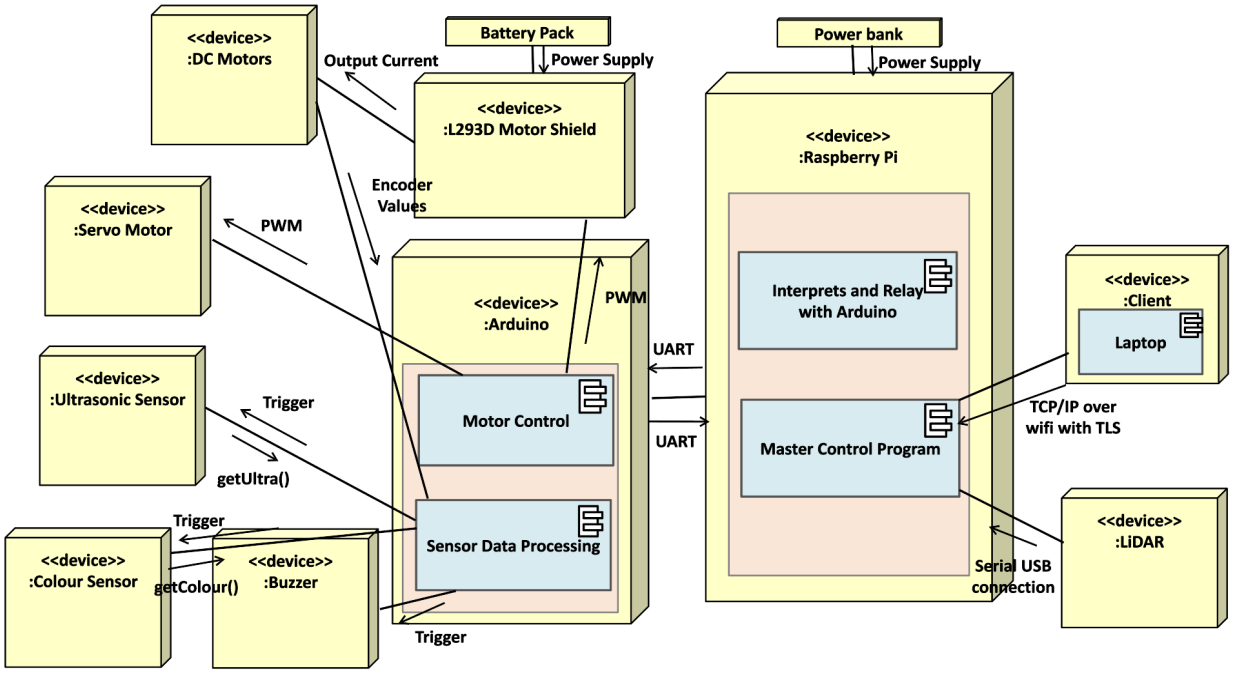
Fig. 2. a) Complete operator station b) GUI c) GUI
(1) Logscreen / Command input, (2) GUI depicted in more detail in b) and c), (3) Control panel for teleoperation, (4) Sensor readings and emergency of switch, (5) Topic visualisation checkbox, (6) Additional teleoperation toolbox, (7) Map visualisation toolbox, (8) Image stream from Robbie's POV

The control panel for teleoperation (Fig. 2 (3)) is used to operator Robbie. Here the steering of the base platform as well as the robot-arm is handled.

Diagram 2.2: Robbie GUI

In summary,teleoperation is based on 2 RGB cameras.A 3D LIDAR Velodyne PUCK VLP-16, two SICK TIM-551-2050001 and a sensor rig with Garmin GPS module is used for localisation. A Robot Operating System (ROS) API package based on Octomap (for map generation) and YOLO-ROS is developed, which detects objects in RGB images using convolutional neural networks (CNNs). Strengths: Robbie could be operated day and night from its 2 LED headlights. It provides 5 image streams as well as 2D and 3D maps. Only one user is required to operate Robbie. Weaknesses: Multiple hardware components such as the Garmin GPS Module and cameras attached to the sensor rig distorts it, making the robot unsteady. [4] YOLO-ROS software is also unstable as it may crash after detecting first images and a continuous message from darknet_ros package.[5][6]

Section 3 System Architecture



Section 4 Hardware Design

Final Form of Alex

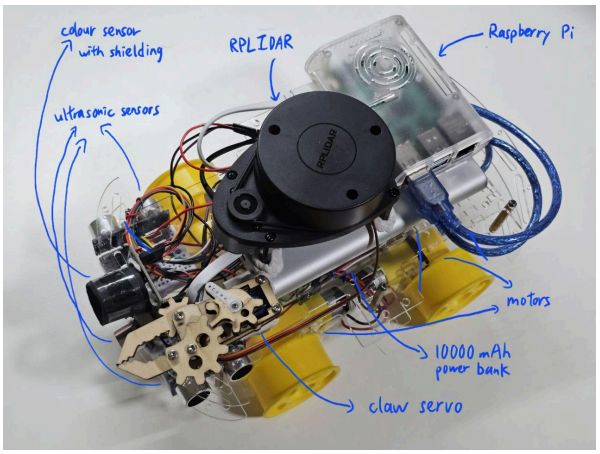


Diagram 4.1: Top view of Alex

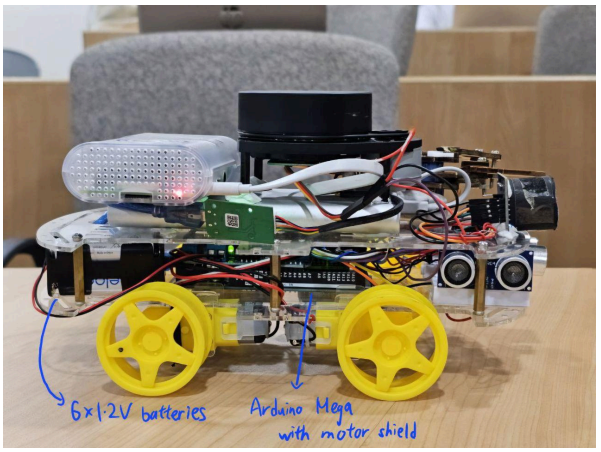


Diagram 4.2: Right view of Alex

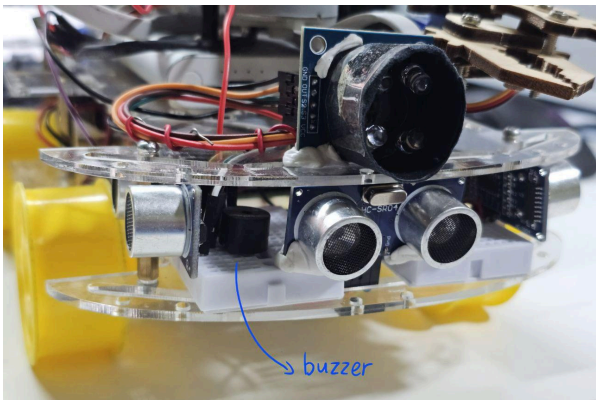


Diagram 4.3: Front view of Alex

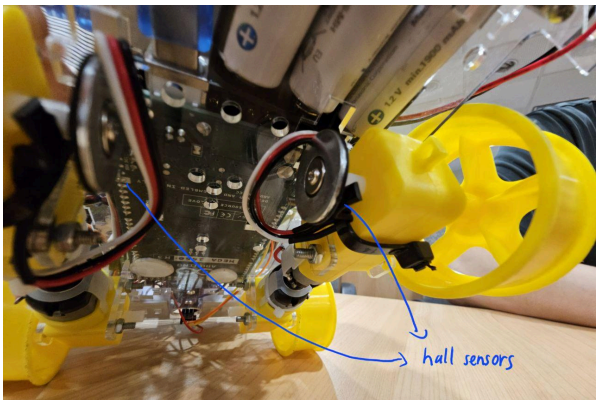


Diagram 4.4: Bottom view of Alex

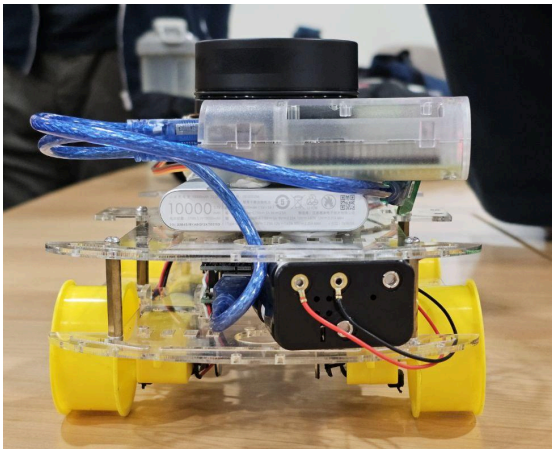


Diagram 4.5: Back view of Alex

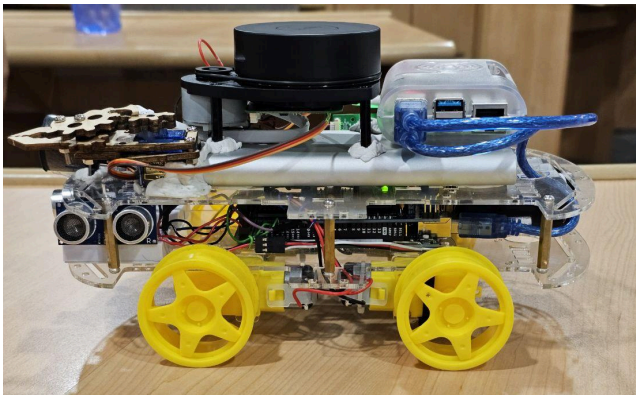


Diagram 4.6: Left view of Alex

Additional Hardware

Ultrasonic Sensors (Appendix 1) - We placed 3 ultrasonic sensors on the front, left and right sides of the robot respectively. By pressing the key "G" on our laptop, the ultrasonic sensor would return the distances between the robot and any objects or walls beside or in front of it, through the Arduino Mega and Raspberry Pi. Thus, it would enable us to better gauge distances and navigate more precisely. This was especially useful in precise movements, such as when the front of the Alex needed to be about 7 cm away from the victims for the most optimal colour sensor readings, and during parking. An example of the data returned would be:

“ Left = 10cm
Front = 5cm
Right = 7cm ”

Buzzer (Appendix 9) - The buzzer is connected to the Arduino Mega. Upon finishing the maze, we would press the key “Y” on our laptop, instructing the buzzer to play a celebratory tone through the Arduino Mega and Raspberry Pi, which is similar to the buzzer’s functionality in EPP1. Additionally, whenever a green or red “victim” is detected, we will press the key “U” or “J” on our laptop, depending on the green or red colour. This is meant to alert “victims” of the presence of

Claw Servo (Appendix 2) - The full excavator as seen in the design report was too large, thus, using the materials provided for the Excavator, we designed and built a small claw consisting of only one servo. The servo motor is connected to the Arduino Mega. The claw would open and close upon pressing “Z” on our laptop, simulating the movement of rubbles, or the dropping of essential equipment when a “victim” is detected.

Diagram 4.7: Designing Claw from scratch

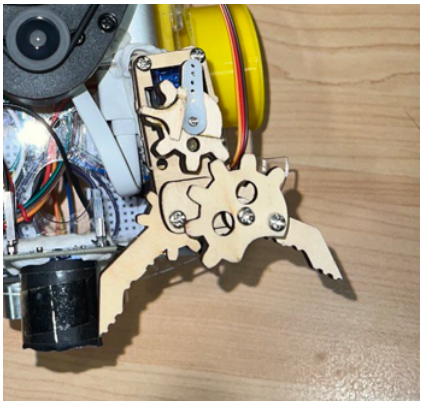


Diagram 4.8: Claw Servo in operation

All components are secured to their position via blu tac and are placed on top of each other on a flat surface. Furthermore, the Raspberry Pi is placed on the top layer to facilitate a wireless connection with the remote client (laptop) by avoiding obstacles from the robot components. This addresses the issues faced by Robbie (lack of stability) and RAPOSA (weakened wireless connection from antenna position) respectively as mentioned in section 2.

Section 5 Firmware Design

High level algorithm (Arduino Mega)

Initialization Process

On startup, the `setup()` function is called to initialise the system's hardware and software components. This involves calculating physical properties of the robot such as Alex’s diagonal and its circumference, setting up communication ports, setting up servos, and activating sensors. External interrupts are enabled to capture wheel encoder ticks, and the system enters a loop where it awaits a "HELLO" packet to confirm successful communication link establishment with the host (Pi). Once the handshake is acknowledged, interrupts are enabled globally.

Main Operational Loop

The core of the system's operation is contained within the `loop()` function, which repeats indefinitely. The primary tasks within this loop involve:

1. Packet Reception:

The system continuously monitors the serial interface for incoming packets using the `readPacket()` function. Packets are evaluated, and based on their integrity, they are either processed or flagged with an appropriate error response.

2. Packet Processing:

Upon receiving a valid packet, the `handlePacket()` function delegates command execution or message handling. The type of packet received dictates the flow, with command packets leading to action and response packets often being used for telemetry purposes.

3. Command Execution:

Execution of commands is managed by the `handleCommand()` function, which dispatches specific tasks like moving forwards or backwards, turning, toggling the claw servo, and activating audio signals based on the command type and parameters within the packet.

4. Distance Tracking:

As movement commands are being executed, the system uses wheel encoder readings from the interrupts to monitor progress. Once the robot traverses the desired distance or rotates the instructed angle, it stops, and the action is considered complete.

5. Status Updates and Messaging:

At any point, the robot may need to send data back to the controller. This can be a status update, including sensor readings after a `COMMAND_GET_STATS` request (explained in detail below) or error notifications in response to packet anomalies.

6. Error Handling:

When packet issues arise, corresponding error functions like `sendBadPacket()` or `sendBadChecksum()` are called to alert the controller, facilitating corrective measures or debugging.

Communication Protocol

The communication protocol comprises serialisation and deserialization routines, ensuring data packets are structured correctly before transmission and upon reception. The protocol governs the flow of information, ensuring coherence and synchronisation between the Raspberry Pi and the robot.

Detailed implementation of communication protocol

The `handleCommand()` **(Appendix 4)** function serves as the decision-making centre for executing instructions received from the Raspberry Pi. It forms the nexus between incoming command packets and the corresponding actions the robot must perform. The function employs a switch-case statement to identify the specific command and to delegate tasks to their designated handlers accordingly.

Each case within the function corresponds to a unique action identified by the command code, with a two-step processing approach for each action:
1. Acknowledgement: An immediate acknowledgment is sent back to the controller indicating that the command has been received and is understood (`sendOK()` function).
2. Action Execution: The robot carries out the requested action, utilising parameters included in the packet for details such as magnitude and speed.

Action-Specific Execution Logic
- Movement Commands (`COMMAND_W`, `COMMAND_D`, `COMMAND_A`, `COMMAND_FORWARD`, `COMMAND_REVERSE`, `COMMAND_TURN_LEFT`, `COMMAND_TURN_RIGHT`):
Due to the addition of new commands, these commands are first added into TCommandType struct **(Appendix 7)**. These cases involve directional movement or rotation of the robot for a specified distance or angle, and at a particular speed. The parameters provided in the command dictate these values. Movement functions such as `forward()`, `backward()`, `left()`, and `right()` are called with these parameters to drive the wheel motors accordingly. However, for commands (`COMMAND_W`, `COMMAND_D`, `COMMAND_A`, 'COMMAND_REVERSE'), we decided to let it run infinitely at a very low speed pre-defined at 45% power so that LiDAR is not affected **(Appendix 6)**. The reason for this is so that we can speed up the process of movement when passing through a large empty or visited terrain. The command 'COMMAND_STOP' is used when we need Alex to stop.

- Actuator Control (`COMMAND_CLAW`):
The actuator, in this case a claw servo, is operated to either open or close. The `toggleClaw()` function is responsible for changing the state of the claw and actuating the servo to the respective positions.

- Audio Feedback (`COMMAND_RED`, `COMMAND_GREEN`, `COMMAND_CELEBRATE`):
To provide audio cues **(Appendix 9)**, different routines can be triggered to play specific tones through the buzzer. Commands are played based on what's detected, like `redSiren()` for the red victim, `greenSiren()` for the green victim, and `celebrate()` when Alex successfully parked into the parking slot.

- Emergency Stop (`COMMAND_STOP`):
This function will halt any ongoing action and bring the robot to a stop. It is an essential feature for safety and control.

- Resetting Counters (`COMMAND_CLEAR_STATS`):
The `clearOneCounter()` function is used here to reset specific counters and variables back to their initial states. It provides a way to clear any tracked movement or sensor data.

- Error Handling within Command Execution
The function also includes default error handling, where if an unrecognised command is received (i.e., does not match any predetermined case in the switch statement), a

`sendBadCommand()` is called. This error feedback ensures that the Mega is aware of invalid or undefined commands.

- Telemetry (`COMMAND_GET_STATS`):
Upon this command, the `getColour()` and `measureDistances()` functions are invoked to collect sensor data including colour detection and ultrasonic distance measurements. The `sendStatus()` **(Appendix 10)** function gathers this data into a response packet that is sent back to the Pi. We decided to combine all the data required to send back into one command 'COMMAND_GET_STATS' and packed it into the same packet as it does not require us to write additional unnecessary message and response functions for the individual data.

Response Packet Overview
A response packet is employed to encapsulate and transmit telemetric data from the robot ("Alex") to the controller. Utilising the `TPacket` structure, the packet can deliver both textual and numerical data in a pre-defined format.

Packet Structure
The `TPacket` structure constituents are:
- `packetType`: A single-byte identifier indicating the type of packet. For response packets, this is set to `PACKET_TYPE_RESPONSE`.

- `command`: This byte is used to designate the specific response command code, such as `RESP_STATUS`, which indicates that the packet contains status information.

- `dummy[2]`: Two bytes reserved for padding, ensuring the structure's size aligns to a multiple of four bytes for potential architectural optimizations or protocol requirements such as the case for Raspberry Pi 4.

- `data[MAX_STR_LEN]`: An array of characters reserved for transmitting textual data, which could be used for descriptive messages or logging purposes, up to `MAX_STR_LEN` characters.

- `params[16]`: An array of 16 unsigned 32-bit integers (because Mega and Pi have different views on how many bytes an int is), providing ample space to carry numerical data such as sensor readings, tick counts, distances, or other telemetry parameters.

Encoding Telemetry Data
When encoding telemetry data into a response packet, the process typically involves the following steps:

1. Data Acquisition: The relevant telemetry data is retrieved from the robot's subsystems, such as ultrasonic range finders, wheel encoders, and colour sensors.

2. Packet Population: The retrieved data is populated into the `params` array of a `TPacket` structure. If necessary, associated textual information or result descriptors may be filled into the `data` field.

3. Serialising and Sending: The `sendResponse()` function serialises the response packet and transmits it over the UART serial connection to the Pi.

For example,
During the `COMMAND_GET_STATS` execution within `handleCommand()`, the `getColour()` and `measureDistances()` functions gather the colour sensor frequencies and ultrasonic sensor distances, respectively. This information is then stored in the `params` array. For example, `params[4]`, `params[5]`, and `params[6]` carries red, green, and blue sensor frequencies, whereas `params[1]`, `params[2]`, and `params[3]` holds left, front, and right distances measured by the ultrasonic sensors **(Appendix 10)**. It is then serialised and sent back to Pi via 'SendResponse()'.

Ultrasonic Sensor Algorithm (Appendix 1)

To put the lessons learnt from our laboratory sessions into practice, our group decided to use bare-metal programming to configure the Ultrasonic Sensor to get distance measurements. This approach heavily utilised the concept of GPIO programming, Timers and Interrupts. By doing so, we were able to replicate the functionalities provided by Arduino's higher-level functions such as digitalWrite and pulseIn through direct register manipulation and low-level hardware interfacing.

Firstly, we configured Timer 5 with an overflow interrupt enabled to set the digital pins 22 to 27 as either outputs or inputs, depending on their function as trigger or echo pins, respectively. This setup ensures that each ultrasonic sensor's trigger and echo signals are correctly handled.

The core of our measurement algorithm relies on the precise timing facilitated by Timer 5, which operates without a prescaler. This means TCNT5 counts up to 65535 at the clock rate of 16 MHz. This counting continues while the echo pin remains high, indicating the receiving of an echo signal from an obstacle. The count halts when the echo pin goes low, marking the end of the echo pulse. The value of TCNT5 at this point is stored in the totalDuration variable. From this, we calculate the pulse duration by dividing totalDuration by 16,000,000 to convert it to seconds. Using the speed of sound, we get an accurate reading of the distance.

However, a challenge arises when the pulse duration is longer than the timer's capacity, causing an overflow and resetting TCNT5 to zero. This scenario limited our initial measurement capability to distances up to approximately 70 cm. To address this, we implemented an interrupt service routine (ISR) for the timer overflow. This ISR increments an overflowCount each time the timer overflows. We then adjust the totalDuration calculation to account for these overflows, effectively extending our measurable range beyond 70 cm. The updated formula, TCNT5 + (overflowCount * 65535), allows us to accurately calculate longer distances by considering the total number of complete timer cycles plus the current count.

Claw Servo Algorithm (Appendix 2)

Similarly, using bare-metal, the digital pin 46 on the Arduino Mega is configured as an output to control the servo. This pin is associated with OC5A, the output compare register for Timer

5. The Timer is set up in Fast PWM mode where the period of the PWM signal is determined by ICR5. Setting ICR5 to 39999 and using a prescaler of 8 configures the PWM frequency to approximately 50 Hz, which is recommended for servo operation.

The toggleClaw function manipulates the duty cycle of the PWM signal to control the servo's position. It adjusts OCR5A between two values representing the open and closed positions of the servo (2000 ticks for 1 ms (0 degrees), and 4000 ticks for 2 ms (180 degrees), respectively). The state of the servo (open or closed) is tracked using a boolean variable clawIsOpen. A delay of 15 milliseconds after each toggle ensures that the servo has enough time to reach the desired position.

Colour Detection Algorithm (Appendix 3)

When configuring the colour sensor, there are two main aspects of consideration (apart from the colour itself): distance and ambient lighting. Firstly, to prevent the fluctuations of frequency values from ambient lighting, black cardboard paper is wrapped around the four LEDs and the colour sensor itself to improve its accuracy. Alex is also tested in rooms with different lighting conditions to ensure that the algorithm is independent of lighting conditions. Nonetheless, due to the sensitivity of the colour sensor, the black shielding is still insufficient to block out all of the ambient light in the room.

Another aspect of consideration was distance. During the calibration process, the reflected light values increase with the distance between the sensor and the coloured surface. Hence, using the forward distance measured by the ultrasonic sensor, our initial plan was to complete colour calibration and identification at a predefined distance of 5cm. However, the process of accurately positioning the robot at a specified distance for colour calibration is time-consuming and risks too many unnecessary movements.

Hence, instead of designing a colour detection algorithm with values, the team opted for a ratio comparison algorithm instead. The flowchart below depicts the colour calibration algorithm. One thing to note is that the colour frequency is inversely proportional to the RGB values. The algorithm has helped us to accurately detect the colour in front of Alex quickly, thus giving us the upper hand.

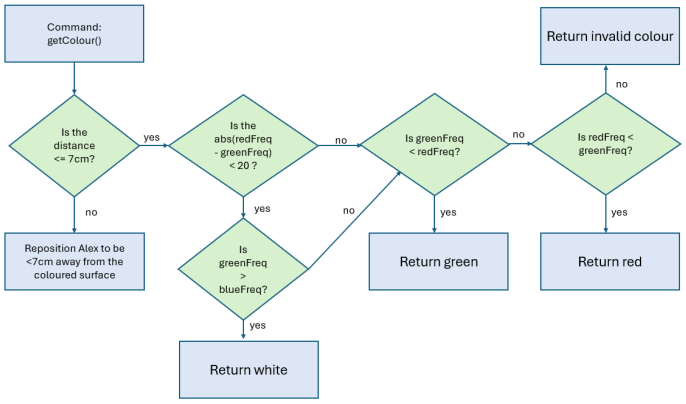


Diagram 5.1: Flowchart of colour sensor algorithm

With this algorithm, it detects colours based on the **differences** in respective red, blue and green frequency which is independent from lighting conditions and distance, mitigating the two issues as stated above. Nonetheless, the distance is still clocked at 7 cm to minimise ambient light.

Section 6 Software Design

High Level algorithm (Raspberry Pi 4)

1. Remote Client Initiation

A remote client (Laptop) initiates a connection to the teleoperation server running on the Raspberry Pi. This connection is secured using TLS, ensuring that the commands sent to the robot are encrypted and secure.

2. TLS Server on Raspberry Pi

The Raspberry Pi runs a TLS server that listens for incoming connections on the specified `SERVER_PORT`. The server uses the provided SSL certificates and keys to authenticate itself to clients and optionally requires client authentication as well.

3. Command Reception and Handling

The `worker` thread is responsible for handling incoming network data from the connected client. Here's how it processes commands:

- The thread calls `sslRead` to read data from the TLS connection into a buffer.
- If the buffer contains a `NET_COMMAND_PACKET`, it is passed to the `handleCommand` function.
- The `handleCommand` function interprets the command and parameters from the buffer and constructs a `TPacket` structure with the appropriate robot command and parameters. Within the TPacket, the command will be set to `COMMAND_W`, `COMMAND_D`, `COMMAND_A`, `COMMAND_FORWARD`, `COMMAND_REVERSE`, `COMMAND_TURN_LEFT`, `COMMAND_TURN_RIGHT`, ‘COMMAND_CLAW’, ‘COMMAND_RED’, ‘COMMAND_GREEN’, ‘COMMAND_CELEBRATE’, ‘COMMAND_CLEAR_STATS’ and ‘COMMAND_GET_STATS’, according to the key pressed such as ‘z’ for ‘COMMAND_CLAW’, ‘y’ for ‘COMMAND_CELEBRATE’, etc, by the user controlling the robot on the client side of the TLS connection.
- This `TPacket` is then serialised and sent over the serial connection to the Arduino using the `uartSendPacket` function.
- Arduino then deserialises this packet, send back an ok status and execute the commands within it.

4. Response and Status Updates

Upon executing a command, the Arduino sends back a response packet to the Raspberry Pi, which may include status information such as colour detection or distances to the front, left and right of Alex or confirmation of the command execution. The Raspberry Pi then uses the `sendNetworkData` function to relay this information back to the remote client over the TLS connection.

5. Communicating with client and Arduino

- The Raspberry Pi communicates with the Arduino using a UART serial connection.
- The `uartReceiveThread` thread listens for incoming packets from the Arduino over the serial connection.
- When a packet is received, it is deserialized and depending on the packet type, the appropriate handler (`handleResponse`, `handleError`, `handleMessage`) is called.
- Responses from the Arduino are sent back to the remote client using the `sendNetworkData` function. This function writes the serialised data to the `tls_conn` TLS connection. Through the readerThread function, the client then reads the data in via sslRead and handles it based on its packet type. In this case, we put it under NET_STATUS_PACKET thus the handle Status() function is called which prints out the data to the user interface at the client side **(Appendix 5)**.

Section 7 Conclusion

<u>Challenge 1 - Movement Commands and Coordination</u>

Mistake made: Manually inputting commands for every movement

Initially, our movement command was based on pressing a key on our laptop, before inputting two parameters, which are the distance to move/angle to turn and the percentage of power the robot would move. As such, we found specific distances/angles and powers that were more ideal for our robot's movement. However, this was difficult to memorise and we had to write it down.

Thus, this led to our team's roles for the trial run to be:
Anderson - Saying of movement parameters + Rviz Map Handler
Tze Yong - Navigation Advisor
Jensen - Overall facilitator + Timekeeper
Nancy - Map Drawer
Hao Zhe - Robot controller

Unfortunately, having to request for various parameters, as well as having to type them down, was extremely time-consuming. Furthermore, it hindered our team coordination. Consequently, we managed to finish our trial run on the dot and did not manage to detect the third object.

Lesson learnt: Allow forward, backward, left and right movement to go to infinity

We decided to implement additional movement commands, 'w' to move forward, 'a' to move left, 'd' to move right, all to infinity. These movement commands would make the robot run to infinity unless commanded to stop. They do not require any parameters for distance/angle and power, as the distance/angle is hard coded to infinity, whereas the power is hardcoded to an ideal power for the robot, such as 45. Additionally, we did not need to press the stop key everytime after a movement was made. Instead, we could press another movement key, and the robot would change its movement instantaneously.

Our team roles switched to:
Anderson - Advisor for navigation and map drawing
Tze Yong - Advisor for navigation + Rviz Map Handler

Jensen - Overall facilitator + Timekeeper
Nancy - Map Drawer
Hao Zhe - Robot controller

With these changes, our time efficiency improved drastically. Our movements were much smoother, and we finished our final run in 3 minutes and 28 seconds while detecting all 3 objects correctly, a massive improvement compared to our trial run.

Challenge 2 - Importance of understanding and managing microcontroller resources

Mistake made: Initially, during the integration of our bare-metal ultrasonic sensor code into the project, we encountered difficulties in getting the ultrasonic sensor to function correctly. After debugging for a long time, this issue was traced back to a timer conflict. Specifically, the timer that we initially selected for the ultrasonic sensor's operation (Timer 1), seemed to be already heavily utilised by the motor shield for generating PWM signals for motor speed control. Furthermore, Timer 2, another potential option, might also be in use by third-party libraries or extensions connected to the shield, particularly for controlling servos or providing additional PWM outputs. Due these timer conflicts, our group found that Timers 1 to 3 were all unavailable for use for our Ultrasonic Sensor. Fortunately, unlike the Arduino Uno, there are 6 sets of Timers made available to us.

Lesson learnt: To resolve this conflict and ensure reliable functionality, we switched to using Timer 5 for the ultrasonic sensor. Timer 5 was chosen because it was not being used by other components in our system, thus avoiding the issues associated with timer sharing. After making this adjustment, the ultrasonic sensor code worked as intended, demonstrating the importance of carefully selecting and managing hardware resources in embedded system programming to avoid conflicts and ensure operational reliability.

All in all, we would like to express our sincere gratitude to the teaching assistants, the lab staff and Professor Boyd Anderson for their invaluable guidance throughout this project. They have helped us countlessly and tirelessly, answering our questions and helping to solve any problems we encountered. Subsequently, also with our dedication and hard work poured into Alex, it bore fruit in the final sprint, where we completed the course in 3 minutes and 28 seconds, while also successfully detecting both victims and mapping out the whole course. Without their help, this project definitely would not have gone as smooth-sailed as it did. Furthermore, Alex has adeptly illuminated numerous real-life applications concerning the movement of robots, lidar and how computers communicate, enriching our understanding and enhancing the project's outcomes. This is definitely the first stepping stone of many to lead us to become real-life computer engineers.

References

1. *Raposa*. Intelligent Robots and Systems Group. (n.d.-a). https://irsgroup.isr.tecnico.ulisboa.pt/raposa/

2. Lima, P. U. (2014, May 25). *Raposa Ng – a search and Rescue Land Wheeled Robot*. old.eu. https://old.eu-robotics.net/sparc/success-stories-old/raposa-ng-a-search-and-rescue-land-wheeled-robot.html?changelang=2

3. Marques, C., Cristóvão, J., Alvito, P., Lima, P., Frazão, J., Ribeiro, I., & Ventura, R. (2007). A search and rescue robot with Tele‐operated Tether Docking System. Industrial Robot: An International Journal, 34(4), 332–338. https://doi.org/10.1108/01439910710749663

4. Novotny, G[eorg]; Emsenhuber, S[imon]; Klammer, P[hilipp]; Poschko, C[ristoph]; Voglsinger, F[lorian] & Kubinger, W[ilfried] (2019). A Mobile Robot Platform for Search and Rescue Applications, Proceedings of the 30th DAAAM International Symposium, pp.0945-0954, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-22-8, ISSN 1726-9679, Vienna, Austria DOI: 10.2507/30th.daaam.proceedings.131

5. Leggedrobotics. (2018, April 26). Yolo v3 crashes after detections on first image · issue #81 · leggedrobotics/darknet_ros. GitHub. https://github.com/leggedrobotics/darknet_ros/issues/81

6. Leggedrobotics. (2024, February 11). *Always waiting for image · issue #142 · leggedrobotics/darknet_ros*. GitHub. https://github.com/leggedrobotics/darknet_ros/issues/142

Appendix

```
#include <avr/io.h>
#include <util/delay.h>
#include <avr/interrupt.h>

// Define pins by their bit positions in the PORT registers
#define TRIG1_PIN_BIT 0  // Bit 0 of PORTA for the first sensor's trigger (Left)D22
#define ECHO1_PIN_BIT 1  // Bit 1 of PIND for the first sensor's echo (Left)D23
#define TRIG2_PIN_BIT 2  // Bit 2 of PORTA for the second sensor's trigger (Front)D24
#define ECHO2_PIN_BIT 3  // Bit 3 of PIND for the second sensor's echo (Front)D25
#define TRIG3_PIN_BIT 4  // Bit 4 of PORTA for the third sensor's trigger (Right)D26
#define ECHO3_PIN_BIT 5  // Bit 5 of PIND for the third sensor's echo (Right)D27

volatile unsigned long distanceLeft = 0;
volatile unsigned long distanceFront = 0;
volatile unsigned long distanceRight = 0;

// Timer overflow count
volatile unsigned long overflowCount = 0;

void setup() {
  Serial.begin(9600);
  ultrasonic_init(); // Initialize the ultrasonic sensors
  timer5_init(); // Initialize the timer
}

// ISR for timer overflows so that we can keep count how many times it overflows if not
max dist is only 70cm
ISR(TIMER5_OVF_vect) {
  overflowCount++;
}

// Timer5 initialization
void timer5_init() {
  cli();
  TCCR5A = 0; // Clear Timer/Counter5 Control Register A
  TCCR5B = 0; // Clear Timer/Counter5 Control Register B
  TCCR5B |= 0b00000001; // Set no prescaler
  TCNT5 = 0; // Clear Timer/Counter5
  TIMSK5 |= 0b00000001; // Enable Timer/Counter5 Overflow Interrupt
  overflowCount = 0; // Reset overflow counter
  sei(); // Enable global interrupts
}
// Ultrasonic sensor initialization
void ultrasonic_init() {
  // Set trigger pins as outputs on PORTA
  DDRA |= (1 << TRIG1_PIN_BIT) | (1 << TRIG2_PIN_BIT) | (1 << TRIG3_PIN_BIT);
  // Set echo pins as inputs on PORTA
  DDRA &= ~((1 << ECHO1_PIN_BIT) | (1 << ECHO2_PIN_BIT) | (1 <<
ECHO3_PIN_BIT));
}
```

```
// Function to trigger ultrasonic pulse
void triggerUltrasonic(uint8_t trigPinBit) {
  PORTA |= (1 << trigPinBit);  // Set the corresponding trigger pin HIGH
  _delay_us(10);               // Wait for 10 microseconds
  PORTA &= ~(1 << trigPinBit); // Set the trigger pin LOW
}

// Function to measure the distance
unsigned long measureDistance(uint8_t trigPinBit, uint8_t echoPinBit) {
  overflowCount = 0; // Reset the overflow counter
  TCNT5 = 0;         // Reset the timer value
  triggerUltrasonic(trigPinBit); // Trigger the ultrasonic pulse

  // Wait for the echo pin to go high (start of the echo signal)
  while (!(PINA & (1 << echoPinBit)));
  TCNT5 = 0; // Reset the timer value again to measure the high pulse width

  // Wait for the echo pin to go low (end of the echo signal)
  while (PINA & (1 << echoPinBit));

  // Calculate the duration in microseconds and distance in cm
  unsigned long totalDuration = TCNT5 + (overflowCount * 65535);
  unsigned long durationInMicroseconds = totalDuration * (1.0 / 16000000.0) * 1000000.0;
  unsigned long distance = (durationInMicroseconds * 0.0346) / 2;

  return distance;
}

void measureDistanceLeft() {
    distanceLeft = measureDistance(TRIG1_PIN_BIT, ECHO1_PIN_BIT);
}

void measureDistanceFront() {
    distanceFront = measureDistance(TRIG2_PIN_BIT, ECHO2_PIN_BIT);
}

void measureDistanceRight() {
    distanceRight = measureDistance(TRIG3_PIN_BIT, ECHO3_PIN_BIT);
}
```

Appendix 1: Bare-metal code for configuration of Ultrasonic Sensor

```
#define SERVO_PIN 46  // Pin 46 uses OC5A (Timer 5)

// Define the pulse widths in terms of timer ticks
const unsigned int openPositionTicks = 4000;   // 2 ms pulse corresponding to 180 deg
const unsigned int closedPositionTicks = 2000; // 1 ms pulse corresponding to 0 deg

bool clawIsOpen = false;  // State of the claw

void setup() {
```

```
    // Set SERVO_PIN as output
    DDRL |= 0b00000001;  // PL0 is Pin 46 on Arduino Mega

    // Setup Timer 5 for PWM
    TCCR5A = 0b10000010;  // Non-inverting mode on OC5A, Fast PWM, TOP is ICR5
    TCCR5B = 0b00011010;  // Fast PWM, TOP is ICR5, Prescaler set to 8

    ICR5 = 39999;  // TOP value for 50Hz frequency & 20ms period with a prescaler of 8

    // Set servo to closed position initially
    OCR5A = closedPositionTicks;
}

void toggleClaw() {
    if (clawIsOpen) {
        OCR5A = closedPositionTicks; // Move servo to closed position
        clawIsOpen = false;
    } else {
        OCR5A = openPositionTicks;   // Move servo to open position
        clawIsOpen = true;
    }
    _delay_ms(15); // Delay to allow the servo to reach position
}
```

Appendix 2: Bare-metal code for configuration of Claw Servo

```
#define S0_PIN_BIT 2 // pin 47 PL2
#define S1_PIN_BIT 4 // pin 45 PL4
#define S2_PIN_BIT 6 // pin 43 PL6
#define S3_PIN_BIT 0 // pin 41 PG0
#define sensorOut_PIN_BIT 2 // pin 39 PG2

int redFreq = 0;
int greenFreq = 0;
int blueFreq = 0;
int color = 0;

int averageReading() {
  int total = 0;
  for (int i = 0; i < 5; i += 1) {
    int reading = pulseIn(39, LOW);
    total += reading;
  }
  return total / 5;
}

int getRed() {
  // Setting RED (R) filtered photodiodes to be read
  PORTL &= ~(1 << S2_PIN_BIT);
  PORTG &= ~(1 << S3_PIN_BIT);
  redFreq = averageReading();
```

```
  return redFreq;
}

int getGreen() {
  // Setting GREEN (G) filtered photodiodes to be read
  PORTL |= (1 << S2_PIN_BIT);
  PORTG |= (1 << S3_PIN_BIT);
  greenFreq = averageReading();
  return greenFreq;
}

int getBlue() {
  // Setting BLUE (B) filtered photodiodes to be read
  PORTL &= ~(1 << S2_PIN_BIT);
  PORTG |= (1 << S3_PIN_BIT);
  blueFreq = averageReading();
  return blueFreq;
}

void getColour() {
  redFreq = getRed();
  delay(50);
  greenFreq = getGreen();
  delay(50);
  blueFreq = getBlue();
  delay(50);


  if (abs(redFreq - greenFreq) < 20 && greenFreq > blueFreq) {
    color = 1;  //white colour detected
  }
  else if (greenFreq < redFreq) {
    color = 2;  // green colour detected
  } else if (redFreq < greenFreq) {
    color = 3;  //red colour detected
  } else {
    color = 0;  //invalid colour
  }
}

void setupColor() {
  // Set S0, S1, S2, S3 as output, sensorOut as input
  DDRL |= (1 << S0_PIN_BIT) | (1 << S1_PIN_BIT) | (1 << S2_PIN_BIT);
  DDRG |= (1 << S3_PIN_BIT);
  DDRG &= ~(1 << sensorOut_PIN_BIT);
  // Set S0 to HIGH, S1 to LOW, so output frequency scaling is set to 20%
  PORTL |= (1 << S0_PIN_BIT);
  PORTL &= ~(1 << S1_PIN_BIT);
}
```

Appendix 3: Bare-metal code for configuration of Colour Sensor

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

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

	TPacket commandPacket;

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

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

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

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

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

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

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

		case 'w':
		case 'W':
```

```
            commandPacket.command = COMMAND_W;
            uartSendPacket(&commandPacket);
            break;

        case 'a':
        case 'A':
            commandPacket.command = COMMAND_A;
            uartSendPacket(&commandPacket);
            break;

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

        case 'c':
        case 'C':
            commandPacket.command = COMMAND_CLEAR_STATS;
            commandPacket.params[0] = 0;
            uartSendPacket(&commandPacket);
            break;

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

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

        case 'u':
        case 'U':
            commandPacket.command = COMMAND_GREEN;
            uartSendPacket(&commandPacket);
            break;

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

        case 'y':
        case 'Y':
            commandPacket.command = COMMAND_CELEBRATE;
            uartSendPacket(&commandPacket);
            break;
```

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

        }
}
```

Appendix 4: handleCommand() function in Raspberry Pi

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

        printf("\n ------- ALEX STATUS REPORT ------- \n\n");
        printf("Colour:\t\t%d\n", data[0]);
        printf("Left distance:\t\t%d\n", data[1]);
        printf("Front distance:\t\t%d\n", data[2]);
        printf("Right distance:\t\t%d\n", data[3]);
        printf("Red:\t\t%d\n", data[4]);
        printf("Green:\t\t%d\n", data[5]);
        printf("Blue:\t\t%d\n", data[6]);
        printf("\n---------------------------------------\n\n");
}
```

Appendix 5: handleStatus() function in Laptop to print out telemetry

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

        while(!quit)
        {
                char ch;
                printf("Command (w = Fast Forward, a = Fast Left, d = Fast Right, f =
Forward, b = Backward, l = Turn Left, r = Turn Right, s = Stop, c = Clear Stats, z = Claw ,
u = Green Siren , j = Red Siren , y = Celebrate , g = Get Stats, q = Exit)\n");
                scanf("%c", &ch);

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

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

                buffer[0] = NET_COMMAND_PACKET;
                switch(ch)
                {
                        case 'f':
```

```
case 'F':
case 'l':
case 'L':
case 'r':
case 'R':
                getParams(params);
                buffer[1] = ch;
                memcpy(&buffer[2], params, sizeof(params));
                sendData(conn, buffer, sizeof(buffer));
                break;
case 's':
case 'S':
case 'c':
case 'C':
case 'g':
case 'G':
case 'z':
case 'Z':
case 'u':
case 'U':
case 'j':
case 'J':
case 'y':
case 'Y':
        params[0]=0;
        params[1]=0;
        memcpy(&buffer[2], params, sizeof(params));
        buffer[1] = ch;
        sendData(conn, buffer, sizeof(buffer));
        break;

case 'w':
case 'W':
case 'b':
case 'B':
        params[0]=0;
        params[1]=45;
        memcpy(&buffer[2], params, sizeof(params));
        buffer[1] = ch;
        sendData(conn, buffer, sizeof(buffer));
        break;


case 'a':
case 'A':
        params[0]=0;
        params[1]=45;
        memcpy(&buffer[2], params, sizeof(params));
        buffer[1] = ch;
        sendData(conn, buffer, sizeof(buffer));
        break;
```

```
                        case 'd':
                        case 'D':
                                params[0]=0;
                                params[1]=45;
                                memcpy(&buffer[2], params, sizeof(params));
                                buffer[1] = ch;
                                sendData(conn, buffer, sizeof(buffer));
                                break;

                        case 'q':
                        case 'Q':
                                quit=1;
                                break;

                        default:
                                printf("BAD COMMAND\n");
                }
        }

        printf("Exiting keyboard thread\n");
        stopClient;
        EXIT_THREAD(conn);
   return NULL;
}
```

Appendix 6: Additional commands in writerThread() for high speed movement

```
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_CLAW = 7,
  COMMAND_GREEN = 8,
  COMMAND_RED = 9,
  COMMAND_CELEBRATE = 10,
  COMMAND_W = 11,
  COMMAND_A = 12,
  COMMAND_D = 13
} TCommandType;
```

Appendix 7: Additional commands in stuct TCommandType to handle new functions

```
void enablePullups()
{
  DDRD &= 0b11110011;
  PORTD |= 0b00001100;
```

```
}

void setupEINT()
{
  cli();
  EICRA = 0b10100000;
  EIMSK = 0b00001100;
  sei();
}

// Global serialisation variables
#define BUFFER_LEN 256     // Buffer is initialized to accept up to "size" characters.
// buffers for UART
TBuffer _recvBuffer;
TBuffer _xmitBuffer;

void setupSerial()
{
  // Initialize the buffer. We must call this before using writeBuffer or readBuffer.
  // Buffer to use is specified in "buffer", size of buffer in characters is specified in "size"
  //set baud rate to 9600 and initialize our transmit and receive buffers
  initBuffer(&_recvBuffer, BUFFER_LEN);
  initBuffer(&_xmitBuffer, BUFFER_LEN);
  UBRR0L = 103;
  UBRR0H = 0;

  //set frame format: 8 bit, no parity, 1 stop bit (8N1)
  UCSR0C = 0b110;
  UCSR0A = 0;
}

void startSerial()
{
  // Enable USART transmitter and receiver
  // USART_RX_vect to be triggered when a character is received
  // USART_UDRE_vect intterupt triggered when sending data register is empty
  UCSR0B = 0b10111000;
}

int readSerial(char *buffer)
{

  int count=0;
  TBufferResult result;

  do{
    result = readBuffer(&_recvBuffer, &buffer[count]);
    if (result == BUFFER_OK){
      count++;
    }
    if (!dataAvailable(&_recvBuffer)) {
       break;
```

```
    }
  } while (result == BUFFER_OK);

  return count;
}

void writeSerial(const char *buffer, int len)
{
  TBufferResult result = BUFFER_OK;
  for(int i = 1; i < len; i += 1){
    result = writeBuffer(&_xmitBuffer, buffer[i]);
    if (result != BUFFER_OK) {
       break;
    }
  }
  //read and write data from UDR0
  UDR0 = buffer[0];
  UCSR0B |= 0b00100000;
}


ISR(USART_RX_vect){
  // Data from UDR0 is read and written to data
  unsigned char data = UDR0;
  writeBuffer(&_recvBuffer, data);
}

ISR(USART_UDRE_vect){
  // Data to be send is copied into UDR0
  unsigned char data;
  TBufferResult result = readBuffer(&_xmitBuffer, &data);

  if (result == BUFFER_OK){
    UDR0 = data;
  }
  else if (result == BUFFER_EMPTY){
     UCSR0B &= 0b11011111;
  }
}
```

Appendix 8: Remaining bare-metal code in Arduino Mega

```
#define buzzerPin 7   // Arduino Mega pin 30 corresponds to PORTC7

void setupBuzzer() {
  // Set the buzzer pin as output
  DDRC |= (1 << DDC7);
}

void tone(unsigned int frequency) {
  // Calculate the period of the wave in microseconds
  unsigned long period = 1000000 / frequency;
```

```
  // Set the prescaler to 64
  TCCR4B = (TCCR4B & 0xF8) | 0x03;

  // Set the output compare register
  OCR4A = period - 1;

  // Set timer mode (CTC mode)
  TCCR4A |= (1 << WGM41);

  // Enable timer interrupt
  TIMSK4 |= (1 << OCIE4A);

  // Start the timer
  TCCR4B |= (1 << CS40);
}

void noTone() {
  // Turn off the timer
  TCCR4B = 0;
  // Turn off the buzzer pin
  PORTC &= ~(1 << PORTC7);
}

// Timer 4 output compare A match interrupt service routine
ISR(TIMER4_COMPA_vect) {
  // Toggle the buzzer pin
  PORTC ^= (1 << PORTC7);
}

void celebrate() {
  // Define timer registers
  TCCR4A = 0; // Clear timer control registers
  TCCR4B = 0;
  TCNT4 = 0; // Reset timer counter
  // Set the buzzer pin as output
  DDRC |= (1 << DDC7);

  // Play tones
  tone(190);
  delay(200);
  tone(220);
  delay(200);
  tone(220);
  delay(200);
  tone(587.33);
  delay(200);
  tone(587.33);
  delay(200);
  tone(739.99);
  delay(200);
  tone(739.99);
```

```
delay(200);
tone(880);
delay(200);
tone(185);
delay(200);
tone(220);
delay(200);
tone(220);
delay(200);
tone(587.33);
delay(200);
tone(587.33);
delay(100);
tone(740);
delay(200);
tone(740);
delay(200);
tone(880);
delay(200);
tone(987.77);
delay(600);
tone(783.99);
delay(150);
tone(1174.66);
delay(1600);
tone(185);
delay(200);
tone(220);
delay(200);
tone(220);
delay(200);
tone(587.33);
delay(200);
tone(587.33);
delay(200);
tone(739.99);
delay(200);
tone(739.99);
delay(200);
tone(880);
delay(200);
tone(185);
delay(200);
tone(220);
delay(200);
tone(220);
delay(200);
tone(587.33);
delay(200);
tone(587.33);
delay(200);
tone(740);
```

```
  delay(200);
  tone(740);
  delay(200);
  tone(880);
  delay(200);
  tone(1046.50);
  delay(500);
  tone(1046.50);
  delay(200);
  tone(1046.50);
  delay(1200);
  tone(1046.50);
  delay(200);
  tone(987.77);
  delay(500);
  tone(1046.50);
  delay(150);
  tone(783.99);
  delay(1000);
  noTone();
}

void greenSiren() {
  // Define timer registers
  TCCR4A = 0; // Clear timer control registers
  TCCR4B = 0;
  TCNT4 = 0; // Reset timer counter
  // Set the buzzer pin as output
  DDRC |= (1 << DDC7);

  // Play green siren
  tone(100);
  delay(300);
  tone(200);
  delay(300);
  tone(100);
  delay(300);
  tone(200);
  delay(300);
  tone(100);
  noTone();
}

void redSiren() {
  // Define timer registers
  TCCR4A = 0; // Clear timer control registers
  TCCR4B = 0;
  TCNT4 = 0; // Reset timer counter
  // Set the buzzer pin as output
  DDRC |= (1 << DDC7);

  // Play red siren
```

```
  tone(800);
  delay(300);
  tone(700);
  delay(300);
  tone(800);
  delay(300);
  tone(700);
  delay(300);
  tone(800);
  noTone();
}
```

Appendix 9: Buzzer code in Arduino Mega

```
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] = color;
  statusPacket.params[1] = distanceLeft;
  statusPacket.params[2] = distanceFront;
  statusPacket.params[3] = distanceRight;
  statusPacket.params[4] = redFreq;
  statusPacket.params[5] = greenFreq;
  statusPacket.params[6] = blueFreq;
  sendResponse(&statusPacket);
}
```

Appendix 10 : sendStatus() function on the Arduino to pack telemetry to send to Pi