CG2111A Engineering Principle and Practice

Semester 2 2024/2025

“Alex to the Rescue”

Final Report

Team: B04-2B

Name	Student #	Main Role
Cai Jiali	A0288553N	Set up programs for Environment Mapping & Teleoperation
Tan Yan Hao Malcolm	A0308944W	Design and build arm
Wen Jun Yu	A0309098U	Wire sensors and cable management
Wesley Low	A0307650J	Program Arduino code for movement and rescue operations

Section 1 Introduction

In March 2025, a magnitude 7.7 earthquake struck near Mandalay, Myanmar's second largest city resulting in over 3735 casualties, 5108 injuries and over 120 missing. This has made search and rescue operations all the more relevant. Using mapping technology such as SLAM to navigate through rough terrain and identify survivors, rescue teams would be able to locate and extricate survivors much more easily, which would reduce the death toll of natural disasters.

An explosion has occurred in Moonbase CEG, and we have two stranded astronauts that need to be rescued. To do so, we designed Alex, a search and rescue robot equipped with LIDAR technology, allowing us to map and navigate its surroundings to identify and rescue the astronauts. There is a green astronaut who requires minimal medical attention only needing a medical package (medpak) deployed nearby, and a red astronaut who is in critical condition and needs to be dragged to the safe zone for further treatment. In order to perform these tasks, Alex is equipped with several sensors such as colour detection to identify the different astronauts, as well as an ultrasonic sensor to avoid collisions with nearby obstacles. During the mission, Alex was remotely operated via teleoperation over the network. It was tasked with mapping the environment, locating and identifying both astronauts, and delivering the appropriate aid—all within a strict 8-minute time limit, while avoiding collisions with walls and other obstacles.

Section 2 Review of State of the Art

2.1 RAPOSA

RAPOSA is a search robot with advanced sensors for navigation and hazard detection. It integrates webcams, thermal and IR cameras, temperature/humidity/toxic gas sensors, and analog tilt sensors with an accelerometer (low-pass filtered) for precise orientation. The docking system allows detachment from its tether in obstructed areas. These features enable RAPOSA to operate effectively in challenging conditions, making it a reliable tool for search missions.

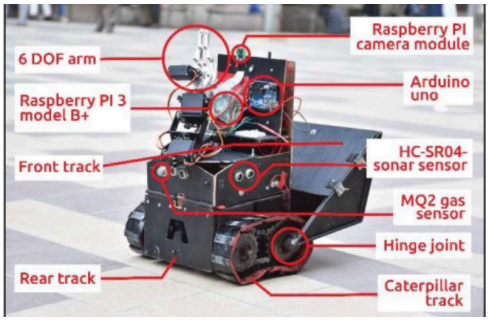
Analysis of the strengths and weaknesses of this platform are summarised in the table below:

Strengths	**Weaknesses**
● Able to determine hostility of the environment through the sensors	● Effectiveness compromised if there is a lost in communication

• Able to get better perception of distance through IR sensors • Operators are able to gauge the state of the robot • Flexibility to operate remotely or through connection	• Ability to navigate through uneven terrain is not tested • Tether docking system prone to malfunction in unpredictable environment

2.2 3-Survivor

3-Survivor is a mobile rescue robot engineered for rough terrain with a double-tracked caterpillar mechanism and passive balance control. It features a 6-DOF robotic arm and a Raspberry Pi-based 12 MP camera for live video streaming. Integrated object detection enhances situational awareness, while a handheld RF transmitter and web-based interface enable real-time remote control. The figure beside shows the overall design of 3-Survivor.



Analysis of the strengths and weaknesses of this platform are summarised in the table below:

Strengths	**Weaknesses**
• Robust mobility over rough terrain. • Enhanced stability via passive balance control. • Precise manipulation with a 6-DOF arm. • High-resolution live streaming with effective object detection. • User-friendly remote control through an intuitive web interface.	• Limited autonomy due to teleoperation reliance. • Vulnerable to RF communication interference. • Complex integration may increase maintenance challenges. • High processing demands can affect real-time responsiveness.

Section 3 System Architecture

Alex involves 14 devices in total: A Raspberry Pi acting as the main controller, connected to a LIDAR, Arduino and Pi camera, which are powered by a power bank. The Arduino acted as a controller for the motors through a motor shield, encoders, colour sensor, ultrasonic sensor and servo motors for both the claw and the dispenser. The motor shield is powered by a 10.5V battery pack. On the operator

end, we connected all our laptops to a Samsung phone for a localised IP address. We used 3 laptops, one for the TLS client to send commands to the Pi via a keyboard, one to visualise the LIDAR and SLAM data and one to activate the Pi camera. The figure below shows the system architecture of Alex, illustrating all components and their interconnections.

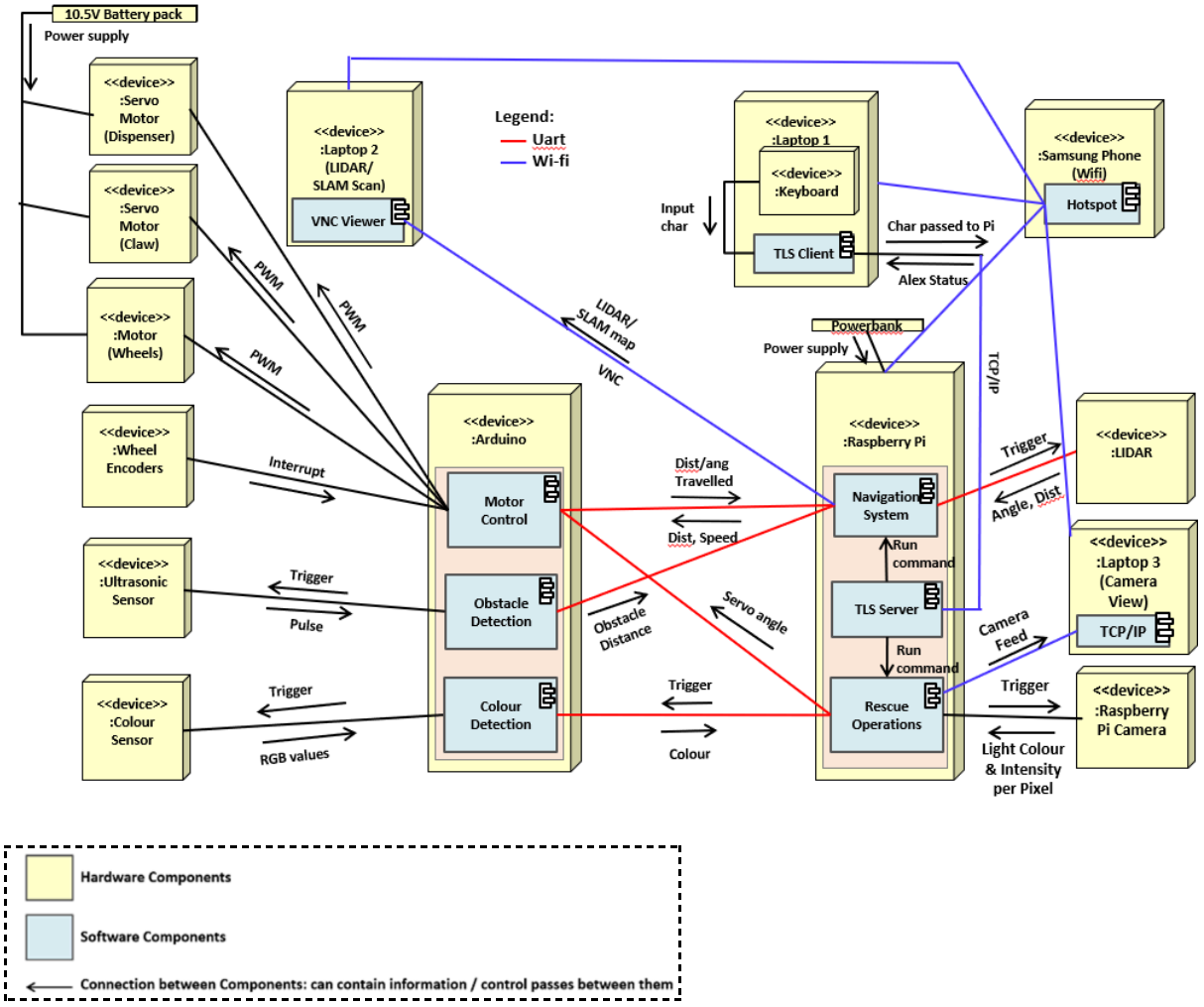


Figure 1: System architecture of Alex

Section 4 Hardware Design

4.1 Placement of components

Figures below presents the fully assembled system, with numbered callouts marking each hardware component. These numbers correspond directly to the “No.” column in the following table, descriptions are provided for each hardware component.

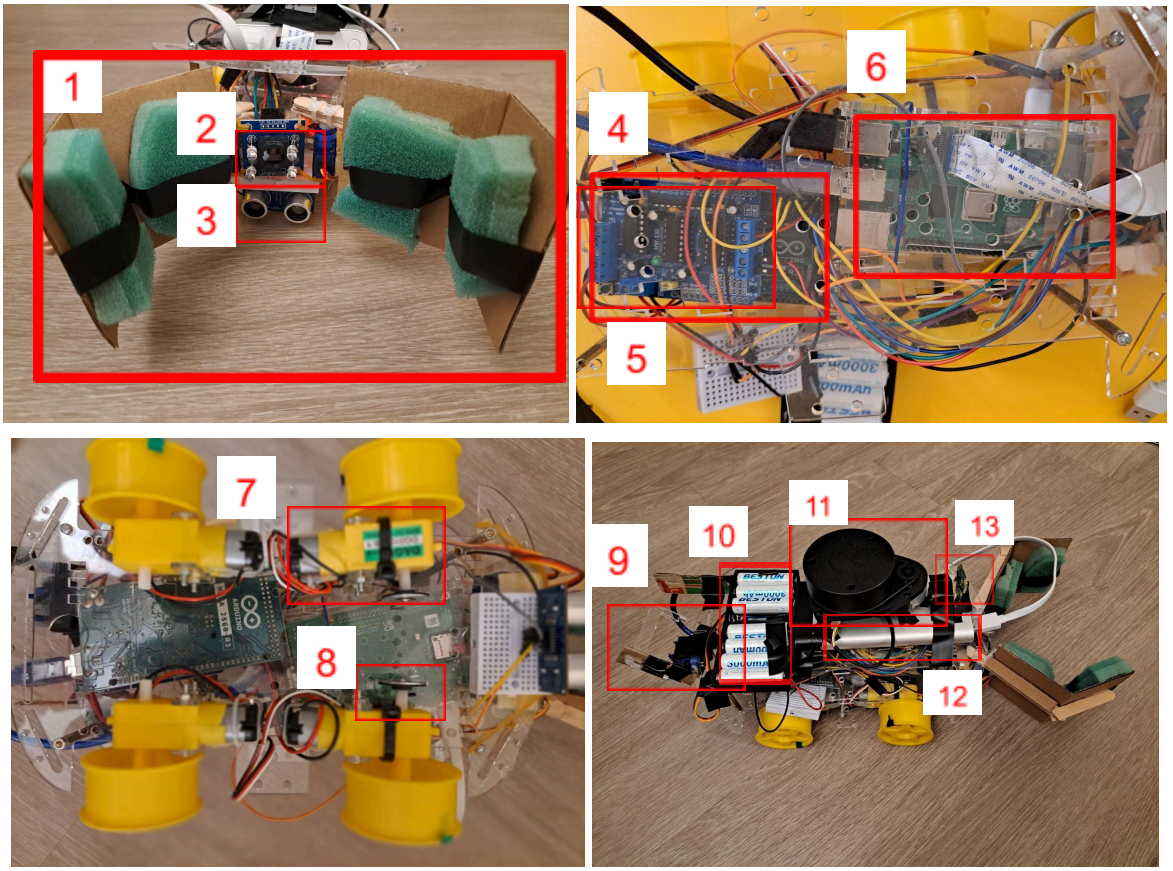


Figure 2: Images of Alex with labelled components (Front/Top/Bottom/Side)

No.	Name	Description
1	Claw	To grab the red astronaut back to the safe zone. Also used to pull astronauts closer to the colour sensor to determine colour of the astronaut. Attached at the front of Alex
2	Colour Sensor	Colour sensor attached at the front to determine the colour of the astronaut.
3	Ultrasonic Sensor	Additional component added to detect the distance between Alex and objects in front, this is to prevent collision. Mounted on the bottom to save space while accurately detecting the distance.
4	Motor Shield	Control the direction and speed of the motors, mounted on the Arduino Mega.
5	Arduino Mega	Controller for external devices. Placed at the back of the board so that wires could be connected to all components on the bot.
6	Raspberry Pi	Main controller for Alex. It is placed at the front such that the Rpi camera can be attached on the top facing forwards.

7	Motors	Mounted at the bottom of Alex.
8	Encoders	Zip-tied to the front motors, track the number of rotations through interrupts to the Arduino.
9	Dispenser	To release Medpak to the green astronaut by lowering the ramp. Mounted at the back of Alex.
10	10.5V Battery Pack	To power the motor shield that powers motors.
11	LiDAR	LiDAR connected to the Rpi, placed at the top such that no component on alex blocks its laser.
12	Power bank	Used to power the Rpi
13	Camera	Used to see objects in front of Alex when activated.

4.2 Additional hardware features:

4.2.1 Ultrasonic sensor

The ultrasonic sensor is mounted on a breadboard at the front of Alex which returns the distance between the obstacle and Alex when Alex approaches an obstacle. The trigger pin tells the sensor to emit ultrasonic sound waves, which will be received by the sensor which sends the duration travelled by the wave through the echo pin. The distance can then be calculated in the code. The ultrasonic is mounted at the bottom of Alex to save space while also checking if the colour sensor is detecting the astronaut at a close enough distance.

4.2.2 Colour sensor

The Cytron Colour Sensor Module is placed at the front of Alex to assist in identifying the Astronauts (Red/Green). The sensor converts the readings from the photodiodes into square waves. From here, we read in the respective frequency from the Red, Green and Blue photodiodes to determine the colour of the object. We will elaborate more on how these values determine the colour of the object in *6.2 Colour detection.*

4.2.3 Servo motors

To complete our runs successfully, we would have to deliver a medpak to the green astronaut and drag the red astronaut to safety. To accomplish our goals, we have installed 2 servo motors attached at the front for the claw, to simulate the opening and closing of the claw. This would allow the claw to be able to grab onto the astronaut for detection of colour, and drag the astronaut if we identify it as the red astronaut. A servo motor is attached at the back as a lever to dispense the medpak when the lever is lowered.

4.3 Other hardware considerations:

4.3.1 Cable management

By using black tape, we made sure that the wires were taped together with their ports. This was to minimise the possibility of loose wiring that could affect Alex's operations. We also made use of the small breadboards as a connector for the wires. This helped us to ensure that the wiring was neat and would not be tangled up with other components on Alex.

4.3.2 Sponging the claw

Our servo motors for the claw are separated by our colour sensor. This makes grabbing the astronaut slightly more challenging due to the additional width of the colour sensor. To address this issue, we decided to add sponge layers to our claw. This would give a firmer grip on the astronaut while also allowing the claw to fully close even if the sponge is already in contact with the astronaut since the sponge can be compressed.

4.3.3 Manoeuvrability

To improve Alex's maneuverability, careful attention was given to weight distribution, motor power, and wheel traction. The power bank was placed vertically at the front half of the top chassis while the 10.5V battery pack was placed horizontally at the back, keeping the center of gravity near the back half of alex for better pivoting. This would be further explained in *7.1 Lesson learnt*. The LIDAR was mounted on top of the power bank to avoid laser interference. Tyres were removed as the bare plastic wheels provided better traction on the test surface.

4.3.4 Small Sidewall

A small sidewall is placed on the back of Alex that helps to keep the medpak in place. Since the medpak is mounted onto Alex, there is a possibility of the mepak flying off Alex during its pivots. To eliminate such possibilities, the sidewall is taped down to keep the medpak between the servo motor and the sidewall.

4.3.5 Rpi camera

To ensure that the camera was giving an optimal view of what was in front of Alex, we decided to tape down the camera to the base of the LIDAR. This gives the camera a top forward view of Alex, without affecting the lasers from the LIDAR.

Section 5 Firmware Design

5.1 High Level Algorithm

After the initial handshake process between the RPi and the Arduino, the Arduino continuously polls for packets from the RPi and responds to them according to the algorithm below.

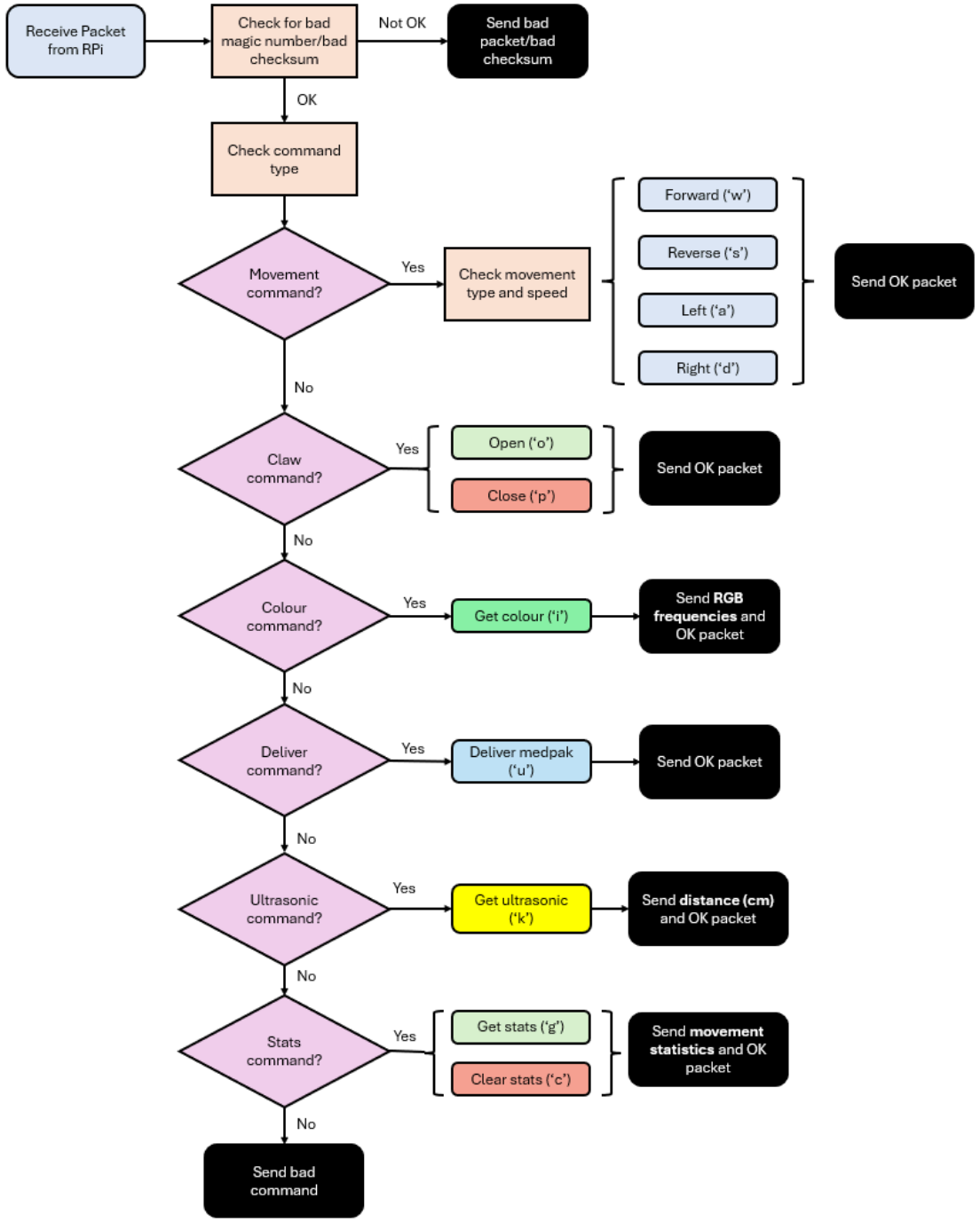
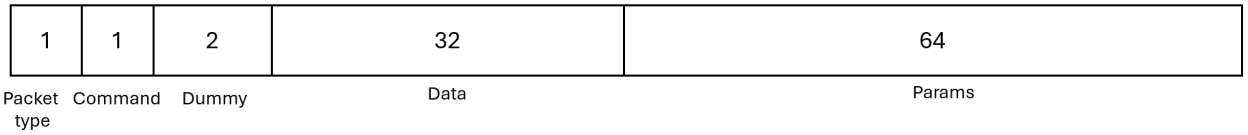


Figure 3: High level algorithm for firmware on Arduino Mega

5.2 Communication Protocol

UART was used for the serial communication between the RPi and the Arduino. The message format used was 8N1 with a baud rate of 9600. The packet format is as shown below.



Each packet contains the packet type, command, data which is used when sending strings, params when sending movement commands, as well as 2 dummy bytes to pad the structure. The two main packet types used by Alex are the command and response packets. The command and response types are detailed below.

Command Types	Response Types

`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_OPEN = 7,`
`  COMMAND_CLOSE = 8,`
`  COMMAND_COLOUR = 9,`
`  COMMAND_DELIVER = 10,`
`  COMMAND_SONIC = 11`
`} TCommandType;`
	`typedef enum`
	`{`
	`  RESP_OK = 0,`
	`  RESP_STATUS=1,`
	`  RESP_BAD_PACKET = 2,`
	`  RESP_BAD_CHECKSUM = 3,`
	`  RESP_BAD_COMMAND = 4,`
	`  RESP_BAD_RESPONSE = 5`
	`} TResponseType;`

5.3 Additional firmware

5.3.1 Servos

The 3 servos used for the claw and the dispenser require a PWM signal of 50 Hz and to do so, we set up the corresponding timers as shown in Appendix 6. When we send a command to the Arduino to open/close the claw, or dispense the medpak, the duty cycle of the PWM signal is adjusted accordingly. The change in the duty cycle alters the servo angle, allowing it to perform the desired action.

5.3.2 Ultrasonic sensor

The ultrasonic sensor operates by emitting sound waves and measuring the time it takes for the waves to reflect off a surface and return to the sensor. To measure this time, we used the `pulseIn()` function, which provides the duration in microseconds (μs). This duration represents the total time for the sound to travel to the surface and back—effectively twice the actual distance between the sensor and the object. To calculate the one-way distance, we used the following formula:

$$Distance\ (in\ cm)\ =\ \frac{duration\ (in\ \mu s) \times 0.034}{2}$$

where 0.034 is the speed of sound in cm/μs.

Section 6 Software Design

6.1 Teleoperation

In our teleoperation setup, three laptops and the Raspberry Pi all join the same phone hotspot. To secure communications, we generate a CA certificate plus server and client keys/certs with OpenSSL and configure the Pi as a TLS server using the libssl APIs provided in the studio.

On the teleoperator's laptop, a client program polls single-keystrokes via getKeypress(), which disables canonical mode and echoing with termios so it can read 'w', 'a', 's', 'd' (and other keys) without pressing Enter. Each keystroke maps to a direction command plus distance/angle and speed, packs into a NET_COMMAND_PACKET, and is sent over the established TLS connection to the Pi server. This will bring an intuitive, video-game-style control scheme requiring no extra keystrokes, allowing the teleoperator to control quickly when looking at the generating map. The table below outlines the key mappings in detail. The complete implementation code can be found in Appendix 5.

Command	Key	Description
Move Forward	w	Sends a move forward command to the RPi, which will then send it to the Arduino. Uses preset distance of 5 cm and power of 70%, which have enough power to start the motor while ensuring smooth movement control.
Turn Right	a	Sends a turn right command to the RPi, which will then send it to the Arduino. Uses preset angle of 30 degrees and power of 70%.

Move Backward	s	Sends a move backward command to the RPi, which will then send it to the Arduino. Uses preset distance of 5cm and power of 70%.
Turn Left	d	Sends a turn left command to the RPi, which will then send it to the Arduino. Uses preset angle of 25 degrees and power of 70%.
Change mode	m	Toggles between WASD control mode and normal typing command mode where parameters can be manually entered.
Clear Stats	c	Sends a command to reset all movement statistics tracked by the Arduino.
Get Stats	g	Retrieves and displays current movement statistics from the Arduino, including tick counts and distances.
Get Colour	i	Activates the color sensor and returns the detected color information.
Get Ultrasonic	k	Activates the ultrasonic sensor and returns the distance detected.
Open Claw	o	Sends a command to open the robot's claw.
Close Claw	p	Sends a command to close the robot's claw.
Deliver Medpak	u	Send a command to deliver medpak.
Exit Program	q	Terminates the client program and closes the connection to the RPi server.

6.2 Colour Detection

From 4.2.2, we read in the frequencies of the Red, Green and Blue photodiodes. Using the raw values of the frequencies, we are able to determine the colour of the astronaut. Since we know that the higher the frequency, the more of that colour is present. We can then deduce the colour of the astronaut through the raw values as the Green value will stand out when detecting the Green astronaut, likewise with the Red value when detecting the Red astronaut.

6.3 LiDAR Mapping

To enhance the accuracy of our SLAM implementation, we developed a data filtering system for the LiDAR. Since our robot moves at higher speeds, raw sensor data often contains noise that negatively impacts mapping quality. Our filter employs three critical thresholds: a minimum quality value (100) to eliminate low-confidence measurements, a minimum distance threshold (150mm) to filter out internal

reflections or dust particles near the sensor, and a maximum distance limit (6000mm) to exclude readings exceeding the maze. This approach ensures only reliable data points contribute to the mapping process.

The filtering system is integrated directly into the LiDAR scan thread, processing raw data before publication to the SLAM algorithm. Additionally, we implemented a validation check to ensure only scans with a sufficient number of valid points (>10) are used for mapping. This prevents the SLAM algorithm from processing sparse or incomplete data that could lead to inaccuracies. Our testing demonstrated that this filtering approach significantly reduced noise, improved processing speed by decreasing computational load, and enhanced overall map quality, particularly when the robot was moving at higher speeds through the maze environment. The functions used to implement this are in Appendix 7.

Section 7 Lessons Learnt - Conclusion

7.1 Lessons Learnt

One lesson we learnt was the importance of checking the functionality of hardware components once we received it. After the demo run, we tried to figure out what was causing our Alex to pivot away from the centre. After testing out different motors, we realised that the issue lied in the motor shield rather than the motor itself. As only the front right port supplied around 3V while the remaining port supplied over 5V, this caused the front right motor to rotate at a lower speed despite our code specifying that all wheels should have the same speed (70%). To solve this issue, we decided to shift the centre of gravity slightly to the back of Alex such that our software code is kept consistent while allowing Alex to pivot on the spot. This experience showed us the possibility of hardware components not working as expected and thus we should always check our hardware components for defects.

Another lesson we learned is the importance of designing solutions specifically tailored to environment constraints rather than developing generic approaches. By studying the maze dimensions and understanding the high speed movement of Alex, we designed a LiDAR filter to remove all returns below 150 mm (near-sensor noise) and above 6000 mm (outside the maze), which dynamically improved SLAM stability and map accuracy. We also designed our robotic arm specifically to match the narrow corridors in the maze, ensuring it could pivot and reach without collisions. By matching our design to the real maze and other project specifications, we got smoother maps and fewer crashes in the actual run.

7.2 Mistakes Made

One of the biggest mistakes made was not testing Alex in an environment similar to the demo run. We validated core functions like movement and LiDAR mapping on open floors, but the real maze was different. This led to unexpected issues during the demo, including noisy LiDAR readings due to the high-speed movement, imprecise movements caused by surface differences and our claw knocking over obstacles in the maze. After spotting these issues, we ran more trials under maze conditions, tweaked our filter thresholds, motion code and claw initial state, which helped us complete all tasks successfully in the final run.

Another mistake was the lack of planning in mounting Alex's components onto the chassis. Our team focused heavily on coding, resulting in less attention being paid to the hardware layout. The messy placement of the RPi and the Arduino made it difficult for us to mount the other sensors like the ultrasonic sensor and the Pi camera in a suitable position such that it would be able to detect obstacles effectively. Eventually we found suitable positions to mount these sensors but it was difficult to route the wires in an organised manner, which led to them being caught in the motors during our testing. To overcome this, we used tape to secure the wires to the chassis which also helped ensure they did not disconnect during our run. If we had planned further ahead for the hardware placement, we would not have faced as many issues as we did.

References

Marques, Carlos & Cristóvão, João & Alvito, Paulo & Lima, Pedro & Frazão, João & Ribeiro, Isabel & Ventura, Rodrigo. (2007). A search and rescue robot with tele-operated tether docking system. Industrial Robot: An International Journal. 34. 332-338. 10.1108/01439910710749663. https://www.researchgate.net/publication/235249986_A_search_and_rescue_robot_with_tele-operated_tether_docking_system

Lima, P. U. (2014, May 25). *Raposa Ng – a search and Rescue Land Wheeled Robot*. euRobotics. https://eu-robotics.net/raposa-ng-a-search-and-rescue-land-wheeled-robot/

Bindu, R. A., Neloy, A. A., Alam, S., Siddique, S., Electrical and Computer Science Department,North South University, & TF-ROS Lab, North South University. (2021). 3-Survivor: A Rough Terrain Negotiable Teleoperated Mobile Rescue Robot with Passive Control Mechanism. In North South University [Journal-article] https://arxiv.org/pdf/2003.05224

Eleven Media. (2025, 21 April). Myanmar earthquake has left 3,735 dead, 5,108 injured, and 120 missing to date https://asianews.network/myanmar-earthquake-has-left-3735-dead-5108-injured-and-120-missing-to-date/

Appendix 1: Bare-metal UART Code

```
/*
 * Setup and start codes for serial communications
 *
 * Alex's setup and run codes
 *
 */

void setBaud(unsigned long baudRate) {
  unsigned int b;
  b = (unsigned int) round(F_CPU / (16.0 * baudRate)) -1;
  UBRR0H = (unsigned char) (b >> 8);
  UBRR0L = (unsigned char) b;
}

void setupSerial()
{
  PRR0 &= ~(1 << PRUSART0); // make sure UART power is on
  setBaud(9600); // set baud rate
  UCSR0C = 0b00000110; // using Async UART, 8N1
  UCSR0A = 0; // clear while setting up
}

// Start the serial connection. For now we are using
// Arduino wiring and this function is empty.

void startSerial()
{
  UCSR0B = 0b00011000; // enable RX and TX (disable bits 6 and 7 -
TX and RX interrupts)
}

// Read the serial port. Returns the read character in
// ch if available. Also returns TRUE if ch is valid.

int readSerial(char *buffer)
{
  int count=0;
  // poll RXC0 bit in UCSR0A
  while((UCSR0A & (1 << RXC0))) {
    // read data from buffer
```

```
    buffer[count++] = UDR0;
  }
  return count;
}

// Write to the serial port

void writeSerial(const char *buffer, int len)
{
  for (int i = 0; i < len; i++) {
    // poll UDRE bit in UCSR0 - if 1 means UDR0 is empty - can
write to it
    while (!(UCSR0A & (1 << UDRE0)));
    UDR0 = buffer[i]; // write each byte of the buffer
  }
}
```

Appendix 2: Ultrasonic Code

```
#define ULTRASONIC_TRIG (1 << 1)
#define ULTRASONIC_ECHO (1 << PD0)

void setupUltrasonic() {

  DDRC |= ULTRASONIC_TRIG; // Set trig pin to output
  DDRH &= ~ULTRASONIC_ECHO; // Set echo pin to input
  PORTC &= ~ULTRASONIC_TRIG; // Clear trig pin
}

void ultrasonic() {
  DDRC |= ULTRASONIC_TRIG;
  delayMicroseconds(10);
  DDRC &= ~ULTRASONIC_TRIG;
  long duration = pulseIn(21, HIGH);
  int dist = duration * 0.034 / 2;
  if (dist == 0) dist = 1000;
  dbprintf("Approaching wall: %d", dist);

}
```

Appendix 3: Colour Sensor Code

```
#include "constants.h"
#include "packet.h"

// Defining Ports used for the Colour Sensor (they are not all the
same Port Data Register)
#define COLOUR_S0 (1 << PB2) // Port B
#define COLOUR_S1 (1 << PB1) // Port B
#define COLOUR_S2 (1 << PA4) // Port A
#define COLOUR_S3 (1 << PA6) // Port A
#define COLOUR_READER (1 << PD1) // Port D
#define COLOUR_ON (1 << PA7) // Port A
#define COLOUR_LED (1 << PB3) // Port B
#define COLOUR_GROUND (1 << PA5) // Port A

void initializeColourSensor() {
  // Set the reading pins to be input and the S pins and power
source to be output pins
  DDRB |= COLOUR_S0 | COLOUR_S1 | COLOUR_LED;
  DDRD &= ~(COLOUR_READER);
  DDRA |= COLOUR_S2 | COLOUR_S3 | COLOUR_ON | COLOUR_GROUND;

  // Set the frequency of the colour sensor to be at 20%
  PORTB |= COLOUR_S0;
  PORTB &= ~(COLOUR_S1);

  PORTA |= COLOUR_ON;
  PORTA &= ~COLOUR_GROUND;

  // Turn off the colour sensor on start up
  PORTK &= ~(COLOUR_LED);
}

long getAvgReading() {
  long total = 0;
  for (int i = 0; i < 50; i++) {
    total += pulseIn(20, LOW);
  }
  return total / 50;
}
```

```
void setColour(Tcolour colour) {
  switch(colour)
  {
    case RED:
      PORTA &= ~COLOUR_S2;
      PORTA &= ~COLOUR_S3;
      break;

    case GREEN:
      PORTA |= COLOUR_S2;
      PORTA |= COLOUR_S3;
      break;

    case BLUE:
      PORTA &= ~COLOUR_S2;
      PORTA |= COLOUR_S3;
      break;

    case WHITE:
      PORTA |= COLOUR_S2;
      PORTA &= ~COLOUR_S3;
      break;
  }
}

void readColour() {
  // Turn on the LED
  PORTB |= COLOUR_LED;
  delay(10);
  setColour(WHITE);
  PORTA |= COLOUR_S2;
  PORTA &= ~COLOUR_S3;
  long control = (double) getAvgReading();
  long rgbArr[3] = {0};
  dbprintf("white: %lu", control);

  // Cycle between red green and blue
  setColour(GREEN);
  rgbArr[0] = getAvgReading();

  setColour(BLUE);
  rgbArr[1] = getAvgReading();
```

```
  setColour(RED);
  rgbArr[2] = getAvgReading();
  dbprintf("Green raw: %ld", rgbArr[0]);
  dbprintf("Blue raw: %ld", rgbArr[1]);
  dbprintf("Red raw: %ld",rgbArr[2]);

  // Turn off colour sensor
  PORTB &= ~(COLOUR_LED);
}
```

Appendix 4: Server Code

Server Command Handling

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

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

	TPacket commandPacket;

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

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

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

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

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

		case 'r':
```

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

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

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

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

        case 'o':
        case 'O':
            commandPacket.command = COMMAND_OPEN;
            uartSendPacket(&commandPacket);
            break;

        case 'p':
        case 'P':
            commandPacket.command = COMMAND_CLOSE;
            uartSendPacket(&commandPacket);
            break;

        case 'i':
        case 'I':
            commandPacket.command = COMMAND_COLOUR;
            uartSendPacket(&commandPacket);
            break;
```

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

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

      }
}
```

Appendix 5: Client Code

Network Packet Types

```
typedef enum {
    NET_ERROR_PACKET = 0,
    NET_STATUS_PACKET = 1,
    NET_MESSAGE_PACKET = 2,
    NET_COMMAND_PACKET = 3,
} TNetConstants;
```

Server Command Handling

```
#include <unistd.h>
#include <termios.h>

// Variables for different gears
int speed = 70;
int distance = 5;
int angle = 30;
int mode = 0;
int readyToReceive = 1; //1 if Arduino is ready to receive command

// Read a single character from terminal without a newline
// character. Done by modifying terminal attributes.
char getKeypress() {
  char key = 0;
  struct termios term_state = {0};
  if (tcgetattr(0, &term_state) < 0)
    perror("tcsetattr()");
  term_state.c_lflag &= ~ICANON; //temporarily disable the
canonical state
  term_state.c_lflag &= ~ECHO; //temporarily disable echo
  term_state.c_cc[VMIN] = 1;
  term_state.c_cc[VTIME] = 0;
  if (tcsetattr(0, TCSANOW, &term_state) < 0)
    perror("tcsetattr ICANNON");
  if (read(0, &key, 1) < 0)
    perror("read()");
  term_state.c_lflag |= ICANON; //re-enable canonical state
  term_state.c_lflag |= ECHO; //re-enable echo
  if (tcsetattr(0, TCSADRAIN, &term_state) < 0)
    perror("tcsetattr ~ICANON");
```

```
  return (key);
}

void writeWASD(void *conn, int *quit){
      printf("Keys: w=forward, s=reverse, d=turn left, a=turn
right\n");
      printf("m = change mode, c=clear stats, g=get stats, i=get
colour, o=open claw, p=close claw, u=deliver medpak, q=exit\n");
      char ch = getKeypress();
      char buffer[10];
      int32_t params[2];
      buffer[0] = NET_COMMAND_PACKET;

      switch (ch)
      {
        case 'w': // go forward
          buffer[1] = 'f';
          params[0] = distance;
          params[1] = speed;
          memcpy(&buffer[2], params, sizeof(params));
          sendData(conn, buffer, sizeof(buffer));
          break;
        case 's': // go backwards
          buffer[1] = 'b';
          params[0] = distance;
          params[1] = speed;
          memcpy(&buffer[2], params, sizeof(params));
          sendData(conn, buffer, sizeof(buffer));
          break;
        case 'a': // go left
          buffer[1] = 'l';
          params[0] = angle;
          params[1] = speed;
          memcpy(&buffer[2], params, sizeof(params));
          sendData(conn, buffer, sizeof(buffer));
          break;
        case 'd': // go right
          buffer[1] = 'r';
          params[0] = 25;
          params[1] = speed;
          memcpy(&buffer[2], params, sizeof(params));
          sendData(conn, buffer, sizeof(buffer));
```

```
      break;
    // todo: commands
    case 'c':
      buffer[1] = 'c';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      buffer[1] = ch;
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'g':
      buffer[1] = 'g';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      buffer[1] = ch;
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'i':
      buffer[1] = 'i';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      buffer[1] = ch;
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'o':
      buffer[1] = 'o';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      buffer[1] = ch;
      sendData(conn, buffer, sizeof(buffer));
      break;
    case 'p':
      buffer[1] = 'p';
      params[0] = 0;
      params[1] = 0;
      memcpy(&buffer[2], params, sizeof(params));
      buffer[1] = ch;
      sendData(conn, buffer, sizeof(buffer));
      break;
```

```
            case 'u':
            buffer[1] = 'u';
            params[0] = 0;
            params[1] = 0;
            memcpy(&buffer[2], params, sizeof(params));
            buffer[1] = ch;
            sendData(conn, buffer, sizeof(buffer));
            break;
    case 'k':
            buffer[1] = 'k';
            params[0] = 0;
            params[1] = 0;
            memcpy(&buffer[2], params, sizeof(params));
            buffer[1] = ch;
            sendData(conn, buffer, sizeof(buffer));
            break;
        case 'm':
            printf("Changing mode to TYPING COMMAND...\n");
            mode = 0;
            break;
        case 'q':
        case 'Q':
            *quit=1;
            break;
        default:
            printf("BAD COMMAND\n");
        }
}
```

Appendix 6: Servo Code

```
#include "constants.h"
#include "packet.h"

#define LEFT_CLAW (1 << PB4)
#define RIGHT_CLAW (1 << PL4)
#define DISPENSER (1 << PH6)

volatile int servoTimerTick = 0;
int claw_PWM;
int dispenser_PWM;

ISR(TIMER2_COMPA_vect) {
  // Link CTC clock to generate PWM for left claw and the
dispenser
  servoTimerTick++;
  if (servoTimerTick >= claw_PWM) {
      PORTB &= ~(1 << PB4);
  }
  if (servoTimerTick >= dispenser_PWM) {
    PORTH &= ~DISPENSER;
  }
  if (servoTimerTick >= 200) {
      servoTimerTick = 0;
      PORTH |= DISPENSER;
      PORTB |= LEFT_CLAW;
    }
}

void openservo() {
  OCR5B = 1500;
  claw_PWM = 15;
}

void closeservo() {
  OCR5B = 2100;
  claw_PWM = 9;
}

void dispense() {
  if (dispenser_PWM == 11) dispenser_PWM = 22;
```

```
  else dispenser_PWM = 11;
}

void setupservo() {
  // set servo pins to output
  DDRL |= RIGHT_CLAW;
  DDRB |= LEFT_CLAW;
  DDRH |= DISPENSER;

  //setup pwm for right claw
  TCCR5A = 0b00100010;
  TCNT5 = 0;
  TCCR5B = 0b00010010;
  ICR5 = 40000;
  OCR5B = 1500;

  // set up CTC timer for dispenser and left claw
  TCCR2A = 0b00000010;
  TCCR2B = 0b00000100;
  TIMSK2 = 0b10;
  OCR2A = 24;
  OCR2B = 24;
  TCNT2 = 0;
  claw_PWM = 15;
  dispenser_PWM = 11;
}
```

Appendix 7: LIDAR Scan Code

```
def filter_lidar_scan(scan_data):
    """Filter out low-quality or invalid LiDAR scan points."""
    # For RPLidar A1M8, extracting data
    angles, distances, qualities = scan_data

    # Create arrays for filtered data
    filtered_angles = []
    filtered_distances = []
    filtered_qualities = []

    # Filter based on quality and distance thresholds
    MIN_QUALITY = 100     # Start with this and adjust based on observation
    MIN_DISTANCE = 150  # 15 cm - filter out very close readings (noise)
    MAX_DISTANCE = 6000  # 6 m - filter out very large readings (outside maze)

    for i in range(len(angles)):
        if (qualities[i] >= MIN_QUALITY and
            MIN_DISTANCE <= distances[i] <= MAX_DISTANCE):
            filtered_angles.append(angles[i])
            filtered_distances.append(distances[i])
            filtered_qualities.append(qualities[i])

    return (filtered_angles, filtered_distances, filtered_qualities)

def lidarScanThread(setupBarrier:Barrier=None, readyBarrier:Barrier=None):

"""...Same code from template…"""

    # Receiving Logic Loop
    try:
        scan_generator = lidar.start_scan_express(scan_mode)
        current_round = {"r":0, "buff":[], "doScan":False}
        for count, scan in enumerate(scan_generator()):
            current_round, results = process_scan((count,scan),
scanState=current_round)
            if results and current_round["r"] > INITIAL_ROUNDS_IGNORED:
                # Filter scan data before publishing
                filtered_angles, filtered_distances, filtered_qualities =
filter_lidar_scan(results)

                # Only publish if we have enough valid points
                if len(filtered_angles) > 10:
                    publish(LIDAR_SCAN_TOPIC, (filtered_angles, filtered_distances,
filtered_qualities))

"""...Same code from template…"""
```