CG2111A Engineering Principle and Practice

Semester 2 2024/2025

"Alex to the Rescue"

Final Report

Team: B02-6B

Name	Student #	Main Role
Karthikeyan Vetrivel	A0307459W	Communications and Command Issuing
Joshua Yeo Wee Tze	A0309329Y	Hardware and Firmware Tuning
Rajeshprabu Sidharth	A0302676A	Hardware and Movement
Goh Shao Ann	A0307779L	Lidar/SLAM

Table of Contents

1. Introduction

Our robot "Alex" is designed to simulate the core functionalities of a search and rescue robot. Teleoperated via a wireless connection, the operator can move Alex through the simulated lunar environment of "Moonbase CEG".

Our Alex robot would be placed in a room filled with various obstacles such as walls, boxes, and tables. As operators, we would teleoperate Alex to locate two astronauts, one marked red and one marked green. Upon finding the red astronaut (injured and trapped), Alex would use its actuator claw to perform a simulated rescue, Alex would then proceed to move the astronaut to a "parking spot" where the astronaut will be attended to. For the green astronaut (healthy but trapped), Alex would deliver a medpak. Finally, Alex has to return back to the "parking spot", completing its objective in the simulated lunar environment.

The system is powered by a Raspberry Pi and an Arduino Mega. The Pi, acting as the central controller, receives operator commands from a laptop over TLS, processes LIDAR data for Breezy-SLAM-based mapping, and forwards motor and sensor commands to the Arduino via UART. The Arduino executes these commands, controls actuators, and returns sensor feedback including ultrasonic distance and colour detection. The operator uses a TLS-secured client to issue commands and leverages ROS topics to visualize LIDAR data and SLAM maps in real-time.

2. Review of State of the Art

Among the many modern designs for Search and Rescue (SAR) robots, these two stood out to us due to their unique capabilities and design.

2.1 Raposa

RAPOSA is a semi-autonomous robot developed for SAR missions primarily in hazardous outdoor environments, such as collapsed structures and debris fields. It consists of a variety of sensors to help accomplish this mission, namely three conventional cameras, a thermal camera, explosive and toxic gas sensors, temperature and humidity sensors, inclinometers, artificial lighting, a microphone, and speakers. RAPOSA is primarily controlled through teleoperation, using a wireless link between the robot and a remote console operated via a conventional GUI and gamepad. The robot runs its control system on an onboard computer and supports both wireless and tethered communication (Silva, Almeida, & Lima, 2006).

Strengths	*Weaknesses*
Compact and Rugged: Can fit into tight spaces and take a beating.	Low Autonomy: Primarily manually controlled, vulnerable to signal distributions
Rich Sensor Suite: Thermal, gas, and environmental sensors onboard.	No Actuators: Can't move objects or interact with surroundings.
Dual Communication Modes: Switches between wireless and tethered	Limited Range: Range depends on signal strength or tether length.

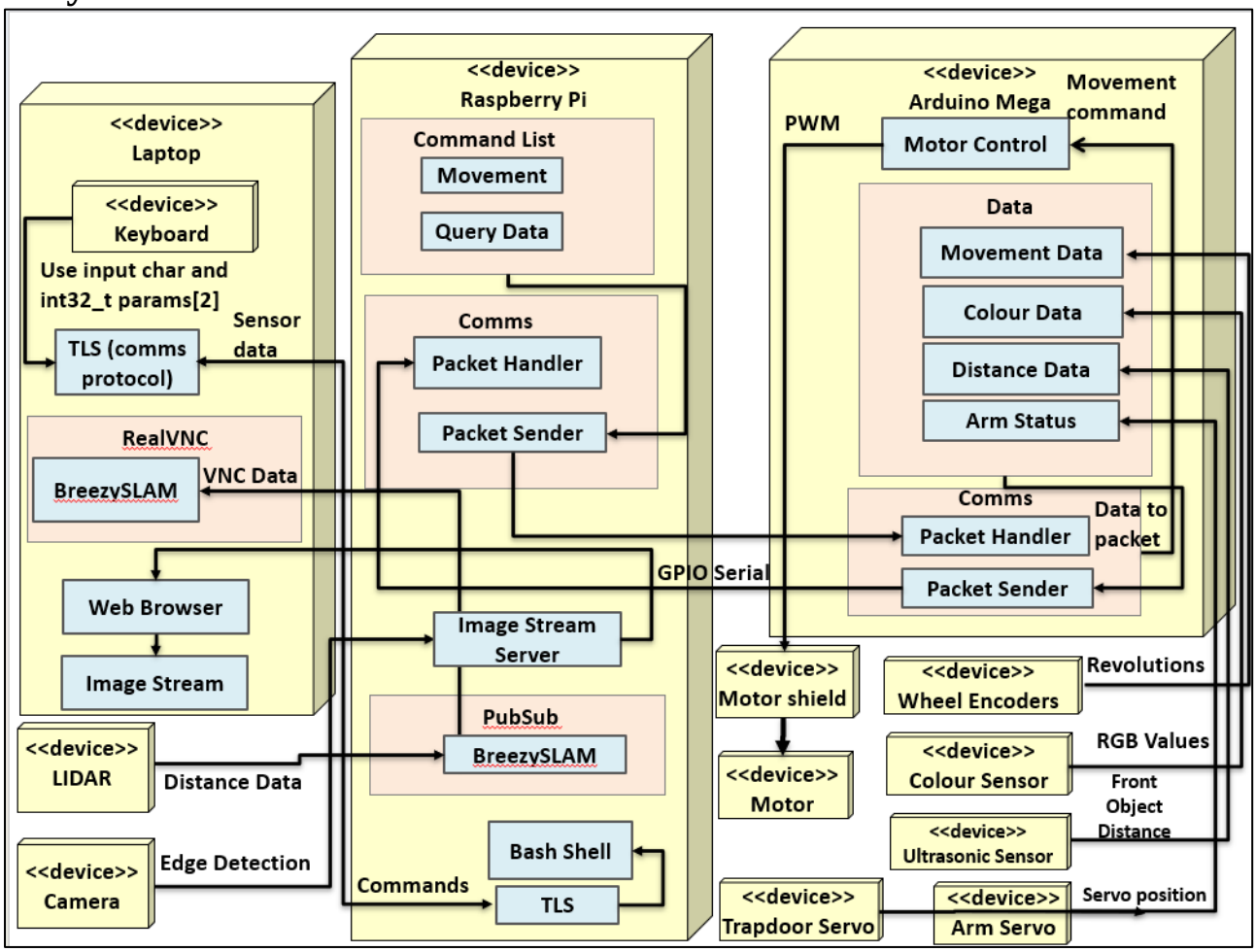
2.2. Earthshaker

Earthshaker is a ground robot built for search and rescue in urban, hazardous, and rubble-filled environments. It was the winning entry in the first Advanced Technology and

Engineering Challenge (A-TEC), demonstrating strong disaster response capabilities. Teleoperation is achieved via a tri-modal communication setup between the Operator PC and the onboard Intel NUC, using a 1.8 GHz MIMO mesh radio, two AT9S transmitters, and a 4G/5G router. Its software enables automatic switching between modes based on signal quality. A STM32 control board handles low-level communication between sensors and actuators, while the system is managed remotely through a standard operator interface. Earthshaker includes a front dozer blade for moving obstacles up to 75 kg and features IP64-rated protection for wet conditions (Zhang, 2023).

Strengths	*Weaknesses*
Water-resistant (IP64)	Battery lasts only 3 hours
Supports multiple communication modes	Struggles with vertical axis navigation
Equipment Dozer blade can push objects up to 75 kg	Large size limits movement in tight spaces

3. System Architecture



Key:

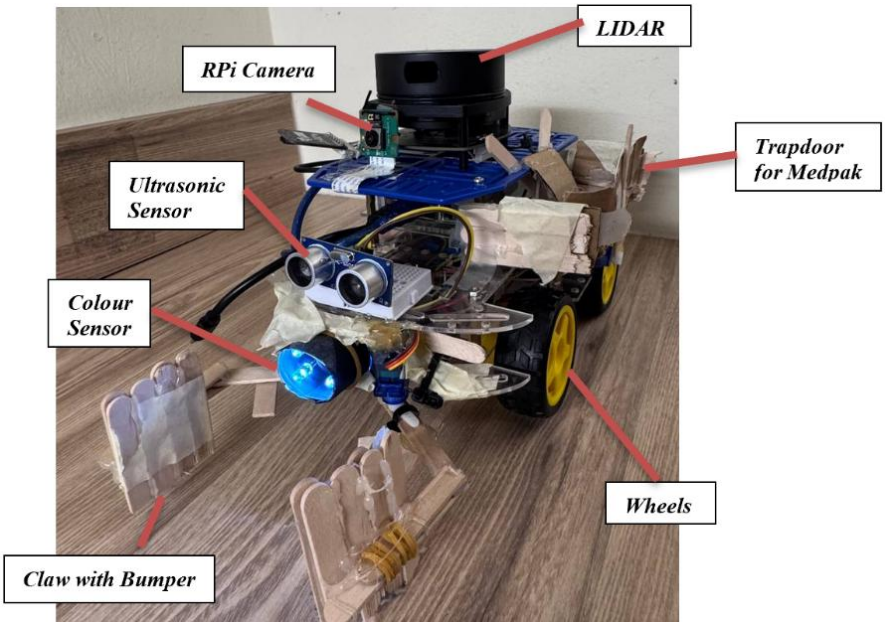
Software

Hardware

4. Hardware Design

4.1. Final Form of the System

The Alex robot was built with an RPi chassis plate and a three-layer 4WD chassis, consisting of 3 acrylic boards. The battery casing is slotted in between the acrylic boards at the rear of the chassis for easy access. Zip ties and masking tape are used to secure the components on Alex's chassis.



4.2. Non-Standard Hardware Components

Ultrasonic Sensor

The Ultrasonic Sensor played two key roles in our mission. Firstly, it helped measure the distance between the robot and the astronaut during the grabbing phase, ensuring we approached carefully without causing a collision. Secondly, it was used to verify that the robot maintained a secure grip on the astronaut while moving.

Due to the accuracy of our LIDAR in detecting obstacles, the Ultrasonic Sensor was not used to detect obstacles in front of Alex. Rather, we used it to approximate the proximity between the astronaut and the robot. The software implementation of the Ultrasonic Sensor will be covered in greater detail in Section 6.6.

Replacement Batteries

With many components running simultaneously, which included servos, motors, sensor and LIDAR, the robot required sufficient power to operate reliably. Therefore, with replacement batteries, we were able to quickly replace drained batteries and continue testing, while the rechargeable batteries were being charged. This was particularly important during long working sessions, where any downtime could disrupt momentum and slow progress.

4.3. Claw with Bumper

The claw mechanism consists of a pair of SG90 micro servos controlled through PWM pins 45 and 46 of the Arduino Mega. The servos enable the grasping and movement of the red astronaut. The claw is controlled through a keypress on a laptop, where the command is sent securely using TLS to the Raspberry Pi, which then forwards the instruction through UART to the Arduino.

The PWM signals for the servos are generated by Timer 5, set to phase-correct mode to create a signal of 50 Hz that is compatible with SG90 motors. A detailed description of the configuration of the timer and OCR calibration parameters is given in Section 6.5.

During development, it was observed that the red astronaut, since it did not have wheels, tended to topple over when towed. Therefore, bumpers were incorporated at the claw area for improved transportation stability to prevent overturning.

4.4. Trapdoor system for Medpak

In earlier trial runs, the Medpak was kept within the claw during transport. Yet this was a significant drawback as if the red astronaut was first encountered, we could not retrieve it since the claw was already occupied. Therefore, this left us in the dilemma of having to postpone the rescue mission and locate the green astronaut.

To get through this obstacle, we designed a servo-controlled trapdoor mechanism. The Medpak is seated on a platform in front of the robot and secured by a specific servo. When the command is triggered by the operator, the trapdoor is opened and the small ramp in the trapdoor makes the Medpak slide down.

This made it possible to deliver the Medpak independently of the claw, giving us full flexibility to prioritize either astronaut first, without having to reconfigure the robot in operation.

5. Firmware Design

5.1. High Level Algorithm on the Arduino Mega

Setup and Initialization

To setup Alex, the user has to run the tls-alex-server executable on the Raspberry Pi through VNC. First, UART communication between Raspberry Pi and Arduino Mega is established at a baud rate of 9600, 8N1 frame format (8 data bits, no parity, 1 stop bit). This setup causes the Arduino to reboot, initializing the connected peripherals such as motors, ultrasonic sensor, color sensor, encoders, and servo motors. After UART setup, a separate thread (pthread) is created to continuously receive messages from the Arduino through the uartReceiveThread. This ensures asynchronous serial communication between the RPi and the Arduino Mega.
Next, the server is set-up using:

- Alex’s private key and certificate (alex.key, alex.crt),
- CA’s certificate (signing.pem) and;
- Expected Client Name (epp_laptop.com).

This TLS setup ensures secure, authenticated communication between Alex and the client device. Finally, a HELLO packet is sent from the Raspberry Pi to the Arduino to confirm the communication has been established. If the connection is stable and error-free, no "Bad Checksum" or "Bad Packet" errors should appear on the Arduino side.

Polling and Processing of User Command

The Arduino Mega continuously polls for incoming user commands by calling the readPacket() function inside its main loop. This function listens for serial data and attempts to deserialize the received bytes into a TPacket structure. Once a complete packet is received, the deserialized data is returned as a TResult, which indicates whether the packet is valid or contains errors such as "Bad Checksum" or "Bad Packet". If the packet is invalid due to a bad checksum or corrupted data, the Arduino responds with an appropriate error message using

sendBadChecksum() or sendBadPacket(), ensuring reliable communication and user feedback. If the packet is valid (i.e., PACKET_OK) it is passed to the handlePacket() function for further processing. If the packetType field is "PACKET_TYPE_COMMAND", the handlePacket() function calls handleCommand(), where a specific action by the teleoperator is carried out.

Execution

Upon receiving each command from the teleoperator, the Arduino Mega responds with an OK message using sendOK() function. For motor and servo-related commands, the corresponding control functions are called to execute the requested movements. If sensor function is called, the appropriate sensor will send back the necessary data using the sendResponse() function.

On the other hand, if the user requests status, sendStatus() is called where the ticks count for each direction, the distance moved by the robot, as well as the Claw and Trapdoor positions are sent back to user. The user is also able to clear status, which resets both the tick counts, and the distance travelled back to zero.

5.2. Communications Protocol

The UART communication was configured to operate at 9600 bps using the 8N1 frame format (8 data bits, no parity, 1 stop bit). Subsequently, Serial communication was enabled by setting the RXEN0 and TXEN0 bits in the UCSR0B register. (Refer to Appendix 1)

Reading Packet

In the main loop, the Arduino Mega continuously polls to receive commands, which are stored in the variable recvPacket. The code then calls readPacket(), which checks if the received packet is incomplete or ready to be deserialised. This function internally calls readSerial(), passing the address of a buffer array declared in readPacket(). When a command is sent to the Arduino from the RPi, the RXC0 bit in UCSR0A register will be set, indicating that data has been received. Upon calling readSerial() function, we continuously check whether the RXC0 bit is set. If it is, we read the data from the UDR0 register and store it in the buffer. Once all the data is read, the function simply returns the number of bytes written to the buffer (i.e. count).

In readPacket(), if the returned byte count is 0, we can declare as "PACKET_INCOMPLETE", else we can deserialise the buffer into a packet of TPacket datatype using the deserialize() function. Inside deserialize(), we further verify whether the packet received has the correct Magic Number and Checksum, before calling memcpy() function, which converts the TComms packet into output of TPacket datatype. We have now successfully received and reconstructed the TPacket sent from the RPi and can proceed to handle the corresponding teleoperation command.

Writing Packet

Communication from the Arduino to the RPi is through sendResponse(). This function takes a TPacket, serializes it into a stream of bytes using a character array, and then calls writeSerial(). In writeSerial(), we wait for the UDRE0 bit in UCSR0A to be set, indicating that the transmit buffer is empty. Once ready, we write the bytes to the UDR0 register for transmission.

As such, we successfully established a two-way communication between the Arduino and RPi.

5.3. Encoder-based Movement

The two encoders are attached to each back wheel of Alex, which measures the distance in the direction (forwards/backwards) in which the robot has travelled. The distance and angle turned are calculated from the formulas:

- $distance = \frac{ticks}{COUNT\ PER\ REV} \times WHEEL\ CIRC$
- $ticks = \frac{angle}{360} \times \frac{alexCirc}{wheelCirc} \times countsPerRev$

The encoders are connected to the Arduino Mega interrupt pins PD2 and PD3. Based on the direction stored in the variable 'dir', the Interrupt Service Routines (ISR) will increment the corresponding counter (*ticks*). Once Alex has moved/turned the given distance/angle, it will stop the movement command. In addition, we configured INT2 and INT3 to trigger on the CHANGE mode, which allowed us to improve the tick resolution. This change increased the encoder resolution from 1 tick per 5.25 cm to 1 tick per 2.63 cm, effectively doubling the positional accuracy. (Refer to Appendix 2)

5.4. Edge Detection Camera

We attached the Raspberry Pi camera module to the front of the RPLiDAR to detect obstacles and astronauts in front of Alex. The camera module is connected to the RPi using the CSI (Camera Serial Interface) connector and the AlexCameraStreamServer.py file is executed for video streaming. The camera helped detect and confirm obstacles and the identity of the astronauts when Alex is about 5cm or more away from them.

6. Software Design

6.1. High-level Algorithm on the Pi

<u>Pre-run Set-up</u>

These are the set-ups to be completed prior to submitting the robot.

Raspberry Pi:

1. Using the Arduino IDE, we upload the latest code to the Arduino Mega from the Raspberry Pi.
2. Next, we compile the latest version of the tls-alex-server.cpp file, using ./build.sh shell script. Shell scripts reduced the need to manually type long gcc compile commands

Laptop:

1. Due to the DHCP Lease persistence, we can safely assume that our IP addresses will remain the same, so long as we frequently connect the WIFI to the RPi.
2. After updating the most recent IP address on the tls-alex-client.cpp, we compile it using the ./build_client.sh shell script.
3. We used RealVNC to remotely access the RPi and obtain LIDAR data. By saving the RPi's VNC server address in the "Address Book," we ensure quick and easy access during the run.

<u>During Run</u>

These steps are executed during the 5-minute setup window.

Raspberry Pi:

1. Through RealVNC we remotely access the RPi.
2. By running the tls-alex-server executable, we setup server on the RPi.
3. Through `source ./env/bin/activate`, we activate the Python Virtual Environment, before running python AlexCameraStreamServer.py file, and enable the RPi camera.
4. Similarly, we activate the Python Virtual Environment, to set up the LIDAR and SLAM using python alex_main.py. Maximise window for better visibility.

Laptop:

1. Since the server is already running, we use the laptop as the client to establish a secure TLS connection.
2. Upon establishing a successful handshake, we can send teleoperation commands to the server, and receiver sensor data.
3. The RPi's Receiver Thread (handleNetworkData()) receives commands from the client and sends them to the Arduino via UART.
4. The RPi's Sender Thread (sendNetworkData()) receives sensor data from the Arduino and transmits it back to the client.

The entire set-up requires 2 laptops, one for client and RPi camera, and the other for LIDAR and SLAM. (Refer to Appendix 3) After successfully executing the steps above, the client can execute the teleoperation commands in Section 6.2 to control the Alex.

6.2. Teleoperation

Letter	**W**	**A**	**D**	**S**	**T**
Command	Forward (Short Range)	Left	Right	Reverse (Short Range)	E-Stop

Letter	**P**	**L**
Command	Forward (Long Range)	Reverse (Long Range)

Letter	**X**	**M**	**U**	**C**	**G**	**Q**	**J**
Command	Colour Detection	Control Arm	Front Distance	Clear Stats	Get Stats	Quit	Release Medpak

6.3. SLAM and LIDAR

We used BreezySLAM algorithm with the RPLiDAR scans to navigate Alex through the maze and spot the astronauts. The lidar_example_simple_plot.py code generates the surroundings of Alex through the LiDAR scan data. The SLAM algorithm constructs the SLAM map by estimating Alex's location and orientation using the scan data. Appendix 3 shows the LiDAR scans and SLAM maps generated.

We adjusted the dimensions of Alex on the SLAM map (represented by an arrow) under SLAM constants in the display node Python script. This adjustment makes the directional arrow more accurately reflect Alex's actual size and proximity to nearby obstacles and astronauts, helping the driver avoid collisions more effectively.

We also implemented a median filter function to smooth out LiDAR scan data by eliminating anomalies and preserving edges more effectively. The filter reduces noise and random spikes that results in false walls and bad pose estimations. This will help produce more reliable scans and SLAM maps that are cleaner and more stable.

6.4. Colour Detection

When a TPacket command for color detection is received, the TCS230 color sensor measures light intensity for red, green, and blue (RGB) wavelengths using an array of photodiodes with corresponding optical filters. The sensor outputs a square wave signal whose frequency is proportional to the intensity of the detected color.

The system reads each channel (red, green, blue) sequentially by setting the sensor's filter control pins (S2, S3) to the appropriate logic levels (Refer to TCS230 pin configurations table).

The pulseIn() function is then used to measure the pulse width of the output frequency (OUT pin), which is converted into an RGB value. These values are transmitted to the Raspberry Pi and laptop server for final color processing.

S2	S3	Photodiode Type
L	L	Red
L	H	Blue
H	L	Clear (no filter)
H	H	Green

S0	S1	Output Frequency Scaling
L	L	Power down
L	H	2%
H	L	20%
H	H	100%

TCS230 pin configurations

Since the driver of Alex cannot determine its front object proximity during color sensing, an ultrasonic sensor is used concurrently to enforce a maximum distance of 10 cm from the target before scanning. This threshold was derived through repeated testing under the same room conditions. The reason for the decreased accuracy is due to the increasing effect of ambient lighting affecting the color sensor readings at greater distances.

For reliable red/green classification, k-nearest neighbours (KNN) algorithm is employed. Before each mission run, three calibration datasets for both red and green astronauts were collected at different locations in the room. Because ambient lighting and distance affect the TCS230's RGB readings, the raw data is first normalized to mitigate distance-dependent variations. The steps are as follows:

1. Normalization:
    - RGB values are scaled to the range of (0 – 1) to reduce its sensitivity to distance from object.
2. Difference Calculation:
    - The normalized test data is compared against each training sample by computing the absolute difference in RGB components.
3. KNN Matching:
    - The training sample with the smallest difference (i.e., closest Euclidean distance in RGB space) is selected with its associated colour (red/green) being outputted.

6.5. Servo Motors

The code for setting up servo motors and ultrasonic sensor were coded out in bare metal to ensure faster operations. (Refer to Appendix 4) The three servo motors utilized the 16-bit timer 5 to generate PWM signals to toggle the claws and trapdoor of Alex as mentioned in section 4. To tune the servo motors to its desired angles, calculations were made to validate the PWM to its desired angle.

Our group decided to configure timer 5 to have a desired frequency of 50hz for compatibility with the SG90 micro server. The formula is as given:

$$f_{OCnxPCPWM} = \frac{f_{clk_{O/I}}}{N \times 2 \times TOP}$$

where $f_{clk_{O/I}} = 16\text{Mhz}$, $f_{PWM} = 50\text{hz}$, N and TOP are to be determined.

Calculations: Finding Appropriate Prescalar We first took TOP to be the max value OC5x could reach which will be 65535. $\frac{16\times10^6}{N\times2\times65535} = 50hz \rightarrow$ N = 2.44 Since the smallest prescaler available that is greater than 2.44 was 8. ∴ N = 8	Finding TOP value (using prescaler 8) $\frac{16\times10^6}{8\times2\times TOP} = 50hz \rightarrow TOP = 20000$ TOP, i.e. ICR5, had to be set to 20000 to achieve 50hz.

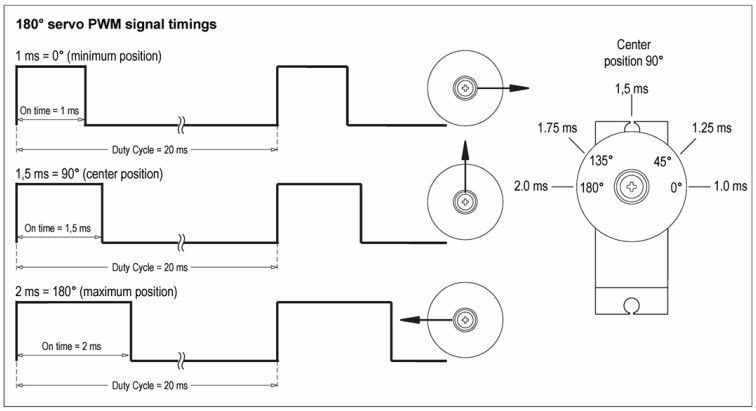


Diagram illustrating SG90 Servo motor's duty cycle

Based on the data above, the calculated minimum and maximum value we can set our values into OCR5x are 1000(1ms) and 2000(2ms) respectively (5-10% PWM output) to deliver a range of 180 degrees. To simulate an opening of the claw, the left and right claws were set to 20(1.11ms) and 160 degrees(1.89ms) respectively and closing claw to 60(1.39ms) and 100(1.56ms) degrees respectively. Similarly, to simulate the opening of trapdoor, the servo has to be set to start at 0(1ms) and 90(1.5ms) degrees.

Finding the appropriate duty cycles						
	Left Servo		Right Servo		Trapdoor	
Initial Pos.	LEFT_SERVO_OPEN	1110	RIGHT_SERVO_OPEN	1890	TRAPDOOR_CLOSE	1000
Final Pos.	LEFT_SERVO_CLOSE	1390	RIGHT_SERVO_CLOSE	1560	TRAPDOOR_OPEN	1500

6.6. Ultrasonic Sensors

Similar to how we used timers to control servos, we use Timer1 (a 16-bit timer) to measure the time taken for an ultrasonic pulse to travel to an obstacle and reflect back. Timer1 can count from 0 to 65535, and suitable for capturing distances between 0-30 cm.

Calculations:
Speed of sound $\approx 343\text{m/s} \approx 0.0343\ \text{cm/µs}$
To be able to measure distance of a range between 0 to 30cm,
Time taken for 30cm $= 2 \times \frac{30}{0.0343} \approx 1749\text{µs}$
From the table on the right, at 0.5 µs per tick, each centimeter is ~58.3 µs → about 117 ticks/cm**.**
∴We choose prescalar value of **8** (provides best resolution and avoids overflow)

Prescalar	Time per tick /µs	Timer count
1	0.0625	27976
8	0.5	3497
64	4	437
256	16	109

Initially, we configure the TCCR1A/B registers and set the TCNT0 register to 0. Then, a 10µs pulse is generated from the Trigger Pin by setting and clearing the PD7 bit in the PORTD register. After sending the pulse, we poll the PC1 bit in the PINC register (which corresponds to the echoPin) to check if the pulse has been received back. Once the pulse received, the Timer stops, and we can use the TCNT1 value to calculate the duration in µs. With the duration, the distance can be measured using the formula below, and this is sent to the client.

$$Distance = \frac{duration \times 0.0343\text{cm/µs}}{2}$$

Refer to Appendix 5 for full bare-metal code.

After extensive testing, we determined that an ultrasonic sensor reading of 3cm or less guarantees a secure grip on the astronaut when closing the claw. As collisions with astronauts are costly, one area of improvement is the potential use of the interrupts available in the Arduino Mega by configuring the EIMSK and EICRA registers directly.

At the start of each loop(), a 10 µs pulse is sent from the trigger pin to initiate a distance measurement. The echo pin is configured to trigger an interrupt on any logical change (both rising and falling edges) using the CHANGE mode. When the echo pin transitions HIGH (pulse is sent), and if INT1 is enabled, the corresponding ISR (INT1_vect, see Appendix 6) is triggered, and it immediately starts Timer1.

Once the echo receives pulse and the pin transitions LOW, the same ISR is triggered again, this time stopping the timer. The value of TCNT1 is used to calculate the echo duration, which is converted into a distance. If the distance is less than 3 cm, we immediately issue a motor stop command. By using bare-metal programming, the HC-SR04 can trigger an emergency stop with significantly lower latency, reducing the response time from ~100 µs (using attachInterrupt) to under 1 µs.

Initially, we avoided this approach because the interrupt would continuously trigger once the astronaut was grabbed by the claw, preventing further movement. However, in hindsight, we could have simply disabled INT1, using EIMSK &= ~(1 << INT1), once the claw was fully closed, allowing Alex to resume motion regardless of proximity. The ISR implementation is shown below.

```
// INT1 ISR: triggered on rising or falling edge of echo
ISR(INT1_vect) {
  if (!measuring && (PIND & (1 << ECHO_PIN))) {
    // Rising edge: start timing
    TCNT1 = 0;
    TCCR1B = (1 << CS11); // Prescaler 8 → 1 tick = 0.5 µs
    measuring = true;
  } else if (measuring && !(PIND & (1 << ECHO_PIN))) {
    // Falling edge: stop timing
    TCCR1B = 0;
    duration_us = TCNT1/2.0; // Prescaler 8 → 1 tick = 0.5 µs
    measuring = false;

    float distance_cm = (duration_us * SPEED_OF_SOUND) / 2.0;
    if (distance_cm < 3.0) {
      stop_motors();
    }
  }
}
```

Code implementation of using interrupts for autonomous anti-collision braking

6.7. Others

GitHub Project Management

GitHub is used as the platform to manage, organise and save our code, while allowing for remote, effective collaboration. In the event of failed code implementation, we can always refer to the previous iteration to debug and fix the errors. GitHub helps organize the codes based on Arduino's C++ code, Python code of the firmware such as LiDAR and TLS code. Furthermore, GitHub enabled us to effectively delegate tasks and work independently. This streamlined our meetings, allowing us to focus solely on testing and integrating the individual components we had developed.

7. Lessons Learnt - Conclusion

Lesson 1: Ensuring sufficient practice given time constraints

We initially wanted to implement IMU sensors and Hector SLAM algorithm to improve the precision of Alex's movements and ease of navigation through the maze after the final quiz. However, these features were too complex and time-consuming to learn and implement effectively with only one week left to the trial run. During the trial run, we made a few mistakes such as unexpected SLAM map regeneration due to abrupt turns and lack of confidence in grabbing the red astronaut when it was in the grasp of Alex. This inevitably led to a lot of time wastage and task failure. We aimed to focus on practicing maze navigation with diverse maze layouts, helping us gain confidence and fine-tune our system for the final run.

Lesson 2: Hotspot Dependency and Network Failures

During the final run, we lost approximately 8 to 9 minutes due to the hotspot connection not being stable. The network was poor that day. It is possible that it was due to the environment we were in or too many signals at once. Due to this, we were not able to maintain a consistent VNC connection to the Raspberry Pi, and we had only a little more than 3 minutes for the actual run. This was extremely daunting, but, learning from Lesson 1, we were helped by the fact that we practiced extensively together at the COM1 lounge. We created virtual mazes using couch cushions to mimic actual running scenarios. Practicing this helped us hone our skills. It improved us at running the robot and coordinating its movement under pressure. With well-defined team organization with tasks delegated, we were able to change tasks quickly, even with limited setup time.

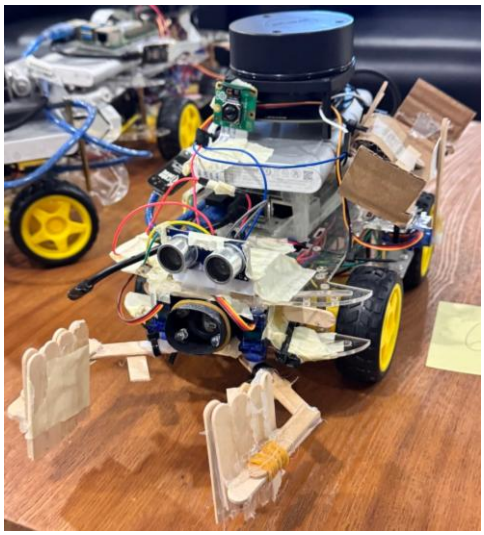
If we could do it again, we would bring a dedicated router to ensure consistent and reliable access, avoiding sole reliance on mobile hotspots.

Mistake 1: Poor Wire Management

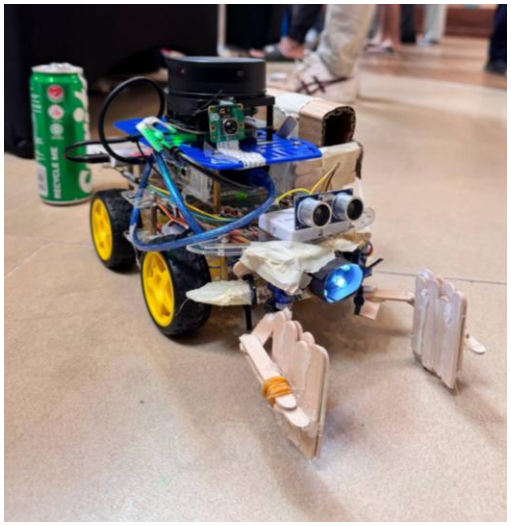
One mistake was poor wiring management, particularly the use of many redundant wire connections on our breadboard. In our trial run, we observed inconsistencies in the activation of the servos and the light sensor. The color sensor's light intensity also flickered at times, indicating unstable performance. Initially, multiple jumper wires were used to connect 5V from one breadboard to another and it worked. Although aware that the setup could be optimized, we chose not to modify it since it was an unwritten rule in robotics to not modify anything that is already functioning. However, this approach proved to be unreliable at later runs. Due to the internal resistance of the wires and the breadboard itself, several functions started failing especially as the battery drained.

This became evident during the trial run, where the 5V supplied by the Arduino Mega was insufficient to power the three servos, color sensor, and ultrasonic sensors simultaneously. As a result, some commands like opening the trapdoor became unresponsive. This caused the medpak to remain undelivered because the servo failed to actuate, even though the TPacket command was successfully transmitted. Additionally, the color sensor's light intensity was unstable, occasionally leading to inaccurate color classifications when we tested post-trial run.

With ample time before the final run, we decided that the most effective and reliable solution was to reconstruct the robot from scratch with optimized wiring and improved power distribution. We consolidated all connections onto a single mini breadboard to minimize internal resistance and eliminate unnecessary wiring. This significantly improved power delivery to all components. As a result, we were able to get all hardware working consistently, even when operating on a partially depleted battery.



Alex Pre-trial run form



Alex final form

Mistake 2: Holding WASD keys for movement

Initially, we were ambitious and aimed for a more intuitive control scheme by holding down the WASD keys to continuously move the robot. However, this approach frequently triggered *Bad Checksum* and *Bad Magic Number* errors, due to packet corruption caused by excessive and rapid serial communication. Additionally, due to latency in SLAM map updates during movement, holding down keys led to unreliable navigation.

To maintain system stability, we adopted a more deliberate and tactical control method. As outlined in Section 6.2, we used the ‘W’ and ‘S’ keys for short-distance movements and ‘P’ and ‘L’ keys for longer ones. By sending one command at a time, we ensured each movement was precise and controlled. Since the distance covered with each key press was known and consistent, we were able to approach the astronaut accurately using data from the ultrasonic sensor.

Furthermore, this helped us avoid collisions with the walls and navigate through tight corners more effectively. Hence, ultimately preserving the condition of our robot.

References

References Silva, J., Almeida, L., & Lima, P. (2006). RAPOSA: Semi-Autonomous Robot for Rescue Operations.

Zhang, J. (2023). Earthshaker: A mobile rescue robot for emergencies and disasters through teleoperation and autonomous navigation. JUSTC, 53.

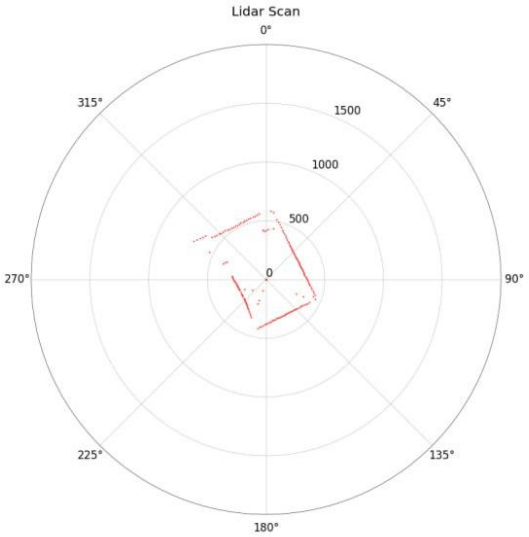
Appendix 1: Bare metal code used during set-up of UART Serial Communications

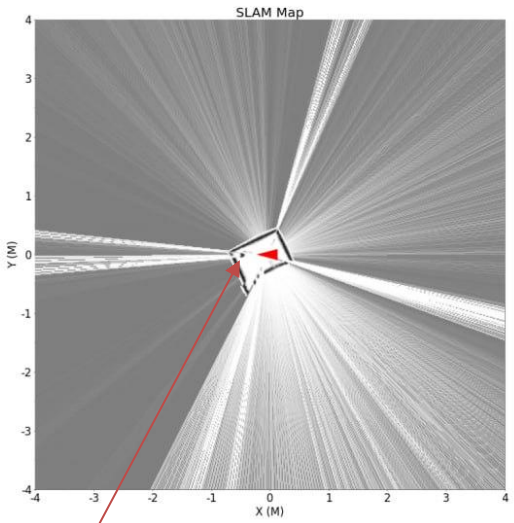
```
#include <avr/io.h>
#include <stdint.h>

#define F_CPU 16000000UL
#define BAUD 9600
#define MYUBRR ((F_CPU / (16UL * BAUD)) - 1)

void setupSerial()
{
  // Configure baud rate
  UBRR0H = (uint8_t)(MYUBRR >> 8);
  UBRR0L = (uint8_t)(MYUBRR);

  // Enable receiver and transmitter
  UCSR0B = (1 << RXEN0) | (1 << TXEN0);

  // Set frame format: 8 data bits, no parity, 1 stop bit (8N1)
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00);
}

int readSerial(char *buffer)
{
  int count = 0;

  // While data is available
  while (UCSR0A & (1 << RXC0))
  {
    buffer[count++] = UDR0;
  }

  return count;
}

void writeSerial(const char *buffer, int len)
{
  for (int i = 0; i < len; i++)
  {
    // Wait for empty transmit buffer
    while (!(UCSR0A & (1 << UDRE0)));

    // Put data into buffer, sends the data
    UDR0 = buffer[i];
  }
}
```

Appendix 2: Bare metal code used for Wheel Encoders

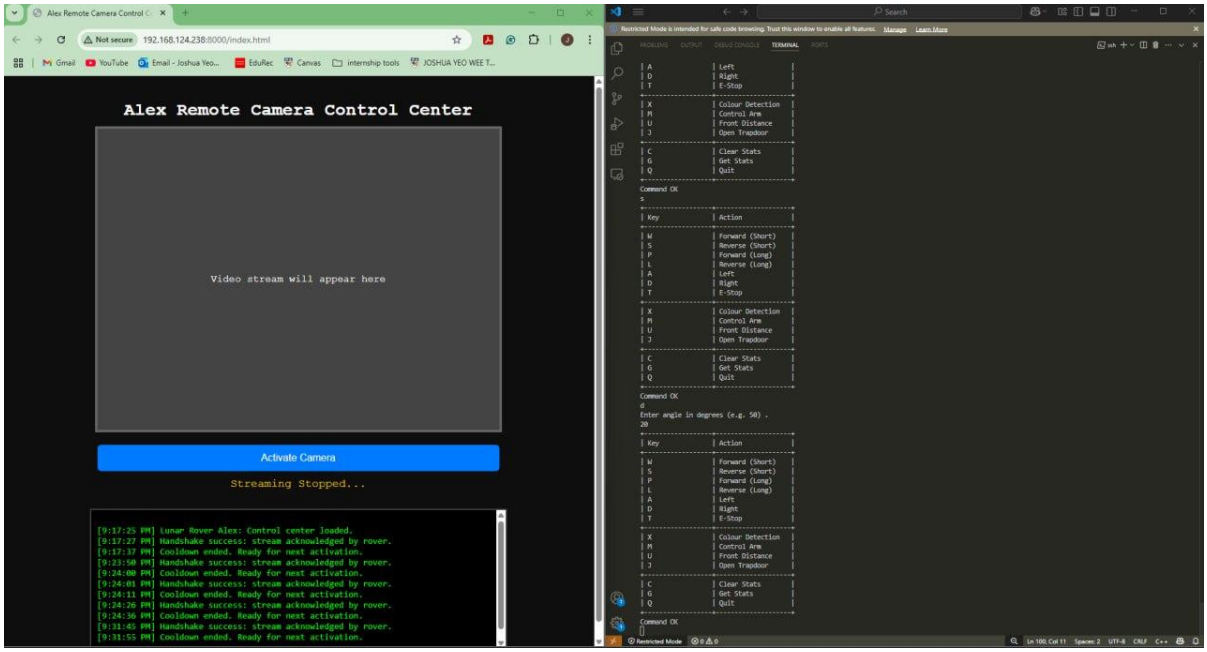
```
void setupEINT()
{
  EICRA |= (1 << ISC20); // Set INT2 and INT3 to CHANGE edge
  EICRA &= ~(1 << ISC21);
  EICRA |= (1 << ISC30);
  EICRA &= ~(1 << ISC31);
  EIMSK |= (1 << INT2) | (1 << INT3); // Enable INT2 and INT3 interrupts
}

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 &= ~((1 << 2) | (1 << 3)); // Clear the relevant bits in DDRE to
configure as input
  PORTD |= (1 << 2) | (1 << 3); // Setting these bits activates the built-in
pull-up resistors, ensuring the line is held high when not actively pulled low
by the sensor
}

ISR(INT2_vect) {
  rightISR();
}
ISR(INT3_vect){
  leftISR();
}
```

Appendix 3: Two Laptop System





Detected Astronaut



Appendix 4: Bare metal code used for Servos

```
// Define servo positions and states
static long left_servo_open = 280, right_servo_open = 1200;
static long left_servo_close = 400, right_servo_close = 700;
static long trapdoor_open = 2000, trapdoor_close = 800;
static long state = 1, trap_state = 1;

void setup_claw() {
    // Set PL3 (OC5A), PL4 (OC5B), and PL5 as outputs
    DDRL |= (1 << PL3) | (1 << PL4) | (1 << PL5);

    // Configure Timer5 for Phase and Frequency Correct PWM
    TCCR5A = (1 << COM5A1) | (1 << COM5B1) | (1 << COM5C1); // Enable OC5A,
OC5B, OC5C outputs
    TCCR5B = (1 << WGM53) | (1 << CS51);    // Mode 8, prescaler = 8
    ICR5 = 20000;

    // Initialize servos to open position
    OCR5A = left_servo_open;
    OCR5B = right_servo_open;
    OCR5C = trapdoor_close;
}

void claws_open() {
    OCR5A = left_servo_open;
    OCR5B = right_servo_open;
}
void claws_close() {
    OCR5A = left_servo_close;
    OCR5B = right_servo_close;
}

void toggle_claw() {
```

```
    if (state) {
        claws_close();
    } else {
        claws_open();
    }
    state = 1 - state;
}

void toggle_trapdoor() {
    if (trap_state) {
        OCR5C = trapdoor_open;
    } else {
        OCR5C = trapdoor_close;
    }
    trap_state = 1 - trap_state;
}
```

Appendix 5: Bare metal code used for Ultrasonic

```
void detectDistance()
{
    uint32_t duration_us = 0;
    uint32_t distance = 0;

    // Clear Timer1
    TCCR1A = 0; // Normal mode
    TCCR1B = 0;
    TCNT1 = 0;

    // Trigger 10us pulse on PD7
    PORTD |= (1 << PD7);
    _delay_us(10);
    PORTD &= ~(1 << PD7);

    // Wait for echo pin (PC1) to go HIGH (start of echo)
    uint16_t timeout = 30000; // ~30ms timeout
    while (!(PINC & (1 << PC1)) && --timeout);

    if (timeout == 0) {
        sendMessage("Echo HIGH timeout");
        return;
    }

    TCCR1B |= (1 << CS11); // Prescaler = 8

    // Wait for echo pin to go LOW (end of echo)
    timeout = 60000; // ~60ms timeout
    while ((PINC & (1 << PC1)) && --timeout);

    // Stop Timer1
    TCCR1B = 0;

    if (timeout == 0) {
        sendMessage("Echo LOW timeout");
        return;
    }
    duration_us = TCNT1/2.0; // Prescaler 8 → 1 tick = 0.5 µs
    distance = (uint32_t)(duration * SPEED_OF_SOUND / 2.0);
```

```
    // Prepare and send response
    TPacket distance_response = {0};
    distance_response.packetType = PACKET_TYPE_RESPONSE;
    distance_response.command = RESP_ULTRASONIC;
    distance_response.params[0] = distance;
    distance_response.params[1] = duration;
    distance_response.data[0] = 'd';

    sendResponse(&distance_response);
}
```

Appendix 6: ISR_vect table

Table 14-1. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7

INT1 → Ultrasonic Sensor [Anti-collision System]; INT2 → Right Encoder; INT3 → Left Encoder