CG2111A Engineering Principle and Practice II

Semester 2 2024/2025

"Alex to the Rescue"

Final Report

Team: B04-3A

Name	*Student #*	*Sub-Team*	*Role*
Tan Pang	*A0307465B*	*Firmware*	*Member*
Thia Yang Han	*A0308175A*	*Software*	*Lead*
Premil Roshan	*A0309256B*	*Mechanical design*	*Lead*
Yang Zhenzhao	*A0309132N*	*Circuitry*	*Lead*
Yao Xiang	*A0299826E*	*Firmware*	*Lead*

Section 1 Introduction **4**
Section 2 Review of State of the Art **4**
2.1 RAPOSA 4
2.1.1 Strengths 4
2.1.2 Weaknesses 5
2.2 Robbie 5
2.2.1 Strengths 5
2.2.2 Weaknesses 5
Section 3 System Architecture **6**
3.1 Core Components & Their Roles 6
3.1.1 Raspberry Pi (High-Level Processing) 6
3.1.2 Arduino (Low-Level Actuation & Sensing) 6
3.2 Key Communication Pathways 6
3.3 Sensor & Actuation Workflow 6
3.3.1 Mapping & Navigation 6
3.3.2 Astronaut Detection 6
3.3.3 Movement & Odometry 7
Section 4 Hardware Design **7**
4.1 Placement of components 7
4.1.1 Ultrasonic sensor(HC-SR04) 7
4.1.2 Claw 8
4.1.3 Colour sensor 8
4.1.4 Trapdoor 8
4.1.5 RPi Camera 8
4.2 Height Considerations 9
4.3 Other hardware Considerations 9
4.3.1 Cable Management 9
4.3.2 Efficiency and Speed 9
Section 5 Firmware Design **9**
5.1 UART 9
5.1.1 Serial Communication Setup 9
5.1.2 Data Packet Structure 10
5.1.3 Enumeration of Data 10
5.1.4 Interpreting and Handling of Data 11
5.2 Ultrasonic sensor 11
5.3 Colour Sensor 11
5.3.1 Choice of Colour Sensor 11
5.3.2 Calibration 11
5.4 Servos 12
5.4.1 Pinouts 12
5.4.2 Claw & Trapdoor 12

Section 6 Software Design..... 13
6.1 Network and TLS..... 13
6.1.1 Raspberry Pi server and Laptop client..... 13
6.1.2 Receiving Response Packet..... 13
6.1.3 Successive Command Delaying..... 14
6.2 Controls..... 14
6.3 Slam mapping..... 14
6.4 RPi Camera..... 14
Section 7 Lessons Learnt - Conclusion..... 15
7.1 Greatest Mistakes Made & Lessons Learnt..... 15
7.1.1 Too caught up on minor problems..... 15
7.1.2 Failure to manage versions of code properly..... 15
Appendix..... 16
Hardware Design..... 16
Firmware Design..... 20
Software Design..... 35
Full Code on Alex.ino code..... 36
Full robot.ino code..... 57
Full Constants.h code..... 60

Section 1 Introduction

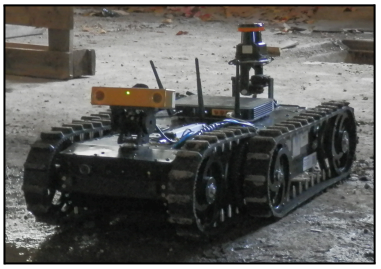
Our robotic vehicle, Alex, is designed to carry out effective search-and-rescue operations within the simulated lunar environment provided. The primary functions include remote navigation, allowing operators to precisely maneuver the vehicle forward, backward, and execute controlled rotations to avoid obstacles at least 21 cm high. Navigation is guided by real-time LIDAR mapping powered by Raspberry Pi.

Using the continuous LIDAR-based data, Alex will accurately map its surroundings, helping operators create a clear, hand-drawn map of its environment. For rescue operations, the system differentiates between astronauts requiring minimal medical assistance (Green) and those critically injured (Red) by sensing their colour via a colour sensor. The onboard robotic arm, powered by servo motors, is designed to securely grip and relocate the red astronauts safely to the designated safe zones.A revolving trap door is used to deliver the medpack to the green astronaut. Additionally, operators will have limited strategic use of an onboard Raspberry Pi camera restricted to four activations lasting no more than ten seconds each for visual assessment during critical rescue interactions, strictly not for navigation purposes. This integrated approach ensures efficient mission execution, satisfying all project constraints and requirements.

Section 2 Review of State of the Art

2.1 RAPOSA

RAPOSA is a search-and-rescue robot designed to operate in hazardous environments. It features a dual-body construction and side-tracked wheels that allow it to move even when flipped upside down. Equipped with various sensors such as temperature, humidity, and gas sensors, it can identify environmental hazards. A thermal camera helps locate heat sources, aiding in human detection. Placement of its sensors can be drawn inspiration from.



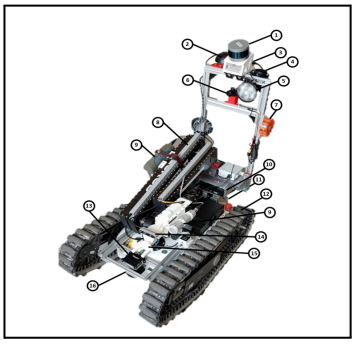
2.1.1 Strengths

The robot is designed to operate effectively in environments with unreliable wireless networking, thanks to its optional tethered connection that can be attached or detached in real time based on mission needs. Its modular and highly customizable platform allows for diverse equipment configurations tailored to specific rescue scenarios. It features autonomous navigation, enabling operators to concentrate on high-level mission control while the robot handles movement. Additionally, its ruggedized chassis and tracked mobility system ensure durability and dependable performance, even if the robot is flipped upside down.

2.1.2 Weaknesses

The robot requires manual tether management during movement, which can cause entanglement and delays in operation. Its wireless communication range is limited, potentially hindering effectiveness in large-scale or remote missions. Additionally, the high costs of production and maintenance may restrict accessibility for smaller organizations or teams with limited budgets. The system's overall performance also depends on the operator's skill level and familiarity with the platform.

2.2 Robbie



Robbie is built to support emergency teams with tasks such as object detection and terrain navigation. It uses a modular hardware setup, including RGB cameras, 2D and 3D lidars, a robotic arm, and advanced computing. Robbie uses ROS with tools like Rviz, Octomap, and YOLO-ROS for mapping and object recognition. It can be remotely controlled and generates real-time maps of its surroundings.

2.2.1 Strengths

- Modular hardware design enables high customizability, allowing the robot to be adapted for a wide range of rescue and reconnaissance tasks.
- Capable of navigating complex and dynamic environments, with support for teleoperation over long distances.
- Equipped with user-friendly software that includes features like automatic obstacle avoidance and voice recognition, enhancing usability and efficiency.

2.2.2 Weaknesses

- Mobility is constrained by its wheeled base, which may limit performance on uneven or debris-filled terrain.
- Robotic arm and gripper may lack the precision required for delicate or fine manipulation tasks.
- Limited battery capacity can restrict operational duration, posing challenges for extended missions in the field

Section 3 System Architecture

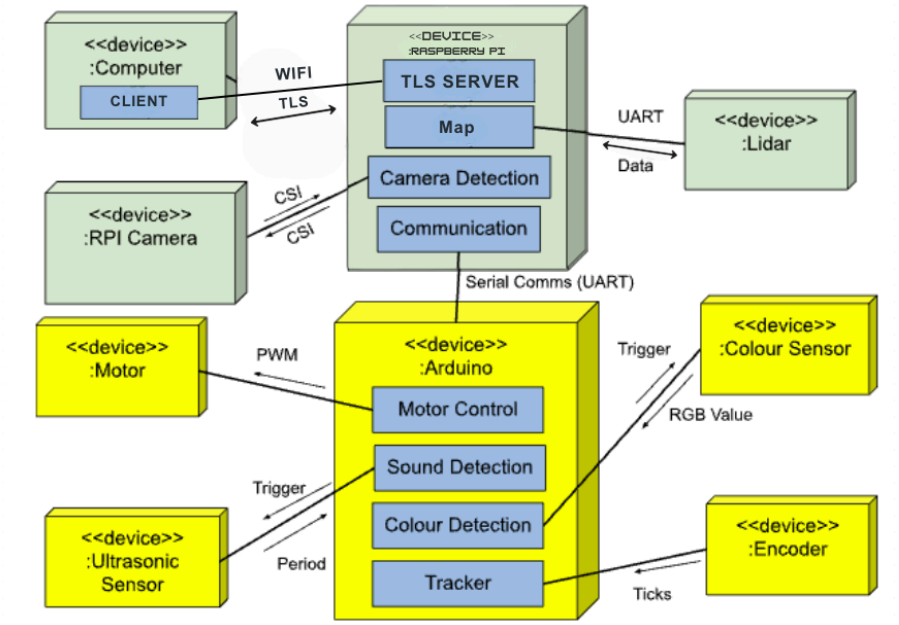


Figure 1: Diagram of System Architecture

3.1 Core Components & Their Roles

3.1.1 Raspberry Pi (High-Level Processing)

- Role: Acts as the central brain for decision-making, mapping, and vision processing.

3.1.2 Arduino (Low-Level Actuation & Sensing)

- Role: Handles real-time motor control, sensor interfacing, and servo actuation.

3.2 Key Communication Pathways

- Raspberry Pi ↔ Computer (User Interface)
- Raspberry Pi ↔ Arduino
- Raspberry Pi ↔ LiDAR
- Arduino ↔ Motors/Encoders/Sensors

3.3 Sensor & Actuation Workflow

3.3.1 Mapping & Navigation

1. LiDAR scans surroundings
2. Raspberry Pi builds 2D map
3. Path planning
4. Commands sent to Arduino
5. Motors execute movement

- Fallback Mechanism:

  Ultrasonic sensor detects close-range obstacles, triggering emergency stops.

3.3.2 Astronaut Detection

- Primary Detection:

- Navigating to the left and right of the object will reveal where it is the edge of a wall or if it is likely an astronaut (a red dot)
- Raspberry Pi Camera streams for 10s to confirm that it is an astronaut.
- The Ultrasonic Sensor will detect if the astronaut is within grabbing range

- Identification of Astronaut(Red/Green):
  - Colour sensor detects RGB values to identify red or green astronauts grabbed by the claw.

3.3.3 Movement & Odometry

- Wheel encoders track rotation ticks
  - Arduino calculates distance
  - Raspberry Pi updates real-time position on the map.

Section 4 Hardware Design

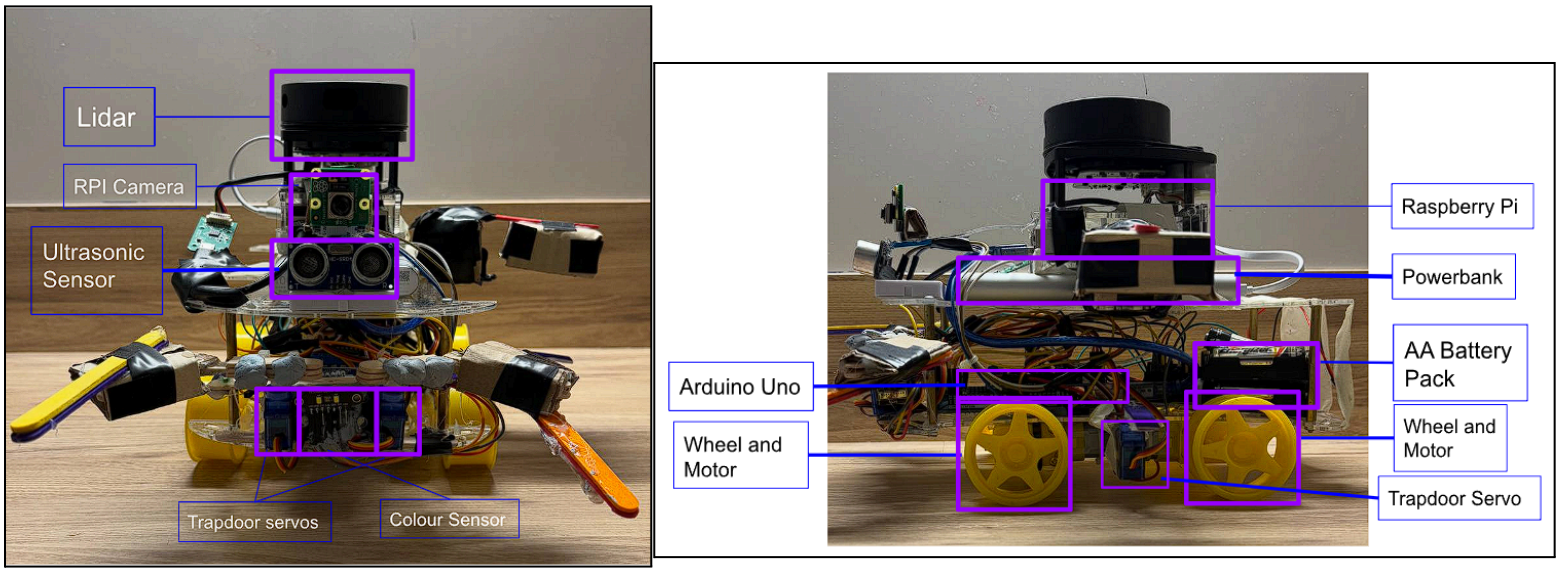
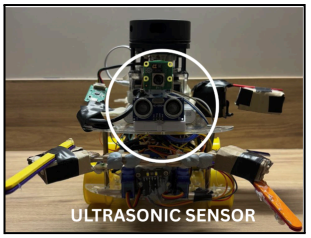


Figure 2: Front and Side Views of Alex with labels

Descriptions and functionalities of the hardware components such as Raspberry Pi, Arduino Uno, LIDAR, Ultrasonic sensor, Colour sensor. (Appendix, Figure 5*: Hardware Components*)

4.1 Placement of components



4.1.1 Ultrasonic sensor(HC-SR04)

The ultrasonic sensor, mounted at the front center of the top acrylic plate, is aligned with the head of the astronaut and positioned centrally within the claw's span. It emits sound waves to measure distance and works alongside the LiDAR to determine when Alex is close enough to begin the rescue. Once a suitable range is detected, the robot automatically

advances, decreasing the distance until the astronaut is within reach, triggering the claw to grab and drag the astronaut accurately.

4.1.2 Claw

Firstly, the elevation of the claw was carefully adjusted to grip around the centre of gravity of the Red astronaut, ensuring it was neither too low nor too high. This prevents toppling during gripping and movement, allowing for stable and secure handling.

Cardboard extensions are added to the outer arms to increase surface area and enhance grip stability, while the inner arms are reinforced with double-layered ice cream sticks to provide added structural rigidity and prevent flexing under load. Additionally, each outer arm—one placed high and the other low though flimsy, they effectively pushes the astronaut toward the claw's embrace, helping Alex secure a firmer hold. The different height placement also increases the claw's range of motion, making it more adaptable when gripping astronauts. We were well aware that having one high one low for claw placement may cause the astronaut to lean towards the lower side, hence we specifically designed the inner arms (the one that is hugging the astronaut) to be at the same level; while only the outer arms (yellow orange) to be at different levels to enhance range of motion when gripping. (Appendix, *Figure 6: Zoomed in view of Claw Placement*)

4.1.3 Colour sensor

The colour sensor is fixed at the very front of the robot, positioned slightly below and between the pincers of the claw. This placement ensures that when Alex grips an astronaut, the sensor is brought into the closest possible proximity to it. By doing this, we minimize external light interference and maximise reading accuracy, allowing the sensor to capture precise RGB values for reliable colour classification. (Appendix, *Figure 7: Zoomed in view of Colour Sensor*)

4.1.4 Trapdoor

The trapdoor is operated by a single servo motor positioned beneath the medpack storage area. This servo controls a hinged platform that holds the medpack in place during navigation. When a green coloured astronaut is sensed, the servo rotates to ensure the swift deployment of the medpak. This is the most efficient method we thought of as the majority of the motion of deployment rely on gravity, our only focus is to make sure that medpack is firm in the clutch of the trapdoor during movements. (Appendix, *Figure 8: Zoomed in view of Trapdoor*)

4.1.5 RPi Camera

Mounted at the front of Alex on top of the Ultrasonic sensor for a bird's eye-view just nice enough yet not blocking the detection of our lidar, the RPi camera provides a real-time visual

context of Alex's front view. The camera is utilized to confirm the presence of an astronaut and, in critical conditions, to identify Alex's surrounding situation. (Appendix, *Figure 9: Zoomed in view of RPi Camera*)

4.2 Height Considerations

Next, the placement of all the hardware components such as the ultrasonic sensor and camera must not block the lasers from the Lidar. Hence we have decided to mount the Lidar at the very top of our robot while at the same time ensuring that it is not taller than the height of the walls of the maze.

4.3 Other hardware Considerations

4.3.1 Cable Management

To ensure efficient troubleshooting and clean circuit organization, all wires were neatly grouped and secured using cable ties based on their respective components. This modular cable management approach allowed for quick identification of faulty connections and reduced the risk of wire entanglement or accidental disconnections during operation. It also greatly sped up the debugging process, especially under time constraints during testing and demonstrations.

4.3.2 Efficiency and Speed

To operate within the tight time constraints of the mission, several optimisations were implemented. One key adjustment was removing the rubber from the wheels. During early testing, the rubber added excessive friction, which hindered the robot's ability to turn smoothly, especially when dragging the Red astronaut. By stripping the rubber, the robot experienced smoother and stable rotational movement, making maneuvering in tight spaces more efficient even under load.The claw mechanism was enhanced using Blu-Tack on each pincer, increasing grip strength and ensuring a firm hold on astronauts during dragging. The robot used a combination of LIDAR and ultrasonic sensors to detect when it was within the optimal range of an astronaut. Once close enough, it would immediately grab the astronaut first, reducing the time spent aligning for sensor readings. If the astronaut was Green, the claw would release it on the spot, and the trapdoor would be used to deliver the medpack. If the astronaut was Red, the robot would begin dragging them to the designated safe zone.

Section 5 Firmware Design

5.1 UART

5.1.1 Serial Communication Setup

To ensure communication between the RPI and the Arduino, serial communication, specifically UART was used.

We used baremetal coding for our serial setup. setupSerial() clears PRUSART0, programs UBRR0H/L for 9600 baud, and sets 8 data bits, no parity, 1 stop bit via UCSR0C. startSerial() enables TX and RX by setting TXEN0/RXEN0 in UCSR0B. readSerial() polls RXC0 in UCSR0A and reads available bytes from UDR0 into a buffer. writeSerial() polls UDRE0 before writing each byte from your buffer to UDR0. (Appendix, *Figure 10: Bare-metal Code on Serial communication*).

5.1.2 Data Packet Structure

UART communication between the Raspberry Pi and Arduino uses a fixed-length, 100-byte packet defined in packeto.h. The packet structure is as follows:

- The packet begins with two single-byte header fields: packetType, which distinguishes Pi-to-Arduino commands from Arduino-to-Pi responses, and command, which identifies the specific action or reply.
- A two-byte dummy field follows to maintain 4-byte alignment for the remaining payload.
- The payload comprises a 32-byte char data array and a 16-element uint32_t params array. They are used to transfer sensor data and movement speed and distance which are preset in our speed gears.

This structure provides a compact, alignment-safe container for all control and telemetry information exchanged over the UART link.

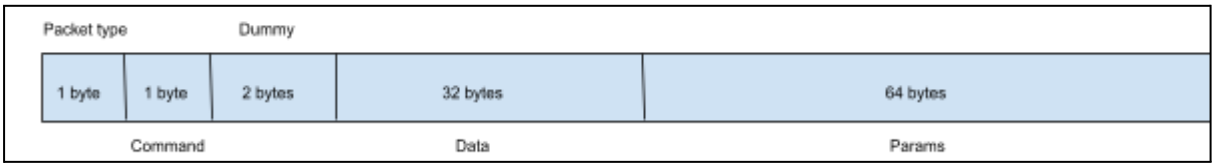


Figure 3: Data packet structure

5.1.3 Enumeration of Data

We use two enums—TCommandType and TResponseType (in constanto.h)—to label Pi to Arduino commands and Arduino to Pi responses. Before sending, each enum is cast to a char and packed into the UART packet for compact, unambiguous encoding; on receipt, it's decoded to invoke the correct action or interpret the reply. TCommandType has been extended with custom commands (speed gears, claw, ultrasonic, colour sensor, trap door), while TResponseType provides default handshake codes to acknowledge success or report errors. (Appendix, *Figure 11: Enum code*)

5.1.4 Interpreting and Handling of Data

When data packets with TCommandTypes are sent from RPI to the Arduino, the packet will be read under readPacket() function to deserialize checks for data integrity. If the data packet is invalid, a bad packet response(bad magic number) will be sent, while if the data is corrupted, a bad checksum response will be sent. If the data packet is sound, handleCommand() function is used to interpret the data packet (Appendix*, Figure 12, HandleCommand function handles the commands sent through TLS on Arduino & Figure 13: Keybindings table*)

5.2 Ultrasonic sensor

To calculate the distance between objects and the ultrasonic sensor placed at the front of Alex, the traveling time for the ultrasonic wave was measured and is calculated later for distance in cm. HC-SR04 ultrasonic sensor was used. We wrote a bare-metal driver for the HC-SR04 on Arduino by configuring PA3 (D25) as a low-driven trigger and PA4 (D26) as echo input in ultraSetup(), which also puts Timer 1 into Normal mode with a prescaler of 8 (0.5 µs ticks). In getDistance() we issue a 10 µs pulse on PA3, reset TCNT1 at the echo's rising edge and read it on the falling edge, convert ticks to microseconds, compute distance_cm = (time × 0.034) / 2, apply a fixed calibration offset, and send the result over UART via dbprintf(). (Appendix, *Figure 14: Ultrasonic sensor code*)

5.3 Colour Sensor

5.3.1 Choice of Colour Sensor

Due to a faulty TCS3200 colour sensor, we decided to take a step up and improvise our own colour sensor instead after coming to a conclusion that colour sensor is the key to the astronaut identification; it holds the final say over whether we fail or manage to rescue the astronauts. Hence, we acquired the TCS34725 colour sensor which offered built-in I2C support, better light filtering, and more stable RGB readings. This shift allowed us to write our own bare-metal I2C communication code for direct control, improving both accuracy and response time. With this sensor, color detection became far more dependable, ultimately strengthening Alex's ability to make mission-critical decisions on the field.

Here is our colour sensor code and description, all in bare-metal via *Arduino (Appendix, Figures 15-17*).

5.3.2 Calibration

```
void TCS34725_Init() {
    TCS34725_WriteReg(0x00, 0x01); // Power on
    _delay_ms(3);
    TCS34725_WriteReg(0x00, 0x03); // Enable RGBC
    TCS34725_WriteReg(0x01, 0xFF); // Integration time
    TCS34725_WriteReg(0x0F, 0x01); // Gain = 4x
}
```

Figure 4: A segment of the initialisation setup function, on selected modes

Little calibration is done because for colour sensors like TCS3200 and TCS34725, they have a built-in LED to overpower the variance of ambient light; all we needed to do was to adjust the modes here to accurately distinguish between red and green.
Here are the values that we have considered: Appendix, *Figure 18: Calibration Checklist.*

5.4 Servos

5.4.1 Pinouts

As shown in *Appendix, Figure 19*, all the 16-bit timers have been covered by the motor-driver aside from timer 5. Hence, in consideration of maintaining 50hz to ensure stability and smooth movements of servo, we dedicated pin 46,45,44 of timer 5 to the left claw, right claw and trapdoor servos respectively.

5.4.2 Claw & Trapdoor

Instead of relying entirely on Arduino's Servo library, we implemented bare-metal PWM control for all of our servos using Timer 5 on the ATmega2560. This gives us precise control over pulse width generation and reduces dependency on higher-level abstractions — which is critical for real-time and timing-sensitive operations. We configured Timer 5 to operate in Phase Correct PWM Mode (Mode 10) with ICR5 as TOP, enabling a servo-compatible PWM period of 20 ms (50 Hz), suitable for standard SG90 servos. The init_servo() function is designed to configure Timer5 on an ATmega2560 microcontroller to control three servos using PWM signals. It first sets the relevant pins (OC5A, OC5B, OC5C) as output pins to generate the PWM signals. The function then configures Timer5 to operate in Phase Correct PWM mode, setting the period of the PWM signal to 20ms, which is standard for servo control. This is achieved by setting the appropriate bits in the timer's control registers. The function also defines the initial pulse widths for each servo, corresponding to different positions, such as full left, full right, and neutral. Finally, the timer is started with a prescaler of 8, ensuring the correct timing and resolution for servo control. The overall setup allows for smooth and accurate movement of the servos. (Appendix, *Figure 20: Servo-Initiation Code (All 3 Servos)*)

The servo_angle() function is designed to adjust the positions of two servos based on the given angles for each. It first checks if the input angles for both the left and right servos are within the valid range of 0 to 180 degrees. If the angles fall outside this range, the function exits early without making any adjustments. If the angles are valid, the function converts the angles to PWM values that correspond to the desired pulse widths for the servos. The conversion maps 0° to 1000 µs, 90° to 1500 µs, and 180° to 2000 µs. These PWM values are then applied to the control registers of Timer5 (OCR5A and OCR5B) to adjust the servo positions accordingly. The result is precise control of the left and right servos based on the specified angle inputs. (Appendix, *Figure 21: Claw-Grabbing Code for Claw*)

This switch-case segment handles various servo control commands related to a robot's claw and trapdoor mechanisms. For the COMMAND_SERVO_OPEN and COMMAND_SERVO_CLOSE cases, the code first acknowledges the command by sending an OK response, then calls the servo_angle() function with specific angles to move the left and right servos to the open or closed positions respectively. These actions are accompanied by debug messages to indicate the current state. The claw operation, handled in the COMMAND_TURN_AND_OPEN_TRAP case, directly sets the OCR5C register to 1000, which controls the servo connected to OC5C to open the trap.Unlike the claw servos, which use a function call for abstraction and safety checks, the trapdoor's servo is controlled directly by assigning a PWM value to the register, bypassing any intermediate logic. This direct manipulation provides immediate response and is meant for *single* operation (drop medpack and we can leave it be). (Appendix, *Figure 22: Command-Interpretation for Claw&Trapdoor*)

Section 6 Software Design

6.1 Network and TLS

The network used for the communication between our laptop and Alex was the hotspot from one of the group members. The laptop and RPi were both connected to the same network allowing the transfer of data between them. To increase network security, TLS was implemented. Public and Private keys were generated using OpenSSL as per our TLS studio.

6.1.1 Raspberry Pi server and Laptop client

In our instance, the RPi was made to host a server that could receive commands sent over the network. The program would start a listener thread which would wait for connections from clients. Once a client is connected to the server, a new worker thread is created. The worker thread is in charge of sending commands from the laptop to the RPi and eventually to the arduino. The program will be required to deserialize the data over the network and reserialize into packets over the serial communication line to be sent to the Arduino. On the laptop side, we created a client program which connects to the RPi server and acts as a user interface between the operator and the main arduino program. The client will constantly poll for user input, taking in a new command every 3 seconds which are then serialised into data packets to be sent over the server to the RPi.

6.1.2 Receiving Response Packet

When a response packet is sent to the laptop over serial communications, the RPi receives the packet from the arduino via the serial communications line. It then deserializes it and reserializes and sends it over the network to the laptop. Ultimately, the laptop deserializes the data packet and displays the corresponding response information. (Appendix, *Figure 23: Pictorial Representation of data flow*)

6.1.3 Successive Command Delaying

In the interest of ease of usage, we decided to remove the need for the enter key. We realised that if data packets were sent faster than Alex could execute the commands, a bad magic error would occur. Subsequent commands were getting corrupted as data packets started to be mashed together. This prompted the use of the 3 second buffer between data packets which was the time needed to execute the longest command movement in gear 3. (Appendix, *Figure 24: Code for 3 second buffer*)

6.2 Controls

To make controlling Alex more intuitive, we replaced the original controls. Initially, the operator was required to input a direction, the distance and speed. We replaced it with WASD and its surrounding keys. This enabled easier maneuverability of Alex. As mentioned in 6.1.3 we also removed the need for the enter key. Using the read() function, it reads 1 byte of data from the terminal every instance of the loop. By disabling canonical mode and input echo using the termios library, the enter key can be omitted as every character(1 byte) will be read and serialised individually into packets. This new version is similar to movement control in games which makes it easier to control Alex. (Appendix, *Figure 13: Keybindings table*)

6.3 Slam mapping

To visualize the data from the LiDAR and to create a 2D map of the surroundings, we made use of the BreezySlam and Bokeh application to plot the map. The ROS master node will register the RPLidar node, the slam node and the GUI nodes. The RPLidar node will then publish the data via the 'lidar/scan' topic to the slam node and the display/GUI node. The slam node will also publish data via the 'slam/mappose' topic to the display/GUI node. Using the data, the bokeh application will construct the map and update in real time. Due to the high latency of the slam map on the VNC server we found that it was extremely inaccurate and difficult to use, hence we chose to use the bokeh application. This not only reduces the workload on the RPi but increases the resolution and responsiveness of the map. (Appendix, *Figure 25: RQT graph of Lidar map publishing*)

6.4 RPi Camera

We use the provided AlexCameraStreamServer.py to handle all RPi-camera streaming—no modifications are made beyond configuring the target IP and port as instructed. During initialization, the script opens the RPi's camera module, encodes each frame into a lightweight black-and-white outline, and buffers up to 10 seconds of video. When Python AlexCameraStreamServer.py is launched, it starts an HTTP server on the configured port (default 8000). We then enter *http://<RPi_IP_Adress>:8000* in one of our laptop's browser, "Start Stream" is then pressed to receive a live, 10-second video segment during our run.

Section 7 Lessons Learnt - Conclusion

7.1 Greatest Mistakes Made & Lessons Learnt

7.1.1 Too caught up on minor problems

Initially, we were too focused on perfecting the base structure and mechanical build of Alex, prioritizing precision over progression. As a result, we delayed critical steps like securing the top plate to mount the LiDAR and conducting integrated system tests. While our individual components performed well in isolation, the final assembly exposed weaknesses in our design: for instance, although the claw successfully gripped the astronaut, the motors lacked sufficient torque to turn the wheels while bearing the load.

From this we learnt that, It is crucial to adopt a system-level approach early in the development process rather than focusing too heavily on component perfection. Iterative integration and testing could have revealed these problems sooner, enabling us to redesign or reposition elements like the claw and sensors before final assembly. We also learned that practical functionality and design compatibility must always be considered in tandem, especially when dealing with spatial constraints and mechanical interactions. This is especially imperative to us when it comes to projects that have such tight constraints in due date such as our Moonbase Rescue Mission.

7.1.2 Failure to manage versions of code properly

Initially, we created multiple versions of code after the multiple loops of testing and rewriting the contents of our code via trial and error. This has led to multiple different copies of our Arduino as well as TLS code. Though we uploaded our codes to GitHub on a weekly basis, we failed to ensure that it was git-pulled to our RPi and tested during the day of our final run. This led to panic, as we realized that the version uploaded onto our Arduino via VNC was not the correct one just 10 minutes before the 'hand-in your robots' time. In a rush, we managed to identify the issue, correct the version, and successfully change the code to make it work in time. Nonetheless, it was one of the most critical mistakes we made and could have cost us all our marks.

This experience emphasized the importance of consistent and disciplined version control practices. Merely pushing updates to GitHub is not enough—ensuring that the correct version is pulled, verified, and actively tested on the actual hardware is equally essential. Going forward, we intend to adopt more structured versioning practices, including maintaining a clearly commented out and tested final version of our codes, verifying updates on all platforms (arduino, laptop, RPi) prior to crucial sessions, and possibly making regular ‘version sync checks’. This approach will greatly reduce the chances of such errors occurring again and reinforce our professional project management standards.

References

ISR Group. (n.d.). *RAPOSA – Search and rescue robot*. Instituto de Sistemas e Robótica. Retrieved March 29, 2025, from https://irsgroup.isr.tecnico.ulisboa.pt/raposa/

ROBBIE : Georg A. Novotny, Simon Emsenhuber, Philipp Klammer, Christoph Pöschko, Florian Voglsinger & Wilfried Kubinger *(2019) A M robot platform for search and Rescue Applications - DAAAM*. (n.d.). https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2019/131.pdf

Appendix

Hardware Design

Components	Functions
Raspberry Pi	Relays user commands to arduino to control the motors, provides power to both Arduino and LIDAR, receives information from the LIDAR and sends collected data to the laptop
Arduino Uno	Receives motor control instruction from Pi, enacts commands for motor movements.
LIDAR	Maps surrounding terrain, transmits collected data points back to Pi
DRV-8833 Dual Motor Driver	Drives speed and direction of motors, provides power from batteries to drive motors.
Ultrasonic sensor	Detects distance between sensor and objects in its vicinity
Colour sensor(TCS34725)	Detects and measures red, blue, green and white light to compute colour
Trapdoor	An ice cream stick is attached to a servo to operate the trapdoor mechanism and release medpacks to Green astronauts with minimal delay.

Claw	Two servos form a two-joint claw system used to grip and drag Red astronauts to the designated safe zone. These servos work in coordination to open/close the claw and control its vertical motion for effective grasping and pulling. Receive precise angle commands from the Arduino, powered via an external 5V supply, and execute controlled rotations to actuate the gripper and arm mechanisms during operation.
RPI Camera	Captures 10-second segments of black-and-white video and streams them on the server provided, providing a real-time visual context of Alex's surroundings.

Figure 5: Hardware Components

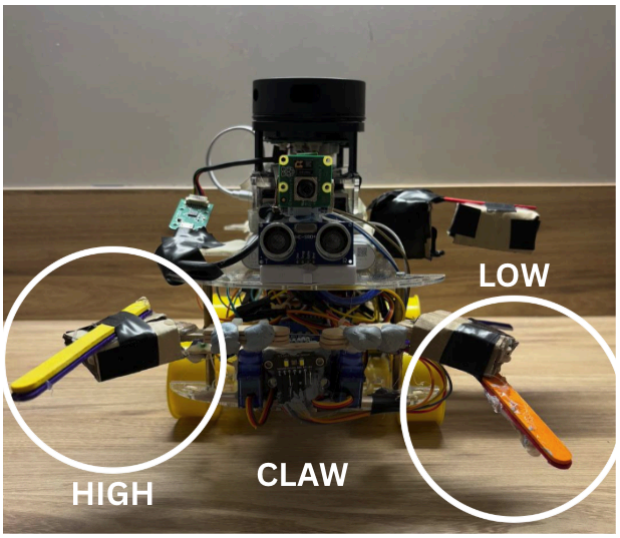


Figure 6: Zoomed in view of Claw Placement

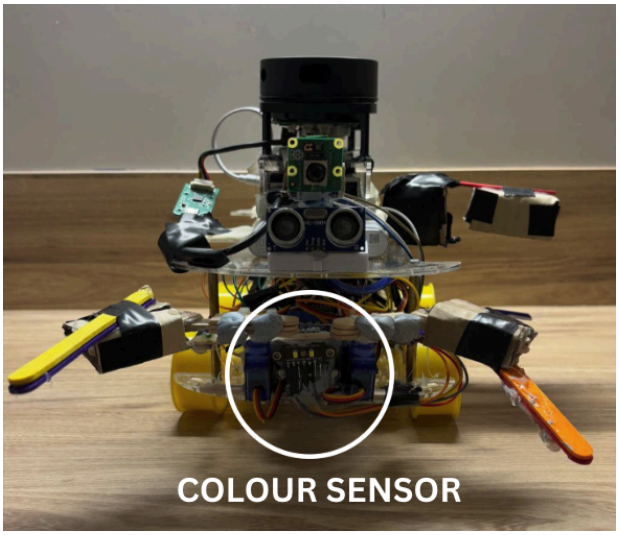


Figure 7: Zoomed in view of Colour Sensor

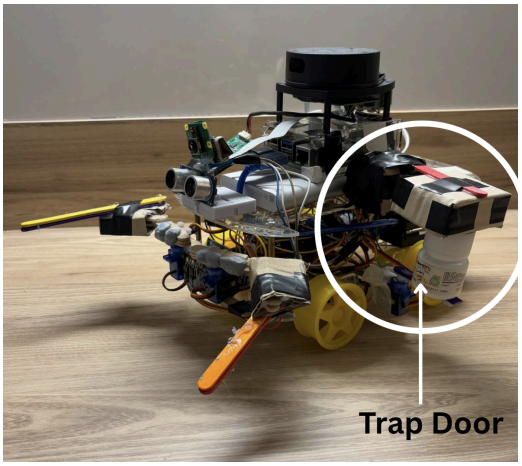


Figure 8: Zoomed in view of Trapdoor

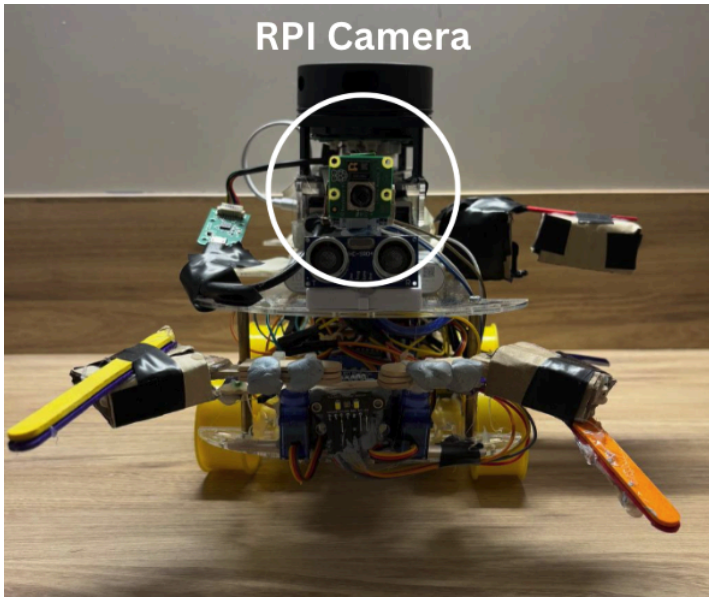


Figure 9: Zoomed in view RPi Camera

Firmware Design

```
void setupSerial() {
    PRR0 &= ~(1 << 1); // switch off power reducing mode
    uint16_t ubrr_value = 103; // set bart rate to 9600
((16000000/16*9600) - 1 = 103)
    UBRR0H = (ubrr_value >> 8); //set higher baud rate register
    UBRR0L = ubrr_value; //set lower baud rate register
    UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); //set asynchronous 8N1
mode
    UCSR0A = 0; // clears status flag of UCR0A for reading later
}

void startSerial() {
    UCSR0B = (1 << RXEN0) | (1 << TXEN0); //enabling USART receive
and transmit
}

int readSerial(char *buffer) {

  int count = 0;
  while (UCSR0A & (1 << RXC0))
    buffer[count++] = UDR0; // read data from data register to
buffer

// while loop checks status received status flag on UCSR0A on
whether data is available to read in data register

  return count;
}

void writeSerial(const char *buffer, int len) {

  for(int i = 0; i < len; i++)
  {
      while (!(UCSR0A & (1 << UDRE0)));

//check whether data register is empty (ready to write) by
checking data register status flag on UCSR0A register

      UDR0 = buffer[i]; // writing data from buffer to data
register
```

```
  }
}
```

Figure 10: Bare-metal Code on Serial communication

```
typedef enum
{
      COMMAND_FORWARD = 0,   //w
      COMMAND_REVERSE = 1,   //s
      COMMAND_TURN_LEFT = 2,
      //a
      COMMAND_TURN_RIGHT = 3,
      //d
      COMMAND_GEAR_1 = 4,    //f
      COMMAND_GEAR_2 = 5,    //g
      COMMAND_GEAR_3 = 6,    //h
      COMMAND_ULTRA = 7,     //z
      COMMAND_COLOUR = 8,    //c
      COMMAND_CAM = 9, //x   [1]
      COMMAND_STOP = 10,     //y
      COMMAND_SERVO_OPEN = 11,
      //q
      COMMAND_SERVO_CLOSE = 12,
      //e
      COMMAND_CLEAR_STATS = 13,
      //t
      COMMAND_GET_STATS = 14,
      //r
      COMMAND_TURN_AND_OPEN_TRAP
= 15,  //o
      COMMAND_SHAKE = 16 //p
} 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;
```

Figure 11: Enum code

[1]COMMAND_CAM was added before studio 19 as we thought the command to RPi is sent via arduino but it turns out that it is operated via RPi, we left it there as a placeholder for system functionality check through dbprintf and sendok();

```
void handleCommand(TPacket *command) {
  switch (command->command) {
    // Movement commands
    case COMMAND_FORWARD:
      sendOK();
      forward(targetDist, distSpeed);
      break;

    case COMMAND_REVERSE:
      sendOK();
      backward(targetDist, distSpeed);
      break;

    case COMMAND_TURN_LEFT:
      sendOK();
      left(targetAngle, angSpeed);
      break;

    case COMMAND_TURN_RIGHT:
      sendOK();
      right(targetAngle, angSpeed);
      break;

    // Speed settings
    case COMMAND_GEAR_1:  // Low speed/precision mode
      sendOK();
      distSpeed = 70;
      angSpeed = 100;
      targetAngle = 20;
      targetDist = 1;
      break;

    case COMMAND_GEAR_2:  // Medium speed/precision mode
      sendOK();
      distSpeed = 70;
      angSpeed = 100;
      targetAngle = 45;
      targetDist = 5;
      break;

    case COMMAND_GEAR_3:  // High speed/precision mode
      sendOK();
      distSpeed = 100;
      angSpeed = 100;
      targetAngle = 90;
      targetDist = 10;
      break;
```

```
    // Sensor commands
    case COMMAND_ULTRA:  // Ultrasonic distance measurement
      sendOK();
      getDistance();
      break;

    case COMMAND_COLOUR:  // Color detection
      sendOK();
      colourmode();
      break;

    case COMMAND_CAM:  // Camera mode (placeholder,handled by RPi)
      sendOK();
      cammode();
      break;

    // Control commands
    case COMMAND_STOP:  // Stop all movement
      sendOK();
      stop();
      break;

    case COMMAND_SERVO_OPEN:  // Open servo
      sendOK();
      servo_angle(120, 0);
      dbprintf("servo opened!\n");
      break;

    case COMMAND_SERVO_CLOSE:  // Close servo
      sendOK();
      servo_angle(30, 90);
      dbprintf("servo closed!\n");
      break;

    case COMMAND_TURN_AND_OPEN_TRAP:  // Open trap door
      OCR5C = 1000;  // Set servo position to open
      dbprintf("trap opened, NOW, start shaking!\n");
      break;

    case COMMAND_SHAKE:  // Shake movement sequence
      dbprintf("SHAKING");
      sendOK();
      forward(2, 100);
      delay(1000);
      backward(2, 100);
      stop();
```

```
      break;

    // Status commands
    case COMMAND_GET_STATS:  // Request status information
      sendStatus();
      sendOK();
      break;

    case COMMAND_CLEAR_STATS:  // Clear counters
      clearOneCounter(command->params[0]);
      sendOK();
      break;

    default:  // Unknown command
      sendBadCommand();
  }
}
```

Figure 12: HandleCommand function handles the commands sent through TLS on Arduino

Functionality	Key	Description
Forward	W	Send a command to move forward to the RPi, which sends the data to the arduino through serial communication. Distance and speed is predefined by the gears.
Backward	S	Send a command to move backward to the RPi, which sends the data to the arduino through serial communication. Distance and speed is predefined by the gears.
Left	A	Send a command to move left to the RPi, which sends the data to the arduino through serial communication. Angle and speed is predefined by the gears.
Right	D	Send a command to move right to the RPi, which sends the data to the arduino through serial communication. Angle and speed is predefined by the gears.
Stop	Y	Send a command to stop to the RPi, which sends the data to the arduino through serial communication.
Gear 1	F	This gear is to be used when traversing tight spaces at a slower speed with high precision.

		Setting parameters: Distance/Speed: 2cm/50% Angle/Speed: 10 deg/70%
Gear 2	G	This gear is to be used when traversing slightly wider spaces, balancing speed and precision. Setting parameters: Distance/Speed: 5cm/70% Angle/Speed: 20 deg/70%
Gear 3	H	This gear is to be used when traversing wide spaces at a faster speed with low precision. Setting parameters: Distance/Speed: 10cm/100% Angle/Speed: 30 deg/70%
Ultrasonic data	Z	Send an ultrasonic data request to the RPi, which sends it to the arduino through serial communication. The arduino then sends back the data to the RPi then to the laptop , printing it on the screen.
Colour sensor data	C	Send a colour sensor data request to the RPi, which sends it to the arduino through serial communication. The arduino then sends back the data to the RPi then to the laptop , printing it on the screen.
Servo open	Q	Send a command to open the servo arms to the RPi, which sends the data to the arduino through serial communication.
Servo close	E	Send a command to close the servo arms to the RPi, which sends the data to the arduino through serial communication.
Get data	R	Sends a data request to Rpi which then requests data from the Arduino. The data will subsequently be printed on the user's laptop
Clear data	T	Sends a reset data command to Rpi which requests Arduino to clear its data.
Quit	V	Disconnects the client

Figure 13: Keybindings table

```
void ultraSetup(void) {

    DDRA  |=  (1 << PA3); // configure PA3 (Arduino D25) as output
(trigger pin)
    PORTA &= ~(1 << PA3); // drive PA3 low

    //configure PA4 (Arduino D26) as input (echo pin)
    DDRA  &= ~(1 << PA4);

    // Timer1: Normal mode, prescaler = 8 (0.5 µs per tick)
    TCCR1A = 0;
    TCCR1B = (1 << CS11);
}

void getDistance() {
    int ticks, time, distance_cm;

    //recommended by datasheet to hold trigger pin down for 2µs
before sending output pulse
    PORTA &= ~(1 << PA3);
    _delay_us(2);
    // send 10 µs trigger pulse on PA3
    PORTA |=  (1 << PA3);
    _delay_us(10);
    PORTA &= ~(1 << PA3);

    // wait for echo rising edge on PA4
    while (!(PINA & (1 << PA4)));
    TCNT1 = 0;

    // wait for echo falling edge
    while  (PINA & (1 << PA4));
    ticks = TCNT1;

    time = (ticks/2); // calculate time in µs using timer ticks
    distance_cm = (time * 0.034)/2; //calculate distance in cm

    //offset distance by value obtained in calibration
    if (distance_cm > offset) distance_cm -= offset;
    else distance_cm = 0;

    dbprintf("DIST:%u\n", distance_cm);//using dbprintf() function
which uses sendMessage() to send distance via 32-byte “data” array
in data packet
```

```
}
```

Figure 14: Ultrasonic sensor code

Colour Sensor

```
// I2C address of the TCS34725 color sensor (7-bit address)
#define TCS34725_ADDR 0x29

// Command bit mask for accessing registers (must be ORed with
register address)
#define TCS34725_CMD 0x80

// Command bit for enabling auto-increment when reading/writing
multiple registers
#define TCS34725_CMD_AUTO_INC 0xA0

// Desired I2C clock speed set to 100 kHz
#define F_SCL 100000UL
```

Figure 15: Bare-metal code on Macros

```
void setup(){
  // put your setup code here, to run once:
  cli();
  setupEINT();
  setupSerial();
  startSerial();
  enablePullups();
  ultraSetup();
  init_servo(); //timer 5 init servo baremetal
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  digitalWrite(TRIG_PIN, LOW);
  DDRD &= ~((1 << PD0) | (1 << PD1));
  // Configure PD0 (SCL) and PD1 (SDA) as inputs for I2C
  PORTD |= (1 << PD0) | (1 << PD1);
  // Enable internal pull-up resistors for SDA/SCL lines
  initializeState();
  sei();
}
```

Figure 16: Setting up the ports for SCL and SDA pins

The setup() function is responsible for initializing all the essential hardware components and configurations before the robot begins operation. It begins by disabling global interrupts using cli() to safely configure peripherals, followed by setting up external interrupts, serial communication, internal pull-up resistors, and initializing the servo using timer 5 in bare-metal mode. The ultrasonic sensor is also set up here by configuring its trigger as an output and echo as an input. Additionally, pull-ups are enabled for PD0 and PD1 to ensure stable logic levels. Finally, the robot's initial state is set, and global interrupts are re-enabled with sei() to allow normal operation.

```
// --- Bare-metal I2C/TWI Functions ---
void I2C_Init() {
    TWSR = 0x00; // No prescaler
    TWBR = ((F_CPU / F_SCL) - 16) / 2; // Set bit rate for 100kHz
}

void I2C_Start() {
    TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN); // Send
START
    while (!(TWCR & (1 << TWINT)));
}

void I2C_Stop() {
    TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN); // Send STOP
}

void I2C_Write(uint8_t data) {
    TWDR = data;
    TWCR = (1 << TWINT) | (1 << TWEN); // Write data
    while (!(TWCR & (1 << TWINT)));
}

uint8_t I2C_ReadACK() {
    TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA); // Read with
ACK
    while (!(TWCR & (1 << TWINT)));
    return TWDR;
}

uint8_t I2C_ReadNACK() {
    TWCR = (1 << TWINT) | (1 << TWEN); // Read without ACK
    while (!(TWCR & (1 << TWINT)));
    return TWDR;
}

// --- TCS34725 Functions ---
```

```
void TCS34725_WriteReg(uint8_t reg, uint8_t val) {
    I2C_Start();
    I2C_Write((TCS34725_ADDR << 1) | 0);
    I2C_Write(TCS34725_CMD | reg);
    I2C_Write(val);
    I2C_Stop();
}

uint8_t TCS34725_ReadReg(uint8_t reg) {
    I2C_Start();
    I2C_Write((TCS34725_ADDR << 1) | 0);
    I2C_Write(TCS34725_CMD | reg);
    I2C_Start();
    I2C_Write((TCS34725_ADDR << 1) | 1);
    uint8_t data = I2C_ReadNACK();
    I2C_Stop();
    return data;
}

void TCS34725_Init() {
    TCS34725_WriteReg(0x00, 0x01); // Power on
    _delay_ms(3);
    TCS34725_WriteReg(0x00, 0x03); // Enable RGBC
    TCS34725_WriteReg(0x01, 0xFF); // Integration time
    TCS34725_WriteReg(0x0F, 0x01); // Gain = 4x
}

void TCS34725_ReadColors(uint16_t *r, uint16_t *g, uint16_t *b,
uint16_t *c) {
    I2C_Start();
    I2C_Write((TCS34725_ADDR << 1) | 0);
    I2C_Write(TCS34725_CMD | TCS34725_CMD_AUTO_INC | 0x14); //
Start from CDATAL
    I2C_Start();
    I2C_Write((TCS34725_ADDR << 1) | 1);

    *c = I2C_ReadACK(); *c |= (I2C_ReadACK() << 8);
    *r = I2C_ReadACK(); *r |= (I2C_ReadACK() << 8);
    *g = I2C_ReadACK(); *g |= (I2C_ReadACK() << 8);
    *b = I2C_ReadACK(); *b |= (I2C_ReadNACK() << 8);

    I2C_Stop();
}

void colourmode() {
    uint16_t r, g, b, c;
    static bool initialized = false;
```

```
    if (!initialized) {
        I2C_Init();
        TCS34725_Init();
        initialized = true;
        _delay_ms(500); // First read delay
    }

    TCS34725_ReadColors(&r, &g, &b, &c);

    if (c > 0) {
        float rn = (float)r / c;
        float gn = (float)g / c;
        float bn = (float)b / c;

        if (rn > gn && rn > bn && rn > 0.4) {
            dbprintf("COLOR:RED");
        } else if (gn > rn && gn > bn && gn > 0.4) {
            dbprintf("COLOR:GREEN");
        } else {
            dbprintf("COLOR:NONE");
        }
        dbprintf(" R:%d G:%d B:%d C:%d", r, g, b, c);
    } else {
        dbprintf("COLOR:NO_LIGHT");
    }
}
```

Figure 17: Functions + Colour Interpretation

This code enables I2C-based communication with the TCS34725 color sensor using bare-metal AVR code.

It defines low-level I2C functions (Start, Stop, Write, ReadACK/NACK) and uses them to configure and read from the sensor. The TCS34725_Init() function powers on the device, sets integration time, and gain for color detection. TCS34725_ReadColors() fetches 16-bit values for clear (C), red (R), green (G), and blue (B) light.

The colourmode() function initializes the sensor once, reads the RGBC data, normalizes the RGB values, and classifies the color as RED, GREEN, NONE, or NO_LIGHT based on threshold logic. This modular approach makes it suitable for use in real-time robotic systems especially in Alex's rescue where the environment's light intensity varies with the shadows casted by the wall etc., lots of fine-tuning is needed.

Register	Value	Setting	Readings	Description
Integration Time				
0x01	0xFF	700 ms Integration	~1.4 Hz	Longest integration time, best sensitivity
0x01	0xF6	400 ms Integration	2.5 Hz	High sensitivity, good for low-light scenes
0x01	0xD5	200 ms Integration	5 Hz	Balanced setting for general use
0x01	0xC0	100 ms Integration	10 Hz	Faster response, less sensitivity
0x01	0x80	50 ms Integration	20 Hz	Good for bright environments
0x01	0x40	24 ms Integration	~41 Hz	Very fast, low light resolution
0x01	0x00	2.4 ms Integration	~400 Hz	Ultra-fast, minimal light detection
Gain Control				
0x0F	0x00	1× Gain	—	No amplification
0x0F	0x01	4× Gain	—	Mild amplification, general use
0x0F	0x02	16× Gain	—	Strong amplification for low light
0x0F	0x03	60× Gain	—	Max amplification, very dark scenes

Figure 18: Calibration Checklist

Servos

Timer	Channel	ATmega Pin Name	Arduino Pin
Timer 0	OC0A	PB7	13
	OC0B	PG5	4
Timer 1	OC1A	PB5	11
	OC1B	PB6	12

	OC1C	PB7	13
Timer 2	OC2A	PB4	10
	OC2B	PH6	9
Timer 3	OC3A	PE3	5
	OC3B	PE4	2
	OC3C	PE5	3
Timer 4	OC4A	PH3	6
	OC4B	PH4	7
	OC4C	PH5	8
Timer 5	OC5A	PL3	46
	OC5B	PL4	45
	OC5C	PL5	44

Figure 19: Table for Pin-Outs on all timers

```
void init_servo() {
  // Set PL3, PL4, and PL5 as output pins for Timer5 channels A,
B, and C
  DDRL |= (1 << PL3); // Set pin 46 (OC5A) as output
  DDRL |= (1 << PL4); // Set pin 45 (OC5B) as output
  DDRL |= (1 << PL5); // Set pin 44 (OC5C) as output

  // Configure Timer5 for Phase Correct PWM mode using ICR5 as TOP
  // Enable non-inverting mode for OC5A, OC5B, and OC5C
  // WGM5[3:0] = 1000 => Phase and Frequency Correct PWM, TOP =
ICR5
  TCCR5A = (1 << COM5A1)
  // Enable PWM on OC5A (Pin 46)
         | (1 << COM5B1)
  // Enable PWM on OC5B (Pin 45)
         | (1 << COM5C1)
  // Enable PWM on OC5C (Pin 44)
         | (1 << WGM51);
  // Part of WGM5[3:0] = 1000 (WGM53 and WGM51)
```

```
  TCCR5B = (1 << WGM53);
  // Complete WGM settings: WGM53:WGM50 = 10:00 (Phase Correct
PWM)

  ICR5 = 20000; // Set TOP value to 20000 (for 20ms period –
typical for servos)

  // Set initial duty cycles (pulse widths in timer counts)
  OCR5A = 2000;  // Servo on OC5A: 2ms pulse (e.g., full left)
  OCR5B = 1000;  // Servo on OC5B: 1ms pulse (e.g., full right)
  OCR5C = 1500;  // Servo on OC5C: 1.5ms pulse (neutral center)

  // Start the timer with prescaler of 8
  // Timer clock = 16 MHz / 8 = 2 MHz → 1 count = 0.5 µs → 20000
counts = 10ms in Phase Correct mode (=> 20ms full cycle)
  TCCR5B |= (1 << CS51); // Set prescaler to 8
}
```

Figure 20: Servo-Initiation Code (All 3 Servos)

```
void servo_angle(const int left_angle, const int right_angle) {
// Example usage: servo_angle(20, 160)

// Check if the input angles are within the valid range (0 to 180
degrees)
  if (left_angle > 180 || left_angle < 0 || right_angle > 180 ||
right_angle < 0) {
    return;
// Exit the function if angles are invalid
  }

// Convert angle (0–180°) to PWM value (1000–2000 microseconds)
// 0° corresponds to 1000 µs, 90° to 1500 µs, 180° to 2000 µs
// Set the PWM duty cycle to control servo position:
  OCR5A = 1500 + (left_angle * 1000 / 180);
// Set PWM for left servo (OC5A pin)
  OCR5B = 1500 + (right_angle * 1000 / 180);
// Set PWM for right servo (OC5B pin)
}
```

Figure 21: Claw-Grabbing Code for Claw

```
case COMMAND_SERVO_OPEN:
    sendOK();  // Send OK response to acknowledge command
    servo_angle(120, 0);  // Set servo to open position (120
degrees, angle 0)
    dbprintf("servo opened!\n");  // Debug message for servo
opening
    break;

case COMMAND_SERVO_CLOSE:
    sendOK();  // Send OK response to acknowledge command
    servo_angle(30, 90);  // Set servo to closed position (30
degrees, angle 90)
    dbprintf("servo closed!\n");  // Debug message for servo
closing
    break;

// Claw operation
case COMMAND_TURN_AND_OPEN_TRAP:
    OCR5C = 1000; // OPEN (adjust value as needed) - Control servo
to open the trap
    dbprintf("trap opened, NOW, start shaking!\n");  // Debug
message for trap opening
    break;

// Trapdoor operation (commented out section needs to be added)
```

Figure 22: Command-Interpretation for Claw&Trapdoor

Software Design

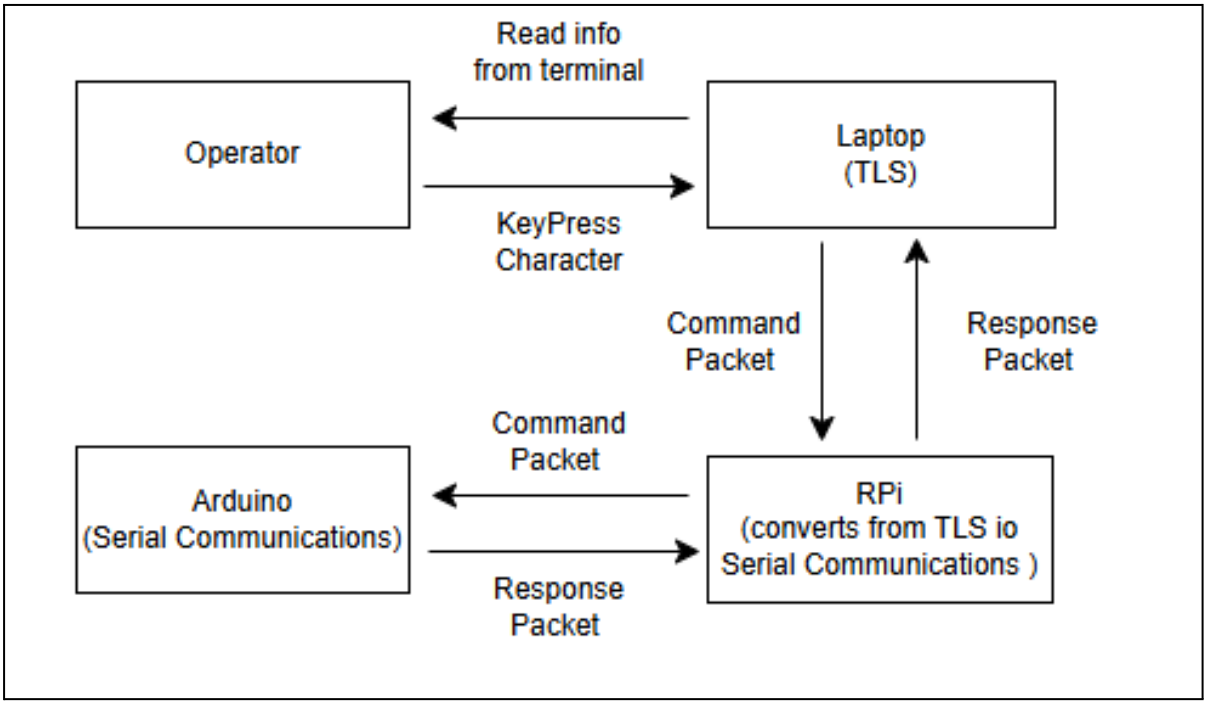


Figure 23: Pictorial Representation of data flow

```
 if (read(STDIN_FILENO, &ch, 1) > 0) {
 printf("You typed: %c\n", ch);
 current_time =clock();
double time_diff = (double)(current_time - last_time)/CLOCKS_PER_SEC;
if (last_time ==0 || time_diff > 3){
```

Figure 24: Code for 3 second buffer

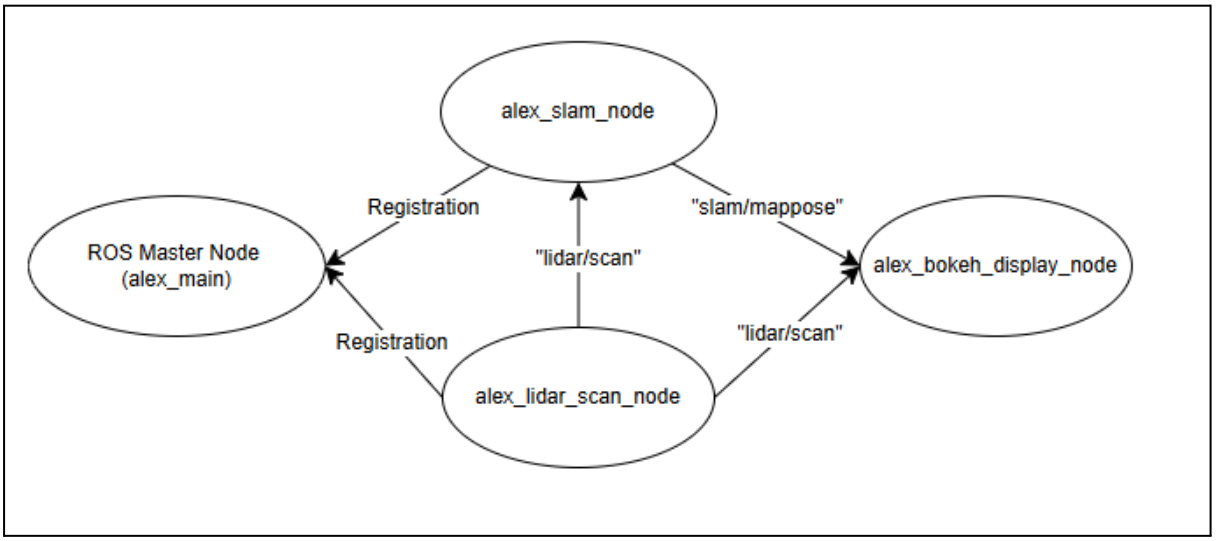


Figure 25: RQT graph of Lidar map publishing

Full Alex.ino code

```
/*
 * THE ULTIMATE ALEXA
 */

#include <serialize.h>
#include "packeto.h"
#include "constanto.h"
#include "stdarg.h"
#include <math.h>
#include <avr/io.h>

/*
 * ========== CONFIGURATION CONSTANTS ==========
 */
volatile TDirection dir;   // Current direction of movement

// Physical dimensions of the robot
#define ALEX_LENGTH 12     // Length of robot in cm
#define ALEX_BREADTH 14    // Width of robot in cm

// Wheel encoder configuration
#define COUNTS_PER_REV 5   // Number of ticks per wheel revolution
#define WHEEL_CIRC 20      // Wheel circumference in cm

// Ultrasonic sensor pins
#define TRIG_PIN 25        // Trigger pin for ultrasonic sensor
#define ECHO_PIN 26        // Echo pin for ultrasonic sensor


// I2C color sensor (TCS34725) configuration
#define TCS34725_ADDR 0x29         // I2C address of the TCS34725 sensor
#define TCS34725_CMD 0x80          // Command bit mask for register access
#define TCS34725_CMD_AUTO_INC 0xA0 // Auto-increment bit for reading multiple
registers
#define F_SCL 100000UL             // I2C clock speed (100 kHz)

int offset = 0;  // Ultrasonic sensor calibration offset

/*
 * ========== STATE VARIABLES ==========
```

```
 */
// Alexa's geometry calculations
volatile float alexDiagonal = 18.44;  // Diagonal measurement of robot
volatile float alexCirc = 57.93;      // Circumference for turning
calculations

// Encoder tick counters for straight movement
volatile unsigned long leftForwardTicks;
volatile unsigned long rightForwardTicks;
volatile unsigned long leftReverseTicks;
volatile unsigned long rightReverseTicks;

// Encoder tick counters for turning movements
volatile unsigned long leftForwardTicksTurns;
volatile unsigned long rightForwardTicksTurns;
volatile unsigned long leftReverseTicksTurns;
volatile unsigned long rightReverseTicksTurns;

// Distance tracking variables
volatile unsigned long forwardDist;   // Forward distance traveled in cm
volatile unsigned long reverseDist;   // Reverse distance traveled in cm
unsigned long deltaDist;              // Distance left to travel
unsigned long newDist;                // Target distance to reach
unsigned long deltaTicks;             // Ticks left to travel
unsigned long targetTicks;            // Target ticks to reach

// Movement parameters - configurable with gear commands
int angSpeed = 100;       // Angular speed (0-100%)
int distSpeed = 70;       // Linear speed (0-100%)
float targetAngle = 45;   // Default target angle in degrees
float targetDist = 5;     // Default target distance in cm

// Ultrasonic sensor timing
unsigned long lastUltrasonicReport = 0;
unsigned long currentMillis;

/*
 * ========== MOVEMENT CONTROL FUNCTIONS ==========
 */

/**
 * Turn left by specified angle at specified speed
 *
 * @param ang - angle to turn in degrees
 * @param speed - speed as percentage (0-100)
 */
void left(float ang, float speed) {
```

```
  if (ang == 0) {
    stop();
    dbprintf("ang 0 detected");
  } else {
    deltaTicks = computeDeltaTicks(ang);
    targetTicks = leftReverseTicksTurns + deltaTicks;
    ccw(ang, speed);
  }
}

/**
 * Turn right by specified angle at specified speed
 *
 * @param ang - angle to turn in degrees
 * @param speed - speed as percentage (0-100)
 */
void right(float ang, float speed) {
  if (ang == 0) {
    stop();
    dbprintf("ang 0 detected");
  } else {
    deltaTicks = computeDeltaTicks(ang);
    targetTicks = rightReverseTicksTurns + deltaTicks;
    cw(ang, speed);
  }
}

/**
 * Calculate the number of wheel encoder ticks needed for a turn
 *
 * @param ang - angle to turn in degrees
 * @return number of encoder ticks required
 */
unsigned long computeDeltaTicks(float ang) {
  // Calculate ticks needed for specified angle
  // Formula: (ang * alexCirc * COUNTS_PER_REV) / (360 * WHEEL_CIRC)
  unsigned long ticks = (unsigned long)(round((ang * alexCirc * 5) / (360.0 *
19.8)));
  return ticks;
}

/*
 * ========== COMMUNICATION FUNCTIONS ==========
 */

/**
 * Read a packet from the serial port
```

```
 *
 * @param packet - pointer to store the received packet
 * @return status code indicating success or error
 */
TResult readPacket(TPacket *packet) {
  char buffer[PACKET_SIZE];
  int len;

  len = readSerial(buffer);

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

/**
 * Send Alexa status information to the Pi
 * Includes encoder ticks and distance traveled
 */
void sendStatus() {
  TPacket statusPacket;                           // Create new packet
  statusPacket.packetType = PACKET_TYPE_RESPONSE;  // Set packet type
  statusPacket.command = RESP_STATUS;             // Set command field

  // Populate params array with current status data
  statusPacket.params[0] = leftForwardTicks;
  statusPacket.params[1] = rightForwardTicks;
  statusPacket.params[2] = leftReverseTicks;
  statusPacket.params[3] = rightReverseTicks;
  statusPacket.params[4] = leftForwardTicksTurns;
  statusPacket.params[5] = rightForwardTicksTurns;
  statusPacket.params[6] = leftReverseTicksTurns;
  statusPacket.params[7] = rightReverseTicksTurns;
  statusPacket.params[8] = forwardDist;
  statusPacket.params[9] = reverseDist;

  sendResponse(&statusPacket);
}

/**
 * Debug print function that sends formatted text messages back to the Pi
 *
 * @param format - format string (like printf)
 * @param ... - variable arguments to format
 */
void dbprintf(const char *format, ...) {
```

```
  va_list args;
  char buffer[128];
  va_start(args, format);
  vsprintf(buffer, format, args);
  sendMessage(buffer);
}

/**
 * Send a text message back to the Pi
 *
 * @param message - the message string to send
 */
void sendMessage(const char *message) {
  TPacket messagePacket;
  messagePacket.packetType = PACKET_TYPE_MESSAGE;
  strncpy(messagePacket.data, message, MAX_STR_LEN);
  sendResponse(&messagePacket);
}

/**
 * Send error response for bad magic number
 */
void sendBadPacket() {
  TPacket badPacket;
  badPacket.packetType = PACKET_TYPE_ERROR;
  badPacket.command = RESP_BAD_PACKET;
  sendResponse(&badPacket);
}

/**
 * Send error response for bad checksum
 */
void sendBadChecksum() {
  TPacket badChecksum;
  badChecksum.packetType = PACKET_TYPE_ERROR;
  badChecksum.command = RESP_BAD_CHECKSUM;
  sendResponse(&badChecksum);
}

/**
 * Send error response for bad command
 */
void sendBadCommand() {
  TPacket badCommand;
  badCommand.packetType = PACKET_TYPE_ERROR;
  badCommand.command = RESP_BAD_COMMAND;
  sendResponse(&badCommand);
```

```
}

/**
 * Send error response for bad response
 */
void sendBadResponse() {
  TPacket badResponse;
  badResponse.packetType = PACKET_TYPE_ERROR;
  badResponse.command = RESP_BAD_RESPONSE;
  sendResponse(&badResponse);
}

/**
 * Send OK response
 */
void sendOK() {
  TPacket okPacket;
  okPacket.packetType = PACKET_TYPE_RESPONSE;
  okPacket.command = RESP_OK;
  sendResponse(&okPacket);
}

/**
 * Serialize and send a packet over serial
 *
 * @param packet - pointer to the packet to send
 */
void sendResponse(TPacket *packet) {
  char buffer[PACKET_SIZE];
  int len;

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

/*
 * ========== INTERRUPT HANDLING ==========
 */

/**
 * Enable pull-up resistors on encoder pins
 * Uses bare-metal code to configure PD2 and PD3 pins
 */
void enablePullups() {
  // Set pins 18 and 19 (PD2 and PD3) as inputs
  DDRD &= ~(0b00001100);  // Clear bits 2 and 3 for input
  PORTD |= 0b00001100;    // Set pullup resistors on those pins
```

```
}

/**
 * ISR for external interrupt 3 (left wheel encoder)
 */
ISR(INT3_vect) {
  leftISR();
}

/**
 * ISR for external interrupt 2 (right wheel encoder)
 */
ISR(INT2_vect) {
  rightISR();
}

/**
 * Left wheel encoder interrupt service routine
 * Updates appropriate tick counters based on direction
 */
void leftISR() {
  if (dir == FORWARD) {
    leftForwardTicks++;
    forwardDist = (unsigned long)((float)leftForwardTicks / COUNTS_PER_REV *
WHEEL_CIRC);
  } else if (dir == BACKWARD) {
    leftReverseTicks++;
    reverseDist = (unsigned long)((float)leftReverseTicks / COUNTS_PER_REV *
WHEEL_CIRC);
  } else if (dir == LEFT) {
    leftReverseTicksTurns++;
  } else if (dir == RIGHT) {
    leftForwardTicksTurns++;
  }
}

/**
 * Right wheel encoder interrupt service routine
 * Updates appropriate tick counters based on direction
 */
void rightISR() {
  if (dir == FORWARD) {
    rightForwardTicks++;
  } else if (dir == BACKWARD) {
    rightReverseTicks++;
  } else if (dir == LEFT) {
    rightForwardTicksTurns++;
```

```
  } else if (dir == RIGHT) {
    rightReverseTicksTurns++;
  }
}

/**
 * Configure external interrupts for wheel encoders
 * Sets up INT2 and INT3 for falling edge trigger
 */
void setupEINT() {
  EIMSK = 0b00001100;  // Enable INT2 and INT3 interrupts
  EICRA = 0b10100000;  // Set INT2 and INT3 for falling edge trigger
}

/*
 * ========== SERIAL COMMUNICATION SETUP ==========
 */

/**
 * Configure the serial port using bare-metal code
 * Sets up USART0 for 9600 baud 8N1 mode
 */
void setupSerial() {
  PRR0 &= ~(1 << 1);         // Disable power reduction for USART0
  uint16_t ubrr_value = 103; // Set baud rate to 9600 ((16000000/16*9600) - 1
= 103)
  UBRR0H = (ubrr_value >> 8); // Set high byte of baud rate register
  UBRR0L = ubrr_value;       // Set low byte of baud rate register
  UCSR0C = (1 << UCSZ01) | (1 << UCSZ00); // 8-bit data, no parity, 1 stop
bit
  UCSR0A = 0;                // Clear USART control register A
}

/**
 * Enable USART0 transmitter and receiver
 */
void startSerial() {
  UCSR0B = (1 << RXEN0) | (1 << TXEN0); // Enable USART receive and transmit
}

/**
 * Read available data from serial port
 *
 * @param buffer - buffer to store received data
 * @return number of bytes read
 */
int readSerial(char *buffer) {
```

```
  int count = 0;
  // Read data when available in the receive buffer
  while (UCSR0A & (1 << RXC0))
    buffer[count++] = UDR0; // Read data from UDR0 register

  return count;
}

/**
 * Write data to serial port
 *
 * @param buffer - data to send
 * @param len - length of data in bytes
 */
void writeSerial(const char *buffer, int len) {
  for(int i = 0; i < len; i++) {
    // Wait until data register is empty
    while (!(UCSR0A & (1 << UDRE0)));
    // Write data to UDR0 register
    UDR0 = buffer[i];
  }
}

/*
 * ========== STATE MANAGEMENT ==========
 */

/**
 * Reset all encoder counters and distance variables
 */
void clearCounters() {
  leftForwardTicks = 0;
  rightForwardTicks = 0;
  leftReverseTicks = 0;
  rightReverseTicks = 0;
  leftForwardTicksTurns = 0;
  rightForwardTicksTurns = 0;
  leftReverseTicksTurns = 0;
  rightReverseTicksTurns = 0;
  forwardDist = 0;
  reverseDist = 0;
  targetTicks = 0;
}

/**
 * Clear a specific counter (currently just clears all)
 *
```

```
 * @param which - index of counter to clear
 */
void clearOneCounter(int which) {
  clearCounters();
  /*
  // Original switch-case implementation left as comment
  switch(which)
  {
    case 0:
      clearCounters();
      break;
    case 1:
      leftTicks=0;
      break;
    case 2:
      rightTicks=0;
      break;
    case 3:
      leftRevs=0;
      break;
    case 4:
      rightRevs=0;
      break;
    case 5:
      forwardDist=0;
      break;
    case 6:
      reverseDist=0;
      break;
  }*/
}

/**
 * Initialize Alex's internal state
 */
void initializeState() {
  clearCounters();
}

/*
 * ========== SENSOR FUNCTIONS ==========
 */

/**
 * Measure distance using ultrasonic sensor
 * Reports the distance via debug print
 */
```

```
void ultraSetup(void) {
    // – configure PA3 (Arduino D25) as output, drive it LOW –
    DDRA  |=  (1 << PA3);
    PORTA &= ~(1 << PA3);

    // – configure PA4 (Arduino D26) as input (echo pin) –
    DDRA  &= ~(1 << PA4);

    // – Timer1: Normal mode, prescaler = 8 (0.5 µs per tick) –
    TCCR1A = 0;
    TCCR1B = (1 << CS11);
}

void getDistance() {
    int ticks, time, distance_cm;

    // – send 10 µs trigger pulse on PA3 –
    PORTA &= ~(1 << PA3);
    _delay_us(2);
    PORTA |=  (1 << PA3);
    _delay_us(10);
    PORTA &= ~(1 << PA3);

    // – wait for echo rising edge on PA4 –
    while (!(PINA & (1 << PA4)));
    TCNT1 = 0;

    // – wait for echo falling edge –
    while  (PINA & (1 << PA4));
    ticks = TCNT1;

    time = (ticks/2);
    distance_cm = (time * 0.034)/2;
    if (distance_cm > offset) distance_cm -= offset;
    else distance_cm = 0;

    dbprintf("DIST:%u\n", distance_cm);
}



/**
 * Empty function for camera mode
 * (Camera is on RPi, not Arduino)
 */
void cammode() {
```

```
  ;  // Function intentionally left empty as we thought we needed to use RPI
via arduino before the studio
}

/*
 * ========== COMMAND HANDLING ==========
 */

/**
 * Process commands received from the Pi
 *
 * @param command - pointer to the command packet
 */
void handleCommand(TPacket *command) {
  switch (command->command) {
    // Movement commands
    case COMMAND_FORWARD:
      sendOK();
      forward(targetDist, distSpeed);
      break;

    case COMMAND_REVERSE:
      sendOK();
      backward(targetDist, distSpeed);
      break;

    case COMMAND_TURN_LEFT:
      sendOK();
      left(targetAngle, angSpeed);
      break;

    case COMMAND_TURN_RIGHT:
      sendOK();
      right(targetAngle, angSpeed);
      break;

    // Speed settings
    case COMMAND_GEAR_1:  // Low speed/precision mode
      sendOK();
      distSpeed = 70;
      angSpeed = 100;
      targetAngle = 20;
      targetDist = 1;
      break;

    case COMMAND_GEAR_2:  // Medium speed/precision mode
      sendOK();
```

```
      distSpeed = 70;
      angSpeed = 100;
      targetAngle = 45;
      targetDist = 5;
      break;

    case COMMAND_GEAR_3:  // High speed/precision mode
      sendOK();
      distSpeed = 100;
      angSpeed = 100;
      targetAngle = 90;
      targetDist = 10;
      break;

    // Sensor commands
    case COMMAND_ULTRA:  // Ultrasonic distance measurement
      sendOK();
      getDistance();
      break;

    case COMMAND_COLOUR:  // Color detection
      sendOK();
      colourmode();
      break;

    case COMMAND_CAM:  // Camera mode (handled by RPi)
      sendOK();
      cammode();
      break;

    // Control commands
    case COMMAND_STOP:  // Stop all movement
      sendOK();
      stop();
      break;

    case COMMAND_SERVO_OPEN:  // Open servo
      sendOK();
      servo_angle(120, 0);
      dbprintf("servo opened!\n");
      break;

    case COMMAND_SERVO_CLOSE:  // Close servo
      sendOK();
      servo_angle(30, 90);
      dbprintf("servo closed!\n");
      break;
```

```
    case COMMAND_TURN_AND_OPEN_TRAP:  // Open trap door
      OCR5C = 1000;  // Set servo position to open
      dbprintf("trap opened, NOW, start shaking!\n");
      break;

    case COMMAND_SHAKE:  // Shake movement sequence
      dbprintf("SHAKING");
      sendOK();
      forward(2, 100);
      delay(1000);
      backward(2, 100);
      stop();
      break;

    // Status commands
    case COMMAND_GET_STATS:  // Request status information
      sendStatus();
      sendOK();
      break;

    case COMMAND_CLEAR_STATS:  // Clear counters
      clearOneCounter(command->params[0]);
      sendOK();
      break;

    default:  // Unknown command
      sendBadCommand();
  }
}

/**
 * Wait for initial connection handshake from Pi
 */
void waitForHello() {
  int exit = 0;

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

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

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

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

/*
 * ========== INITIALIZATION ==========
 */

/**
 * Arduino setup function - runs once at startup
 */
void setup() {
  // Disable interrupts during setup
  cli();

  // Initialize hardware components
  setupEINT();       // Set up external interrupts
  setupSerial();     // Set up serial communication
  startSerial();     // Start serial communication
  enablePullups();   // Enable pull-up resistors on encoder pins
  init_servo();      // Initialize servo motor using Timer5
  ultraSetup();      // Initialize ultrasonic sensor using Timer1 and pin 25
and 26

  // Set up ultrasonic sensor pins
  pinMode(TRIG_PIN, OUTPUT);
  pinMode(ECHO_PIN, INPUT);
  digitalWrite(TRIG_PIN, LOW);

  // Set up pull-ups for PD0 and PD1
  DDRD &= ~((1 << PD0) | (1 << PD1));  // Set as INPUT
  PORTD |= (1 << PD0) | (1 << PD1);    // Enable pull-ups

  // Initialize robot state
  initializeState();

  // Enable interrupts
  sei();
}
```

```
/**
 * Process different packet types received from Pi
 *
 * @param packet - pointer to the received packet
 */
void handlePacket(TPacket *packet) {
  switch (packet->packetType) {
    case PACKET_TYPE_COMMAND:
      handleCommand(packet);
      break;

    case PACKET_TYPE_RESPONSE:
      break;

    case PACKET_TYPE_ERROR:
      break;

    case PACKET_TYPE_MESSAGE:
      break;

    case PACKET_TYPE_HELLO:
      break;
  }
}

/**
 * Initialize servo motors using Timer5
 * Sets up PWM for servo control
 */
void init_servo() {
  // Set pins for Timer5 PWM output channels
  DDRL |= (1 << PL3);  // Pin 46 (OC5A) as output
  DDRL |= (1 << PL4);  // Pin 45 (OC5B) as output
  DDRL |= (1 << PL5);  // Pin 44 (OC5C) as output

  // Configure Timer5 for Phase Correct PWM mode
  // WGM5[3:0] = 1000 => Phase and Frequency Correct PWM, TOP = ICR5
  TCCR5A = (1 << COM5A1)  // Enable PWM on OC5A (Pin 46)
         | (1 << COM5B1)  // Enable PWM on OC5B (Pin 45)
         | (1 << COM5C1)  // Enable PWM on OC5C (Pin 44)
         | (1 << WGM51);  // Part of WGM5[3:0] = 1000

  TCCR5B = (1 << WGM53);  // Complete WGM settings: WGM53:WGM50 = 10:00

  // Set TOP value for 20ms period (standard for servos)
  ICR5 = 20000;
```

```
  // Set initial duty cycles for servos
  OCR5A = 2000;  // Servo A: 2ms pulse (e.g., full left)
  OCR5B = 1000;  // Servo B: 1ms pulse (e.g., full right)
  OCR5C = 1500;  // Servo C: 1.5ms pulse (center position)

  // Start Timer5 with prescaler of 8
  // Timer clock = 16 MHz / 8 = 2 MHz
  // 1 count = 0.5 µs, 20000 counts = 10ms in Phase Correct mode (20ms full
cycle)
  TCCR5B |= (1 << CS51);  // Set prescaler to 8
}

/*
 * ========== I2C/TWI FUNCTIONS FOR COLOR SENSOR ==========
 */

/**
 * Initialize I2C (TWI) interface
 */
void I2C_Init() {
  TWSR = 0x00;  // No prescaler
  TWBR = ((F_CPU / F_SCL) - 16) / 2;  // Set bit rate for 100kHz
}

/**
 * Send I2C START condition
 */
void I2C_Start() {
  TWCR = (1 << TWINT) | (1 << TWSTA) | (1 << TWEN);  // Send START
  while (!(TWCR & (1 << TWINT)));  // Wait for completion
}

/**
 * Send I2C STOP condition
 */
void I2C_Stop() {
  TWCR = (1 << TWINT) | (1 << TWSTO) | (1 << TWEN);  // Send STOP
}

/**
 * Write a byte to I2C bus
 *
 * @param data - byte to write
 */
void I2C_Write(uint8_t data) {
  TWDR = data;  // Load data into data register
  TWCR = (1 << TWINT) | (1 << TWEN);  // Start transmission
```

```
  while (!(TWCR & (1 << TWINT)));  // Wait for completion
}

/**
 * Read a byte from I2C bus with ACK
 *
 * @return byte read from I2C bus
 */
uint8_t I2C_ReadACK() {
  TWCR = (1 << TWINT) | (1 << TWEN) | (1 << TWEA);  // Read with ACK
  while (!(TWCR & (1 << TWINT)));  // Wait for completion
  return TWDR;  // Return received data
}

/**
 * Read a byte from I2C bus with NACK
 *
 * @return byte read from I2C bus
 */
uint8_t I2C_ReadNACK() {
  TWCR = (1 << TWINT) | (1 << TWEN);  // Read without ACK
  while (!(TWCR & (1 << TWINT)));  // Wait for completion
  return TWDR;  // Return received data
}

/*
 * ========== TCS34725 COLOR SENSOR FUNCTIONS ==========
 */

/**
 * Write to TCS34725 register
 *
 * @param reg - register address
 * @param val - value to write
 */
void TCS34725_WriteReg(uint8_t reg, uint8_t val) {
  I2C_Start();
  I2C_Write((TCS34725_ADDR << 1) | 0);  // Write address
  I2C_Write(TCS34725_CMD | reg);  // Register with command bit
  I2C_Write(val);  // Write value
  I2C_Stop();
}

/**
 * Read from TCS34725 register
 *
 * @param reg - register address
```

```
 * @return value read from register
 */
uint8_t TCS34725_ReadReg(uint8_t reg) {
  I2C_Start();
  I2C_Write((TCS34725_ADDR << 1) | 0);  // Write address
  I2C_Write(TCS34725_CMD | reg);  // Register with command bit
  I2C_Start();  // Repeated start
  I2C_Write((TCS34725_ADDR << 1) | 1);  // Read address
  uint8_t data = I2C_ReadNACK();  // Read data
  I2C_Stop();
  return data;
}

/**
 * Initialize TCS34725 color sensor
 */
void TCS34725_Init() {
  TCS34725_WriteReg(0x00, 0x01);  // Power on
  _delay_ms(3);  // Wait for power-up
  TCS34725_WriteReg(0x00, 0x03);  // Enable RGBC
  TCS34725_WriteReg(0x01, 0xFF);  // Set integration time
  TCS34725_WriteReg(0x0F, 0x01);  // Set gain to 4x
}

/**
 * Read color values from TCS34725
 *
 * @param r - pointer to store red value
 * @param g - pointer to store green value
 * @param b - pointer to store blue value
 * @param c - pointer to store clear value
 */
void TCS34725_ReadColors(uint16_t *r, uint16_t *g, uint16_t *b, uint16_t *c)
{
  I2C_Start();
  I2C_Write((TCS34725_ADDR << 1) | 0);
  I2C_Write(TCS34725_CMD | TCS34725_CMD_AUTO_INC | 0x14);  // Start from
CDATAL
  I2C_Start();  // Repeated start
  I2C_Write((TCS34725_ADDR << 1) | 1);  // Read address

  // Read color values (low byte first, then high byte)
  *c = I2C_ReadACK(); *c |= (I2C_ReadACK() << 8);  // Clear
  *r = I2C_ReadACK(); *r |= (I2C_ReadACK() << 8);  // Red
  *g = I2C_ReadACK(); *g |= (I2C_ReadACK() << 8);  // Green
  *b = I2C_ReadACK(); *b |= (I2C_ReadNACK() << 8); // Blue
```

```
  I2C_Stop();
}

/**
 * Detect and report color using TCS34725 sensor
 */
void colourmode() {
  uint16_t r, g, b, c;
  static bool initialized = false;

  // Initialize color sensor on first call
  if (!initialized) {
    I2C_Init();
    TCS34725_Init();
    initialized = true;
    _delay_ms(500);  // First read delay
  }

  // Read color values
  TCS34725_ReadColors(&r, &g, &b, &c);

  // Determine color based on normalized RGB values
  if (c > 0) {
    float rn = (float)r / c;  // Normalized red
    float gn = (float)g / c;  // Normalized green
    float bn = (float)b / c;  // Normalized blue

    // Simple color detection logic
    if (rn > gn && rn > bn && rn > 0.4) {
      dbprintf("COLOR:RED");
    } else if (gn > rn && gn > bn && gn > 0.4) {
      dbprintf("COLOR:GREEN");
    } else {
      dbprintf("COLOR:NONE");
    }

    // Report raw values for debugging
    dbprintf(" R:%d G:%d B:%d C:%d", r, g, b, c);
  } else {
    dbprintf("COLOR:NO_LIGHT");
  }
}

/*
 * ========== MAIN LOOP ==========
 */
```

```
/**
 * Arduino main loop function - runs repeatedly
 */
void loop() {
  // Process incoming packets from the Pi
  TPacket recvPacket;  // This holds commands from the Pi
  TResult result = readPacket(&recvPacket);

  if (result == PACKET_OK) {
    handlePacket(&recvPacket);  // Triggers handleCommand -> which triggers
all action functions
  } else if (result == PACKET_BAD) {
    sendBadPacket();
  } else if (result == PACKET_CHECKSUM_BAD) {
    sendBadChecksum();
  }

  // Check if forward/backward movement is complete
  if (deltaDist > 0) {
    if (dir == FORWARD) {
      if (forwardDist > newDist) {
        deltaDist = 0;
        newDist = 0;
        stop();
        dir = (TDirection)STOP;
      }
    } else if (dir == BACKWARD) {
      if (reverseDist > newDist) {
        deltaDist = 0;
        newDist = 0;
        stop();
        dir = (TDirection)STOP;
      }
    } else if ((Tdir)dir == STOP) {
      deltaDist = 0;
      newDist = 0;
      stop();
      dir = (TDirection)STOP;
    }
  }

  // Check if rotation (left/right) is complete
  if (deltaTicks > 0) {
    if (dir == LEFT) {
      if (leftReverseTicksTurns >= targetTicks) {
        deltaTicks = 0;
        targetTicks = 0;
```

```
        stop();
        dir = (TDirection)STOP;
      }
    } else if (dir == RIGHT) {
      if (rightReverseTicksTurns >= targetTicks) {
        deltaTicks = 0;
        targetTicks = 0;
        stop();
        dir = (TDirection)STOP;
      }
    } else if ((Tdir)dir == STOP) {
      deltaTicks = 0;
      targetTicks = 0;
      stop();
      dir = (TDirection)STOP;
    }
  }
}
```

Full Robot.ino code

```
#include "AFMotor.h"



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


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

```
void move(float speed, int direction)
{
  int speed_scaled = (speed/100.0) * 255;
  motorFL.setSpeed(speed_scaled);
  motorFR.setSpeed(speed_scaled);
  motorBL.setSpeed(speed_scaled);
  motorBR.setSpeed(speed_scaled);

  switch(direction){
      case GO:
        motorFL.run(FORWARD);
        motorFR.run(FORWARD);
        motorBL.run(FORWARD);
        motorBR.run(FORWARD);
      break;
      case BACK:
        motorFL.run(BACKWARD);
        motorFR.run(BACKWARD);
        motorBL.run(BACKWARD);
        motorBR.run(BACKWARD);
      break;
      case CW:
        motorFL.run(FORWARD);
        motorFR.run(FORWARD);
        motorBL.run(BACKWARD);
        motorBR.run(BACKWARD);
      break;
      case CCW:
        motorFL.run(BACKWARD);
        motorFR.run(BACKWARD);
        motorBL.run(FORWARD);
        motorBR.run(FORWARD);
      break;
      case STOP:
      default:
        motorFL.run(STOP);
        motorFR.run(STOP);
        motorBL.run(STOP);
        motorBR.run(STOP);
    }
}
```

```
void forward(float dist, float speed)
{
  if(dist > 0){
    deltaDist = dist; //
  }
  else{
    stop();
    dbprintf("deltadist is 0_f");
  }

  newDist=forwardDist + deltaDist;
  dir=(TDirection) FORWARD;
  move(speed, FORWARD);
}

void backward(float dist, float speed){
  if(dist > 0){
    deltaDist = dist;
  }
  else{
    stop();
    dbprintf("deltadist is 0_b");
  }

  newDist=reverseDist + deltaDist;
  dir=(TDirection) BACKWARD;
  move(speed, BACKWARD);
}

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

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

Full

```
void servo_angle(const int left_angle, const int right_angle) {
//20,160
  if(left_angle>180 || left_angle<0 || right_angle>180
||right_angle<0){
    return; // do it again
  }
  // Convert angle (0-180) to PWM value (1000-2000 for ~1ms-2ms)
   OCR5A = 1500 + (left_angle * 1000 / 180);
   OCR5B = 1500 + (right_angle * 1000 / 180);
}

void stop()
{
  dir=(TDirection) STOP;
  move(0, STOP);
  newDist=0;
  deltaDist=0;
  deltaTicks=0;
  targetTicks=0;
}
```

Constants.h code

```
#ifndef __CONSTANTS_INC__
#define __CONSTANTS_INC__

/*
 *  This file contains all the packet types, commands
 *  and status constants
 *
 */

// Packet types
typedef enum
{
  PACKET_TYPE_COMMAND = 0,
  PACKET_TYPE_RESPONSE = 1,
  PACKET_TYPE_ERROR = 2,
  PACKET_TYPE_MESSAGE = 3,
```

```
  PACKET_TYPE_HELLO = 4
} TPacketType;

// Response types. This goes into the command field
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;


// Commands
// For direction commands, param[0] = distance in cm to move
// param[1] = speed
typedef enum
{
      COMMAND_FORWARD = 0,  //w
      COMMAND_REVERSE = 1,  //s
      COMMAND_TURN_LEFT = 2,      //a
      COMMAND_TURN_RIGHT = 3,     //d
      COMMAND_GEAR_1 = 4,   //f
      COMMAND_GEAR_2 = 5,   //g
      COMMAND_GEAR_3 = 6,   //h
      COMMAND_ULTRA = 7,    //z
      COMMAND_COLOUR = 8,   //c
      COMMAND_CAM = 9, //x
      COMMAND_STOP = 10,    //y
      COMMAND_SERVO_OPEN = 11,    //q
      COMMAND_SERVO_CLOSE = 12,   //e
      COMMAND_CLEAR_STATS = 13,   //t
      COMMAND_GET_STATS = 14,     //r
      COMMAND_TURN_AND_OPEN_TRAP = 15, //o
      COMMAND_SHAKE = 16 //p
} TCommandType;

typedef enum{
  FORWARD=1,
```

```
  BACKWARD=2,
  LEFT=3,
  RIGHT=4
}TDirection;
// Direction values

typedef enum Tdir
{
  STOP, //0
  GO, //1
  BACK,
  CCW,
  CW
} Tdir;
#endif
```