Table of Contents

1. Introduction 3
   1.1. Introduction 3
   1.2. Project Objective 3
2. Review of State of the Art 3
   2.1 Boston Dynamics' Spot 3
   2.2 ANYbotics' ANYmal 4
3. System Architecture 4
4. Hardware Design 5
5. Firmware Design 7
   5.1 High Level Algorithm on the Arduino Mega 7
   5.2 Communication Protocol 8
   5.3 Noteworthy Firmware Implementations 9
   5.4 Neat Organisation of Arduino Code 9
6. Software Design 10
   6.1 High Level Algorithm on the Pi 10
      6.1.1 Arm Control 10
      6.1.2 Color Sensor 11
   6.2 Notable Software Implementations 11
      6.2.1 Raw Drive Mode for Responsive Teleoperation 11
      6.2.2 Automated Color Polling with Color Classification 12
      6.2.3.1 LIDAR Data Sampling and Forwarding (Pi) 12
      6.2.3.2 SLAM and Map Visualisation Setup (Desktop) 13
      6.2.3.3 Receive LIDAR Data and Live Rendering 13
7. Lessons Learnt - Conclusion 13
   7.1 Lessons 13
   7.2 Mistakes 14
References 15
Appendix A: Diagrams and Pictures from 1.1 16
Appendix B: Code snippets 17
Appendix C: Motor, Sensor and Servo Control Command List 28

1. Introduction

1.1. Introduction

Teleoperation extends human capability into hazardous or unreachable environments. NASA's Robonaut 2 [1], the first humanoid in space, performs ISS maintenance via teleoperation from crew or ground controllers. Shark Robotics' Colossus [2], deployed at the 2019 Notre-Dame fire, streams thermal imagery and sensor data to pilots up to 300m away. Such systems share robust mobility, multi-modal sensing, and low-latency links — principles applied here to the lunar search-and-rescue.

1.2. Project Objective

The objective of this project is to design and implement Alex, a teleoperated rescue robot for the Moonbase SG disaster scenario, in which a cascading explosion has injured all astronauts and disrupted their medical supplies. The primary objectives are:
(1) **Enable real-time remote teleoperation** over a LAN connection, with two operators sharing control to reduce task load;
(2) **Implement LiDAR-based SLAM** to navigate the bounded arena without collisions and to map the base for further rescue efforts;
(3) Locate and retrieve medpaks by **identifying valid colored drop zones** with an RGB color sensor, distinguishing them from decoys;
(4) Execute payload delivery using a robotic arm to transport medpaks safely to each injured astronaut;
(5) **Integrate safety and visual verification** via an emergency-stop button and camera stream for situational awareness during the mission.

2. Review of State of the Art

2.1 Boston Dynamics' Spot



Boston Dynamics' Spot is a quadrupedal unmanned ground vehicle (UGV) designed for rapid situational awareness in hazardous or unstructured environments. As an update to the L3Harris T7, it shifts from the focus from heavy, high-mass tracked power to agile, legged navigation that can traverse human-centric spaces like stairs and narrow corridors. The hardware features a 360° camera suite, LiDAR for obstacle avoidance, and a modular robotic arm. It is tele-operated via a tablet interface but supports semi-autonomous pathing. Spot is similar to Alex as both utilize remote control and camera feedback to interact with the environment via a robotic arm.
Strengths: Exceptional mobility on uneven debris and steep slopes that typically high-center tracked robots; capable of generating high-fidelity 3D maps of disaster zones for responders.
Weaknesses: Limited battery life of approximately 90 minutes restricts the duration of deep-entry missions; its 14 kg payload capacity is significantly lower than heavy-duty predecessors, limiting its ability to move large obstructions.

2.2 ANYbotics' ANYmal

ANYbotics' ANYmal is a specialized quadrupedal platform built for autonomous and tele-operated search and rescue in high-risk industrial sites. Compared to the Quince robot, it replaces basic tracked locomotion with AI-driven legged mobility and a far more sophisticated sensor suite for survivor detection. The system integrates a pan-tilt head with thermal imaging, gas sensors, and ultrasonic microphones. Like Alex, ANYmal relies on a remote-control interface and multi-modal sensor feedback to perceive and navigate its surroundings.



Strengths: Its IP67-rated waterproof and dustproof design allows for operation in extreme weather or flooded zones; the advanced sensor suite provides detection of gas leaks or structural whistling that visual cameras might miss.

Weaknesses: The 2–4 hour battery life may pose limitations for extensive multi-shift operations; despite its agility, it lacks the raw lifting and pulling power required for heavy debris clearance or casualty extraction.

3. System Architecture

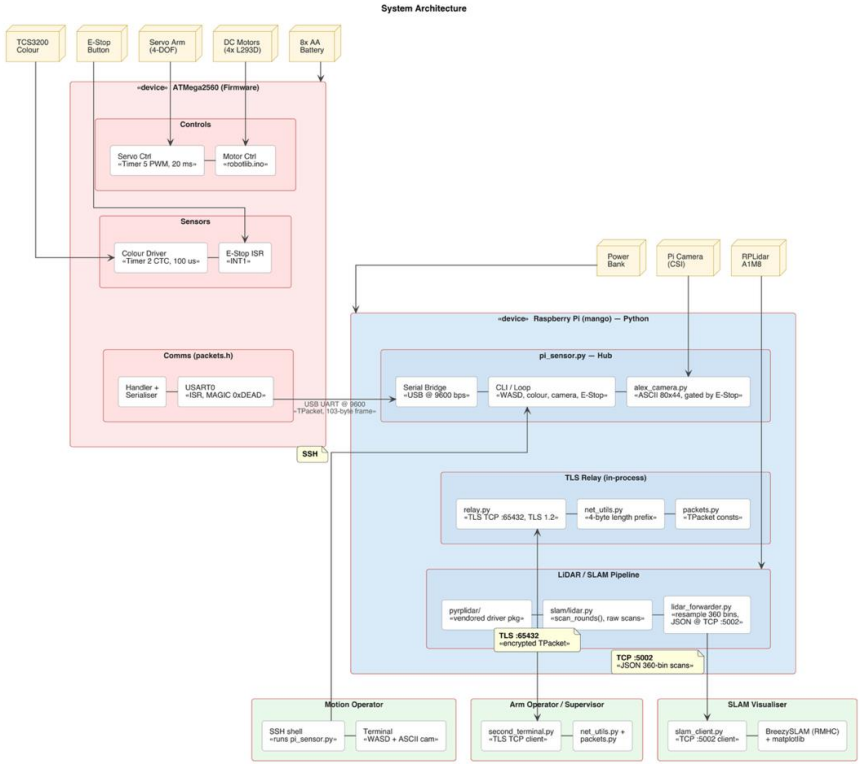


Figure 3-1 System Architecture Diagram

The robot system is built around a two-processor core: an Arduino Mega 2560 handles all real-time hardware control, while a Raspberry Pi acts as the central coordination hub. The Arduino actuates four PMDC motors via an AFMotor Motor Shield, four servo joints for robotic arm manipulation via Timer 5, reads the TCS3200 color sensor for correct medpak zone identification using Timer 2 CTC, and responds to the physical E-Stop button through INT1. All communication between the two processors flows over serial UART using a shared 100-byte

TPacket struct wrapped in a 103-byte frame. Error handling protocols are in place across all communication layers to ensure robust operation.

The Raspberry Pi offloads computationally heavy tasks to thin clients over the network rather than processing them locally. LIDAR scan data from the RPLidar A1M8 passes through a driver stack; *lidar_forwarder.py* resamples the raw scans into 360 equal-angle bins, before streaming the processed data as JSON over a TCP server on port 5002. A dedicated SLAM thin client receives this stream, runs the BreezySLAM algorithm locally, and renders a continuous occupancy map using matplotlib. The Pi Camera captures greyscale frames using the picamera2 library, which are rendered directly in the primary operator's SSH terminal via the *alex_camera* module, avoiding any separate video streaming infrastructure.

Remote teleoperation is split across two operator terminals. The primary operator controls robot movement by typing WASD keys over an SSH connection directly into the Pi. A secondary Robot Arm Operator connects to a second terminal session (second_terminal.py) running over the same LAN to send arm joint commands, which are transmitted securely to the Pi using TLS encryption — ensuring commands cannot be intercepted over the network. On the Pi's end, an encrypted server receives these commands and writes the already-framed TPacket directly to the Arduino over serial; any replies from the Arduino are translated back and sent to the operator's screen.

4. Hardware Design

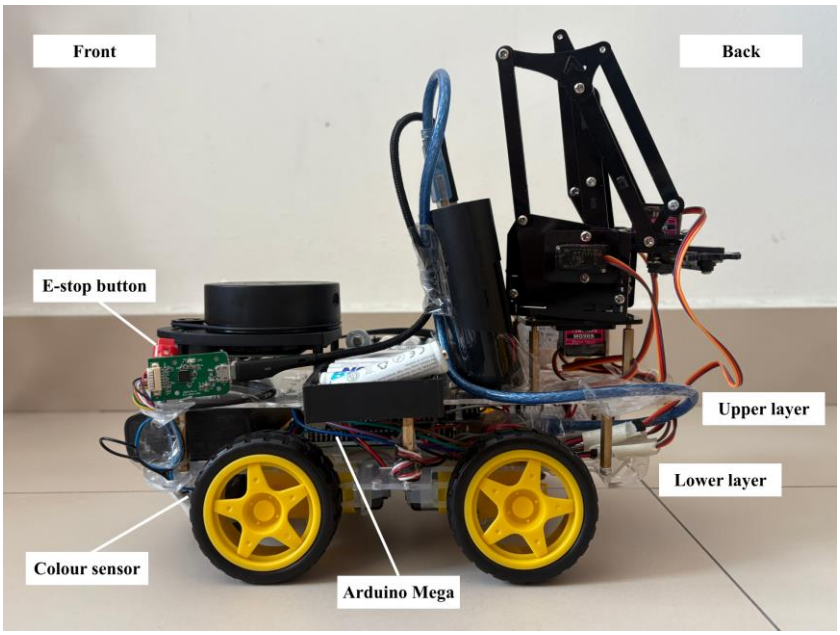


Figure 4-1 Overall Structure of the Hardware Design (Side View)

The robot is designed with a two-layer structure to optimise space and functionality. The bottom layer contains the Arduino Mega, while the top layer supports the LIDAR module, robot arm, and battery packs.



Figure 4-2 Colour Sensor Attached to the Bottom Layer (Front View)

A colour sensor is mounted on the underside of the base layer at the front of the robot; to allow colour detection the moment the robot drives over a coloured zone. It is enclosed by black paper to minimise interference from ambient light and improve sensing accuracy.

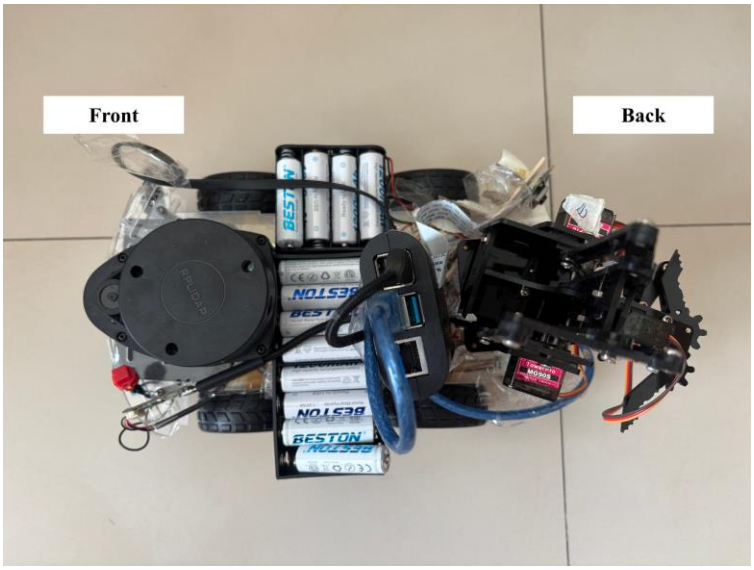


Figure 4-3 Top-Down View of the Hardware Design

For navigation and obstacle detection, the LIDAR module is positioned at the front of the robot, enabling effective mapping of the surrounding environment and timely collision avoidance. The battery packs are arranged flat on the upper layer to maintain a low height, ensuring they do not obstruct the LIDAR's field of view, as seen in Figure 4-1.

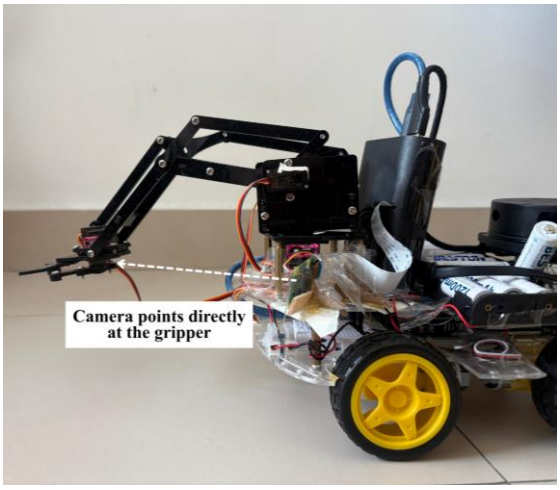


Figure 4-4 Side View of the Robot Arm and Camera



Figure 4-5 Maximum Opening of the Gripper due to constrains in our Code

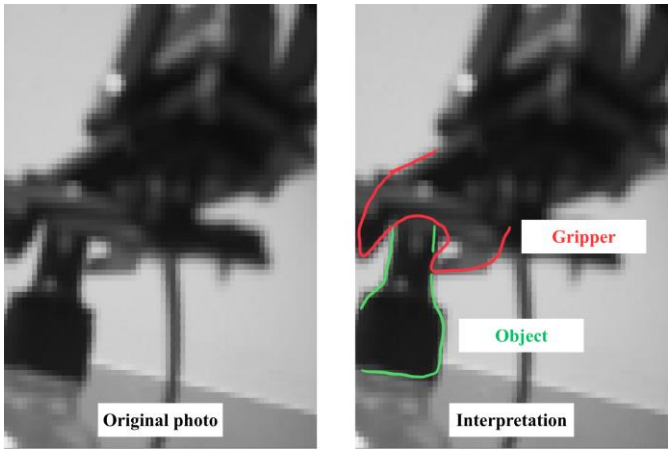


Figure 4-6 Camera View and Interpretation

The robot arm is mounted at the rear of the robot, with the Raspberry Pi camera oriented towards the gripper. This configuration allows for image capturing during the gripping process of the medpak. Each servo motor was then constrained within predefined limits to prevent overextension, ensuring its movements remain within safe bounds and reducing the risk of burnout. To prevent interference of wires with both the LID AR and the robot arm, the wires were trimmed short or secured onto the chassis board.

5. Firmware Design

5.1 High Level Algorithm on the Arduino Mega

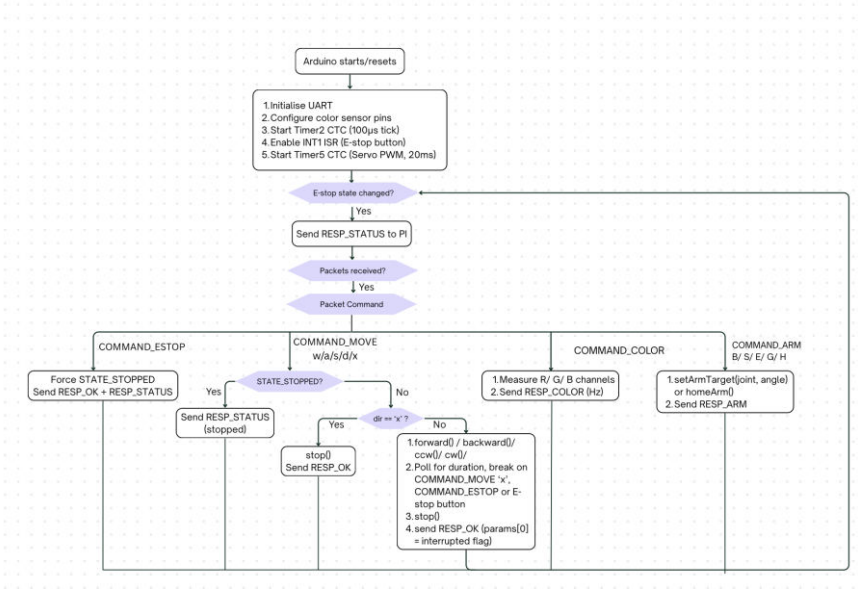


Figure 5-1 Flow Chart for Arduino Firmware

The Arduino Mega serves as the low-level real-time controller in this project. On reset (triggered whenever the Raspberry Pi opens the USB serial port), it initialises its peripherals - UART via the bare-metal USART0 driver, the TCS3200 color-sensor pins, Timer 2 for the 100 µs color-sensing tick, the INT1 E-Stop interrupt, and Timer 5 for the 20ms servo PWM - before entering its main loop. Its primary task is to drive the actuators for traversing the map and performing the rescue operation, which is executed upon receiving command packets from the Pi (MOVE, ARM, COLOR, ESTOP). In addition, it sends asynchronous RESP_STATUS updates back to the Pi whenever the E-Stop state changes, and RESP_COLOR / RESP_ARM frames back in response to sensor and arm commands, for decision-making on the Pi side.

5.2 Communication Protocol

The communication protocol between the Arduino Mega and the Raspberry Pi is serial UART with a baud rate of 9600 and standardised at the 8N1 frame format. The UART on the Arduino is driven directly through the ATMega2560 USART0 registers, with transmit and receive buffers serviced by interrupt handlers so the main loop is never blocked on I/O. The main data format used for communication is the *TPacket*, defined as a struct with a total of 100 bytes. It contains: *packetType*, a byte indicating whether the frame is a command, response or message; command, a byte specifying the specific command or response type based on pre-defined constants; data, a 32-byte string buffer for additional information such as the direction character of a move command; and params, an array of sixteen 32-bit integers for passing command-specific parameters such as duration, joint angle or color readings [Code in Appendix B Code Snippet 1A]

Every *TPacket* on the wire is framed with a two-byte magic number 0xDEAD, followed by the 100-byte struct, followed by a one-byte XOR checksum of the struct -103 bytes per frame. On receive, the Arduino verifies the magic to re-synchronise the byte stream and recomputes the checksum; frames with an invalid magic or checksum mismatch are discarded silently, and incomplete frames simply wait for more data. On the Pi side, the same *TPacket* is re-wrapped with a four-byte big-endian length prefix before being forwarded over TLS 1.2

on TCP port 65432 to the second terminal, so the second terminal de-frames cleanly on a reliable stream without scanning for magic bytes. [Code in Appendix B Code Snippet 1b]

Four commands are defined (ESTOP, COLOR, MOVE, ARM) and four responses (OK, STATUS, COLOR, ARM). RESP_STATUS carries the current RUNNING or STOPPED state in params[0] and is sent automatically whenever the E-Stop state changes, so the Pi never has to poll for it.

5.3 Noteworthy Firmware Implementations

Two firmware features make teleoperation more forgiving. Interruptible moves let any in-progress WASD command be aborted immediately by a stop, software E-Stop, or physical button press, instead of forcing the robot to finish its full timed motion. Lerp'd servo motion turns an operator-issued arm target into a smooth ramp rather than a hard jump, which prevents wear and tear on the servos and makes the arm easier to track visually.

5.3.1 Non-Blocking Motor Stop

Rather than blocking for the full move duration, *handleCommand()* runs a non-blocking *millis()* loop that drains incoming packets on each iteration. An 'x' stop command, a COMMAND_ESTOP, or the physical button flipping the state out of STATE_RUNNING all break the loop, halt the motors, and reply with a flag indicating whether the move was interrupted or completed. The 'x' command is deliberately distinct from an E-Stop: it only cuts motor power and does not change the state, so the color sensor and other peripherals continue to operate normally. A mistaken forward command therefore no longer commits the robot to two full seconds of motion - the operator can cancel within milliseconds and keep using the rest of the system. [Code in Appendix B Code Snippet 2]

5.3.2 Lerp'd Arm Movement

The arm driver separates the commanded target angle from the currently applied one. When the operator issues a new arm command, *setArmTarget()* only updates the target. On every 20ms servo cycle, the Timer 5 COMPA interrupt calls *lerpTicks(),* which steps the current angle toward the target by a small, fixed increment - roughly two degrees per cycle - and the pulse generator emits its pulse using that interpolated value. Typing a large angle change therefore produces a smooth sweep instead of a violent snap, preventing wear and tear on the servos and stopping the whole robot from jerking due to spontaneous arm movements. [Code in Appendix B Code Snippet 3]

5.4 Neat Organisation of Arduino Code

In large embedded systems projects, proper code factoring and a clear file structure are critical for maintainability and for allowing multiple developers to work on overlapping regions without conflict. In this project, the Arduino sketch is split across several *.ino* and header files, each scoped to a single responsibility. The main sketch retains only the shared constants, the cross-subsystem volatile state, the command dispatcher and the *setup / loop* entry points, while the arm, color sensor, estop and motor drivers have their own .ino files, and the USART0 transport and *TPacket* wire format live in *serial_driver.h* and *packets.h* respectively. This separation has proven useful throughout the project: timer allocation is easier to reason about, subsystems can be debugged without scrolling past the others, and protocol

changes, transport experiments and application-logic edits can progress independently, simplifying version control and collaboration. [Image in Appendix B Code Snippet 4]

6. Software Design

6.1 High Level Algorithm on the Pi

6.1.1 Arm Control

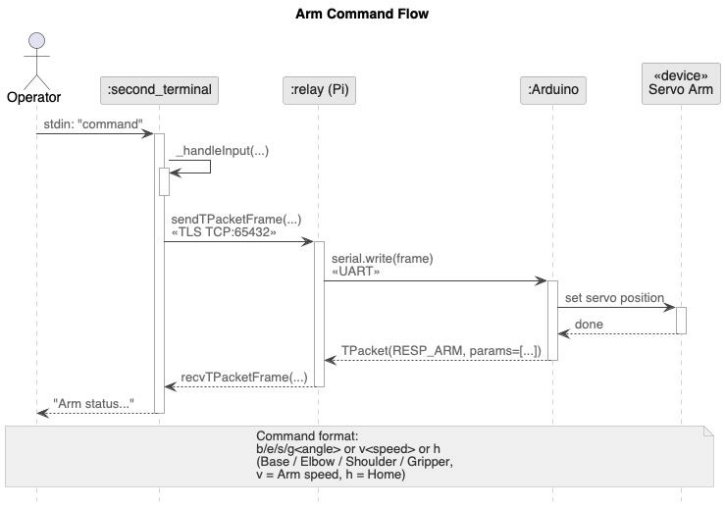


Figure 6-1 High level diagram of Arm command flow

The robot arm PWM generation is implemented using Timer 5, which splits a 20ms PWM period into four segments to respectively control the four servos (base, shoulder, elbow and gripper). The COMPB interrupt acts like a scheduler, where each stage either turns a servo pin HIGH (start of pulse) or LOW (end of pulse). The timeline is divided using fixed checkpoints for each servo and within each segment, the pulse width determines how long the signal stays HIGH before being turned off. This time-division multiplexing of PWM allows all the servos to share a single timer while still receiving independent pulse widths within the same 20ms frame, giving the effect of simultaneous motion. The motion is kept smooth by gradually moving each servo's current position toward its target instead of jumping instantly. [Code in Appendix B Code Snippet 5]

On the software side, arm commands are entered on *second_terminal.py* as single-character prefixes followed by a numeric angle (e.g. b90 for Base → 90°, s140 for Shoulder, e100 for Elbow, g10 for Gripper, or h to return to home). *_handleInput()* in *second_terminal.py* parses the input, packs it into a COMMAND_ARM *TPacket* via *_packFrame(),* and sends it over the TLS-encrypted relay on TCP port 65432 to pi_sensor.py on the Pi. The Pi's relay (*relay.py*) receives the framed packet and writes it directly to the Arduino over USB serial. The Arduino's arm driver (described in Section 5.3.2) then interpolates the servo position smoothly toward the commanded target. Arm status responses (RESP_ARM) are returned from the Arduino, forwarded back through the relay, and displayed on *second_terminal.py* showing the target angle of each joint.

6.1.2 Color Sensor

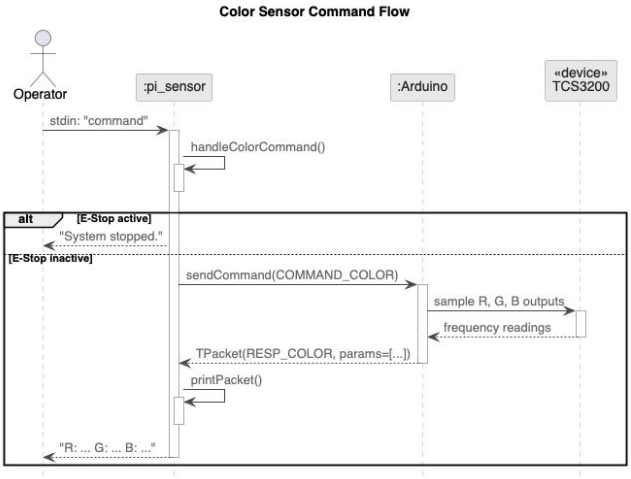


Figure 6-2 High level diagram of Color sensor command flow

The color sensor code outputs a square wave whose frequency corresponds to the intensity of the selected color channel. To measure this frequency, we use Timer 2 in CTC mode and maintain a global tick counter, which increments at a fixed interval via an interrupt. The function *measureChannel()* sets the desired filter, records the starting tick, and then loops until a fixed number of ticks have elapsed. During this window, it repeatedly checks the sensor output pin and counts rising edges by incrementing a counter whenever the signal goes HIGH, then waiting until it goes LOW again to avoid double counting. After the measurement window, the total edge count is scaled (x10) to convert from counts per 100ms to frequency in Hz. The *readColorChannels()* function calls this measurement three times (for R, G, B), storing the frequencies in separate variables. When a command color is received, these values are packed into a response packet and sent to the Raspberry Pi. [Code in Appendix B Code Snippet 6]

6.2 Notable Software Implementations

6.2.1 Raw Drive Mode for Responsive Teleoperation

The default teleoperation interface accepts line-based commands of the form w <ms>, where the operator types a direction key followed by a duration in milliseconds and presses enter. While explicit, this is awkward for fine manoeuvring in tight obstacle gaps or during med-pak-zone approaches. To address this, a raw drive mode is implemented in *runRawDriveMode()* in *pi_sensor.py*, which is entered by default on startup. The line-based mode is reached by pressing m to toggle out, and m again to return. When active, the terminal is reconfigured using *tty.setcbreak* together with *termios* so that each keystroke is read immediately without waiting for a carriage return. A press of w, a, s, or d is dispatched through *handleMovementCommand()* to send a COMMAND_MOVE *TPacket* with the direction character in the data field and a fixed short pulse duration in the params field, so the robot moves in short pulses for as long as the key is held or tapped. The loop also supports keys for E-stop, immediate stop, one-time color reading, camera captures, auto-color toggling, and incremental speed adjustments. Before every motion pulse, *handleMovementCommand()* calls *isEstopActive()* and refuses to drive if the safety state is latched, so the operator retains the same safety interlocks as in line-based mode. The result is a game-like feel that removes the need to estimate durations mid-mission and noticeably reduces cognitive load during fine positioning. [Code in Appendix B Code Snippet 7]

6.2.2 Automated Color Polling with Color Classification

The color sensor is the primary means of distinguishing valid medpak zones from the floor and from the decoy non-medpak zone, but requiring the operator to manually trigger COMMAND_COLOR while also driving the robot is impractical during a timed mission. To address this, *toggleAutoColor()* in *pi_sensor.py* is bound to a single keypress and flips an internal flag that causes the main loop to emit a color request at a fixed interval without any operator input. The sensor returns a frequency from each of its red, green, and blue photodiode filters -higher frequency indicating more reflected light at that wavelength -and these three values are forwarded as RESP_COLOR packets through the TLS relay to the second terminal. To keep the main terminal uncluttered while the Pi-side operator issues drive commands, color readings are printed on the second terminal instead.

Besides printing the raw frequency values, *_printPacket()* in *second_terminal.py* runs the values through a classifier whose thresholds were tuned empirically against the actual arena floor and color papers at the maze site, mapping the raw readings into one of four semantic labels: RED, GREEN, BLUE, or FLOOR. Both the raw frequency triplet and the label are printed, so the operator can confirm a medpak zone at a glance while retaining the numerical reading for diagnostics. Color printing can itself be toggled off via a dedicated key, which is particularly useful when the second-terminal operator is controlling the arm -since arm commands are also entered as keystrokes on the same terminal, a constant stream of incoming color prints would otherwise collide with the operator's typed input and disrupt operator inputs. [Code and terminal output in Appendix B Code Snippet 8]

6.2.3.1 LIDAR Data Sampling and Forwarding (Pi)

To offload the SLAM mapping to a more powerful client, *lidar_forwarder.py* forwards the LIDAR data to a connected client over a TCP network. The forwarder talks directly to the Lidar through a thin hardware wrapper (*lidar.py*) built on top of the PyRPLidar library, whose *scan_rounds()* generator yields one complete 360° rotation at a time. Low-quality readings are filtered out along the way: the first few rotations are discarded while the motor spins up, zero-distance measurements are dropped in *_resample()*, and samples beyond the configured maximum range are clipped so that spurious long-range returns do not destabilise SLAM.

In *lidar_forwarder.py*, the *serve()* function creates a TCP server socket that listens on port 5002. This port is chosen so that it does not collide with the TLS-encrypted second-terminal relay on port 65432, allowing the operator-control and SLAM data channels to run simultaneously from the same Pi. When a client connects, the forwarder enters a loop that continuously sends LIDAR data. For each rotation, *_resample()* bins the raw measurements into 360 equal-angle slots, converts the Lidar's native clockwise angle convention into BreezySLAM's counter-clockwise convention, and applies a configurable mounting offset so that 0° always corresponds to robot-forward. The resampled data is serialised into newline-delimited JSON and sent to the client with *conn.sendall()*. If the client disconnects, the forwarder drops the connection and returns to accepting new ones, without restarting the Pi or the LIDAR motor.

On the desktop client, *slam_client.py* uses a TCP socket to receive the LIDAR data, processes the scan using BreezySLAM, and displays the SLAM map and LIDAR point cloud dynamically to visualise the robot's position, heading, and environment in real time. [Code in Appendix B Code Snippet 9]

6.2.3.2 SLAM and Map Visualisation Setup (Desktop)

The SLAM algorithm and visualisation are initialised first. A Laser object represents the sensor, and an RMHC_SLAM object is instantiated with that laser, the map dimensions (1,200 pixels covering 7m of real space), and a map-quality setting. 7m was chosen because the map plots from the origin; since the robot can start from anywhere within the 3m × 3m maze, the map must be at least 6m on a side, and 7m gives a safety margin. Two subplots are created side by side: ax_map displays the SLAM map, robot position, and heading, while ax_pts displays the LIDAR point cloud in robot-relative coordinates. The robot's footprint is plotted as a blue rectangle representing its physical boundaries, which allows the operator to judge clearance in the maze, and the heading is visualised as a green arrow whose tip rotates to mirror the robot's physical heading. A "Full Map / Both Maps" toggle lets the operator expand the occupancy grid across the full figure when finer detail is needed. [Code and image in Appendix B Code Snippet 10]

6.2.3.3 Receive LIDAR Data and Live Rendering

A TCP socket then connects to the Pi on port 5002. Data is received in JSON format, each message containing 360-entry angle and distance arrays; because TCP is a byte stream, incoming bytes are appended to a buffer and split on the newline delimiter so that partial or concatenated messages are parsed correctly. The SLAM algorithm is updated with the latest scan, and if fewer than MIN_VALID_POINTS rays are valid -typically during rapid rotation or when facing a wide-open space -the previous known-good scan is reused so that the particle filter never receives a near-empty input. The robot's position (x, y, θ) is retrieved via *slam.getpos(),* and the scan is converted into Cartesian coordinates in *_scan_to_xy* so that the raw point cloud can be drawn alongside the accumulated occupancy grid. The loop repeats until the connection is closed or the user exits. Malformed JSON lines are skipped silently and socket errors drop the connection cleanly, so a transient network glitch or single bad frame does not bring down the pipeline. [Code in Appendix B Code Snippet 11]

7. Lessons Learnt - Conclusion

7.1 Lessons

1. Independent component testing is far easier than system integration; we had many instances where parts worked perfectly in isolation (e.g. our arm servos) but stopped working once combined with the rest of the code. This made us realise how important it is to design with the full system in mind - especially shared resources like timers, interrupts, and communication channels - rather than trying to hand-optimise each component or code file independently. Thinking in terms of interactions helps make the system far more cohesive and reliable. We saw this first-hand when the servo arm,

which was running on Timer 1, collided with the AFMotor library's own use of Timers 1, 3 and 4, causing both the motors and servos to stop responding. Moving the servo driver to Timer 5 resolved the conflict.

2. Another key lesson was the importance of having a clear strategy alongside technical execution. While our robot hardware and coding functioned reliably, we underestimated the need to plan an efficient approach to complete the objectives. Without a clear vision of the path the robot should take, we moved it through the maze inefficiently, resulting in unnecessary time loss. In addition, we relied too heavily on individual components rather than integrating multiple sensors. For instance, we depended solely on the colour sensor to detect the medpak zone, which required the robot to physically pass over the area before obtaining a reading. Similarly, we relied only on the LIDAR map to identify the astronaut. A more effective approach would have been to combine sensors, for example, using the camera to verify points detected on the LIDAR map, allowing for faster and more reliable identification. This experience underscored the value of both strategic planning and sensor fusion in improving overall system performance

7.2 Mistakes

1. Our hardware design could have been improved. We placed the robot arm and camera at the back, and the LIDAR and color sensor at the front. Since our main bottleneck during the trial run was timing rather than functionality, this layout forced us into inefficient movement patterns (e.g. scanning color first, then reversing), which wasted valuable time. In addition, the LIDAR's placement restricted its field of view as the robot arm was blocking it, leading to poor retention of the LIDAR map. A more intentional layout by separating the layers cleanly would have made the system far more efficient, such as the LIDAR having its own layer such that no other hardware component will block its field of view.
2. Our camera mounting, which was constructed using tape and a popsicle stick was notably flimsy, lacking stability and consistency. As the robot moved, even slight vibrations would shift the camera's position or angle, causing the camera's image to be no longer aligned as expected. This meant that identical scenes could appear differently across runs, introducing unnecessary variability into what should have been a clear and controlled input. In addition, the camera's field of view was also constrained by its close proximity to the robot arm. This limited line of sight meant that we failed to capture the medpak on camera despite being right next to it, making the identification and retrieval of it rather inefficient. Furthermore, the setup lacked any real depth perception. While this was acceptable within our project scope - since we could assume a fixed height for the medpak - it is not a robust approach in practice. Any small change in camera angle, object height, or surface level would affect the accuracy of detection.

References

1.Nasa. (2025). *Robonaut2* https://www.nasa.gov/robonaut2/

2.Shark-Robotics. (2025). *Notre-Dame Fire: How Technology helped preserve a global icon.* https://www.shark-robotics.com/notre-dame-fire-how-technology-helped-preserve-a-global-icon/

3. Boston Dynamics. (2026). *Spot Specifications*. https://support.bostondynamics.com/s/article/Spot-Specifications-49916

4. Boston Dynamics. (2026). *Spot Core I/O Setup and Usage*. https://support.bostondynamics.com/s/article/Spot-Core-IO-Setup-and-Usage-73140

5. ANYbotics. (2026). *ANYmal Technical Specifications: Research and Industrial Inspection.* https://www.anybotics.com/anymal-technical-specifications.pdf

6. ETH Zurich Robotic Systems Lab. (2025). *ANYmal: A Quadrupedal Robot for Autonomous Operation in Challenging Environments*. https://rsl.ethz.ch/robots-media/anymal.html

Appendix A: Diagrams and Pictures from 1.1

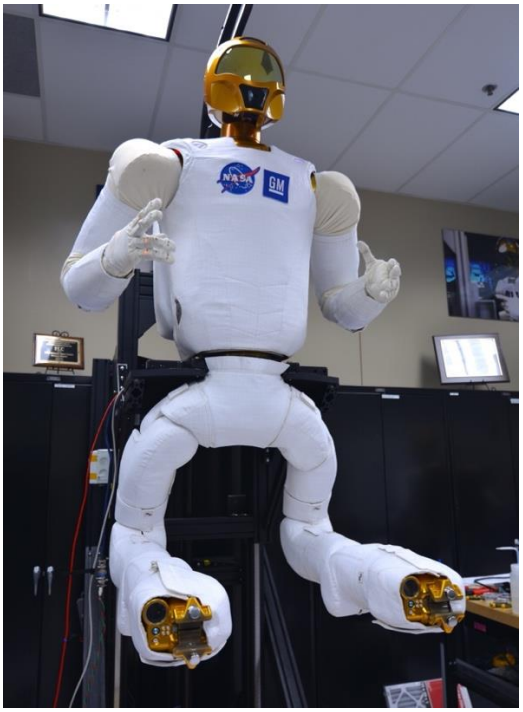


Figure 1-1 NASA Robonaut 2



Figure 1-2 Shark-Robotics Colossus Fire-fighting Robot

Appendix B: Code snippets

Note: All non-essential debug messages/ print statements and comments are omitted

Code Snippet 1 : TPacket definition, Arduino framing and checksum computation

Snippet 1a:

```
typedef struct {
  uint8_t packetType;
  uint8_t command;
  uint8_t dummy[2];
  char data[32];
  uint32_t params[16];
} TPacket;
```

Snippet 1b:

```
static uint8_t computeChecksum(const uint8_t *data, uint8_t len) {
  uint8_t cs = 0;
  for (uint8_t i = 0; i < len; i++)
    cs ^= data[i];
  return cs;
}

static void sendFrame(const TPacket *pkt) {
  uint8_t frame[FRAME_SIZE];
  frame[0] = MAGIC_HI;
  frame[1] = MAGIC_LO;
  memcpy(&frame[2], pkt, TPACKET_SIZE);
  frame[2 + TPACKET_SIZE] = computeChecksum((const uint8_t *)pkt, TPACKET_SIZE);
#if USE_BAREMETAL_SERIAL
  while (!txEnqueue(frame, FRAME_SIZE))
    ;
#else
  Serial.write(frame, FRAME_SIZE);
#endif
}

// Returns true only when *pkt holds a checksum-valid frame.
static bool receiveFrame(TPacket *pkt) {
#if USE_BAREMETAL_SERIAL
  while (((rx_head - rx_tail) & RX_BUFFER_MASK) >= FRAME_SIZE) {
    uint8_t hi = rx_buf[rx_tail];
    uint8_t lo = rx_buf[(rx_tail + 1) & RX_BUFFER_MASK];

    if (hi == MAGIC_HI && lo == MAGIC_LO) {
      uint8_t frame[FRAME_SIZE];
      for (uint8_t i = 0; i < FRAME_SIZE; i++)
        frame[i] = rx_buf[(rx_tail + i) & RX_BUFFER_MASK];

      uint8_t expected = computeChecksum(&frame[2], TPACKET_SIZE);
      if (frame[FRAME_SIZE - 1] == expected) {
        memcpy(pkt, &frame[2], TPACKET_SIZE);
        rx_tail = (rx_tail + FRAME_SIZE) & RX_BUFFER_MASK;
        return true;
```

```
    }
  }

  rx_tail = (rx_tail + 1) & RX_BUFFER_MASK;

}
 return false;
```

Code Snippet 2: Non-blocking motor stop

```
case COMMAND_MOVE: {
        if (buttonState != STATE_RUNNING) {
          sendStatus(STATE_STOPPED);
          break;
        }

        uint8_t speed = (uint8_t)cmd->params[0];
        uint32_t duration = cmd->params[1];
        char dir = cmd->data[0];

        switch (dir) {
          case 'w': forward(speed); break;
          case 's': backward(speed); break;
          case 'a': ccw(speed); break;
          case 'd': cw(speed); break;
          case 'x':
          default:
            stop();sendResponse(RESP_OK, 0);
            break;
        }

        if (dir == 'w' || dir == 's' || dir == 'a' || dir == 'd') {
          unsigned long start = millis();
          bool interrupted = false;
          while (millis() - start < duration) {
            TPacket incoming;
            if (receiveFrame(&incoming)) {
              if (incoming.packetType == PACKET_TYPE_COMMAND &&
                  incoming.command == COMMAND_MOVE && incoming.data[0] == 'x') {
                interrupted = true;
                break;
              }
              if (incoming.packetType == PACKET_TYPE_COMMAND &&
                  incoming.command == COMMAND_ESTOP) {
                interrupted = true;
                break;
              }
            }
            if (buttonState != STATE_RUNNING) {
              interrupted = true;
              break;
            }
          }
          stop();
```

```
        sendResponse(RESP_OK, interrupted ? 1 : 0);
      }
      break;
    }
```

Code Snippet 3: Lerp'd arm movement

```
volatile int TICKS_PER_PERIOD = 50;// lerp speed: ticks per 20ms period per servo

volatile int arm_curr_ticks[4];
volatile int arm_target_ticks[4];

// set target for one servo (degrees) — converts to ticks once here
static void setArmTarget(int idx, int angle) {
  int clamped = constrainAngle(idx, angle);
  arm_target_ticks[idx] = angleToPulse(clamped);
}

static void lerpTicks() {
  for (int i = 0; i < 4; i++) {
    if (arm_curr_ticks[i] < arm_target_ticks[i]) {
      if (arm_curr_ticks[i] + TICKS_PER_PERIOD > arm_target_ticks[i])
        arm_curr_ticks[i] = arm_target_ticks[i];
      else
        arm_curr_ticks[i] += TICKS_PER_PERIOD;
    } else {
      if (arm_curr_ticks[i] - TICKS_PER_PERIOD < arm_target_ticks[i])
        arm_curr_ticks[i] = arm_target_ticks[i];
      else
        arm_curr_ticks[i] -= TICKS_PER_PERIOD;
    }
  }
}
ISR(TIMER5_COMPA_vect) { lerpTicks(); }
```

Code Snippet 4: Arduino code structure

```
├── alex_firmware.ino   # Main sketch: setup()/loop(), command dispatcher (handleCommand), shared volatile state
├── arm.ino             # 4-servo arm driver on Timer 5: setArmTarget, lerpTicks, staggered pulse-generation ISRs
├── color_sensor.ino    # TCS3200 colour-sensor driver on Timer 2: 100 us tick, pulse counting, readColorChannels
├── estop.ino           # E-Stop button handler: INT1 ISR on D20, toggles buttonState between RUNNING and STOPPED
├── packets.h           # TPacket protocol: struct layout, packet/command/response enums, MAGIC and FRAME_SIZE
├── robotlib.ino        # Motor primitives over AFMotor (Timers 1/3/4): forward, backward, cw, ccw, stop
└── serial_driver.h     # USART0 transport: ISR-driven TX/RX ring buffers, sendFrame/receiveFrame, XOR checksum
```

Code Snippet 5: Arm control

```
#define BASE_CHECKPOINT 0      // staggered checkpoint (timer ticks)
#define SHLD_CHECKPOINT 10000  // staggered checkpoint (timer ticks)
#define ELBW_CHECKPOINT 20000  // staggered checkpoint (timer ticks)
#define GRIP_CHECKPOINT 30000  // staggered checkpoint (timer ticks)

volatile int arm_curr_ticks[4];
volatile int arm_stage = 0;
```

```
static void setupArmTimer() {
  DDRC |= 0x17; // PC0-PC2, PC4 as outputs
  PORTC &= ~0x17;
  // initialise current ticks to home position
  for (int i = 0; i < 4; i++) {
    arm_curr_ticks[i] = angleToPulse(constrainAngle(
        i, (int[]){BASE_HOME, SHLD_HOME, ELBW_HOME, GRIP_HOME}[i]));
    arm_target_ticks[i] = arm_curr_ticks[i];
  }
  cli();
  TCCR5A = 0;
  TCCR5B = 0;
  TCNT5 = 0;
  OCR5A = 39999; // 20ms cycle at prescaler 8 / 16MHz
  OCR5B = 0;
  TCCR5B |= (1 << WGM52);
  TCCR5B |= (1 << CS51);
  TIMSK5 |= (1 << OCIE5A) | (1 << OCIE5B);
  sei();
}
ISR(TIMER5_COMPB_vect) {
  switch (arm_stage) {
    case 0: PORTC |=  (1 << BASE_PIN); OCR5B += arm_curr_ticks[BASE_IDX]; break;
    case 1: PORTC &= ~(1 << BASE_PIN); OCR5B  = SHLD_CHECKPOINT;          break;
    case 2: PORTC |=  (1 << SHLD_PIN); OCR5B += arm_curr_ticks[SHLD_IDX]; break;
    case 3: PORTC &= ~(1 << SHLD_PIN); OCR5B  = ELBW_CHECKPOINT;          break;
    case 4: PORTC |=  (1 << ELBW_PIN); OCR5B += arm_curr_ticks[ELBW_IDX]; break;
    case 5: PORTC &= ~(1 << ELBW_PIN); OCR5B  = GRIP_CHECKPOINT;          break;
    case 6: PORTC |=  (1 << GRIP_PIN); OCR5B += arm_curr_ticks[GRIP_IDX]; break;
    case 7: PORTC &= ~(1 << GRIP_PIN); OCR5B  = BASE_CHECKPOINT; arm_stage = -1; break;
  }
  arm_stage++;
}
```

Code Snippet 6: Color sensor

```
// Incremented by Timer 2 COMPA ISR every 100µs
volatile unsigned long _timerTicks = 0;
// Measure one color by counting rising edges on SENSOR_OUT over a 100ms window
static uint32_t measureChannel(uint8_t s2, uint8_t s3) {
  if (s2)
    PORTA |= S2;
  else
    PORTA &= ~S2;
  if (s3)
    PORTA |= S3;
  else
    PORTA &= ~S3;
  uint32_t count = 0;
  unsigned long start = _timerTicks;
  while ((_timerTicks - start) < 1000) {
    if (PINA & SENSOR_OUT) {
      count++;
      while (PINA & SENSOR_OUT);
```

```
    }
  }
  return count;
}
// Multiply by 10 to scale from edges per 100ms into Hz
static void readColorChannels(uint32_t *r, uint32_t *g, uint32_t *b) {
  *r = measureChannel(0, 0) * 10;
  *g = measureChannel(1, 1) * 10;
  *b = measureChannel(0, 1) * 10;
}
// Timer 2 in CTC mode, prescaler = 8 -> 100µs period
static void setupTimer() {
  cli();
  TCCR2A = (1 << WGM21);
  TCCR2B = 0;
  OCR2A = 199;
  TIMSK2 = (1 << OCIE2A);
  TCNT2 = 0;
  sei();
}
static void startTimer() {
  TCCR2B |= (1 << CS21);
}
ISR(TIMER2_COMPA_vect) { _timerTicks++; }
```

Code Snippet 7: Raw drive mode

```
RAW_MOVE_MS = 100    # duration per keypress in raw drive mode
_estop_state = STATE_RUNNING

def isEstopActive():
    return _estop_state == STATE_STOPPED

def handleMovementCommand(direction, duration_ms):
    if isEstopActive():
        return
    sendCommand(COMMAND_MOVE, data=direction.encode(), params=[_motor_speed,
    duration_ms] + [0] * 14)

def runRawDriveMode():
    fd = sys.stdin.fileno()
    old_settings = termios.tcgetattr(fd)
    _last_color_time = 0.0
    try:
        tty.setcbreak(fd) # read each keystroke immediately
        while True:
            _processSerial() # drain incoming TPackets

            now = time.time()
            if _auto_color and COLOR_ENABLED and (now - _last_color_time) >= COLOR_SENSOR_INTERVAL:
                handleColorCommand()
                _last_color_time = now
            # non-blocking 50ms wait for a key
            rlist, _, _ = select.select([sys.stdin], [], [], 0.05)
```

```
        if not rlist:
            continue

        ch = sys.stdin.read(1).lower()

        if ch == 'm': # toggle back to line mode
            print("\n-- NORMAL MODE --")
            return

        if ch in ('w', 'a', 's', 'd'):
            handleMovementCommand(ch, RAW_MOVE_MS)
        elif ch == 'x':
            print("Stopping robot...")
            sendCommand(COMMAND_MOVE, data=b'x', params=[0, 0] + [0] * 14)
        elif ch == 'e':
            print("Sending E-Stop command...")
            sendCommand(COMMAND_ESTOP, data=b'This is a debug message')
        elif ch == 'c':
            handleColorCommand()
        elif ch == 't':
            toggleAutoColor()
        elif ch == 'n':
            togglePrintColor()
        elif ch == 'p':
            handleCameraCommand()
        elif ch == '+':
            handleSpeedChange(SPEED_STEP)
        elif ch == '-':
            handleSpeedChange(-SPEED_STEP)
    finally:
        termios.tcsetattr(fd, termios.TCSADRAIN, old_settings)
```

Code Snippet 8: Automated color polling and classification

Pi side:

```
COLOR_SENSOR_INTERVAL = 2.0 # seconds between auto color requests
_auto_color = False # auto-request color when True
_print_color = False # print color responses (off by default on the Pi; on by default on second terminal)

def toggleAutoColor():
    global _auto_color
    _auto_color = not _auto_color

def togglePrintColor():
    global _print_color
    _print_color = not _print_color

while True:
        _processSerial()

        now = time.time()
        if _auto_color and COLOR_ENABLED and (now - _last_color_time) >= COLOR_SENSOR_INTERVAL:
```

```
        handleColorCommand()
        _last_color_time = now
```

Second_terminal:

```
elif cmd == RESP_COLOR:
    if _print_color:
        r = pkt['params'][0]
        g = pkt['params'][1]
        b = pkt['params'][2]
        print(f"[robot] R: {r} Hz, G: {g} Hz, B: {b} Hz")
        # Thresholds tuned against the maze floor + colored papers on site
        if r > 5000 and g < 3000 and b < 3000:
            label = "RED"
        elif g > 4000 and r < 4000 and b < 4000:
            label = "GREEN"
        elif b > 7000 and r < 4000 and g < 6000:
            label = "BLUE"
        else:
            label = "FLOOR"
        print(f"[robot] Color: {label}")
```

```
[robot] R: 1480 Hz, G: 1430 Hz, B: 1710 Hz
[robot] Colour: FLOOR
[robot] R: 6880 Hz, G: 2780 Hz, B: 2800 Hz
[robot] Colour: RED
[robot] R: 6930 Hz, G: 2780 Hz, B: 2800 Hz
[robot] Colour: RED
[robot] R: 6930 Hz, G: 2790 Hz, B: 2800 Hz
[robot] Colour: RED
[robot] R: 6150 Hz, G: 2800 Hz, B: 2840 Hz
[robot] Colour: RED
[robot] R: 1480 Hz, G: 1430 Hz, B: 1630 Hz
[robot] Colour: FLOOR
```

Sample of second terminal's output when autoColor is enabled.

Code Snippet 9: LIDAR data sampling and forwarding

Pi side:

```
HOST = '0.0.0.0'
PORT = 5002

def _resample(raw_angles, raw_distances):

    bin_sums = [0.0] * SCAN_SIZE
    bin_counts = [0] * SCAN_SIZE

    for angle, dist in zip(raw_angles, raw_distances):
        if dist <= 0: # drops zero-distance samples
            continue
        ccw_angle = (-angle + LIDAR_OFFSET_DEG) % 360
```

```
        idx = int(round(ccw_angle)) % SCAN_SIZE
        bin_sums[idx] += dist
        bin_counts[idx] += 1

    angles = []
    distances = []
    for i in range(SCAN_SIZE):
        angles.append(float(i * 360 / SCAN_SIZE))
        if bin_counts[i] > 0:
            distances.append(min(bin_sums[i] / bin_counts[i], float(MAX_DISTANCE_MM))) # clip long-range
        else:
            distances.append(float(MAX_DISTANCE_MM)) # empty bin -> max-range

    return angles, distances

def serve():
    lidar = lidar_driver.connect()
    if lidar is None:
        return

    scan_mode = lidar_driver.get_scan_mode(lidar)

    server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    server.bind((HOST, PORT))
    server.listen(1)
    server.settimeout(1.0)
    try:
        while True:
            conn = None
            while conn is None:
                try:
                    conn, addr = server.accept()
                except socket.timeout:
                    continue
            round_num = 0
            try:
                for raw_angles, raw_dists in lidar_driver.scan_rounds(lidar, scan_mode):
                    round_num += 1
                    if round_num <= INITIAL_ROUNDS_SKIP: # skip first few rotations
                        continue

                    angles, distances = _resample(raw_angles, raw_dists)
                    payload = json.dumps({'angles': angles, 'distances': distances}) + '\n'
                    try:
                        conn.sendall(payload.encode('utf-8'))
                    except (BrokenPipeError, OSError):
                        print('[forwarder] Client disconnected')
                        break
            except Exception as exc:
                print(f'[forwarder] Scan error: {exc}')
            finally:
                conn.close()
```

```
    except KeyboardInterrupt:
        print('\n[forwarder] Shutting down')
    finally:
        lidar_driver.disconnect(lidar)
        server.close()
```

Code Snippet 10: SLAM and map visualisation

```
SCAN_SIZE = 360
SCAN_RATE_HZ = 5
SCAN_FOV_DEG = 360
MAX_DIST_MM = 4000
MAP_PIXELS = 1200 # pixels per side of the square map
MAP_METERS = 7 # real-world metres covered by the map
MAP_QUALITY = 5
MIN_VALID_POINTS = 100 # below this, reuse previous scan
# SLAM
laser = Laser(SCAN_SIZE, SCAN_RATE_HZ, SCAN_FOV_DEG, MAX_DIST_MM)
slam = RMHC_SLAM(laser, MAP_PIXELS, MAP_METERS, hole_width_mm=HOLE_WIDTH_MM,
      map_quality=MAP_QUALITY,)
mapbytes = bytearray(MAP_PIXELS * MAP_PIXELS)

# Matlibplot: two subplots side by side
plt.ion()
fig, (ax_map, ax_pts) = plt.subplots(1, 2, figsize=(14, 6))

# Left subplot: occupancy-grid-map
map_img = ax_map.imshow(
        np.full((MAP_PIXELS, MAP_PIXELS), 127, dtype=np.uint8),
        cmap='gray', origin='lower', extent=[0, MAP_METERS, 0, MAP_METERS],
        vmin=0, vmax=255,)

# Robot footprint rectangle (corners updated each frame)
robot_patch = Polygon(
        np.zeros((4, 2)),
        closed=True,
        edgecolor='blue', facecolor='blue', alpha=0.35,
        linewidth=1.5, label='robot',)
    ax_map.add_patch(robot_patch)

# Heading of the quiver (updated each frame)
robot_quiver = ax_map.quiver(
        [0], [0], [0], [0.15],
        color='green', scale=1, scale_units='xy', angles='xy',)

# Right subplot (LIDAR point cloud)
half = MAP_METERS / 2.0
    ax_pts.set_xlim(-half, half)
    ax_pts.set_ylim(-half, half)
    ax_pts.set_aspect('equal')
    ax_pts.grid(True)
    pts_plot,  = ax_pts.plot([], [], '.', color='black', markersize=POINT_SIZE)

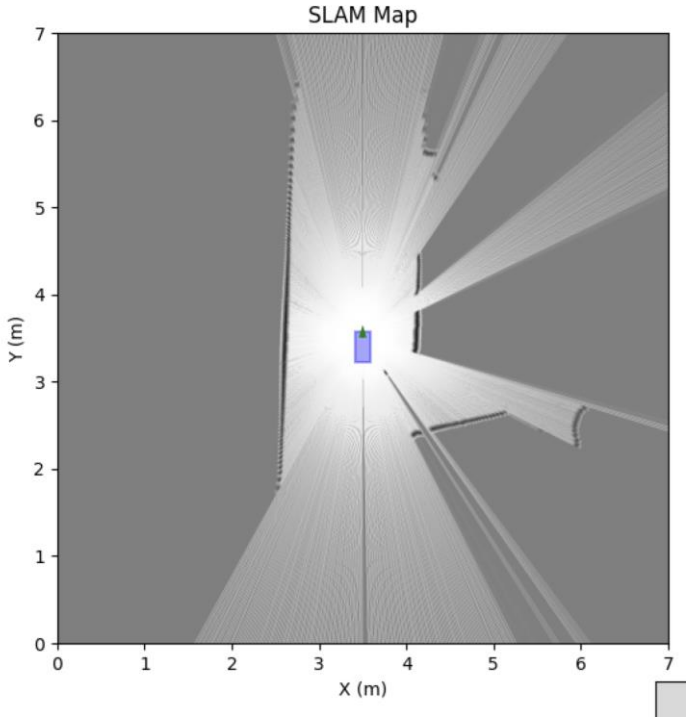
# Full map / Both maps toggle button
orig_pos_map = ax_map.get_position().frozen()
```

```
orig_pos_pts = ax_pts.get_position().frozen()

ax_btn = fig.add_axes([0.43, 0.02, 0.14, 0.06])
btn_toggle = Button(ax_btn, 'Full Map', color='0.85', hovercolor='0.70')

_fullscreen = [False]
def _toggle_fullscreen(event):
    _fullscreen[0] = not _fullscreen[0]
    if _fullscreen[0]:
        ax_pts.set_visible(False)
        ax_map.set_position([
            orig_pos_map.x0,
            orig_pos_map.y0,
            orig_pos_pts.x0 + orig_pos_pts.width - orig_pos_map.x0,
            orig_pos_map.height,
        ])
        btn_toggle.label.set_text('Both Maps')
    else:
        ax_pts.set_visible(True)
        ax_map.set_position(orig_pos_map)
        ax_pts.set_position(orig_pos_pts)
        btn_toggle.label.set_text('Full Map')
    fig.canvas.draw_idle()

btn_toggle.on_clicked(_toggle_fullscreen)
```



Example of the lidar map in a hallway

Code Snippet 11: Receive LIDAR data, and map rendering

```
def _scan_to_xy(distances, angles):
    d = np.asarray(distances, dtype=np.float64)
    a = np.asarray(angles,    dtype=np.float64)
    mask = (d > 0) & (d < MAX_DIST_MM)
    d, a = d[mask], a[mask]
    rad = np.radians((a + 90) % 360)
```

```
    d_m = d / 1000.0
    return d_m * np.cos(rad), d_m * np.sin(rad)



#TCP Connection
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect((host, port))
sock.settimeout(2.0)
buf = '' # byte-steam buffer; JSON objects are \n-delimited

while plt.fignum_exists(fig.number):
        # receive bytes, handle transient errors
        try:
            chunk = sock.recv(65536).decode('utf-8')
            if not chunk:
                break
            buf += chunk
        except socket.timeout:
            plt.pause(0.05)
            continue
        except OSError as exc:
            break
        # split buffer on '\n' to recover complete scans
        while '\n' in buf:
            line, buf = buf.split('\n', 1)
            line = line.strip()
            if not line:
                continue
            try:
                scan = json.loads(line)
            except json.JSONDecodeError:
                continue
            angles    = scan['angles']
            distances = scan['distances']
            # reuse previous scans if too sparse
            d_arr = np.asarray(distances, dtype=np.float64)
            valid = int(np.count_nonzero((d_arr > 0) & (d_arr < MAX_DIST_MM)))
            if valid >= MIN_VALID_POINTS:
                slam.update(distances, scan_angles_degrees=angles)
                prev_distances = list(distances)
            elif prev_distances is not None:
                slam.update(prev_distances, scan_angles_degrees=angles)
```

Appendix C: Motor, Sensor and Servo Control Command List

Note: Raw Drive Mode will be abbreviated as RDM

Command Name	Key	Command Description
Movement controls		

COMMAND_MOVE	W, w	Drive forward for 100ms in RDM Drive forward for 2000ms in line mode
COMMAND_MOVE	A, a	Rotate CCW for 100ms in RDM Rotate CCW for 2000ms in line mode
COMMAND_MOVE	S, s	Drive backward for 100ms in RDM Drive backward for 2000ms in line mode
COMMAND_MOVE	D, d	Rotate CW for 100ms in RDM Rotate CW for 2000ms in line mode
COMMAND_MOVE	X, x	Stops motors
Miscellaneous commands		
COMMAND_ESTOP	E, e	Send E-stop to Arduino
COMMAND_COLOR	C, c	Request for one color-sensor reading
toggleAutoColor	T, t	Acts as a toggle. When enabled, calls COMMAND_COLOR every 2s
togglePrintColor	N, n	Acts as a toggle. When enabled, prints the color-sensor values and associated color to terminal
handleCameraCommand	P, p	Capture one frame using the given camera
handleSpeedChange	+	Increase _motor_speed by 25
handleSpeedChange	-	Decrease _motor_speed by 25
runRawDriveMode	M, m	Allows the operator to toggle between Raw Drive Mode and Line mode.
KeyboardInterrupt	Q, q	Disconnect and exit
Arm commands		
COMMAND_ARM	B, b<angle>	Move the base to specified angle
COMMAND_ARM	S, s<angle>	Move the shoulder to specified angle
COMMAND_ARM	E, e<angle>	Move the elbow to specified angle
COMMAND_ARM	G, g<angle>	Move the gripper to specified angle
COMMAND_ARM	V, v<speed>	Set the arm speed. Speed units: **ticks per 20ms period**
COMMAND_ARM	H, h	Moves the arm to its home position