CG2111A Engineering Principle and Practice II

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B05-6A

Name	Student #	Sub-Team	Role
Thiru Vageesan	A0269930U	Software	Software Lead
Chen Yiyang	A0271796M	Hardware	Procurement & Hardware Lead
Tay Guang Sheng	A0273178W	Software	Software Programmer
Zhong Shu	A0281729U	Software	Software Debugger
Tong Jia Jun	A0271852B	Hardware	Hardware & Report Editor

Table of Contents

Section 1 Introduction

“A man who dares to waste one hour has not discovered the value of life” - Charles Darwin. In natural or manmade disasters, the fight to rescue is a fight against time, more specifically, the fight to beat the 72-hour golden period after any disaster during which humans can survive with physical strength without food and water (Hakami, 2013, 783). Thus, we must maximise search and rescue efforts within the golden period to save as many survivors as possible. As brave and determined as our frontline rescuers are, they are still human and are limited by the human body's flaws, which are susceptible to injuries and exhaustion. This massively hinders the search and rescue effort as our rescuers also need to rest and recuperate during the golden period. As such, we have decided to build a robotic vehicle, called Alex, with search and rescue functionalities to help alleviate some of our rescuers' workload.

Being a robot, Alex is not hindered by the flaws of the human body and can keep moving so long as someone is controlling him. Alex will be able to look for survivors more efficiently than our rescuers which allows us to eliminate any downtime due to fatigue by powering Alex with batteries that will last longer than 72 hours. To replace rescuers, Alex would also need to be able to move around, using a four-wheel drive, and have a set of “eyes”, the RPLidar, to detect obstacles and survivors around him. This allows the operators to control Alex remotely from a safe place far away from the area of disaster. Our Alex is also equipped with sensors to detect the presence of a survivor and speakers to alert on-site rescuers when a survivor has been detected as well as reassure the survivor that help is on the way.

Section 2 Review of State of the Art

This section shows the strengths and weaknesses of 2 tele-operating search and rescue robotic platforms. Refer to **Annex A** for more details on the search and rescue robots.

2.1. PackBot 525

Background & Components. Packbot is a teleoperated robot used for bomb disposal and in hazardous environments. It features a tracked mobility system for diverse terrains, a manipulator arm, and buoyancy control for water operations. It has cameras, laser rangefinders, and chemical sensors for situational awareness, powered by batteries with onboard computers. Its modular design allows for easy customisation, making it adaptable for military and law enforcement use. PackBot's software includes an operator control unit (OCU) for teleoperation and real-time sensor feedback. It also has autonomous features for tasks like waypoint navigation and obstacle avoidance, functioning without direct human control (Teledyne Flir, 2023).

Strength. [1] Operating the PackBot remotely from a safe distance mitigates the risk of injury or fatality in hazardous environments. [2] The PackBot offers real-time feedback through its sensors and cameras, enabling operators to receive immediate information.
Weakness. [1] The PackBot's performance in remote and inaccessible areas may be hindered by a limited communication range, particularly in regions lacking communication infrastructure. [2] The robot's effectiveness is limited in certain tasks due to its inability to lift objects that are too heavy or large, indicating limited manipulation capabilities.

2.2. Spot, Boston Dynamics Robot

Background & Components. Spot, developed by Boston Dynamics, is a versatile quadruped robot designed for remote operations. It can navigate various terrains, including hazardous

environments and disaster areas, thanks to its flexible legs. Spot collects detailed 2D and 3D information using onboard sensors, crucial for inspections, surveys, and mapping environments. Its hardware includes sensors like cameras, LIDAR, and IMUs, along with a payload mounting system for adding additional sensors or tools. The software enables Spot to perform tasks ranging from basic navigation to complex manipulation and decision-making. The Spot API allows applications to control Spot and access sensor information via a client-server model over a network connection (Boston Dynamics, n.d.).

Strength. [1] Spot's four-leg design enables it to navigate diverse terrain with ease, making it suitable for various environments. [2] Spot's modular design allows for easy customisation with different payloads, sensors and tools, making it highly customisable to adapt to a wide range of tasks and applications.
Weakness. [1] While Spot can carry various payloads, its weight capacity is more limited compared to larger robots, which may restrict the types of tasks it can perform. [2] Spot Battery provides power for about 90 minutes of normal operation, hence it needs to be recharged regularly.

2.3. Lessons Learnt

Connectivity. By equipping Alex with an antenna, we have enhanced its communication radius. This improvement allows Alex to maintain more reliable communication over greater distances, which is crucial for remote operations, especially in areas with obstacles or difficult terrain.

Battery Life. We have increased Alex's battery capacity by using six 1.5V batteries instead of four 1.5V batteries to ensure longer operation. With this upgrade, Alex can operate for longer periods without needing to recharge, improving its overall utility and efficiency in the field.

Section 3 System Architecture

This section shows the various components in Alex and how each component communicates with each other. Alex comprises 6 devices, namely the motors, wheel encoders, ultrasonic & colour sensors, as well as the RPlidar and Bluetooth speaker. Figure 1 shows the system architecture of Alex showing all components of Alex and its connections.

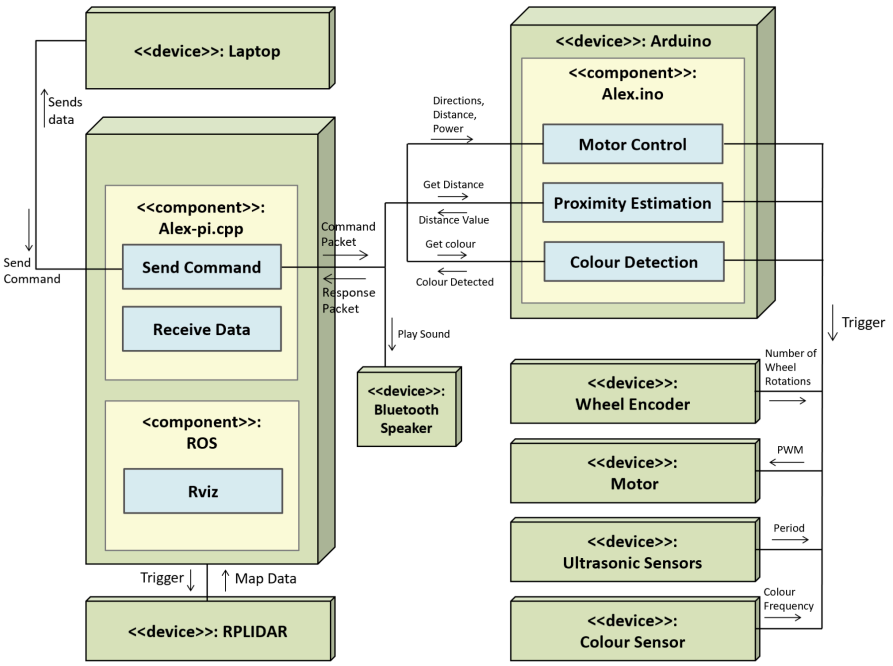


Figure 1: System Architecture

Commands from the Pi directly trigger the following devices. The Pi will send a TPacket of PacketType PACKET_TYPE_COMMAND with the relevant CommandType. Based on this Command Type, the Arduino will send signals to the appropriate devices to perform the command. Once done, it will return a response TPacket of PacketType PACKET_TYPE_RESPONSe with the relevant ResponseType, as well as any additional data the command queried.

Device Name	Received	Response	Details
Colour Sensor	COMMAND_COLOUR	RESP_COLOUR	Arduino will flash the 4 LEDs on the Colour Sensor to obtain the Red, Green & Blue values, scaled down to the 0-255 range. It will then use the Ultrasonic sensor to get the current distance (in cm) to the target it is detecting. It will then return the four values together.
	No Additional Data	uint32_t red uint32_t green uint32_t blue uint32_t distance	
Motor	COMMAND_FORWARD /BACKWARD/ TURN_LEFT/ TURN_RIGHT/ STOP	RESP_OK	In manual mode, Arduino will receive the move command together with an expected move time (in milliseconds) and the power percentage to move with. In auto mode, Arduino will use a hard coded constant value for each move command. Arduino sends the appropriate PWM frequencies to the motors (through the use of the ADFruit Motor shield library) such that Alex moves according to the given command.
	uint32_t time float power	No Additional Data	

Ultrasonic Sensor	COMMAND_ COLOUR	RESP_DIST	Arduino sends a pulse and then measures the duration it takes for an echo to return. The duration is converted into distance using the speed of sound, and this calculated distance is returned as the output.
	No Additional Data	uint32_t distance	
Bluetooth Speaker	COMMAND_ COLOUR	RESP_OK	The Bluetooth speaker receives the command from RPi with the name of the sound file it will then play using the <sox.h> header file.
	char* file_name	No Additional Data	

*A table of additional commands and their respective responses is available in **Annex B**.

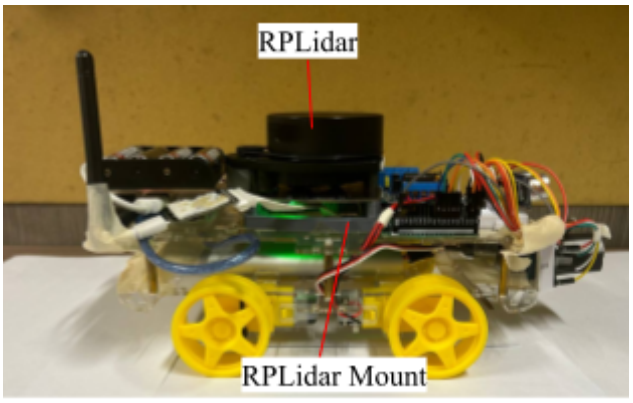
Some devices are not directly controlled using the TPacket Command/Response structure.

Wheel Encoders. The Arduino directly configures interrupts to handle signals from wheel encoders, tracking the number of ticks for wheels. The encoders send tick counts to the Arduino, which calculates and updates the distance values accordingly. These are used when measuring how far Alex has moved, which is used to execute Pi's movement commands.

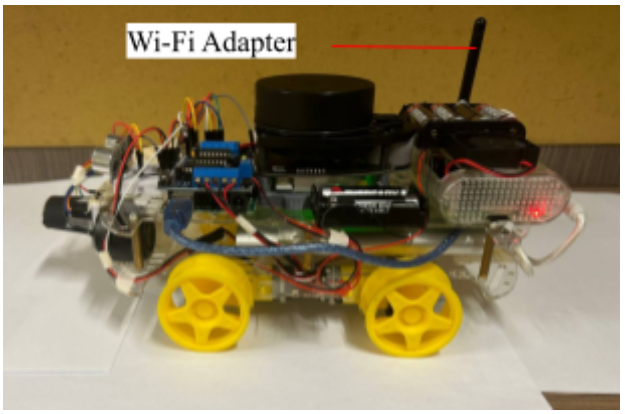
RPLidar. As a separate program utilising the ROS middleware, the Raspberry Pi sends commands to the LIDAR serially through a USB connection, instructing it to perform 360° environmental scans. The LIDAR measures distances based on the duration between sending and receiving light pulses and sends the scanned data back to Raspberry Pi in the format [INFO][TIMESTAMP][ANGLE, DISTANCE]. A separate node in the ROS middleware then publishes said data to a Hector Slam Node, which then publishes the generated map data to the RVIZ Node for the viewer to see the constructed map.

Section 4 Hardware Design

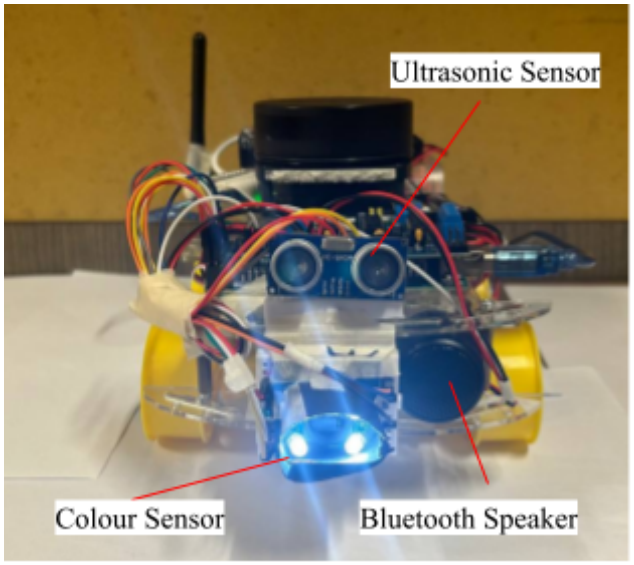
This section explains the design considerations and the hardware specifications in Alex. **Figure 2** shows the final form of our Alex, along with its various hardware components.



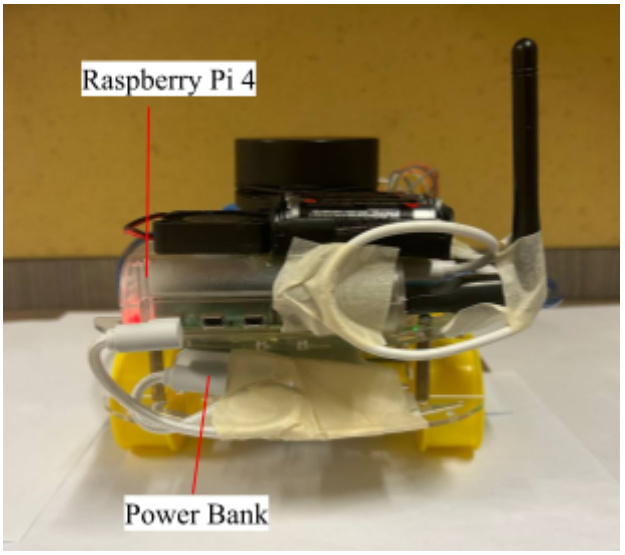
Alex Right View



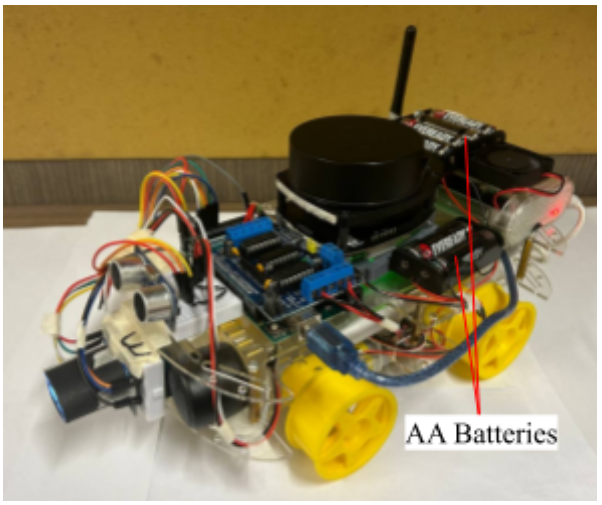
Alex Left View



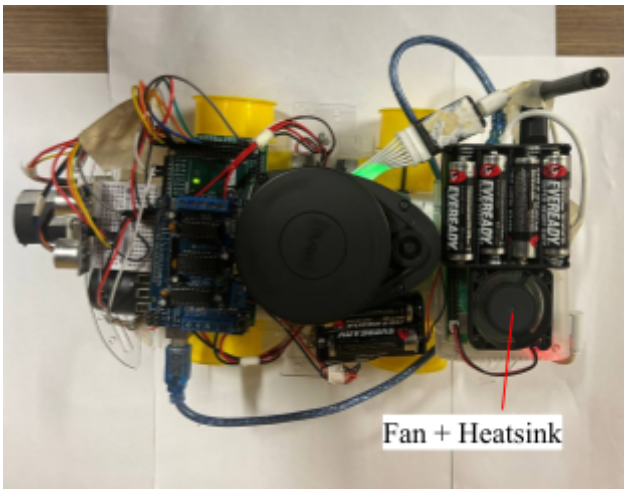
Alex Front View



Alex Back View



Alex Isometric View



Alex Top View

Figure 2: Photograph of Alex

While designing Alex, we have taken accessibility and stability into consideration. For instance, the LIDAR is positioned at the centre and top of Alex to scan the environment unobstructed, while the colour and ultrasonic sensors are mounted at the front to scan objects. Our breadboard is also placed at the top layer so that any changes in the wiring can be done quickly. Moreover, we have done cable management using tape to prevent any unnecessary collision. To maintain stability, we have also aligned the components laterally as much as possible to the centre of gravity approximately in the middle.

4.1. Hardware Components

Alex is supplemented by the following hardware components to enhance its functionalities. Refer to **Annex C** for the Details on Hardware Components.

4.1.1. HC-SR04 Ultrasonic Sensor

The first of the non-standard components is an ultrasonic sensor mounted at the front of the Alex. Measuring the distance between the Alex and any obstacle in front of it allows us to

maintain a specific distance for the colour sensor to obtain accurate and reliable readings. Additionally, the LIDAR is not 100% accurate, especially in tight spaces. Having the ultrasonic sensor in front is crucial in preventing any unwanted bumps or collisions with nearby objects.

4.1.2. Colour Sensor

Our colour sensor is attached to a 3D-printed mount and is placed in front. When the 4 LEDs are lit up, the RGB values of the reflected light are measured and used to determine the colour of the object in front of it - Green, Red, or White. To ensure accurate readings, we have taken measures to shield the sensor from ambient light. We have surrounded the LEDs with black paper, which helps to minimise the impact of external light sources. This shielding ensures that the sensor receives only the light reflected from the object, improving the accuracy of the colour detection process.

4.1.3. Bluetooth Speaker

A speaker connected via Bluetooth to the RPi is mounted near the front of the Alex, placed in between its chassis. When a victim is located by Alex, a specific command will be given and an audio clip will be played according to the colour of the victim. There are 3 different audio files in the RPi corresponding to Green, Red, and White. In the context of Alex's intended purpose - search and rescue, this system is crucial in alerting and swiftly relating information about located victims and their statuses to nearby personnel.

4.1.4. Fan and Heatsink

A fan and heatsink have also been attached to the RPi to manage and dissipate the heat generated by the RPi while it is running. The heatsink acts as a passive cooling component that absorbs and disperses heat away from the RPi while the fan acts as an active cooling component by blowing air over the heatsink, further enhancing heat dissipation. These two prevent our RPi from overheating and aid in preventing performance degradation, ensuring smooth operations.

4.1.5. Lidar Mount

The Lidar Mount is 3D printed to elevate the LIDAR above the rest of the component. This elevation helps to provide an unobstructed view for the LIDAR sensor, allowing it to scan the environment more effectively. Additionally, the mount is designed to securely hold the LIDAR sensor in place, ensuring that it remains stable and aligned with the body of Alex during operation.

Section 5 Firmware Design

This section discusses the algorithm, communication protocol and bare metal implementation aspect of firmware design.

5.1. High-Level Algorithm on Arduino Uno

The Arduino functions as the middle man between the Pi and the individual components of Alex, taking in command packets from the Pi and executing the relevant functions. Once it has completed said command, it will respond with a response packet to signal that the command has been executed.

The Arduino will continuously poll for data packets in its serial connection with the Pi. Once a packet has been successfully received, it will process the data packet using a chain of handle functions, visualised like so:

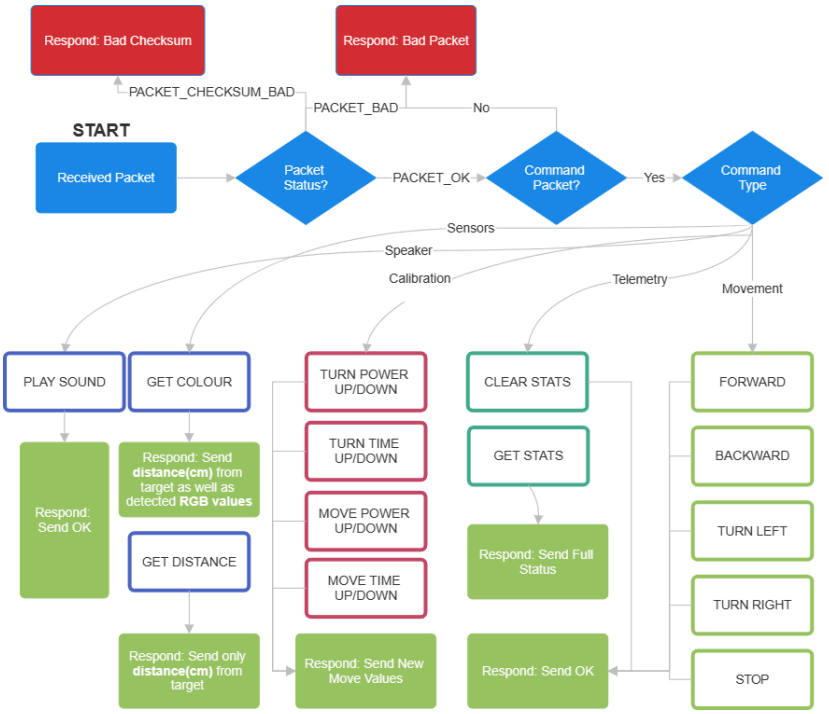


Fig 3: Flowchart for handling received Packet

5.2. Communication Protocol

The Arduino communicates with the Pi via UART serial communication, using 8N1 format with a baud rate of 9600 bps. First, the data is formatted into a TPacket. TPacket is a struct which contains the packet type, the command type (if any) as well as the string and uint32_t data being transmitted as params. There is an additional padding of 2 bits to bring the TPacket size to the 100-bit size that the Pi expects. Then, TPacket is serialised into a TComms struct, where additional values such as the checksum, magic number and more padding are added to the packet before being transmitted, which brings the overall number of bits being transmitted per instruction to 140 bits.

While this approach ensures the correct transmission and receiving of data between the two, it requires constant polling by both sides of the communication protocol. The major downside of this was that we could not “interrupt” our move commands if we accidentally put in a wrong command.

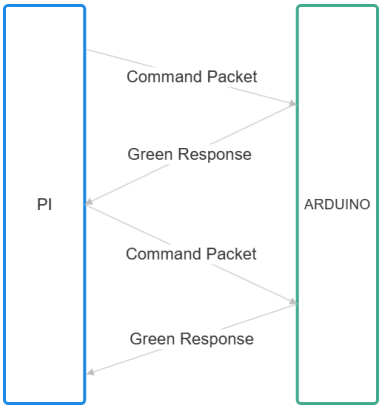


Fig 4: Communication Protocol between Pi and Arduino

Given that the received packets are being read in chunks (i.e. whole 140 bits are not being read at once), it is difficult to "interrupt" an incoming packet with a new one. For example, a "Move Forward" Packet cannot be immediately interrupted by a "Stop Moving" Packet. The Arduino will need to fully read the Forward Packet and execute the following command before it can read the next packet. The Arduino expects to receive one packet at a time and will send a bad packet response if multiple packets are being received at once.

We mitigated this by making each movement command only move the bot a tiny amount. Instead of 1 "Forward" command moving the bot 10 cm, we would use 10 "Forward" commands that would move the bot 1 cm each. This meant that the erroneous move command would not be as dangerous in terms of collisions while keeping movement fluid.

5.3. Bare Metal Implementations

To practise the bare metal programming skills we learnt during this course, we decided to implement our encoders, colour sensor and ultrasonic in bare metal. The relevant code has been added to the **Annex D**.

Section 6 Software Design

6.1 High-Level Algorithm on Raspberry Pi

The Pi acts as the "Brain" of Alex, taking in inputs from the user and sending the corresponding command packets to the Arduino to execute. Additionally, the Pi will run Hector Slam and visualise the resulting map through RVIZ for the operator to reference as they navigate the maze. The summarised algorithm for the Pi is as follows:

1. Initialise communication between Raspberry Pi, Arduino and PC
2. Generating Map using Hector SLAM, visualised with Rviz
3. Pi collects user input and relays the corresponding command to Arduino
4. Arduino processes and executes the command
5. Repeat Steps 2 - 4 until navigation is complete

6.1.1 Teleoperation

We wanted the operation of the bot to feel smooth and fluid to the operator. Therefore, we decided to simulate video game movement with our teleoperation code. In video games, players will hold down the 'w' key to move forward in a straight line. They can also move in 'steps' or

increments by tapping the key instead. We wanted to recreate this 'feel' with our code to make Alex easy to pilot. To do so, we used a combination of stream inputs (which will be expanded upon in 6.2) and a flag to control which inputs will be converted to commands. (Relevant code in **Annex E**)

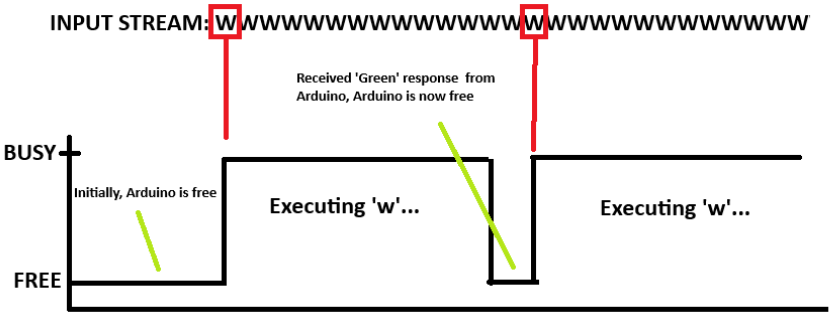


Fig 5: Visualization of Input control

As seen above, the whole stream of 'w' inputs will only trigger the 'MOVE FORWARD' command twice, when the Arduino is free and ready for a new command. Thus, the user could continuously hold down the 'w' key such that Alex will keep moving forward without encountering bad checksum/packet issues when sending too many commands at once. Combining this with our earlier decision to make our move functions move only a tiny amount (5.2), we can now smoothly change Alex's movement and control him like a video game character.

On top of our WASD control commands, we also added calibration commands. In auto mode, our Arduino uses fixed values of both turn/move duration as well as turn/move power. For instance, our move forward command is calibrated to move Alex forward for 300 ms at 80% power. However, there will be situations in the maze where we might want to make smaller movements (when aligning the colour sensor to victims) or larger movements (when rushing back to the parking spot at the end). Therefore, we added commands to increase/decrease these variables to let the pilot change the turn radius/move distance on the fly. This gives us the flexibility to navigate the many challenges of the maze.

As for our sensors, we had a command to retrieve distance, detect RGB + distance as well as our telemetry data from our encoders. We decided to have our GET COLOUR command include the distance value (from ultrasound) with the detected RGB values as we realised that the colour sensor's accuracy varied with the distance to its target. We determined that there was an optimal distance of 7cm where the colour sensor was most accurate. Thus, we displayed the current distance along with the RGB detection to the user so that they could adjust Alex's position before confirming the colour it was detecting.

Finally, we added commands to play sounds using the Bluetooth speaker. Each command will play a different sound to indicate what 'type' of victim it is looking at. This is to simulate real emergencies, where Alex would alert other evacuation personnel when it finds a victim. The sound would automatically play once Alex detected a victim's status, but we bound it to a command in case we wanted to manually 'call' a victim's status.

The overall command schema with the associated purposes can be found in **Annex E**.

6.1.2. Colour Detection

In our project, we utilised the Euclidean distance algorithm to identify the colour of the victim. Initially, we believed there could be multiple colours (beyond just red, green & white), and wanted a more robust colour detection algorithm than simple conditionals. For our algorithm, we first defined a list of expected RGB values for each colour. Then, we calculated the Euclidean distance between the detected RGB value set with our list of expected RGB values. The colour with the "smallest distance" from our detected colour would thus be output as the detected colour. The code for this has been placed in the **Annex F.**

Initially, we faced difficulty in detecting the 'Green' colour. It gave us inconsistent results with there being a 'High' Green value and a 'Low' Green value being detected when the detection occurred at different angles. Thus, we decided to account for both 'light green' and 'dark green' cases with different RGB values and printed both results as 'Green'.

6.2. Additional Software Features

6.2.1. Streaming Input

As mentioned earlier, we used a stream input instead of scanf input to simulate a video game interface. We wanted the movement to be smooth and uninterrupted by the inconvenience of pressing enter after each command. To do this, we first used the getch() function that is available on the conio.h header. This function would allow us to capture each individual character input without needing to press enter. While this worked on the Pi, it would not work on our team's (only Windows) laptop terminals. Thus we shamelessly stole code off the internet for a getch() function that would work regardless of the terminal's platform. (Code in **Annex G**)

6.2.2. Wifi Adapter

We noticed that the RPi tended to occasionally 'drop' connection with the network, which would pose a challenge if it happened frequently during our run. The network speed was also quite slow, with the RVIZ frequently lagging on our VNC view. This threw off our pilot's movement, causing accidental collisions.

To fix this, we bought an adapter to boost the RPi's wifi capabilities. Unfortunately, the adapter was not 'plug and play' and required extensive setup to add to our Pi. This included configuring drivers, updating firmware, and adjusting network settings to ensure compatibility and optimal performance.

Despite the initial challenges, once the adapter was properly set up, we noticed a significant improvement in network stability and speed. The RVIZ application ran smoothly without lag, providing accurate tracking of Alex's movements and reducing the occurrence of accidental collisions. Overall, the effort invested in setting up the adapter was worthwhile.

6.2.3. Customized RVIZ Pose Transform

Another issue with the RVIZ we faced was the use of the red 'Pose' arrow. We noticed that the dimensions of the pose arrow were much longer than the actual length of our Alex. It also did

not account for the width of the Alex. Thus, it was hard to estimate the actual position of Alex in real life from the RVIZ map.

Thus, we added additional 'TF' nodes in our RVIZ launch file to mark a bounding box around our Alex using additional markers offset from the 'Pose' topic. We used standard code from online at first but realised the bounding box was not accurate to our Alex dimensions. After adjusting the values, we now had an accurate bounding box that would greatly improve our "view" of Alex's position in the maze.

6.2.4. Bluetooth Speaker Integration

For our 'cool feature', we wanted to add a Bluetooth speaker to play music cues to accompany the victim detection. We chose this over the traditional buzzer as it played to the RPi's strength of having built-in Bluetooth capabilities. Additionally, buzzers played with the Arduino are limited in that the Arduino is single-threaded. The Arduino cannot play music and move at the same time. The Pi, with POSIX multithreading, could theoretically play music from the speaker while traversing the maze.

Once again, however, the Bluetooth speaker was not as easy to set up and use as we'd hoped. To play the music itself from our code we had to use an external Linux library in libsox. We ended up writing a custom driver code to unpack our sound files and play them through the speaker directly (Code in **Annex H**). Luckily, the concepts we were taught in the course (buffers and UART transmission) were quite applicable in using the sox library. Given our limited time, we ended up binding the "play sound" functionality to a command instead of having it play using another POSIX thread while Alex moved around.

Section 7 Conclusion

Overall, it has been an enriching journey for us in CG2111A. From first learning about GPIO and Bare Metal Programming to Getting Started on Alex and now finishing the Final Run of Alex to the Rescue. Throughout this module, here are the two most important lessons we have learned in this project and the two greatest mistakes we made as a group during our project.

7.1. Lessons Learnt

7.1.1. Hardware Issues

One of our most important lessons is how to properly and calmly identify and deal with hardware-related issues. During the project, there were two significant hardware issues that we faced, one being a faulty colour sensor and another being insufficient power supplied to motors. When we were implementing the colour sensor, we realised that only 3 out of 4 of the LEDs were lit up. This prompted us to ask for a replacement from DSA Lab as we suspected that it was faulty. After the replacement, we found out that the new colour sensor also only had 3 out of 4 of the LEDs lit up. This made us question whether that was the intended function of the colour sensor. After much studying of the colour sensor datasheet and code and asking other groups about their colour sensor, we finally realised that the new colour sensor was also faulty.

Moreover, while we were configuring the motor controls, we realised that Alex couldn't turn. We assumed it was our code that was preventing Alex from turning and spent a great amount of time trying to debug a problem that didn't exist. In the end, when we lifted Alex up, we realised that the wheels were turning correctly all this time and that they couldn't turn as the 6V batteries were not providing enough power to overcome the weight of Alex while turning.

From these experiences, we learnt that we should always check the functionality of a device with the Lab technician on the spot to prevent time from being wasted in finding out whether a device is faulty or not. Also, when something wasn't working, we learnt to be mindful of all the possible causes of failure instead of just tunnel-visioning into one of them.

7.1.2. Effective Teamwork during Runs

Another important lesson we've learnt is the importance of teamwork and communication during the run itself. Before the trial run, we didn't properly plan and take into account all the processes that needed to be done during the run itself. This caused us to be very rushed when drawing out the map on the piece of paper during the minute provided after parking Alex as we decided to pre-draw the map out on our tablet before finalising it on the paper to be submitted. Due to this, we almost didn't manage to finish drawing our map in time for the trial run.

After this experience, we took time to properly delegate roles for the final run and emulate trail runs for ourselves as practice. We've also decided to have 2 people draw the map separately, one on the tablet and the other on the actual paper. While the group in charge of movement and colour sensing are figuring out the colours, the person who is drawing the map on the tablet will then confirm certain areas of the map with his partner who will then start drawing the confirmed area out on the paper all before Alex is parked. Thus, we had plenty of time left after parking Alex as we planned all our actions to be as efficient as possible.

7.2. Mistakes Made

7.2.1. Size/Cable Management

One mistake we made was not planning the cable management as we assembled the robot and added different devices along the way. This caused our wires to be messy with the heavy use of long jumper wires. Moreover, we didn't properly colour-code every wire which made error tracking hard for us. This became apparent when our breadboard, which our ultrasonic was connected to, had some connectivity issues as the ultrasonic wasn't properly reading the distance measured during tests. When we were trying to solve this issue, we wasted a lot of time trying to figure out which wire was connected to which Arduino pin as all the wires were jumbled together and their colours had no meaning. If we had taken a bit of extra time colour coding and neatly wiring the jumper wires when assembling Alex, we could have saved a large amount of time later on when bug-fixing hardware components.

7.2.2. Inexperience in operating Alex

Another mistake we made was not having enough practice for our Alex operator to become fully comfortable with manoeuvring Alex using the keyboard controls before our trial run. As our Alex was quite bulky, partly due to the poor cable management mentioned above, it had a high tendency to bump or scrape into obstacles while turning. During our trial run, when we were manoeuvring Alex to try and scan the victims' colour, we collided with the victim six times. Even though we managed to secure most of the marks for the rest of the components, the collisions cost us a great deal of marks that could have been avoided if we had realised that manoeuvring Alex using keystrokes can be quite challenging, especially under time and space constraints, and had honed our Alex operating skills before the run.

Section 8 References

1. Anderson, B., & Chen, G. (n.d.). *Raspberry Pi LiDAR & SLAM*. Tutorial: Robot Operating System for EPP2. Retrieved April 25, 2024, from https://www.comp.nus.edu.sg/~guoyi/tutorial/cg2111a/ros-slam/

2. Boston Dynamics. (n.d.). *Spot*. Boston Dynamics. Retrieved April 24, 2024, from https://bostondynamics.com/products/spot/

3. Car Expert. (2021, February 26). *How Do Parking Sensors Work? Radar and Remote Parking Technology Explained*. JustLuxe. Retrieved April 25, 2024, from https://www.justluxe.com/lifestyle/luxury-cars/feature-1971332.php

4. Hakami, A. (2013, November 12). Application of Soft Systems Methodology in Solving Disaster Emergency Logistics Problems. *International Journal of Industrial Science and Engineering*, *7*(12), 8. Retrieved April 24, 2024, from https://www.researchgate.net/publication/259674878_application-of-soft-systems-methodology-in-solving-disaster-emergency-logistics-problems#pf2

5. Teledyne Flir. (2023). *PackBot® 510*. Teledyne FLIR. Retrieved April 24, 2024, from https://www.flir.com/products/packbot/?vertical=ugs&segment=uis

Section 9 Annex

9.1. Annex A: Elaboration on the State of the Art

This section allows us to examine the strengths and weaknesses of the robot and how we can improve and integrate the functions into Alex.

PackBot 525

Packbot 525 is the newest iteration of the Packbot robot which can perform a wide range of operations. The enhancements include state-of-the-art HD cameras, improved illumination, optional in-situ charging, the addition of a laser range finder, enhanced accessory ports, and more attachment points for add-on accessories (Teledyne Flir, 2023). **Figure 9.1.1.** illustrates the Packbot 525.

Figure 9.1.1: Packbot 525

The specifications and observations on how we can integrate them into our Alex is detailed in **Table 9.1.1.** Do note that due to considerations regarding our project funds and duration, it may not be possible to include some of the specifications in the current project. However, with more funds and time, the following advancements are possible for Alex's functionalities.

Table 9.1.1: Specifications and Improvements

Specifications	Improvements
Extended Arm	This feature can be integrated into our search and rescue robot, as its strong extended arm can help remove rubble in emergency situations.
HD Cameras with Night Vision	In emergency situations, visibility can be severely limited, especially during nighttime or in environments with low light conditions. These cameras allow the robot to capture high-definition images even in the dark, enabling it to navigate and gather important visual information in areas where human rescuers might struggle to see.
Optical Zoom	It allows the robot to zoom in on distant objects or areas of

	interest without sacrificing image quality, providing a closer and more detailed view of the surroundings.This can be used to identify specific details, such as signs of life or potential hazards, from a safer distance.
Distribute GPS Location	This capability allows the robot to share its precise location with rescue teams, enabling them to coordinate their efforts more effectively. Knowing the exact location of the robot can help rescue teams deploy resources more efficiently, such as directing them to the most critical areas or guiding them to the robot's location for maintenance or retrieval.
Live Video	It allows real-time visualization of the robot's surroundings, enabling operators and rescue teams to assess the situation and make informed decisions quickly. Operators can remotely control the robot and navigate it through complex environments, providing valuable situational awareness without putting human rescuers at risk.

Spot, Boston Dynamics Robot

Spot is another tele-operated robot that can be used for search and rescue. It is able to navigate rough terrain and inaccessible areas, where human rescuers might have difficulty reaching or maneuvering. Spot is also equipped with advanced sensors and cameras that allow it to collect data and provide real-time information about the environment. **Figure 9.1.2** illustrates Spot, Boston Dynamics Robot

Figure 9.1.2: Spot, Boston Dynamics Robot

The specifications and observations on how we can integrate them into our Alex is detailed in **Table 9.1.2.** Do note that due to considerations regarding our project funds and duration, it may not be possible to include some of the specifications in the current project. However, with more funds and time, the following advancements are possible for Alex's functionalities.

Table 9.1.2: Specifications and Improvements

Specifications	Improvements
Mobility	Spot's four-legged design allows it to navigate various terrains, unlike traditional robots that use tracks or wheels. This design enables Spot to recover from falls and climb stairs, capabilities that wheeled robots do not typically have.

9.2. Annex B: Additional Command/Response Packets

Table 9.2. details the additional command/response packets built in Alex.

Table 9.2: Additional Command/Response Packets

Command Packet		**Response Packet**	
Command	**Params**	**Command**	**Params**
MOVE_TIME_UP	uint32_t new_time float new_power	RESP_UPDATE	uint32_t curr_move_time float curr_move_power uint32_t curr_turn_time float curr_turn_power
MOVE_TIME_DOWN			
TURN_TIME_UP			
TURN_TIME_DOWN			
MOVE_POWER_UP			
MOVE_POWER_DOWN			
TURN_POWER_UP			
TURN_POWER_DOWN			
COMMAND_GET_STATS	None	RESP_STATUS	uint32_t leftForwardTicks uint32_t rightForwardTicks uint32_t leftReverseTicks uint32_t rightReverseTicks uint32_t leftForwardTicksTurns uint32_t rightForwardTicksTurns uint32_t leftReverseTicksTurns uint32_t rightReverseTicksTurns uint32_t forwardDist uint32_t

			reverseDist
COMMAND_CLEAR_STATS	None	RESP_OK	None
<Any Bad Command>	<Any Params>	RESP_BAD_PACKET	None
		RESP_BAD_CHECKSUM	
		RESP_BAD_COMMAND	
		RESP_BAD_RESPONSE	

9.3. Annex C: Details on Hardware Components

HC-SR04 Ultrasonic Sensor

The ultrasonic sensor comprises one emitter and one receiver. The emitter will emit high-frequency sound waves and when the emitted sound waves encounter an object, they bounce off the surface and return back to the receiver. By measuring the time it takes for the sound waves to travel to the wall and back, we are able to measure the distance by applying the following formula:

$$Distance = (Time\ x\ Speed\ of\ Sound) / 2,\ where\ Speed\ of\ Sound = 340m/s$$

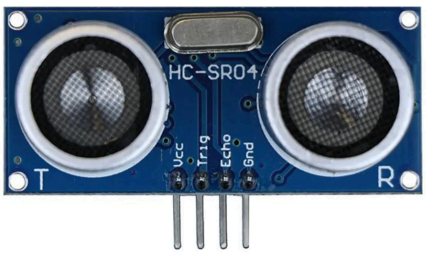


Figure 9.3A: HC-SR04 Ultrasonic Sensor

The inspiration to implement the ultrasonic sensor was drawn from car parking sensors. In situations where the LIDAR sensor becomes inaccurate at very close distances from an obstacle, it can be challenging to determine the exact distance away from the obstacle. Similar to how cars use ultrasonic sensors for parking assistance, implementing an ultrasonic sensor in Alex can help improve distance detection and prevent collisions by providing more accurate measurements in close proximity to obstacles.

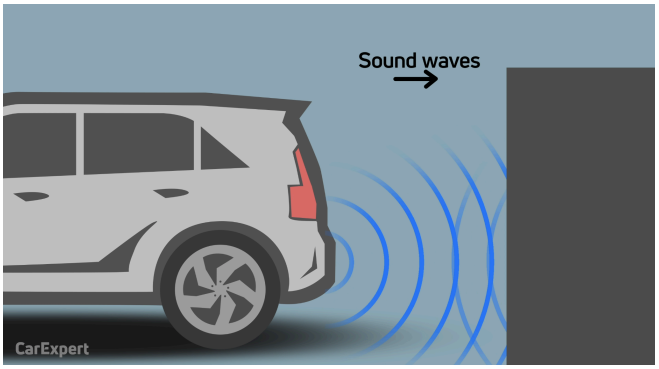


Figure 9.3B: Parking Ultrasonic Sensor on Car (Car Expert, 2021)

The ultrasonic sensor is positioned directly above the color sensor to measure the distance between the color sensor and the victim accurately. This setup aims to enhance the accuracy of color detection by ensuring that the color sensor is consistently positioned at a specific distance from the victim. Calibration of the color sensor based on this fixed distance further improves the accuracy of color detection.

Colour Sensor

The color sensor has been positioned at the front of the robot using a 3D printed mount to ensure it is securely attached to the main body. This setup helps maintain the stability of the color sensor during operation, ensuring accurate and consistent readings.

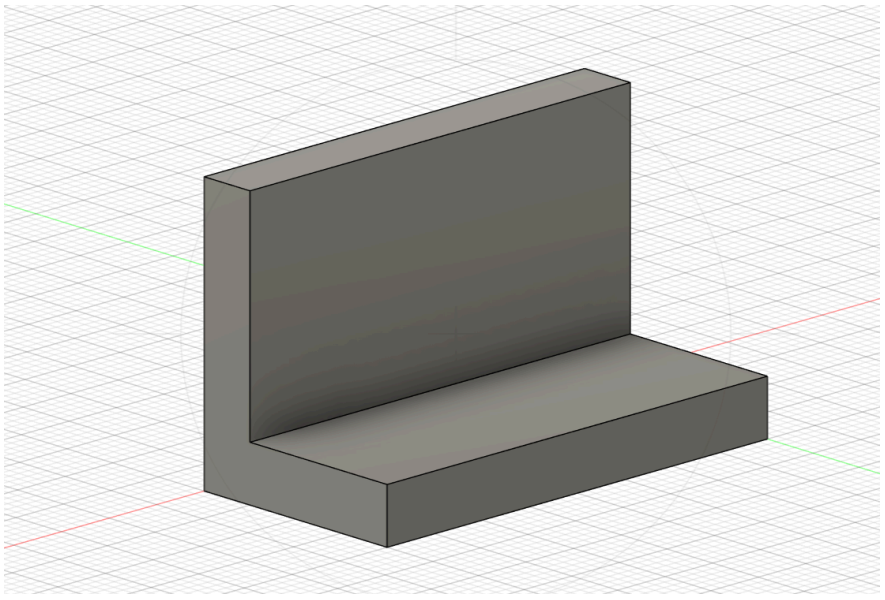


Figure 9.3C: Colour Sensor Mount

Bluetooth Speaker

We have chosen this bluetooth speaker due to its small size and loud volume. Given the limited space on the robot, the compact size of the speaker allows for efficient use of space, leaving more room for other components like sensors and medical aid items. Additionally, the speaker's loud volume is crucial for effectively delivering instructions to victims located at a distance, ensuring they can hear and respond to commands clearly.

Figure 9.3D: Bluetooth Speaker

Fan and Heatsink

The operating temperature for a Raspberry Pi is between 0°C and 85°C. Due to the extensive calculations required to render teh SLAM Map from LIDAR data can significantly burden the CPU of Raspberry Pi (Anderson & Chen, n.d.). We have set up one fan to actively push air over your Raspberry Pi's surface to keep it cool. A continuous airflow helps maintain an optimal temperature for your Raspberry Pi. The fan is powered directly from the GPIO pin.

We also attach heatsink which is a small metal block to the CPU. They are made out of thermally conductive material, and these small blocks helps to conduct the heat out of the processor and release it into the air.

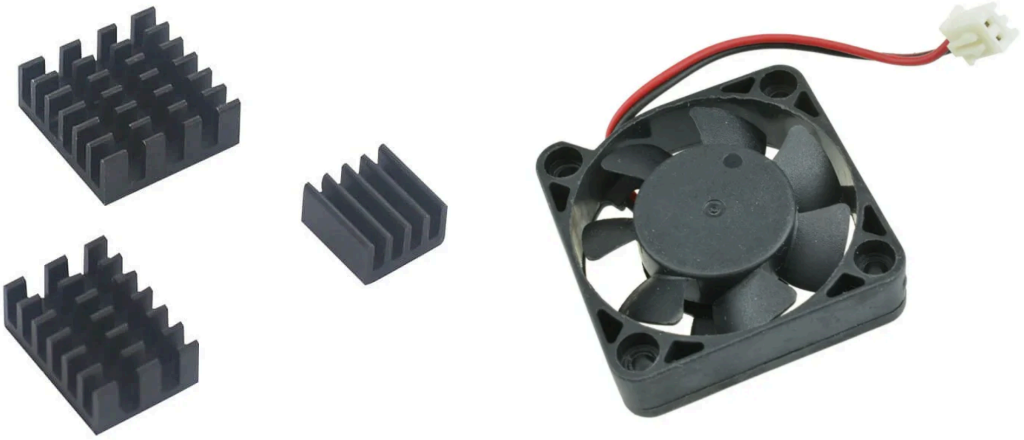


Figure 9.3D: Heat Sink and Fan

Lidar Mount

The Lidar mount is designed to securely hold the LIDAR in place by allowing its four legs to be inserted into corresponding holes. This design ensures that the LIDAR remains stationary, improving the accuracy of the map generated in Rviz. Additionally, the holes in the mount serve a dual purpose by also aiding in wire management, keeping the setup neat and organized.

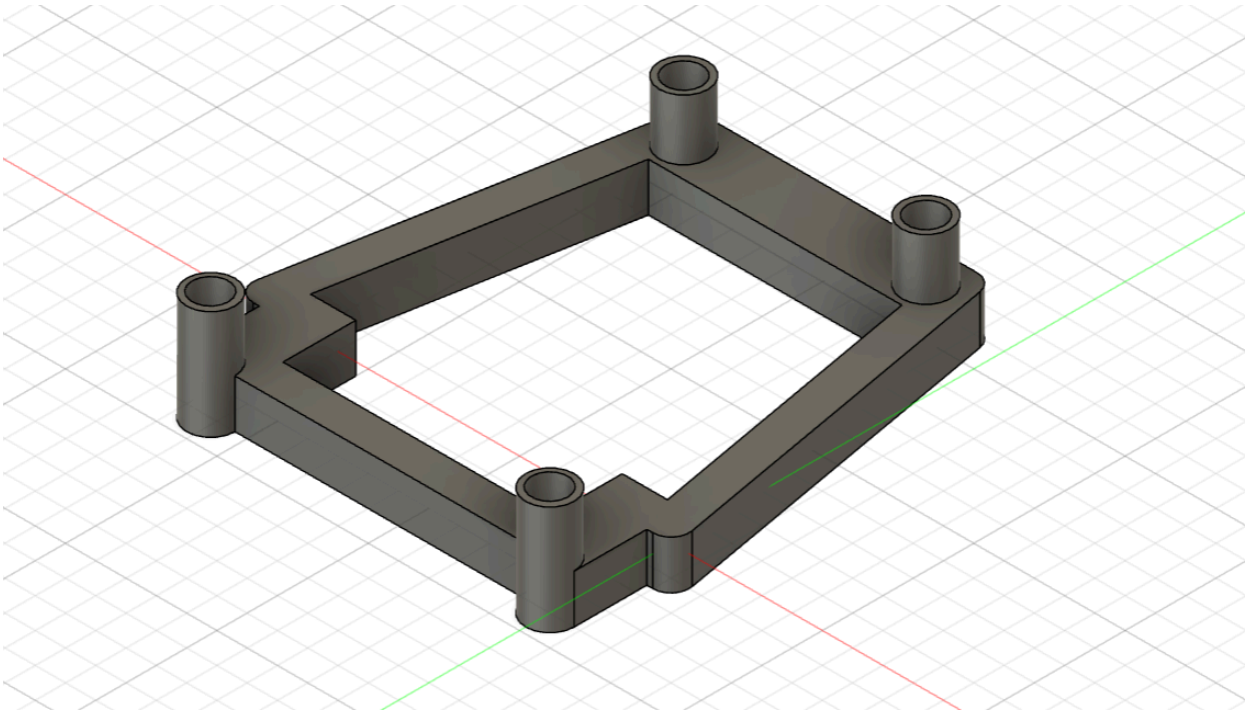


Figure 9.4E: Lidar Mount

9.4. Annex D: Bare Metal Implementations

```
void enablePullups()
{
  //Sets pins 18,19 to pullup resistor mode
  int state = 0b00001100;
  DDRD  &= ~state; //set INPUT
  PORTD |= state; //set HIGH
}

void setupEINT()
{
  //Enable INT2, INT3 interrupts
  EIMSK |= (1 << INT3) | (1 << INT2);
  //Set interrupt trigger mode to rising edge
  EICRA |= (1 << ISC31) | (1 << ISC30) | (1 << ISC21) | (1 << ISC20);
}
```

```
// Setting up colour pins
void setupColour() {
  //set sensor out pin to input
  DDRA &= ~(1 << 7);
  //set s0-4 pins to output
  DDRC |= (1 << 6) | (1 << 4) | (1 << 2) | 1;

  // Setting frequency scaling to 20%
  PORTC |= 1 << 6; //s0 high
  PORTC &= ~(1 << 4); //s1 low
}
```

```
void getColour(){
  // Setting Red filtered photodiodes to be read
  PORTC &= ~( 1 << 2 | 1 << 0 );
  delay(100);
  // Reading the output frequency for RED
  redFreq = getAvgFreq();
  delay(100);
  // Setting Green filtered photodiodes to be read
  PORTC |=  1 << 2 |  1 << 0;
  delay(100);
  // Reading the output frequency for GREEN
  greenFreq = getAvgFreq();
  delay(100);
  // Setting Blue filtered photodiodes to be read
  PORTC &= ~( 1 << 2 );
  PORTC |= 1 << 0;
  delay(100);
  // Reading the output frequency for BLUE
  blueFreq = getAvgFreq();
  delay(100);
}
```

```
void setupSensor(){
  DDRA &= ~(1 << 0); //echo pin input
  DDRA |= 1 <<  1; // trig pin output
}

uint32_t getSensorDistance(){
  // Clears trigPin
  PORTA &= ~(1 << 0);
  delayMicroseconds(2);

  // Sets trigPin to HIGH for 10 microsecs
  PORTA |= 1 << 0;
  delayMicroseconds(10);
  PORTA &= ~(1 << 0);

  unsigned long microsecs = pulseIn(echoPin, HIGH);
  return microsecs * SPEED_OF_SOUND / 2;
}
```

9.5. Annex E: Input Stream Control Code & Overall Command Schema

```
void getCommand()
{
  uint32_t distance;
  char ch;
  while(!exitFlag) {
    distance = 0;

    if (manual) {
      printf("\nManual Command Mode\n");
      scanf("%c",&ch);
      if (ch != 'x') scanf("%d",&distance);
      flushInput();
    } else {
      ch = getch();
    }

    if (!busyFlag) {
      busyFlag = 1;
      printf("\nchr is %c\n",ch);
      sendCommand(ch, distance);
    }
  }
}
```

Command	Key	Purpose
Move Forward	W	Move forward for 300 ms at 80% power.
Move Backward	S	Move backward for 300 ms at 80% power.
Turn Left	A	Turn left for 400 ms at 100% power.
Turn Right	D	Turn right for 400 ms at 100% power.
Stop	X	Stop moving.
Toggle Manual Mode	M	Instead of input stream, Alex will switch to being controlled by scanf inputs in format direction,distance,power.
Move Power Up	U	Increment Move Power by 10%
Move Power Down	Y	Decrement Move Power by 10%
Turn Power Up	J	Increment Turn Power by 10%
Turn Power Down	H	Decrement Turn Power by 10%
Move Time Up	T	Increment Move Time by 100ms
Move Time Down	R	Decrement Move Time by 100ms
Turn Time Up	G	Increment Turn Time by 100ms
Turn Time Down	F	Decrement Turn Time by 100ms
Get Distance Reading	Q	Get current distance reading (cm)
Get Colour Reading	E	Get both distance reading & detected RGB values
Reset Statistics	C	Clear all stats to 0
Get Statistics	Z	Get all Alex statistics
Play ‘Green’ Sound	I	Alert personnel that Alex has found a victim
Play ‘Red’ Sound	K	Alert personnel that Alex had a false alarm
Play ‘White’ Sound	L	Alert personnel that Alex needs a double check of the victim’s status

9.6. Annex F: Colour Detection Algorithm

```
int colours[numOfColours][3] = {
    {255,255,255},
    {0,0,0},
    {255,0,0},
    {255,165,0},
    {255,225,0},
    {0,255,0},
    {0,128,0},
    {0,0,255},
};

const char* colourName [numOfColours] = {
    "White", "Black",
    "Red","Orange",
    "Yellow","Green",
    "Green","Blue"
};

void detectColour (uint32_t R, uint32_t G, uint32_t B){
    int nearestIndex = 0;
    int nearestDistance = 255 * 255 * 255;
    for (int i = 0;i<numOfColours;i++){
        int deltaR = colours[i][0] - R;
        int deltaG = colours[i][1] - G;
        int deltaB = colours[i][2] - B;

        int distance = (deltaR*deltaR) + (deltaG*deltaG) + (deltaB*deltaB);
        if (distance < nearestDistance){
            nearestDistance = distance;
            nearestIndex = i;
        }

    }
    printf(colourName[nearestIndex]);
}
```

9.7. Annex G: Streaming Input Code

```
char getch() { // im gonna be real no clue how this works but it does :)
        char buf = 0;
        struct termios old = { 0 };
        if (tcgetattr(0, &old) < 0)
                perror("tcsetattr()");
        old.c_lflag &= ~ICANON;
        old.c_lflag &= ~ECHO;
        old.c_cc[VMIN] = 1;
        old.c_cc[VTIME] = 0;
        if (tcsetattr(0, TCSANOW, &old) < 0)
                perror("tcsetattr ICANON");
        if (read(0, &buf, 1) < 0)
                perror("read()");
        old.c_lflag |= ICANON;
        old.c_lflag |= ECHO;
        if (tcsetattr(0, TCSADRAIN, &old) < 0)
                perror("tcsetattr ~ICANON");
        return (buf);
}
```

9.8. Annex H: Bluetooth Speaker Driver Code

```
void playSound(char* filename) {
    // Initialize SoX
    sox_init();
    //read from file on Pi i.e. 'red.wav'
    sox_format_t* in = sox_open_read(filename, NULL, NULL, NULL);
    //output to "-", which sets output to default audio device
    sox_format_t* out = sox_open_write("-", &in->signal, NULL, "alsa");

    size_t buffer_size = 2048;
    vector<sox_sample_t> buffer(buffer_size);
    size_t samples_read;

    while ((samples_read = sox_read(in, buffer.data(), buffer_size)) > 0) {
        sox_write(out, buffer.data(), samples_read);
    }

    // cleanup
    sox_close(in);
    sox_close(out);
    sox_quit();

    return 0;
}
```