

**CG2111A Engineering Principles and Practice II**

Semester 2, AY 2021-2022

**“Alex to the Rescue”**

**Final Report**

Team: B04-4A

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Student number** | **Sub-Team** | **Role** |
| Poh Wei Pin | A0233849U | Hardware | Hardware IC |
| Samuel Tan Sze Wee | A0234575Y | Firmware | Firmware IC |
| Vu Van Dung | A0243323R | Software | Software IC |

# Table of Contents

[**Table of Contents**](#_eajkiw4zasx8) **1**

[**1. System Functionalities**](#_2ts2eqwdvb33) **2**

[**2. Review of the State of the Art**](#_3l2wafequt93) **2**

[2.1. VGTV-Xtreme Robot](#_3xbg6n7l5h64) 2

[2.1.2. Strengths of VGTV-Xtreme](#_a13l42djjth7) 2

[2.1.3. Weaknesses of VGTV-Xtreme](#_gdxhkgjugyz4) 2

[2.2. Spot Robot from Boston Dynamics](#_ur68146bses3) 3

[2.2.2. Strengths of Spot](#_gkk5bnvlbrhe) 3

[2.2.3. Weakness of Spot](#_wp0vt9binsl3) 3

[**3. System Architecture**](#_7jnrh462fx9l) **3**

[**4. Hardware Design**](#_m9g1lbuu5vkw) **3**

[4.1. Section Overview](#_9hr1p8wqk4uj) 3

[4.2. Overall Alex Design](#_eci66gxor2wq) 4

[4.3. Additional Hardware Component](#_iy588294jcyp) 4

[**5. Firmware Design**](#_t8gtqllouloa) **5**

[5.1. Section Overview](#_7qxxgbd2d2h3) 5

[5.2. Communication Protocol](#_fv636u8aypyi) 5

[5.3. USART Bare Metal](#_mu2264jpvys4) 7

[5.4. Navigation Algorithm Overview](#_lial84jp12o9) 7

[5.5. Setup & Initialisation of Alex](#_i33ymjs1iokf) 7

[5.6. Execution of Movement Commands](#_v3y3pamn1sin) 8

[5.7. Bonus Movement Features](#_d5cokksjfthc) 9

[**6. Software Design**](#_fh13mtukievs) **9**

[6.1. RPLidar Data Handling](#_vaxpa4xld681) 9

[6.1.1. Web Server and Front-end Browser-based Client to Display Data](#_sn7gu6eyp7mo) 9

[6.1.2. RPLidar Code Running on Raspberry Pi](#_16babv9yprlw) 10

[**7. Lessons Learnt & Conclusion**](#_v5u178gd2n3a) **11**

[7.1. Two Key Takeaways from CG2111A Project](#_wrmkacxt04k5) 11

[7.2. Two Greatest Mistakes Made](#_m7hrf8rlsq61) 11

[**References**](#_3atzey3c7hsa) **12**

# 1. System Functionalities

We intend to create a robot, named Alex, that is intended to be a “search and rescue” robot that can be piloted around remotely in an unknown environment. Its role is to serve as an “eye” of the user and provide real-time information on the environment to the user to allow him/her to navigate the whole terrain and draw a detailed map of the environment. This is useful as robots such as ALEX are often deployed after natural disasters such as earthquakes, to identify survivors.

The robot is equipped with a LIDAR device to map the said unknown environment. The user would then be able to send movement commands to Alex and receive environment information (the angle and distance to the nearest obstacle in any particular direction) back to the user. The user can then plot the points with a plotting program such as Gnuplot, or post-process the data points with a SLAM algorithm, to progressively draw a detailed map of the obstacles within the unknown environment, without having to look at that environment in person.

# 2. Review of the State of the Art

## 2.1. VGTV-Xtreme Robot

VGTV-Xtreme is a remotely operated search and rescue robot by American Standard Robotics. It is a small robot with black tracks with high friction and is equipped with a video camera that is meant for search and rescue operations. It can change its layout to move in complicated terrains, hence with the camera delivering real-time broadcast, the robot can help rescue teams in searching and rescuing victims in difficult environments. [1].

### 2.1.2. Strengths of VGTV-Xtreme

* Small size and polymorphic shape-changing ability
* Different positions of the robot have different functionalities

### 2.1.3. Weaknesses of VGTV-Xtreme

* Unable to operate while it is flipped to the side or inverted
* Has de-tracking issues when operating on high traction surfaces

## 2.2. Spot Robot from Boston Dynamics

Spot is a four-legged robot that is used to reach and navigate rough terrains that are not accessible or risky by humans [2]. It is equipped with 5 high-tech cameras, which can deliver a full 360 degrees view of the surrounding environment. The operator uses this view to access the situation and control the robot remotely [3].

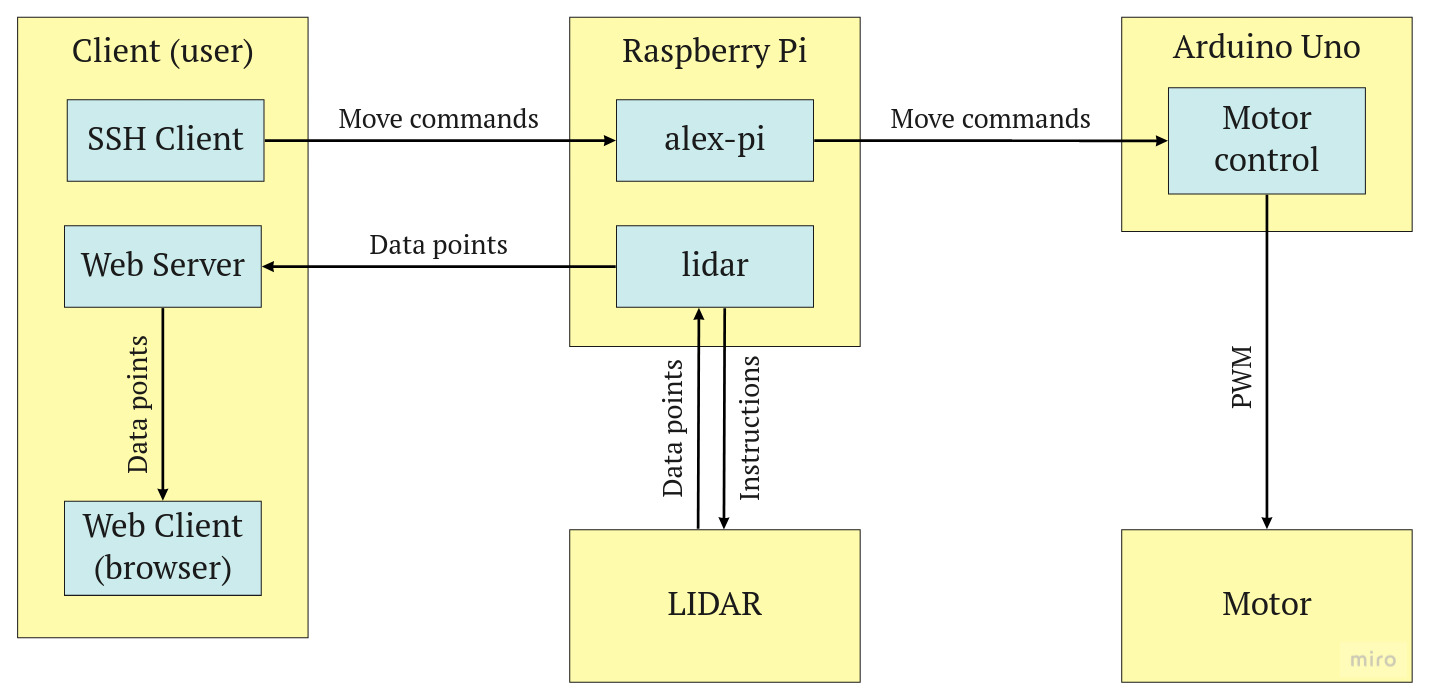
### 2.2.2. Strengths of Spot

* Can get back up even when inverted, which is useful for remote surveillance
* Can automatically avoid obstacles

### 2.2.3. Weakness of Spot

* The cameras are unable to tell apart hard and soft terrain
* Large size makes it harder to traverse tight areas

# 3. System Architecture



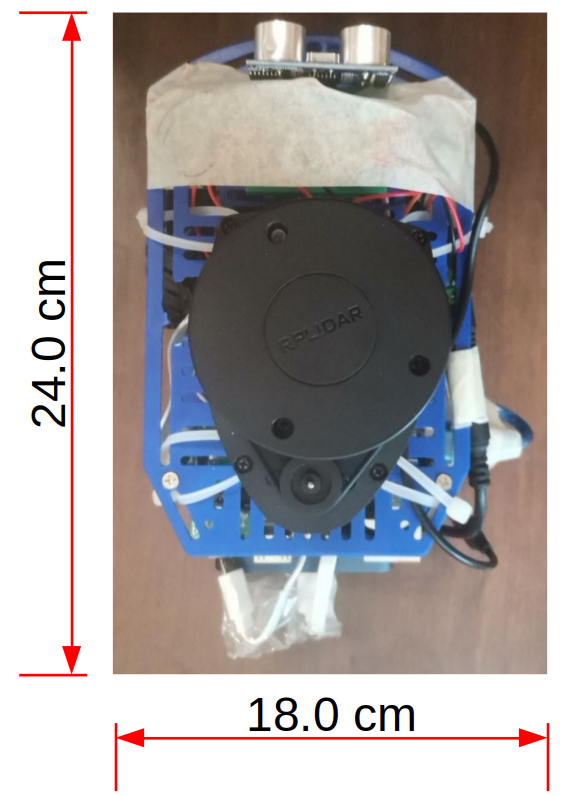
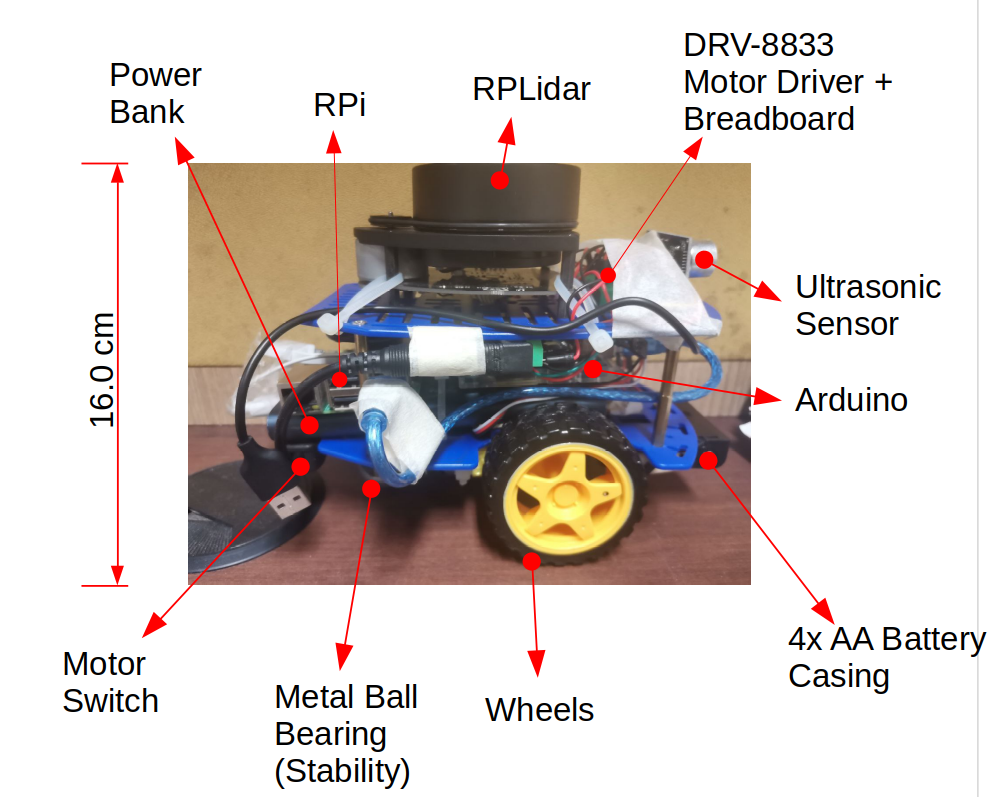
# 4. Hardware Design

## 4.1. Section Overview

A basic search and rescue robot needs to have a low centre of gravity (CG), area mapping ability, and obstacle avoidance, to operate successfully. Hence, there need to be careful considerations when designing the robot. In this section, we will be discussing Alex’s hardware design choices and the rationales behind them.

## 4.2. Overall Alex Design

The final design of Alex is shown in the pictures below. It is configured to position the heavier components such as batteries and power bank at the bottom of Alex which helps to lower the CG of Alex and prevents it from toppling when traversing over humps.



Lighter components such as the RPi, and Arduino are placed above the power bank in the middle section of Alex. The lightest component, which is the breadboard, is placed at the top of Alex alongside the RPLidar. As the CPUs are close to the breadboard, the wiring of the circuits is made easier. Shorter wires can also be used which reduces the overall bulk of Alex. A masking tape is placed over the breadboard to prevent the wires from fidgeting and loosening when Alex is operating. Lastly, RPLidar is placed at the highest position of the Alex to have a clear and unobstructed view of the area which helps to map out the surrounding. The lidar is placed 16 cm above ground which is still lower than the height of the “walls” used for our Final Demo. This allows Alex to map the environment without the fear of scanning over the “walls”.

## 4.3. Additional Hardware Component

An ultrasonic sensor is placed in front of Alex which stops Alex when it senses a certain distance between Alex and the “wall”. This helps to serve as an obstacle avoidance for Alex in situations when it overshoots the distance it is supposed to travel. (We used this as a secondary safety net on top of our lidar navigation system).

# 5. Firmware Design

## 5.1. Section Overview

The Arduino Uno plays a pivotal role in the execution of navigation commands as well as status tracking in our project. In this section, we will explore the high-level algorithm implemented by our team on the Arduino Uno as well as the communication protocol that enables smooth communication between the Arduino and RPi.

## 5.2. Communication Protocol

Text

Description automatically generated with medium confidence

First, the communication protocol between RPi and Arduino is interfaced using USART (Universal Synchronous Asynchronous Receiver Transmitter) which is inbuilt on Arduino Uno’s Atmega328p Microcontroller. Prior to any transfer of data/communication, the Arduino and RPi will agree on a set of communication specifics. In this project, our communication configuration is as follows:

1. Baud Rate: 9600
2. Number of Data Bits: 8
3. Parity: None
4. Stop Bits: 1

By default, these settings can be easily managed using standard Arduino libraries but in our implementation of bare metal serial communication, these settings are configured on the respective USART Bit registries (i.e UCSR0C).

Apart from this, another key aspect that is very important would be to align the communication protocols between RPi and Arduino in order to ensure that data is being written and read correctly on both computers. For instance, data sizes, checksums, and data padding are very important attributes to consider when designing the communication protocol. Below are some examples:

|  |
| --- |
| typedef struct {  // Type of data packet transmitted (Command/Response/Error)  char packetType;  // Movement/Get Status Commands  char command;  // Padding to make up 4 bytes  char dummy[2];  // String data  char data[MAX\_STR\_LEN];  // Used for storing relevant data/parameters for specific commands  uint32\_t params[16]; } TPacket; |

*Code Snippet 5.1 from packet.h - TPacket (Used in Transfer of Data)*

|  |
| --- |
| typedef struct comms {  // The magic number used to verify the communication works  uint32\_t magic;  uint32\_t dataSize;  char buffer[MAX\_DATA\_SIZE];  // Storing checksum to be checked  unsigned char checksum;  // Padding  char dummy[3]; } TComms; |

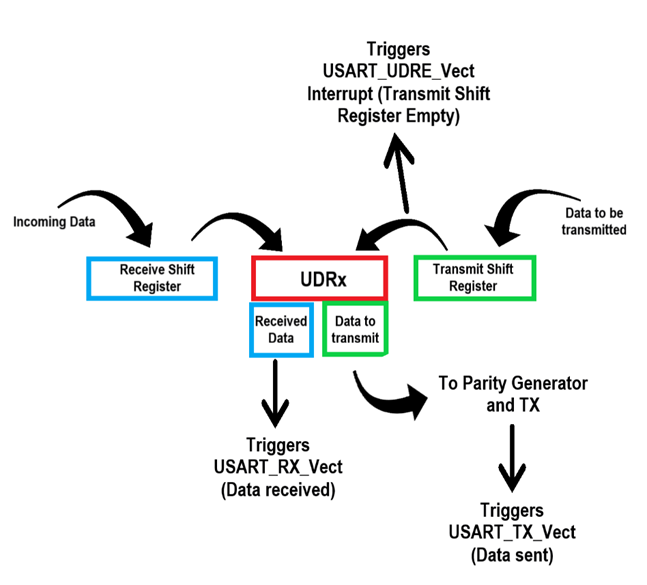
*Code Snippet 5.2 from serialize.cpp - TComms (Used in Serialisation & Deserialization)*

After configuring the communication protocol as seen above, commands can be sent and received between the computers (RPi and Arduino) in the form of TPacket (Code Snippet 5.1). Furthermore, on the Arduino’s end, an interrupt is triggered whenever data is read or written onto the buffer. This prevents unnecessary and unproductive waiting by the processor as seen in data-polling.

As illustrated in the diagram in section 5.6, once TPacket sent from the RPi is received by the Arduino, the contents in TPacket are read and processed accordingly by readPacket(), handlePacket(), handleCommand(). To resolve the issue of bad packet (Checksum) or packet loss (ACK), Arduino will return an ‘OK’ packet upon successful reading and processing of the data packet back to the RPi. This is essential as the RPi would be able to tell whether there is any issue in the communication. Similarly, the checksum would be able to verify the integrity of the packet received by the computer(s). However, we should also recognise that checksum may not work well if the order of the data sequence is changed or the differences offset each other. (Even number of errors)

## 5.3. USART Bare Metal

|  |
| --- |
| void setupSerial() {  // Set Baud Rate to 9600BPS  UBRR0L = 103;  UBRR0H = 0;  // Mode: Asynchronous USART, Parity: DISABLED, STOP BIT: 1 BIT, Bit Size: 8  UCSR0C = 0b00000110;  UCSR0A = 0;  // Initialise 2 Circular Buffers, 1 for read, 1 for write  initBuffer(&RBuffer, 128);  initBuffer(&XmitBuffer, 128); } |



From above, the USART implementation uses a circular buffer that was provided to us in the buffer library. On top of the setup code above, we would also have to implement the necessary interrupt service routines (see image beside) as well as configure the serial read and write function for the communication (please refer to our team’s code submission).

## 5.4. Navigation Algorithm Overview

1. Setup and initialisation of Alex (Lidar, RPI, Arduino Communication Protocol)
2. Environment Surroundings Mapping using Lidar and user navigation input via RPI to Arduino (Serialisation of command into data packets)
3. Deserialization of data packets into commands in Arduino (USART communication)
4. Execution of command(s) (forward, reverse, left, right, get/clear status, exit).
5. Concurrently, Lidar readings will be processed and displayed on the operator's laptop from Alex.
6. Repeat steps 2 to 5 until the end of navigation.

## 5.5. Setup & Initialisation of Alex

1. Establishing LAN network between laptop (Client) and Alex (Server) via mobile hotspot or Wi-Fi
2. Running Alex’s Lidar service on the local network
3. Starting up ./alex-pi on RPI to kickstart the communication protocol
4. Serial connection to Arduino from RPI begins, RPI will spawn a receiver thread and send a “helloPacket” to test connection (sendPacket -> serialWrite to buffer). (Simultaneously, Arduino will wait for a hello packet via waitForHello() function)
5. helloPacket received, Arduino will invoke the sendOK() function back to RPI to acknowledge. Hereon, connection between RPI and Arduino has been successfully established. Arduino will now listen for user commands.

## 5.6. Execution of Movement Commands

Movement control of Alex mainly involves Arduino’s PWM Pins, 1x DRV-8833 Dual Motor Driver Carrier, 2 x Wheel Encoders, and 2 x Motors.

1. The DRV-8833 Dual Motor Driver will convert the low-current signal from the controller circuit and amps it up into a high-current signal that will drive the 2 motors. Each input will be responsible for the forward and reverse movement of each respective wheel.
2. To measure the distance travelled by Alex, wheel encoders will be used to measure the number of times the wheels have rotated.
3. Once the command is received from RPI to Arduino (Direction, Distance, and Power), the speed of the rotation of the wheels is controlled by PWM timers on the Arduino. (We used Timer 0 for the left wheel, Timer 1 for the right wheel)

Timeline

Description automatically generated

|  |  |
| --- | --- |
| **Movement** | **Instructions | Timer OCRx Counter** |
| Move Forward (‘f’) | Left Wheel Forward (OCR0B)  Right Wheel Forward (OCR1A) |
| Move Backwards (‘b’) | Left Wheel Reverse (OCR0A)  Right Wheel Reverse (OCR1B) |
| Turn Right (‘r’) | Left Wheel Forward (OCR0B)  Right Wheel Reverse (OCR1B) |
| Turn Left (‘l’) | Left Wheel Reverse (OCR0A)  Right Wheel Forward (OCR1A) |

1. For each movement based on the command received from the RPI, the respective PWM timer counter(s) shown above will be initialised to a PWM value (Duty cycle) calculated based on the power specified by the operator. (Counters not shown are initialised to 0)
2. Once movement is completed, Arduino will send an OK packet back to RPI to indicate that the movement command has been successfully executed.

## 5.7. Bonus Movement Features

During our testing phase, we realised that there was a limitation with the speed at which the user commands are sent over to Alex (Human factor). In consideration of the search and rescue scenario where time is limited, our team added additional commands on top of what was given which we believe would improve the movement accuracy and efficiency of our vehicle.

|  |  |
| --- | --- |
| **Additional Commands** | **Movement** |
| ‘w’ | Forward (5 cm, 80% Power) |
| ‘a’ | Turn Left (30 Degrees, 80% Power) |
| ‘s’ | Backwards (5 cm, 80% Power) |
| ‘d’ | Turn Right (30 Degrees, 80% Power) |

In a time constraint situation, we do not have the luxury to specify the distance and power of each movement as it would be too time-consuming and more often than not, we observe that the power that was used is generally the same. Hence, the new commands have all the distance and power standardised as shown in the table. This enables Alex to **move/turn in small but very quick steps** in tandem with our Lidar mappings and the WASD would also imitate what we see in racing video games thereby deriving a sense of familiarity to the operator.

# 6. Software Design

## 6.1. RPLidar Data Handling

### 6.1.1. Web Server and Front-end Browser-based Client to Display Data

We decided that a method like Gnuplot should be used for mapping, as it is sufficient for mapping during our experiments, and it is very simple and efficient. However, we would need a few more features, namely a scale on the map so that we know the length of a wall, or a rectangle representing Alex for us to avoid wall bumps. While it is possible to extend Gnuplot for this purpose, we decided to follow an approach that we are more familiar with: using a web server and rendering the map with HTML.

We used the express framework [8] for this purpose, and the server would expose two endpoints:

* /data: the endpoint to retrieve data. Alex will send data to this endpoint by a POST request. The front-end will subscribe to updates by sending a GET request; this connection would be kept alive with server-sent events [6].
* /: the endpoint that will serve the HTML front-end. In this HTML file, we use a <canvas> element [7], of size 600 × 600 (pixels). On there we draw thin grid lines such that each grid cell equates to 10 × 10 (cm) in the maze. These grids would turn out to be not only helpful in measuring wall length, but also any non-square angles in the grid. We also draw a rectangle at the centre - where Alex is - to represent Alex; this helped us avoid many collisions, and also helped us park Alex successfully.

### 6.1.2. RPLidar Code Running on Raspberry Pi

On Alex, we decided to use the code from the Lidar studio, with reference to the RPLidar SDK example code [4], but with some significant modifications for our need:

* We let the RPLidar take readings repeatedly instead of only once. Each time, we would also retrieve its health, and stop reading when Lidar is not healthy, since we consider the reading in such cases to be unreliable.
* We modified the calculation formulas in the Lidar studio, such that on the front end, the front direction is upwards, and left/right directions are correct. We also modified the coefficients of these numbers, such that 10cm in the maze corresponds to 1cm on the front-end screen.
* We also take into account the “brightness” variable in the Lidar studio code (and change the opacity of points on the front-end according to this brightness value).
* Instead of printing the reading results to a file, we store them in a std::vector. After a whole reading round is completed, that vector is converted to a string in a format the server can understand, and then the string is sent to the server with libcurl [5]. The vector is then cleared and RPLidar will repeat the loop again.

After implementing these, we found that the lag between movement instructions and the new map reflected on the screen is approximately one second. Since that also includes the time Alex takes to move, we consider that to be fast enough. In the final run, it did turn out that the one-second lag was negligible.

# 7. Lessons Learnt & Conclusion

## 7.1. Two Key Takeaways from CG2111A Project

1. We learned **not to take standard libraries for granted**. Having learned the implementation and uses of bare-metal code, we have a newfound appreciation for the convenience that open-source libraries have brought us. Back in EPP1, while we used serial communication, timer, and PWM to power our vehicle, it was all implemented on the bare-metal level by Arduino standard libraries. Hence, we did not fully understand the underlying operation for these commands (or simply put, we have been using them blindly). As computer engineers, the key takeaway from this course is always to understand what runs beneath the technology/library that we are using. This is especially important for us when we are handling/debugging both the hardware and software aspects of our vehicle. Furthermore, knowing how to handle bare-metal code would enable us to optimise our system and exert more control over its operations (i.e. power-saving features, minimising overheads)
2. **Things will go wrong no matter how many times you test it**. Unlike software which will work as intended as long as it is debugged and tested correctly, the hardware is not as merciful. Despite repeated testing of our LIDAR and navigation system, connectivity and unexpected problems still arise every now and then, no matter how prepared we think we are. While this is unavoidable even in well-funded and managed major infrastructures, another takeaway from this course is learning how to handle unexpected errors. Most importantly, determining its root cause and preventing/mitigating its impact while keeping its core functions unhindered.

## 7.2. Two Greatest Mistakes Made

1. A great mistake that we made, which almost turned our final run to a disaster, was to not test Alex properly in an environment most similar to the final run. During our testing process, we always used the NUS wifi for connection between our devices and Alex. However, not willing to lose a device just to get Alex’s IP address during the final run, we decided to go with our mobile hotspot, and we only had 1 hour before the final run to test Alex on that hotspot. The inevitable happened: we irreparably lost connection to Alex during the whole of the final run. Thankfully, the teaching team granted us the chance for a second run, where we decided to use NUS wifi as we usually did. Because of that, we could complete the final run quite well, but we also understand that in an actual emergency case scenario, such failure may be a matter of life and death, hence this mistake is serious and unacceptable.
2. The second greatest mistake made was not realising that hardware faults can also be detrimental to the robot and caused it to fail. This is because we spent weeks trying to correct Alex’s turning movements as it was not turning at the desired angle we commanded it to. Most of the time was spent trying to debug the software movement codes which still did not manage to solve the issue. This resulted in Alex being stuck at the starting point during our Trial Demo for around 4 minutes, as we are unable to angle Alex to point towards the entrance and exit the starting point. Hence, we were unable to completely map the environment. After consulting Prof Ravi about the issue, we managed to find the fault, which was due to the motor driver chip and also a misconfiguration in the PWM setting (Fast PWM -> PWM Phase Corrected). This made us realise that hardware debugging is equally as important as software debugging as hardware faults can lead to undesirable behaviour of Alex, and they are hard to be spotted and rectified. If we had tried to debug the hardware of Alex earlier, we would be able to rectify the issue much quicker and spend the rest of the time working on other parts of the project.

# References

[1] Micire M., *Evolution and Field Performance of a Rescue Robot*, <https://doi.org/10.1002/rob.20218>

[2] Meet Doshi, *Boston Dynamics – Use Cases, SWOT Analysis & Future Scope*, [https://datamahadev.com/boston-dynamics-use-cases-swot-analysis-future-scope](https://datamahadev.com/boston-dynamics-use-cases-swot-analysis-future-scope/)

[3] *How Ford is using a Boston Dynamics robot in one of its plants*, [https://www.cnet.com/videos/how-ford-is-using-a-boston-dynamics-robot-in-one-of-its-plants](https://www.cnet.com/videos/how-ford-is-using-a-boston-dynamics-robot-in-one-of-its-plants/)

[4] *Open source SDK for Slamtec RPLIDAR series products*, <https://github.com/Slamtec/rplidar_sdk>

[5] *libcurl - the multiprotocol file transfer library*, <https://curl.se/libcurl>

[6] *Server-sent events*,MDN Web API Documentation, <https://developer.mozilla.org/en-US/docs/Web/API/Server-sent_events>

[7] *<canvas>: The graphics canvas element*, MDN HTML Documentation, <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/canvas>

[8] *Express - Node.js web application framework*, <https://expressjs.com>