CG2111A Engineering Principles and Practice

Semester 2 2023/2024

"Alex to the Rescue"

Final Report

Team: B01-6A

Name	Student #	Main Role
Jonathan Ong Joe	A0283070L	Hardware
Quan Teng Wai	A0281416H	Software
Piraveen Alexander	A0273022U	Software
Chan Zun Mun Terence	A0282142M	Hardware

Table of Contents

1. Introduction

Our Alex Robot is designed to emulate the functionalities of a search and rescue robotic vehicle. It would be tele-operated to move in a simulated environment.

Our Alex would be used in a room with various objects, such as tables, chairs and boxes. We would pilot Alex to find at least 2 victims in a 6-minute window. These victims are green (healthy but trapped), or red (injured and trapped).

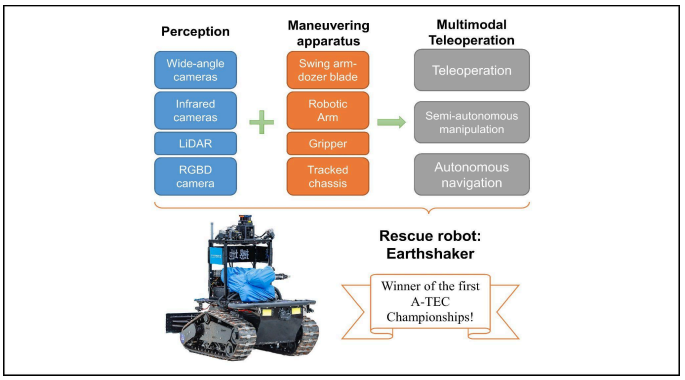
Alex will include a Raspberry Pi and an Arduino Mega. The Raspberry Pi has a Master Control Program to communicate with Arduino via UART. The Arduino is then able to control the motors and return information from the ultrasonic sensor and colour sensor to the Raspberry Pi, for Alex to gather data and navigate through the Course. At the same time, the Raspberry Pi receives data from the LIDAR to map out the environment. A Laptop is used to communicate with the Raspberry Pi via TLS for controlling movement & sensors, SSH for debugging and ROS Networking for retrieving LIDAR data to conduct SLAM mapping.

Finally, our robot would also have one of our teammate's face on it, since he has the same name as our robot, Alex.

2. Review of State of the Art

There are many different designs in the modern search and rescue robotic arsenal. Here are 2 that interested us.

1. *Earthshaker*, winner of the first Advanced Technology and Engineering Challenge (A-TEC) championships.



Teleoperation was achieved with 3 different communication methods between the Operator PC and the On-board computer (NUC) using a 1.8G MIMO mesh radio, 2xAT9S, STM32 Control board and a 4G/5G router. Software enabled automated switching between communication modes depending on connection quality/ range.

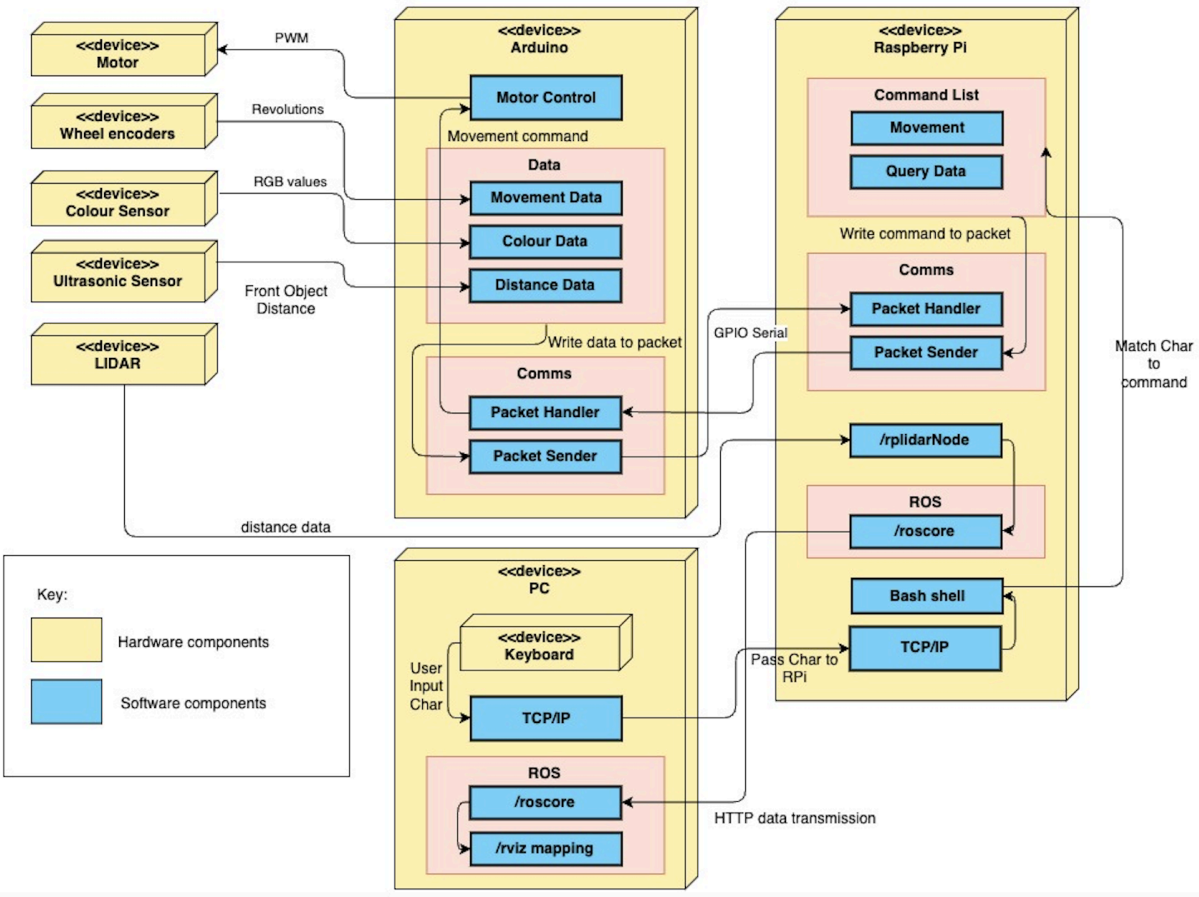
Strengths	**Weaknesses**
• Chassis and components minimally IP64 water resistant, able to handle wet environments • Dozer Blade allows it to push obstacles under 75kg away • Multiple communication methods, from 2.4Ghz direct communication to 1.8 GHz MIMO-mesh radios to suit a different ranges and conditions	• Excessive size limited flexibility of movement • Lacking in gripping power for modified robot arm, unable to manipulate heavier loads. • Battery life of 3h • Cumbersome body slower at navigating stairs

2. *Vapor 55 Drone*, used in spotting sharks, dropping life rafts and medical equipment in Australia, originally designed to deliver blood and supplies to wounded soldiers.

Hardware:
Trillium HD-25 Optical sensor (Camera)
Lidar/ PPK Mapping Options
Software:
Advanced Flight Control System (FCS)
[inclusive of autopilot]
Automatic Autorotation, smart auto-home functions

Strengths	**Weaknesses**
• Modular radio options (Common Radio Interface connector) • Flexible Payload - rail design enables many different kinds of payloads to be equipped • Aerial - Unrestricted by Terrain (Rubble/ Ocean)	• Battery life of 1h • Environmental Operational Limits from -17C to 49C, unable to be used in extreme temperatures • Comparatively low Usable payload of 4.5kg compared to ground options

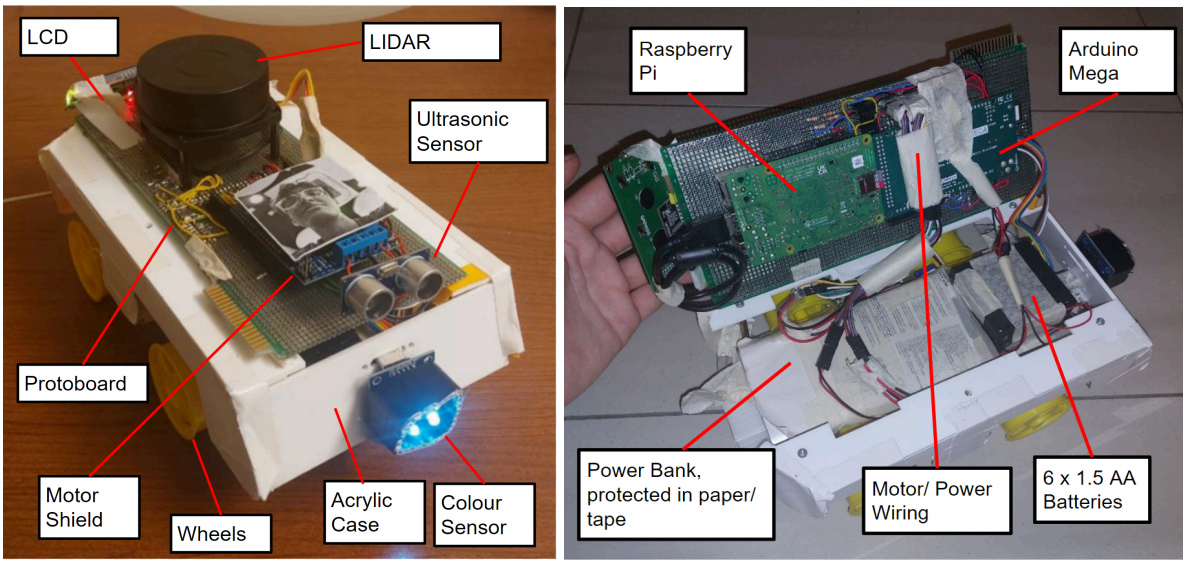
3. System Architecture



4. Hardware Design

4.1. Final Form of the System

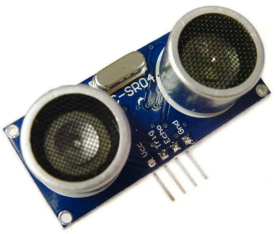
The robot is built with a given chassis and powered by 4 motors. There is a protoboard to connect most of the electronic components together as well as an Acrylic casing to protect the robot and mount the colour sensor and protoboard.



Images of Robot & its internals

4.2. Non-Standard Hardware Components

Ultrasonic Sensor

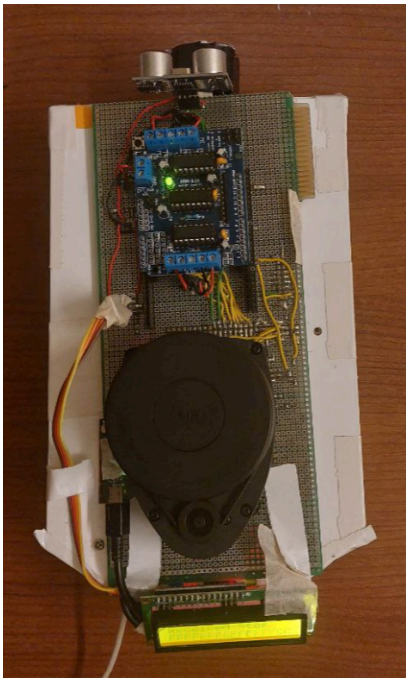


An ultrasonic sensor was used to move the robot (and hence the colour sensor) closer to the target without hitting it. This would allow for more accurate colour detection. More information is in Section 5.4

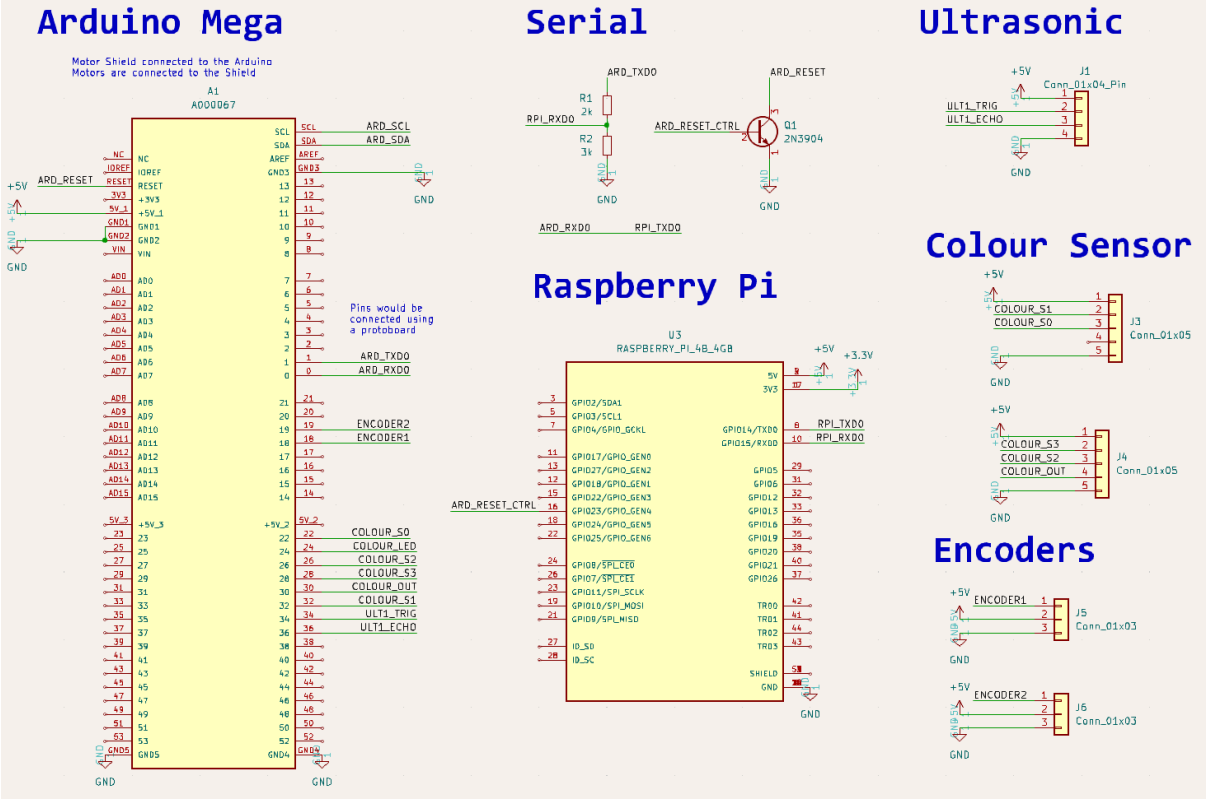
I2C 16x2 Character LCD

A 16x2 Character LCD was used to display text and various other debugging information. This was a useful alternative way of troubleshooting potential issues on the Arduino quickly

4.3. Protoboard



1. The LIDAR would be connected to the Raspberry Pi via USB and is secured onto the protoboard using screws
2. The Motor Shield connects to the protoboard through female headers. The screw terminals on the protoboard are connected to solid core wires, which are then plugged into the protoboard's female headers (if need be, the wires are soldered to male headers)
3. The Ultrasonic Sensor plugs into the top of the protoboard
4. The Arduino and RPi plug into the bottom of the protoboard.
5. The Motors, Encoders, Colour Sensor, LCD and 6 x 1.5 AA batteries (which power the motor) plug into the female headers on the protoboard
6. The Raspberry Pi is powered by a USB Power Bank



With our protoboard, we could mount the components in a modular and neat manner. Our motors, power source, Raspberry Pi, Arduino and Ultrasonic Sensors could be easily attached and detached. Furthermore, we could have our wiring secured and soldered down, which made the robot's connections more secure.

Another benefit was the ability to access the Arduino Pins which were blocked by the motor shield. This allowed us to easily implement GPIO Serial Communication between the

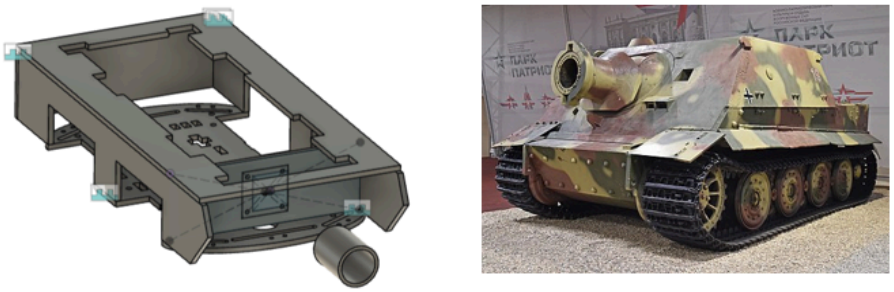
Raspberry Pi and the Arduino, as well as Arduino Reset & Programming through the Raspberry Pi. More of this will be discussed in Section 6.4.

We have also included other connectors, such as a connector for an MPU6050 IMU, as well as various other connectors for other ultrasonic sensors. However, we did not utilise those ports in our mock/final run.

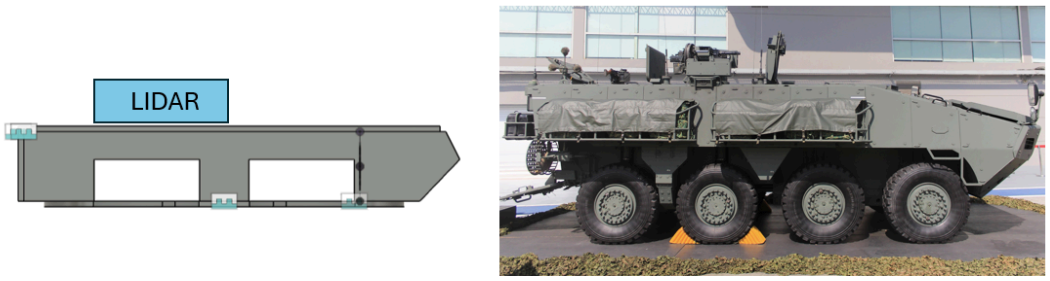
4.4. Acrylic Casing

Our Acrylic Casing was laser cut using the Equipment from Makers@SoC.

We first modelled casing for the Alex using Fusion360. The model was based off the Terrex ICV, and the Sturmtiger tank, with initial plans to repurpose the barrel to fit the colour sensor. The barrel was shifted downwards closer towards the main body of Alex so as to prevent the colour sensing module from obscuring the mapping capabilities of the LIDAR system.



Alex Main Structure CAD model and SturmTiger Replica[1]



Side-by-side comparison of Alex and Terrex ICV side profile[2]

The benefits of an acrylic case allowed us to fit the components of the robot in a more structured environment, and keep the wires compact and neat. It also improved the stability of Alex, helping to secure loose weight such as the LIDAR mounted on the protoboard.

Further modifications such as reduction of overall length and trimming of the side profile guards were then made to reduce weight and increase the manoeuvrability and speed of Alex, allowing it to fit through tighter corners and turn more easily.

[1] "Sturmtiger '19 red'" - by Alan Wilson is licensed under CC BY-SA 2.0.
[2] "Terrex Infantry Carrier Vehicle side profile AOH 2022" - by Seloloving is licensed under CC BY-SA 4.0.

The LIDAR system was also adjusted more towards the rear wheels for better weight distribution and to create space for the motor driver and ultrasonic sensor modules. It was then secured by drilling holes into the protoboard and securing using the M3 button-head screws available within the DSA lab.

5. Firmware Design

5.1. High-level algorithm on the Arduino Mega

1. Setup
    a. Set up UART at a baud rate of 9600, 8N1 Frame format.
    b. Set up the motors, the ultrasonic sensor, and the colour sensor by setting the pin modes of the relevant pins.
    c. Set up the I2C line and initialise the LCD
2. Arduino polls for serial data
    a. It reads the Serial data & stores it in a buffer
    b. The data is then deserialized into a TPacket packet and TResult result
    c. It checks TResult result for the status of the packet
        i. If the Packet received is OK,
            1. it handles it (such as in Further Breakdown Step 4a),
            2. It sends an OK packet to the Pi, or a packet with the required information depending on the command.
        ii. Else, it sends an error back to the Pi, be it an unknown error, bad magic number, or bad checksum

5.2. Communication Protocol

We set up UART communications with 9600bps, 8N1 format.

We receive command packets through polling. Data is then serialised and deserialized from a struct format “TPacket” as shown in Code Snippet 5.2.1 (Appendix C).

There are a few packet types, response types and commands initialised as shown in Code Snippet 5.2.2 (Appendix C). COMMAND_GET_STATS utilises a data array (as shown below) to generate a status packet. The red, blue and green frequencies of the colour sensor, detected colour from the Arduino, along with the ultrasonic sensor's pulse duration are sent in the packet. (Appendix C). All other commands do not utilise such an implementation of data arrays.

In the Arduino, we set the appropriate values in UBRR0H, UBRR0L (for baud rate), UCSR0A, UCSR0C (to configure Async UART), and UCSR0B (to start UART) on setup. When the Arduino sends data to respond to the command packet, it writes the data to a buffer. The Arduino then polls the UDRE0 bit in UCSR0A for the transmit buffer to be empty, before writing a byte from the buffer to UDR0 to send 1 byte at a time to the Raspberry Pi.

5.3. Timer Based Movement

We utilised the timers to manoeuvre the robot for a specified duration. This is useful for precision control of the robot if we need it to be aligned to the robot. Our algorithm is as such

1. On movement, enable the timer and set the relevant variables
    a. We have a counter variable and the target value.
2. The Timer counts every millisecond and increments a variable
    a. Timer 5 with a Prescalar of 1 was used
3. In the main loop, the Arduino polls for when the counter reaches the target value.
    a. If the target value is reached, the Arduino stops the timer and stops the motors.
4. If the Arduino receives another movement command, the timers would be reset, and the motors would be stopped/ set accordingly

The bare-metal code used is provided in Appendix C. We decided on milliseconds as the unit of time counted to allow for control accuracy. Using timers instead of blocking delays (eg. `delay(1000)`) allows us to quickly stop the movement of the robot.

We can switch between continuous, timer-based and encoder-based movement on the fly through the client program. Depending on which type of movement is desired, the corresponding type of packet and value is sent. This gives us more control of the robot during our run. However, as timer-based movement turned out to be more accurate than encoder-based movement, we defaulted to timer-based for our testing and during our mock/final runs.

5.4. Ultrasonic-Based Movement

Reading Distance from the Ultrasonic Sensor

1. Set the trigger pin to high for 10 microseconds to send the pulse by setting the `PORTC` register accordingly.
2. Reads the Echo Pin and gets the length of the pulse on the echo pin
3. Convert the period to the distance between the ultrasonic sensor and the wall
    a. Distance = (*Speed of Sound* × *Time take*n) / 2
4. Sends a packet with the length of the pulse to the RPi
    a. We use this value instead of distance to prevent any potential data loss from the conversion of value
5. The Distance Value is used for movement

Utilising the Ultrasonic Sensor for Movement

1. Move Forward for a specified distance/ timing
2. Poll the Ultrasonic Sensor for the current distance
    a. If the distance is less than a specified value, stop
3. If the Arduino has finished moving that specified amount of encoder distance/ timing, stop

An issue we faced was that reading from the ultrasonic sensor takes a significant amount of time, which may cause the Arduino to be unavailable to process Serial Packets. This would mean that we cannot stop the Arduino movement when we tell it to move forward

As such, the ultrasonic sensor is only used when the Arduino is told to do so. For WASD control and other forms of movement, we would not use the ultrasonic sensor, as we want to

maintain control over the robot. In the worst-case scenario, we would force reset the Arduino to forcibly stop the movement of the robot.

5.5. Colour Sensor

Reading Values from the Colour Sensor

Our Colour Sensing Algorithm is as Such

1. Set the Trigger pin to HIGH to trigger Colour Sensing
2. Colour Sensor Module to cycle between RGB lighting
    a. Individually toggle each of the colour LED Pins to HIGH, delay, then LOW.
    b. Read the respective frequencies from the colour sensor during each colour phase
3. With reference to the ROYGBIV table that has the respective range of frequencies for each colour, determine the colour of the object in front of the Color Sensor.
4. Relay information to the RPi, which is then sent to the laptop.

Detecting Colours from the Frequencies

We utilised a simple range-based colour detection algorithm on the Arduino to detect the colour. It checks if the frequencies are within certain ranges (based on empirical testing), and identifies the colours accordingly. As we wanted to minimise data loss from conversion (eg. division and conversion to double type cause loss in precision), we compared the frequencies directly. We also sent the raw colour frequencies through the packet to the Raspberry Pi, and by extension to the Laptop. This allows the operator to manually read the values and identify the colour if need be. However, from our testing and calibration, we figured that our simple range-based colour detection algorithm suffices.

5.6. LCD Display

The main purpose of our LCD Display would be to handle debugging and various other issues. Every time a command is sent from the Raspberry Pi to the Arduino, the LCD displays the command sent along with other information.

The LCD would also reset and the backlight would blink whenever the Arduino resets. This acted as a quick way for us to check if the Arduino and Raspberry Pi were still connected, or if the Arduino was functioning as intended.

6. Software Design

6.1. High-level algorithm on the Raspberry Pi

Initial Setup

1. On the Laptop
    a. Install an Ubuntu VM with bridged networking and set up ROS networking.
        i. A VM was used to allow for snapshots, hence reducing the potential errors in system configuration
    b. Download and Compile the TLS Client Code
2. On the Raspberry Pi
    a. Download and Setup PurplePet OS

b. Download/ Compile the TLS Server Code/ Arduino Code

Raspberry Pi

1. Runs the LIDAR ROS node
2. TLS Client Setup
    a. Opens and sets up a Serial Connection to the Arduino Mega
    b. Opens a TLS server and binds to a port, which the Laptop authenticates and connects to
3. Receiver Thread
    a. Laptop sends commands to Pi through the TLS Client
    b. Pi handles packets and sends commands to Arduino
4. Sender Thread
    a. Receives Serial Data from the Arduino
    b. Handles the Packets and sends relevant messages to the TLS Client. These include
        i. Message Packets
        ii. Error Messages from the TLS Client (eg. Bad Magic Number)
            1. (modification to tls-alex-server was needed)
        iii. Error Messages from the Arduino

Laptop

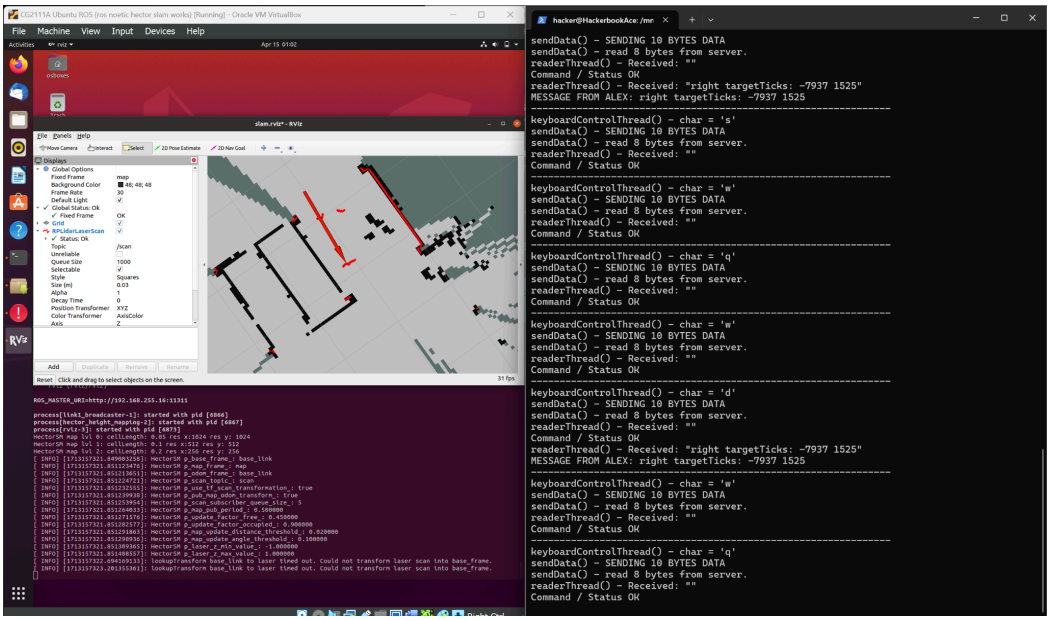
1. TLS Client
    a. Connects to the TLS Server
    b. Receives keystrokes and sends the appropriate data to the TLS Server. Some Actions Include:
        i. Continuous WASD Movement,
        ii. Stop the robot,
        iii. Retrieving the Status of the robot,
        iv. Force kill/ unkill the robot
        v. Clear the Serialize Buffer on the TLS Server
    c. For certain special keystrokes, the client would open a prompt to fill in more details. The relevant commands are
        i. Change the Speed, Move the robot for a specified speed/ distance
2. Runs ROS Networking, HectorSLAM and start RViz Mapping

6.2. WASD Control

We have implemented a rudimentary form of WASD control of the robot. We can press down the key, and the robot will move in the said direction until the robot is told to stop. WASD control is implemented by setting the terminal to raw mode with no buffering. The program reads in character by character, instead of needing to type in one line at a time. This allows us to move the robot continuously and map, saving time, instead of specifying the distance and speed of the robot multiple times.

We have added speed control and encoder distance/time/angle control. For Angle control, we calculate the approximate amount of time the robot needs to turn left/right based on its current speed category. For example, if it is at 50% speed, it would take 20 seconds to turn 360 degrees, and we would only need to turn the robot for 10 seconds to turn 180 degrees.

This makes it easier for the operator to manoeuvre the robot in tight spaces.



RViz & TLS Client View.

However, sending inputs too fast results in commands being registered incorrectly due to the processing time/ latency on the Arduino. As such, we have implemented a simple 500-millisecond delay between each keystroke. If by any chance, the commands are still registered incorrectly, we can reset the Arduino through the client and clear the serialize buffer, hence resetting the serial connection quickly.

6.3. Remote-Controlled Programming of the Arduino

Soft Kill Switch

The reset pin of the Arduino can be controlled by the Raspberry Pi directly. As such, we can force reset the Arduino through the Raspberry Pi, and force the Arduino to stop the motors. This is useful in the event of any malfunction during the run (eg. when the Arduino receives Bad Magic Number). Typically, we would need to re-upload code to the robot using the Arduino IDE. Considering the time taken to compile and upload the Arduino Code, it would take about 10 seconds, if not more. With this functionality, we would only need 5 seconds to reset the Arduino and get the robot up and running. Furthermore, if need be, we can keep the robot held at reset for a prolonged period of time. This forces the motors and the robot to stop, allowing us to investigate any firmware issues.

We also modified the TLS Server and Client Programs to allow this reset functionality to be done remotely, through the client itself. When done through the Client program, the TLS Server would automatically clear the serialise buffer, to allow the TLS Server to parse data from the Arduino accordingly. Modifications to serialize.cpp were done to allow for this functionality.

Command Line Upload Script

We were able to use a simple command line script to upload our new Arduino code to the Arduino. Without the need to open up the Arduino IDE in the GUI (through either SSH X11

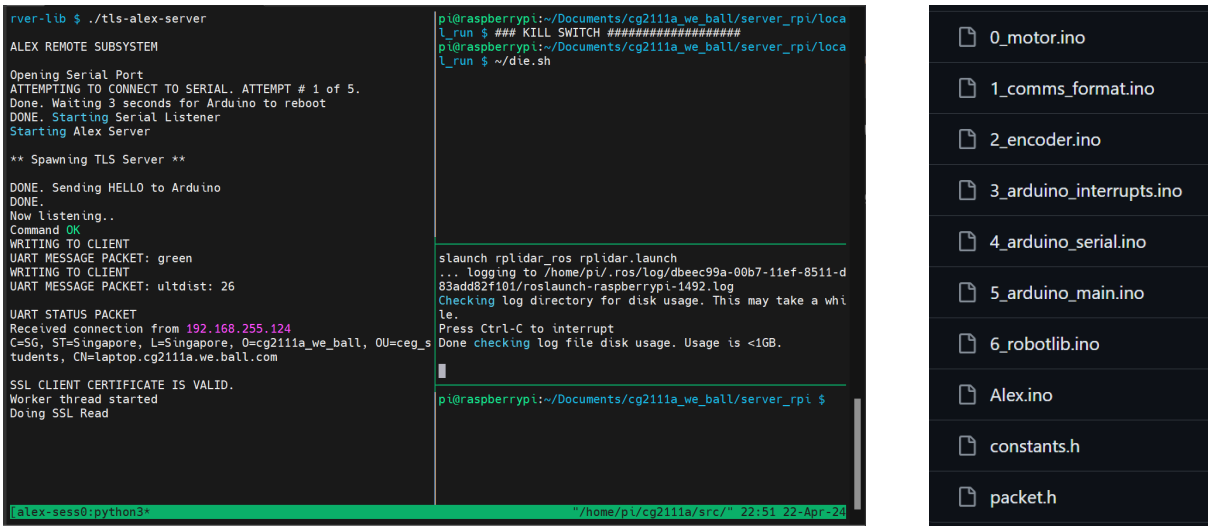
Forwarding or VNC), this allowed us to fine-tune our parameters quickly through the CLI, saving precious time during the run (if need be)

6.4. Others

tmux script

We created a bash script to automatically start the relevant processes in a tmux window. tmux was used as it allows for the tiling of various shell windows. The user can switch the various windows quickly and easily do any debugging as and when needed.

Furthermore, in the event the SSH connection is disconnected, the tmux session would just detach. The TLS Server, as well as the ROS node, would not be forcibly killed. Instead, we can SSH back into the Raspberry Pi and attach it back to the tmux session, reducing the time needed to set up all the required processes again.



Left: Sample tmux window, with terminals for TLS Server, ROS node, soft kill (for emergencies) and debugging. Right: Our code structure, with split files.

GitHub Collaboration

We used Github as a way to keep track of our version history as well as to collaborate in a team. In the event of any mistakes, we can revert to an older commit. Thankfully, we did not need to do this. Furthermore, we split up our code into separate files to ensure readability and ease of maintenance, such as our Arduino code and our TLS code.

Implementation of HectorSLAM on Laptop

To allow for near-real-time control of the movement of our Alex, we needed to reduce the latency of SLAM map refreshes, which was primarily hindered by the limited processing power of the RPi. Therefore, we opted to offload the generation of the SLAM map onto the client's laptop, while using the RPi to publish the LIDAR data onto a ROS topic (based on the instructions provided in the optional ROS Networking tutorial).

However, we found that the laptop (which was configured as a ROS Master) was unable to launch the packages needed to generate a SLAM map. After extensive troubleshooting, we

figured we needed to install the prerequisite ros-noetic-hector-slam package. We were then able to visualise a SLAM map which was updated in near-real-time.

7. Conclusion - Lessons and Mistakes

Lesson 1: Working within Time-Cost Limitations

We initially wanted to include a Printed Circuit Board, as that would lead to neater and more reliable electronics, as well as an Inertial Measurement Unit, for more accurate movement. However, in the case of the PCB, cost and implementation time concerns led us to utilise a protoboard sourced from the DSA lab. To save costs, the protoboard used was repurposed from one initially meant to be placed inside PCs, with a PCI connector. In the case of the IMU, we were unable to source it with sufficient time to implement its usage. Hence, we used other methods, particularly the Arduino's timers as well as the encoders to achieve smaller turns. This taught us that instead of solving problems through buying additional components, we can also make do with what we have to solve the problem more cost-effectively, through the in-built peripherals.

Lesson 2: Optimising Peripheral Usage

We initially struggled with Serial performance. We constantly received error messages from the Arduino and the Raspberry Pi Side. This error was resolved by reducing our dbprintf usage. Our theory is that the excessive usage of debugging print statements in a short period resulted in a non-negligible strain on the serial connection/ code, resulting in data corruption/loss. As such, we should be optimal in our usage of the various peripherals, to minimise bandwidth strain.

Mistake 1: Insufficient Preventive Measures

A lack of preventive measures contributed to our mis-navigation during the Actual Run. This resulted in lost points and time wasted to confirm our positioning and prevent any further contact with the barricades and walls. One solution would be to implement an additional remote kill switch, to allow a secondary person to soft-kill the robot through another SSH terminal. One more solution to this would be to configure RViz such that the direction of the front of the robot is clear, to avoid such direction mistakes. Either way, planning for measures to reduce mistakes and further improve ease of use would have helped us in our runs.

Mistake 2: Power Issues

On our various tests, our motors kept stalling. We eventually realised this was due to the batteries for our motors draining quickly. This caused the voltages supplied to the motors to drop quickly, which resulted in our motors having different behaviour as time passed. Our solution to this is to change the battery before the run. However, this was not an ideal solution, as the voltage of the battery can drop very quickly. In hindsight, we should have focused on securing a more stable power source for our motors. This could be done through capacitors, a buck/boost converter, or other measures.

References

1. Zhang Y, Li Y X, Zhang H F, et al.
   Earthshaker: A mobile rescue robot for emergencies and disasters through teleoperation and autonomous navigation.
   JUSTC, 2023, 53(1): 4 doi: 10.52396/JUSTC-2022-0066

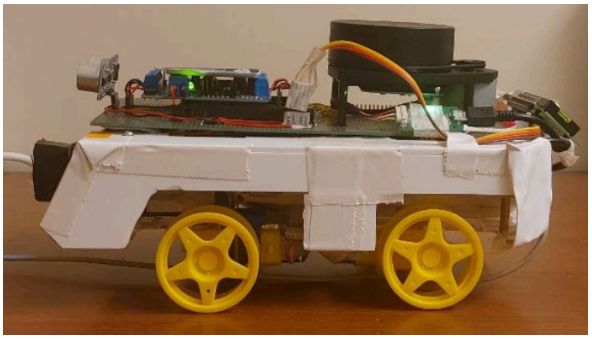
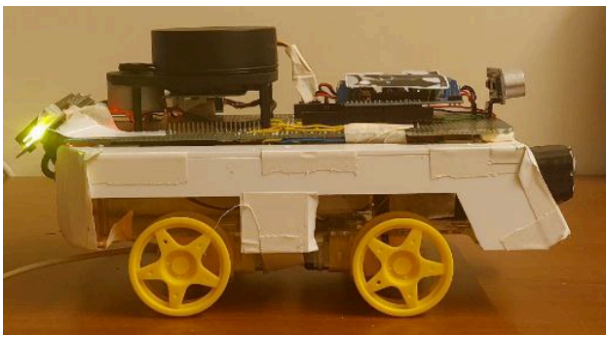
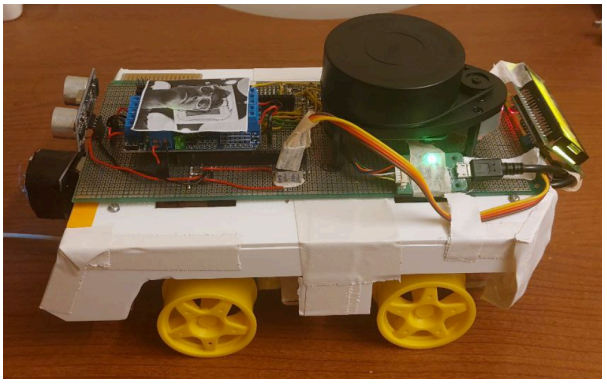
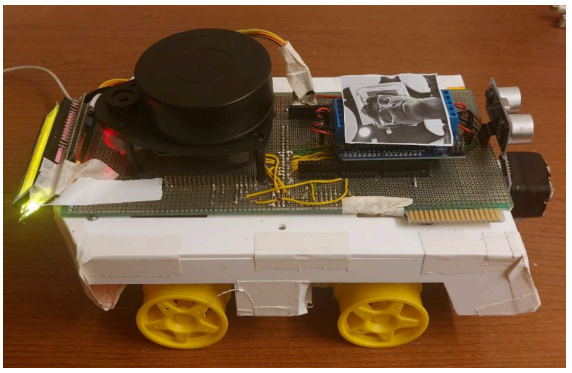
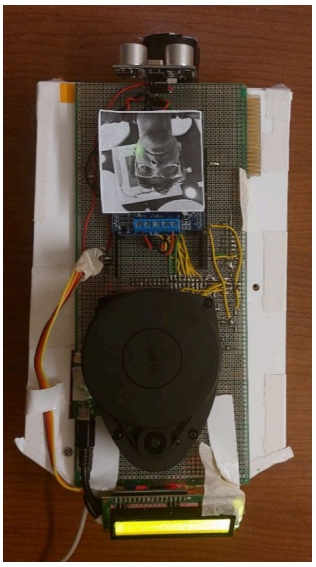
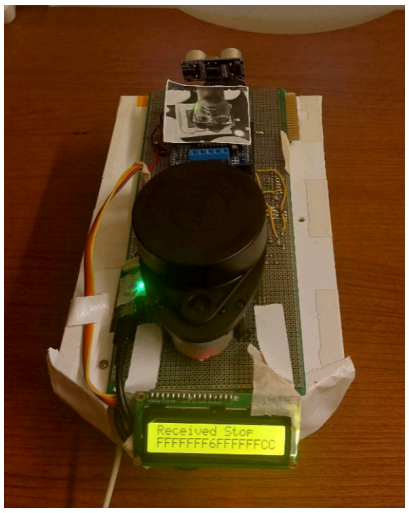
2. Avinc (n.d.) *Product catalog*. Retrieved March 28, 2024, from https://www.avinc.com/images/uploads/product_docs/ProductCatalog.pdf

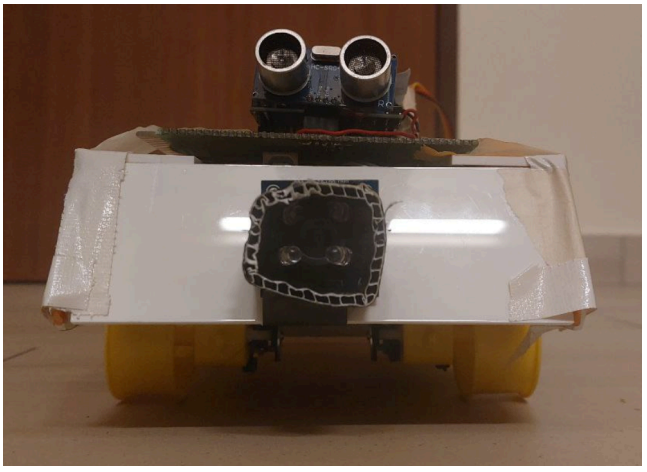
3. Avnic (August 30, 2019). *Vapor55 Datasheet*. Retrieved March 28, 2024, from, https://www.avinc.com/images/uploads/product_docs/VAPOR55_datasheet_08302019.pdf

4. Little Ripper (n.d.) *Our Amazingly Awesome & Proven UAV Fleet: Westpac Lifesaver Series.* Retrieved March 28, 2024, from, https://littleripper.com/our-fleet/

Appendix

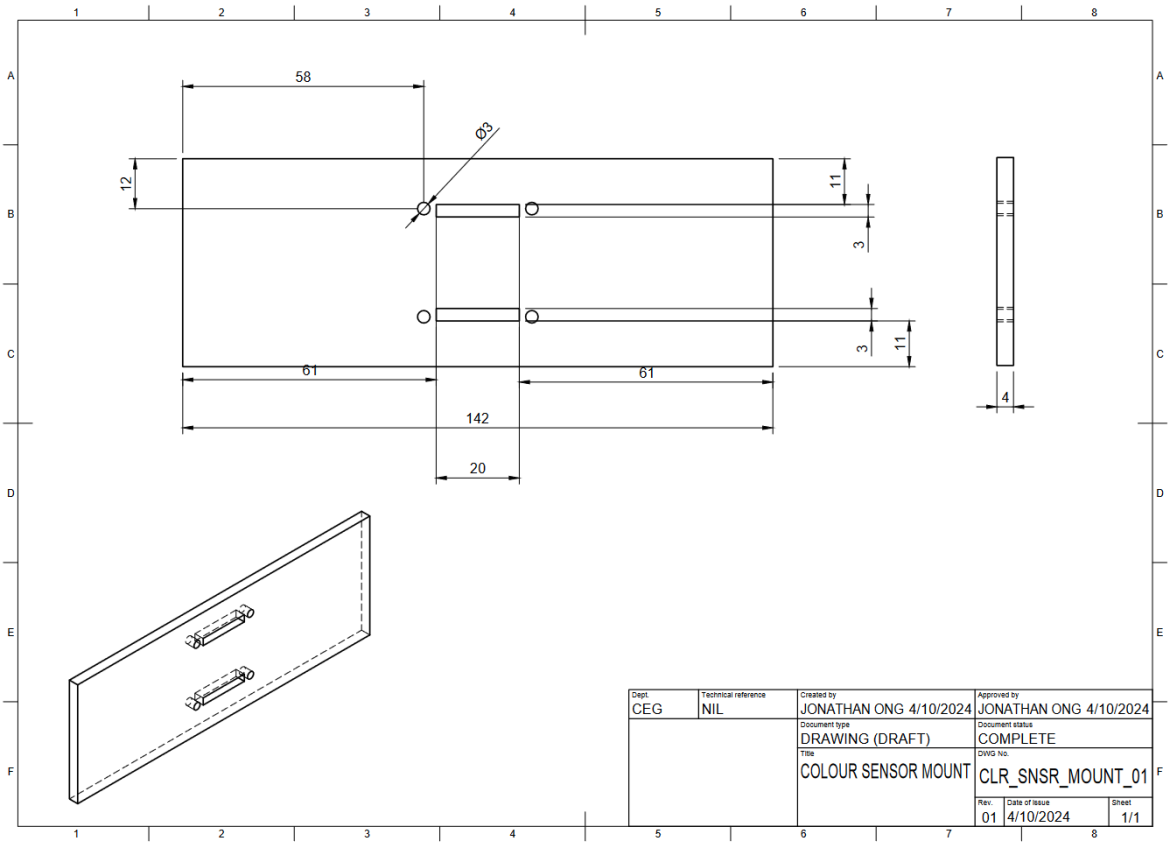
Appendix A

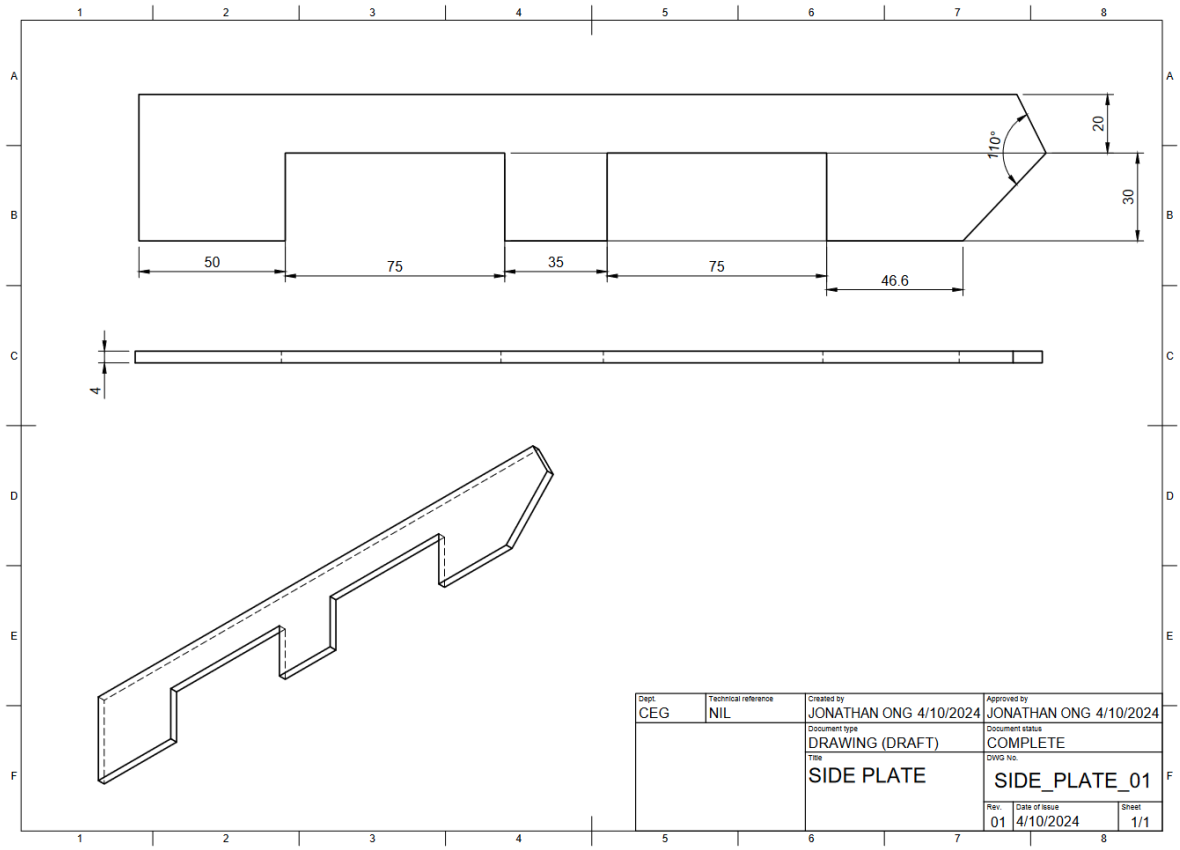


Appendix B

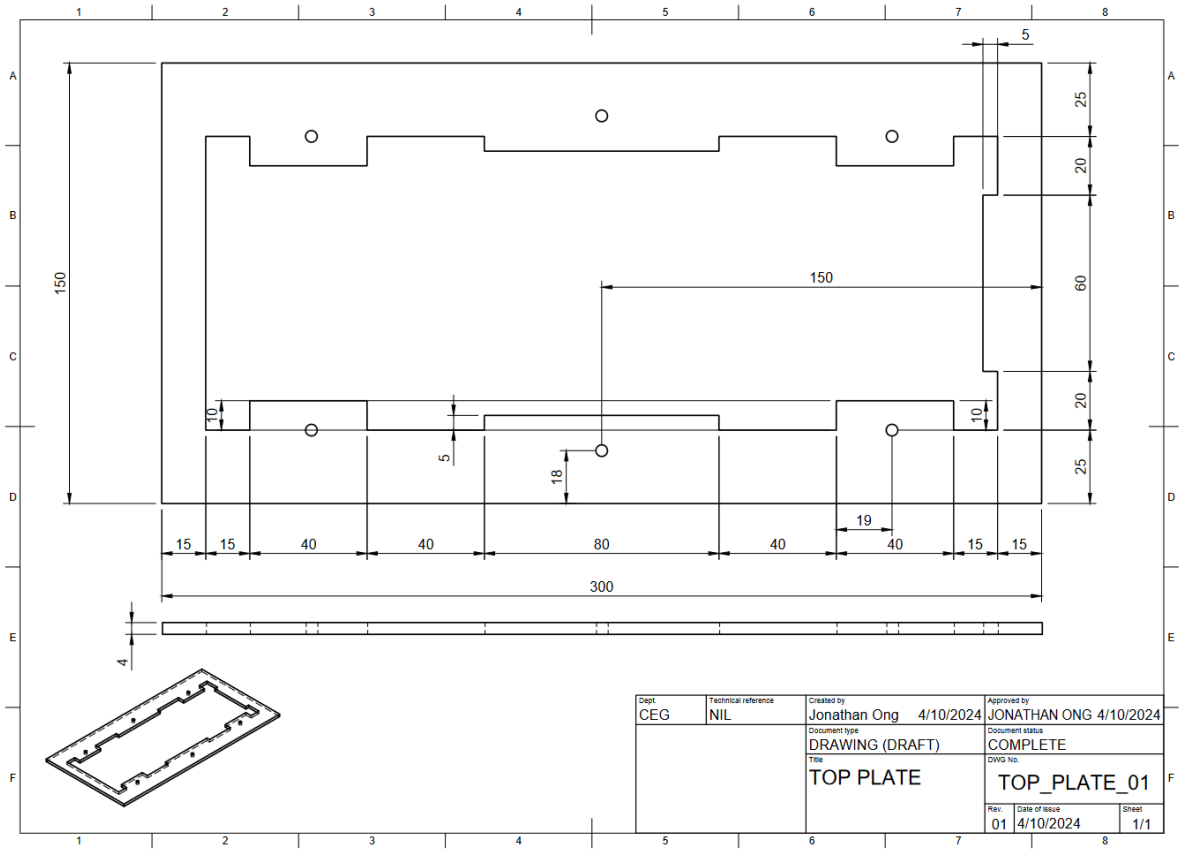
Acrylic Colour Sensor Mount



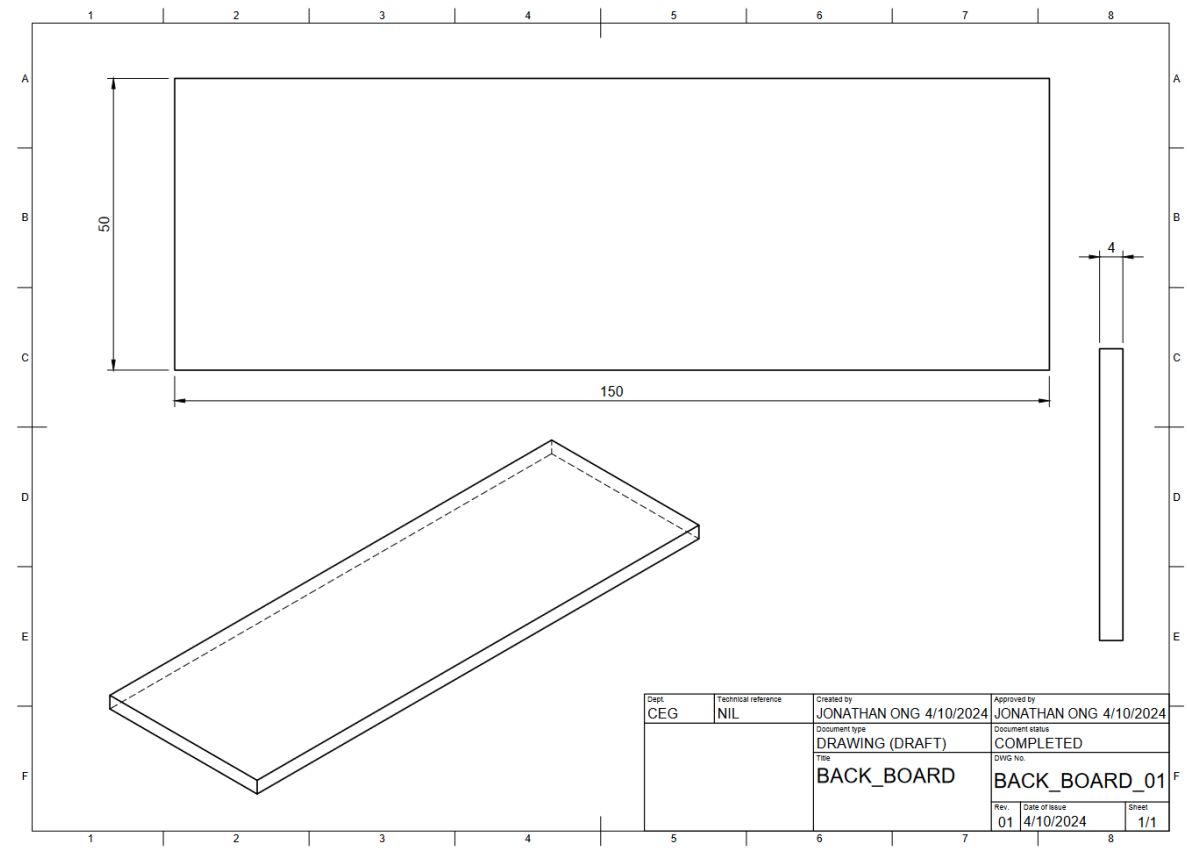
Acrylic Side Plate



Acrylic Top Plate



Acrylic Back Plate



Appendix C

Code Snippet 5.2.1: UART Communications Initialisation

```
#define MAX_STR_LEN   32
// This packet has 1 + 1 + 2 + 32 + 16 * 4 = 100 bytes
typedef struct
{

     char packetType;
     char command;
     char dummy[2]; // Padding to make up 4 bytes
     char data[MAX_STR_LEN]; // String data
     uint32_t params[16];
} TPacket;
```

Code Snippet 5.2.2: Packet, Response and Command Initialisation

```
// Packet types
typedef enum
{
  PACKET_TYPE_COMMAND = 0,
  PACKET_TYPE_RESPONSE = 1,
  PACKET_TYPE_ERROR = 2,
  PACKET_TYPE_MESSAGE = 3,
  PACKET_TYPE_HELLO = 4
} TPacketType;

// Response types. This goes into the command field
typedef enum
{
  RESP_OK = 0,
  RESP_STATUS=1,
  RESP_BAD_PACKET = 2,
  RESP_BAD_CHECKSUM = 3,
  RESP_BAD_COMMAND = 4,
  RESP_BAD_RESPONSE = 5
} TResponseType;

// Commands
// For direction commands, param[0] = distance in cm to move
// param[1] = speed
typedef enum
{
  COMMAND_FORWARD = 0,
  COMMAND_REVERSE = 1,
  COMMAND_TURN_LEFT = 2,
```

```
  COMMAND_TURN_RIGHT = 3,
  COMMAND_STOP = 4,
  COMMAND_GET_STATS = 5,
  COMMAND_CLEAR_STATS = 6
} TCommandType;
```

```
  TPacket statusPacket;
  statusPacket.packetType = PACKET_TYPE_RESPONSE;
  statusPacket.command = RESP_STATUS;
  statusPacket.params[0] = leftForwardTicks;
  statusPacket.params[1] = rightForwardTicks;
  statusPacket.params[2] = leftReverseTicks;
  statusPacket.params[3] = rightReverseTicks;
  statusPacket.params[4] = leftForwardTicksTurns;
  statusPacket.params[5] = rightForwardTicksTurns;
  statusPacket.params[6] = leftReverseTicksTurns;
  statusPacket.params[7] = rightReverseTicksTurns;
  statusPacket.params[8] = forwardDist;
  statusPacket.params[9] = reverseDist;
  // Stuff
  statusPacket.params[10] = redFrequency;
  statusPacket.params[11] = blueFrequency;
  statusPacket.params[12] = greenFrequency;
  statusPacket.params[13] = (uint32_t)ultDuration;
  statusPacket.params[14] = (uint32_t)colourEnum;
```

Code Snippet 5.3.1: Baremetal Code for Timer-based Movement

```
void movementTimerSetup(){
    // Set CTC (Clear Timer on Compare Match) mode
    TCCR5A = 0;
    TCCR5B = 0;
    TCCR5A &= ~(1 << WGM51 | 1 << WGM50);
    TCCR5B |= (1 << WGM52);

    // Set prescaler to 1 - Max accuracy
    TCCR5B |= (1 << CS50);

    // Enable global interrupts - done later in setup();
    //sei();
}
```

```
void movementTimerEnable(){
    TCNT5 = 0;
    OCR5A = 14974; //time_val; //15624 - 1 second;
    TIMSK5 |= (1 << OCIE5A);
}
void movementTimerDisable(){
    TIMSK5 &= ~(1 << OCIE5A);
}

volatile int32_t timeDiff = -1;
volatile int32_t milliseconds = 0;
ISR(TIMER5_COMPA_vect) {
  milliseconds++;
}
```

Code Snippet 5.3.2 Baremetal code for serial communication

Serial Write (Baremetal) -

```
void writeUART( const char *buffer, int len){
  for ( int i = 0 ; i < len ; i ++ ){
    while ( !(UCSR0A & (1<< UDRE0))){}
    UDR0 = buffer[i];
  }
}
```

Serial Setup/ Start

```
volatile char buffer[1024];
volatile int counts = 0 ;

void setupUART()
{
  // // Set Baud Rate to 9600
  UBRR0H = 0;
  UBRR0L = 103;
  UCSR0A = 0;

  UCSR0C = (1<< UCSZ01) | (1 << UCSZ00);
}

void startUART()
{
```

```
  UCSR0B = 0b10111000;
}
```