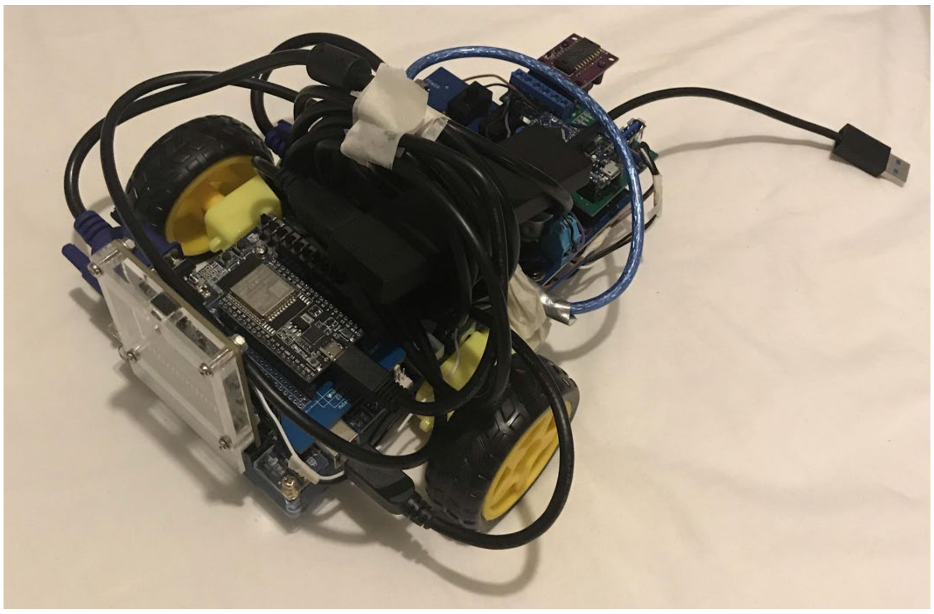
ELEC50008

Engineering Design Project 2

Mars Rover



Aryan Ghobadi, Brandon Cann, David Amor-Segan, Faris Al-Kayssi, Holly Solomon, Tobias Cook

ag519, CID: 01705556 | bc319, CID: 01724765 | da1018, CID: 01539078 |

fa3018, CID: 01498998 | hs2119, CID: 01724762 | tjc219, CID: 01718771

Department of Electronic and Electrical Engineering

Imperial College London

Contents

Abstract:

This report was written with the purpose of presenting the reader with a layered, technical explanation of the design, implementation and testing process applied to develop a fully remotely controlled rover-based working with an Arduino, FPGA, and microcontroller architecture. The principles of such a design could theoretically be transferred to a rover for use on Mars, or in similar environments. It is written in such a way that enables the reader to have a formative and functional understanding of how the team went from the initial idea to a specification and then on to the final design and deliverables, and the logistical organisation that was required as a team to reach that point.

The following report aims to communicate an organic product development process, from the motivation and context behind the rover's development, outlining the team's goals, developing hypothetical solutions and showing its procedural development and final implementation, alongside the experimental findings that are needed to reach that point. It further documents the logistical and communication-based commitments the team made throughout the course of the project. Concluding, it will contain the final reflections and conclusions from the final product and the lessons learnt by the team, and how they could refine their approach to development if they were to develop a newer iteration of the rover.

Introduction/Contextual Background:

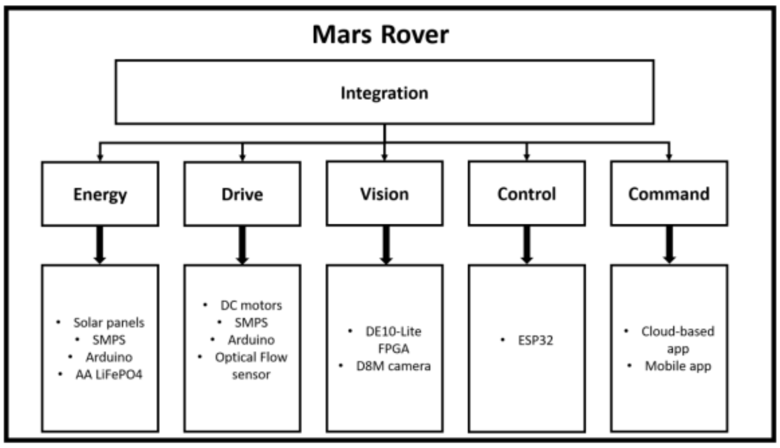
For decades, rovers have been an integral part to understanding and providing a human insight to surrounding areas without mandating a human presence. Most notably used during lunar missions, and recently applied on Mars missions, as well as on other planets, the rover delivers an autonomous level of perspective, whilst also maintaining a compact design. Many rovers possess a versatile range of capabilities in the modern era, being able to utilise computer vision to recognise aspects of the terrain and obstacles, as well as take samples of surrounding material. The rover must perform all these features in an intense and hostile operational environment, whilst also regularly and uniformly relaying status updates and information to a human team. It is this real-time interfacing and regular data-exchange which illustrates the value that rovers have in exploring what would normally be an unassailable environment.

Objectives:

This project aims to design an autonomous rover system which would be capable of mirroring the functionality of industrial systems. The team seeks to produce a system that would be able to operate remotely without human intervention; utilising computer vision in such a way that it is able to recognise obstacles and its immediate surroundings in its local area of operation, and procedurally building a map of the area. The rover will be powered with photovoltaic panels, and the batteries managed by a computer. Whilst in operation, it must also regularly send status updates and associated data to a website, and process and execute remote commands in real-time.

The rover system must be designed and assembled as a series of individual sub-systems, whereupon each sub-system plays a crucial role in the operation of the rover. Defined as: Energy, Drive, Vision, Control, Command, and Integration. To give a visual overview of how these sub-systems will operate alongside each other, below is a block diagram from the project specification illustrating this:

Figure 1: Block Diagram of the Project from the Briefing [1]



The sub-systems individually have their own scope of objectives, detailed below:

- Energy: This subsystem will provide the rover with charged batteries using solar panels, entailing the design of charger systems, status reading, and safety implementation, among other related features.
- Drive: This subsystem provides the movement functionality to the rover, covering speed and direction control, turning and distance measurement. The subsystem also tracks the rover's movement.
- Vision: This subsystem provides allows the rover to detect and avoid obstacles in the surroundings and map the local area by recognising objects of interest

- Control: This subsystem liaises with the other subsystems of the rover using the ESP32 microcontroller, allowing full-duplex communication, processing commands from the command subsystem and relaying status back, as well as providing real-time responses to the motors.
- Command: This subsystem provides a front-end interface to the overall system, by creating a web app which controls the rover remotely and facilitates communication with the Control subsystem.
- Integration: This specific task is undertaken to combine, install and assemble the overall rover as well as testing the key functionality in a remote manner.

Initial Specification and Design

This section is concerned with the design and thought process behind the design and planned implementation of the subsystems of the Mars Rover. The capabilities of each subsystem and how they interacted with each other were researched and a diagram produced of the required information from each.

After the capabilities of each subsystem had been researched, a high-level specification was produced, consisting of several functional and non – functional requirements, against which the finished rover could be measured to determine whether it was successful. A structural diagram was also produced to aid design, which depicted how the various subsystems interacted and some of the considerations that should be considered. This can be found in Appendix A.

Table 1: Specification for the Rover

Specification Point:	The Rover Must:	Reason:
1	Be capable of travelling at a speed of at least 50 mm/s	The rover should be able to move at a reasonable pace.
2	Be able to execute a command to drive forwards or backwards up to at least 0.5m	The rover should be able to move in a straight line so that basic position control can be achieved.
3	Be capable of pivoting at an angular rate up to at least 1.2 rad/s.	The rover needs to turn at a reasonable speed.
4	Be capable of pivoting to face an angle between – pi and pi radians.	Turning to the required direction is important so that objects can be avoided, or a specific path followed.
5	Detect the colour of coloured balls in front of it.	By detecting coloured objects, the rover can be made to avoid or move towards them depending on the circumstance.
6	Detect distance of coloured objects in front of it.	This will be important in calculating the speed to reach destination, in moving around an obstruction and ensuring the camera is positioned to identify features.
7	Track X and Y position of the Rover relative to the Start Position ± 1cm	Tracking location is crucial to efficient movement of the rover.
8	Track Angle of the Rover Relative to the Start Position $\pm$0.04 radians.	See above.
9	Track the current battery charge.	Allows the rover to perform range estimations and allows decisions on prioritising power usage to be made.
10	Estimate the range it can travel from the current battery charge using movement data.	This will allow decisions to be made regarding when the rover should be in use.
11	Be capable of receiving position instructions wirelessly from a website.	This enables manual control of the rover in cases where necessary, for instance a set of obstructions limiting the rover’s ability to move.
12	Be capable of transmitting position, movement, mapping, and battery capacity data wirelessly to a website.	Remote monitoring of rover conditions allows for the data it captures to be passed on and used, and for its status to be recorded in case manual override is required.
13	Be capable of detecting an object in close proximity and performing maneuvers to avoid it if necessary.	It would be desirable to avoid obstacles.

Control

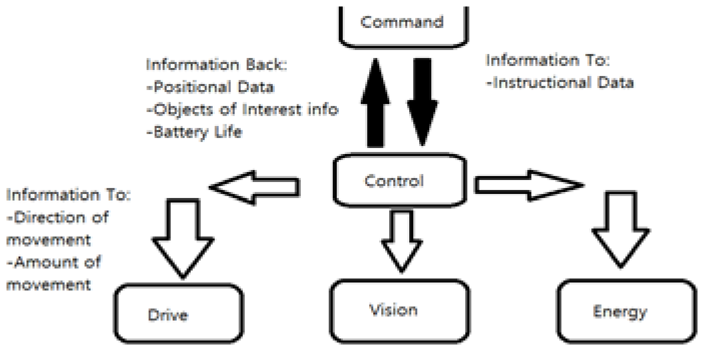
This section is dedicated to the design and thought process behind the design and planned implementation of the Control subsystem of the Mars Rover.

Before planning and designing an overall implementation, it was necessary to analyse and understand the apparatus the Control subsystem was required to utilise, and the functional purpose of the subsystem in the larger context of the rover. The Control subsystem consists of an ESP32 microcontroller chip. The system in question is required to act as the mutual access-point of all the other subsystems that make up the rover: It must receive instructions from a remote web-based interface (the Command subsystem,) perform local filtering and treatment of the data, and then decode and transmit the relevant information to each subsystem with individual consideration. For instance, the system must be able to decipher the meaning of an instruction, such as moving, extracting the relevant parameters such as distance, or turn-angle, and then efficiently transmit this information to the Drive subsystem to enable rover movement.

The communication however is not necessarily half-duplex in nature, the ESP32 will be required to receive and process information and feedback sent back from its connected subsystems. It must receive constant streams of positional feedback data from the Drive subsystem, information from the Vision subsystem about colours and identification of objects of interest, and battery usage information from the Energy subsystem. The system must identify and process the information separately and transmit this back to the web-based Command subsystem to provide the user with a usable, real-time connection with the rover system.

Understanding this information, the next phase of the design of the system was to propose a method of communication, and the protocols required to mediate interfacing between Control and the other individual subsystems. To help understand the functions of the subsystem, a diagram was produced to observe the flow of information.

Figure 2: Block Diagram of Rover



Condensing the requirements, a specification for the Control subsystem was devised, taking inspiration from the rover specification:

Table 2: Control Specification

Specification Point	This Subsystem Must:	Reason
1	Be able to receive instructions from a remote Command terminal.	These commands will be needed to control the appropriate subsystems.
2	Be able to locally process instructions and determine the correct action to conduct.	This subsystem will be the controller for the other subsystems.
3	Provide a minimal lag wireless interface with the remote webserver, ideally with a latency of less than 1 second.	Minimal lag will improve the communication with the website interface.
4	Process instructions locally and efficiently to reduce lag to the rest of the rover.	As this subsystem controls the other subsystems, it should be able to react to data quickly enough to respond in time.
5	Use local ports to write processed instructions to relevant subsystems.	The subsystem must have a method to communicate with the other subsystems.
6	Be able to provide coordinate (X, Y) data to the Command subsystem.	The command subsystem will produce a map of the rover's movement and will require the position data.
7	Be able to uniquely identify colours, coordinates of objects as well as distances to objects.	By detecting obstacles, they can be avoided or moved towards to be examined more closely.
8	Be able to self-act depending on the proximity to the objects in question.	It may not be possible for a user to manually avoid the obstacles in time.
9	Provide a mixture of manual and autonomous instructions. Being capable of manual operation whilst also being able to self-drive within a defined area.	In some circumstances the user may want to take control.
10	Use computer vision to identify colours of objects, map their area to Command and perform deterministic evasive actions if too close to the object.	By adding obstacle's locations to a plot, a map of the area can be built.

Given the nature of Control within the rest of the Rover, it was necessary to communicate with the group members responsible for other subsystems to better understand and agree upon the most effective protocols for data-transfer. To simplify and make design and assembly more meaningful, it was decided to 'pair off' subsystems, by ensuring individual links work rather than trying to assemble and control all the subsystems at the end. By working to ensure Command and Control communicated, and then to Drive, Vision and Energy, this would result in a reduction of errors when completing the final rover and gives a higher degree of confidence in the chosen implementation.

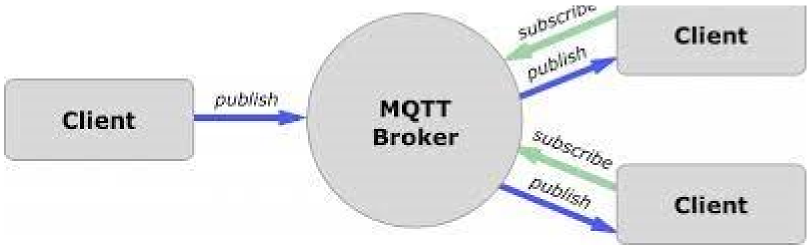
To liaise with the hardware-based subsystems, the UART (Universal Asynchronous Receiver-Transmitter) protocol was utilised for serial-communication, effectively writing the relevant information in bytes for the other systems to read and act upon. The UART protocol effectively frames transferrable information such that the start and end of the binary data is interpretable, with bits being read in big-endian format, such that the most significant bits are read left-most and thus first. The ESP32 has 3 UART ports, 1 of which is by-default used by the connected computer that uploads the code, the other one is dedicated to the Drive subsystem, which leaves a UART port to communicate directly through to Vision.

Facilitating wireless communication between the remote Command system and the ESP32 involved deciding between various internet-based protocols. The initial preference was towards HTTP, as that was the market standard protocol, providing useful information and packet security to ensure cohesive and uncompromised information transfer to the Control subsystem. However, upon consultation with the Command member and with experienced GTAs, it was concluded that HTTP based communication systems would have unnecessarily extra delay, due to the increased likelihood of package loss when using a shared bandwidth, so multiple instructions sent one after another may cause corruption of data. The extra overhead and security reading is helpful, although not necessary for any local processing and is only there to regard security. Considering the system to be designed, one where constant streams of instructions are needed in real-time, HTTP was seen as an inefficient manner to transmit them.

It was decided that the MQTT (Message Querying Telemetry Transport) protocol would be a more suitable choice for communication between Command and Control. This was after further research into its performance and utilisation behaviours, according to associated studies and tests [2]. MQTT was up to 93 times faster than HTTP in communicating time-valuable information, the reduced overhead and simpler package design meant that it was more bandwidth-efficient, and more instructions could be sent in succession over the same connection with a significantly lesser risk of corruption of data, and with a much lower latency. All of this was solidified whilst having a precise degree of quality of service (QoS) ensuring integrity of data with a compressed size. Further consultation and research led to an agreement on the use of this protocol.

The MQTT protocol consists of a simplified transferal and receival based pair system, known as 'publishing' and 'subscribing' to topics, which allows for paired interfaces with associated systems, as seen below. [3]

Figure 3: MQTT Protocol



To use in further context, the ESP32 would be subscribed (listening) to the 'instructions' topic. Which means that updates to this made from the Command subsystem, which are published (sent) to the 'instructions' topic will be transmitted to the ESP32 and recognised by it, the ESP32 will also publish information back to Command via separate topics, namely 'positional data', 'object feedback' and 'battery life'. The Command subsystem acts as the 'broker', which is hosted on a remote server and supplies the ESP32, the central client, with instructions.

Going on to write to the other subsystems would just be a matter of opening the UART ports programmatically and writing serial data from the instructions to them. The overall Control system should ideally be programmed in the Arduino IDE to utilise the full extent of the Arduino based libraries, which is relevant to the ESP32 and provides support for it. This means that writing through UART ports is extremely simple and can be implemented in just a few lines of code.

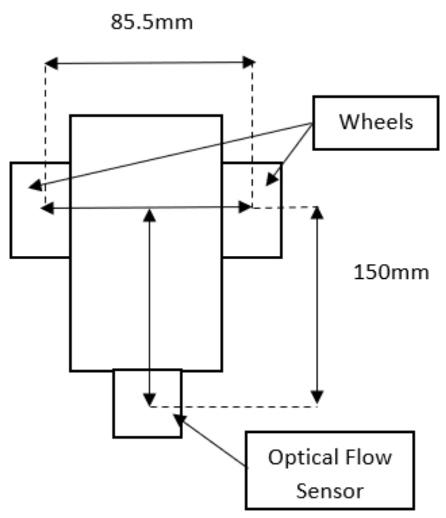
Learning to program the ESP32, what was first task was to learn how to connect it to Wi-Fi, which was completed by using the WiFi.h standard Arduino library's standard example, which simply passed in the SSID and password into an API included connect function. Throughout design and implementation, learning to program the ESP32 required frequent cross-checking with the official Arduino reference [4] which was used as a means of understanding the details of the datatypes that would be used, and the arguments that many of the communication functions required.

Finally, researching MQTT Brokers and Clients, the Mosquitto-broker Arduino connection presented numerous resources and examples for establishing a connection, by using the PubSubClient library [5], which provided standard functions to pass in parameters and connect to MQTT [6]

Drive

The drive subsystem is responsible for controlling the motion of the rover using PWM to adjust the speed of the two wheels and tracking the motion of the rover using an optical flow sensor like the sensors used in computer mice. Figure 4 shows an overhead diagram of the rover: there are two wheels mounted 85.5mm apart, and from the centre of this line, the optical flow sensor is mounted 150mm away at the rear.

Figure 4: Overhead Sketch of Rover



The motors are connected to H drivers so that the direction of them may be controlled as well as the speed. The motors, optical flow sensor and the Arduino that controls them are powered by a buck SMPS. According to the team member responsible for energy, the SMPS should provide a voltage of 3.6V. Two pieces of example code for the drive subsystem were given to the team at the start of the project, a program with functions to operate the camera, written by Dr Zohaib Akhtar and a program to control the motors and SMPS written by Dr Zohaib Akhtar and Yue Zhu at Imperial College London, shown in appendix B. Upon examination of the mouse camera code and datasheet [7], the camera is capable of tracking changes in the x and y directions, returning a value of 400 counts per inch it has moved. By dividing by 400 and multiplying by 25.4, the counts can be converted to mm changes. By sampling sufficiently quickly, these small changes can be summed to find the total displacement of the rover. However, as the camera measures changes in X and Y relative to itself, the changes will have to be processed to work out the rover's coordinates relative to the start position.

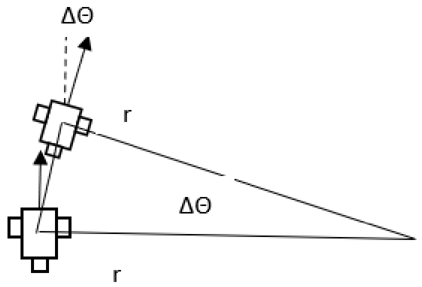
Angle Tracking

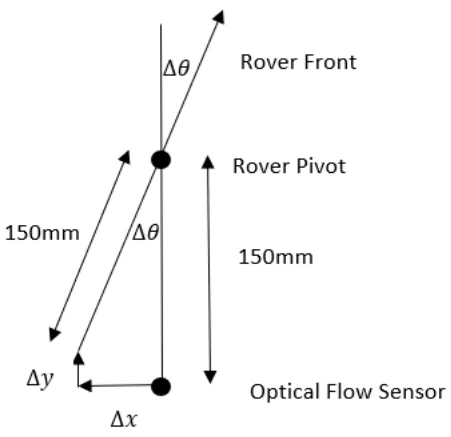
To work out the current X and Y position of the rover, the angle the rover is facing should be known, so that when it moves, the X and Y components of the move can be calculated. For the general case of motion for the rover, where the wheels are moving at different speeds, the rover will move in an arc, the radius of which is determined by the difference in speed of the wheels. Figure 5 illustrates this. If the outer wheel moves with speed v_1 and the other at v_2, then approximations for small changes in the rover's angle can be constructed: $\Delta\theta = \frac{v_1 \Delta t}{r+\frac{w}{2}}$ and $\Delta\theta = \frac{v_2 \Delta t}{r-\frac{w}{2}}$ where w is the distance between the wheels. Rearranging, $r = \frac{w(v_1+v_2)}{(v_1-v_2)}$.

To calculate the change in angle then requires accurate knowledge of the speed at which each wheel moves. With the equipment on the rover this is not simple as the individual wheel speeds cannot be directly measured by the optical mouse sensor so disturbances cannot be accounted for. To achieve accurate angle tracking, r must be known, so to simplify the problem, when a change in direction is required, the wheels can be PWM controlled with the same magnitude but different directions. This will cause them to turn with approximately equal and opposite velocities, and therefore r will be approximately 0. The rover should pivot on the spot.

Because the optical flow sensor measures change to x and y relative to itself, large changes in angle will be seen to be changes in the x direction. Therefore, to find an accurate estimate for changes to the angle, measurements must be taken from the sensor at a high enough frequency relative to the angular rate of the rover around the pivot point. Figure 6 illustrates the geometry of small angle changes. The change can be approximated as $\Delta\theta \approx \arctan\left(\frac{\Delta x}{150-\Delta y}\right)$. By summing these changes together, the angle of the rover relative to its starting position can be estimated.

Figures 5 and 6: Turning in an Arc and Angle Tracking Diagram





Position Tracking

With two wheels, the rover can turn on the spot or drive forwards and backwards. When the rover moves in a straight line, the changes in x and y relative to the starting position can be estimated if the rover's angle is known, by multiplying the change in y relative to the rover measured by the optical flow sensor by $\cos(\theta)$ to get the change in x coordinates and by $\sin(\theta)$ to estimate the change in y coordinates. The position relative to the start position can then be estimated by adding these changes to the current coordinates.

Power Supply

According to the team member responsible for energy, the power supply should produce a voltage of 3.6V. The SMPS will have to be configured accordingly and when implementing motor controls, this should be considered.

Angle and Position Control

The rover should be able to accurately control the angle it is facing, as this is important for allowing the rover to drive to the desired coordinates. As previously explained, the rover can only accurately track its angle when rotating on the spot. The speed of the rotation can be measured with the mouse camera and closed loop control can be used to ensure the rover accurately turns to the correct angle. An initial test on a rough surface (a carpet) showed that when the wheels are PWM controlled with values 255, the maximum, and in opposite directions, the rover can make 5 rotations in approximately 26 seconds, and therefore has an angular velocity of approximately 1.2 Rad/s. Controlling the angular velocity to be greater than this would therefore be unnecessary, and so this should be the maximum value.
As with angle control, the rover should be able to drive to a set of coordinates with a suitable degree of accuracy. Measuring the coordinates using the mouse camera, a closed loop controller can be implemented to implement this. From the test to determine angular rate, it can be calculated that on a rough surface the wheels have a maximum speed of 50mm/s.

UART

The Arduino controlling the drive subsystem must be able to communicate with the control subsystem so that position data can be sent and instructions for movement can be received. This was discussed and it was agreed that UART would be used. The Arduino Nano Every has a second serial port that can be used to send and receive data.

Subsystem Specification

Taking the above information of the capabilities, limitations and requirements of the drive subsystem into account, a specification was produced that the finished subsystem could be tested against when determining whether the subsystem is successfully implemented.

Table 4: Specification for the Drive Subsystem

Drive Specification Point	This Subsystem Must:	Reason:
1	Measure changes to the angle the rover is facing.	The change of the rover's angle will be required to track the rover's position. It should be reasonably accurate to reduce long term inaccuracies. 0.04 Radians is approximately 2.5 degrees, which is relatively small.
2	Track the current angle of the rover relative to the starting angle with an accuracy of ±0.04 radians per 2π radians turned.	The current angle will be required to implement control over the rover's rotation and to track x and y coordinates.
3	Measure changes to the coordinates of the rover.	The changes in and x and y coordinates will need to be measured to a suitable degree of accuracy for measuring the x and y coordinates A tolerance of 1 cm per metre moved is reasonably accurate.
4	Track the current coordinates of the rover relative to the starting position with an accuracy of ±10mm per 1000m moved.	The x and y coordinates are required so that other subsystems can track the rover's position and for control of the rover's movement.
5	Receive instructions to turn to a specified angle from the control subsystem using UART.	The control subsystem will integrate the other subsystems and as such will require control of the rover's movement.

6	Receive instructions to move forwards a specified distance from the control subsystem using UART.	The control subsystem will integrate the other subsystems and as such will require control of the rover's movement.
7	Send x and y coordinate data to the control subsystem using UART.	The control subsystem will communicate with other subsystems which may require the rover's position.
8	Be capable of turning at a specified angular rate between -1.2 and 1.2 rad/s with a tolerance of ±0.5 rad/s and a response time of 0.5s.	Fast angular rate response will be important for controlling the angle of the rover.
9	Be capable of turning to a specified angle ±2.5 degrees within 5 seconds.	In a rover for exploration, accuracy of position control is likely to be more important than speed.
10	Be capable of moving forwards or backwards by a specified distance of ±5mm with a velocity of at least 100mm a second.	In a rover for exploration, accuracy of position control is likely to be more important than speed.
11	Be powered from a 3.6 ±0.05V buck SMPS.	The energy subsystem produces a voltage of 3.6V.

Vision

The vision system is responsible for detecting the coloured balls (obstacles) and sending the distance readings to the control subsystem for that data to be processed and sent to other subsystems where necessary. The initial design given by Dr Edward Stott's GitHub repository [8] was a design that would highlight objects in red when the camera was detecting that something was red and would also draw a bounding box around it. Using this as a basis the next steps forward were to expand the colour detection so that it would be possible to identify pixels that could be part of the balls, expand on the current bounding box so that there is a bounding box for each of the ball colours, then using that to calculate the distance to the closest obstacle, and finally filtering out any glitches and unreasonable results. Once theses functionalities are achieved, then we could start adding a means to communicate with the control subsystem. This was done through UART as the drive subsystem was already using this means of communication and it already posed some familiarity due to being used in previous projects.

Detection - Using the template for red detection expand on this for the other coloured balls. If there's time filter out/in some pixels by storing row of pixels and running some sort of algorithm on them, implement edge detection for the black ball.

Distance - Find a way to calculate the distance to an object using the actual width of the object and the perceived width of the object.

Glitches - Apply some logic to the distance readings to see whether the reading is impossible or not to filter out the extremes, values that would be too close to the camera and values that would be too far from the camera.

Communication - Add a UART serial block to the qsys block design and add a means of testing the UART connection to the main file of the camera test. From there conducted any debugging to make sure that the correct data and format is sent to the control subsystem.

Command

The command subsystem, together with control, allow the rover to be controlled remotely via instructions input by a user. This is the first step in the process of controlling the Mars Rover. There is no hardware component to implement the command portion of the project, so various software solutions were explored. This alone gives some insight into the initial requirements.
Initial Requirements for Command:

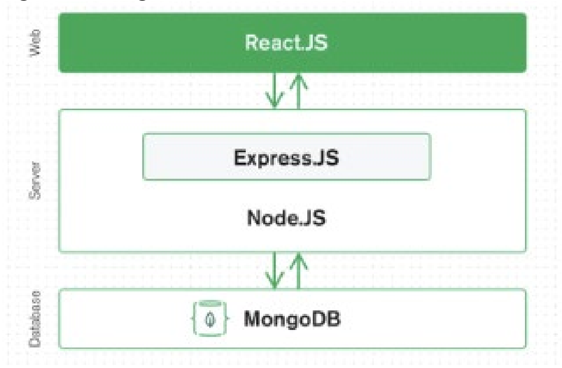
1. Creating a way to input instructions for the rover.
2. Storing instructions so they can be accessed and manipulated.
3. Decide on a protocol to communicate commands with and listen to status updates from the ESP32.
4. Present status updates to the user

Throughout the project, as research was done, tasks were completed and following discussions with the whole group, the requirements became more detailed and addressed problems that were faced at the time. To provide a more comprehensive log of how these initial requirements developed, the versions will be made clear in the implementation section and will assessed in the conclusion.
To address the first requirement, the decision to create a web application to serve as the frontend was made. This would provide a user-friendly way to input commands as well as display updates to the user which is the last requirement stated. The MERN stack provided a way for a full-stack web app to be developed.

- MongoDB – Open source, document-based database to store application data
- Express – Web framework for Node.js
- React.js – Open-source, frontend JavaScript library for creating user interfaces.
- Node.js – Open-source, cross platform, JavaScript runtime environment

React.js is a frontend JavaScript framework that uses Components to build different parts of a Web Application and connect them to data on the backend server. The rendering method uses HTML to build an interactive user interface.
Express.js is a server-side application that wraps HTTP requests and responses. Express.js is a Node.js framework that makes it easier to write back-end code and to develop REST APIs to perform requests. Node.js is very fast due to asynchronous language which means it has better performance for data access.
MongoDB is a document-based, NoSQL database which means data is saved using collections and documents allowing for easier integration with web applications. It works well with Node.js to store, manipulate and represent JSON data which also allows for a fast exchange of data between client and server. [14]

Figure 7: MongoDB Communication



Together, these technologies create an efficient method to develop a Web Application as well as a seamless and integrated user experience, using entirely JavaScript and JSON. The user interacts with the React.js components at the application frontend in the browser and the frontend is served by the application backend residing in the server. JSON data flows naturally from frontend to backend which is how the second requirement will be met. For new instructions being posted by the user and historic instructions being displayed and updated on the web application, the react client should send HTTP requests and retrieve HTTP responses. [14]
Naturally, the HTTP protocol could also be used to wirelessly communicate with the ESP32, but upon further research and discussion with Adam Bouchaala and Control, MQTT was decided on due to its advantages discussed in the Control section.

Integration

Implementing all five subsystems at once will lead to a large amount of debugging and testing, so integration will be done by parts: two pairs of subsystems will integrate with one another and ensure that their objectives are fully met without any issues in communication or operation of the double-system, before being combined with the fifth to fully realise the rover design. As it is unlikely that all modules of the project will progress at an equal rate, thought should be given to how integration will be done if one or more modules are finished significantly before or after others.

Integrating by parts should enable greater functionality in each module to be maintained, refined and tested with each iteration of the integration, whereas in all-at-once integration some more advanced features in one subsystem may be lost in favour of a basic implementation as there may not be sufficient time to test these features and hence are not retained in the final combined design. Additionally, integrating all the modules at once could lead to an unresolvable number of issues where they cannot be debugged faster than they appear, leading to a final design that is unreliable due to unresolved issues – this especially will be avoided by integrating by parts.

The Drive and Vision subsystems are a logical choice for subsystems to combine, as together they ensure that the rover can fulfil the baseline objectives of the project by detecting objects and moving around, towards or away from them.

- A sufficient baseline for the Drive subsystem before starting to combine with Vision is reasonably precise manual control of the movement of the rover, with the ability to turn on the spot and move in a direction for a desired amount of time or distance, in addition to the development of some position tracking and control.
    - Ideally, before integration the Drive subsystem will also be able to specify movement in terms of distance instead of time elapsed, combine sets of instructions to perform more complex manoeuvres and have more advanced position control capable of dealing with a small obstruction or change in command, but if Vision is completed before Drive these can be implemented and iterated during the integration of the two subsystems.
    - Initial tests will be simple movements in one direction and turning in large angles can be used to test that the rover is reasonably accurate in its motion, responds to commands and to ensure that it correctly and accurately records position data. Calibration may need to be done to account for the conditions of the test area, or markings on it, in addition to the effect of test surface e.g., a carpeted surface instead of a smooth one on how precise radial and directional movement is – a printed protractor will be used to assist in visually identifying the accuracy of angular movement.
    - Further testing will be more rigorous to ensure that the system is precise enough for the rover's purpose – a square area will be marked by tape on the floor and the rover will trace this area multiple times, ideally remaining directly over the tape and making turns of exactly 90°. The ability of the rover to navigate more complex shapes can then be tested, for instance movement across a star-shape and radial motion around a semi-circular area.
- Before combining with Drive, the Vision subsystem should be able to use the on-board camera to correctly identify objects of interest by at least colour and discern simple obstructions.
    - The identification of more complex obstructions can be tested once this is established, with contrasting colours or placebo objects-of-interest offering potential methods to explore the efficacy of the colour detection system.
    - Additional testing can be done with different objects to determine the effective field of view of the camera and maximum distance at which identification can reliably be done. Testing can also determine if the camera has blind spots where an obstruction – potentially with some vertical element – is not noticed by the camera, as this will need to be accounted for when doing more advanced testing later.
- After testing has been done regarding both precision of movement of the rover and the distance at which objects, especially obstructions, can be identified, an optimal speed can be determined where the rover can within all margins stop before hitting an obstruction, but move at a speed to sufficiently map a test area as efficiently as possible. The 'reaction time' or latency that manual commands require before they are executed may contribute to this, in addition to how long it takes for them to be manually selected and submitted to the rover.
- A final test area, initially square with no obstructions, will be used to iteratively test how the completed rover performs. The combined Drive/Vision system will ideally use this area to fully determine the efficacy of both subsystems and to test manual rover control fully before automatic operation is implemented.

The Command and Control subsystems enable the rover to function effectively by providing a platform for data to be monitored, commands to be given and for these commands & data to be transferred between the relevant modules as necessary.

- By the time integration with other modules begins, the Command subsystem should comprise of a simple working web-app, with methods for manually sending movement commands to the rover and displaying relevant information.
    - As the Command module is integrated with other subsystems the data that is required to be displayed on the website will likely change, so a robust construction capable of showing plenty of data legibly is preferable. Frequent communication with Control will ensure that the data will be in a known format.
    - The ability to change which information is displayed quickly and simply in the browser will improve the visual presentation of the webpage but is not a high priority to implement before integration with other modules begins – similarly a home page and information that describe and inform about the project are important for a professional front-facing page but ensuring that Command integrates well with more physical modules takes precedence if time becomes a limiting factor.
    - A full test of the ability to send & receive actual live information will be one of the first test cases done when integrating with other systems. Preliminary tests using ideal data, or data approximated or sampled from real Drive/Vision data can be done to ensure that the web dashboard is fit for purpose, but live tests with real should be done as soon as is reasonable, depending on the progress of the other modules.
- The Control subsystem is focused around ensuring that all modules communicate effectively, with the motors, FPGA, movement commands and operation status of the rover needing to be transferred between the four other subsystems. The Control subsystem will be crucial for the implementation of multiple modules and before any pair of the other modules are integrated, Control must ensure that a format for information to be sent and received has been established.
    - In the project as a whole Control ensures that all modules communicate effectively, but during initial implementation and throughout testing this can to some extent be tailored to the modules that are in an integrable state if time is a limiting factor in how much the Control system can accomplish.
    - Implementing and testing Control with other modules will change with each module, but generally the Control, Integration and relevant module[s]' team members will join a Teams call and collaboratively work to implement all of the necessary criteria on the Integration team member's rover.

The Energy subsystem provides power to the other modules from a set of rechargeable batteries and has objectives that are largely independent of other modules, so should be tested as soon as they are implemented by the Energy team member.

- Recording the power consumption of the rover when it is operating and the charging behaviour of the batteries and monitoring the state of charge (SoC) while the rover is in use will ensure that it will have the ability to perform its objectives while safely returning to the charging station.
- While some of Energy's objectives are independent of other modules, it will eventually need to integrate with Control, to ensure that it communicates with the rest of the rover as required, and Drive, to implement and test range estimation alongside position tracking/distance measurement.
- As Energy requires more minimal Integration than the other modules, it is largely unaffected by faster or slower development than any of the other subsystems, but sufficient time should be allowed for testing with Control and Drive and there should be frequent communication with all the modules throughout the course of the project.

Implementation and Testing:

Control

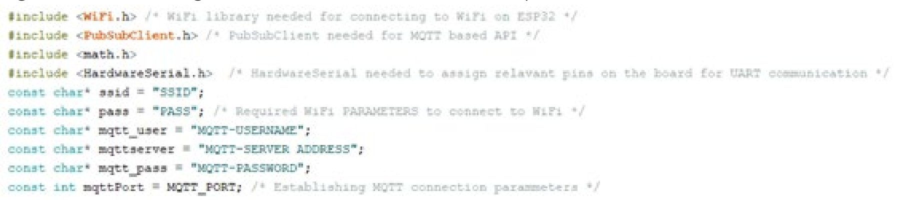
The following section details the implementation of the Control subsystem. Distinct to the other subsystems, the Control structure was developed as 3 main versions, rather than a general subsystem, to realise the full capabilities of Control. The implementation can be divided into 3 main phases of work: Communication, Operation, and Automation (/Observation). Each version added new methods and functions and successively these versions built on top of each other, to complete the full subsystem.

First Version of Control (Control V1: COMMUNICATION)

The first iteration of the Control subsystem was the primary backbone of the current system, made as a primitive way to test the connections from server to ESP32, and from ESP32 to related subsystems.

To begin with, a sample broker system would be used to test the flow of information from online server to the ESP32, using MQTTLens to provide a simple testing interface for both publishing and subscribing to relevant topics, as well as for quick debugging while the Command system was still being developed. To initialise the connection, primary parameters globally defined were required to be used as variables throughout the body of code, viewed in figure 8:

Figure 8: Establishing network constants which will be passed to connection methods to establish communication with the broker



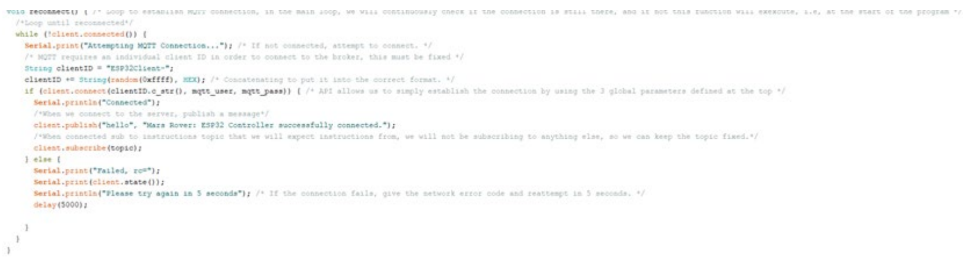
```
#include <WiFi.h> /* WiFi library needed for connecting to WiFi on ESP32 */
#include <PubSubClient.h> /* PubSubClient needed for MQTT based API */
#include <math.h>
#include <HardwareSerial.h>  /* HardwareSerial needed to assign relavant pins on the board for UART communication */
const char* ssid = "SSID";
const char* pass = "PASS"; /* Required WiFi PARAMETERS to connect to WiFi */
const char* mqtt_user = "MQTT-USERNAME";
const char* mqttserver = "MQTT-SERVER ADDRESS";
const char* mqtt_pass = "MQTT-PASSWORD";
const int mqttPort = MQTT_PORT; /* Establishing MQTT connection parammeters */
```

The const char arrays contain the relevant Wi-Fi and MQTT information needed to maintain communication. Constant SSID and Password variables are fixed and adjusted by the user. Ultimately during the integration and testing of the rover, the SSID and Password are simply the Wi-Fi details of the integration member's home network. The mqtt_user, mqttserver and mqtt_pass variables contain relevant information regarding the server hosting the broker, as well as the port that it will be listening on.

By using the WiFi.h and PubSubClient libraries respectively, both Wi-Fi and MQTT connections were supported on the Arduino IDE for use with ESP32 boards, by creating a stack instance of these clients at the start of the code, this will allow the system to use the full functionality of their respective APIs and ultimately connect to the webserver.

Connecting to Wi-Fi simply involved adapting the example code from the Design section's research for purpose. The loop simply attempts a connection to the network's SSID and password, continuing to loop until it connects, displaying connection information to the Serial debug terminal, shown in figure 9.

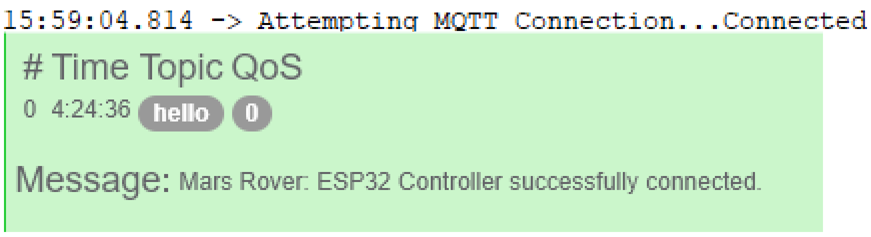
Figure 9: Algorithm to connect to MQTT, inspired from a basic example.



```
void reconnect() { /* Loop to establish MQTT connection, in the main loop, we will continuously check if the connection is still there, and if not this function will exexcute, i.e, at the start of the program */
  /*Loop until reconnected*/
  while (!client.connected()) {
    Serial.print("Attempting MQTT Connection..."); /* If not connected, attempt to connect. */
    /* MQTT requires an individual client ID in order to connect to the broker, this must be fixed */
    String clientID = "ESP32Client-";
    clientID += String(random(0xffff), HEX); /* Concatenating to put it into the correct format. */
    if (client.connect(clientID.c_str(), mqtt_user, mqtt_pass)) { /* API allows us to simply establish the connection by using the 3 global parameters defined at the top */
      Serial.println("Connected");
      /*When we connect to the server, publish a message*/
      client.publish("hello", "Mars Rover: ESP32 Controller successfully connected.");
      /*When connected sub to instructions topic that we will expect instructions from, we will not be subscribing to anything else, so we can keep the topic fixed.*/
      client.subscribe(topic);
    } else {
      Serial.print("Failed, rc=");
      Serial.print(client.state());
      Serial.println("Please try again in 5 seconds"); /* If the connection fails, give the network error code and reattempt in 5 seconds. */
      delay(5000);

    }
  }
}
```

Figure 9 describes the main body of code that facilitates the connection to MQTT, which drew inspiration from a further example when researching the mosquito-MQTT broker system. The main loop first does a standard Boolean check to establish whether the system is already connected to the broker, if this is false, it will establish communication with the broker by using the .connect() function, which is included in the PubSubClient's MQTT API. This function, when passed with the broker username and password, as well as with a unique, fixed client ID to distinguish the ESP32 as a client to the network, allows the ESP32 to connect to the broker. Upon a successful connection it will display a message on the Serial debug monitor and publish a message to the broker via a test topic, indicating a successful MQTT connection, as seen in figures 10 and 11, respectively, finally subscribing the ESP32 to the instructions topic to initialise it for receiving instructions. Failing this, it will provide a network error code to enable the user to debug the issue and reattempt a connection after 5 seconds.

Figures 10 and 11: Successful connection to MQTT, followed by a transmission from ESP32 to Broker, which is subscribed to it.



```
15:59:04.814 -> Attempting MQTT Connection...Connected
```

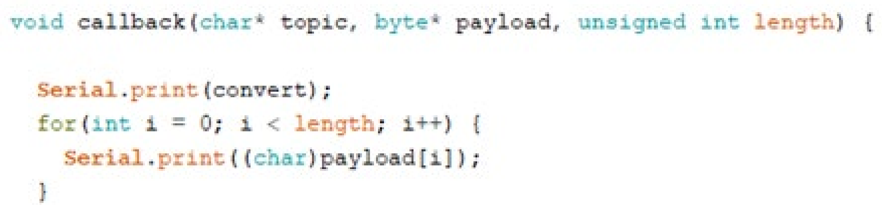
Time Topic QoS
0 4:24:36 hello 0

Message: Mars Rover: ESP32 Controller successfully connected.

Initially, the former if statement did not pass in username and password parameters and consisted solely of binding to the IP Address of the broker's server, however it was changed in preference of the username and password authentication-based connection, which provides a layer of security and assurance to the end user, as well as preventing errors regarding code portability, meaning that it was able to be reuploaded and reproduced on any team-member's system.

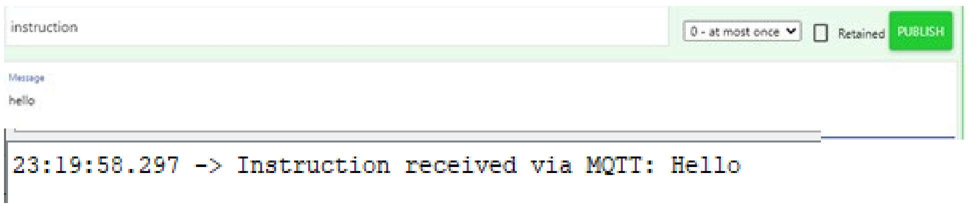
To develop a basic system of instruction processing, a callback function was devised, this function enables the ESP32 to act whenever an MQTT transmission is received and provides a protocol for how to treat and process the incoming data. The callback function in MQTT passes in 3 parameters every time a message is passed: The topic that the message is being received from, the 'payload', which is a byte array containing the contents of the message, and the length of the payload. The initial function that was written was to characterise the initial payload, seen in figure 12, which simply loops over the elements of the payload array and displays the characterised elements on the Serial screen. This was to ensure that messages could be successfully read from the Broker to the ESP32. Figures 13 and 14 show the Broker and Client sending and receiving a test message respectively, showing successful transmission.

Figure 12: An initial implementation of the callback function, which simply iterates through the characters of the payload received from MQTT.



```
void callback(char* topic, byte* payload, unsigned int length) {

  Serial.print(convert);
  for(int i = 0; i < length; i++) {
    Serial.print((char)payload[i]);
  }
```

Figures 13 and 14: Demonstrating sending and receiving an instruction from a simulated Command broker to the ESP32 (subscribed to instruction feedback), via MQTT.



instruction

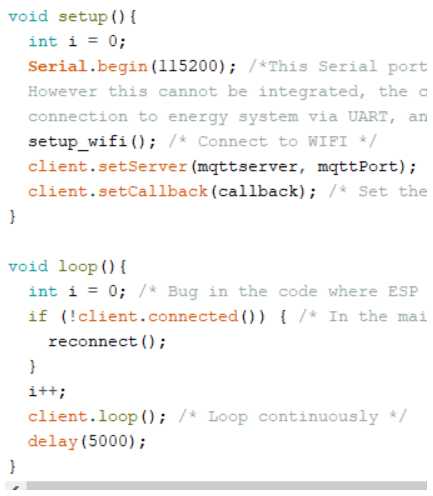
0 - at most once ☐ Retained PUBLISH

Message
hello

```
23:19:58.297 -> Instruction received via MQTT: Hello
```

The setup and main loops in Figure 15 initialise the WIFI and MQTT connections, passing in the MQTT server name and port number to listen on, and define the callback protocol to be the function designed prior. As well as initialising the Serial port for debugging at a baud rate of 115200. The main loop simply repeats to check if the connection to the Broker still exists, and loops the client, reconnecting when the connection is broken, once every 5 seconds. Ultimately, the connection to the Command system proper was trivial, as all that was necessary was to change the port number and server parameters to connect to the Command terminal, which is pre-subscribed to feedback topics, as well as the ESP32 being pre-subscribed to instruction topics, enabling a real-time response to be read equivalently.

Figure 15: SETUP and LOOP voids for the v1, which simply passed in the appropriate network parameters to initialise the environment.



```
void setup(){
  int i = 0;
  Serial.begin(115200); /*This Serial port
  However this cannot be integrated, the c
  connection to energy system via UART, an
  setup_wifi(); /* Connect to WIFI */
  client.setServer(mqttserver, mqttPort);
  client.setCallback(callback); /* Set the
}

void loop(){
  int i = 0; /* Bug in the code where ESP
  if (!client.connected()) { /* In the mai
    reconnect();
  }
  i++;
  client.loop(); /* Loop continuously */
  delay(5000);
}
```

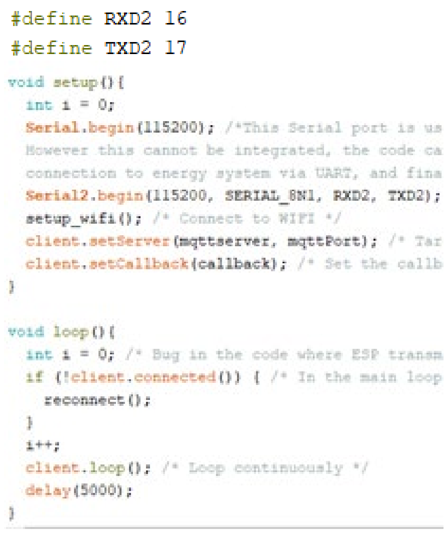
Having established the full range of communication from broker to ESP32. The next natural implementation step was to process and manipulate the information that was received. Integrating Control with the Drive subsystem was a straightforward process, shown in figures 16 and 17, which simply mandated assigning suitable GPIO pins from the ESP32 as the respective RX and TX pins for UART transfer, and then connecting the corresponding pins with the Drive subsystem's equivalents, initialising the UART connection entailed simply initialising the Serial port, set to Serial2, passing in the relevant RX and TX macros defined earlier, with a matching baud rate of 115200 bps, enabling valid and consistent information transfer. The Drive subsystem that was completed consisted of 3 main movement capabilities:

- Moving Forward by X mm – Indicated by transmitting the following bytes: 254, followed by the number X, between 0 and 255.
- Turning to position X, defined as turning an angle equal to radians clockwise– Indicated by transmitting the following bytes: 253, followed by the number X, between 0 and 255.
- Stopping, which ceases movement when sent. This is the default state of the Drive subsystem – Indicated by transmitting a constant 255 in bytes.

Adapting the Callback function to handle these different classes of instruction was done by enforcing a defined format of instructions:

- FORWARD X
- TURN X
- STOP

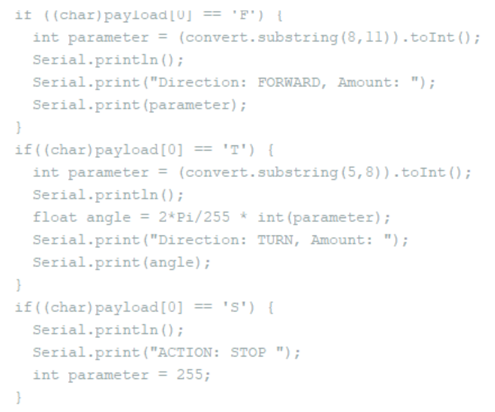
Figures 16 and 17: Globally defining the GPIO pins to connect the ESP32 to the Drive's Arduino Nano Every, initialising the UART port to communicate with Drive at a Baud Rate of 115200 BPS, to be able to align communication.



```
#define RXD2 16
#define TXD2 17

void setup(){
  int i = 0;
  Serial.begin(115200); /*This Serial port is us
  However this cannot be integrated, the code ca
  connection to energy system via UART, and fina
  Serial2.begin(115200, SERIAL_8N1, RXD2, TXD2);
  setup_wifi(); /* Connect to WIFI */
  client.setServer(mqttserver, mqttPort); /* Tar
  client.setCallback(callback); /* Set the callb
}

void loop(){
  int i = 0; /* Bug in the code where ESP transm
  if (!client.connected()) { /* In the main loop
    reconnect();
  }
  i++;
  client.loop(); /* Loop continuously */
  delay(5000);
}
```

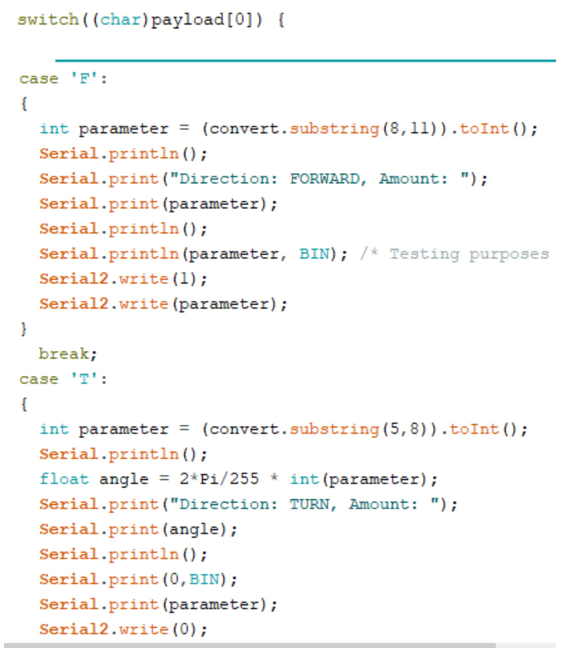
Which perform the priorly defined actions. By enforcing a uniform format of instructions, substrings can be created to isolate the parameters, and the first character can be read to distinguish the instructions themselves, and perform different actions based on what was read. Figure 18 shows an initial example of how this could be treated, by using a simple branch of if-else statements. This structure simply checks against the first character of the byte array, and acts upon it based on whether that letter is an 'F', a 'T' or an 'S'. However, by considering the context and purpose of the system, a long branch of if-statements, with possible further embedded if-statements, could contribute negatively to the time complexity of this IoT dedicated code, where time-management is valuable and key for good functionality. It was elected that in lieu of this system, a Switch statement would be much more efficient for the purposes of instruction response. This is demonstrated in the adapted callback function in figure 19. Figure 20 provides further reasoning to this decision, showing a graph displaying that after averaging 50000 test runs, for an intermediate number of conditions, the switch statement has a significantly faster response. Whilst this pertains to JavaScript, it is a similar trend across all supported programming languages; and for the rover's range of conditions this can be up to 10ms faster, which is a good gain for an IoT system [9].

Figure 18: Initial approach to instruction processing, by embedding actions within an if statement in the CALLBACK function.



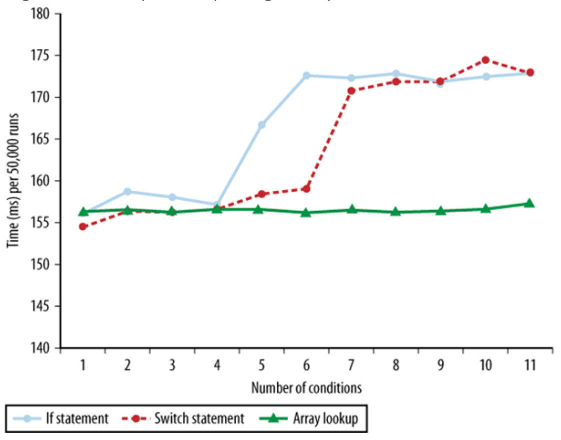
```
if ((char)payload[0] == 'F') {
  int parameter = (convert.substring(8,11)).toInt();
  Serial.println();
  Serial.print("Direction: FORWARD, Amount: ");
  Serial.print(parameter);
}
if((char)payload[0] == 'T') {
  int parameter = (convert.substring(5,8)).toInt();
  Serial.println();
  float angle = 2*Pi/255 * int(parameter);
  Serial.print("Direction: TURN, Amount: ");
  Serial.print(angle);
}
if((char)payload[0] == 'S') {
  Serial.println();
  Serial.print("ACTION: STOP ");
  int parameter = 255;
}
```

Figure 19: Adapted approach to instruction processing, embedding actions within a switch statement, which has a reduced time complexity to embedded if-statements, switches based on first character.



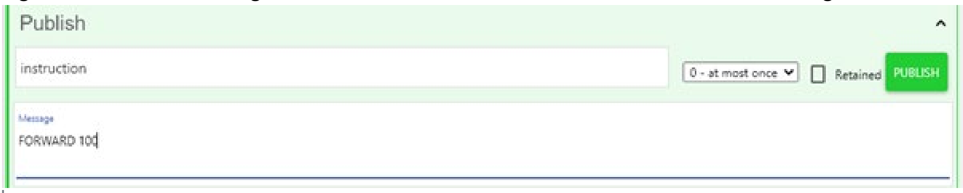
```
switch((char)payload[0]) {

case 'F':
{
  int parameter = (convert.substring(8,11)).toInt();
  Serial.println();
  Serial.print("Direction: FORWARD, Amount: ");
  Serial.print(parameter);
  Serial.println();
  Serial.println(parameter, BIN); /* Testing purposes
  Serial2.write(1);
  Serial2.write(parameter);
}
  break;
case 'T':
{
  int parameter = (convert.substring(5,8)).toInt();
  Serial.println();
  float angle = 2*Pi/255 * int(parameter);
  Serial.print("Direction: TURN, Amount: ");
  Serial.print(angle);
  Serial.println();
  Serial.print(0,BIN);
  Serial.print(parameter);
  Serial2.write(0);
```

Figure 20: Graph comparing the speed of if-else statements vs switch-statements as condition no. increases.

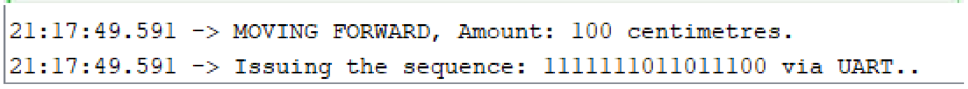


The overall callback function would initially take the payload array and use the Arduino's String library to cast the payload data into a String with the standard constructor. It then switches based on the first character of the payload, acting based on whether it is an 'F', a 'T' or an 'S', based on which case elected, it will then parameterise the number by doing an integer cast on the substring, using the space character in the instruction as a delimiter to the end of the string, which is null terminated. It will then write this as a byte through UART and have a preceding byte written depending on the case. It also displays Serial debugging information, in particular the action taken and the amount, this successfully drove rover movement and the debugging screens in Figures 21 and 22 show this being executed as commands. It writes these bytes through Serial2, which is initialised in the setup loop along with the initial Serial port, where Serial2 is configured to the RX and TX ports 8 and 9 respectively (GPIO16 and 17), which are connected to the corresponding RX and TX ports on the Drive's Arduino NANO.

Figures 21 and 22: Sending a FORWARD instruction via the MQTT broker and receiving it on the ESP32 serial terminal.



FORWARD 100



```
21:17:49.591 -> MOVING FORWARD, Amount: 100 centimetres.
21:17:49.591 -> Issuing the sequence: 111111011011100 via UART..
```

Reflections and Bugs from Control Version 1:

Whilst this system fulfilled some of the initial criteria pertaining communication, there were still some aspects left to be addressed by the system. More specifically receiving and transmitting position data from the Drive subsystem, as well as treating and developing more sophisticated operation checking, these would be major features that would need to be implemented in the next working version of Control.

Whilst instructions were being transmitted successfully, they were being transmitted with a slight lag in between the time of publishing the instruction, and that instruction being acknowledged and acted upon by the ESP32, which corresponded to approximately 5 seconds of difference, which was simply too high to be realistically usable for an IoT device where a user may want to perform precise movements. This would be one major aspect that the next version would need to improve upon.

Instructions being correctly sent and executed is under the assumption and knowledge that the user types in the instructions flawlessly every time during the running of the rover, which is unrealistic for the context of application, whilst the rover is operating the user may need to perform manoeuvres in quick succession to each other and could type an incorrect number, or spell an instruction wrong and execute the wrong command, issue a negative number or a number beyond 255, or even a command where one was not meant to be. In the next working version, this issue needed to be tackled to ensure that only valid instructions will be processed by the rover. Extra tests were devised to validate the code:

Table 5: Test Instructions

TEST	EXPECTED OUTPUT
FORWARD X (Valid Forward Instruction)	Acknowledge and output forward instruction for the appropriate amount.
TURN X (Valid Turn Instruction)	Acknowledge and output turn instruction for the appropriate amount.
STOP (Valid STOP Instruction)	Acknowledge and output stop instruction for the appropriate amount.
FOR (Invalid attempt at FORWARD)	Output Error message, suggesting if they meant FORWARD.
TN (Invalid attempt at TURN)	Output Error message, suggesting if they meant TURN.
FORWARD Y (Y < 0 OR Y > 255)	Output Error message, prompting to type a number between 0 and 255.
ASDFASDF (Random string of letters)	Output a general Error message.

By writing specific error messages for each possible variant of error that can arise, this will give the user greater understanding of the possible errors that arise during the use of the system, and how they can correct them to ensure the rover operates as intended.

Second Version of the Control subsystem (Control V2: OPERATION):

The second version of the subsystem kept the same fundamental connection-based functions, connecting to WIFI and the MQTT broker in the exact same fashion. However, the first aspect to be improved upon in this version and how it connected to its associated systems was the observed input lag between publishing an instruction and the ESP32 receiving this instruction. During the diagnosis and consideration of this issue, the causes were narrowed down to two main factors:

- The main loop is dependent on a defined time delay for checking connectivity and looping the client. The client was looping every 5 seconds, which caused this greatest contribution to the delay.
- The ESP32 has a parameter for Wi-Fi Power Saving mode, by default this feature is set to ON (WIFI_PS_MODEM) when operating. This means that when a Wi-Fi based interaction is not happening, the system will put Wi-Fi based apparatus to sleep, reinitialising the system when a Wi-Fi based function is called. The time needed to restart these systems can contribute to input lag from publishing instructions.

To rectify this main issue, the loop delay was changed in the main loop to run once every 250 milliseconds, which drastically reduced the input delay to provide near-seamlessness between publishing an instruction and receiving it on the ESP32. This change is observed in Figure Z.

When considering the power saving feature on the ESP32, this presented itself as a compromise for the user – although turning it off reduces the lag by a minor amount, in turn this will increase the power consumption from the ESP32. For the testing purposes of this rover, energy and battery consumption is not a factor as it is being powered directly through a USB to the main computer, however in future and ideal conditions, the energy subsystem and charging would also be a factor to consider when using this mode. This allows the user to tailor their experience of the rover to fit their specific needs. For the context of a harsh environmental operation, longevity of battery life over improvements to seamlessness is an advantage, so having power saving mode set to ON may be beneficial. However, in a home-based environment, the user can afford to have more seamlessness and performance power and may wish to turn it OFF.

When attempting to parameterise and validate the initial callback code, a bug was observed in the process of design and testing, whereupon the String conversion of the payload received via MQTT was including extra information unnecessarily, seen in figure 23. By cross-consulting with earlier figures, particularly those that pertain with the MQTT connection, one may be able to observe that extra information such as the ESP ClientID and the MQTT parameters are being included in the received payload string. When considering the source of this error, and attempting to replicate the issue, the following two potential issues were highlighted:

- The String conversion of the data made in the earlier version of the Control system took specific lengths of the substring (i.e., from characters 8 to 11), this meant that the numbers had to have been typed in a form XXX, e.g.: 21 = 021. When changing this to simply take the substring from the 8th character on FORWARD for instance, extra characters were included too which corrupted the instruction, this is due to a mistaken assumption that the payload string must be null terminated. However, as MQTT is a network protocol like any other, there must be some level of extra overhead with each network packet, for instance the ClientID destination, the parameters and/or port, the Arduino String constructor may have converted these characters, intended to be hidden when outputting them as standard characters, to be included with the overall string.
- The Arduino String library is known to allocate at a much more consumptive rate than equivalent String handling equivalents, the ESP32 in comparison has a very limited memory with which to handle such strings, which allocate much more than is needed. This may have led to corruption of the ESP32 memory registers, which led to the excess characters in each string response.

This was eventually fixed by converting the String differently, then using the C String library as an alternative, as seen in figure 24. The characters are individually appended to an empty string, stopping when the bytes of the payload itself stop, preventing the extra overhead data being included, then the instruction string is simply converted to a C string by doing .c_str(). This seemed to remedy the error, as displaying the overall strings seemed to provide the correct output.

Figure 23: Recreation of the bug caused by incorrect/corrupt string parsing.

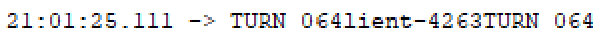
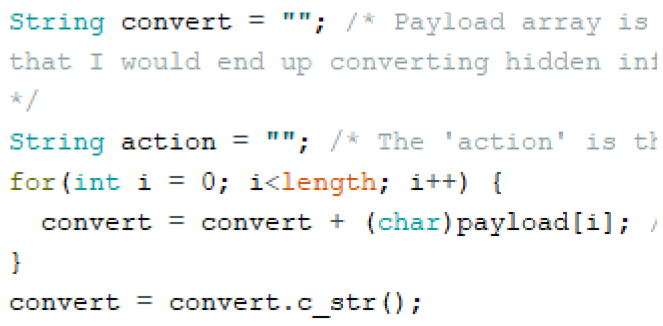
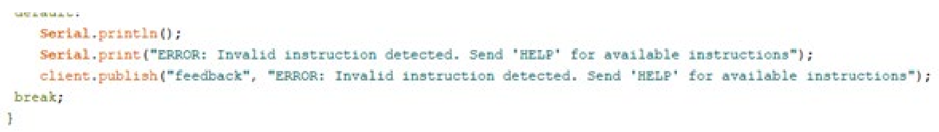

```
21:01:25.111 -> TURN 064lient-4263TURN 064
```

Figure 24 Alternative String parse assignment method.


```
String convert = ""; /* Payload array is
that I would end up converting hidden inf
*/
String action = ""; /* The 'action' is th
for(int i = 0; i<length; i++) {
  convert = convert + (char)payload[i]; /
}
convert = convert.c_str();
```

Another bug during testing was that standard strings that did not match an instruction would simply crash the program. This was because the switch statement did not possess a default clause, which provides it with an action for when none of the cases match, adding this case, shown in figure 25, also provided the first addition to the error checking mechanic of the Control subsystem, as the program could now give an error if no valid instruction is typed.

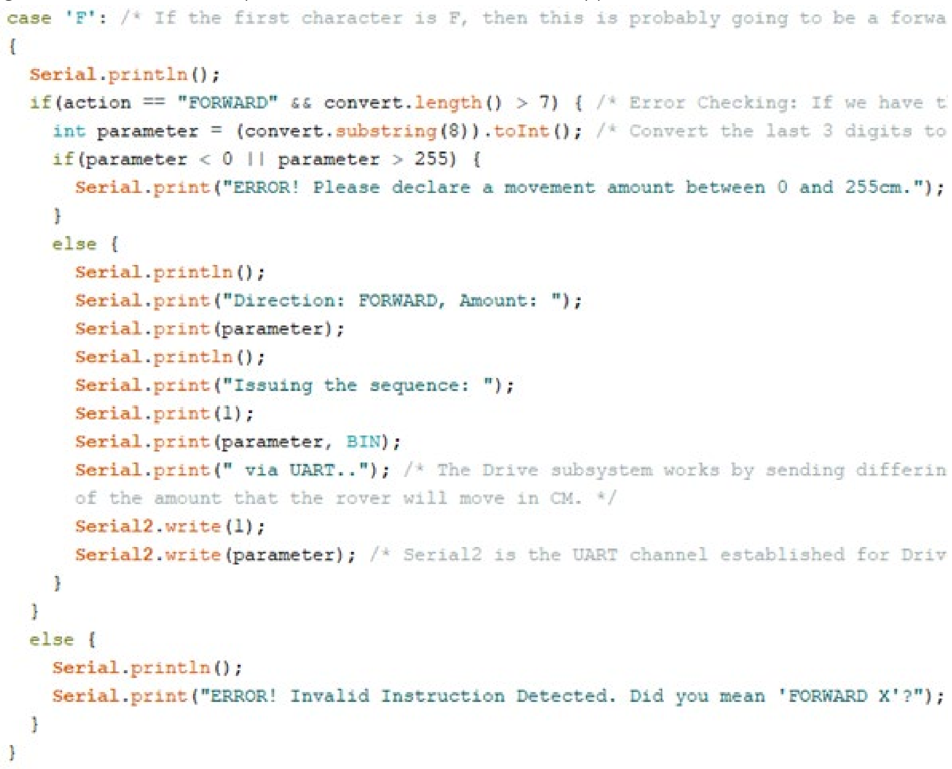
Figure 25: Default case in CALLBACK switch statement.


```
  default:
    Serial.println();
    Serial.print("ERROR: Invalid instruction detected. Send 'HELP' for available instructions");
    client.publish("feedback", "ERROR: Invalid instruction detected. Send 'HELP' for available instructions");
  break;
}
```

To validate the instructions, a separate string was generated from the overall payload string, using the space as a delimiter to parse the 'action' of the instruction (i.e., for the instruction FORWARD X, the 'action' is FORWARD), with similar string generation to the prior case. Each case of the switch statement can be asserted by checking to see if the action exactly matches the valid instruction word before acting. If the action does not match but the first character matches, then the switch case recognises this case as being a situation of a misspelt instruction and provides specific error messages prompting the user to type the correct instruction.

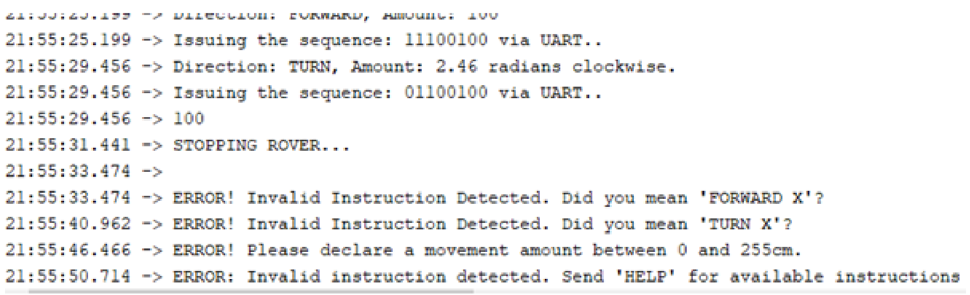
The parameter is parsed by simply taking the first post-space character to the end of the string, rather than using a defined range of characters as before. This enables further error checking, by enclosing within the action checking if statement another if statement to verify the range of the parameter. This means that action is only taken if the action is a matched instruction word, and the parameter is between 0 and 255. If the range is different but the action corrects then it will display a range error, prompting the user to type a number within range, if the word is spelt incorrectly then it will suggest the correct spelling of the instruction to the user. The overall validation process can be seen in figure 26. Having added these validations, the instructions were tested and performed successfully in accordance with the test objectives laid out prior and are shown in figure 27.

Figure 26: Validation example for the FORWARD instruction, applied to all other instructions.



```
case 'F': /* If the first character is F, then this is probably going to be a forwa
{
  Serial.println();
  if(action == "FORWARD" && convert.length() > 7) { /* Error Checking: If we have t
    int parameter = (convert.substring(8)).toInt(); /* Convert the last 3 digits to
    if(parameter < 0 || parameter > 255) {
      Serial.print("ERROR! Please declare a movement amount between 0 and 255cm.");
    }
    else {
      Serial.println();
      Serial.print("Direction: FORWARD, Amount: ");
      Serial.print(parameter);
      Serial.println();
      Serial.print("Issuing the sequence: ");
      Serial.print(1);
      Serial.print(parameter, BIN);
      Serial.print(" via UART.."); /* The Drive subsystem works by sending differin
      of the amount that the rover will move in CM. */
      Serial2.write(1);
      Serial2.write(parameter); /* Serial2 is the UART channel established for Driv
    }
  }
  else {
    Serial.println();
    Serial.print("ERROR! Invalid Instruction Detected. Did you mean 'FORWARD X'?");
  }
}
```

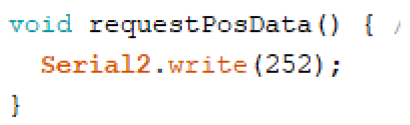
Figure 27: Control subsystem passing all validation tests.



```
21:55:25.199 -> Direction: FORWARD, Amount: 100
21:55:25.199 -> Issuing the sequence: 11100100 via UART..
21:55:29.456 -> Direction: TURN, Amount: 2.46 radians clockwise.
21:55:29.456 -> Issuing the sequence: 01100100 via UART..
21:55:29.456 -> 100
21:55:31.441 -> STOPPING ROVER...
21:55:33.474 ->
21:55:33.474 -> ERROR! Invalid Instruction Detected. Did you mean 'FORWARD X'?
21:55:40.962 -> ERROR! Invalid Instruction Detected. Did you mean 'TURN X'?
21:55:46.466 -> ERROR! Please declare a movement amount between 0 and 255cm.
21:55:50.714 -> ERROR: Invalid instruction detected. Send 'HELP' for available instructions
```

The next addition to the Control subsystem would be to receive positional data from the Drive subsystem and display them on the Command subsystem. This is implemented with two methods, one to issue a request to the drive subsystem to send positional data, which displays the X and Y coordinates when the number 252 is issued as a byte through the UART port, shown in Figure X. The function to acquire the data after a request is made is done in figure 28, which utilises the fact that the UART Serial port uses a FIFO (First in First Out) data structure, meaning the oldest data is at the front of the queue, i.e., the first coordinate that the rover occupies.

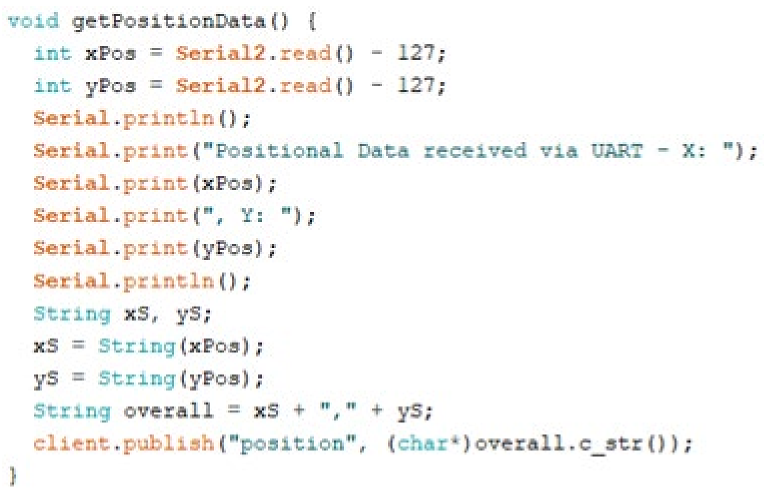
Figure 28: Requesting position data from the Drive subsystem.



```
void requestPosData() { /
  Serial2.write(252);
}
```

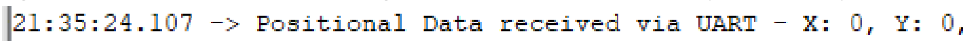
The Drive subsystem sends the coordinate data back in two separate bytes, first writing the X coordinate and then writing the Y coordinate, and both values are offset by +127. The getPositionalData() function, shown in figure 29 reads sequentially two bytes from the UART Serial port at a time, which corresponds to X+127, and Y+127. Upon compensating for the offset value to reach the true values, it then displays this data to the Serial output for debugging, it converts the X and Y coordinates into a string format, and then concatenates them into one set of coordinates in the form 'X,Y', finally publishing this to the MQTT Broker by using the topic 'position'. On the test broker, this is shown to work in figure 30, when the default coordinate 0,0 is transmitted initially; on the Command subsystem, this is shown to work in figure 31 where the initial coordinate 0,0 is 'mapped' onto the coordinate map.

Figure 29: Getting position data.



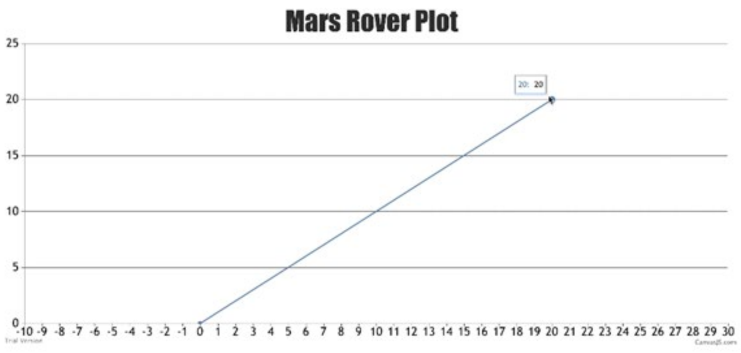
```
void getPositionData() {
  int xPos = Serial2.read() - 127;
  int yPos = Serial2.read() - 127;
  Serial.println();
  Serial.print("Positional Data received via UART - X: ");
  Serial.print(xPos);
  Serial.print(", Y: ");
  Serial.print(yPos);
  Serial.println();
  String xS, yS;
  xS = String(xPos);
  yS = String(yPos);
  String overall = xS + "," + yS;
  client.publish("position", (char*)overall.c_str());
}
```

Figure 30: Default coordinate data being transmitted from the Drive subsystem, read by the ESP32.



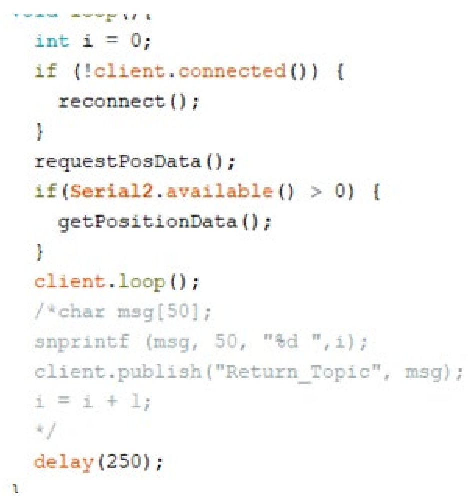
```
21:35:24.107 -> Positional Data received via UART - X: 0, Y: 0,
```

Figure 31: Command subsystem reading, and mapping coordinates transmitted every 250ms via MQTT.



Overall, both functions are used in the main loop, this means that once every 250 milliseconds, the ESP32 will issue a position request to the Drive's Arduino NANO, getting data back. A simple if statement is encoded in the main loop which executes the getPositionalData() function and reads the coordinates if the UART Serial port has bytes available to read in it, shown overall in figure 32.

Figure 32: Arrangement of position gathering within the synchronous part of the code.



```
  int i = 0;
  if (!client.connected()) {
    reconnect();
  }
  requestPosData();
  if(Serial2.available() > 0) {
    getPositionData();
  }
  client.loop();
  /*char msg[50];
  snprintf (msg, 50, "%d ",i);
  client.publish("Return_Topic", msg);
  i = i + 1;
  */
  delay(250);
}
```

Reflections and Bugs from Control Version 2:

Version 2 overall provided a more sophisticated array of functionality to the Control subsystem and to the rover proper, especially achieved in acquiring coordinate data, which can later be used to drive more sophisticated algorithms in the next overall version. The use of validation in the second system allows for a level of security and integrity in design. Testing this on the rover provided a high degree of confidence in its manual motion and overall manual control. However, to realise the full spectrum of specification objectives, this would need to be reworked heavily to support larger abstractions, mainly because the actions the rover takes are entirely determined and acted upon in the callback function, meaning that higher level operations will not be able to use them. In the 2nd version of Control (Operation), the means to move, rotate and stop was implemented, this combined with position tracking presented a sizeable opportunity for the facilitation of local automation. By creating a means of abstraction, a function can be designed to move to a coordinate automatically, without needing to intervene at all during the process. Further abstraction can create near complete autonomy from the Control system's perspective and will be the primary objective of the final version of the Control subsystem.

Vision data will be processed in a similar manner to the Positional data from the drive subsystem and processed in a way to produce a means of colour distinction, and position tracking of objects in the rover's surrounding area, with a key emphasis on autonomous intervention by self-manoeuvring when the rover is too close to a given object.

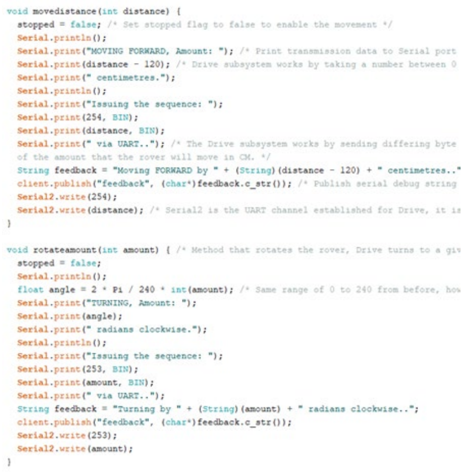
<u>Third Version of the Control subsystem (Control V3: AUTOMATION):</u>

As mentioned above and from the name, the third version of the Control subsystem built heavily upon the fundamental movement operations designed and validated in the first two versions, and developed a basis for automation and observation, enabling the rover to not only be manually operated but also make acting decisions, and enabled for a degree of self-deterministic actions, with a minimal user interaction to achieve precise results.

A key issue that was observed and noted from the second version was the lack of versatility in operation. Due to the callback function entailing the list of possible operation, and with no operations defined outside of it, this resulted in a lack of flexibility in the context of real-time operations, as MQTT is an asynchronous networking protocol. This mandated a major reorganisation of the fundamental operations of the rover to create more complex functionality.

Due to additions to the drive software, intended for tighter precision of angle turning, the range of angle positioning was reduced from 0 to 240; the movement distance was also reduced from 0 to 120, but was still called in the range 0 to 240. This was because the Drive subsystem had later implemented a reversing operation, which meant that entries X between 0 to 120 would reverse a distance of (120 – X) centimetres, and numbers between 120-240 would move forward by (X-120) centimetres, allowing for a new operational method. Demonstrated in example figure 33, external methods were devised as replacements to the operations originally in callback, such as moving forward and rotating, this allowed for the same functionality as calling the methods simply wrote the appropriate byte to Serial2 (Drive) via UART and passing a parameter as the appropriate operation amount, which is already pre-validated in callback. Overall, by organising the code in this way, only does this enable code readability in the main callback function, but this means that these functions can be called in other areas of the program, allowing for layers of instructions to be stacked on top of each other, which is a step towards further autonomy.

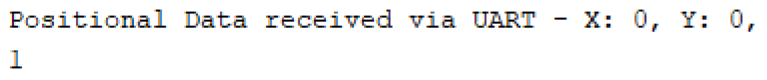
Figure 33: Organising movement actions into methods.



```
void movedistance(int distance) {
  stopped = false; /* Set stopped flag to false to enable the movement */
  Serial.println();
  Serial.print("MOVING FORWARD, Amount: "); /* Print transmission data to Serial port
  Serial.print(distance - 120); /* Drive subsystem works by taking a number between 0
  Serial.print(" centimetres.");
  Serial.println();
  Serial.print("Issuing the sequence: ");
  Serial.print(254, BIN);
  Serial.print(distance, BIN);
  Serial.print(" via UART.."); /* The Drive subsystem works by sending differing byte
  of the amount that the rover will move in CM. */
  String feedback = "Moving FORWARD by " + (String)(distance - 120) + " centimetres.."
  client.publish("feedback", (char*)feedback.c_str()); /* Publish serial debug string
  Serial2.write(254);
  Serial2.write(distance); /* Serial2 is the UART channel established for Drive, it is
}

void rotateamount(int amount) { /* Method that rotates the rover, Drive turns to a giv
  stopped = false;
  Serial.println();
  float angle = 2 * Pi / 240 * int(amount); /* Same range of 0 to 240 from before, how
  Serial.print("TURNING, Amount: ");
  Serial.print(angle);
  Serial.print(" radians clockwise.");
  Serial.println();
  Serial.print("Issuing the sequence: ");
  Serial.print(253, BIN);
  Serial.print(amount, BIN);
  Serial.print(" via UART..");
  String feedback = "Turning by " + (String)(amount) + " radians clockwise..";
  client.publish("feedback", (char*)feedback.c_str());
  Serial2.write(253);
  Serial2.write(amount);
}
```

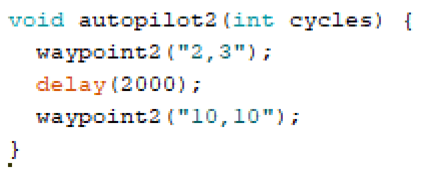
Testing two instructions sequentially after the other caused a realisation of conflicting instructions being a caveat of rover movement. The Drive subsystem is designed in such a way that to realise instructions completely, instructions issued during prior ones are discarded, this caused the rover to either discard the second instruction in the loop, or to simply conflict with the instruction and not move at all. A means of debugging was devised in which the Drive subsystem would additionally send among position data a Boolean value that would be active high when the rover is ready for a new movement instruction, and low when the rover is already in operation. By observing in the debug monitor when this is gathered, as seen in figure 34, the team was able to understand the timings of most instructions and proceed with an idea to wait for one instruction to complete before producing the next one. The first idea for performing this was initially by using a delay function, by performing a delay of a few seconds between instructions, shown in figure 35, this provided a means of halting further operation before issuing the following instruction, and this method would successfully issue consecutive instructions.

Figure 34: Displaying a boolean value at the bottom which is active high when the rover is stationary and ready to operate.



```
Positional Data received via UART - X: 0, Y: 0,
1
```

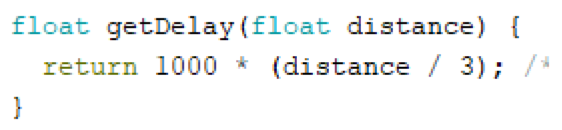
Figure 35: A primitive test function to test sequential movement.



```
void autopilot2(int cycles) {
  waypoint2("2,3");
  delay(2000);
  waypoint2("10,10");
}
```

Further care to timing was adapted in figure 36, where a linear delay method is generated. This method allowed for the right amount of delay time to be consistently added to sequential movement functions, which was done by considering the average speed of the rover during motion (approximately 1.5cm/s), and shifting it by a further dividing constant, this would then take a distance and multiply it by this and then by 1000 to get the ideal delay time in seconds. Calling this would enable a delay that would be long enough to not cause an instruction conflict but would be short enough to not cause the rover to wait for extremely long periods of time before the next motion.

Figure 36: A linear delay function which scales with distance travelled.



```
float getDelay(float distance) {
  return 1000 * (distance / 3); /*
}
```

Further usage of this function however unearthed a major flaw to this design style. As the delay method in Arduino halts the entire code for the duration of the delay, this not only meant that the rover would be unable to respond to other methods that require constant use, such as positional data, but in the larger context of operation would mean that during autonomous actioning, data would be generated at a greatly reduced role, and future obstacle detection would not be able to be called, meaning that an obstacle would not be detected until the rover has already crashed into it. However, the delay function can be entirely phased out of the code by thoughtful use of the millis() timing method included in the Arduino's Chrono sub-section of the standard library.

By default, millis() returns a value which displays the time in milliseconds that the ESP32 has been powered on, however by placing such a feature in void loop, which consistently updates the value of millis(), counting can be done effectively by saving a time reference and updating millis() until 5000 milliseconds, or 5 seconds has elapsed. Embedding operations in an if statement comparing the times allows for operations to be effectively halted in the process of other operations, while being able to do key features such as gathering position data.

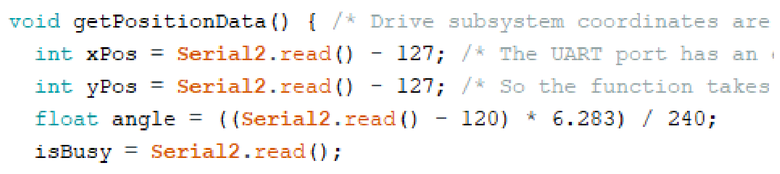
The next level of abstraction was to design an algorithm that would allow the rover to independently move to a given coordinate, which will be the sole interaction from the user. The algorithm itself required the comparative use of the rover's current coordinates with the destination coordinates, using constant position tracking as an indication of correct motion,

The naïve implementation of such an algorithm would be to simply direct the rover forward in the magnitude of the difference in Y coordinates (between the current position and the target position), and subsequently turn 90 degrees in either clockwise or anticlockwise orientations to move the magnitude of the difference in X coordinates. Not only would this be tedious to realise, given that the rover must be initially aligned to be facing forward each time it does this, but there would need to be separate cases for all 4 quadrants of a general X,Y plane, which would be inefficient to design and write. More notably, such an implementation would not necessarily reflect a 'smart' waypoint movement system that could demonstrate precise rover-determined movements, as it would be a basic operation in nature.

A more consistent and streamlined way to perform waypoint movement would be to take the rover's current coordinates and the target coordinates, and subtend both points to a right angled triangle in the X-Y plane, this would effectively suspend the triangle by a given angle, the rover calculates this angle by determining the inverse tangent of the ratio of differences in value, while calculating the hypotenuse of the aforementioned triangle by using Pythagoras' theorem, and moving in that magnitude towards the point. This not only provided a further detailed and consistent method of moving to a point in space, but was also far more efficient in comparison, as the only two use cases would be dependent on whether the rover was moving towards the left or right half of the plane, shifting the argument of the angle by Pi and ensuring it is within range of the Drive subsystem.

Some changes to the position data gathering algorithm needed to be changed to realise this method, as seen in figure 37. Most notably, the 3rd byte of the UART transmission from the Drive subsystem would give the angle subtended from the reference position, with the 4th byte adjusted to transmit the stationary Boolean flag. A global array instantiated to 0,0 would be declared, which tracks the current coordinates of the rover every 250ms. The array values are updated every time the method is called, which allows for use of the rover's current coordinates in other functional uses in the system.

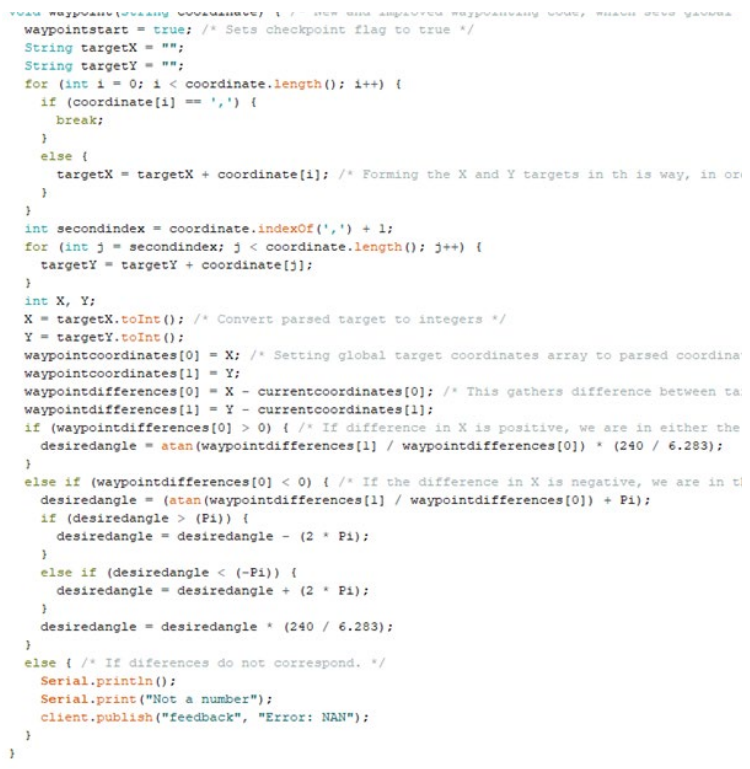
Figure 37: Adjusted position gathering function.



```
void getPositionData() { /* Drive subsystem coordinates are
  int xPos = Serial2.read() - 127; /* The UART port has an
  int yPos = Serial2.read() - 127; /* So the function takes
  float angle = ((Serial2.read() - 120) * 6.283) / 240;
  isBusy = Serial2.read();
```

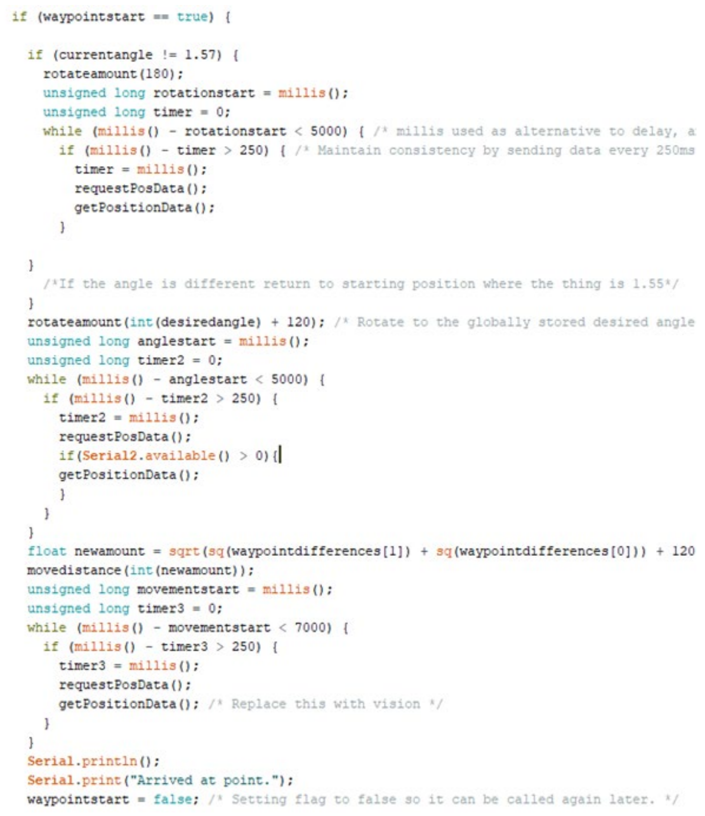
Within callback, the user may type WAYPOINT X,Y to move the rover to point X,Y, which simply calls the way-pointing method, detailed in figure 38, which appends the desired coordinates to a global array, as well as appending the differences to effectively 'form' the lengths of the right-angled triangle in space, ultimately calculating the angle by referencing the differences formed and calculating their arctangent, then setting a Boolean flag to true. Figure 39 describes the final operations conducted by the rover to move to the point. The Boolean flag is checked against in void loop() every 250ms, and when detected as true, will immediately cause the rover to rotate to its initial orientation (by checking against the reset orientation, which is 1.57 radians), and rotating this amount to the starting angle position if detected as a different angle. Following this it will take the desired subtended angle, stored globally from the waypoint method, and rotate the rover to this angle position. Finally, it will then consider the globally appended differences in Y and X to generate the hypotenuse of the triangle, which allows it to travel in the shortest overall distance to reach the target. During the whole process, the rover is also fetching and updating position data at the same frequency during waiting compared to manual operation, providing a constant stream of coordinate and tracking data to Command.

Figure 38: Waypoint method



```
void waypoint(String coordinate) { /* New and improved waypointing code, which sets global
  waypointstart = true; /* Sets checkpoint flag to true */
  String targetX = "";
  String targetY = "";
  for (int i = 0; i < coordinate.length(); i++) {
    if (coordinate[i] == ',') {
      break;
    }
    else {
      targetX = targetX + coordinate[i]; /* Forming the X and Y targets in th is way, in or
    }
  }
  int secondindex = coordinate.indexOf(',') + 1;
  for (int j = secondindex; j < coordinate.length(); j++) {
    targetY = targetY + coordinate[j];
  }
  int X, Y;
  X = targetX.toInt(); /* Convert parsed target to integers */
  Y = targetY.toInt();
  waypointcoordinates[0] = X; /* Setting global target coordinates array to parsed coordina
  waypointcoordinates[1] = Y;
  waypointdifferences[0] = X - currentcoordinates[0]; /* This gathers difference between ta
  waypointdifferences[1] = Y - currentcoordinates[1];
  if (waypointdifferences[0] > 0) { /* If difference in X is positive, we are in either the
    desiredangle = atan(waypointdifferences[1] / waypointdifferences[0]) * (240 / 6.283);
  }
  else if (waypointdifferences[0] < 0) { /* If the difference in X is negative, we are in t
    desiredangle = (atan(waypointdifferences[1] / waypointdifferences[0]) + Pi);
    if (desiredangle > (Pi)) {
      desiredangle = desiredangle - (2 * Pi);
    }
    else if (desiredangle < (-Pi)) {
      desiredangle = desiredangle + (2 * Pi);
    }
    desiredangle = desiredangle * (240 / 6.283);
  }
  else { /* If diferences do not correspond. */
    Serial.println();
    Serial.print("Not a number");
    client.publish("feedback", "Error: NAN");
  }
}
```

Figure 39: Waypoint execution within synchronous void loop() section

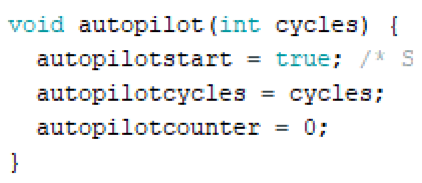


```
  if (waypointstart == true) {

    if (currentangle != 1.57) {
      rotateamount(180);
      unsigned long rotationstart = millis();
      unsigned long timer = 0;
      while (millis() - rotationstart < 5000) { /* millis used as alternative to delay, a
        if (millis() - timer > 250) { /* Maintain consistency by sending data every 250ms
          timer = millis();
          requestPosData();
          getPositionData();
        }

    }
      /*If the angle is different return to starting position where the thing is 1.55*/
    }
    rotateamount(int(desiredangle) + 120); /* Rotate to the globally stored desired angle
    unsigned long anglestart = millis();
    unsigned long timer2 = 0;
    while (millis() - anglestart < 5000) {
      if (millis() - timer2 > 250) {
        timer2 = millis();
        requestPosData();
        if(Serial2.available() > 0){
        getPositionData();
        }
      }
    }
    float newamount = sqrt(sq(waypointdifferences[1]) + sq(waypointdifferences[0])) + 120
    movedistance(int(newamount));
    unsigned long movementstart = millis();
    unsigned long timer3 = 0;
    while (millis() - movementstart < 7000) {
      if (millis() - timer3 > 250) {
        timer3 = millis();
        requestPosData();
        getPositionData(); /* Replace this with vision */
      }
    }
    Serial.println();
    Serial.print("Arrived at point.");
    waypointstart = false; /* Setting flag to false so it can be called again later. */
```

Producing the final major autonomous function, AUTOPILOT X, would allow the user to travel to X different coordinates through X cycles of operation, abstracting by using the waypoint function developed earlier to enhance the level of self-operation of the rover. The reasoning behind using a fixed cycle count determined by the user is that due to the asynchronous nature of the MQTT protocol, calling a function in this manner will block other payloads from being read by the ESP32 during operation, meaning that it could theoretically never stop moving. This is not ideal as there may be instances in the use-case where the user may want to manually make alterations to the orientation of the rover, or investigate proximity objects of interest, so having a fixed number of cycles allows the user to define their own automatic length of operation, and ensures a natural stop to the instruction, so that following ones can then be processed.

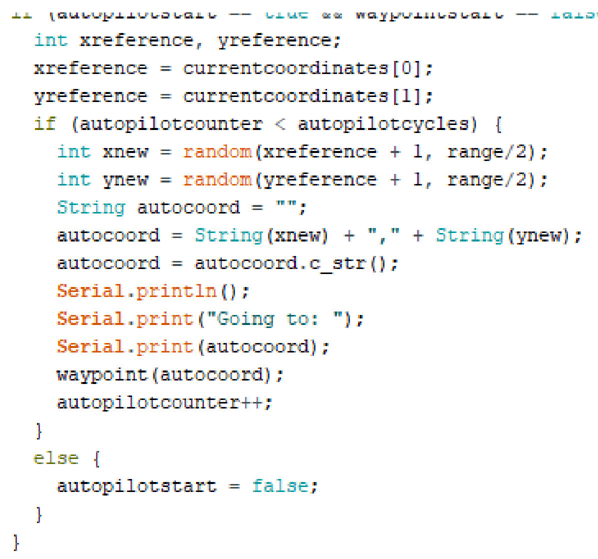
The implementation of the function is shown in figure 40 and is similarly designed to the way-pointing algorithm. When called in callback, the method called will simply pass in the number of cycles and update a global variable, (allowing it to be used in void loop() given the synchronous need of the operation), as well as resetting a global counter and setting an autopilot flag to true. Figure 41 depicts the void loop() addition, as checking against the value of the flag allows the system to act accordingly. A global value of the maximum range of autonomous movement is fixed in code, this value is set to 100 to meet the constraints of the Integration testing environment but can be adjusted freely. Random coordinates are generated by taking a range between the user's current position up to the maximum range divided by a constant, and an iterative statement allows for way-pointing to those random coordinates (taken forwards from the rover), until the required number of cycles has elapsed. When tested, this worked as intended and navigated to the random coordinates with a notable degree of accuracy, which will be later reflected on.

Figure 40: Autopilot callback method



```
void autopilot(int cycles) {
  autopilotstart = true; /* S
  autopilotcycles = cycles;
  autopilotcounter = 0;
}
```

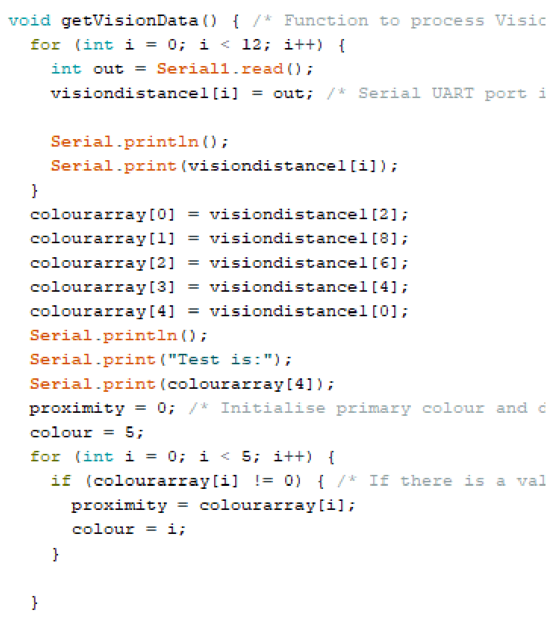
Figure 41: Autopilot synchronous execution within void loop().



```
  if (autopilotstart == true && waypointstart == false) {
    int xreference, yreference;
    xreference = currentcoordinates[0];
    yreference = currentcoordinates[1];
    if (autopilotcounter < autopilotcycles) {
      int xnew = random(xreference + 1, range/2);
      int ynew = random(yreference + 1, range/2);
      String autocoord = "";
      autocoord = String(xnew) + "," + String(ynew);
      autocoord = autocoord.c_str();
      Serial.println();
      Serial.print("Going to: ");
      Serial.print(autocoord);
      waypoint(autocoord);
      autopilotcounter++;
    }
    else {
      autopilotstart = false;
    }
  }
```

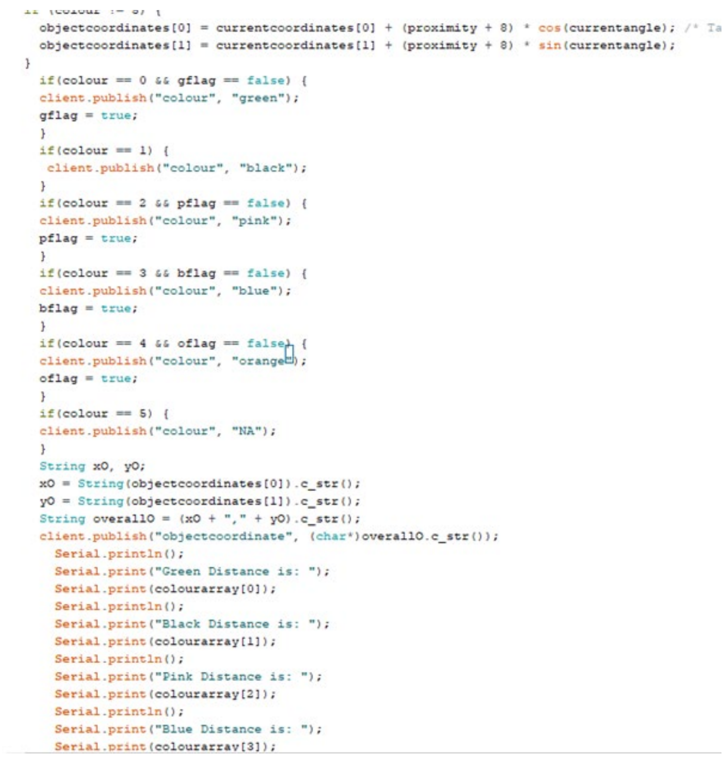
The final implementation would be to process Vision data from the FPGA, to derive obstacle tracking and colour detection. The Vision subsystem communicates colour data as a 96-bit binary number, with 16-bit fields in fixed order representing the distance to the bounding box of a given colour, in the following order: Green, Black, Blue, Pink, Orange. The last 16-bit number is fixed to 0, to indicate the end of the data. The data is fixed to one colour, meaning that when one colour is detected, all other colours are set to 0, reducing the local processing required. Interpreting the data can be processed in a similar fashion to the autonomous motion functions by storing the data in a global array and appending sequentially the 12 bytes of the UART port used by Drive to send data in, ensuring the port is read in its entirely with each transmission and not reading in prior bits. The overall implementation of the Vision communication is visualised in Figures 42 and 43.

Figure 42: getVisionData Function

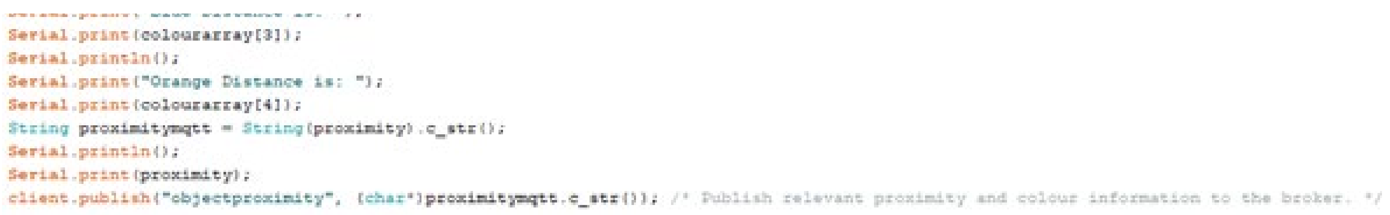


```
void getVisionData() { /* Function to process Visic
  for (int i = 0; i < 12; i++) {
    int out = Serial1.read();
    visiondistance1[i] = out; /* Serial UART port i

    Serial.println();
    Serial.print(visiondistance1[i]);
  }
  colourarray[0] = visiondistance1[2];
  colourarray[1] = visiondistance1[8];
  colourarray[2] = visiondistance1[6];
  colourarray[3] = visiondistance1[4];
  colourarray[4] = visiondistance1[0];
  Serial.println();
  Serial.print("Test is:");
  Serial.print(colourarray[4]);
  proximity = 0; /* Initialise primary colour and c
  colour = 5;
  for (int i = 0; i < 5; i++) {
    if (colourarray[i] != 0) { /* If there is a val
      proximity = colourarray[i];
      colour = i;
    }

  }
```

Figure 43: Implementation of Colour Detection



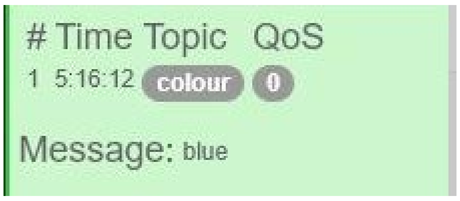
```
  if (colour != 5) {
    objectcoordinates[0] = currentcoordinates[0] + (proximity + 8) * cos(currentangle); /* Ta
    objectcoordinates[1] = currentcoordinates[1] + (proximity + 8) * sin(currentangle);
  }
    if(colour == 0 && gflag == false) {
    client.publish("colour", "green");
    gflag = true;
    }
    if(colour == 1) {
     client.publish("colour", "black");
    }
    if(colour == 2 && pflag == false) {
    client.publish("colour", "pink");
    pflag = true;
    }
    if(colour == 3 && bflag == false) {
    client.publish("colour", "blue");
    bflag = true;
    }
    if(colour == 4 && oflag == false) {
    client.publish("colour", "orange");
    oflag = true;
    }
    if(colour == 5) {
    client.publish("colour", "NA");
    }
    String xO, yO;
    xO = String(objectcoordinates[0]).c_str();
    yO = String(objectcoordinates[1]).c_str();
    String overallO = (xO + "," + yO).c_str();
    client.publish("objectcoordinate", (char*)overallO.c_str());
      Serial.println();
      Serial.print("Green Distance is: ");
      Serial.print(colourarray[0]);
      Serial.println();
      Serial.print("Black Distance is: ");
      Serial.print(colourarray[1]);
      Serial.println();
      Serial.print("Pink Distance is: ");
      Serial.print(colourarray[2]);
      Serial.println();
      Serial.print("Blue Distance is: ");
      Serial.print(colourarray[3]);
```



```
Serial.print(colourarray[3]);
Serial.println();
Serial.print("Orange Distance is: ");
Serial.print(colourarray[4]);
String proximitymqtt = String(proximity).c_str();
Serial.println();
Serial.print(proximity);
client.publish("objectproximity", (char*)proximitymqtt.c_str()); /* Publish relevant proximity and colour information to the broker. */
```

The data is then appended to a smaller array to store the values to a similarly declared 5 valued array of colour values, these are indexed in the same order that data is read in, indexed to align with the bytes from the FPGA. The colour indexes are sorted and publish to Command depending on when the colour of the object changes, as well as sending the proximity to the object in centimetres, as well as calculating the approximate coordinates of the object by taking the proximity (offset by the distance from the camera to the rover's centre) and adding on the vertical and horizontal coordinates formed by the current angle of the rover, generating a rough measure of positioning relative to its current position. Local filtering was required in order to reduce lag to the Command terminal, as driving towards the same object will produce an arc of coordinates, which are distinct approximations of the same point, hence, the coordinate sent would be the first coordinate generated, and any points subsequently generated that are within a marginal difference of the original coordinates are ignored. All 3 of these values are published to Command regularly. Figures 44, 45 and 46 show the rover correctly identifying distances as well as colours and calculated coordinates of objects, updating the broker's colour topic to 'blue' to correctly identify the ball, and subsequently identifying the distance from an orange ball.

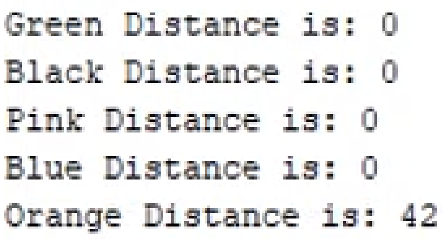
Figure 44: Colour recognition of blue ball



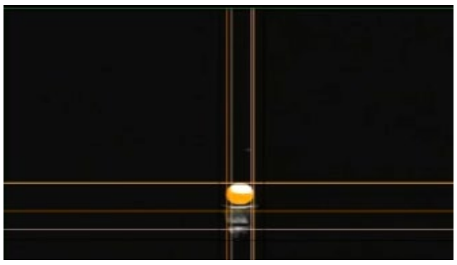
Time Topic QoS
1 5:16:12 colour 0

Message: blue

Figures 45 and 46: Distance and colour recognition of orange ball.

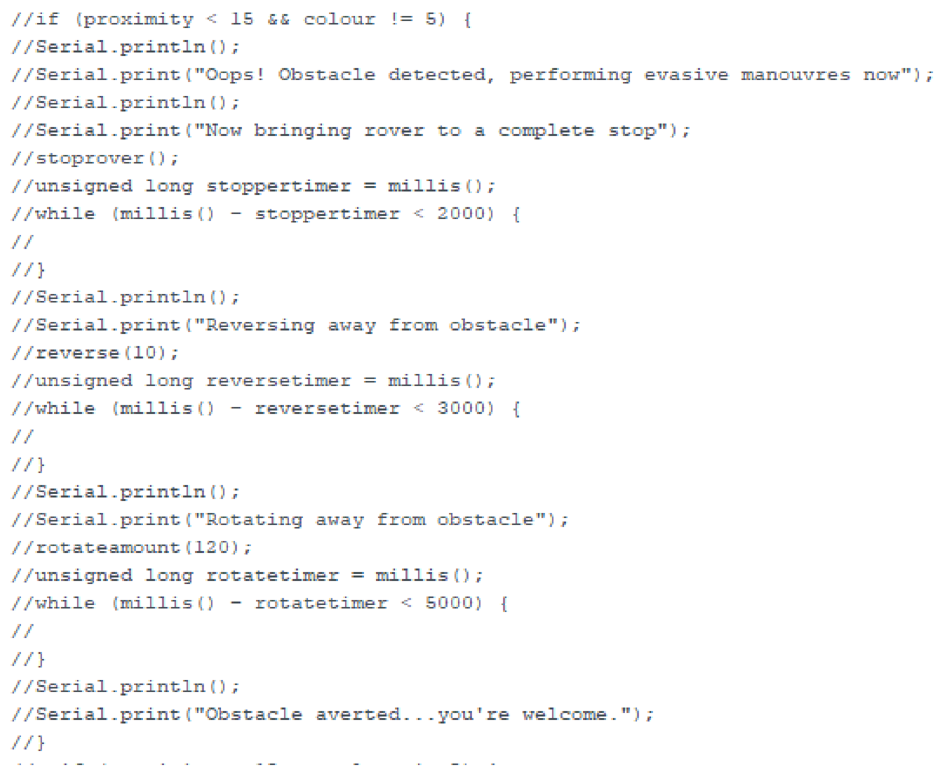


```
Green Distance is: 0
Black Distance is: 0
Pink Distance is: 0
Blue Distance is: 0
Orange Distance is: 42
```



The final implementation to the Control subsystem would utilise the Vision subsystem in such a way to ensure rover security, by performing evasive manoeuvres when the proximity is below a certain threshold. By using the same principle of sequential instructions when an object is detected and possesses a proximity below a certain threshold, it will initially stop the rover automatically, reverse it by a fixed distance and rotate so that the rover is facing away from the obstacle, then allowing the user to issue further instructions. Testing this functionality on the overall rover was successful and provided consistent results, as while there was inconsistency with colour detection, the object's proximity was correctly identified regardless of this, and evasive manoeuvres were able to be undertaken, with the overall code from the explanation demonstrated in figure 47.

Figure 47: Autonomous distance recognition and evasive manoeuvring code.



```
//if (proximity < 15 && colour != 5) {
//Serial.println();
//Serial.print("Oops! Obstacle detected, performing evasive manouvres now");
//Serial.println();
//Serial.print("Now bringing rover to a complete stop");
//stoprover();
//unsigned long stoppertimer = millis();
//while (millis() - stoppertimer < 2000) {
//
//}
//Serial.println();
//Serial.print("Reversing away from obstacle");
//reverse(10);
//unsigned long reversetimer = millis();
//while (millis() - reversetimer < 3000) {
//
//}
//Serial.println();
//Serial.print("Rotating away from obstacle");
//rotateamount(120);
//unsigned long rotatetimer = millis();
//while (millis() - rotatetimer < 5000) {
//
//}
//Serial.println();
//Serial.print("Obstacle averted...you're welcome.");
//}
```

Interfacing with Command would be the final tweak to the Control subsystem, by considering the Serial feedback generated by the program during operation and automatically publishing this to the Command subsystem under the topic of 'feedback', providing Command with popups issuing the relevant information and parameters of movement as operation proceeds.

Final reflections of the Control subsystem:

The third and final version of the Control subsystem provided it with the most expansive and up to date functionality compared to any previous version, however, was not short of minor bugs that could be refined in a newer version if time had allowed. Namely the bugs entailed an occasional incorrect coordinate value to be issued, namely due to integer overflow and underflow where a coordinate would display as -128,-128, or 127,-128 and so on. These bugs were for the most part filtered out manually in the position acquisition function, where these values were reduced to 0 (primarily as they would appear at the start of the rover's initialisation, and most likely due to values from the UART terminal not being flushed at the start of use, causing the ESP32 to read invalid bytes as coordinates), but this could have been further refined to setting the glitched value to be the current position, which was already stored globally. Further debugging with more time would have enabled the system to filter out all possible overflow or underflow glitches, but this issue was highly infrequent and by communicating with the Command subsystem, was filtered out in that module by simply ignoring the specific coordinate during mapping.

Bytes from the vision subsystem can at occasions be misaligned, due to a slight frequency difference in sending and receiving with both subsystems, causing a byte to be read in the wrong index and thus detect an incorrect colour. With further time this frequency can be matched to ensure the correct colour ball is detected; however, despite this, the correct distance is consistently recognised allowing autonomous intervention to work regardless of the detected colour, meaning that valuable data has still been extracted.

Another limitation of the Control subsystem is the sensitivity to changes of terrain and environment, namely with regards to the wiring of the rover. The testing conditions of the rover meant that multiple USB ports and wires were in use at any given time, meaning large scale rotations were likely to tangle the wires, as well as catch underneath the rover, obstructing not only the optical flow sensor used by drive but also the wheels, meaning that operations are halted until the tangling is resolved, and then incorrectly perceiving the coordinates and moving to the wrong waypoint, or just being in a permanent operational (Boolean for stationary being permanent 0) loop of performing an operation and not being responsive. This issue is rare to encounter during use but the Control code with more time could include further error checks and flag issuing, by using the currently established coordinate arrays to check if an excess distance has been travelled or if a certain amount of time has elapsed that is beyond the 'usual' (calculated in the getDelay linear delay function) time of operation, this could revert the action of the rover or move it to its start position, cancelling further instructions. A final addition that Control could include would be a password authentication mechanism before allowing the user to issue commands, allowing for greater security, particularly given the remote nature of the rover's control.

A final validation fix that was developed for AUTOPILOT and WAYPOINT but lacked time for adequate testing. Due to flooring and tangling issues, or general blocking of the optical flow sensor, the rover may deviate from its way-pointed coordinate, as a means of manual intervention, a simple solution to this was to embed a condition in the main millis() loop, which checked after the expected period of time whether the rover had deviated over a given threshold from its intended coordinates. Instead of completing its journey, the rover will automatically detect this and set any autonomous function flags to false, and call the stoprover() method immediately, breaking all loops and returning control to the operator.

Regarding specific objectives met – the rover can receive commands from the Command subsystem, act upon them after local processing with minimal lag, coordinate with the Drive subsystem and also receive coordinate data back, as well as conduct both manual and autonomous operation. Satisfying completely objectives 1,2,3,4,5,6 and 9. Computer vision was able to identify colours and read proximities, as well as act autonomously based on them and map approximations of their location, albeit with bugs meaning that this was not entirely consistent, however this partially completed objectives 7, 8 and 10. Satisfying these objectives also corresponded with satisfying the Control based main specification points laid out for the general rover system.

However overall, all the initial subsystem objectives were met by the final Control subsystem. By accurately and seamlessly receiving instructions via a remote network protocol, acting upon them in real time and also communicating with multiple hardware systems simultaneously, as well as delivering a significant amount of autonomy to the user in balance with traditional manual commands, as well as adequately utilising the computer vision provided by the D8M camera, the system successfully conducts the operation of the rover to not only meet a high level of performance-based precision, but with the use-case kept in mind, allowing the user to issue accurate and quick-acting instructions without causing significant glitches or undefined behaviour.

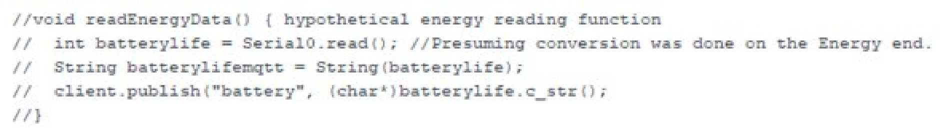
Control and Energy interfacing

Regarding the connection of the Control subsystem to the Energy subsystem, due to the regulations and circumstances of COVID-19, the integration member was not sent an Energy kit for testing, meaning that the rover draws its power from the USB connected to the main computer. For this reason, there was a slight difference in how this was designed and implemented in Control, however the Control subsystem has been designed in such a way that if the Energy subsystem is available to integrate, accommodating it would be seamless and would not compromise any key functionality, allowing for more integration to be added if wanted.

The initial design plan was to use the 3rd and final UART instance (TX0 and RX0 specifically) on the ESP32 to specifically interface with the Energy subsystem, to extract battery life and range values using the same principle as how Drive sends coordinates, by periodically sending data requests and offsetting the responses to gather the values, and then publishing this to Command via MQTT. Helper functions that are commented in the code, as seen in Figure 48, demonstrate the feasibility of this, by simply reading the next byte from the Serial UART port and publishing the number. The 3rd UART instance for the sake of testing has been assigned to debugging on the main console on the computer via USB, which simply registers the action the user has taken, and any errors that arise.

Since information data is already stored on a MongoDB database on the Command subsystem, as well as error data being transmitted back in real time in parallel to the debugging window, this means that all the user has to do to integrate Energy is uncomment the lines which define new RX and TX pins, specifically chosen to connect to Energy, as well adding a brief line to reassign the pins for Serial0 to the macro defined Energy pins, meaning that Serial0 now communicates with Energy, upon uncommenting the helper methods and simply deleting the debug lines, Energy can be integrated immediately into the rest of the subsystem, and no rover functionality is compromised at all.

Figure 48: Hypothetical energy reading function



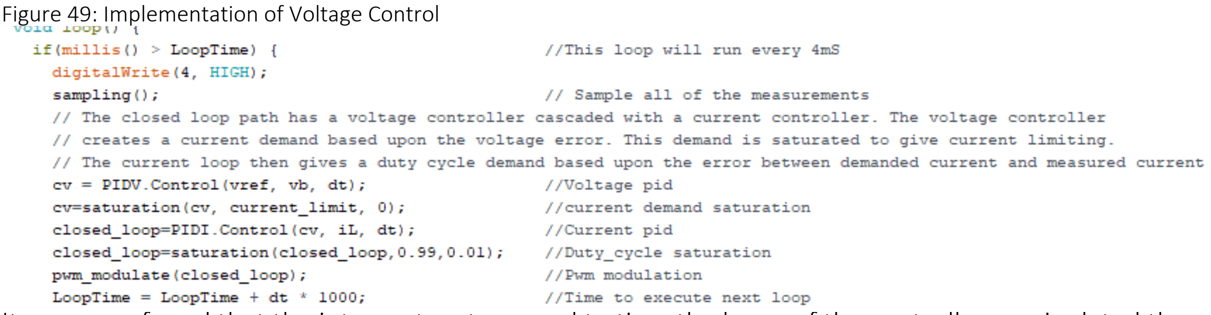
```
//void readEnergyData() { hypothetical energy reading function
//  int batterylife = Serial0.read(); //Presuming conversion was done on the Energy end.
//  String batterylifemqtt = String(batterylife);
//  client.publish("battery", (char*)batterylife.c_str();
//}
```

Drive

Power Supply

Firstly, the example power supply code was modified. The unnecessary parts of the program were removed, such as the sections to determine if the SMPS was configured to open loop or closed loop, or if it should behave as a buck or a boost. The parts that remained were the functions to sample the sensors and the section of void loop to implement closed loop control of the output voltage, as shown in figure 49. The PID controllers were replaced with the generalised PID functions explained in the PID library section.

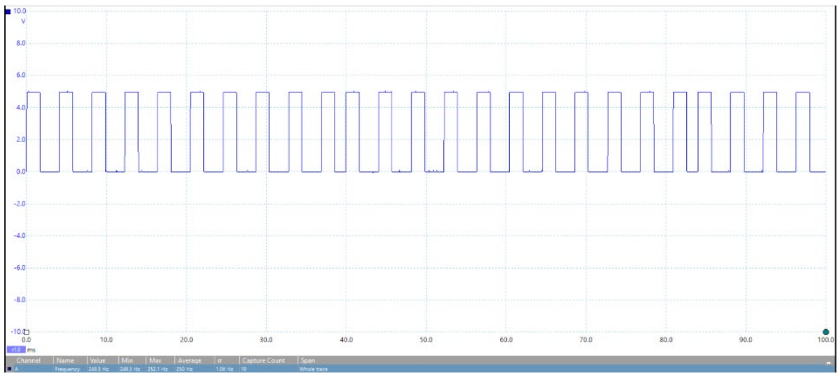
Figure 49: Implementation of Voltage Control



```
void loop() {
  if(millis() > LoopTime) {                    //This loop will run every 4mS
    digitalWrite(4, HIGH);
    sampling();                                // Sample all of the measurements
    // The closed loop path has a voltage controller cascaded with a current controller. The voltage controller
    // creates a current demand based upon the voltage error. This demand is saturated to give current limiting.
    // The current loop then gives a duty cycle demand based upon the error between demanded current and measured current
    cv = PIDV.Control(vref, vb, dt);           //Voltage pid
    cv=saturation(cv, current_limit, 0);       //current demand saturation
    closed_loop=PIDI.Control(cv, iL, dt);      //Current pid
    closed_loop=saturation(closed_loop,0.99,0.01);  //Duty_cycle saturation
    pwm_modulate(closed_loop);                 //Pwm modulation
    LoopTime = LoopTime + dt * 1000;           //Time to execute next loop
```

It was soon found that the interrupt system used to time the loops of the controller manipulated the registers used for timing [10] which were also used by the Arduino PWM modulation functions, resulting in a 25% duty cycle as the maximum that could be controlled. Furthermore, as the clock signal that caused the timers to increase was divided by a 16 as opposed to 64 which was the default value, the millis() function no longer accurately returned the time since the program started in milliseconds. Several viable solutions were considered. The registers could have been manipulated in such a way to retain the functionality of the interrupt service routine without affecting the PWM functions, however this proved to be complicated. Ultimately the decision was made to remove modifications to the registers of timer A so that PWM and millis() could be used as normal, and the modifications to timer B were retained so that the PWM for pin 6, the pin that controlled the gates of the MOSFETs of the SMPS, still operated at 62.5kHz. To time the controller for the SMPS, the millis() function was employed so that the segment of code controlling the SMPS would run only if the current time exceeded the time of the previous loop after the desired time period had been added. The frequency was set at 250 Hz, which can be seen from figure 50, where an unused pin was set to 5V at the start of the loop and set to 0V at the end. Later this method was used to time the entire drive void loop function so that it operated with a predictable time period. The P values of the PID controllers had to be increased to 0.2 to increase the rate of response due to new lower frequency, but a response acceptable for testing purposes was found. If the rover were to be implemented, the energy subsystem would provide a better regulated power supply.

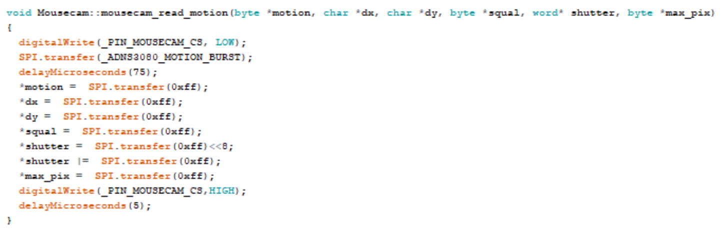
Figure 50 :Void Loop at 250Hz



Mouse Camera Library

The mouse camera example program produced by Dr Zohaib Akhtar contained several functions for initialising and operating the mouse camera. These were taken from the example program and moved to a header file and a .cpp file so that they could be implemented as a library. This would keep the main.cpp file for the rover cleaner, improving readability and simplifying the functions use in future. Figure 51 shows the method mousecam_read_motion that can be used on a Mousecam object to read movement data to a struct.

Figure 51:Mousecam Read Motion Function

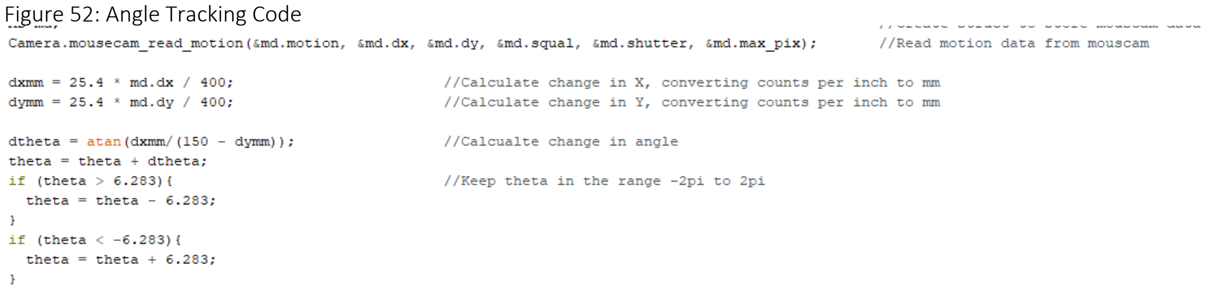


```
void Mousecam::mousecam_read_motion(byte *motion, char *dx, char *dy, byte *squal, word* shutter, byte *max_pix)
{
  digitalWrite(_PIN_MOUSECAM_CS, LOW);
  SPI.transfer(_ADNS3080_MOTION_BURST);
  delayMicroseconds(75);
  *motion =  SPI.transfer(0xff);
  *dx =  SPI.transfer(0xff);
  *dy =  SPI.transfer(0xff);
  *squal =  SPI.transfer(0xff);
  *shutter =  SPI.transfer(0xff)<<8;
  *shutter |=  SPI.transfer(0xff);
  *max_pix =  SPI.transfer(0xff);
  digitalWrite(_PIN_MOUSECAM_CS,HIGH);
  delayMicroseconds(5);
}
```

Angle Tracking

Angle tracking was implemented using the method discussed previously in initial design and specification. The rover's angle was initialised to 1.57 radians (90◦) when first switched on. In each cycle of void loop, the x and y changes to the mouse camera's position in mm were found and then the change in angle was calculated from $\Delta\theta = \arctan\left(\frac{dx}{dy-150}\right)$. This change was then added to the rover's angle, using the lines of code in figure 52. If the angle ever became larger than 2π or less than -2π then 2π would have been added or subtracted from the current angle to keep it in range without affecting the outcome of trigonometric functions for position tracking. The range -2π to 2π was chosen as initial tests of functions to control the angle when a range of $-\pi$ to π demonstrated that if the rover were instructed to turn to π then any overshoot would cause the variable storing the angle to become $-\pi$ which in turn would cause the rover to turn a complete circle in an attempt to follow the setpoint. By increasing the range of the variable that stores the angle and restricting inputs to functions controlling the angle to $\pm\frac{\pi}{2}$ so as not to cause any overflow of the range, this non ideal behaviour was eliminated.

Figure 52: Angle Tracking Code

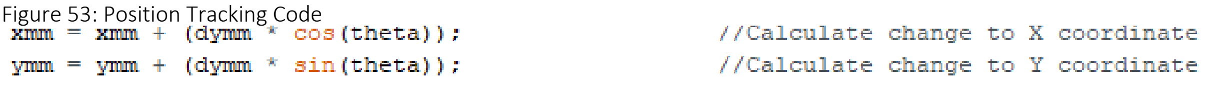


```
Camera.mousecam_read_motion(&md.motion, &md.dx, &md.dy, &md.squal, &md.shutter, &md.max_pix);      //Read motion data from mouscam

dxmm = 25.4 * md.dx / 400;                    //Calculate change in X, converting counts per inch to mm
dymm = 25.4 * md.dy / 400;                    //Calculate change in Y, converting counts per inch to mm

dtheta = atan(dxmm/(150 - dymm));             //Calcualte change in angle
theta = theta + dtheta;
if (theta > 6.283){                           //Keep theta in the range -2pi to 2pi
  theta = theta - 6.283;
}
if (theta < -6.283){
  theta = theta + 6.283;
}
```

Position Tracking

Every cycle of void loop, the change in y measured by the mouse camera (which corresponds to the rover's forwards and backwards motion) was used to calculate the change in the x and y coordinates of the rover relative to the starting position by multiplying it by $\cos(\theta)$ and $\sin(\theta)$ respectively. These were then added to the variables to store the x and y coordinates. Figure 53 shows the code to achieve this.

Figure 53: Position Tracking Code



```
xmm = xmm + (dymm * cos(theta));                  //Calculate change to X coordinate
ymm = ymm + (dymm * sin(theta));                  //Calculate change to Y coordinate
```

PID Library

The PID controller functions from the SMPS example program written by Yue Zhu were removed from the program and placed into a header file and a .cpp file and made into a PID object which could then be controlled with a new function. By generalising the PID controller, as many as required for movement control could be easily created while maintaining readability of the code. The definition of the PID Control method is shown in figure 54. It was decided that PID controllers would provide suitable closed loop control, as initial tests of the code demonstrated reasonable response times and accuracy, and they are relatively simple to manually tune. If more accuracy were to be required, then closed loop controllers could be created using the loop shaping method, however in the time scale of the project and with the available equipment, determining the open loop transfer functions of the rover's motion would have been too complex. The PID code works by taking an error signal and calculating the change in P, I and D terms from the previous iteration and summing them together to achieve integration.

Figure 54: PID Controller Function

```
float PID::Control(float input, float measout, float dt){
      e0 = input - measout;                                             //Error signal calculation
      e_integration = e0;
      T = dt;                                                           //Change in time

      //Anti-windup, if last-time pid output reaches the limitation, this time there won't be any integrations.
      if(u1 >= u_max) {
        e_integration = 0;
      } else if (u1 <= u_min) {
        e_integration = 0;
      }

      float delta_u = kp*(e0-e1) + ki*T*e_integration + kd/T*(e0-2*e1+e2);   //Incremental PID programming.
      u0 = u1 + delta_u;                                                //This time's control output

      //Output limitation
      if (u0 > u_max){
        u0 = u_max;}
      if (u0 < u_min ){
        u0 = u_min;}

      u1 = u0; //Update last time's control output
      e2 = e1; //Update last last time's error
      e1 = e0; //Update last time's error
      return u0;
    };
```

Angle Control

A new function was created to control the angle of the rover. Two PID controllers were cascaded, the inner controller to control the angular rate of the rover, and the outer one to control the angle by calculating an angular rate as an input for the inner controller. The output of the inner controller was used to PWM control the wheels in equal and opposite directions, with the direction of the wheels and therefore the rotation determined by the sign of the output of the inner controller. To prevent the rover from constantly trying to adjust itself to match the angle, an error tolerance was added. The variable ThetaFlag would store a value of 0 when the function was not in use and be made high when required. If the angle of the rover came withing the error tolerance around the desired angle, then ThetaFlag would be made 0 again, and the rover would stop moving. Information on how the PID values were obtained can be found in the testing section for drive.

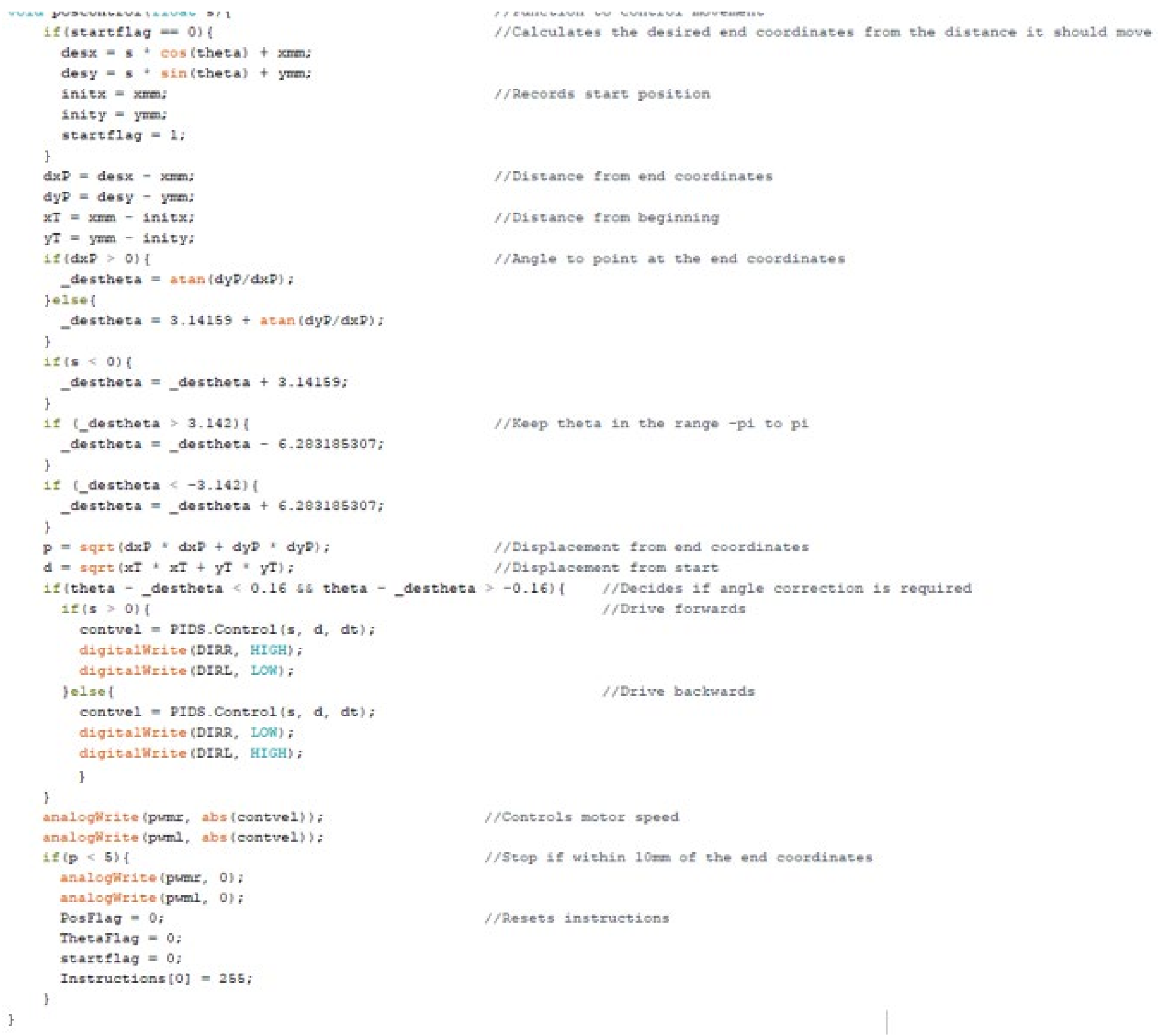
Figure 55: Angle Control Function

```
void angleControl(float destheta){                  //Function to control angle
   contangvel = PIDT.Control(destheta, theta, dt);   //PID to calculate the required angular velocity
   angvelPID = PIDW.Control(contangvel, angvel, dt); //PID to calculate the PID
   analogWrite(pwmr, abs(angvelPID));                //Write PID to motors
   analogWrite(pwml, abs(angvelPID));
 if(contangvel > 0){                                 //Control direction of wheels
   digitalWrite(DIRR, HIGH);
   digitalWrite(DIRL, HIGH);
   }
 else{
   digitalWrite(DIRR, LOW);
   digitalWrite(DIRL, LOW);
   }
 if(destheta - theta < 0.04 && destheta - theta > -0.04 ){         //Stops when theta is within 5 degrees
   analogWrite(pwmr, 0);
   analogWrite(pwml, 0);
   ThetaFlag = 0;                                                  //Resets instructions
   PosFlag = 0;
   Instructions[0] = 255;
   }
 }
```

Position Control

A function to control the rover's position was created that would allow the rover to move forwards or backwards in a straight line. When first called, the variable startflag would be set to 1, and using trigonometry, the desired x and y coordinates would be calculated from the distance to move. The direction of the wheels would be set depending on whether the rover should move forwards or backwards, and the PWM values to control the wheels would be calculated by a PID controller. Every cycle that the function ran, the angle that the rover should be pointing towards to reach its destination was calculated, and if it deviated by too much, the function would run a similar section of code to control the angle until the rovers' orientations was close to being correct. If the rover's position was within 1 cm of the desired coordinates, then PosFlag would be set to 0 and the rover would stop moving. The decision to drive forwards or backwards by a specific distance as opposed to specific coordinates was made so that the drive subsystem which does not have information on obstacles would not drive into them and instead the control system which receives information from vision could give the required commands.

Figure 56: Position Control Function

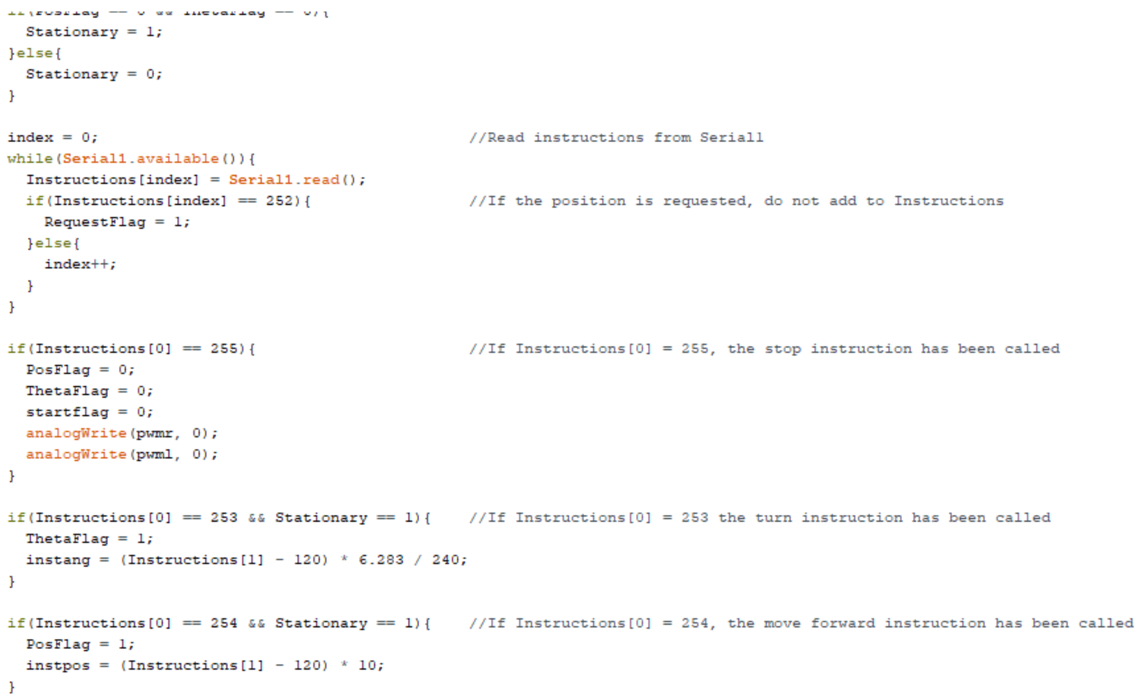


```
void poscontrol(float s){                                //Function to control movement
    if(startflag == 0){                                  //Calculates the desired end coordinates from the distance it should move
      desx = s * cos(theta) + xmm;
      desy = s * sin(theta) + ymm;
      initx = xmm;                                       //Records start position
      inity = ymm;
      startflag = 1;
    }
    dxP = desx - xmm;                                    //Distance from end coordinates
    dyP = desy - ymm;
    xT = xmm - initx;                                    //Distance from beginning
    yT = ymm - inity;
    if(dxP > 0){                                         //Angle to point at the end coordinates
      _destheta = atan(dyP/dxP);
    }else{
      _destheta = 3.14159 + atan(dyP/dxP);
    }
    if(s < 0){
      _destheta = _destheta + 3.14159;
    }
    if (_destheta > 3.142){                              //Keep theta in the range -pi to pi
      _destheta = _destheta - 6.283185307;
    }
    if (_destheta < -3.142){
      _destheta = _destheta + 6.283185307;
    }
    p = sqrt(dxP * dxP + dyP * dyP);                     //Displacement from end coordinates
    d = sqrt(xT * xT + yT * yT);                         //Displacement from start
    if(theta - _destheta < 0.16 && theta - _destheta > -0.16){     //Decides if angle correction is required
      if(s > 0){                                                   //Drive forwards
        contvel = PIDS.Control(s, d, dt);
        digitalWrite(DIRR, HIGH);
        digitalWrite(DIRL, LOW);
      }else{                                                       //Drive backwards
        contvel = PIDS.Control(s, d, dt);
        digitalWrite(DIRR, LOW);
        digitalWrite(DIRL, HIGH);
        }
    }
    analogWrite(pwmr, abs(contvel));                    //Controls motor speed
    analogWrite(pwml, abs(contvel));
    if(p < 5){                                          //Stop if within 10mm of the end coordinates
      analogWrite(pwmr, 0);
      analogWrite(pwml, 0);
      PosFlag = 0;                                      //Resets instructions
      ThetaFlag = 0;
      startflag = 0;
      Instructions[0] = 255;
    }
}
```

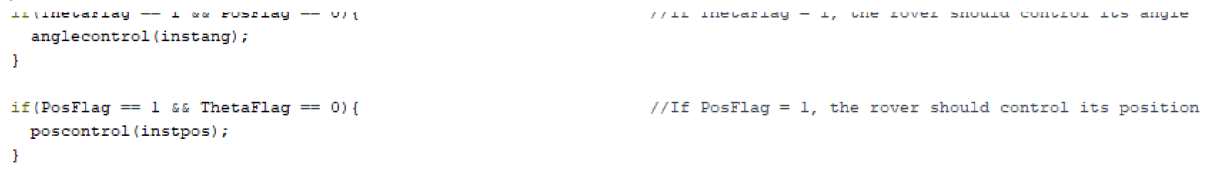
Communications with control

It was decided that the drive subsystem would communicate with the control subsystem using UART. The board provided to the team had connections to the Rx and Tx pins of the Arduino Nano Every serial port 1. Commands such as read() and write() can be used to receive and transmit single bytes over the serial connection [11]. It was agreed that the drive subsystem should be able to receive instructions to drive forwards or backwards by a specific amount, turn to face a specific angle, and stop. The drive subsystem should also send its coordinates and angle to control. To receive instructions, a system of instructions was devised. Sending a byte with value 255 would command the rover to stop. Sending a byte with value 253 would tell the rover to turn to a specific angle, which was encoded as a second byte with value $\frac{x \times 240}{2\pi} + 120$, where x is the desired angle from the x axis in radians. The offset of 120 was so that negative angles could be sent and along with the factor of 240 the value would not overlap with instruction values (see testing and debugging.) A byte value of 254 would instruct the rover to move forwards or backwards, by a number of centimetres encoded as $x + 120$, where x could take values from -120 to 120. When a command to turn or move was received, the appropriate flag would be set to 1, and the program would run the appropriate movement function accordingly as shown in figure 57.

Figure 57: Drive Interface with Control



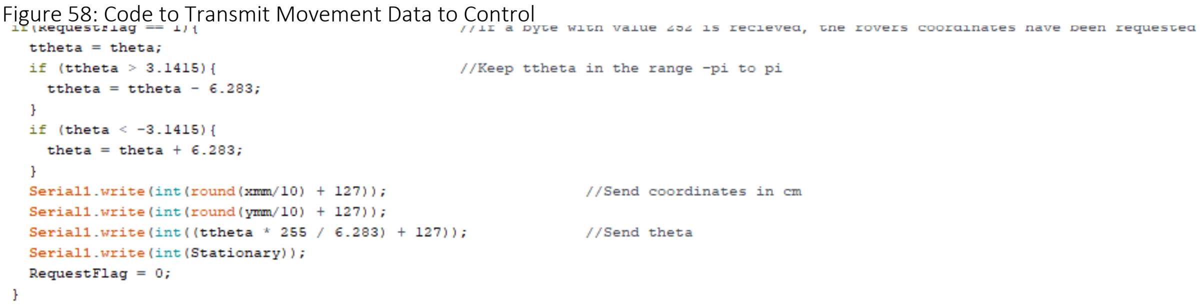
```
if(PosFlag == 0 && ThetaFlag == 0){
  Stationary = 1;
}else{
  Stationary = 0;
}

index = 0;                                //Read instructions from Serial1
while(Serial1.available()){
  Instructions[index] = Serial1.read();
  if(Instructions[index] == 252){         //If the position is requested, do not add to Instructions
    RequestFlag = 1;
  }else{
    index++;
  }
}

if(Instructions[0] == 255){               //If Instructions[0] = 255, the stop instruction has been called
  PosFlag = 0;
  ThetaFlag = 0;
  startflag = 0;
  analogWrite(pwmr, 0);
  analogWrite(pwml, 0);
}

if(Instructions[0] == 253 && Stationary == 1){    //If Instructions[0] = 253 the turn instruction has been called
  ThetaFlag = 1;
  instang = (Instructions[1] - 120) * 6.283 / 240;
}

if(Instructions[0] == 254 && Stationary == 1){    //If Instructions[0] = 254, the move forward instruction has been called
  PosFlag = 1;
  instpos = (Instructions[1] - 120) * 10;
}
```



```
if(ThetaFlag == 1 && PosFlag == 0){                       //If ThetaFlag = 1, the rover should control its angle
  anglecontrol(instang);
}

if(PosFlag == 1 && ThetaFlag == 0){                       //If PosFlag = 1, the rover should control its position
  poscontrol(instpos);
}
```

Because the drive subsystem would run the loop of the program at 250Hz, whereas the control system would only run at approximately 4Hz, to prevent information being sent more often than it could be read, a request instruction was added. If a byte of value 252 was received, the drive subsystem would transmit its current x and y coordinates as centimetres and the current angle using the same scaling with which angles were sent for the turning instruction. A byte of value 1 was sent if the drive subsystem were not currently executing any instructions so that the control subsystem could determine when to send the next instruction to prevent conflicting instructions.

Figure 58: Code to Transmit Movement Data to Control

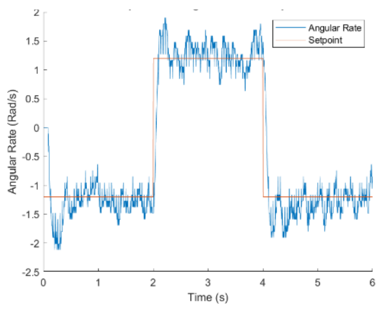


```
if(RequestFlag == 1){                     //If a byte with value 252 is recieved, the rovers coordinates have been requested
  ttheta = theta;
  if (ttheta > 3.1415){                   //Keep ttheta in the range -pi to pi
    ttheta = ttheta - 6.283;
  }
  if (theta < -3.1415){
    theta = theta + 6.283;
  }
  Serial1.write(int(round(xmm/10) + 127));              //Send coordinates in cm
  Serial1.write(int(round(ymm/10) + 127));
  Serial1.write(int((ttheta * 255 / 6.283) + 127));     //Send theta
  Serial1.write(int(Stationary));
  RequestFlag = 0;
}
```

PID Tuning, and Testing

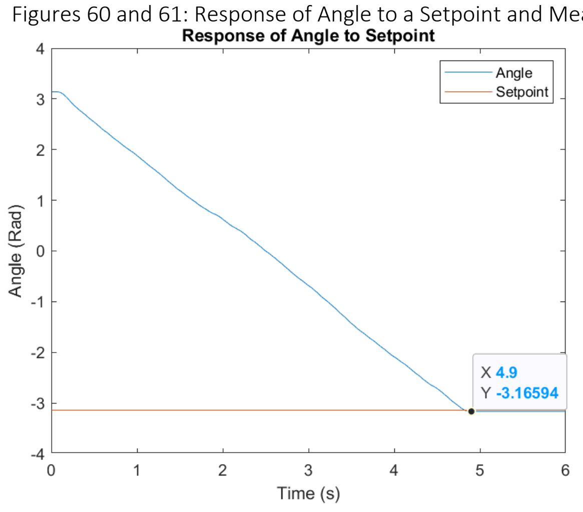
The PID controllers were tuned starting with the controller for angular rate. The angular rate was measured by sending the data to the serial monitor and then plotting this in MATLAB. As it is known that the program runs at 250Hz, the time step between data points is known and the information from the serial monitor can be plotted against time. Starting with all the gains at 0, the P gain was first increased to achieve a sufficiently quick response to the setpoint and in this case a value of 1500 was found to be sufficient. Then the I gain was increased to reduce static error and to improve the initial response time. A value of 10 was settled on. Finally small values of D were trialled in an attempt to reduce oscillations and improve the settling time. Again, a value of 10 was found to work sufficiently well. Figure 59 shows the response to a setpoint of -1.2 rad/s, then 1.2 rad/s and finally -1.2 rad/s. Although the plot is noisy, it can be seen that the angular rate does track the setpoint. While the value changes every 4mS, the average of the angular rate during the period that the setpoint was 1.2 rad/s was found to be 1.24 rad/s, within the $\pm$0.05 rad/s tolerance specified. The average angular rate in the window 0.25 to 0.5 seconds after the step was found to be 1.238 and it was decided that this was sufficient to conclude that the response time was less than 0.5s, and that the 8th specification point for the drive subsystem had been fulfilled.

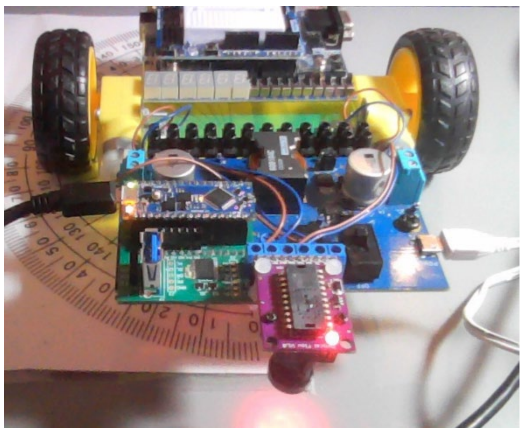
Figure 59: Response of Angular Rate to a Setpoint



The PID controller for the angle was tuned in the same way, and PID gains of 1, 10 and 0 were found to be sufficient. The rover was instructed to turn from π to $-\pi$, a turn of 2π radians, as this is theoretically the largest angle the rover should ever have to turn. The results measured by the rover is shown in figure 60. The rover reached -3.17 radians within 5 seconds, within the tolerances of the specification. The angle was measured with a protractor as shown in figure 61 and found to be approximately -181 degrees or -3.159 radians, which is less than 0.04 radians away from the angle measured by the rover. These tests demonstrate the rover fulfils points 1, 2 and 9 of the drive specifications.

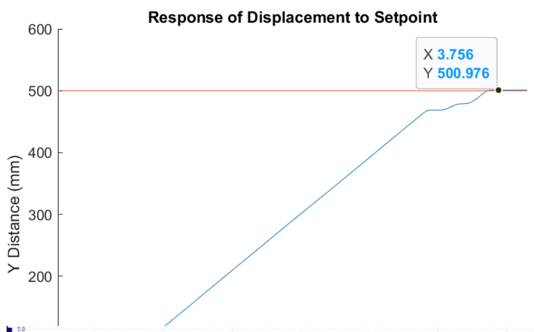
Figures 60 and 61: Response of Angle to a Setpoint and Measurements of the Rover's angle after turning.

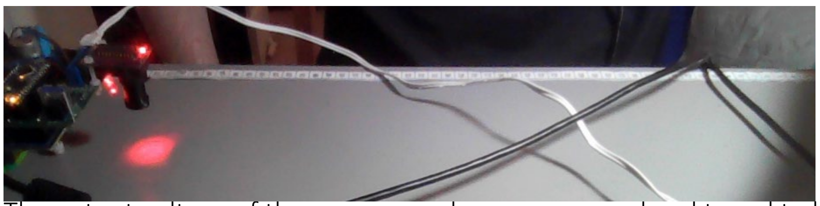




The PID controller for the distance control was tuned and PID gains of 1000, 500 and 0 were found to be sufficient. The rover was instructed to drive 500mm forward and the measured y coordinates plotted in figure 62. The rover reached a distance of 500.976mm, within the ±10mm tolerance specified. It took 3.756s, and had an average speed of 133.3mm/s, above the minimum 50mm/s speed specified. The distance moved was measured with a tape measure as shown in figure 63. The distance moved was measured to be approximately 498mm, so there was a measurement error of approximately 3mm per 500mm moved, or 6mm per 1000mm moved, within the specified 10mm tolerance. The subsystem was found to meet points 3, 4 and 10 of the specification.

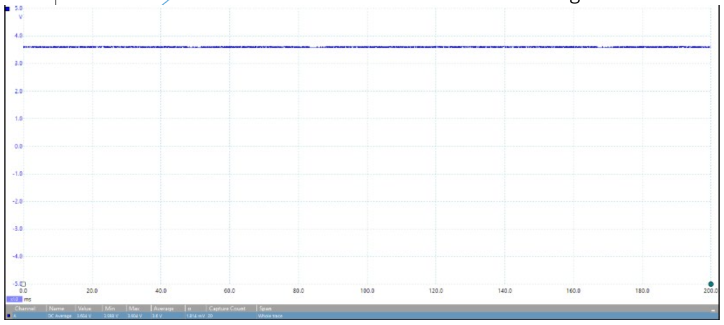
Figures 62 and 63: Response of Displacement to Setpoint and Measurment of Displacement





The output voltage of the power supply was measured and tuned to have an average 0f 3.6V, with a deviation of approximately 0.04V, within the specified tolerance. The subsystem therefore fulfilled point 11 of the specification.

Figure 64: Measurement of SMPS Output Voltage



The remaining specification points were demonstrated to be met when trialled with the control subsystem.

Problem Solving

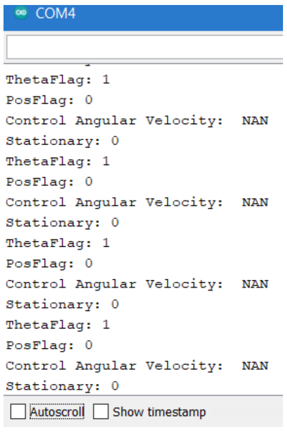
Instruction Reworking

Before the current system for sending instructions to drive was implemented, the turning and movement functions would receive a byte with a value between 0 and 255, and the instruction to turn, move, request data and stop were bytes valued 0, 1, 127 and 255 respectively. The implementation of this system meant that values of 127 that were supposed to be read as arguments for movement functions were taken as requests for data. Also, the values that were supposed to be read as arguments for movement functions overlapped with the values to denote which instruction was desired. This was changed so that the instructions were in the range 252 to 255 and parameters were valued from 0 to 240, so more instructions could be added if required.

Uninitialized variables

Another problem encountered was that in some instances, the output of PID controllers was found to be NAN, not a number. This was found to be caused by the PID controllers attempting to perform mathematical operations on variables that had not had an initial value assigned to them. By setting the values to 0 at the start of the program, this problem was fixed.

Figure 65: Not A Number Error



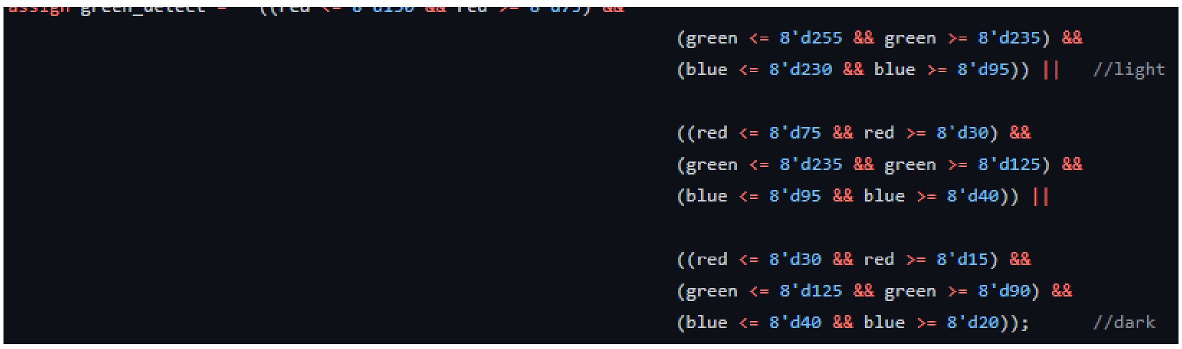
```
COM4

ThetaFlag: 1
PosFlag: 0
Control Angular Velocity:   NAN
Stationary: 0
ThetaFlag: 1
PosFlag: 0
Control Angular Velocity:   NAN
Stationary: 0
ThetaFlag: 1
PosFlag: 0
Control Angular Velocity:   NAN
Stationary: 0
ThetaFlag: 1
PosFlag: 0
Control Angular Velocity:   NAN
Stationary: 0
```

Inaccuracy in Position Measurement
Initially the rover's estimates of its coordinates were highly inaccurate. This was fixed by focusing the camera using the example camera code provided by Dr Zohaib Akhtar.

Vision

Detection - Initially used the given template code by Dr Edward Stott `assign red_detect = red[7] & ~green[7] & ~blue[7];` or equivalent for other colours but after some experimentation found that using decimal RGB values was an easier and more precise way to represent the shown values. From there those singular numbers were changed to a range of values: `assign green_detect =   ((red <= 8'd150 && red >= 8'd75)`. Picking the initial decimal values was conducted by taking a picture using the FPGA camera and using an RGB colour picker to see the RGB values of each pixel in the picture. After that trial and error was conducted on these decimal values until the highlighted areas were more representative of the balls. After more experimentation, using multiple ranges for different shades of each colour made the colour detection more precise.

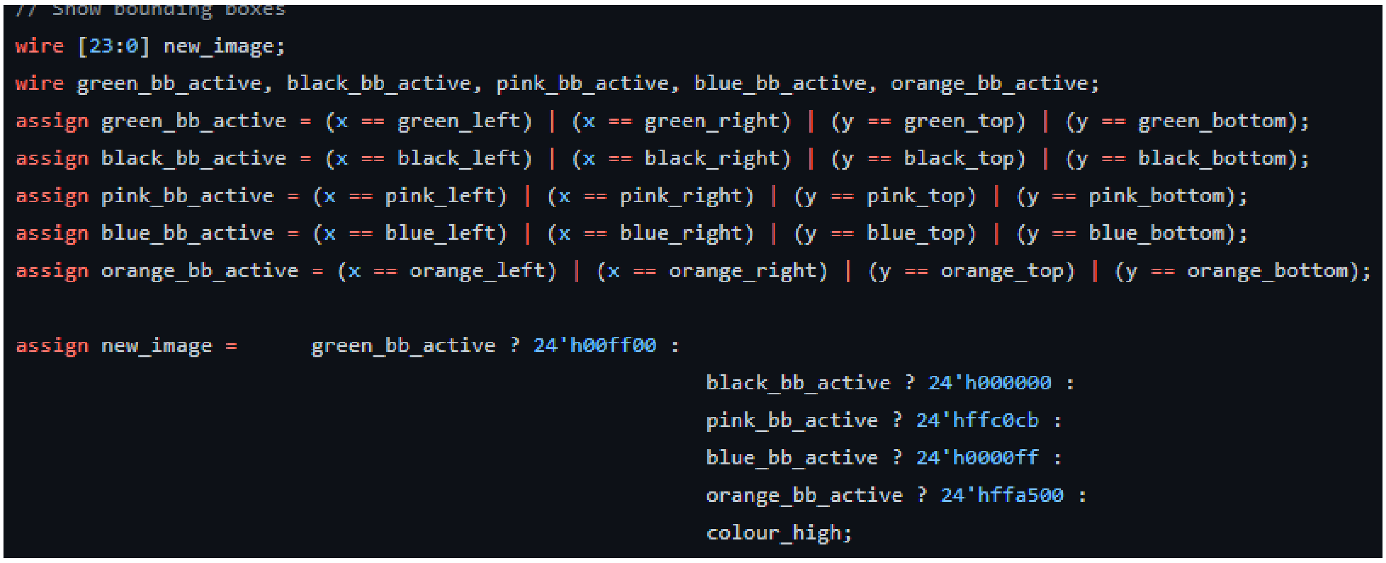
Figure 66: More Precise Colour Detection Values



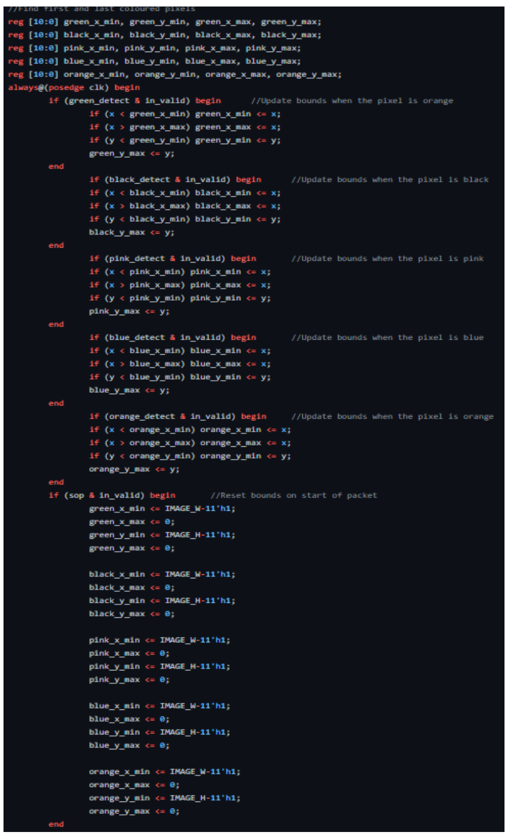
```
assign green_detect =   ((red <= 8'd150 && red >= 8'd75) &&
                                                  (green <= 8'd255 && green >= 8'd235) &&
                                                  (blue <= 8'd230 && blue >= 8'd95)) ||   //light

                                                  ((red <= 8'd75 && red >= 8'd30) &&
                                                  (green <= 8'd235 && green >= 8'd125) &&
                                                  (blue <= 8'd95 && blue >= 8'd40)) ||

                                                  ((red <= 8'd30 && red >= 8'd15) &&
                                                  (green <= 8'd125 && green >= 8'd90) &&
                                                  (blue <= 8'd40 && blue >= 8'd20));      //dark
```

Distance - After some research, PyImageSearch [12] had a formula for distance calculation using a camera. Using this method, the required measurements were obtained by setting a ball at a known distance using a ruler, measuring the diameter of the ball using a vernier calliper, and the observed width of the ball was obtained via the bounding box coordinates. Before doing this, additional bounding boxes for each colour were required to be made as none of the balls were red as can be seen:

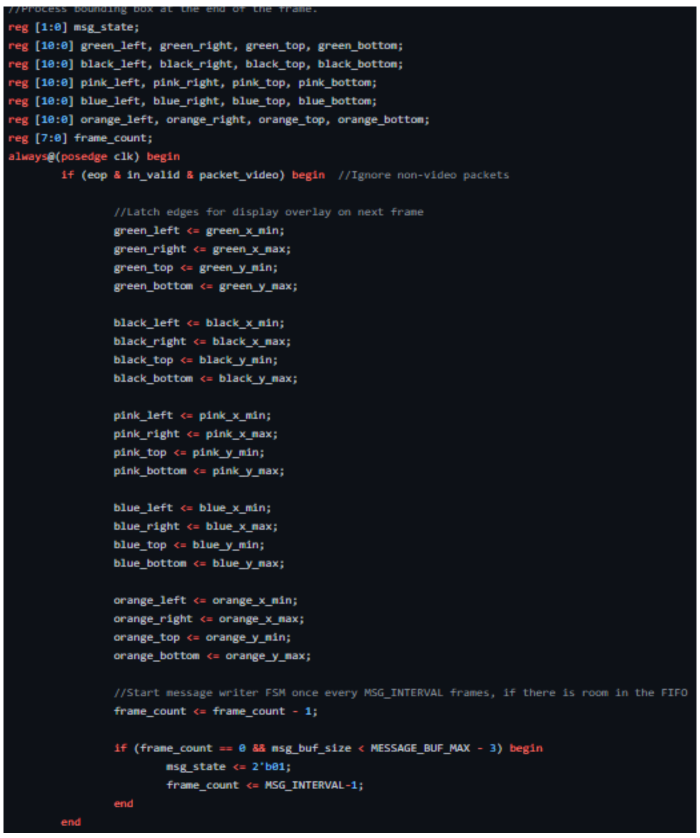
Figures 67, 68 and 69: Assign Bounding Boxes for Each Colour



```
// Show bounding boxes
wire [23:0] new_image;
wire green_bb_active, black_bb_active, pink_bb_active, blue_bb_active, orange_bb_active;
assign green_bb_active = (x == green_left) | (x == green_right) | (y == green_top) | (y == green_bottom);
assign black_bb_active = (x == black_left) | (x == black_right) | (y == black_top) | (y == black_bottom);
assign pink_bb_active = (x == pink_left) | (x == pink_right) | (y == pink_top) | (y == pink_bottom);
assign blue_bb_active = (x == blue_left) | (x == blue_right) | (y == blue_top) | (y == blue_bottom);
assign orange_bb_active = (x == orange_left) | (x == orange_right) | (y == orange_top) | (y == orange_bottom);

assign new_image =      green_bb_active ? 24'h00ff00 :
                                  black_bb_active ? 24'h000000 :
                                  pink_bb_active ? 24'hffc0cb :
                                  blue_bb_active ? 24'h0000ff :
                                  orange_bb_active ? 24'hffa500 :
                                  colour_high;
```



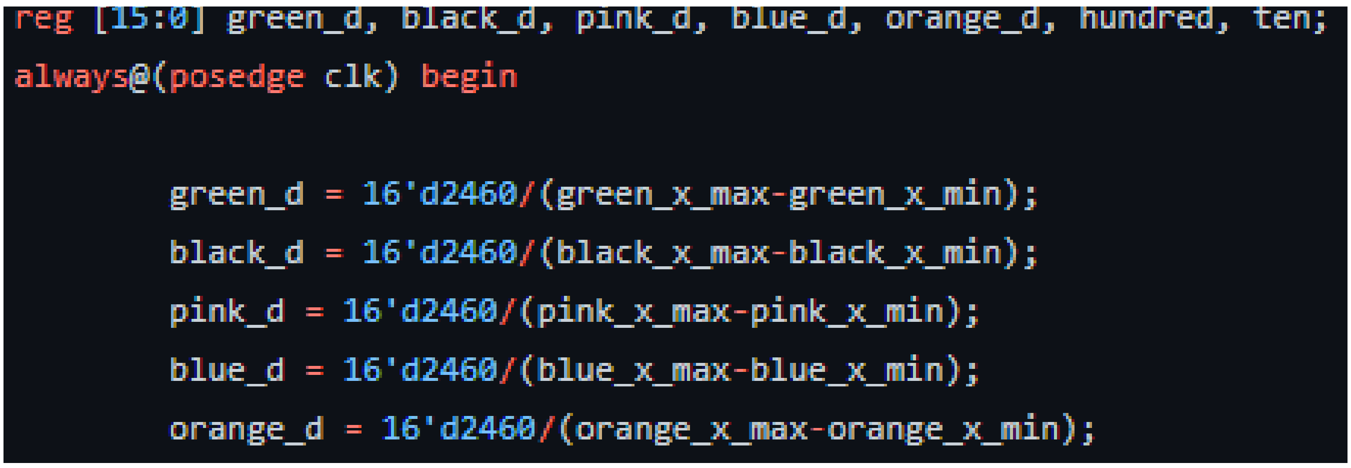
```
//Find first and last coloured pixels
reg [10:0] green_x_min, green_y_min, green_x_max, green_y_max;
reg [10:0] black_x_min, black_y_min, black_x_max, black_y_max;
reg [10:0] pink_x_min, pink_y_min, pink_x_max, pink_y_max;
reg [10:0] blue_x_min, blue_y_min, blue_x_max, blue_y_max;
reg [10:0] orange_x_min, orange_y_min, orange_x_max, orange_y_max;
always@(posedge clk) begin
        if (green_detect & in_valid) begin      //Update bounds when the pixel is orange
                if (x < green_x_min) green_x_min <= x;
                if (x > green_x_max) green_x_max <= x;
                if (y < green_y_min) green_y_min <= y;
                green_y_max <= y;
        end
                if (black_detect & in_valid) begin      //Update bounds when the pixel is black
                if (x < black_x_min) black_x_min <= x;
                if (x > black_x_max) black_x_max <= x;
                if (y < black_y_min) black_y_min <= y;
                black_y_max <= y;
        end
                if (pink_detect & in_valid) begin       //Update bounds when the pixel is pink
                if (x < pink_x_min) pink_x_min <= x;
                if (x > pink_x_max) pink_x_max <= x;
                if (y < pink_y_min) pink_y_min <= y;
                pink_y_max <= y;
        end
                if (blue_detect & in_valid) begin       //Update bounds when the pixel is blue
                if (x < blue_x_min) blue_x_min <= x;
                if (x > blue_x_max) blue_x_max <= x;
                if (y < blue_y_min) blue_y_min <= y;
                blue_y_max <= y;
        end
                if (orange_detect & in_valid) begin     //Update bounds when the pixel is orange
                if (x < orange_x_min) orange_x_min <= x;
                if (x > orange_x_max) orange_x_max <= x;
                if (y < orange_y_min) orange_y_min <= y;
                orange_y_max <= y;
        end
        if (sop & in_valid) begin       //Reset bounds on start of packet
                green_x_min <= IMAGE_W-11'h1;
                green_x_max <= 0;
                green_y_min <= IMAGE_H-11'h1;
                green_y_max <= 0;

                black_x_min <= IMAGE_W-11'h1;
                black_x_max <= 0;
                black_y_min <= IMAGE_H-11'h1;
                black_y_max <= 0;

                pink_x_min <= IMAGE_W-11'h1;
                pink_x_max <= 0;
                pink_y_min <= IMAGE_H-11'h1;
                pink_y_max <= 0;

                blue_x_min <= IMAGE_W-11'h1;
                blue_x_max <= 0;
                blue_y_min <= IMAGE_H-11'h1;
                blue_y_max <= 0;

                orange_x_min <= IMAGE_W-11'h1;
                orange_x_max <= 0;
                orange_y_min <= IMAGE_H-11'h1;
                orange_y_max <= 0;
        end
```



```
//Process bounding box at the end of the frame.
reg [1:0] msg_state;
reg [10:0] green_left, green_right, green_top, green_bottom;
reg [10:0] black_left, black_right, black_top, black_bottom;
reg [10:0] pink_left, pink_right, pink_top, pink_bottom;
reg [10:0] blue_left, blue_right, blue_top, blue_bottom;
reg [10:0] orange_left, orange_right, orange_top, orange_bottom;
reg [7:0] frame_count;
always@(posedge clk) begin
        if (eop & in_valid & packet_video) begin  //Ignore non-video packets

                //Latch edges for display overlay on next frame
                green_left <= green_x_min;
                green_right <= green_x_max;
                green_top <= green_y_min;
                green_bottom <= green_y_max;

                black_left <= black_x_min;
                black_right <= black_x_max;
                black_top <= black_y_min;
                black_bottom <= black_y_max;

                pink_left <= pink_x_min;
                pink_right <= pink_x_max;
                pink_top <= pink_y_min;
                pink_bottom <= pink_y_max;

                blue_left <= blue_x_min;
                blue_right <= blue_x_max;
                blue_top <= blue_y_min;
                blue_bottom <= blue_y_max;

                orange_left <= orange_x_min;
                orange_right <= orange_x_max;
                orange_top <= orange_y_min;
                orange_bottom <= orange_y_max;

                //Start message writer FSM once every MSG_INTERVAL frames, if there is room in the FIFO
                frame_count <= frame_count - 1;

                if (frame_count == 0 && msg_buf_size < MESSAGE_BUF_MAX - 3) begin
                        msg_state <= 2'b01;
                        frame_count <= MSG_INTERVAL-1;
                end
        end
```

Once the measurements were obtained, different distances were tested and compared to the actual distance. After some tweaking of the constant focal length, more accurate measurements were obtained. This was the final constant and code used:

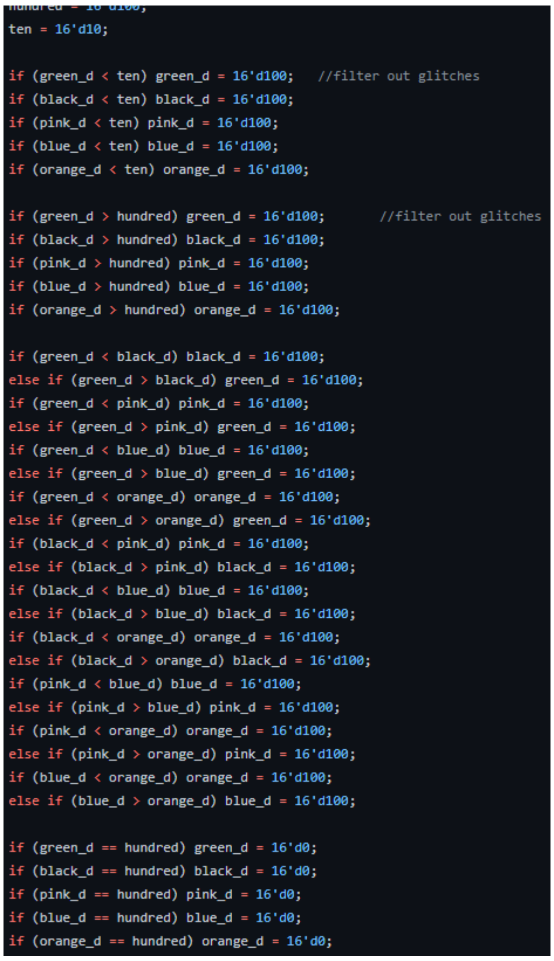
Figure 70: Measurement Code



```
reg [15:0] green_d, black_d, pink_d, blue_d, orange_d, hundred, ten;
always@(posedge clk) begin


        green_d = 16'd2460/(green_x_max-green_x_min);
        black_d = 16'd2460/(black_x_max-black_x_min);
        pink_d = 16'd2460/(pink_x_max-pink_x_min);
        blue_d = 16'd2460/(blue_x_max-blue_x_min);
        orange_d = 16'd2460/(orange_x_max-orange_x_min);
```

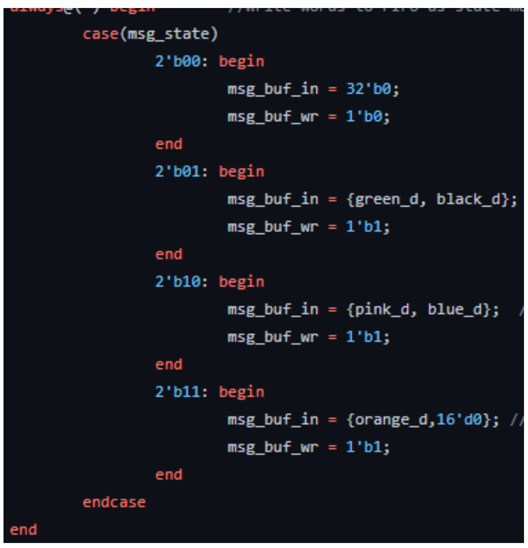
Glitches – Initially the vision subsystem was going to send the distances to all the coloured balls if they were all in line of site and then the processing of this data to find the closest ball was to be done on the control side of things but due to limited resources on the esp32, it was decided that should be done on the vision side of things. Thus, to filter out any unreasonable readings, distances less than 10cm were considered a glitch as the rover would be too close to the ball if this were really the case and due to the testing area being a square meter, any value above 100cm was also considered a glitch. If for some reason a single pixel was highlighted the program would claim there is a ball in sight very far away, but this upper limit prevents that reading from being acknowledged. Within the distance calculation block this was the logic applied for the glitch filtering:

Figure 71: Glitch Filtering Code

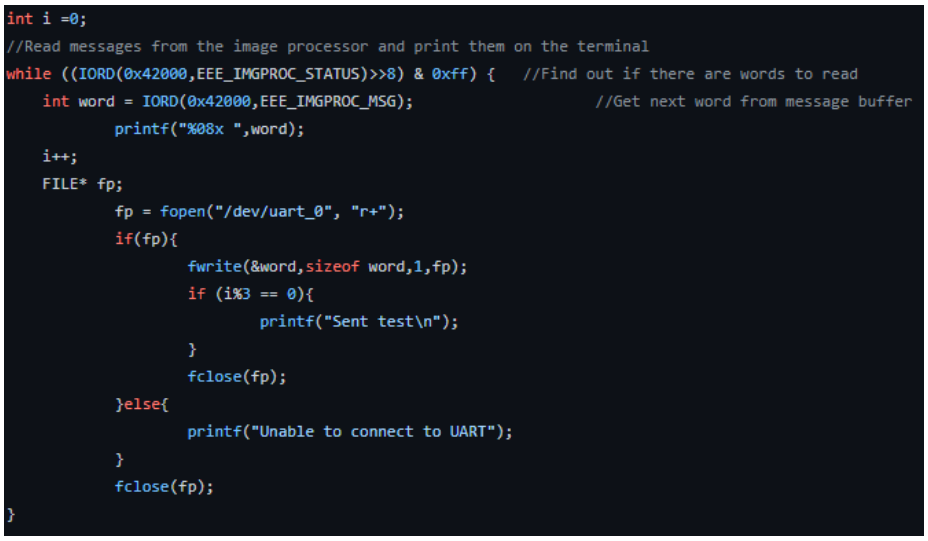

```
ten = 16'd10;

if (green_d < ten) green_d = 16'd100;   //filter out glitches
if (black_d < ten) black_d = 16'd100;
if (pink_d < ten) pink_d = 16'd100;
if (blue_d < ten) blue_d = 16'd100;
if (orange_d < ten) orange_d = 16'd100;

if (green_d > hundred) green_d = 16'd100;       //filter out glitches
if (black_d > hundred) black_d = 16'd100;
if (pink_d > hundred) pink_d = 16'd100;
if (blue_d > hundred) blue_d = 16'd100;
if (orange_d > hundred) orange_d = 16'd100;

if (green_d < black_d) black_d = 16'd100;
else if (green_d > black_d) green_d = 16'd100;
if (green_d < pink_d) pink_d = 16'd100;
else if (green_d > pink_d) green_d = 16'd100;
if (green_d < blue_d) blue_d = 16'd100;
else if (green_d > blue_d) green_d = 16'd100;
if (green_d < orange_d) orange_d = 16'd100;
else if (green_d > orange_d) green_d = 16'd100;
if (black_d < pink_d) pink_d = 16'd100;
else if (black_d > pink_d) black_d = 16'd100;
if (black_d < blue_d) blue_d = 16'd100;
else if (black_d > blue_d) black_d = 16'd100;
if (black_d < orange_d) orange_d = 16'd100;
else if (black_d > orange_d) black_d = 16'd100;
if (pink_d < blue_d) blue_d = 16'd100;
else if (pink_d > blue_d) pink_d = 16'd100;
if (pink_d < orange_d) orange_d = 16'd100;
else if (pink_d > orange_d) pink_d = 16'd100;
if (blue_d < orange_d) orange_d = 16'd100;
else if (blue_d > orange_d) blue_d = 16'd100;

if (green_d == hundred) green_d = 16'd0;
if (black_d == hundred) black_d = 16'd0;
if (pink_d == hundred) pink_d = 16'd0;
if (blue_d == hundred) blue_d = 16'd0;
if (orange_d == hundred) orange_d = 16'd0;
```

Communication - Implementing the UART connection was done via adding a UART serial port to the qsys file. From there, the Piazza post [13] was used as a guideline to help test that UART connection was working. Once the connection was successful, much time was spent debugging the communication with control as at first the data was not being sent correctly. The following format was used to send the data and the constant 0 was one of the ways the debugging process was conducted as that is always an expected value.

Figures 72 and 73: Code to Transmit Vision Data to Control



```
        case(msg_state)
                2'b00: begin
                        msg_buf_in = 32'b0;
                        msg_buf_wr = 1'b0;
                end
                2'b01: begin
                        msg_buf_in = {green_d, black_d};
                        msg_buf_wr = 1'b1;
                end
                2'b10: begin
                        msg_buf_in = {pink_d, blue_d};  /
                        msg_buf_wr = 1'b1;
                end
                2'b11: begin
                        msg_buf_in = {orange_d,16'd0}; //
                        msg_buf_wr = 1'b1;
                end
        endcase
end
```



```
int i =0;
//Read messages from the image processor and print them on the terminal
while ((IORD(0x42000,EEE_IMGPROC_STATUS)>>8) & 0xff) {   //Find out if there are words to read
    int word = IORD(0x42000,EEE_IMGPROC_MSG);                  //Get next word from message buffer
            printf("%08x ",word);
    i++;
    FILE* fp;
            fp = fopen("/dev/uart_0", "r+");
            if(fp){
                    fwrite(&word,sizeof word,1,fp);
                    if (i%3 == 0){
                            printf("Sent test\n");
                    }
                    fclose(fp);
            }else{
                    printf("Unable to connect to UART");
            }
            fclose(fp);
}
```

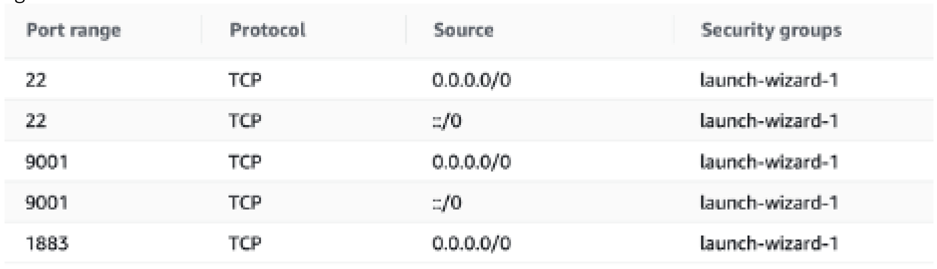
Command

Connection to ESP32 (Control Subsystem)

- Creating broker in AWS instance: https://aws.amazon.com

An MQTT broker running on a remote Ubuntu EC2 instance in AWS cloud will handle the task of data transmission and communication between clients using the publish and subscribe model. MQTT is an application layer protocol (layer 7 of OSI model) that runs on top of TCP/IP. This will allow messages to be sent between the web application and the ESP32, kicking off the first step to remotely control the Mars Rover.
An AWS instance is created to run the Broker remotely and can be accessed on ports 22 (SSH), 9001 (WebSocket's) and 1883 (MQTT). It has been configured to allow anyone in the world with the correct Public IPv4 address to access through these ports to allow for a smooth connection for the ESP32.

Figure 74: Ports



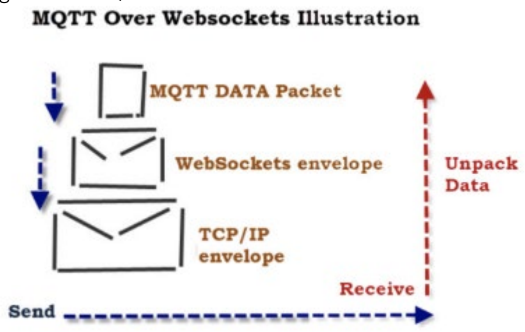
Port range	Protocol	Source	Security groups
22	TCP	0.0.0.0/0	launch-wizard-1
22	TCP	::/0	launch-wizard-1
9001	TCP	0.0.0.0/0	launch-wizard-1
9001	TCP	::/0	launch-wizard-1
1883	TCP	0.0.0.0/0	launch-wizard-1

Mosquitto is a lightweight open-source message broker that implements the MQTT protocol. After accessing the remote server over SSH, the EC2 instance is updated: *sudo apt-get update* then the Mosquitto MQTT Broker is installed: *sudo apt-get install mosquito mosquito-clients.*

Version 1: mosquito -v
This will run the default broker and will start listening on port 1883. There are also no configuration settings which will allow anyone to connect simply with the correct IP address. The -v flag enables verbose mode to see new connections as well as messages being published, and topics being subscribed to which is helpful when debugging.
The Mosquitto install includes the client testing programs, in which there is a simple subscriber client *mosquitto_sub* and a publisher client *mosquitto_pub* that meant simple tests could be devised in the command line with three terminals open and visually seeing the transmission of messages from the sender to the receiver.
This version worked well when connecting to the ESP32 but was developed on in version 2 to accommodate to receiving MQTT messages directly in the web browser.

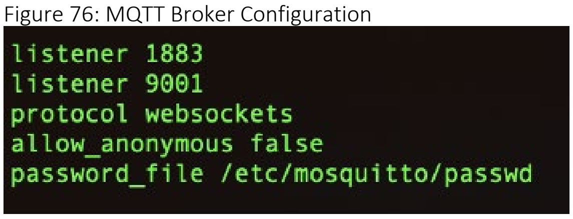
Version 2: mosquito -c … -v
WebSocket is a computer communications protocol that provides full-duplex communication channels over a single TCP/IP connection. The client and server connect using HTTP but can upgrade to WebSockets, switching the connection from HTTP to WebSocket. The MQTT WebSocket support for web browsers is included in the JavaScript client. [15]

Figure 75: MQTT Over Websockets



The file default.conf has configured security group to allow ports 1883 and 9001, which listens over WebSocket. This also includes details of who can access the Broker, which has been set to require the correct user and password to begin the exchange of MQTT data.

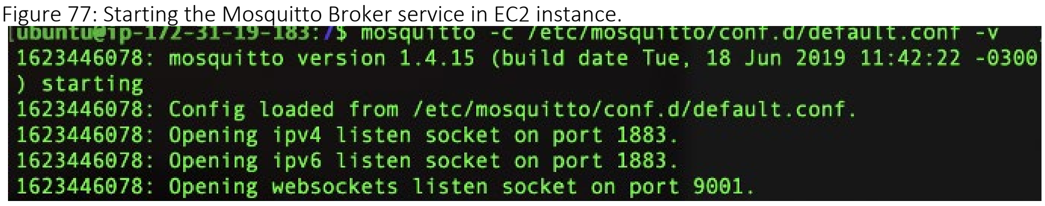
Figure 76: MQTT Broker Configuration



```
listener 1883
listener 9001
protocol websockets
allow_anonymous false
password_file /etc/mosquitto/passwd
```

The -c flag indicates that Mosquitto is configured using the file created above. To test this locally, a simple script calls the mqtt.connect(URL, configuration options) method to connect to the Broker and returns a client class, assigned to the variable *client*. The MQTT protocol acknowledges a connection with the CONNACK message, which raises the on_connect event in the client and a listener waits for a connect event. This connection is confirmed when client.connected is true, so this is logged within the on_connect function, as well as a new client successfully joining on the Mosquitto Broker which is clear to see. If there is an error in this process, the node client provides an error event that will flag up for authentication failures. However, any mistakes with the IP address or the port number will not throw an error but will instead keep attempting to reconnect. [16]

Figure 77: Starting the Mosquitto Broker service in EC2 instance.



```
[ubuntu@ip-172-31-19-183:/$ mosquitto -c /etc/mosquitto/conf.d/default.conf -v
1623446078: mosquitto version 1.4.15 (build date Tue, 18 Jun 2019 11:42:22 -0300
) starting
1623446078: Config loaded from /etc/mosquitto/conf.d/default.conf.
1623446078: Opening ipv4 listen socket on port 1883.
1623446078: Opening ipv6 listen socket on port 1883.
1623446078: Opening websockets listen socket on port 9001.
```

Once a connection was established, the next step was to publish and subscribe to the Broker using simple client functions. The client.publish(topic,message) method will simply publish a message with the given topic given that the client is still connected. To ensure this is the case, client.publish is called within an if statement with the condition that client.connected == true. [16] There are also options to indicate the QoS (quality of service) of the message. There are three QoS levels in MQTT:

- QoS 0 – at most once, “fire and forget”.

This is the lowest level of service in MQTT. There is no guarantee of delivery, and the recipient does not acknowledge receipt of the message.

- QoS 1 – at least once

Guarantees that a message is delivered at least once to the receiver and the sender stores the message until it gets a PUBACK packet from the receiver that acknowledges receipt of the message. If the PUBACK packet associated with the PUBLISH packet is not received within a reasonable time, the sender will resend the PUBLISH packet.

- QoS 2 – exactly once

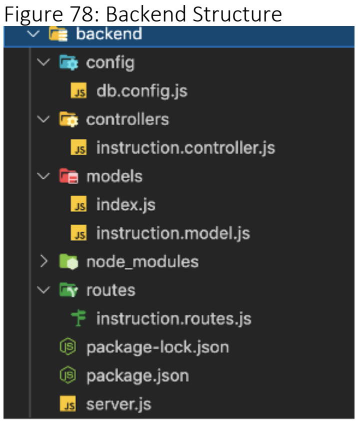
This is the highest level of service in MQTT. Guarantees that each message is received only once by the intended recipient by employing at least two request/response messages.
This gives the client control to choose a level of service depending on the network reliability and the importance of the messages being sent. However, there is a trade-off, as the Quality of Service increases the overheads and transmission time also increase so these factors should be assessed and taken into consideration. QoS 1 served the purposes of our application and was used in the publish function. [17]
After providing the Control member with the Public IPv4 address, MQTT port (1883) and user and password details, a wireless connection was once again established between the ESP32 and the Broker.

Backend and Database development - Creating Database in MongoDB Atlas Cluster

To get started with MongoDB, a Cluster was built and deployed with Microsoft Azure Cloud Provider. Then the ‘marsrover’ database user was created within the Cluster. Within this database, various combinations of collections were made depending on requirements at the time and are detailed in the versions below:

- Version 1: An ‘instructions’ collection was created purely for the backend and frontend to be integrated through methods that would be developed to interact with the ‘instructions’ data.
- Version 2: After MQTT was set up, ‘position’, ‘vision’ and ‘battery’ collections were set up corresponding to the topics being subscribed to at the time. At this point, Command and Drive had integrated to and expected position data was being inserted into the collection as coordinates.
- Version 3: As Vision and Command integrated, and Energy was no longer included, the collections became ‘instructions’, ‘position’, ‘colour’, ‘objectproximity’ and ‘objectcoordinates’ to facilitate to the data being sent from vision. These had to be sent separately, as opposed to the initial overarching ‘vision’ topic, due to the lightweight nature of the MQTT protocol.

Figure 78: Backend Structure



```
backend
  config
    db.config.js
  controllers
    instruction.controller.js
  models
    index.js
    instruction.model.js
  node_modules
  routes
    instruction.routes.js
  package-lock.json
  package.json
  server.js
```

DEPENDANCIES: express, cors, mongodb, mongoose, mqtt

server.js

Sets up an Express web server. Imports Express for building REST APIs and cors, and to set the origin to http://localhost:8081. Listens on port 8080 for incoming requests. Also initialises MongoDB connection and requires the instruction routes.

config/db.config.js

Contains the connection string URL taken directly from the cluster created in Atlas to connect to MongoDB database.

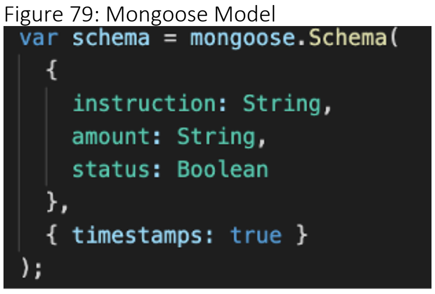
models/index.js

Imports configuration to database and defines Mongoose which is an Object Data Modelling (ODM) library for MongoDB and Node.js. It is used to translate between objects in code and how these objects are represented in MongoDB.

models/instruction.model.js

The Mongoose model shown in the scheme in figure 79 is defined and represents the 'instructions' collection in the 'marsrover' database in MongoDB. The instruction and amount are both strings input by the user and the status is a Boolean. [18]

Figure 79: Mongoose Model



```
var schema = mongoose.Schema(
  {
    instruction: String,
    amount: String,
    status: Boolean
  },
  { timestamps: true }
);
```

controllers/instruction.controller.js

Version 1: The Mongoose model supports the CRUD (Create, Read, Update, Delete) functions described in table 6. These form the standard database commands to manage instructions. [18]

Table 6: Database Commands

Action	CRUD function
Create new instruction	create
Find instruction by id	findOne
Retrieve all instructions	findAll
Update an instruction by id	update
Remove an instruction	delete
Remove all instructions	deleteAll
Find particular instruction	findAll(condition)

Version 2: The instruction controller seemed like the appropriate place to publish instructions to the MQTT Broker. In the create function, where an instruction is created and saved, an MQTT message is simultaneously published using connect.publish. This integrated quite well and never caused any problems. The function insertMessage(topic, message) was also placed here and called when incoming messages were detected as a result of the backends subscriptions. This function connected to the corresponding collection based on the topic and simply used the insertOne function to save the message in the database. This method became redundant as the frontend was developed and MQTT messages were directed there, but served as a way to extract these messages and make them readable for debugging.

routes/instruction.routes.js:

Routes are set up to determine how the server will respond when a client sends a request for an endpoint using a HTTP request. Table 7 contains the methods associated with the URLs so that the correct actions take place requests are made.
Each request uses a CRUD function defined in controllers/instruction.controller.js.

Table 7: APIs that Node.js Express App export

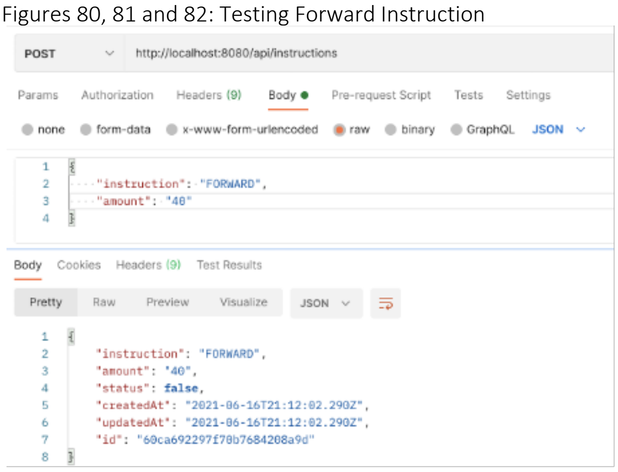
Methods	URLs	Actions
GET	api/instructions	Get all instructions
GET	api/instructions/:id	Get instruction with id
GET	api/instructions/complete	Get all completed instructions
POST	api/instructions	Add new instruction
PUT	api/instructions/:id	Update instruction by id

DELETE	api/instructions/:id	Remove instruction by id
DELETE	api/instructions	Remove all instructions
GET	api/instructions?instruction=x	Find instructions with containing x

TESTING: REST APIs

REST (representational state transfer) is a set of architectural constraints for network-based interactions. After completing the backend, https://www.postman.com/ was used to ensure that the database was properly connected, and methods were working correctly, whilst the backend server was running. Postman could test the various requests associated with a particular URL and show the HTTP response.

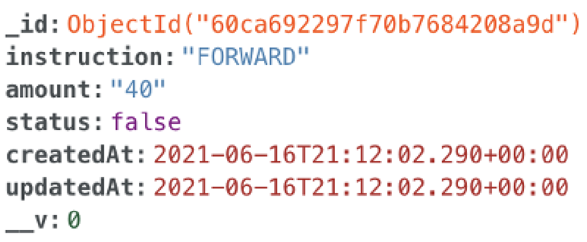
Figures 80, 81 and 82: Testing Forward Instruction



http://localhost:8080/api/instructions

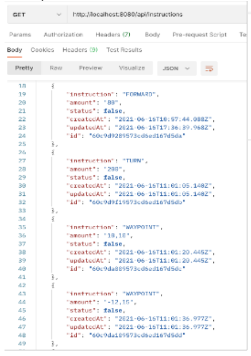
```
{
    "instruction": "FORWARD",
    "amount": "40"
}
```

```
{
    "instruction": "FORWARD",
    "amount": "40",
    "status": false,
    "createdAt": "2021-06-16T21:12:02.290Z",
    "updatedAt": "2021-06-16T21:12:02.290Z",
    "id": "60ca692297f70b7684208a9d"
}
```



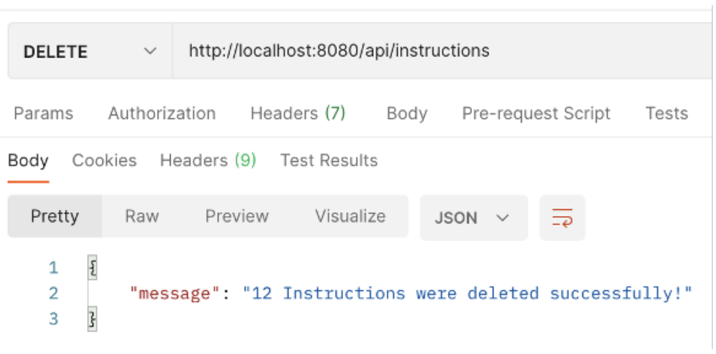
```
_id: ObjectId("60ca692297f70b7684208a9d")
instruction: "FORWARD"
amount: "40"
status: false
createdAt: 2021-06-16T21:12:02.290+00:00
updatedAt: 2021-06-16T21:12:02.290+00:00
__v: 0
```

Figure 83: Testing the POST request by typing out raw data that would be expected from the user. After sending the request, the correctly completed entry is shown and has also been inserted in the 'instructions' collection.



http://localhost:8080/api/instructions

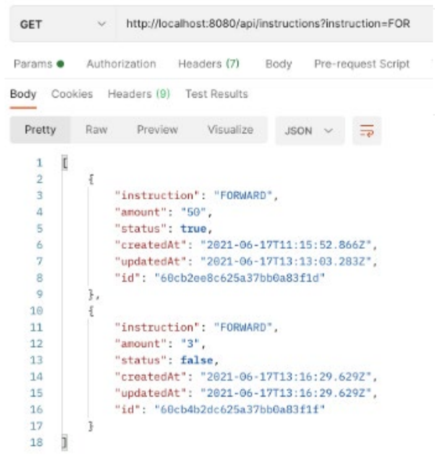
```
    {
        "instruction": "FORWARD",
        "amount": "80",
        "status": false,
        "createdAt": "2021-06-16T10:57:44.088Z",
        "updatedAt": "2021-06-16T17:36:39.968Z",
        "id": "60c9d9289573cd6ed167d5da"
    },
    {
        "instruction": "TURN",
        "amount": "280",
        "status": false,
        "createdAt": "2021-06-16T11:01:05.148Z",
        "updatedAt": "2021-06-16T11:01:05.148Z",
        "id": "60c9d9f19573cd6ed167d5db"
    },
    {
        "instruction": "WAYPOINT",
        "amount": "10,18",
        "status": false,
        "createdAt": "2021-06-16T11:01:20.445Z",
        "updatedAt": "2021-06-16T11:01:20.445Z",
        "id": "60c9da889573cd6ed167d5dc"
    },
    {
        "instruction": "WAYPOINT",
        "amount": "-12,15",
        "status": false,
        "createdAt": "2021-06-16T11:01:36.977Z",
        "updatedAt": "2021-06-16T11:01:36.977Z",
        "id": "60c9da189573cd6ed167d5dd"
    },
```



http://localhost:8080/api/instructions

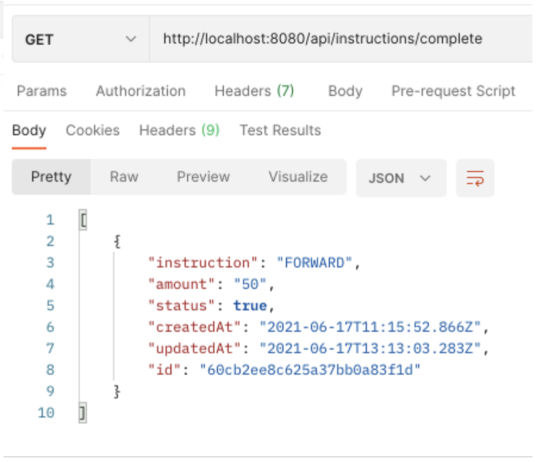
```
{
    "message": "12 Instructions were deleted successfully!"
}
```

Figure 84: Testing the GET request and DELETE request.



http://localhost:8080/api/instructions?instruction=FOR

```
[
    {
        "instruction": "FORWARD",
        "amount": "50",
        "status": true,
        "createdAt": "2021-06-17T11:15:52.866Z",
        "updatedAt": "2021-06-17T13:13:03.283Z",
        "id": "60cb2ee8c625a37bb0a83f1d"
    },
    {
        "instruction": "FORWARD",
        "amount": "3",
        "status": false,
        "createdAt": "2021-06-17T13:16:29.629Z",
        "updatedAt": "2021-06-17T13:16:29.629Z",
        "id": "60cb4b2dc625a37bb0a83f1f"
    }
]
```



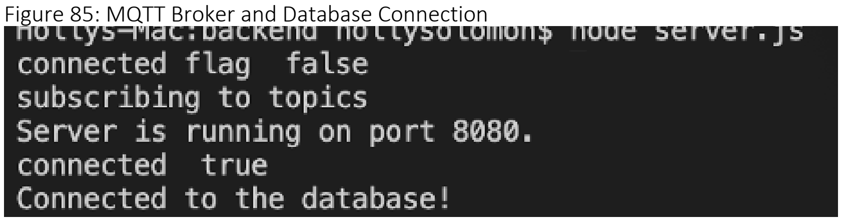
http://localhost:8080/api/instructions/complete

```
[
    {
        "instruction": "FORWARD",
        "amount": "50",
        "status": true,
        "createdAt": "2021-06-17T11:15:52.866Z",
        "updatedAt": "2021-06-17T13:13:03.283Z",
        "id": "60cb2ee8c625a37bb0a83f1d"
    }
]
```

Testing different with different routes.
Testing at this stage meant that the backend could be isolated and debugged and after getting the requests and responses correct as displayed above, the backend was ready for HTTP requests from the React.js frontend.

TESTING: MQTT

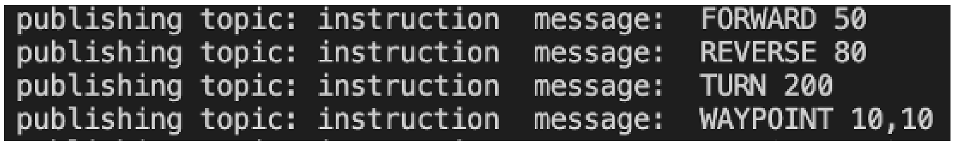
Once the backend had been integrated with MQTT, the backend server is deployed, and console logs should determine whether connections have been made to the MongoDB Database as well as the MQTT Broker.

Figure 85: MQTT Broker and Database Connection


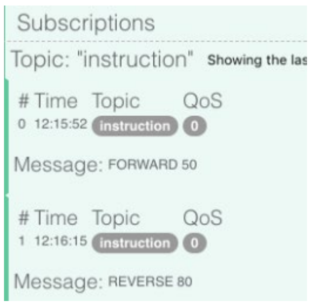
```
Hollys-Mac:backend hollysolomon$ node server.js
connected flag  false
subscribing to topics
Server is running on port 8080.
connected  true
Connected to the database!
```

Initially MQTTLens was used here to connect to the same MQTT Broker using the same credentials to test the publish and subscribe model from the web application. The web application publishes instructions with the topic 'instruction' that Control subscribes to via the ESP32. This format was discussed in a meeting and agreed on as the best way to transmit instructions to be processed on his end. MQTTLens simply subscribed to any topic expected to be received and presented all the messages it received, which was very helpful when debugging. Logging what is published in the console, combined with MQTTLens could usually pinpoint exactly where a problem was occurring, and when further research needed to be done. When this process worked correctly, Command and Control were able to seamlessly integrate and the publishing requirement for Command has been completed.

Figures 86 and 87: Publishing Instructions



```
publishing topic: instruction  message:  FORWARD 50
publishing topic: instruction  message:  REVERSE 80
publishing topic: instruction  message:  TURN 200
publishing topic: instruction  message:  WAYPOINT 10,10
```



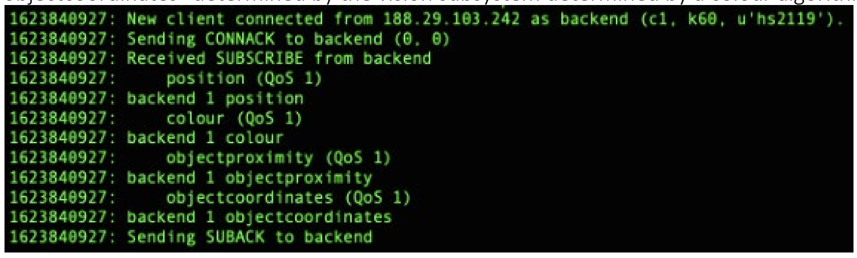
Subscriptions
Topic: "instruction" Showing the las

#	Time	Topic	QoS
0	12:15:52	instruction	0

Message: FORWARD 50

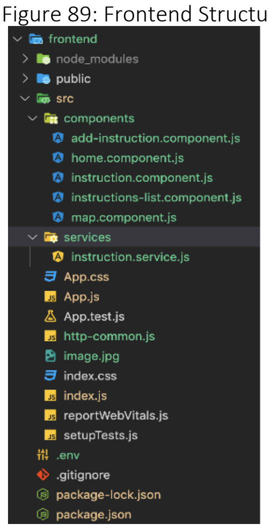
#	Time	Topic	QoS
1	12:16:15	instruction	0

Message: REVERSE 80

Figure 88: The backend subscribes to topics "position" determined by the drive subsystem using the optical flow sensor, and "colour", "objectproximity", "objectcoordinates" determined by the vision subsystem determined by a colour algorithm and bounding boxes.


```
1623840927: New client connected from 188.29.103.242 as backend (c1, k60, u'hs2119').
1623840927: Sending CONNACK to backend (0, 0)
1623840927: Received SUBSCRIBE from backend
1623840927:     position (QoS 1)
1623840927: backend 1 position
1623840927:     colour (QoS 1)
1623840927: backend 1 colour
1623840927:     objectproximity (QoS 1)
1623840927: backend 1 objectproximity
1623840927:     objectcoordinates (QoS 1)
1623840927: backend 1 objectcoordinates
1623840927: Sending SUBACK to backend
```

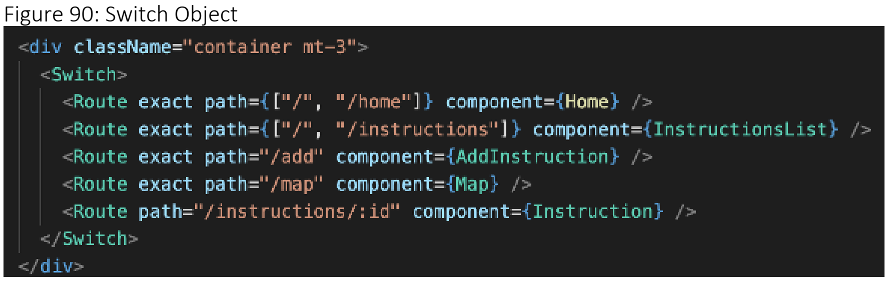
Figure 89 Shows connection acknowledgment, as well as subscriptions from the backend
Frontend development .

Figure 89: Frontend Structure


```
frontend
  node_modules
  public
  src
    components
      add-instruction.component.js
      home.component.js
      instruction.component.js
      instructions-list.component.js
      map.component.js
    services
      instruction.service.js
    App.css
    App.js
    App.test.js
    http-common.js
    image.jpg
    index.css
    index.js
    reportWebVitals.js
    setupTests.js
  .env
  .gitignore
  package-lock.json
  package.json
```

DEPENDANCIES: bootstrap, react-router-dom, axios, canvasjs-react-charts, mqtt – all in package.json
src/app.js
The App component in App.js details the navbar that links to Route paths. These paths are determined in a container with a Switch object comprising of several Route's each pointing to a different react component.

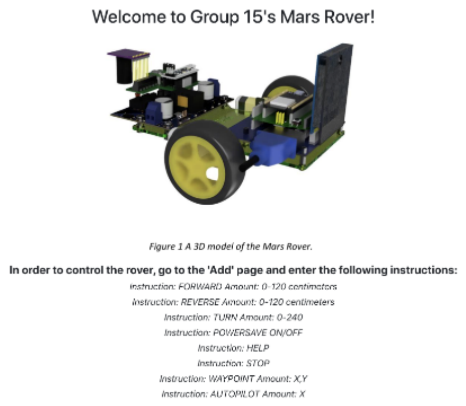
Figure 90: Switch Object



```
<div className="container mt-3">
  <Switch>
    <Route exact path={["/", "/home"]} component={Home} />
    <Route exact path={["/", "/instructions"]} component={InstructionsList} />
    <Route exact path="/add" component={AddInstruction} />
    <Route exact path="/map" component={Map} />
    <Route path="/instructions/:id" component={Instruction} />
  </Switch>
</div>
```

src/http-common.js
Initialises axios with http base and URL headers depending on RESP API that server configures to
src/services/instruction.service.js
Contains the InstructionDataService component which uses an axios object and creates methods for sending HTTP requests to the Web APIs. Axios has the get, post, put and delete methods which go hand in hand with the HTTP requests GET, POST, PUT and DELETE to form the CRUD operations. [18]
Components
The components involving instructions call InstructionDataService methods which use axios to make HTTP requests and receive HTTP responses. A description of each component routed to from the navbar and the various methods they use to each meet their requirements are presented below.
{Home} - A homepage welcoming users to the web app and displaying a list of accepted instructions. Simply uses bold and italic styling in a HTML element and imports a .jpg image presented under the welcome message.

Figure 91: Webpage Welcome Message



Welcome to Group 15's Mars Rover!

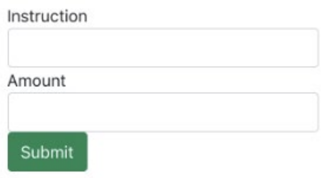
Figure 1 A 3D model of the Mars Rover.

In order to control the rover, go to the 'Add' page and enter the following instructions:

Instruction: FORWARD Amount: 0-120 centimeters
Instruction: REVERSE Amount: 0-120 centimeters
Instruction: TURN Amount: 0-240
Instruction: POWERSAVE ON/OFF
Instruction: HELP
Instruction: STOP
Instruction: WAYPOINT Amount: X,Y
Instruction: AUTOPILOT Amount: X

{AddInstruction} - Allows instructions to be submitted by the user. The constructors are defined, the initial states are set, and *this* is bound to different events. The information needed form the user is typed in through the form, indicating the instruction and the amount which requires two functions to track the values of the input and set that state for changes. The function saveInstruction() gets the value of the form and sends a POST request to the web API using the InstructionDataService.create() method, which also publishes an MQTT message for the ESP32 to receive. For the render() method, the state 'submitted' is checked and if it is true, the *Add* button will show up allowing the user to submit another form.

Figure 92: Instruction Input



Version 2:
To implement the feedback and validation discussed in control, a feature was added to display this useful information to the user as an alert. The lifecycle component componentDidMount() calls the newFeedback() function which listens for MQTT messages with the topic 'feedback' and uses alert() to directly display this to the user. This worked quite well, but sometimes repeats feedback messages.

{InstructionList} - Gets and displays an array of submitted instructions, allows for instruction list to be searched by name of instruction and displays the details of a selected instruction on the right.
Contains the following states:

- searchInstruction:
  Methods from InstructionDataService that affect this state: findByInstruction()
  Initially set to an empty string and its value is determined by text entered in the search bar by the user. This then calls the onChangeSearchInstruction(e) function, changing the state. When the search button is clicked, the searchInsruction() function uses the findByInstruction method which the HTTP response changes the state of the instruction array.
- instructions:
  Methods from InstructionDataService that affect this state: getAll(), deleteAll()
  Initially set to an empty array and when the InstructionList is mounted, the retrieveInstructions() function is called which uses getAll(), changing the state of the instruction array depending on the response. A similar process takes place when all instructions are removed by clicking the 'Remove All' button which uses deleteAll() method, and also calls refreshList(), calling retrieveInstructions() and setting the instructions state to an empty array.

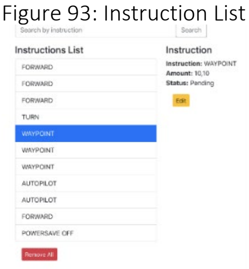
- currentInstruction:

Initially set to null since an instruction has not been selected yet. After one has been selected by the user, setActiveInstruction(instruction, index) is called so that the correct instruction is retrieved from the array instructions and displayed.

- currentIndex:

Initially set to -1 as this is an impossible index for the instructions array, so the appropriate value is set in this state when an instruction is selected from the list and setActiveInstruction(instruction, index) is called.
The **Edit** button will direct the user to a page dedicated to that instruction. The React Router Link for accessing this page has the URL /instruction/:id.

Figure 93: Instruction List



Search by instruction | Search

Instructions List
- FORWARD
- FORWARD
- FORWARD
- TURN
- WAYPOINT
- WAYPOINT
- WAYPOINT
- AUTOPILOT
- AUTOPILOT
- FORWARD
- POWERSAVE OFF

Remove All

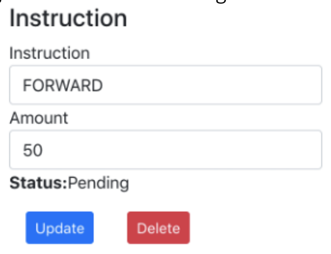
Instruction
Instruction: WAYPOINT
Amount: 10,10
Status: Pending

Edit

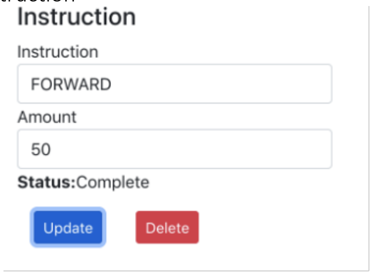
{Instruction} – Displays a form dedicated to the instruction selected based on its id, where the instruction, amount and status can be updated as well as the ability to delete the instruction all together. The component lifecycle method componentDidMount() is used to call getInstruction(id), which uses the id for the instruction requested by the user to update the state of the currentInstruction immediately once the Edit button has been clicked and is displayed to them in a form via the render() method. [18]
Methods used from InstructionDataService:

- get()
- update()
- delete()

Version 1:
Initially the status was updated when a button was clicked, and this worked well but had no use in this application.
Version 2:
The idea to use an update from the ESP32 when the rover is stationary to represent when an instruction had been completed worked well with the status feature in the Web Application. Following discussions with the control subsystem, he was able to send this information to me and we agreed on the topic 'active'. The method updateStatus() was then changed to accommodate to this by using a global variable to store the messages coming in for MQTT. The messages expected were "true" and "false" which corresponded to the Boolean type of the state status, so the global variables directly set the state of status and clicking on the update button will change this to complete if status is true. This also updates in the InstructionList component.
Figures 94 and 95: Testing Forward Instruction

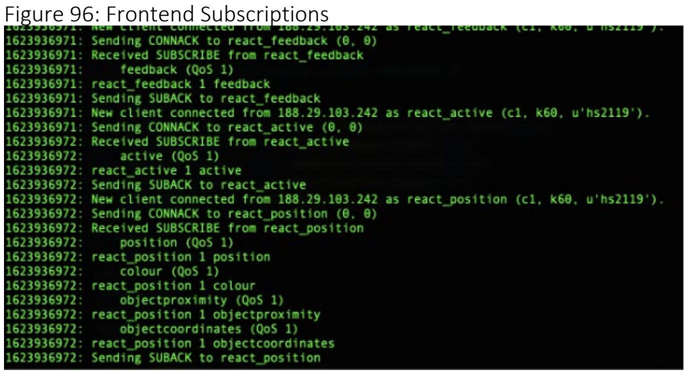


Instruction
Instruction
FORWARD
Amount
50
Status: Pending
Update Delete



Instruction
Instruction
FORWARD
Amount
50
Status: Complete
Update Delete

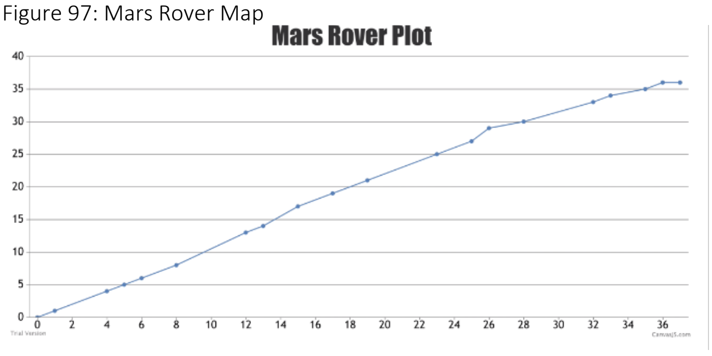
Figure 96: Frontend Subscriptions



```
1623936971: New client connected from 188.29.103.242 as react_feedback (c1, k60, u'hs2119').
1623936971: Sending CONNACK to react_feedback (0, 0)
1623936971: Received SUBSCRIBE from react_feedback
1623936971:     feedback (QoS 1)
1623936971: react_feedback 1 feedback
1623936971: Sending SUBACK to react_feedback
1623936971: New client connected from 188.29.103.242 as react_active (c1, k60, u'hs2119').
1623936971: Sending CONNACK to react_active (0, 0)
1623936972: Received SUBSCRIBE from react_active
1623936972:     active (QoS 1)
1623936972: react_active 1 active
1623936972: Sending SUBACK to react_active
1623936972: New client connected from 188.29.103.242 as react_position (c1, k60, u'hs2119').
1623936972: Sending CONNACK to react_position (0, 0)
1623936972: Received SUBSCRIBE from react_position
1623936972:     position (QoS 1)
1623936972: react_position 1 position
1623936972:     colour (QoS 1)
1623936972: react_position 1 colour
1623936972:     objectproximity (QoS 1)
1623936972: react_position 1 objectproximity
1623936972:     objectcoordinates (QoS 1)
1623936972: react_position 1 objectcoordinates
1623936972: Sending SUBACK to react_position
```

Map - Visually displays X,Y coordinates sent from the rover, from MQTT message delivered with the topic 'position', on a graph to give an idea of the rovers starting and stopping positions, as well as its path. The CanvasJS library can be used here to plot a dynamic line graph, that automatically scales depending on the coordinates provided. There is an update interval in which the updateChart() function is called and global variables are used to save the values received via MQTT and are refreshed and plot. As well as the rover's position, data from vision with details of colour, distance and proximity is also received and stated under the map so the user has an idea of where the obstacles are detected and how the Mars Rover moves to avoid them. This was tested using MQTTLens but due to the camera malfunctioning before our demonstration, we were unable to show this feature.

Figure 97: Mars Rover Map



Ref: https://canvasjs.com/

Conclusions & review of requirements

Table 8: Table of final requirements

Requirement	Successful?	Description
Create an intuitive and sleek Web Application.	YES	Throughout the project, the functionality of the Web Application improved, and the group was happy with the user interface.
Develop methods to interact with instruction data.	YES	The REST APIs methods were tested in Postman and integrated with the frontend. When an action is made by the user in the interactive frontend, the appropriate functions and methods are called to aid the flow of instruction data from the database to the backend then frontend and vice versa.
Integrating frontend and backend.	YES	Through the REST APIs and HTTP requests, information flowed from the backend to the front end successfully.
Create MQTT Broker to enable communication with ESP32.	YES	Mosquitto Broker running on a remote AWS EC2 instance allowed the ESP32 to establish a wireless connection, opening communications via port 1883.
Connect to Broker from Web Application Client.	YES	Enabling WebSockets (port 9001) in the MQTT Broker meant that the web app could directly connect to the broker and receive MQTT messages.
Publish instructions.	YES	Testing locally with MQTTLens allowed for debugging before seamlessly publishing instructions to the ESP32.
Subscribe to various topics.	YES	This is achieved simply in s few lines of code as soon as the backend server and the front end are running.
Use Messages from MQTT Connection to ESP32 to improve functionality of Web Application.	YES	Implementing the WebSockets connection in React.js components meant that these messages could be translated to an action in the web application.
Map the rover’s path.	YES	After establishing a connection over WebSockets and finding an appropriate library to plot x,y coordinates on a graph, the map was formed.

Integration

Design and implementation of the five modules did not progress at equal rates, so an integration directly in line with the design plan laid out in Section 5 was not possible. As this was a likely possibility there had been allowances made in the integration plan and frequent communication between module developers during the design process ensured that while process was staggered, integration largely was smooth when modules were ready.

As the Drive subsystem was able to quickly establish control over movement and some position control and reach an operational state, this was the first module to be implemented. Over the course of a single Teams call the Integration-side rover implemented all the features of Drive at that time, calibrated the position-sensor and began preliminary testing.

- Initial tests were simple movement commands of up to 30cm across a desktop surface and large angle motions, to ensure that the rover carried out these instructions effectively. However, when placed on the 1m-square area that had been identified as a possible demonstration area, it became clear that significant precision in the rover’s motion was lost compared to the flat surfaces the drive subsystem had been initially tested on.
    - A smooth surface was identified and after ensuring that precision remained high and that the position-sensor worked effectively on it, another 1m-square was created in preparation for future large-scale tests and the video demonstration of the rover.
- Overall, implementing the Drive subsection was simple, efficient and provided a crucial base for further iteration and integration, and for other modules to begin integration.
    - After this initial implementation, the ability to track the rover’s angular position relative to a start position and move the rover backwards were identified as useful additions to the system’s functionality requiring minimal development from other module developers and were added with little issue.

Control was the next module implemented as it would enable the connection of Drive to any other subsystem and establishing of full manual control over the rover. Using set-up instructions provided by Control, a Wi-Fi connection was established using the ESP32 board, enabling commands to be sent by Control and carried out by the rover.

- Preliminary testing of this integration used simple movement commands similar to those used for Drive alone to ensure the subsystems communicated effectively before built up into compound instructions to fully test the capabilities of the rover.
    - An issue encountered early during this integration was that the commands to turn, move, request data and stop were assigned the values 0, 1, 127 and 255 respectively, however the argument for these commands determining the amount of turning or movement required could take any value from 0 to 255. This led to command argument values being interpreted as a new command or function call, causing stuttering

movement. Upon identifying the cause of the issue, arguments were changed to take values from 0 to 240, and command values were assigned the value range 241-255, separating instructions and arguments and leaving room for any additional instructions that may be required.

- Integration of Drive & Control was done efficiently with all testing and debugging of movement problems e.g. stuttering taking place over a two-hour Teams meeting, at the end of which full manual control over the Rover had been established.
    - Subsequently the ability to set waypoints and have 'automatic' rover movement, where the rove can move to a set of random coordinates, were added to the Control subsystem. During the implementation of these features, it was noted that some PID controllers had an output of NAN, not a number, which caused uncontrollable and erratic movement.
    - Stringent version control of both the Drive and Control Arduino codes made it simple to roll back to previous versions and rule out new functions as the cause of the issue, which was later identified as being due to un-initialised values in the Drive subsystem being operated on by the PID controllers. Initialising these values as zeros solved the problem after a couple of hours of debugging.

The third subsystem to be integrated was Command, as frequent communication between Control and Command ensured that it could easily be implemented on top of the existing system to establish a method of sending commands to and receiving data from the Rover using a web browser.

- Early in the development of both subsystems, the Control and Command subsystems agreed to use MQTT as the network protocol. Once this was established front-end development took place on using Mongo DB to store information sent to and by the rover with a Mosquitto message broker.
    - The same set of simple position and angle commands as used to test manual motion control previously were employed to ensure that full functionality was retained in this design iteration. It was confirmed that instructions could be sent by Command and were carried out by the rover, although Control lacked a mechanism to send information from the rover to Command during this initial period of testing.
    - Following this proof-of-integration session, channels for sending information from the rover to Command were created. Initially the rate of date being recorded by the rover was too high, causing lag on the browser, so an algorithm was created to compare old and new data values and only upload them to the database when there was a significant change.
    - Some end-to-end message/commend latency was now observed however these delays were not significant enough to prevent successful integration at any point in the project.
- The addition of Command to the establish Drive/Control implementation went smoothly with no significant issues with the implemented features and useful next steps in establishing a method of sending data from the rover to Command and filtering it as needed.
    - Subsequently to initial integration some changes were made in how information is stored by the Command system, with incoming information from the MQTT sufficiently filtered to be presented directly on the front-end of the webpage, leaving the DB as a log of instructions sent to the rover.
    - The UI and homepage of the browser were developed in the background as further physical iteration and testing of the rover took place, with the ability to work with new commands, such as stopping the rover's movement and travelling backwards, being added as those features were implemented.

During further testing of these integrations over wider areas, the wires connecting the rover to a laptop was identified as a potential issue that could hamper the rover's range of motion due to the two wires – the power cable for the SMPS and USB-B cable powering the ESP-32 board – being on opposite corners of the rover. Connecting the USB-B cable into the open-USB socket on the SMPS-mounted Arduino ensured that at this stage, with Drive, Control and Command fully implemented, the rover could fully operate using only the SMPS power cable.

The final subsystem to be implemented was Vision, as in keeping with the justification for Integrating by Parts in the specification section integration of Vision would only start once Command was fully integrated. By establishing connections to the onboard camera on the rover and uploading colour-detection algorithms, the rover could now identify the colours of specific objects – five small balls – while still retaining full movement capabilities.

- The initial tests of the Vision system were separate to any other subsystems, with different balls, distances to the balls and lighting conditions being used to identify the ideal conditions under which the colour-detection system can operate. For all tests from this point involving the Vision system the test environment was darkened, torches were used to illuminate balls for easier identification, and a grey towel was used to prevent the blue bathtub in the testing area from producing a constant positive identification for blue.
    - The field of view of the camera prevented it from seeing objects that are too close, so this was accounted for automatically in the Control system, so that the rover would stop moving towards a ball while it could still see it to identify.
    - Overall integration of the Vision subsystem on its own was smooth, although it required a specific version of Quartus to the one on the Integrator's laptop, which cost several hours of time, and the Eclipse system within Quartus was prone to lagging, or in some cases crashing, without warning.
- Following this, the Vision subsystem was integrated with the existing implementation of the other three modules, with the Control subsystem editing the Arduino code to print the colour-detection data it was receiving, which could then be sent to the Command system.
    - Implementation took considerable debugging over three consecutive days due to issues with the colour-detection algorithms employed by Vision, how the output from Nios II/Quartus could be read and used by the Control system, and the rate at which information was sent filling the Mongo database too fast.
- Once debugging had concluded, the finished rover design was implemented, capable of precise movement and position tracking following manual and automatic commands sent from a web browser, the detection of five separate-coloured balls and presenting this information on the browser front-end.

For the operation of the Vision subsystem to be monitored through a laptop screen the USB-B port previously connected to the SMPS Arduino and output of the FPGA-USB converter must always be plugged into the laptop. This meant that the rover permanently had three relatively short wires coming out of three corners of its square frame, drastically hindering the practical range of motion of the system.

- A 4-USB to 1-USB adapter was mounted on top of the unused area of the rover board, with the FPGA-USB, USB-B and Arduino cables connected into it, reducing the number of wires coming out of the rover to one which in conjunction with a 1-meter USB-USB extension cable ensured it had the necessary range of motion.
    - Tape was used to tie down all wires on the rover to ensure that they could not hinder the wheels, and to mount the FPGA-USB adapter to the rover board.
    - It is worth noting that without the USB-USB extension cable the rover would still have had a limited range of motion. If the rover were being implemented with only the parts provided, no 4-to-1 adapter or extension cable, its range of movement would be incredibly small, and it would be functionally incapable of turning at angles greater than 90° or risk tangling cables and loosening connections. If the rover were to be implemented by an Integrator whose laptop did not have 3 USB ports, they would be incapable of fully realising a design involving all the modules. These issues can only be mitigated on the Integrator's end and would require more forward-thinking during the planning and design stages of the rover components to ensure that the rover would remain practically functional with only the parts provided.

During the integration process, as each subsystem met their own requirements, it was found that the high-level specification created for the rover in table 1 had been fulfilled.

Project Management and Organisation:

To realise the rover, delegation was required to implement the individual subsystems. These systems were to be developed in parallel to one another and tested rigorously before installing them on the final system, to ensure a higher degree of confidence in the collective operation and formative assessment of the rover. Each subsystem role was assigned to a team member catering to their specific skillset to maximise the performance of the deliverables. Given the remote setting of the general project, it was necessary to communicate frequently between all team-members and to maintain team roles in such a way to maximise the overall team performance.

As these subsystems will be developed in parallel to one another, they will have their own respective theoretical areas of research and development, which will be recorded individually before assessing the overall functionality of the system.

The team produced a Gantt chart, in Appendix C, to help organise the project and keep the development on schedule. This worked well for the most part. Most of the subsystems were developed on schedule, however due to unforeseen circumstances, the energy subsystem was not able to be completed and therefore has been omitted from this report.

Project Conclusion and Team Reflections

Technical Conclusion

The project management team concluded that the rover's development process had been an overall success in both a micro and macro-based context. Not only had the individual subsystem specification objectives been met with precise performance, but the system was able to maintain consistent and coherent interfacing with these subsystems in parallel, resulting in a high-level system with accurate and streamlined performance margins, executing complex operations seamlessly and utilising all subsystems at all times to provide the most well-developed level of functionality to the end-user.

Whilst most bugs and performance issues mainly stemmed from the manner of testing, namely with wiring, given more time, the team would have been able to realise the Energy subsystem as well, and allowed for a larger range of operation and testing without constraints. Further autonomy and refinements to automatic movement could be further developed or with other subsystems and modules to be attached to the rover, for example a temperature sensor or an air pressure sensor, to gather more real-time information of where the rover is operating.

Team Logistics and Communication Conclusion:

Upon reflection of the logistics and communication of the project coordination overall, the team concluded that the way in which communication was maintained was of a high quality. Communication between team members was done entirely on Microsoft Teams, with smaller arrangements discussed outside of main meetings on a unified WhatsApp group chat. Communicating on a limited spread of platforms allowed for a greater degree of team unity when designing the rover and allowed every member to be in understanding of the current status of affairs.

When integrating subsystems in pairs, the relevant team members were able to quickly and without difficulty communicate with one another, to keep each other aware and informed of the status of their subsystem and the intended method of communication between them, coming to unified decisions rather than individual based decisions that allowed the priority of design to be the rover, as opposed to their individual subsystem. By keeping integration informed of any developments, debugging sessions were instantaneous to arrange and allowed for a quick turn-around between making changes and implementing them on the rover proper. Integrating the rover in pairs also allowed for a greater degree of confidence in the functionality and link between two subsystems, which allowed for those pairs to be linked on a higher level of abstraction with a high degree of functional certainty of the rover.

Organizational Conclusion:

The planning of the project was successfully undertaken overall, by initially dedicating time to researching and learning about the subsystem of choice, as well as being in initial communication with other team-members early in the project, allowed for a greater depth of teamworking operability between the team-members. Dedicating the necessary hours for integration and implementation reflected a sense of priority and dedication from the team to achieve the development of a high-level system, demonstrating a strong organizational work ethic.

The team conducted their meetings in a frequent but relevant manner. Meetings were regularly attended by the entire team and were performed with purpose, where each meeting takes a structured format of reflecting over the advancements and developments in the elapsed time since the last meeting. The next matter of affairs is the current week and day's plans for design and implementation, followed by an individual status report from the subsystem members. The members then create a unified team action list and an individual subsystem action list to be worked on following the meeting. By ensuring all meetings took this format the team was able to come in and exit meetings with a sense of heightened understanding and perspective of the status of the overall project but also on their fellow team-mate's subsystems.

Team organization and the structure of operation was conducted smoothly in spite of the challenges of working remotely and being spread across multiple time-zones, by keeping in frequent communication and compromising with each other on ideal meeting times, the team was able to ensure consistent and maximised attendance and participation, reflecting a deepened flexibility of team operation and a strong shared development mindset, working with each other rather than individually to realise a system more complex than their individual subsystems.

Individual subsystem conclusions:

Control:

The Control subsystem functioned in accordance with the specification goals set both in the main rover objective list and in the subsystem specific specification criteria. However, extra improvements with more time would have enabled the system to securely and consistently regulate the rover's widescale operation, in particular:

- Coordinating with other methods of communication to not use up all the UART instances, for instance using SPI or I2C with Vision.
- Delegating less core functionality, with coordinate way-pointing to the ESP32, owing to its finite and small memory capacity, ideally these instructions can be written through UART to perform less operation locally, and thus reducing the time complexity of overall operations.
- Using more flags and various validation checking algorithms to safeguard the rover from a range of operational scenarios, such as unexpected collisions from another object or rover, or in the event the software becomes unresponsive, and the user needs to take manual control. This style of Control is most typically seen in UAVs and other drones, and would allow for separate control via Radio communication, which Arduino also provides support for.
- Include a Request type system to coordinate with Vision as well, to align the data consistently.
- Local filtering of the object coordinates was functional, although could have been made more accurate by taking an arithmetic or geometric mean of an array of 10 sample coordinates generated, to further refine the accuracy of object detection.

Command:

The command subsystem served its functional requirements by communicating with control and with the user. However, if there was more time perhaps the web application could be improved on in the following ways:

- If instruction could be input in a quicker manner involving buttons.
- The obstacles detected by Vision could be plot on the map along with the rover's path to give a clear indication when the rover moves to try and avoid this.

Drive:

The drive subsystem worked well, fulfilling all the specification points for the subsystem. However, there are several improvements that were felt could be made:

- The function to control driving in a straight line works by calculating coordinates that the rover needs to reach and uses closed loop control to reach them. This could be easily adapted to allow the control subsystem to send the coordinates of the desired position and allow the drive subsystem to control the movement, as opposed to the control subsystem calculating the angle and distance and sending the instructions to turn and move.
- With rotational sensors on each wheel, it would be possible to accurately measure the speed of each wheel. This would allow the rover to maintain accurate angle tracking while turning in an arc. The range of types of motion would be increased.
- The rover could accurately track the angle and coordinates on most surfaces so long as the camera was calibrated for that surface. Perhaps it would be possible with more time and equipment to implement an auto focus function so that the rover could transition more easily between different surfaces.

Vision:

The basic specification was fulfilled but, if more time were available, the colour detection could have been improved by running an algorithm on the stored rows of pixels to make sure more correct pixels were highlighted and less incorrect pixels were highlighted. Furthermore, some sort of edge detection could have been implemented to help detect the black ball as a lot of the background was considered black if the room were not extremely well lit.

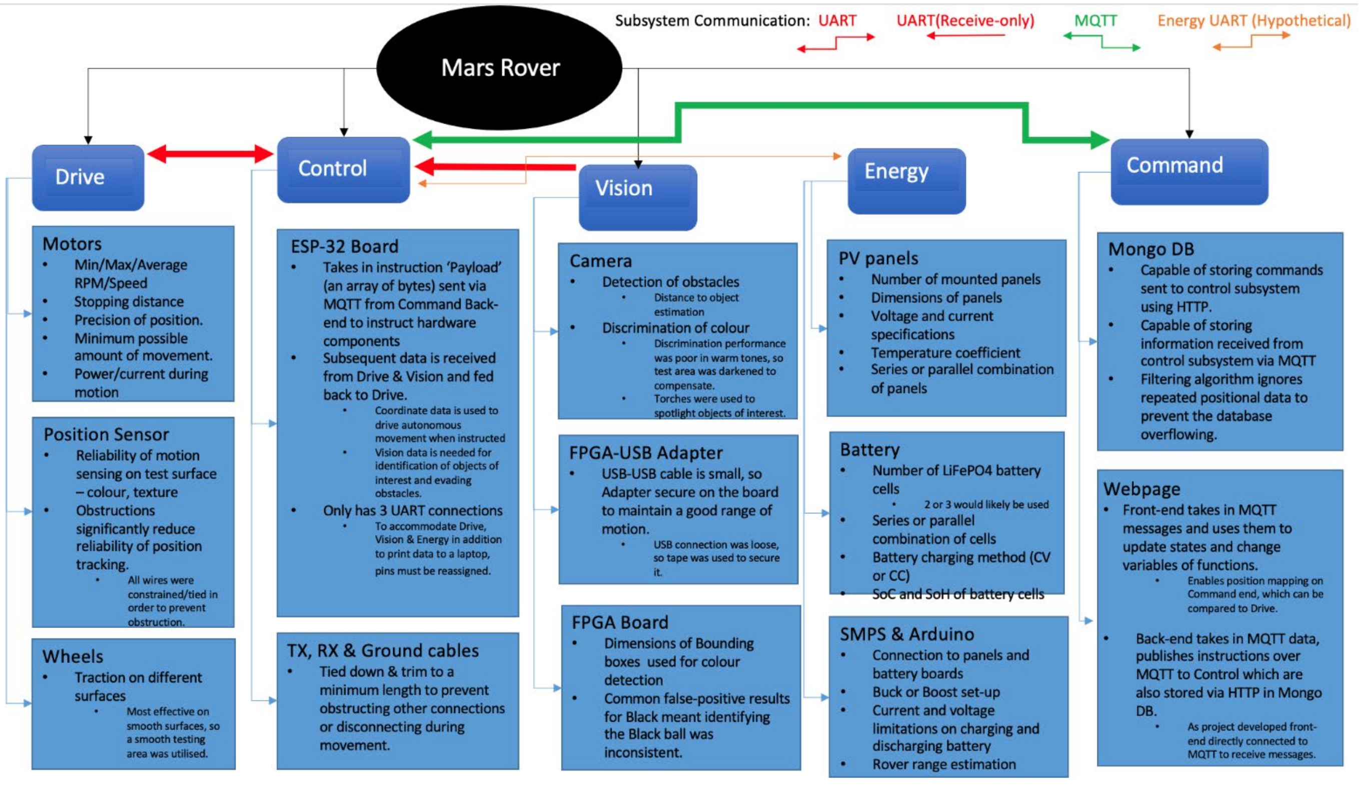
Integration Conclusion:

Intellectual Property Statement:

Intellectual property (IP) is defined as intangible creations of the mind that can be bought and sold like any other property, of which the most globally recognised are patents, trademarks, copyrights and industrial designs. IP law exists mainly to encourage creativity and variation in resulting products/businesses beneficial to society, as well as to protect people's ideas or concepts from being copied and used, manufactured, sold or imported by others. IP rights such as copyrights are unregistered, which means proof of copying must be proven for infringement (violation of IP rights) and the rights are automatically valid. On the other hand, for registered IP such as patents, no proof of copying for infringement is needed but an application is required. Given the nature of this project, and the focus on functionality over aesthetic design, the most relevant type of registered intellectual property to the Mars Rover is patents since design rights do not protect the function of a creation.

Patents exist to protect inventions and encourage innovation. The main criteria for an invention to qualify for a patent are that it must be new, applicable to industry, not obvious, and no information about it can be published before applying for the patent. This means that abstract work such as scientific theories and software coding do not meet the requirements to apply for a patent. For the Mars Rover to qualify for a patent, a major aspect of its functionality would need to be new and an improvement amongst industrial rovers, while also being able to meet all other functional requirements. Potential for novelty and eligibility for a patent could arise from aspects such as the remote control of and communication with the rover, its ability to identify objects and map the local area, and design of the rover charging system.

Appendix A – Structural Diagram



Motors
- Min/Max/Average RPM/Speed
- Stopping distance
- Precision of position.
- Minimum possible amount of movement.
- Power/current during motion

Position Sensor
- Reliability of motion sensing on test surface – colour, texture
- Obstructions significantly reduce reliability of position tracking.
    - All wires were constrained/tied in order to prevent obstruction.

Wheels
- Traction on different surfaces
    - Most effective on smooth surfaces, so a smooth testing area was utilised.

ESP-32 Board
- Takes in instruction 'Payload' (an array of bytes) sent via MQTT from Command Back-end to instruct hardware components
- Subsequent data is received from Drive & Vision and fed back to Drive.
    - Coordinate data is used to drive autonomous movement when instructed
    - Vision data is needed for identification of objects of interest and evading obstacles.
- Only has 3 UART connections
    - To accommodate Drive, Vision & Energy in addition to print data to a laptop, pins must be reassigned.

TX, RX & Ground cables
- Tied down & trim to a minimum length to prevent obstructing other connections or disconnecting during movement.

Camera
- Detection of obstacles
    - Distance to object estimation
- Discrimination of colour
    - Discrimination performance was poor in warm tones, so test area was darkened to compensate.
    - Torches were used to spotlight objects of interest.

FPGA-USB Adapter
- USB-USB cable is small, so Adapter secure on the board to maintain a good range of motion.
    - USB connection was loose, so tape was used to secure it.

FPGA Board
- Dimensions of Bounding boxes used for colour detection
- Common false-positive results for Black meant identifying the Black ball was inconsistent.

PV panels
- Number of mounted panels
- Dimensions of panels
- Voltage and current specifications
- Temperature coefficient
- Series or parallel combination of panels

Battery
- Number of LiFePO4 battery cells
    - 2 or 3 would likely be used
- Series or parallel combination of cells
- Battery charging method (CV or CC)
- SoC and SoH of battery cells

SMPS & Arduino
- Connection to panels and battery boards
- Buck or Boost set-up
- Current and voltage limitations on charging and discharging battery
- Rover range estimation

Mongo DB
- Capable of storing commands sent to control subsystem using HTTP.
- Capable of storing information received from control subsystem via MQTT
- Filtering algorithm ignores repeated positional data to prevent the database overflowing.

Webpage
- Front-end takes in MQTT messages and uses them to update states and change variables of functions.
    - Enables position mapping on Command end, which can be compared to Drive.
- Back-end takes in MQTT data, publishes instructions over MQTT to Control which are also stored via HTTP in Mongo DB.
    - As project developed front-end directly connected to MQTT to receive messages.

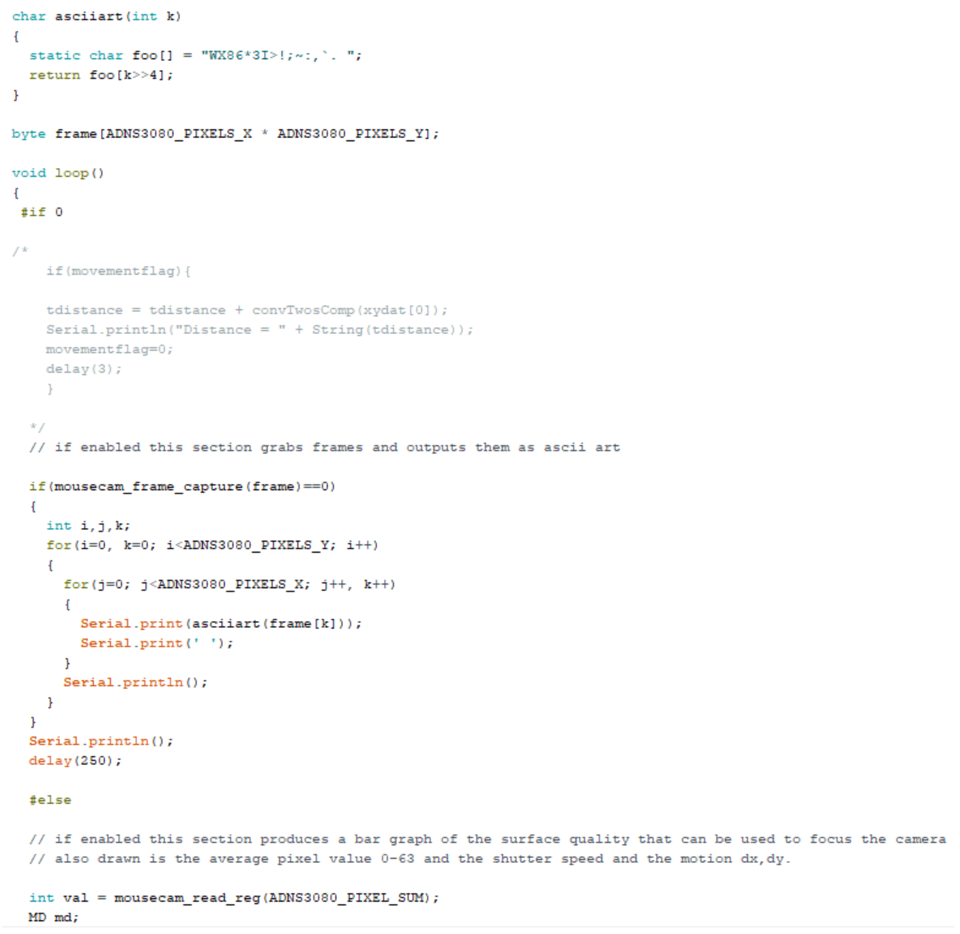
Appendix B – Example Optical Flow Sensor Code and SMPS Code

Here is the example code for the optical flow sensor provided by Dr Zohaib Akhtar

```
#include "SPI.h"

// these pins may be different on different boards
// this is for the uno
#define PIN_SS        10
#define PIN_MISO      12
#define PIN_MOSI      11
#define PIN_SCK       13



#define PIN_MOUSECAM_RESET     8
#define PIN_MOUSECAM_CS        7

#define ADNS3080_PIXELS_X                 30
#define ADNS3080_PIXELS_Y                 30

#define ADNS3080_PRODUCT_ID            0x00
#define ADNS3080_REVISION_ID           0x01
#define ADNS3080_MOTION                0x02
#define ADNS3080_DELTA_X               0x03
#define ADNS3080_DELTA_Y               0x04
#define ADNS3080_SQUAL                 0x05
#define ADNS3080_PIXEL_SUM             0x06
#define ADNS3080_MAXIMUM_PIXEL         0x07
#define ADNS3080_CONFIGURATION_BITS    0x0a
#define ADNS3080_EXTENDED_CONFIG       0x0b
#define ADNS3080_DATA_OUT_LOWER        0x0c
#define ADNS3080_DATA_OUT_UPPER        0x0d
#define ADNS3080_SHUTTER_LOWER         0x0e
#define ADNS3080_SHUTTER_UPPER         0x0f
#define ADNS3080_FRAME_PERIOD_LOWER    0x10
#define ADNS3080_FRAME_PERIOD_UPPER    0x11
#define ADNS3080_MOTION_CLEAR          0x12
#define ADNS3080_FRAME_CAPTURE         0x13
#define ADNS3080_SROM_ENABLE           0x14
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_LOWER      0x19
#define ADNS3080_FRAME_PERIOD_MAX_BOUND_UPPER      0x1a
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_LOWER      0x1b
#define ADNS3080_FRAME_PERIOD_MIN_BOUND_UPPER      0x1c
#define ADNS3080_SHUTTER_MAX_BOUND_LOWER           0x1e
#define ADNS3080_SHUTTER_MAX_BOUND_UPPER           0x1e
#define ADNS3080_SROM_ID               0x1f
#define ADNS3080_OBSERVATION           0x3d
#define ADNS3080_INVERSE_PRODUCT_ID    0x3f
#define ADNS3080_PIXEL_BURST           0x40
#define ADNS3080_MOTION_BURST          0x50

#define ADNS3080_SROM_LOAD             0x60

#define ADNS3080_PRODUCT_ID_VAL        0x17



int total_x = 0;

int total_y = 0;


int total_x1 = 0;

int total_y1 = 0;


int x=0;
int y=0;

int a=0;
int b=0;



int distance_x=0;
int distance_y=0;



volatile byte movementflag=0;
volatile int xydat[2];



int convTwosComp(int b){
  //Convert from 2's complement
  if(b & 0x80){
    b = -1 * ((b ^ 0xff) + 1);
    }
  return b;
  }



  int tdistance = 0;
```

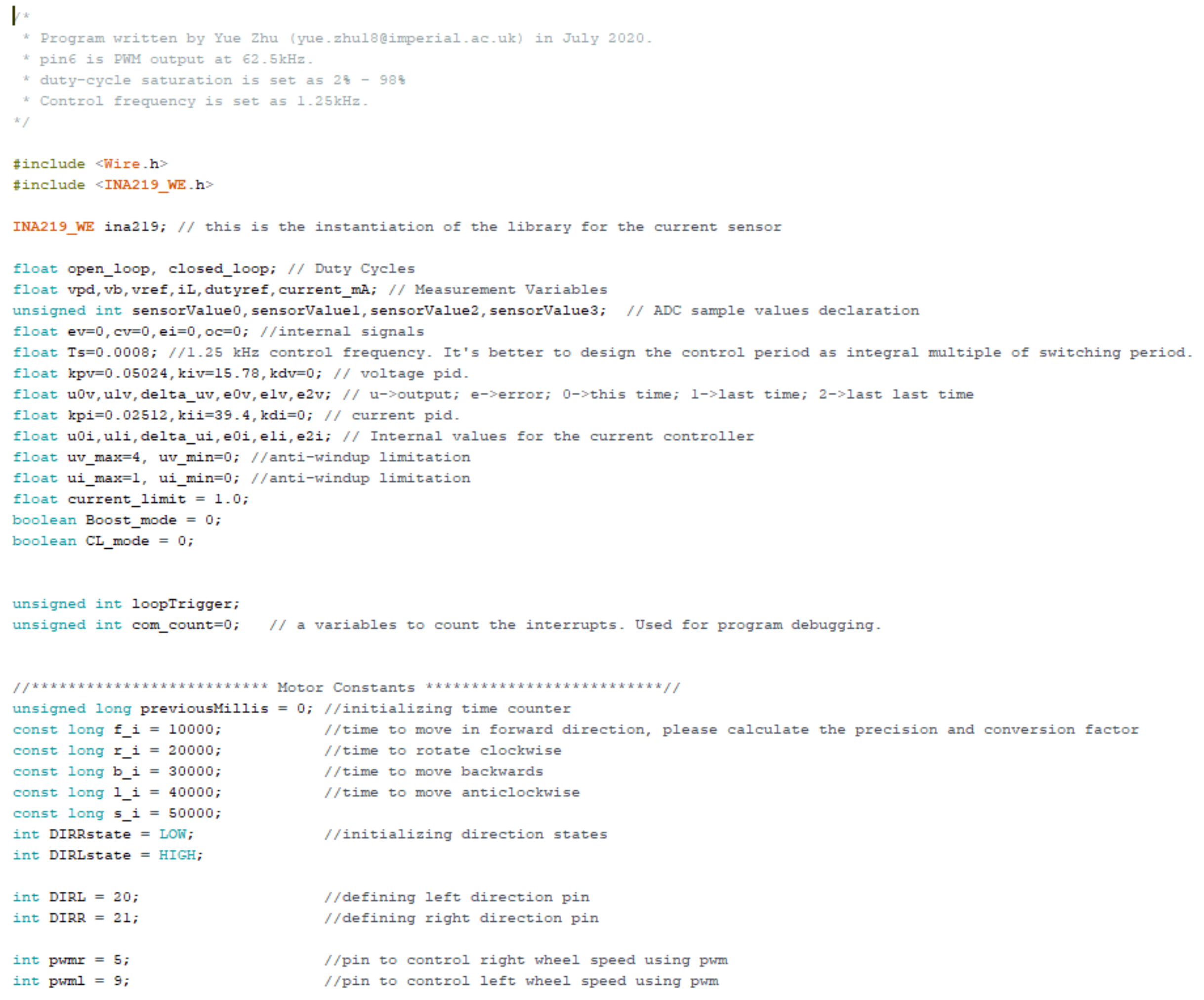
```
void mousecam_reset()
{
  digitalWrite(PIN_MOUSECAM_RESET,HIGH);
  delay(1); // reset pulse >10us
  digitalWrite(PIN_MOUSECAM_RESET,LOW);
  delay(35); // 35ms from reset to functional
}


int mousecam_init()
{
  pinMode(PIN_MOUSECAM_RESET,OUTPUT);
  pinMode(PIN_MOUSECAM_CS,OUTPUT);

  digitalWrite(PIN_MOUSECAM_CS,HIGH);

  mousecam_reset();

  int pid = mousecam_read_reg(ADNS3080_PRODUCT_ID);
  if(pid != ADNS3080_PRODUCT_ID_VAL)
    return -1;

  // turn on sensitive mode
  mousecam_write_reg(ADNS3080_CONFIGURATION_BITS, 0x19);

  return 0;
}

void mousecam_write_reg(int reg, int val)
{
  digitalWrite(PIN_MOUSECAM_CS, LOW);
  SPI.transfer(reg | 0x80);
  SPI.transfer(val);
  digitalWrite(PIN_MOUSECAM_CS,HIGH);
  delayMicroseconds(50);
}

int mousecam_read_reg(int reg)
{
  digitalWrite(PIN_MOUSECAM_CS, LOW);
  SPI.transfer(reg);
  delayMicroseconds(75);
  int ret = SPI.transfer(0xff);
  digitalWrite(PIN_MOUSECAM_CS,HIGH);
  delayMicroseconds(1);
  return ret;

}

struct MD
{
 byte motion;
 char dx, dy;
 byte squal;
 word shutter;
 byte max_pix;
};


void mousecam_read_motion(struct MD *p)
{
  digitalWrite(PIN_MOUSECAM_CS, LOW);
  SPI.transfer(ADNS3080_MOTION_BURST);
  delayMicroseconds(75);
  p->motion =  SPI.transfer(0xff);
  p->dx =  SPI.transfer(0xff);
  p->dy =  SPI.transfer(0xff);
  p->squal =  SPI.transfer(0xff);
  p->shutter =  SPI.transfer(0xff)<<8;
  p->shutter |=  SPI.transfer(0xff);
  p->max_pix =  SPI.transfer(0xff);
  digitalWrite(PIN_MOUSECAM_CS,HIGH);
  delayMicroseconds(5);
}

// pdata must point to an array of size ADNS3080_PIXELS_X x ADNS3080_PIXELS_Y
// you must call mousecam_reset() after this if you want to go back to normal operation
int mousecam_frame_capture(byte *pdata)
{
  mousecam_write_reg(ADNS3080_FRAME_CAPTURE,0x83);

  digitalWrite(PIN_MOUSECAM_CS, LOW);

  SPI.transfer(ADNS3080_PIXEL_BURST);
  delayMicroseconds(50);

  int pix;
  byte started = 0;
  int count;
  int timeout = 0;
  int ret = 0;
  for(count = 0; count < ADNS3080_PIXELS_X * ADNS3080_PIXELS_Y; )
  {
    pix = SPI.transfer(0xff);
    delayMicroseconds(10);
    if(started==0)
    {
      if(pix&0x40)
        started = 1;
      else
      {
        timeout++;
        if(timeout==100)
        {
          ret = -1;
          break;
        }
      }
    }
    if(started==1)
    {
      pdata[count++] = (pix & 0x3f)<<2; // scale to normal grayscale byte range
    }
  }

  digitalWrite(PIN_MOUSECAM_CS,HIGH);
  delayMicroseconds(14);

  return ret;
}

void setup()
{
  pinMode(PIN_SS,OUTPUT);
  pinMode(PIN_MISO,INPUT);
  pinMode(PIN_MOSI,OUTPUT);
  pinMode(PIN_SCK,OUTPUT);

  SPI.begin();
  SPI.setClockDivider(SPI_CLOCK_DIV32);
  SPI.setDataMode(SPI_MODE3);
  SPI.setBitOrder(MSBFIRST);

  Serial.begin(38400);

  if(mousecam_init()==-1)
  {
    Serial.println("Mouse cam failed to init");
    while(1);
  }
}
```

```
  mousecam_read_motion(&md);
  for(int i=0; i<md.squal/4; i++)
    Serial.print('*');
  Serial.print(' ');
  Serial.print((val*100)/351);
  Serial.print(' ');
  Serial.print(md.shutter); Serial.print(" (");
  Serial.print((int)md.dx); Serial.print(',');
  Serial.print((int)md.dy); Serial.println(')');

  // Serial.println(md.max_pix);
  delay(100);


    distance_x = md.dx; //convTwosComp(md.dx);
    distance_y = md.dy; //convTwosComp(md.dy);

total_x1 = (total_x1 + distance_x);
total_y1 = (total_y1 + distance_y);

total_x = 10*total_x1/157; //Conversion from counts per inch to mm (400 counts per inch)
total_y = 10*total_y1/157; //Conversion from counts per inch to mm (400 counts per inch)


 Serial.print('\n');


Serial.println("Distance_x = " + String(total_x));

Serial.println("Distance_y = " + String(total_y));
 Serial.print('\n');

  delay(100);

  #endif
}
```



```
char asciiart(int k)
{
  static char foo[] = "WX86*3I>!;~:,`. ";
  return foo[k>>4];
}

byte frame[ADNS3080_PIXELS_X * ADNS3080_PIXELS_Y];

void loop()
{
 #if 0

/*
    if(movementflag){

    tdistance = tdistance + convTwosComp(xydat[0]);
    Serial.println("Distance = " + String(tdistance));
    movementflag=0;
    delay(3);
    }

  */
  // if enabled this section grabs frames and outputs them as ascii art

  if(mousecam_frame_capture(frame)==0)
  {
    int i,j,k;
    for(i=0, k=0; i<ADNS3080_PIXELS_Y; i++)
    {
      for(j=0; j<ADNS3080_PIXELS_X; j++, k++)
      {
        Serial.print(asciiart(frame[k]));
        Serial.print(' ');
      }
      Serial.println();
    }
  }
  Serial.println();
  delay(250);

  #else

  // if enabled this section produces a bar graph of the surface quality that can be used to focus the camera
  // also drawn is the average pixel value 0-63 and the shutter speed and the motion dx,dy.

  int val = mousecam_read_reg(ADNS3080_PIXEL_SUM);
  MD md;
```

Here is the example code to control the SMPS and motors by Yue Zhu and Dr Zohaib Akthar.



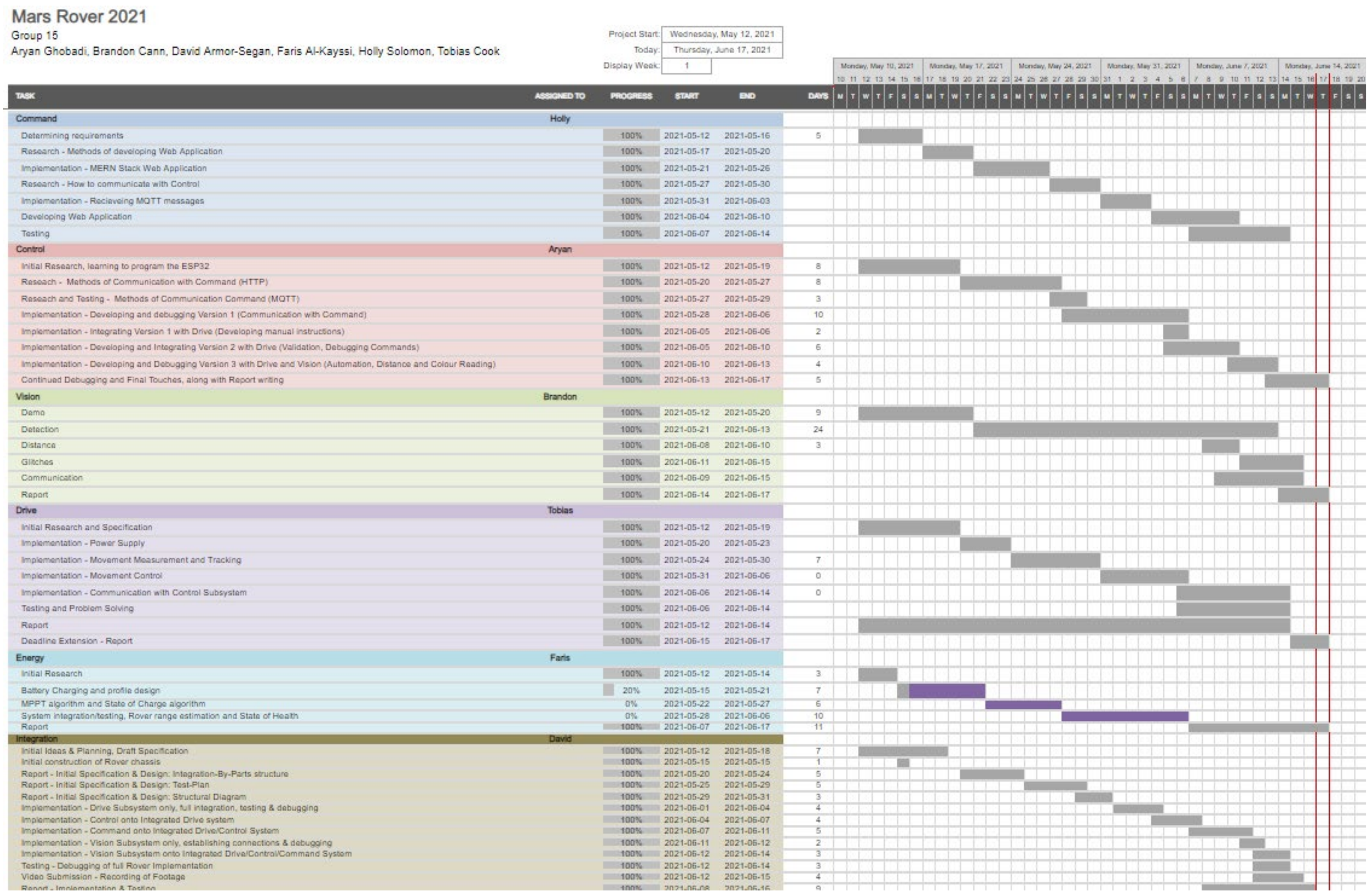
```
/*
 * Program written by Yue Zhu (yue.zhu18@imperial.ac.uk) in July 2020.
 * pin6 is PWM output at 62.5kHz.
 * duty-cycle saturation is set as 2% - 98%
 * Control frequency is set as 1.25kHz.
*/

#include <Wire.h>
#include <INA219_WE.h>

INA219_WE ina219; // this is the instantiation of the library for the current sensor

float open_loop, closed_loop; // Duty Cycles
float vpd,vb,vref,iL,dutyref,current_mA; // Measurement Variables
unsigned int sensorValue0,sensorValue1,sensorValue2,sensorValue3;  // ADC sample values declaration
float ev=0,cv=0,ei=0,oc=0; //internal signals
float Ts=0.0008; //1.25 kHz control frequency. It's better to design the control period as integral multiple of switching period.
float kpv=0.05024,kiv=15.78,kdv=0; // voltage pid.
float u0v,u1v,delta_uv,e0v,e1v,e2v; // u->output; e->error; 0->this time; 1->last time; 2->last last time
float kpi=0.02512,kii=39.4,kdi=0; // current pid.
float u0i,u1i,delta_ui,e0i,e1i,e2i; // Internal values for the current controller
float uv_max=4, uv_min=0; //anti-windup limitation
float ui_max=1, ui_min=0; //anti-windup limitation
float current_limit = 1.0;
boolean Boost_mode = 0;
boolean CL_mode = 0;


unsigned int loopTrigger;
unsigned int com_count=0;   // a variables to count the interrupts. Used for program debugging.


//************************** Motor Constants **************************//
unsigned long previousMillis = 0; //initializing time counter
const long f_i = 10000;           //time to move in forward direction, please calculate the precision and conversion factor
const long r_i = 20000;           //time to rotate clockwise
const long b_i = 30000;           //time to move backwards
const long l_i = 40000;           //time to move anticlockwise
const long s_i = 50000;
int DIRRstate = LOW;              //initializing direction states
int DIRLstate = HIGH;

int DIRL = 20;                    //defining left direction pin
int DIRR = 21;                    //defining right direction pin

int pwmr = 5;                     //pin to control right wheel speed using pwm
int pwml = 9;                     //pin to control left wheel speed using pwm
```

```
//**********************************************************************//



void setup() {

  //*************************** Motor Pins Defining **************************//
  pinMode(DIRR, OUTPUT);
  pinMode(DIRL, OUTPUT);
  pinMode(pwmr, OUTPUT);
  pinMode(pwml, OUTPUT);
  analogWrite(pwmr, 255);       //setting right motor speed at maximum
  analogWrite(pwml, 255);       //setting left motor speed at maximum
  //**********************************************************************//

  //Basic pin setups

  noInterrupts(); //disable all interrupts
  pinMode(13, OUTPUT);  //Pin13 is used to time the loops of the controller
  pinMode(3, INPUT_PULLUP); //Pin3 is the input from the Buck/Boost switch
  pinMode(2, INPUT_PULLUP); // Pin 2 is the input from the CL/OL switch
  analogReference(EXTERNAL); // We are using an external analogue reference for the ADC

  // TimerA0 initialization for control-loop interrupt.

  TCA0.SINGLE.PER = 255; //
  TCA0.SINGLE.CMP1 = 255; //
  TCA0.SINGLE.CTRLA = TCA_SINGLE_CLKSEL_DIV64_gc | TCA_SINGLE_ENABLE_bm; //16 prescaler, 1M.
  TCA0.SINGLE.INTCTRL = TCA_SINGLE_CMP1_bm;

  // TimerB0 initialization for PWM output

  pinMode(6, OUTPUT);
  TCB0.CTRLA=TCB_CLKSEL_CLKDIV1_gc | TCB_ENABLE_bm; //62.5kHz
  analogWrite(6,120);

  interrupts();  //enable interrupts.
  Wire.begin(); // We need this for the i2c comms for the current sensor
  ina219.init(); // this initiates the current sensor
  Wire.setClock(700000); // set the comms speed for i2c

}

 void loop() {
  unsigned long currentMillis = millis();
  if(loopTrigger) { // This loop is triggered, it wont run unless there is an interrupt
   digitalWrite(13, HIGH);   // set pin 13. Pin13 shows the time consumed by each control cycle. It's used for debugging.

   // Sample all of the measurements and check which control mode we are in
   sampling();
   CL_mode = digitalRead(3); // input from the OL_CL switch
   Boost_mode = digitalRead(2); // input from the Buck_Boost switch

   if (Boost_mode){
     if (CL_mode) { //Closed Loop Boost
         pwm_modulate(1); // This disables the Boost as we are not using this mode
     }else{ // Open Loop Boost
         pwm_modulate(1); // This disables the Boost as we are not using this mode
     }
   }else{
     if (CL_mode) { // Closed Loop Buck
         // The closed loop path has a voltage controller cascaded with a current controller. The voltage controller
         // creates a current demand based upon the voltage error. This demand is saturated to give current limiting.
         // The current loop then gives a duty cycle demand based upon the error between demanded current and measured
         // current
         current_limit = 3; // Buck has a higher current limit
         ev = vref - vb;  //voltage error at this time
         cv=pidv(ev);  //voltage pid
         cv=saturation(cv, current_limit, 0); //current demand saturation
         ei=cv-iL; //current error
         closed_loop=pidi(ei);  //current pid
         closed_loop=saturation(closed_loop,0.99,0.01);  //duty_cycle saturation
         pwm_modulate(closed_loop); //pwm modulation
     }else{ // Open Loop Buck
         current_limit = 3; // Buck has a higher current limit
         oc = iL-current_limit; // Calculate the difference between current measurement and current limit
         if ( oc > 0) {
           open_loop=open_loop-0.001; // We are above the current limit so less duty cycle
         } else {
           open_loop=open_loop+0.001; // We are below the current limit so more duty cycle
         }
         open_loop=saturation(open_loop,dutyref,0.02); // saturate the duty cycle at the reference or a min of 0.01
         pwm_modulate(open_loop); // and send it out
     }
   }
   // closed loop control path

   digitalWrite(13, LOW);   // reset pin13.
   loopTrigger = 0;
  }

  //************************** Motor Testing **************************//
  //this part of the code decides the direction of motor rotations depending on the time lapsed. currentMillis records the time lapsed once it is called.
  //set your states
    DIRRstate = LOW;
    DIRLstate = LOW;


    digitalWrite(DIRR, DIRRstate);
    digitalWrite(DIRL, DIRLstate);
  //*******************************************************************//



}


// Timer A CMP1 interrupt. Every 800us the program enters this interrupt.
// This, clears the incoming interrupt flag and triggers the main loop.

ISR(TCA0_CMP1_vect){
  TCA0.SINGLE.INTFLAGS |= TCA_SINGLE_CMP1_bm; //clear interrupt flag
  loopTrigger = 1;
}

// This subroutine processes all of the analogue samples, creating the required values for the main loop

void sampling(){

  // Make the initial sampling operations for the circuit measurements

  sensorValue0 = analogRead(A0); //sample Vb
  sensorValue2 = analogRead(A2); //sample Vref
  sensorValue3 = analogRead(A3); //sample Vpd
  current_mA = ina219.getCurrent_mA(); // sample the inductor current (via the sensor chip)

  // Process the values so they are a bit more usable/readable
  // The analogRead process gives a value between 0 and 1023
  // representing a voltage between 0 and the analogue reference which is 4.096V

  vb = sensorValue0 * (4.096 / 1023.0); // Convert the Vb sensor reading to volts
  vref = sensorValue2 * (4.096 / 1023.0); // Convert the Vref sensor reading to volts
  vpd = sensorValue3 * (4.096 / 1023.0); // Convert the Vpd sensor reading to volts

  // The inductor current is in mA from the sensor so we need to convert to amps.
  // We want to treat it as an input current in the Boost, so its also inverted
  // For open loop control the duty cycle reference is calculated from the sensor
  // differently from the Vref, this time scaled between zero and 1.
  // The boost duty cycle needs to be saturated with a 0.33 minimum to prevent high output voltages


  if (Boost_mode == 1){
    iL = -current_mA/1000.0;
    dutyref = saturation(sensorValue2 * (1.0 / 1023.0),0.99,0.33);
  }else{
    iL = current_mA/1000.0;
    dutyref = sensorValue2 * (1.0 / 1023.0);
  }

}

float saturation( float sat_input, float uplim, float lowlim){ // Saturatio function
  if (sat_input > uplim) sat_input=uplim;
  else if (sat_input < lowlim ) sat_input=lowlim;
  else;
  return sat_input;
}

void pwm_modulate(float pwm_input){ // PWM function
  analogWrite(6,(int)(255-pwm_input*255));
}

// This is a PID controller for the voltage

float pidv( float pid_input){
  float e_integration;
  e0v = pid_input;
  e_integration = e0v;

  //anti-windup, if last-time pid output reaches the limitation, this time there won't be any intergrations.
  if(u1v >= uv_max) {
    e_integration = 0;
  } else if (u1v <= uv_min) {
    e_integration = 0;
  }

  delta_uv = kpv*(e0v-e1v) + kiv*Ts*e_integration + kdv/Ts*(e0v-2*e1v+e2v); //incremental PID programming avoids integrations.there is another PID program called positional PID.
  u0v = u1v + delta_uv;  //this time's control output

  //output limitation
  saturation(u0v,uv_max,uv_min);

  u1v = u0v; //update last time's control output
  e2v = e1v; //update last last time's error
  e1v = e0v; // update last time's error
  return u0v;
}
```

```
// This is a PID controller for the current

float pidi(float pid_input){
  float e_integration;
  e0i = pid_input;
  e_integration=e0i;

  //anti-windup
  if(u1i >= ui_max){
    e_integration = 0;
  } else if (u1i <= ui_min) {
    e_integration = 0;
  }

  delta_ui = kpi*(e0i-e1i) + kii*Ts*e_integration + kdi/Ts*(e0i-2*e1i+e2i); //incremental PID programming avoids integrations.
  u0i = u1i + delta_ui;  //this time's control output

  //output limitation
  saturation(u0i,ui_max,ui_min);

  u1i = u0i; //update last time's control output
  e2i = e1i; //update last last time's error
  e1i = e0i; // update last time's error
  return u0i;
}


/*end of the program.*/
```

Appendix C – Gantt Chart

Here is a picture of the Gantt chart the team used to schedule the project.



Mars Rover 2021

Group 15

Aryan Ghobadi, Brandon Cann, David Armor-Segan, Faris Al-Kayssi, Holly Solomon, Tobias Cook

Project Start: Wednesday, May 12, 2021

Today: Thursday, June 17, 2021

Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END	DAYS
Command	Holly				
Determining requirements		100%	2021-05-12	2021-05-16	5
Research - Methods of developing Web Application		100%	2021-05-17	2021-05-20	
Implementation - MERN Stack Web Application		100%	2021-05-21	2021-05-26	
Research - How to communicate with Control		100%	2021-05-27	2021-05-30	
Implementation - Recieveing MQTT messages		100%	2021-05-31	2021-06-03	
Developing Web Application		100%	2021-06-04	2021-06-10	
Testing		100%	2021-06-07	2021-06-14	
Control	Aryan				
Initial Research, learning to program the ESP32		100%	2021-05-12	2021-05-19	8
Reseach - Methods of Communication with Command (HTTP)		100%	2021-05-20	2021-05-27	8
Reseach and Testing - Methods of Communication Command (MQTT)		100%	2021-05-27	2021-05-29	3
Implementation - Developing and debugging Version 1 (Communication with Command)		100%	2021-05-28	2021-06-06	10
Implementation - Integrating Version 1 with Drive (Developing manual instructions)		100%	2021-06-05	2021-06-06	2
Implementation - Developing and Integrating Version 2 with Drive (Validation, Debugging Commands)		100%	2021-06-05	2021-06-10	6
Implementation - Developing and Debugging Version 3 with Drive and Vision (Automation, Distance and Colour Reading)		100%	2021-06-10	2021-06-13	4
Continued Debugging and Final Touches, along with Report writing		100%	2021-06-13	2021-06-17	5
Vision	Brandon				
Demo		100%	2021-05-12	2021-05-20	9
Detection		100%	2021-05-21	2021-06-13	24
Distance		100%	2021-06-08	2021-06-10	3
Glitches		100%	2021-06-11	2021-06-15	
Communication		100%	2021-06-09	2021-06-15	
Report		100%	2021-06-14	2021-06-17	
Drive	Tobias				
Initial Research and Specification		100%	2021-05-12	2021-05-19	
Implementation - Power Supply		100%	2021-05-20	2021-05-23	
Implementation - Movement Measurement and Tracking		100%	2021-05-24	2021-05-30	7
Implementation - Movement Control		100%	2021-05-31	2021-06-06	0
Implementation - Communication with Control Subsystem		100%	2021-06-06	2021-06-14	0
Testing and Problem Solving		100%	2021-06-06	2021-06-14	
Report		100%	2021-05-12	2021-06-14	
Deadline Extension - Report		100%	2021-06-15	2021-06-17	
Energy	Faris				
Initial Research		100%	2021-05-12	2021-05-14	3
Battery Charging and profile design		20%	2021-05-15	2021-05-21	7
MPPT algorithm and State of Charge algorithm		0%	2021-05-22	2021-05-27	6
System integration/testing, Rover range estimation and State of Health		0%	2021-05-28	2021-06-06	10
Report		100%	2021-06-07	2021-06-17	11
Integration	David				
Initial Ideas & Planning, Draft Specification		100%	2021-05-12	2021-05-18	7
Initial construction of Rover chassis		100%	2021-05-15	2021-05-15	1
Report - Initial Specification & Design: Integration-By-Parts structure		100%	2021-05-20	2021-05-24	5
Report - Initial Specification & Design: Test-Plan		100%	2021-05-25	2021-05-29	5
Report - Initial Specification & Design: Structural Diagram		100%	2021-05-29	2021-05-31	3
Implementation - Drive Subsystem only, full integration, testing & debugging		100%	2021-06-01	2021-06-04	4
Implementation - Control onto Integrated Drive system		100%	2021-06-04	2021-06-07	4
Implementation - Command onto Integrated Drive/Control System		100%	2021-06-07	2021-06-11	5
Implementation - Vision Subsystem only, establishing connections & debugging		100%	2021-06-11	2021-06-12	2
Implementation - Vision Subsystem onto Integrated Drive/Control/Command System		100%	2021-06-12	2021-06-14	3
Testing - Debugging of full Rover Implementation		100%	2021-06-12	2021-06-14	3
Video Submission - Recording of Footage		100%	2021-06-12	2021-06-15	4
Report - Implementation & Testing		100%	2021-06-08	2021-06-16	9

References

[1] E.P.Borobio, A.Bouchaala. (2021, May 14). Mars Rover - Project introduction, deliverables, and timeline.

[2] M.Serozhenko. (2017, Mar 20). MQTT vs. HTTP: which one is the best for IoT? [Online]. Available: https://medium.com/mqtt-buddy/mqtt-vs-http-which-one-is-the-best-for-iot-c868169b3105\

[3] Amr. (2018, July 4). Why MQTT? [Online]. Available: https://1sheeld.com/mqtt-protocol/

[4] Arduino, "Language Reference" [Online]. Available: https://www.arduino.cc/reference/en/

[5] PubSubClient, "API Documentation" [Online]. Available: https://pubsubclient.knolleary.net/api

[6] A.T.Giant. (Unknown). Introduction to MQTT [Online]. Available: https://learn.sparkfun.com/tutorials/introduction-to-mqtt/all

[7] Avago Technologies "ADNS-3080 High Performance Optical Mouse Sensor" ADNS-3080 Datasheet, October 2008

[8] E.Stott. (2021). Github - EEE2Rover [Online]. Available: https://github.com/edstott/EEE2Rover

[9] O'Reilly Media. "Algorithms and Flow Control" [Online]. Available: https://www.oreilly.com/library/view/high-performance-javascript/9781449382308/ch04.html

[10] C.Vislan, Microchip "Getting Started with TCA" TB3217, 2018

[11] Arduino, "Serial" [Online]. Available: https://www.arduino.cc/reference/en/language/functions/communication/serial/

[12] A.Rosebrock (2015, Jan 19). Find Distance from Camera to Object/Marker using Python and OpenCV [Online]. Available: https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/

[13] E.Stott (2021, Jun 3) Piazza [Online]. Available: https://piazza.com/class/koiow33gz3578w?cid=228

[14] MongoDB, "MERN Stack" [Online]. Available: https://www.mongodb.com/mern-stack

[15] S.Cope (2021, Jan 5) ''Using MQTT over WebSockets with Mosquitto" [Online]. Available: http://www.steves-internet-guide.com/mqtt- websockets/

[16] S.Cope (2020, Jun 9) "Using the Node.js MQTT Client – Starting Guide" [Online]. Available: **http://www.steves-internet-guide.com/using-node-mqtt-client/**

[17] HiveMQ (2015, Feb 16) "Quality of Service 0, 1 and 2 – MQTT essentials: Part 6" [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/

[18] bezkoder (2021, April 7) "React.js + Node.js + Express + MongoDB example: MERN stack CRUD App" [Online]. Available: https://bezkoder.com/react-node-express-mongodb-mern-stack/