



DEBONAIR: MARS ROVER PROJECT REPORT

ELEC50008 – Electronics Design Project 2

Margherita Contri – 01749008

Yash Rajput – 01776074

Ruwan Silva – 01505106

Simon Staal – 01719944

Khayle Torres – 01753211

Yuna Valade - 01765409

Table of Contents

1	Introduction.....	2
1.1	Purpose, Intended Audience and Use.....	2
1.2	System Requirements.....	2
1.3	Project Management.....	3
1.4	Intellectual Property (Lecture Reflection).....	5
2	Structural Design	5
2.1	Overview	5
2.2	AWS Instance	6
2.2.1	MQTT Broker.....	6
2.2.2	NginX Web Server	7
2.2.3	React Web-App	7
2.2.4	Node.js REST API	7
2.3	ESP32 Microcontroller	9
2.3.1	Inter-module Communication.....	9
3	Functional Design & Implementation	11
3.1	– Command (Server).....	11
3.1.1	Front-End Web App	11
3.1.2	Back-End REST API.....	15
3.1.3	Security	22
3.2	Control (ESP32)	23
3.2.1	UART	24
3.2.2	SPI.....	24
3.2.3	MQTT	25
3.3	Vision.....	25
3.3.1	Image processor.....	26
3.3.2	HSV Converter.....	26
3.3.3	Filter	27
3.3.4	Performance Metrics	28
3.3.5	Variable Lighting	28
3.3.6	Distance Calculation.....	30
3.3.7	Communication of Vision.....	30
3.3.8	Testing.....	30
3.4	Drive.....	31
3.4.1	The Design Process	31
3.4.2	Position Control	32
3.4.3	Speed Control.....	33
3.4.4	Code Structure	34
3.4.5	Controlling movement in different modes	34
3.5	Power	36
3.5.1	Cell Characterisation	36
3.5.2	State of charge (SOC)	36
3.5.3	State of health (SOH)	37
3.5.4	Solar panel characterisation	37
3.5.5	Maximum power point tracking (MPPT).....	38
3.5.6	Charging method and balancing	38
3.5.7	Overall integration of system.....	40
3.6	Improvements.....	40
4	Conclusion	40

5	Bibliography	41
6	Appendix	46
6.1	MQTT Custom Certificate.....	46
6.2	External Assets used in Web-App	46

1 Introduction

1.1 Purpose, Intended Audience and Use

The design and deployment of autonomous rovers have been of vital importance in the exploration of high-risk environments that humans are unable to access easily. This is typically for planetary exploration with the aim of collecting information about the terrain and retrieval of soil, rock or dust samples. The need for autonomous performance is due to the inability to operate a rover in real time as radio signals travel too slow for real-time/near-real-time communication. NASA report the time taken for a signal to travel from Earth to Mars is generally between 5 to 20 minutes depending on planet positions [1], highlighting the importance of reliable autonomous operation. With this in mind, these rovers must function with minimal assistance from the operators when it comes to navigation and data collection, but still need human input to provide a desired destination and other details. Providing the rover with a simple visual recognition capacity to judge its surroundings allows an increase in the speed of reconnaissance. Combining this with a reliable power source that is able to recharge using solar energy, a fully functional autonomous rover can, in theory, be manufactured. The goal of this project was to design and build such a system, capable of mapping out its working environment, detecting obstacles as well as sending and receiving data to a remote user interface.

1.2 System Requirements

To develop an integrated rover system, 3 different operational modes were deemed necessary to ensure Codename: Debonair could handle a variety of situations:

- **Exploration:** The rover autonomously explores the target environment, identifying and reporting obstacle positions.
- **Coordinate:** Given a set of destination coordinates, the rover will autonomously navigate through the environment to reach the destination while avoiding obstacles.
- **Remote control:** The rover drive is fully controlled by the user, allowing them to explore the environment as they see fit.

A web application would be designed as the entry point to the system, where the rover could be controlled and information regarding rover position, status and detected obstacles would be available in real-time. Reliable and fast communication is vital to ensure that the user receives up to date information, and instructions sent to the rover are received and acted upon. As such, end-to-end communication latency should be kept below 200ms to keep the rover responsive. Security is another important requirement, as this system could be used in potentially dangerous environments to gather proprietary information, so the system should be only interfaceable via the web-app, and any wireless communication should be encrypted to prevent data leakage. Debonair should be able to identify any of the obstacles provided with the project kit, and due to COVID limitations regarding finding appropriate testing environments, an initial environment size of 4m² was designated.

1.3 Project Management

At the start of the project, initial meetings were held to reallocate responsibilities. The key changes that were made were:

- Command (Yuna Valade) was now solely responsible for the front-end webapp design.
- Control (Simon Staal) would take on responsibility for the back-end design and other miscellaneous server services in addition to his current responsibilities.
- Integration (Ruwan Silva) would take on a flexible role to provide assistance with inter-module communication and any other features.

These changes were made to ensure enough resources were allocated to provide a dynamic and fluid user interface, with the remaining module scopes being left largely unchanged.

The 5 weeks of the project were blocked out to use the first 1.5 weeks for initial research and the next 1.5 weeks to develop basic functionality to have a bare-bones system to allow for testing. Over the following 1.5 weeks, gradual functionality would be added under an agile development model [2] until all requirements were met. The final half-week was allocated to ensure sufficient time was left to complete the project deliverables. Under this framework, regular bi-weekly meetings were held, with team members meeting in-person when possible to combine module tests in order to perform system-wide analysis.

Mars Rover 2021

Debonair Phil

Yuna Valade, Simon Staal, Khayle Torres, Margherita Contri, Yash Rajput, Ru

Project Start: Wednesday, 12 May 2021

Project End:	Thursday, 17 June 2021
--------------	------------------------

Display Week:	1
---------------	---

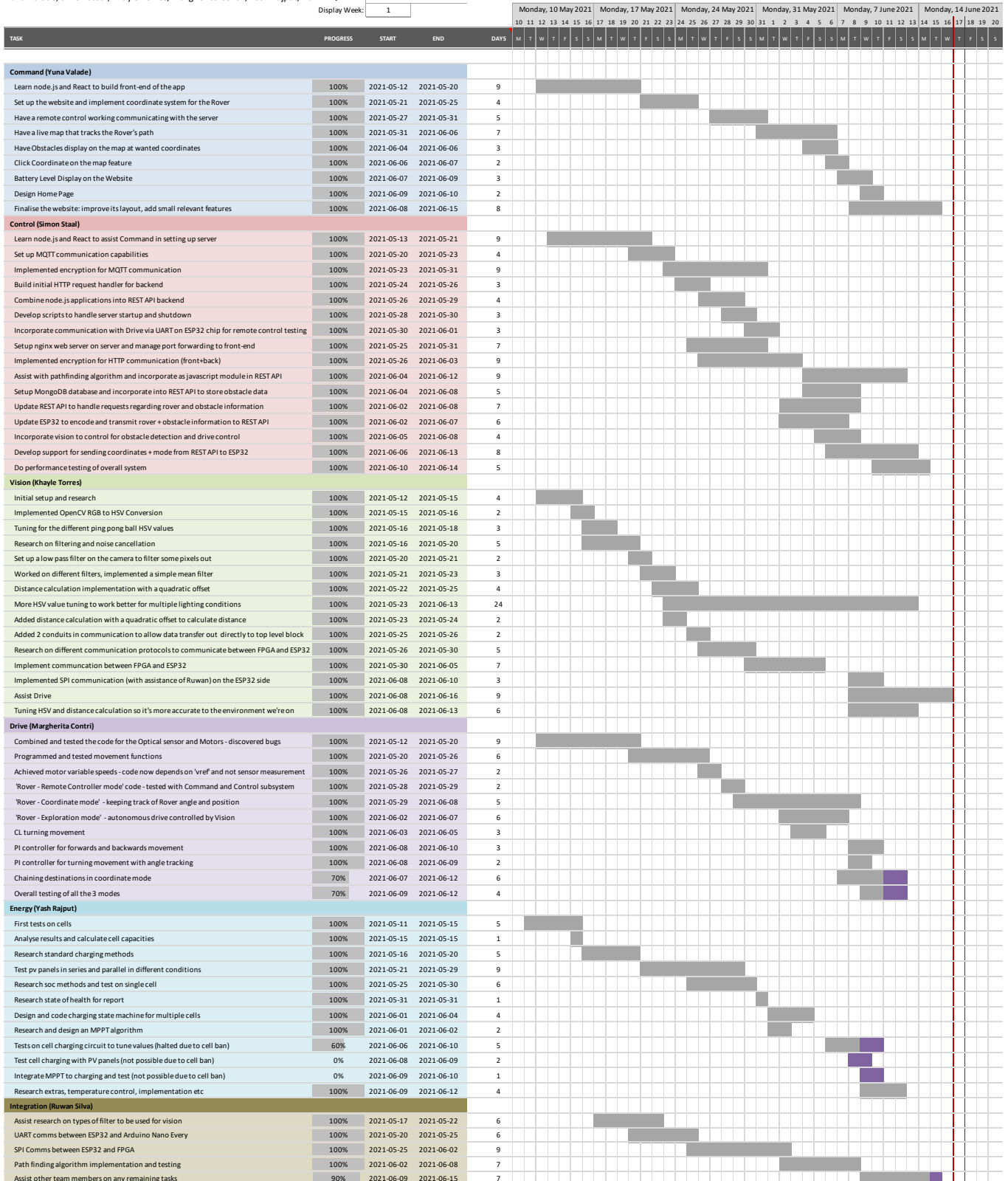


Figure 1.1 - Gantt Chart

1.4 Intellectual Property (Lecture Reflection)

Intellectual property (IP) can be defined as something created by one's mind, and which can be owned and bought or sold. IP rights are defined as negative rights, meaning that they give the IP owner the right, which can be registered or unregistered, to stop other people using what is protected by the right. Registered rights need to have an application filed for them (they grant a monopoly right) and once obtained there is no need to prove copying for infringement. These include patents, registered designs etc. Unregistered rights arise automatically and there is a need to prove copyright for infringement. This can include copyright and unregistered design rights. Types of IP protection can be copyright, trademarks, designs, patents etc. Copyright protects original expression where the author is the first owner unless the work was commissioned (company/employer is the first owner) and last for the life of the creator and 70 years after their death. Trademarks are symbols that identify services or products from a particular company, lasting 10 years, but can be renewed indefinitely. Designs protect the appearance of a product, but not its functionality and are usually owned by the designer (or employer if work is commissioned) and last for 25 years. Patents protect new inventions, which must be something that can be used or made. The owner can prevent people from using, making, selling, and importing the invention and they last for 20 years. For the project Debonair none of these forms of IP protection would be applicable. Copyright mainly applies to creative media such as plays, films, songs etc. Trademarks apply to symbols, which we do have, but it is not unique or original. The appearance of our rover is not unique ruling out design and an autonomous rover is not a new invention, meaning a patent application would be rejected.

2 Structural Design

2.1 Overview

The overall rover system has 2 primary structural nodes: an onboard ESP32 microcontroller with integrated wireless capabilities, and the cloud-based infrastructure for the web-app. These two nodes must be able to exchange information, with the ESP32 relaying that information to the other on-board nodes, and the user interfacing with the entire system through the web app. A diagram providing the overview of how this is achieved can be seen below:

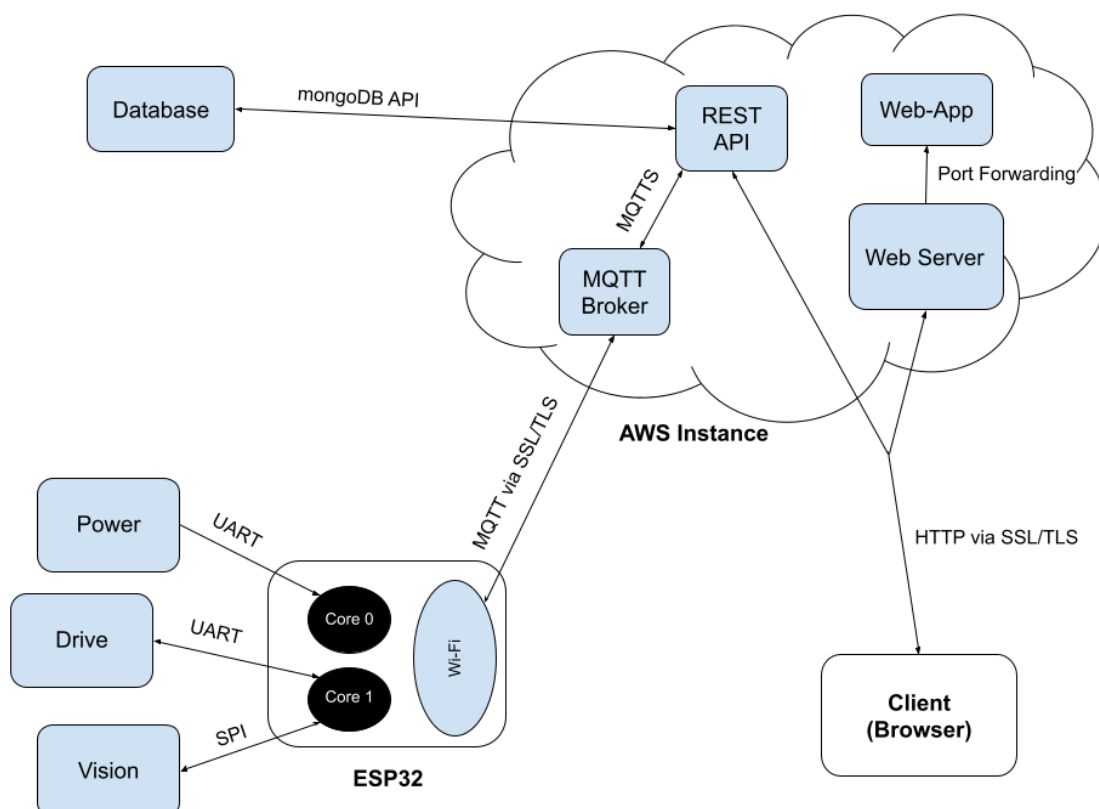


Figure 2.1 - Overview of subsystem architecture

In order to meet the overall system requirements, the core design philosophy employed for the development of this system focused on ensuring reliable, rapid, and secure communication between the front-end user and the rover. Furthermore, maintaining a flexible server infrastructure capable of easily implementing new features or upscaling existing functionality was of vital importance to fit our development model.

2.2 AWS Instance

AWS EC2 was used to host the majority of the web services and infrastructure required for command, as they offer a stable, standardised platform [3] from which any required applications can be built and deployed. Four primary services were required for this system to ensure the necessary communication protocols were available for smooth user-to-rover communication:

Service	Protocol Associated	Port(s) used
Mosquitto Broker	MQTT / MQTTS	1883: (MQTT localhost only) 8883: (MQTTS - encrypted)
NginX Web Server	HTTP / HTTPS	80: (HTTP) 443: (HTTPS)
React Web App	HTTP / NAT	3000: (HTTP) 80/443: (available through port forwarding)
Node.js REST API	HTTP / HTTPS	8080: (HTTP – disabled in deployed build) 8443: (HTTPS)

2.2.1 MQTT Broker

To interact between the rover and server, a suitable communication protocol needed to be selected to ensure the structural design principles outlined above were met. Various protocols were considered, including raw TCP or UDP, which as transport layer protocols were a bit too low level to be desirable, and HTTP, but after some research, MQTT, a more lightweight [4] application layer protocol was selected. MQTT uses a publish/subscribe (pub/sub) communication model as opposed to the more traditional client-server architecture employed by other protocols, where a broker manages connections between all the clients connected to it, allowing to maximize the available bandwidth [5] [6].

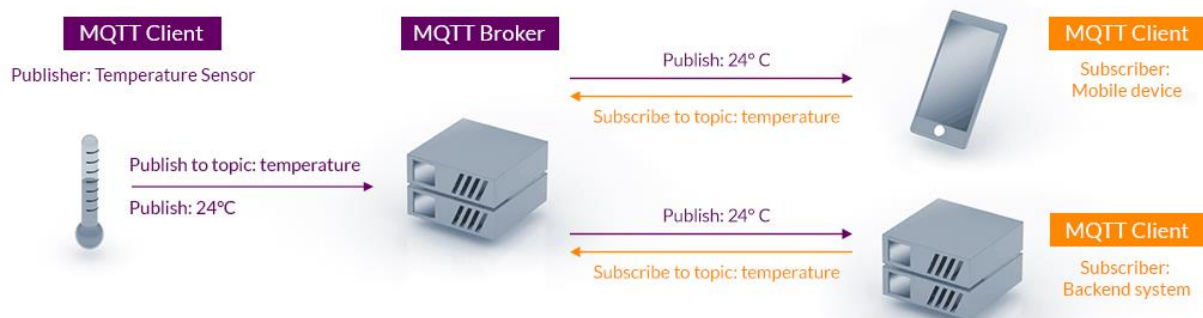


Figure 2.2 - MQTT Protocol Overview [6]

This protocol is ideal for an Internet of Things (IoT) type system, and whilst in this particular case there are only 2 nodes in the overall system, using such a protocol leaves the possibility of easily increasing the number of nodes (i.e. adding more servers or rovers) without having to fully rework inter-node operations. A TCP/IP connection is initially made between the client and broker, ensuring reliable communication, and user authentication as well as communication using Secure Sockets Layer (SSL)/Transport Layer Security (TLS) can be used to achieve a high level of security. Furthermore, the overhead of MQTT is extremely low, with each message consisting of a fixed 2-byte header, an optional variable header, a

quality-of-service (QoS) level, and a message payload of up to 256 MB, meaning it should have a minimal impact on the latency of communication between the rover and web-app.

The quality-of-service level allows the reliability of messages to be specified, with 3 defined levels: 0 meaning messages are received at most once, 1 meaning they are received at least once, and 2 meaning exactly once. In this case, a QoS of 1 is sufficient for messages sent, as receiving duplicate messages should not be an issue, but instructions from the server / obstacles detected from the rover must be received at the other end. Furthermore, for the broker used (Mosquitto), using a QoS of 1 has a minimal impact on the overall broker CPU usage and message latency in the <2000 publisher range when compared with a QoS of 0 [7], meaning very little performance-wise is sacrificed for this increased reliability.

2.2.2 NginX Web Server

To access a web-app, a typical client connecting through a browser would need to make a HTTP/HTTPS request to a given endpoint. Whilst a web-server is not strictly necessary for this, and direct connections could be made to the web-app itself, it offers a several advantages. To improve the accessibility of the web-app, the AWS instance's public IP address is dynamically mapped to the hostname `debonair.duckdns.org` using DuckDNS, a free service which can dynamically point a DNS to an IP [8]. When a browser attempts to access this hostname, it will try to connect to ports 80/433 by default, and so by using an HTTP server like NginX Open Source [9], the instance will automatically forward any connections made to these ports to port 3000 instead, where the web-app is running. This ensures that any changes or issues with web-services can be solved quickly and directly, rather than having to rely on an external web-hosting service, as well as minimising costs. Additionally, SSL/TLS certificates can be installed to the web-server to ensure any data being transferred can be kept encrypted and secure.

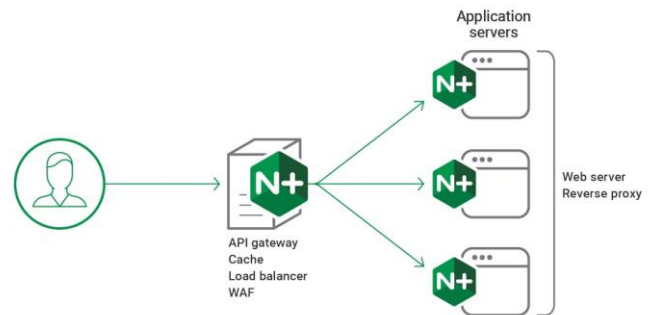


Figure 2.3 - NginX Web Server [9]

2.2.3 React Web-App

This is the front-end entity on the AWS instance that clients using web-browsers will actually interact with. The web-app was developed with Client-Side Rendering architecture in mind, where a single request is made to the server to load the main skeleton of the app, with content then being dynamically generated using JavaScript. This means that after an initially slower load than Server-Side rendering, the web-app can render much faster and have richer site interactions [10]. These interactions cause the client to make HTTPS request to the back-end REST API on behalf of the web-app, updating the rendered webpage with required information or sending instructions to the rover.

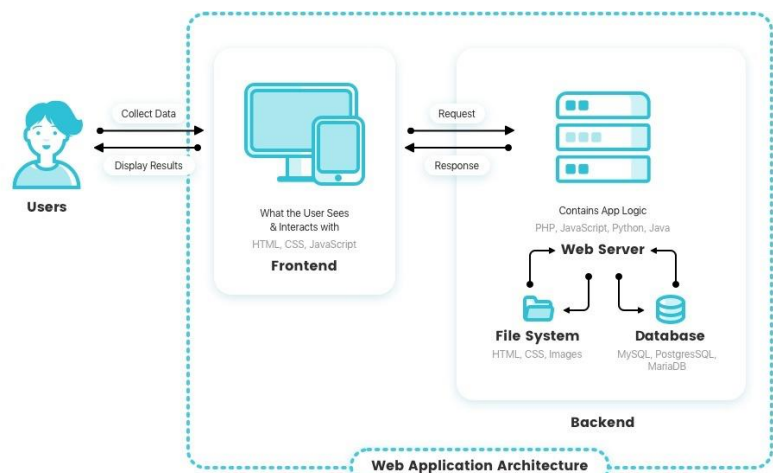


Figure 2.4 - Web-app Architecture [10]

2.2.4 Node.js REST API

In web application architecture, the frontend client communicates with a backend server that contains all the application logic and functionality of the app. A Representation State Transfer (REST) API is an

application programming interface which conforms to the design defined by Dr. Roy Feilding in his 2000 doctorate dissertation [11]. Such APIs usually take advantage of HTTP when used for web APIs, meaning that back-ends can be designed in such a style without the need of installing additional libraries or software. Unlike Simple Object Access Protocol (SOAP), REST is not constrained to XML, but can return XML, JSON, YAML or any other format depending on what the client requests. This flexibility, as well as its lower overhead and better performance made it ideal for this project. Since it's built over HTTP, HTTPS can also be used to ensure any data sent in response by the API is encrypted and securely delivered back to the frontend.

When designing the API for this project, the following constraints needed to be met for it to be considered RESTful [12]:

- A client-server architecture needs to be maintained, meaning that the client (frontend) and server (backend) should be independent from one another and allowed to evolve individually, creating a separation of concerns, and allowing changes to be made to the front or back-end freely from one another.
- The API is stateless, meaning that requests to the API are separate and unconnected, and each call contains all the data necessary to complete itself successfully. Thus the API should not rely on information being stored on the server when responding to calls, keeping the application as scalable as possible by reducing memory requirements. A database must therefore be connected to the backend to enable data storage, with the backend querying the database for information requested by the client.
- The client should be instructed on how to use cacheable data to minimise the number of requests by encouraging the client to store information for an amount of time. This reduces the number of interactions with the API, reducing internal server usage, and providing the front-end application with the tools to be as fast and efficient as possible, as it can spend less time waiting for responses from the API.
- The API should have a uniform interface to allow for the independent evolution of the application without having its' services tightly coupled to the API layer itself, letting the client and server talk to each other in the same language. Since communication was already done over HTTP, URI resources were used in combination with different HTTP request methods, such as GET and POST, to represent essential CRUD (Create, Read, Update, Delete) functionality. JSON objects were employed in conjunction with this to provide an unchanging, standardized means of communicating between server and client.
- The API system should be layered, with each layer having a specific functionality and responsibility, and different layers of the architecture working together to build a hierarchy that can keep the application scalable and modular. Since this API needs to handle communications from the ESP32 via MQTT, a 3 layered hierarchical system was developed, with all logic driven by the interactions from the front end.
- Code should be able to be transmitted via the API for use within the application when requested, extending client functionality. This final constraint is considered optional and not adopted as often, as the transmission of code can not only cause issues for Web APIs that are consumed across multiple languages, but more importantly in this case raises security concerns.

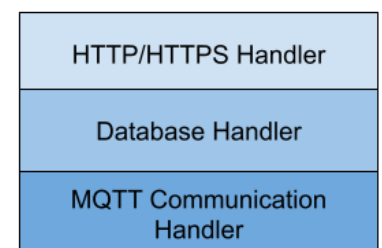


Figure 2.5 - Conceptual API Hierarchy

Together, these constraints formed the core structure of the design principles for the back-end design and will be elaborated upon when discussing its' functional design.

2.3 ESP32 Microcontroller

Communication between the subsystems is integral to a fully functioning rover. Information needs to be received by the server from the Vision subsystem detecting obstacles, the Energy subsystem about

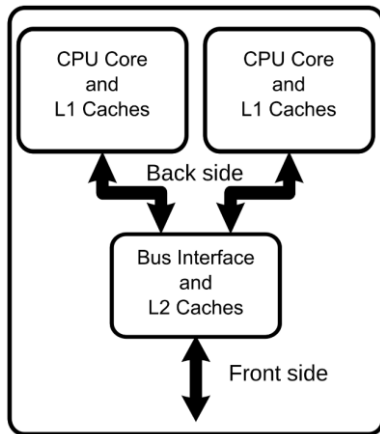


Figure 2.6 - Dual Core Interaction

battery levels and details regarding positioning from the Drive subsystem, as well as sending instructions regarding coordinates or directions to Drive. The ESP32 acts as the hub for all communication between the user interface and rover modules themselves, needing to communicate with both Arduino Nano Every in the Drive and Energy subsystems, as well as the FPGA for the Vision subsystem. The chip comes with 2 Xtensa 32-bit LX6 microprocessors: core 0 and core 1, meaning code can be run in parallel on both cores to reduce the workload of a single processor [13] and improve performance.

Since Drive and Vision are used in tandem with one another during exploration mode to identify and avoid obstacles, processing for both were done in one core to avoid having to pass data through memory, which would slow down both processors. The second core was therefore used to handle data from the Energy module and transmit it to the server, where it could be relayed to the user, and range calculations using data from Drive could potentially be made.

2.3.1 Inter-module Communication

To communicate between the rover modules, a few options were researched, including I2C (inter integrated circuit), UART (universal asynchronous receiver transmitter) and SPI (serial peripheral interface) [14]. Of the three protocols, SPI is the fastest, in some cases being up to 3 times faster than UART [15], and thus it initially seemed like a promising protocol. However, the ESP32 has a total of 4 SPI buses, but one is connected to flash cache, and another is connected to memory [16], meaning that only 2 are available for use. Furthermore, the SPI library only supports the Arduino as a master device within the IDE [17]. This is an issue, as although the ESP-IDF offers an SPI slave driver [18], the recommendation for this project was to use the Arduino IDE to program the ESP32, and therefore it could only be a master.

Because of this, it was decided that UART would be used for communicating with the Drive and Energy subsystems, while SPI would support information transfer with the Vision subsystem. The ESP32 has exactly two UART/serial ports that can be used to communicate with the Arduinos, and because of its' straightforward communication protocol, and the fact that the same base code/process could be used on both Arduinos and the ESP32, it was an ideal fit. To communicate with the server, an MQTT client was created using the PubSubClient library [19], which supported authentication, and could potentially be implemented over a secure Wi-Fi client to allow for encrypted packets to be sent from the ESP32 to the server.

2.3.1.1 UART

UART initialization is different for the ESP32 and the Arduino Nano Every. For the ESP32 there are multiple GP/IO (general purpose input/output) pins on the header that can be used for transmitting and receiving. The user must assign one pin as the transmitter and another as the receiver when the serial port is defined in the code, with the baud rate and transfer protocol being defined as well. The baud rate states the number of bits that are transmitted/received per second. The protocol covers the number of bits per byte

of data, whether an even or odd parity bit are used and the number of stop bits contained in the data packet. The default protocol was selected: 8-bit, with no parity bit and one stop bit.

For the Arduino Nano Every, the only variable that must be declared when initializing the serial port is the baud rate as the transfer and receive pins are already defined and appear as TX and RX pinouts on the header connected to the Nano Every for the Drive and Energy subsystems. The baud rates on both boards would have to match for successful transfer of information and the TX and RX pins must be cross connected. Once serial ports have been initialized, information can be sent and received via the defined pins on ESP32 and the TX and RX pins on the Arduino Nano Every. This information can be in the form of all standard variable types (ints, chars etc.), but only chars were used to standardize communication.

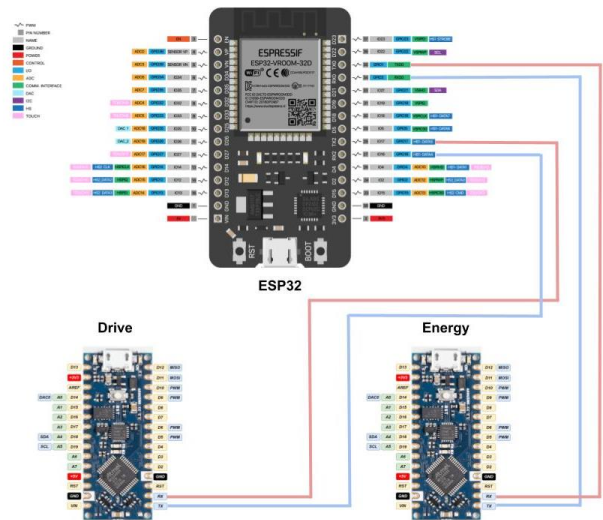


Figure 2.7 - UART Wiring between modules

2.3.1.2 SPI

SPI (Serial Peripheral Interface) is a synchronous serial communication that is used for short distance communication in embedded systems [20]. SPI uses 4 pins, MOSI (Master Out Slave In), MISO (Master In Slave Out), SS (Slave Select) and SCK (Slave clock). The master will set the SS pin to low when a data transfer is wanted from the slave, and then provide a clock at the rate at which each bit is transferred. At each rising/falling clock edge a bit is sent/received by either the master or the slave and is then shifted to send the next bit on the following clock edge. In the case of the rover, the MSB is shifted out first moving towards the LSB. If transmission has finished the master sets the SS pin to high indicating end of transmission with the specific slave.

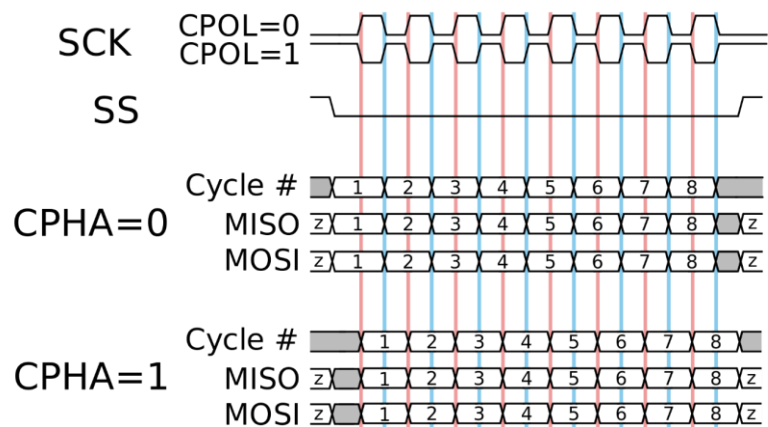


Figure 2.8 - SPI Wiring [20]

When implementing SPI on the FPGA side 2 conduits were made on the **EEE_IMGPROC.v** block to send and receive data. This is because data needs to enter and leave the image processing block to detect which ball the camera is currently seeing. One of the conduits is an output to take a command generated by the block to do a specific task (e.g. rotating right, moving forward) and send it to the top level block. That data is then stored in a register which, when the SS pin goes low, sends the data to the ESP32.

Whenever the slave clock edge is transmitted by the master (ESP32) the transmitted MSB is shifted by 1 across to the left. When data transmission is finished, it is then forwarded to either the Drive or the web server depending on the command Vision specified.

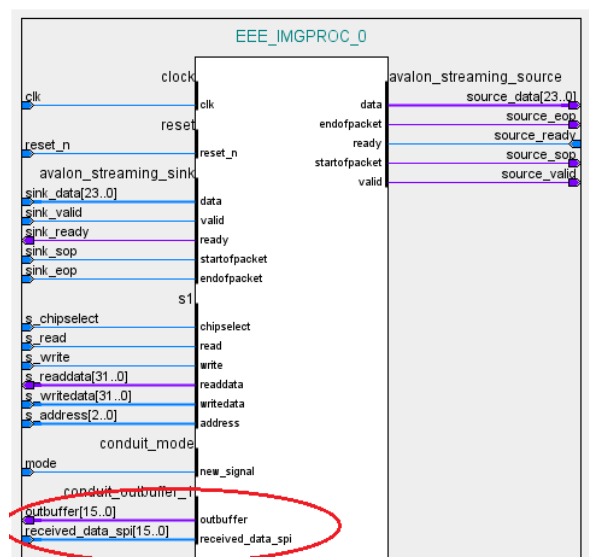


Figure 2.9 - Vision I/O Block

The other conduit is an input to the **EEE_IMGPROC.v** block, which is used as a flag to indicate whenever the rover has finished recording a specific object and to move on to try to discover other objects that have not been recorded yet. This is necessary because Control filters out bad distance calculations by taking the most common calculated length within 100 loops and Vision will only keep sending distance calculations until it receives a flag to do otherwise.

3 Functional Design & Implementation

For the functional design of the overall system, a top-down approach was employed, starting with the needs of the client outlined in section 1.2 and defining the requirements for each module by building up a tree of dependencies.

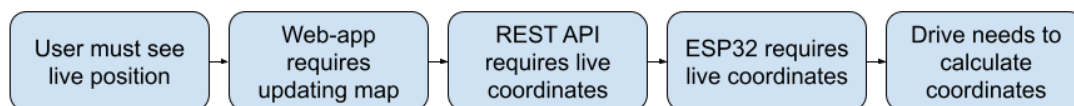


Figure 3.1 - Dependency tree for position requirement

3.1 – Command (Server)

3.1.1 Front-End Web App

The front-end is the interface users directly interact with, which should be intuitive and contain user-friendly features with clear purposes. That is why, this website is designed in a simplistic layout offering essential information on Debonair. The rover's position, battery level and detected obstacles as it discovers its environment are accessible, with all these elements updated real time. Switches on different tabs allow users to freely choose between three different modes:

- **Exploration:** The rover autonomously explores its environment, identifying its' encountered obstacle's position and colour. The rover's location and the data regarding the obstacles are updated on the website's map in real time.
- **Coordinate:** After providing a pair of coordinates, the path finding algorithm will return a set of points to optimally get to the destination. The rover will autonomously navigate following the optimized path.
- **Remote Control:** The user manually controls the rover around its environment.

The website was developed using a combination of React and Nodejs. Not only were these tools heavily recommended in the specifications for the Command subsystem but, additionally, React is a "JavaScript library for building user interfaces" [21], making it perfectly suited for the job. Furthermore, as a beginner who never coded in JavaScript nor with React, when creating a project its structure is clear and easy to navigate. Two essential directories are noticeable: **public** and **src** which essentially constitute the whole display of the website. The public folder contains all the files that are available directly via a URL, including the HTML files that are rendered by the browser and public image assets. The src folder instead contains all the files which style the website and provide interaction i.e., all the JavaScript files with their CSS files. The key files in both folders are index.html and index.js. Essentially React uses Webpack to compile all JavaScript linked files into "a single bundle" [22], which then "injects the scripts" into index.html and builds the page [22]. This gives freedom for web developers to either write all their code in these two files or branch out into multiple separate subfiles which are all linked. Since there are also a lot of resources available, all these reasons made React an ideal library to develop the web-interface.

After learning basics of JavaScript and how to use Nodejs' command-line interface (CLI), as well as getting accustomed to React's environment, front-end development began in earnest. To have a streamlined

design process, it was vital to first have a general idea of the webpage layout. This ensured there was a clear picture to organize where features could be implemented and have a more visual sense of what to work towards. Having a webpage with multiple tabs was deemed optimal as each tab would indicate a different feature, making the layout intuitive and user friendly, with the skeleton for the web-app based on a pamphlet for a dynamic navigation tab [23]. Each page is coded in JavaScript and linked using “Router”, “Switch” and “Route” from the ‘react-router-dom’ library. The “Router” feature allows developers to display different pages to their users, “Route” links one page to a particular path and “Switch” allows the webpage to load one route at a time. All features facilitated the implementation of the links behind each click of the navigation bar.

The “Coords” tab was the first page to be designed with the implementation of two input boxes to intake a pair of coordinates X and Y. An online guide [24] facilitated the implementation of these text fields with a creative touch. They were imported from Material UI [25] which is a website providing creative designs for buttons, textboxes, and icons, and was used numerous times to create webpage assets. By defining the type of the Text Fields to numbers, it limits the user’s input to only numbers, avoiding any cases of letters as coordinates. When coordinates x and y are inputted, they trigger an event function which stores typed numbers in an array which is updated as the user types. The submit button triggers a different event function which will take the first element of the array and sends it to the REST API. The node.js library **axios** was used to make all HTTP/HTTPS requests [26] to the back end. POST and GET requests are sent to different server endpoints to ensure a wide variety of functionality is available. Another available option was the **fetch** library, but **axios** has several advantages, featuring Cross Site Forgery (XSRF) Protection for added security and better error handling which facilitates debugging [27]. Logrocket [28] and Jason Watmore’s Blog [29] were the most informative and helpful to successfully implement Axios requests.

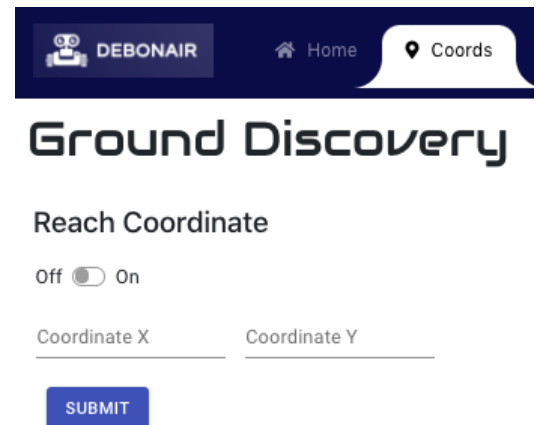


Figure 3.2 - Coordinate input

The second tab, “Controller”, was designed to house the remote-control functionality of the rover. After researching different ways to implement remote controls, it was decided to proceed with implementing buttons that will send characters to the backend. Each character translates to a movement: ‘F’ for forwards, ‘B’ for backwards, ‘R’ for right and ‘L’ for left, with icons from Material UI [30] and Font Awesome [31] used. After initial testing, a problem was encountered: when the buttons were clicked, they were sending a command to the rover which would continuously execute until another button was clicked, making control of the rover quite challenging. To solve this, the *onMouseDown* and *onMouseUp* events [32] were used, where handler functions were created to send a movement character when a direction was clicked, and send a stop character ‘S’ when released. This meant that while a button was held down, the rover would move in the desired direction, and stop once the button was released, making the use of the remote much more intuitive and facilitating control over the rover.

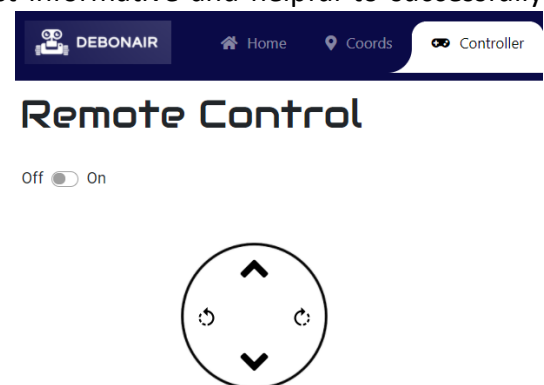


Figure 3.3 - Remote Control

The third, most challenging, feature involved creating a map which would update the rover’s position in real time as well as the obstacles it encounters, which would ideally be displayed on both tabs involving rover movement. The biggest challenge consisted in finding a way to have coordinates available scaled in

reference to the origin of the map. The first approach consisted of creating a grid using CSS grid layout, based on implementations of online chessboards, which designed a grid to their liking and could easily reference the coordinates and place elements as they wished. Since an operational area of 4m² had been designated, and the rover's measurement units are in millimetres, that would mean loading a grid of 2000x2000. This was problematic because the grid would drastically increase web-page load times, making accessing the app almost impossible as the object was too big to be generated. Another approach consisted of creating a canvas and draw a grid using JavaScript's "Canvas" library specialized in drawing graphics. After encountering a multitude of bugs and external libraries to download, this approach was also scrapped. Finally, after further research uncovered the ability to have a smooth movement of images across the screen using only page styling [33], a new approach was considered.

The final implementation of the map used references to the HTML window's predefined coordinates, meaning the map functionality needed to be written in the index.html. The image of a grid [34] and Mars' ground were taken as the map's background [35]. The centre of the grid would represent the origin and serve as reference point for all the rover's movements and positioning of the obstacles. Every 200ms, rover coordinates are requested from the backend, with this speed chosen to match the communication latency requirement outlined in section 1.2. When the Rover detects a new obstacle, a flag sent as part of the rover position package triggers the request of all the coordinates of the obstacles resulting on instant map update of the Rover's findings.

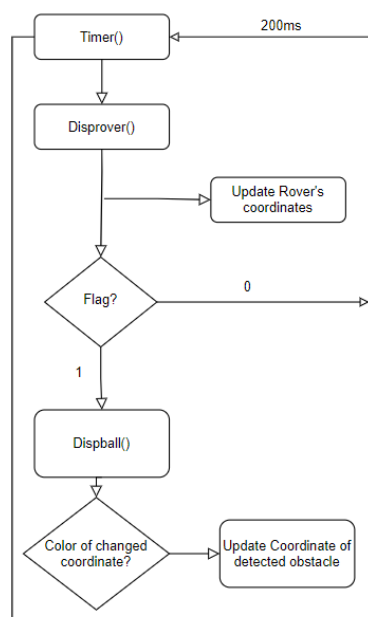


Figure 3.4 - Timer Event Loop

This is achieved through four functions in index.html, which are followed by the insertion of all the necessary images and message displays. **reset()** resets the timer and map to their original state, with all the obstacles sent to their original position outside of the map and the rover placed at the origin. **disprover()** requests the rover positional information from the backend, checking the **newObstacle** flag, extracting the rover's position, and translating it to window's pixel scale. Once it is converted, the x and y axis are correctly portrayed on the map using the "left" and "top" styling of the rover's image. The following **dispball()** function gets called only when the flag from **disprover()** is high, where it requests the position of all the obstacles and verifies if there are any changes compared to its held values, updating coordinates as needed. Similar to **disprover()**, the new coordinates are translated to a windows referenced scale before modifying the styling of the obstacle's images using "left" and "top" styling. Finally, **timer()**, as its' name implies, serves as a timer which calls **disprover()** and sets a timer to call itself every 200ms. Two buttons on the webpage, called "Start" and "Reset", trigger **timer()** and trigger **reset()** functions respectively, allowing the user to decide when they want to start receiving data and when they want to stop and reset their map.

Importing the axios package into HTML required additional research and was solved after importing the necessary scripts [36]. After running tests on average rover speed, it was found that by requesting for the positions every 200ms, the position of the rover would only change by ~33mm, making the movements of the rover appear smooth. This means that real-time data from the rover can now be displayed to the user in seamless and digestible graphical interface, containing all relevant information.

The fourth implemented feature is the display of battery level and **Rover's State of Health** on the **Status** Page. Icons imported from Material UI display battery level at various percentages, keeping the user informed on the battery's status and alerting them when it is low. Similar to the rover position

implementation, when the “Battery Level” button is pressed, the **batterylevel()** function is called which calls **getBattery()** and itself every 60 seconds. This timing was selected after the Power submodule confirmed integer changes in battery percentage would occur every ~72 seconds, meaning requesting every minute should be sufficient to maintain up-to-date information. **getBattery()** requests the backend for the relevant information and updates the displays accordingly. Documentation on how to display when wanted material UI icons was not easily accessible, requiring more lengthy trial and error from previous built knowledge to achieve this.

To enhance the user experience, an additional feature was implemented enabling the user to click on a map position to set destination coordinates. Instead of manually inputting coordinates, with a simple click within the map the rover is informed of its new destination. To achieve this feature, a prewritten **MousePosition()** function was used [37], which retrieves x and y coordinates of the mouse’s position in reference to window’s webpage. The output of this function is rescaled so that the displayed coordinates are in reference to the centre of the map, and the click then triggers an event function which send the rescaled coordinates to the back end.

As part of path finding (see section 3.1.2.1), after a destination is sent to the backend, an array of points forming an optimal path to the destination is returned. To display this path to the user, additional dots were implemented to be displayed whenever Coordinate Mode is used. This way, the map illustrates the waypoints the rover navigates between to reach its target coordinates.

Switch functionality was also added to allow the user to select the rover’s desired mode of operation, which was necessary to instruct the on-board system how to handle different inputs. These switches are implemented the same way the buttons were, with the switch underneath “Reach Coordinate” sending ‘C’ to indicate coordinate mode, the switch underneath “Rover Explore” sending ‘E’ to indicate exploration mode and the switch on the Controller page sending ‘M’ to indicate manual (remote control) mode.

Three different stages of testing were done to verify the functionality and layout of the website. For the layout saving and reloading the website was the best practice to immediately see how the changes affected the layout. To solve the majority of the encountered issues, further research was necessary due to the unfamiliarity with React’s libraries. When testing the communication between the Front-end and Back-end the console was a key debugging tool, allowing data received to be logged to ensure the appropriate information was being received / sent. Iterative testing was done in parallel with the backend, where small changes would be made between runs until the desired communication was achieved. An example of this was during path-finding integration, where the ability of the front end to successfully update obstacle positions, as well as update the position of waypoints based on the received path was confirmed.



Figure 3.5 - Pathfinding integration testing

The last feature developed for the webpage is the Home Page, which was not prioritized as it didn’t include any functionality. The Home Page includes an animation of a rover exploring Mars [38] as well as three sub sections: “Our Mission”, “Rover’s features” and finally “About us”. This page gives more

context of on the team's aim throughout the project, introduces Debonair's features and finally credits each team member on the work they have contributed. The most challenging aspect of this page was the visual design, where significant research needed to be performed to render the page the way it was envisioned. Due to the map's implementation in index.html, the obstacles, rover and displayed text that were hard coded in the file would appear in the middle of the Home Page which was problematic. To solve this issue a path checking function which would display images depending on the route's path. With this finalised, the full, deployment ready web-interface was complete, delivering the full functionality of Debonair to users in an intuitive and visual pleasing way.

3.1.2 Back-End REST API

The structural design requirements outlined previously had an important role in the overall implementation of the server back-end, as a core part of its functionality was to enable smooth communication between the front-end UI and the remainder of the rover system. In order to develop this API, Node.js was selected, as its focus on using non-blocking, event driven I/O was perfect to keep the data transmission lightweight and efficient, particularly when considering the needs of updating data rapidly in real time. Node.js can achieve scalability levels of over 1M concurrent connections, up to 250 times more than more traditional web-serving techniques [39]. Additionally, Node.js offers an expansive package library, which can easily be installed using the Node package manager npm [40]. The first step

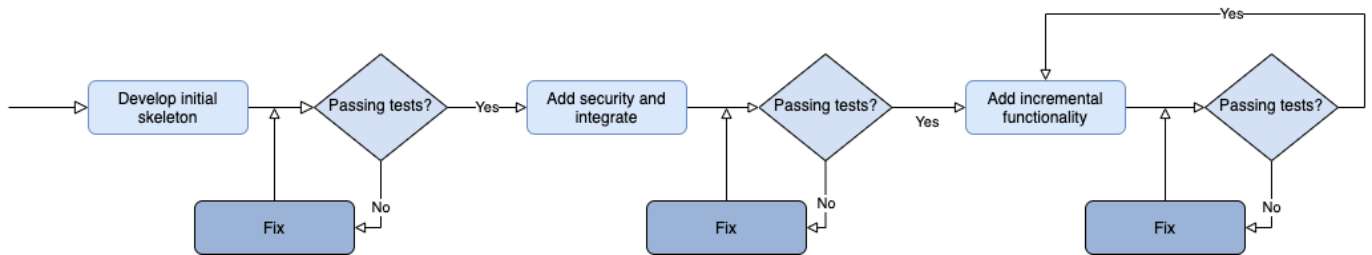


Figure 3.6 - Back-End Implementation Process

to developing the backend was to implement the communication capabilities outlined in the system's structural design: an HTTP request handler and MQTT client.

For the HTTP side, Express [41], a fast, minimalist web framework for Node.js, was used, allowing straightforward handling of any HTTP request to a particular endpoint [42]. This was ideal, as it allowed for a highly scalable [43], top-layer entity to drive asynchronous logic based on the requests received from the client. An HTTP server was then initialised on port 8080 of the server domain

(debonair.duckdns.org) to listen for any requests that would be handled by the express.js app used within the API. Some initial tests between the client and API indicated issues with Cross-Origin Resource Sharing (CORS), where the web browser would send a preflight request before initiating the actual HTTP request to the server [44]. As the API didn't have the functionality to respond to the preflight request, the browser would cancel the actual HTTP request, meaning that data could not be transmitted between the two. To solve this issue Node.js CORS middleware [45] was used [46].

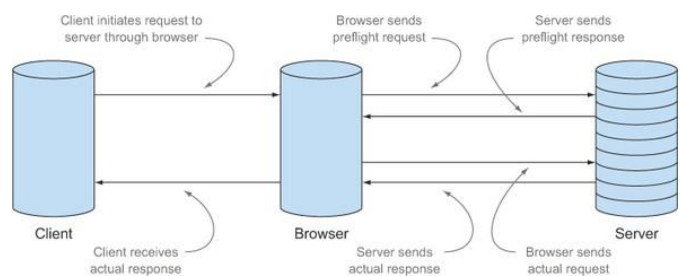


Figure 3.7 - Preflight request [44]

between the two. To solve this issue Node.js CORS middleware [45] was used [46].

To implement the MQTT client, another package, MQTT.js [47], was used. This client subscribes to the `fromESP32/#` topic (where # is a wildcard for any subtopic), and, similar to express, listens for any publishes on this topic, where call-back functions can be defined for each topic to be executed when a particular message is received from a topic. Once basic communication capabilities had been established, a database was required to keep the API stateless, meeting the RESTful design criterion. To implement this, MongoDB, a general purpose, document-based, distributed database [48] was used. To

communicate between the API and the server, the official MongoDB Node.js driver was used. A MongoClient was instantiated within the back-end application, and connection pooling [49] was integrated to re-use the connection to the database for requests, rather than opening and closing new connections each time the database needed to be accessed.

Together, these 3 sections formed the basis for the layered design for the API outlined previously. After some initial stress testing, the rover coordinate position and angle data were moved out of the database back into a temporary container within the REST API. This was because the position of the rover was being requested and updated every 200ms to ensure the rover position would smoothly update on the UI, which was putting a high load on the database client. The free-tier database has an upper limit of 100 operations per second [50], meaning that if rover positional data were exclusively passed through the database, the scalability of the back-end system would be extremely limited, as each new node would add a minimum of 5 operations per second. By storing the rover data in a temporary container, the same free-tier database can be used without having to compromise the seamlessness of the frontend user interface. To ensure statelessness was maintained, rover positions were still sent to the database, but this could potentially cause the same capping issue if more rovers were to be added to the system, in which case a more powerful database could be used.

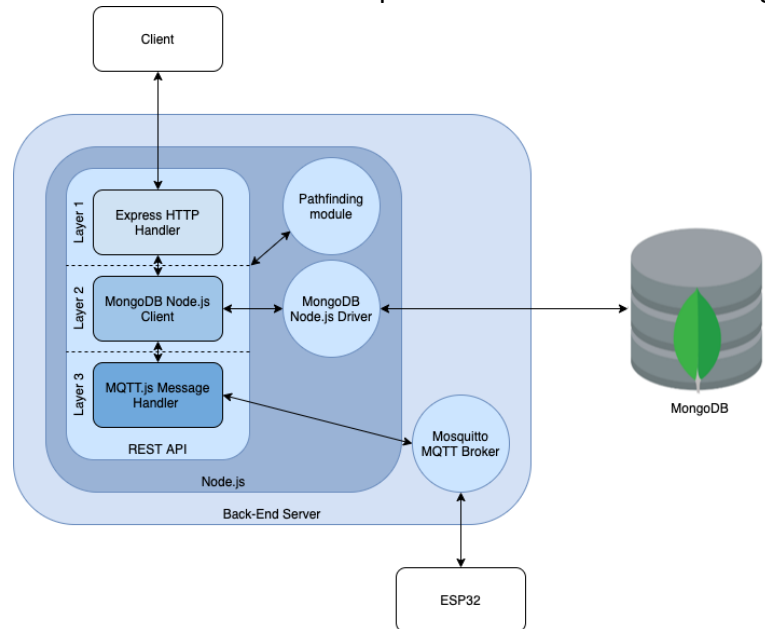


Figure 3.8 - Back-End in-depth Structural Diagram

More dependency trees were built until all requirements outlined in section 1.2 were met. Thanks to the asynchronous, modular structure of the REST API, each feature could be tested and added independently, until the full functionality could be implemented. In an effort to keep the latency between the client and API low, the size of responses was kept as light as possible, as larger payloads cause increased latency [51], particularly for frequently requested data such as rover position. Rather than including obstacle data in this payload, a flag was sent indicating whether a new obstacle had been identified, allowing the client to perform a separate request for the obstacle data in a larger payload.

More dependency trees were built until all requirements outlined in section 1.2 were met. Thanks to the asynchronous, modular structure of the REST API, each feature could be tested and added independently, until the full functionality could be implemented. In an effort to keep the latency between the client and API low, the size of responses was kept as light as possible, as larger payloads cause increased latency [51], particularly for frequently requested data such as rover position. Rather than including obstacle data in this payload, a flag was sent indicating whether a new obstacle had been identified, allowing the client to perform a separate request for the obstacle data in a larger payload.

As the scope of the API expanded, it became increasingly important to automate server processes, as well as have some sort of error handler within the API to prevent critical errors from crashing the server and rather exit gracefully. This is important as if the application exits suddenly, there may be unfinished requests, and these will remain unhandled, which can lead to unknown behaviour [52]. The script **run_server.sh** prepares the entire AWS instance for testing, updating the code to the most recent version available and launching all the required processes in Linux screens [53]. This ensures that the processes running on the instance are independent from the client SSH-ed into the server, meaning that if the connection fails these processes will continue to run. The **halt_server.sh** halts these processes, using the *SIGTERM* signal to invoke the critical error handler of

```
1 // Handles shutting down application on critical errors
2 process.on('SIGTERM', () => {
3   httpsServer.close(() => {
4     console.log('HTTPS server terminated');
5   });
6   client.end(() => {
7     console.log('MQTT client disconnected');
8   });
9   db_client.close(() => {
10    console.log("Disconnected from MongoDB");
11  });
12 })
```

Figure 3.9 - API Shutdown Handler

The **halt_server.sh** halts these processes, using the *SIGTERM* signal to invoke the critical error handler of

the API and ensure all connections are cleaned up before halting. Within the API, the error handler is also called in extreme cases, such as if the application is unable subscribe to *fromESP32/#* or connect to the database.

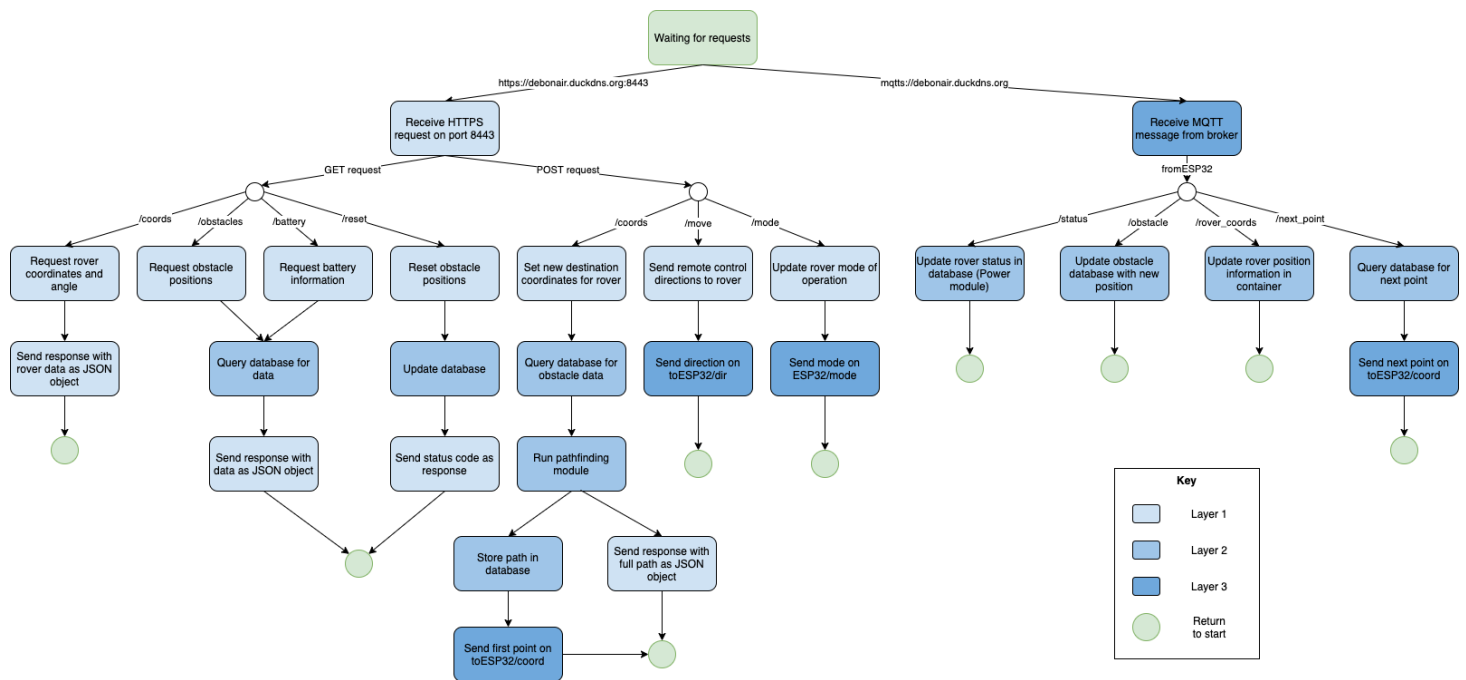


Figure 3.10 - Back-End Functional Diagram

3.1.2.1 Pathfinding

In order to navigate through an environment containing obstacles, the backend should be able to generate a path for the rover to reach a destination given by the user so the rover is able to correctly navigate through its environment. However, JavaScript is single-threaded, meaning that any complex, CPU-bound tasks, involving complex computations, are extremely undesirable as any other requests the program would receive would be blocked for the duration of this task [54]. It is therefore of critical importance that the pathfinding process should be as efficient as possible to keep the time requests that are blocked small. To achieve this, the pathfinding function was developed as a Node.js addon [55], written in C++, as it performs around 10 times faster than JavaScript [56]. This addon could then be loaded as a Node.js module into main codebase, where it would run significantly faster than if it had been implemented directly in JS. Node.js modules can even spawn auxiliary threads separate to the main event loop thread, which is what allows node to execute asynchronous I/O operations so effectively [57].

Ideally, the pathfinding algorithm should provide the rover with a close to optimal path without too large of a time-complexity. A heuristic approach was imposed such that finding this path would have a greedy-choice property, such that in order to reach a path, the ‘most optimal’ route could be found by avoiding the first obstacle in the rover’s path and recursing from the rover’s new position until the destination could be reached. This reduces the time-complexity from exponential to polynomial time, providing an enormous asymptotic speedup to the algorithm.

The algorithm is composed of 3 sub-functions, which are used by the primary `genPath` function. The main one `genIntermed` generates a waypoint for the rover to travel to if an obstacle is in its path to the destination and if not returns the destination coordinates. It should be noted for clarification, that the path of the rover is always a straight line to the destination coordinates unless an obstacle lies along this path or within a certain distance of a point along this path. The first thing that is addressed in `genIntermed` is the range of x coordinate values and y coordinate values between the rover and the destination. This is done by using two pairs, which store the maximum and minimum x and y coordinates when looking at the destination coordinates and the rover’s current position. A distinction needs to be

made for when the rover is travelling into quadrants with negative x and y coordinates and in these cases the most positive value (closest to 0) is taken as the maximum value.

After the x and y ranges have been stored, a function `inTheWay` is called to determine all the obstacles in the potential path of the rover. First, the angle of the path from the x-axis is calculated using the inverse tan of the difference between the y range divided by the difference of the x range values. Then a for loop runs through the coordinates of every obstacle, which are first checked for if they are within the x-range values with added clearance either side. If this is the case, using the x-coordinate of the obstacle, the rover coordinates and the angle of the path, the y-coordinate along the path at the obstacle's x-coordinate is calculated. The rover coordinates are known due to them being one of the x and y range values. It has also been noted that paths in all four quadrants of a grid centred at (0,0) require slightly different implementations and these have been used.

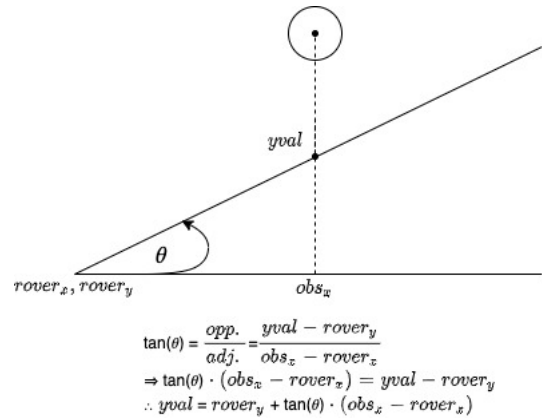


Figure 3.11 - Obstacle Detection (1)

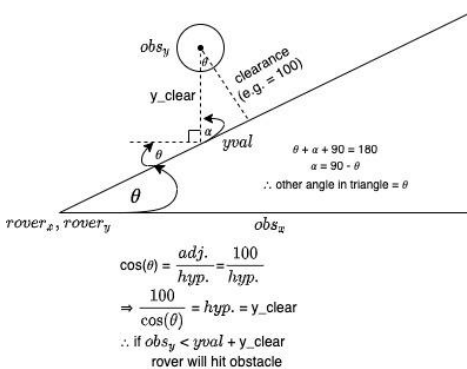


Figure 3.12 - Obstacle Detection (2)

Next using trigonometry, a clearance range is calculated for the y-coordinate of the obstacle. This uses angle of the path, the y-coordinate of the obstacle, the y-coordinate along the path from the previous step and a clearance value (set to 100mm by default) at the shortest distance along the path from the obstacle (when a line drawn from the centre of the obstacle intersects the path at 90°). This is able to predict if the rover will hit the obstacle along the path and clearance is checked above and below the obstacle to take into account if the obstacle is to the left or right of the path. If the obstacle is seen to be in the way it is stored in a vector, which is then run through another loop to calculate which object is actually

the closest to the rover using Pythagoras' theorem on the difference between the rover and obstacle coordinates. The closest obstacle has now been identified and now the next sub-function is called.

The `avoid` function calculates the coordinates of the next waypoint for the rover to avoid the obstacle. Using the rover's coordinates, the obstacle's y-coordinate and trigonometry, the x-coordinate along the rover's path at the y-coordinate of the obstacle is calculated. This x-coordinate along the path (x_{pos}) is then compared with the x-coordinate of the obstacle to see if it is on the left or right of the path. Depending on the quadrant and if the obstacle is on the left or right of the path, the avoid function will move the waypoint away from the obstacle by addition/subtraction to x_{pos} and the obstacle's y-coordinate. Originally the x coordinate for the waypoint was set as this calculated x coordinate (x_{pos}) and the y coordinate adjusted, but after testing, in the case of large path angles (θ getting close to 90°) the rover would not clear the obstacle. Instead, the waypoint x-coordinate was adjusted to be x_{pos} plus a certain clearance distance that was scaled depending

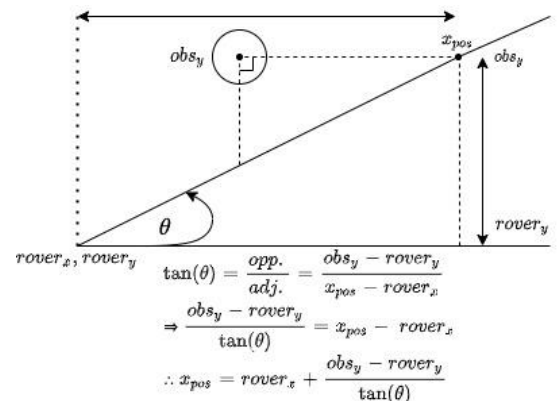


Figure 3.13 - Rover x-position calculation

on the size of θ . When $\theta = 90^\circ$ the x-coordinate of the waypoint would be x_{pos} plus the full clearance distance and when equal to 0° it would be just x_{pos} .

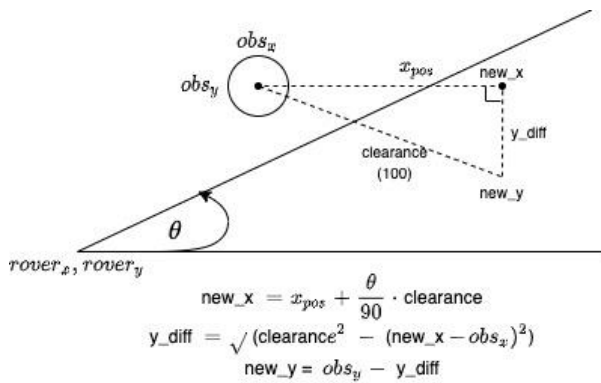


Figure 3.14 - Waypoint y-position calculation

The y-coordinate of the waypoint is calculated using the diagram on the left. Using the x-coordinate along the path calculated before with the added scaled clearance, the difference between this and the x-coordinate of the obstacle is used with a certain clearance distance as two sides of a right-angle triangle. The difference in y-coordinate of the obstacle with this clearance is found using Pythagoras' theorem and this is subtracted from the y-coordinate of the obstacle to find the y-coordinate below the path. This is then stored as the y coordinate of the waypoint. A redundancy statement was added for when the difference between the new x-coordinate and

the x coordinate of the obstacle was greater than the clearance distance, the hypotenuse of right-angle triangle, as this would return an error when using Pythagoras' theorem. This tended to happen at smaller path angles and was adjusted after testing so that in this case the y-coordinate of the waypoint was stored as the clearance distance added/subtracted from the y-coordinate of the obstacle. This along with the added clearance on top of the x-coordinate on the path ensures that the waypoint coordinates are always at least the clearance distance away from the obstacle and in majority of cases more than that. This waypoint is then returned at the end of the `genIntermed` function and stored in a vector of pairs (coordinates) for the rover to take as its path in `genPath`. `genIntermed` is then called with this new waypoint, and the function runs again to see if any other obstacles need to be avoided from this new position to the destination, until the destination is reached.

Calculations for the `avoid` function have been adjusted slightly for each quadrant as x range and y range values are different for each of them. These different calculations are separated by looking at the maximum values for x and y ranges and if they are greater than 0, which for example would be the case for both x and y values for travel in the top right quadrant. A diagram is shown on the right displaying the conditions needed for each quadrant. This also had to be considered when calculating the y coordinate for the `inTheWay` function. Separate cases were also added for when the destination was on one of the x or y axis after testing. This was in both `inTheWay` and `avoid` as x/y range would be 0 at the start leading to problems when $\tan(\theta)$ or $\cos(\theta)$ was called.

Testing for all 4 quadrants was conducted before and after the adjustments mentioned above and an example for each quadrant is shown on a grid for reference with the path drawn (labelled points are waypoints, unlabelled points are obstacles) along with the actual output of the code and a table showing the distance between each waypoint and the obstacle they are avoiding (this is using a minimum clearance of 100mm).

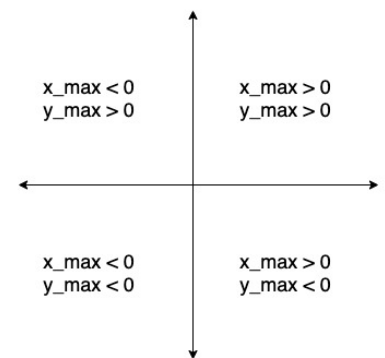


Figure 3.15 - Quadrant adjustment

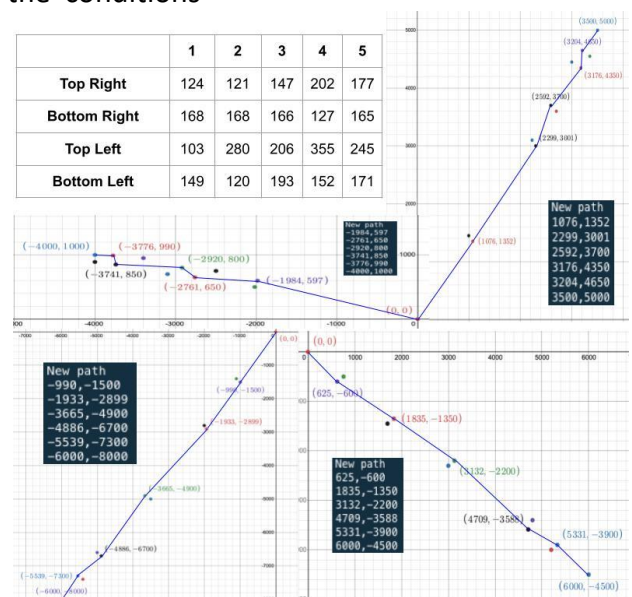


Figure 3.16 - Pathfinding testing

To make `genPath` into a Node.js addon, Node-API (N-API), an Application Binary Interface (ABI) which is stable across versions of Node.js [58], was added. To set up a new addon, a basic boilerplate setup was created [59]. To interface between the C++ code and node, a matching N-API wrapped function needs to be created and added to the node exports object. Due to difficulties in handling I/O using arrays, JSON strings were instead chosen to be used to pass arguments to and receive outputs from the pathfinding addon. `genPath` needed to be extended to accommodate these changes, parsing the input into the objects that were usable by its subfunctions, and converting the output path vector into a JSON string that could be parsed by the backend.

Once the module was ready, node-gyp, a Node.js native addon build tool [60], was used to compile C++ code into a node module that could be imported into Node.js. To

```
1 #include <napi.h>
2
3 namespace pathfinding {
4     // Generates path as a JSON string {"points":[{"x":point1, "y":point1}, {"x":point2, "y":point2}, ect...]}
5     std::string genPath(std::string pos, std::string dest, std::string obs);
6     // Wrapper function to pass arguments / return value between node.js and genPath(), designates entry point
7     Napi::String GenPathWrapped(const Napi::CallbackInfo &info);
8
9     // Used to set export key with wrapped function
10    Napi::Object Init(Napi::Env env, Napi::Object exports);
11 }
```

Figure 3.17 - Pathfinding module header

keep the API modular, a separate `pathfinding.js` file was created to load the module built by node-gyp and act as an intermediary between the API and the module itself. This allowed functional testing to be performed independently of the API, to

```
21 position = {std::stoi(posX), std::stoi(posY)}; // generates pair position
22 }
23 catch(std::exception &err) {
24     printf("Pathfinding Error: Invalid position input, could not parse\n");
25     return "";
26 }
```

Figure 3.18 - Input Error Handling Example

solve any issues without affecting the rest of the back-end functionality. After running some tests, an exit condition was added to `genPath` such that past a certain depth of recursion, the algorithm would give up and return the current position, meaning that the pathfinding module would never be 'stuck' on finding an impossible path, clogging up server resources. The module was also updated to enable C++ exception handling [61], so if there are errors with parsing the inputs to the pathfinding algorithm, the module exits gracefully, rather than crashing the entire server.

Once satisfactory testing had been performed, the module was then imported into the backend API for performance testing on the AWS instance. This was done by calling `genPath` on a given input 10,000 and measuring the average time taken for a function call to resolve. Using the Q1 testcase from above, the pathfinding module is able to generate a path avoiding all 5 obstacles in 70 μ s, which is an extremely fast execution time that will not cause any freezing up of the server.

```
1 // ----- Pathfinding module -----
2 const Pathfinding = require('./build/Release/pathfinding.node');
3 console.log('Pathfinding module loaded:', Pathfinding);
4
5 module.exports = Pathfinding;
6
7 // ----- REST API -----
8 const Pathfinder = require('./pathfinding/pathfinding');
9 // Generate input strings
10 let pos = `${rover.x},${rover.y}`;
11 let dest = `${parseInt(req.body.coordinateX, 10)},${parseInt(req.body.coordinateY, 10)}`;
12 let obstacles_string = ""
13 for (let i in obstacles) {
14     if (obstacles[i].x) obstacles_string += `${obstacles[i].x},${obstacles[i].y}`;
15 }
16
17 path_res = Pathfinder.genPath(pos, dest, obstacles_string); // Call genPath on inputs
18 if (path_res) { // Checks for empty result
19     path = JSON.parse(path_res);
20 }
```

Figure 3.19 – Pathfinding module usage

3.1.2.2 Performance

Evaluating the performance of communication between the REST API and the other nodes would give a key insight into where the limiting factors in the system are. Wireless communication is the slowest part of the system, and so knowing the speeds that are achievable would provide the

```
1 setInterval(() => {
2     start = Date.now();
3     axios.get('https://debonair.duckdns.org:8443/')
4         .then(response=>{
5             end = Date.now();
6             // {Data processing ...}
7             }, 100)} // Performed every 100ms
```

Figure 3.20 - Dummy client code snippet

most accurate picture of the system's overall performance capabilities. All testbenches were performed on the final, encrypted communications used by the production system.

To test latency between the front-end client and API, a dummy client was written to request messages from the backend every 100ms, recording the time the request was sent and received. This time would be stored in an array and exported in a CSV format after 500 iterations to take an average of round-trip times. This resulted in an average round-trip time of 20.76ms. The spike at around 100 samples is likely due to perturbations in the network of the client, but without these outliers, communication speeds are very reliable, with a standard deviation of 6.00ms.

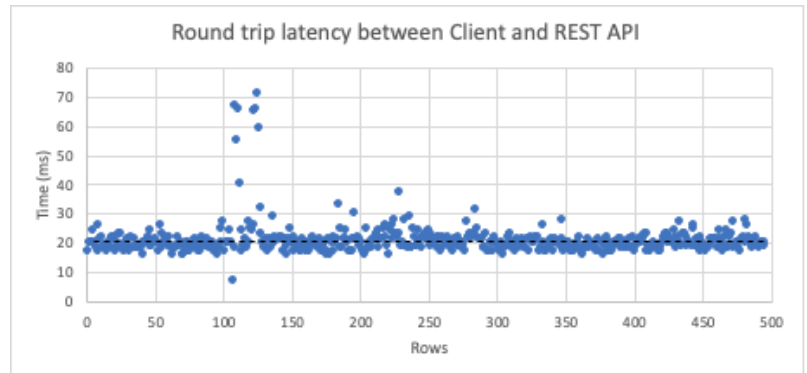


Figure 3.21 - Client <-> API round trip latency scatter plot

For latency between the ESP32 and the API, a call-back function was written on the ESP32 to publish a message to fromESP32/test when it received a message on toESP32/test. On the API side, a message would be published to this topic every 500ms, with the send time recorded, and another call-back, this time subscribed to messages on fromESP32/test, would be waiting to receive the response from the ESP32 and log the end time. Similarly, to the latency testing between Client and API, 500 iterations were

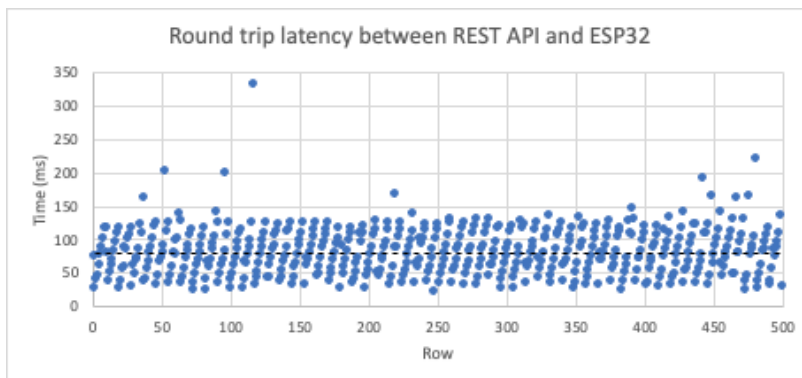


Figure 3.22 - ESP32 <-> API round trip latency scatter plot

run, and results were once again exported in a CSV format. The average for this latency test revealed a round-trip time of 81.29ms, with a much larger standard deviation of 34.90ms. This was surprising as MQTT is a lighter weight protocol, but reasons for this slower latency could be linked to the limitations of the Mosquitto broker hosted on the AWS instance, or to the capabilities of the of the Wi-Fi module on the ESP32.

However, comparing the round-trip times directly is slightly disingenuous, as unlike HTTP, MQTT communication doesn't fit the same request/response model, and instead sends one-way messages between clients via the broker. Therefore, in the context of this system, using an estimate of ~40ms as the average latency of communications between the ESP32 and REST API is more accurate.

This means that an approximate 60ms delay can be expected between user input and rover response, with all other communication, such as updating the rover position, being faster as the MQTT and HTTP communications are handled separately. This is a perfectly acceptable level of input delay, comparable to commercial TV input delays [62] and is likely to be completely unnoticeable by the user. Since all requests to the database are very simple, it's expected these queries resolve in <1ms [63], and so these were not considered to add a delay, along with physical communication. It's important to note that these benchmarks are highly dependent on the network used for both the client and ESP32, and better / worse latency could arise from a change in network used. This was confirmed in more extensive testing, when using cellular data instead of a network on the ESP32 cause the average round trip time to increase to ~220ms.

3.1.3 Security

Security was an important requirement of our overall system, so it was necessary to ensure that the rover could only be controlled via the web-app UI, and that any data being transmitted wirelessly was fully encrypted. These requirements were implemented on top of existing, lightweight, tested functionality, ensuring minimal refactoring would need to be done once security was successfully integrated.

Secure Sockets Layer (SSL) / Transport layer Security (TLS) is a session layer protocol [64] which both MQTT and HTTP can run over to encrypt their contents [65]. This is done through a complex handshake process [66], where the client requests the server for a secure connection, the pair select a cipher suite that will be used for the remainder of the connection, and using a certificate establish authenticity and a session key that both the server and client will use for the remainder of the connection. These SSL certificates are at the centre of the SSL/TLS protocol, providing the client with the public cryptographic key necessary to initiate secure connections, as well as authenticating that the key is associated with the organization offering it to the client.

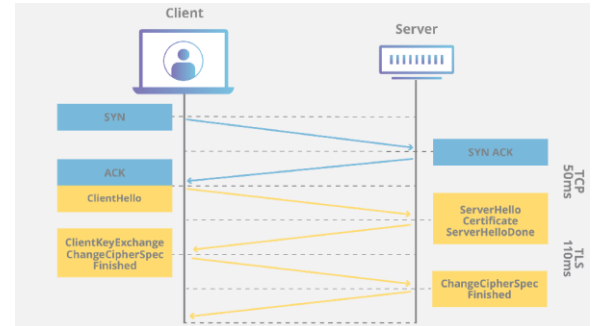


Figure 3.23 - TLS Handshake [66]

Certificates are issued by Certificate Authorities (CAs) who serve as the equivalent of a passport office when it comes to confirming identities [65]. Both the MQTT broker and NginX web server require these certificates, but their requirements are slightly different. Since the web-app needs to be accessible via a browser, the SSL certificate must be obtained from a source that is trusted by the browser, which depends on the browser [67] [68]. A certificate must therefore be obtained from a generally trusted CA to ensure compatibility with the different browsers and operating platforms that the site could be accessed from. LetsEncrypt is a non-profit CA created by the Internet Security Research Group (ISRG) which issues certificates for free [69], and is trusted by a large range of OS and browsers [70]. Using the domain debonair.duckdns.org, the command-line Let's Encrypt client certbot was used to obtain a trusted certificate, and the NginX web server configuration file was updated to use this certificate for connection to port 443 (reserved for HTTPS) [71].

```

1  const HTTPS_port = 8443;
2
3  // Setup certificates for encrypted communication with front-end (HTTPS)
4  const cert = fs.readFileSync('/etc/letsencrypt/live/debonair.duckdns.org/fullchain.pem', 'utf8');
5  const key = fs.readFileSync('/etc/letsencrypt/live/debonair.duckdns.org/privkey.pem', 'utf8');
6  const SSL_options = {
7    key: key,
8    cert: cert
9  };
10
11  const httpsServer = https.createServer(SSL_options, app);
12
13  httpsServer.listen(HTTPS_port, () => {
14    console.log(`Listening at URL https://debonair.duckdns.org:${HTTPS_port}`);
15  })

```

Figure 3.24 - REST API HTTPS server

Whilst this meant that any requests to the web-app were encrypted, requests to the REST API were still using HTTP, and these requests were blocked by the browser as the connection to the app was using HTTPS. The REST API also needed to be updated so that the express handler would be able to handle HTTPS requests, which was done using the https.js package [72]. Since the REST API was hosted on the same server as the web-app, the Let's Encrypt certificate could be re-used for the API. This HTTPS server was instantiated on port 8443, and once it was functional, the HTTP server on port 8080 was deprecated, meaning the API only supported HTTPS requests on the new port. This meant that even if a client connected to the web-app using HTTP, they would send HTTPS requests to the API, keeping the communication chain secure. One drawback of Let's Encrypt certificates is that they expire after 90 days, meaning that a new certificate would need to be requested to keep encryption capabilities. To remedy this, a **cron** [73] job was added to the AWS instance, renewing and reloading the certificate using certbot every week.

Implementing SSL/TLS for MQTT communication was more problematic, as for HTTPS, the client's browser performs the certificate request behind the scenes, whilst for the ESP32 client, everything needed to be handled manually. In order to use MQTT over SSL/TLS, the PubSubClient that was being used could be instantiated over WifiClientSecure, a class which implements support for encrypted communication for the ESP32 [74]. This works by providing the secure client with CA certificate which it can then use to authenticate the server and negotiated the encrypted connection, meaning the certificate must be hard coded into the ESP32 so that it has access to it on start-up. Using Let's Encrypt certificates as was done previously is therefore extremely limiting, as this certificate will need to be manually updated on the ESP32 each time it is renewed. A better option would be to create an independent, private certificate authority to be used by MQTT communications, as there is no requirement for the CA to be 'trusted' since all communication using MQTT is managed internally.

This new CA could then be used to issue a certificate to the broker on the server, where the certificate expiration date could be set to be arbitrarily long, and then program that certificate into the ESP32 so that it can be used by WifiClientSecure [75]. OpenSSL, a robust, commercial-grade toolkit for SSL/TLS protocols [76], was used on AWS to create the cryptographic pairs necessary to form a CA [77]. A custom CA key-pair was then generated based on the IP address of the server the ESP32 was connecting to using a modified script [78], which would be used by all MQTT clients, as well as the broker itself (see Appendix 5.1). After making these updates, communication was tested using the local mosquitto CLI to ensure the ESP32 client was able to send and receive messages. Due to its higher-level nature, updating the REST API to use this new encryption was simpler, the connect method simply needed to be updated to specify MQTTS, and a client option needed to be updated to accept the self-signed certificate that had been created [79].

```

9  # mqtt (only allow localhost)
10 listener 1883 localhost
11 # mqtt over TLS/SSL
12 listener 8883
13 certfile /etc/mosquitto/certs/3.8.182.14.crt
14 cafile /etc/mosquitto/certs/ca.crt
15 keyfile /etc/mosquitto/certs/3.8.182.14.key
16 protocol mqtt

```

Figure 3.26 - Updated MQTT broker configuration

```

75 // Parameters for the mqtt connection
76 const char* mqtt_server = "3.8.182.14";
77 const int mqtt_port = 8883;
78
79 WifiClientSecure espClient; // Secure client to connect over SSL/TLS
80 PubSubClient client(espClient); // MQTT client setup over secure client
81
82 void setup {
83   // {...}
84   // Set up wireless comms
85   setup_wifi();
86   espClient.setCACert(ca_cert); // Set SSL/TLS certificate
87   client.setServer(mqtt_server, mqtt_port); // Set MQTT server
88   // {...}
89 }

```

Figure 3.25 - Updated ESP32 client

```

163 // ----- MQTT client -----
164 const clientOptions = {
165   clientId:"mqttjs01",
166   username:"webapp",
167   password:"=ZCJ=4uzfZZ#36f",
168   rejectUnauthorized : false // Required for using self-signed certificate
169 }
170 const client = mqtt.connect("mqtt://debonair.duckdns.org", clientOptions);

```

Figure 3.27 - Updated REST API client

As well as encrypting MQTT over SSL/TLS, additional authentication was added to the broker to limit which clients had permissions to post on certain topics, The rover should only be able to be controlled via the REST API, and similarly only the REST API should be able to receive information from the user. To enforce this, 2 user id/password pairs were generated and encrypted, and the broker configuration files were updated to allow only the ESP32 user to publish to the *fromESP32/#* topic and subscribe to the *toESP32/#* topic, and vice-versa for the REST API user. This prevents any external party connecting to the MQTT broker and subscribing to / publish on these restricted topics without having access to these user details.

3.2 Control (ESP32)

Control serves an almost entirely structural role as the hub for communication between the different on-board subsystems, and as such had very little stand-alone functionality to keep the code being ran by the ESP32 as lightweight as possible. Once the means of communication between the sub-modules had been established (see section 2.3), all that needed to be determined were the various message encoding schemes that would be used to transfer different messages.

3.2.1 UART

When communicating over UART, the Serial library functions [80] are available to help parse messages. To take advantage of this, a standardized packet structure was developed, where a start of frame character would indicate to the board receiving the data that valid information was being sent, followed by a series of comma-separated values (CSV), and finally an end of frame character to indicate the packet was complete. This meant that once a valid packet was recognized, the `readStringUntil()` function [81] could be used to extract each value and convert it into an integer. For more simple communication, single chars were also sent to reduce unnecessary overhead.

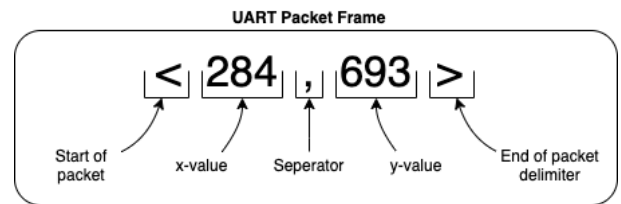


Figure 3.28 - Destination xy-coordinates UART packet

Sender	Receiver	Data Type	Method Used
Control	Drive	Destination xy-coordinates	Packet (x, y)
Control	Drive	Mode	Single char
Control	Drive	Remote control directions	Single char
Drive	Control	Rover positional data	Packet (x, y, angle)
Power	Control	Battery data	Packet (energy %, state of health %)

An important thing to note was that despite the rover having 3 different operational modes, Drive itself only recognized 2 different modes, coordinate (C) and remote control (M). This was because for both remote control and exploration mode, Drive would simply be following directions, but taking instructions from the web-app in the former and the Vision submodule in the latter. Control would keep track of the 'true' mode, allowing it to determine whether to forward directions from the web-app or the Vision subsystem. Unfortunately, communication with the power submodule was unable to be tested due to COVID limitations. To avoid saturating the UART buffer, information was only sent when necessary, i.e. to indicate a change in direction for remote control mode.

3.2.2 SPI

SPI communication operates on a lower level than UART, where a specifically sized payload in bits (8, 16, 32, etc.) needs to be designated to transfer all the necessary information. In exploration mode, Vision needs to direct Drive or send the position of an obstacle to Command, so a single bit was allocated to indicate whether the payload being received was an instruction or data. For drive instructions, only 4 different directions are possible, which means only 2 further bits are needed. However, for data, both the colour of the obstacle as well as its' distance from the rover need to be sent. There are 5 different coloured obstacles, meaning 3 bits needed to be allocated to differentiate between them. At the time of design, it was believed that Drive could provide a distance in the range of [1cm, 30cm], meaning 5 bits needed to be allocated (this was later tightened to a 5cm range between [26,30] for increased accuracy). This meant a total of 5 bits were needed for instructions, and 9 bits for data, and therefore a 16-bit payload was allocated.

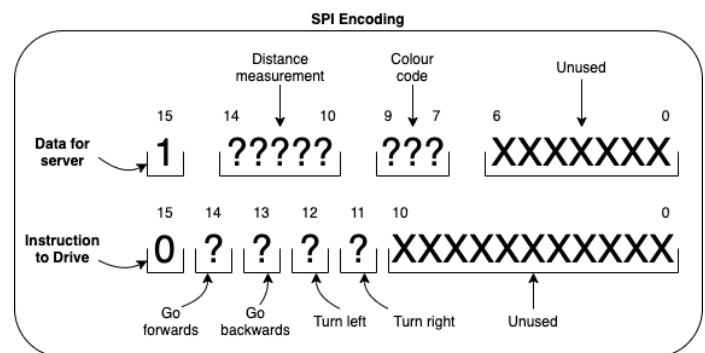


Figure 3.29 - SPI Encoding from Vision

A smaller 8-bit payload could've been designed, but as SPI was by far the fastest type of communication in the entire system, having these wasted bits had essentially no performance costs whilst almost

guaranteeing any developments leading to requiring more data from Vision could be accommodated without completely reworking the payload structure. It was concluded that the added flexibility the 16-bit payload provided was worth it and the makeup of the packet was finalised (see Figure 3.30). Once the distance has been measured accurately (see Section 3.3.6), a flag is sent back to Vision to indicate the obstacle has successfully been recorded and it can move on to the next one. To convert the data from Vision into the format that will be used by the web-app, the coordinates of the obstacle need to be calculated. This was done with simple trigonometry, using the rover's current position and angle, which is regularly updated by Drive, and the ball's distance from the rover. The information is then stored, and a flag is set indicating a new obstacle position has been recorded and is ready to be encoded and sent to command.

3.2.3 MQTT

Communication between Command and Control was designed in a way to facilitate the transfer of data depending on the direction information is being sent. This means that for any instructions that were being sent from command to the rover, the UART encoding was used, so that Control simply needed to forward the packets or single characters when appropriate. When sending data up to the server, the ESP32 would encode the data as JSON strings [82], which could be easily parsed by the REST API using `JSON.parse()` [83] to construct an object that made data processing easier. All data being sent via MQTT was split into different subtopics to streamline communication (see Figure 3.31 - Back-End Functional Diagram). The ESP32 would publish data about the rover position every 200ms to match the refresh rate of the map, data about a new obstacle once it had been detected and power information every 60s.

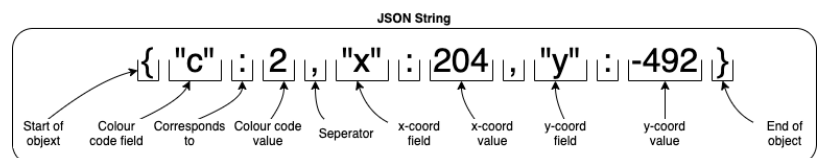


Figure 3.30 - JSON string for obstacle data

3.3 Vision

The purpose of the Vision subsystem is to use the rover's camera to identify different obstacles/objects of interest and to send commands or information to the other subsystems. To achieve this, there are 3 fundamental tasks that the Vision subsystem is required to achieve. The first is the ability to identify targets of interest/obstacles and with that, comes the ability to differentiate between the objects using an object detection algorithm. The second is to be able to calculate an approximate distance from the rover to the said object so it can be mapped by the Command subsystem. The third is to set up an interface where the Vision subsystem can communicate with the Control subsystem (ESP32) so the data can be used by the rover.

The process of object detection is only needed for one out of three of the rover operating modes outlined in the system requirements. For exploration mode the rover will need to detect the objects one at a time and report back the colour and distance of the nearest one. Using this, the Control module will calculate the coordinates of the ball and send it to the server, where the web-app will update the map by the adding object on the coordinates specified. This information will be used when the rover is in coordinate mode to avoid these obstacles when travelling to a user defined destination, meaning that for the rover to function autonomously it is essential to identify these obstacles correctly and judge their distance accurately. To accomplish the first task, data will have to be passed through an image processor to modify the video data to implement an appropriate filtering/object detection algorithm. The `EEE_IMGPROC.v` is a module that is used for this exact purpose.

3.3.1 Image processor

The **EEE_IMGPROC.v** module is an image processor that allows data modification to the video data. The module has two pipeline stages implemented as instances of a streaming register submodule “STREAM_REG” as seen on the diagram:

The implementation allows changes to be made as the data passes between the two streaming registers which allows one clock cycle of logic latency (10ns) which very is important for an autonomous system because data needs to be transferred in real time needing the lowest latency possible.

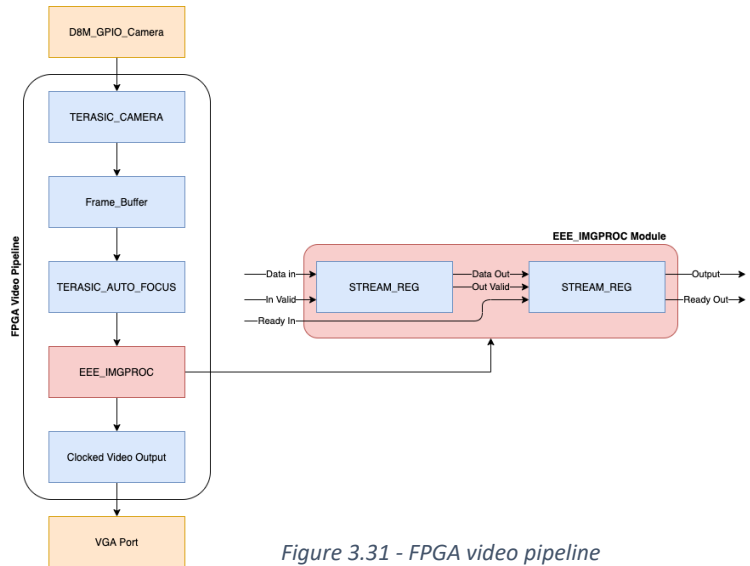


Figure 3.31 - FPGA video pipeline

Pixel data enters the **EEE_IMGPROC.v** block one at a time starting from the bottom right of the screen going horizontally towards the top left. This is because the Avalon-ST Video Packet is currently being transmitted as an “Avalon-ST RGB Video Packet”. The order of which the data for RGB video packets is transmitted is in the order of Blue, Green, Red [84].

The pixels vary in RGB values with each colour being 8 bits big thus implementation of classic filters such as a median filter/gaussian filter would be a challenge since all pixels are currently accessed sequentially [84]. Not to mention most filters require a pixel radius which means pixels above and below are needed, thus entire lines of pixels will have to be stored to access them [85]. To achieve this, pixels in parallel will have to be accessed and stored on a big register or a smart implementation sequentially could also be implemented but would be difficult.

Avalon-ST RGB Video Packet

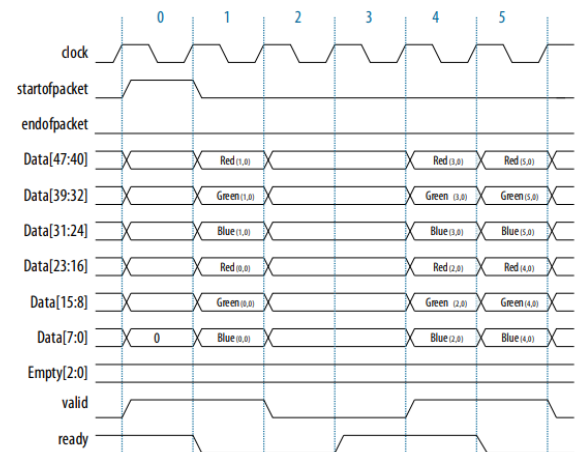


Figure 3.32 - Avalon RGB Video Packet [84]

3.3.2 HSV Converter

The first change made to the object detection algorithm is to implement a RGB to HSV converter. This is crucial to the object detection algorithm because HSV is more reasonable to work with than RGB - “HSV separates luma, or the image intensity, from chroma or the colour information” [86]. This means Hue is the only attribute that controls colour value, and a combination of saturation and value controls the intensity. This is an integral part of the image processor as it allows easier differentiation between objects. The OpenCV HSV converter [87] was used since its saturation and value conversion are between 0 to 255 instead of 0 to 1 which uses integers instead of floats. This speeds up computation time significantly because it reduces floating point calculations to integer calculations. However, the conversion had to be implemented more carefully since division with integers will always round down not to mention division with negative numbers does not work in the same way in Verilog as it does for regular arithmetic which could cause some unexpected numbers so the equation had to be rearranged to compensate for this.

$$e.g. \frac{120 + \frac{60(B - R)}{V - \min(R, G, B)}}{2} \rightarrow \left(\frac{120(V - \min(R, G, B)) + 60(B - R)}{V - \min(R, G, B)} \right) \gg 1$$

Shifts were also used instead of division whenever possible because the FPGA does not have an embedded divider block which could use up a lot of logic cells and slow it down [88]. After a good HSV range was found it was noticed that the camera seems to have noise that is always seen no matter how specific the HSV range is. This causes the bounding boxes for distance calculation to be inaccurate which is a big problem for the rover.

3.3.3 Filter

The second change made to the object detection algorithm was to implement filters that will eliminate noise. The first noise chosen to eliminate is salt and pepper noise, which are random impulse disturbances of pixels in the video signal [89]. A blurring filter is used as a low pass filter to eliminate the salt and pepper noise [90], because it averages out a pixel to the colour of its area [90]. Although the image quality is worse due to the blur, it is considered a very good trade-off for reduction in noise as the image output is not going to be seen by the user because only information about colour and distance to obstacles are sent to the web app.

The next noise to eliminate was Gaussian noise, noise with a probability density function equal to a normal distribution which is added to the video signal [91]. For the second filter, multiple filters were considered to eliminate Gaussian noise. Increasing the pixel radius of the mean filter/adding another mean filter was initially implemented which was then taken away since the blurring and mean filter combined would cause the bounding boxes of the objects to be inaccurate due to the edges of the objects being smudged with the background.

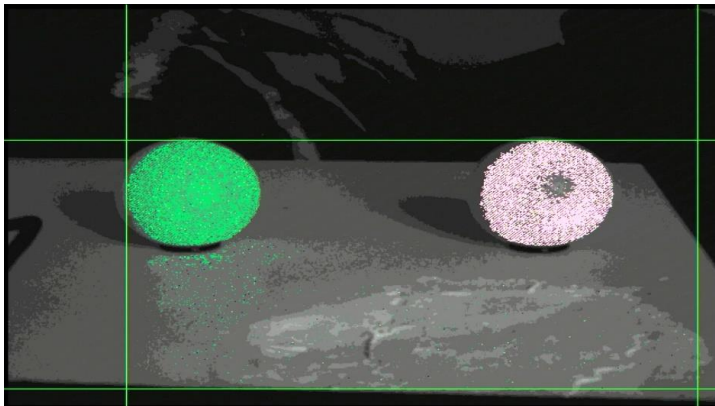


Figure 3.33 - HSV tuned object detection output

The second filter implemented was a 1-pixel radius median filter which would be good for edge preservation, as “For small to moderate levels of Gaussian noise, the median filter is demonstrably better than Gaussian blur at removing noise whilst preserving edges for a given, fixed window size” [92]. A sorting algorithm was implemented as it was needed to make the filter work. The filter was later removed due to the sheer computation size of accessing 3 lines of pixels and storing all them whilst also being able to have a sorting algorithm of $O(n \log(n))$ complexity, all implemented in Verilog, which was considered an excessive use of memory (as we are storing 3 values R,G,B for each pixel and 3 lines worth of pixels to access 9 pixels) and computation power of the FPGA. Furthermore, the 1-pixel radius median filter did not work that well since a 3-pixel radius median filter was more ideal for the video data. Other edge preserving filters such as a Kuwahara-type filters and the Bilateral filter [93] were considered but not implemented as it would be too complex to implement in Verilog and would use up the same amount of memory (if not more) as the median filter.

The third filter implemented was a 5-pixel mode filter [94] that takes 5 consecutive pixels and if the 5 pixels are of the same HSV detection bound, will turn the current pixel evaluated into that bound and otherwise grey. Upon applying this filter, a more generous bound had to be changed for the HSV values to allow the highest probability of success for the objects. This was the filter used in production, as it used the least amount of computation power out of all the filters ($O(1)$ complexity overall), kept edge preservation the same and used the least amount of memory. This allows a very fast lightweight filter that does not use much of the FPGA’s resources which is important for real-time image processing.

Examples of the mean and mode filters in effect are shown below:

3.3.4 Performance Metrics

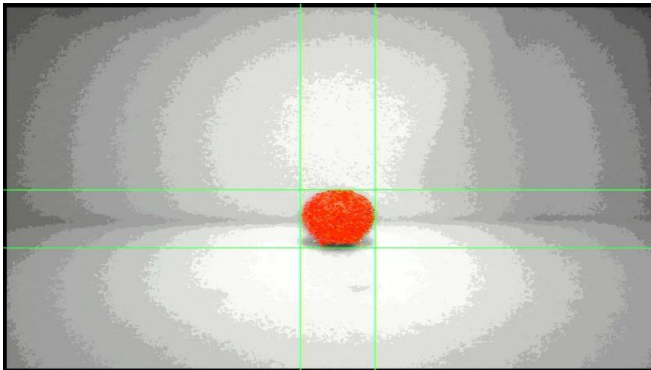


Figure 3.3432 - Red ball detection

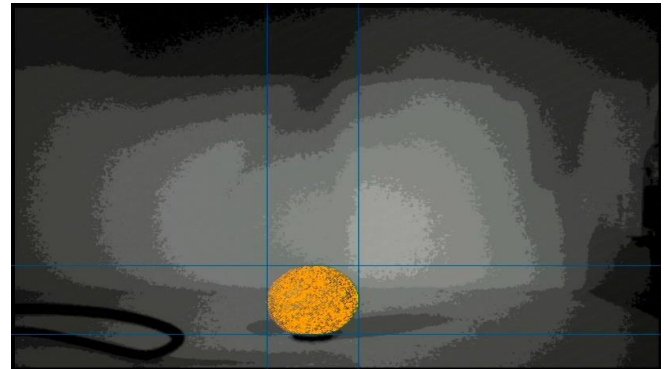


Figure 3.35 - Yellow ball detection

On top of using up to **45975 more 1-bit registers** for a **1-pixel radius median filter** (and being a worse filter) the timing and power analysis shows a very small improvement between the 2 filters.

Median Filter –

Timing Analysis (MAX10_CLK1_50):

- **Slow 1200mV 85C Model:** 62.25MHz
- **Slow 1200mV 0C Model:** 68.05MHz

Power Analysis (Total Thermal Power Dissipation): 681.56mW

Mode Filter (Used in production) –

Timing Analysis (MAX10_CLK1_50):

- **Slow 1200mV 85C Model:** 64.81MHz
- **Slow 1200mV 0C Model:** 69.83MHz

Power Analysis (Total Thermal Power Dissipation): 671.05mW

On top of the significant improvement in memory usage and better object detection using the mode filter, a small improvement in performance can be observed for the use of the mode filter over the median filter.

3.3.5 Variable Lighting

A problem encountered whilst using the camera is that there are multiple varying lighting conditions throughout the day and depending on the weather/time of day objects can go from perfect to undetectable (e.g., the sun can give a ball a shade of yellow but blue skies can be a shade of blue).

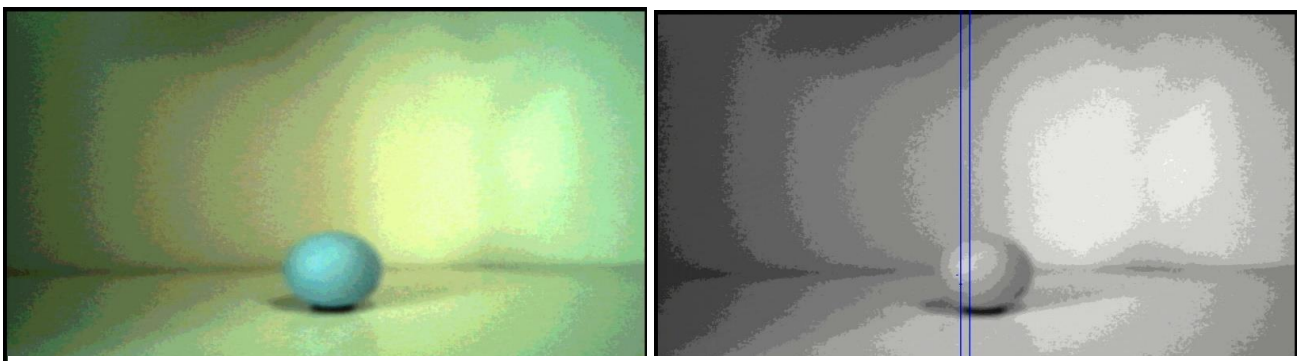


Figure 3.3633 - Undetected object

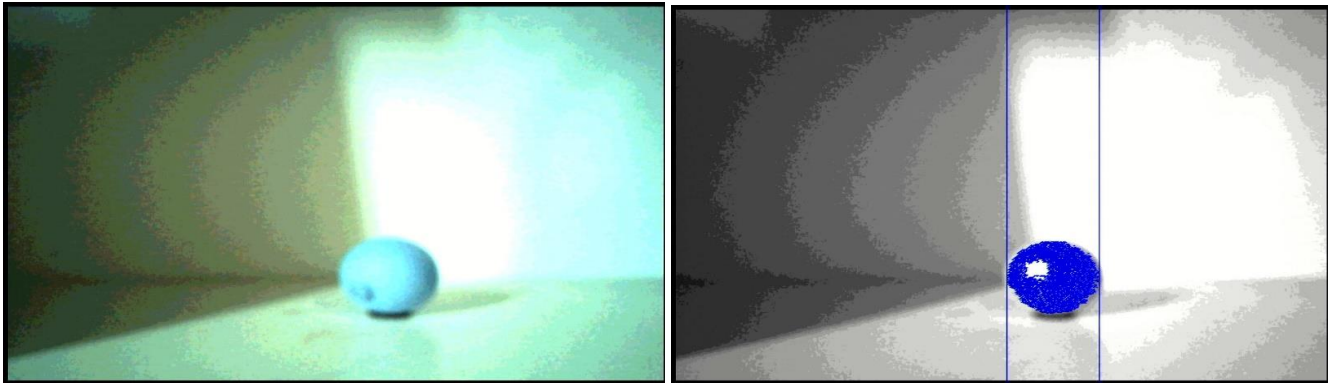


Figure 3.3734 - Detected object

This issue was fixed by taking pictures of the objects under different lighting conditions and evaluating its HSV ranges by analysing the pixels of each picture using Paint.NET [95]. The ranges consist of bright light, dark light, medium light and for some objects, blue lighting, and yellow lighting. Small incremental changes were added until all objects were able to be detected.

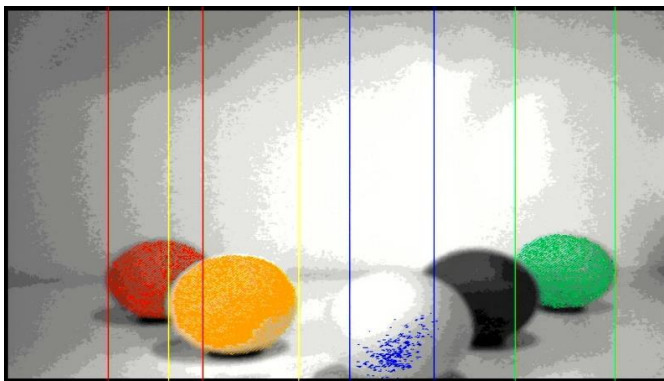


Figure 3.38 - Tuning of bright lighting

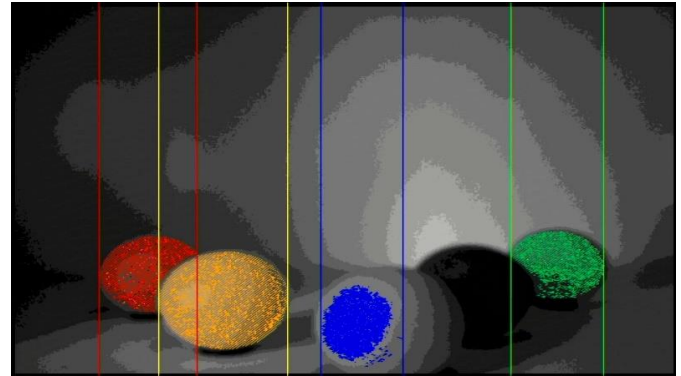


Figure 3.39 - Tuning of dark lighting

Another big issue that was resolved was dark objects (grey ball). This was because every other object (including the grey ball) has a shadow which can be perceived as the exact same HSV values of the dark object not to mention the background would have a lot of dark spots with similar looking colours. To resolve this issue the y bounds of objects were not considered as that would be where the main issue would arise (the shadows of the balls) and instead of calculating distance in 3 dimensions using the x and y bounds, the Vision subsystem instead will calculate the distance in 2 dimensions. This is to remove the inaccuracy of the y bounds due to shadow. The focal length of black was also unique compared to the rest due to the bounds of black being a tiny bit smaller which could cause some inaccuracy in distance calculations. Although 1 dimension is removed distance calculation is still accurate due to the methods implemented to compensate for it. The object detection post HSV value tuning can be seen below:

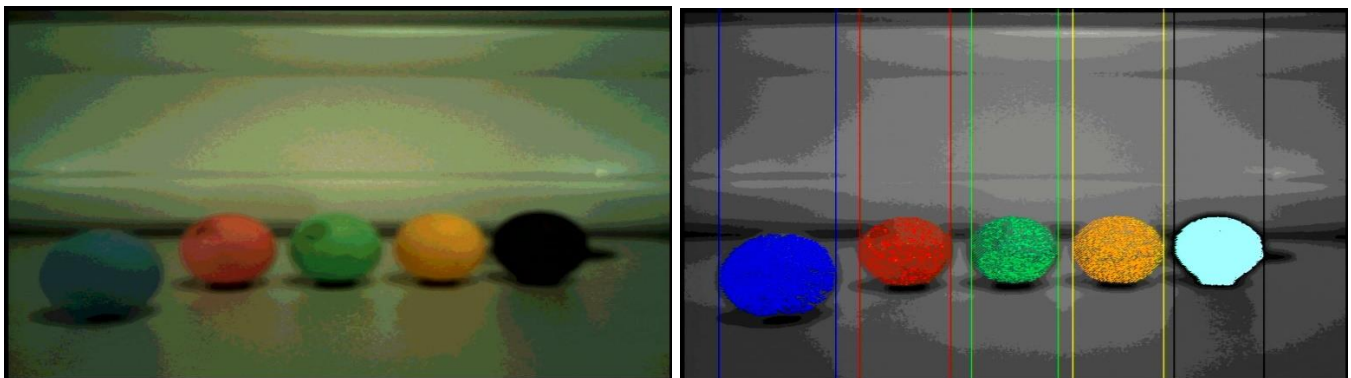


Figure 3.40 - Object detection in dark lighting

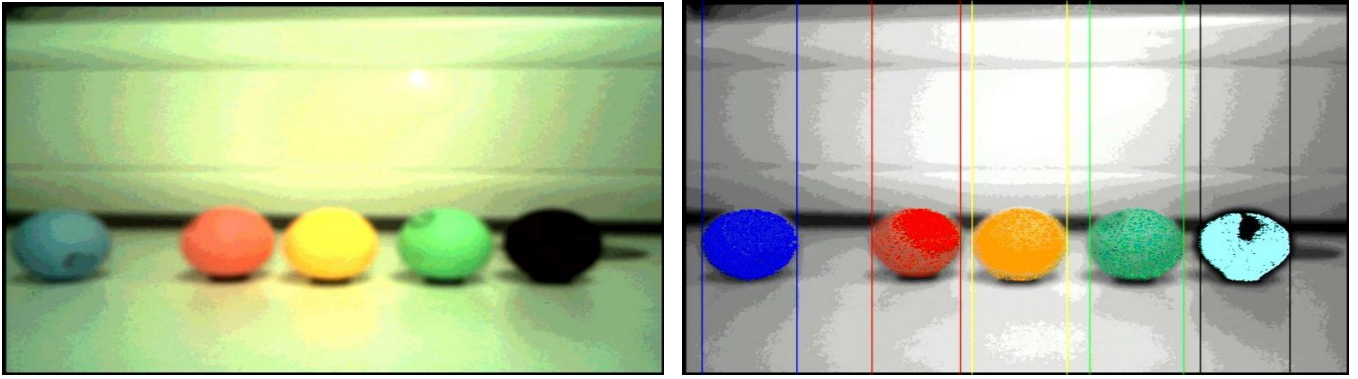


Figure 3.41 - Object detection in bright lighting

3.3.6 Distance Calculation

Distance calculation is done in 3 dimensions, the x, y, and z axis. The formula: $F = \frac{(P*D)}{W}$ [96] was used to calculate a focal length to reverse the formula for calculating distance in 1 dimension $D' = \frac{(W*F)}{P}$ (P = pixel length, D = distance from rover to object, W = diameter of object). This means there are 2 dimensions still not considered. Since the camera is at constant height from the rover, the second dimension can be considered by calibrating the constant F in the equation and providing an offset to fit the equation well (since focal length was variable depending on how far the objects are). When testing to find F, a range between 26cm to 30cm was used due to a variably changing focal length of the camera F, which was found to be (744.3). A quadratic offset is added between these ranges to get the most precise distance calculation regardless of a variably changing focal length. The Vision subsystem will tell the Drive subsystem to either move forward or move backward if it is not within the range of values considered (26cm to 30cm). For the last dimension, the Vision subsystem sends a command to the Drive subsystem to rotate left or right depending on the x_{max} or x_{min} . Vision will keep notifying Drive to rotate until the length of the x_{min} to the middle pixel on the screen is within 30 pixels of the x_{max} to the middle pixel, as this was seen to give the best approximation in trial-and-error tuning. After the rover is within 26-30 cm and the ball is centred on the screen the rover enters distance calculation mode which stops the movement of the rover and sends calculated distances to the ESP32. The ESP32 then takes all the calculated distances and takes the most common calculated distance by vision using a counter up to 100. This eliminates any unwanted noise in distance calculation and provides the best probability of success in measuring the distance accurately.

3.3.7 Communication of Vision

As stated, before Vision communicates with the Control subsystem using an SPI interface. The Verilog code, heavily inspired from an online implementation [97], was modified to connect the 2 conduits to send and receive from the ESP32. The data sent falls under 2 categories, data for the server, and command for the Drive. If a command to Drive is sent this indicates the rover is not within range to calculate distance and will have to move towards an object to get a more precise measurement. If the Vision subsystem is within range, then it will enter distance calculation mode as stated previously. This means its constantly sending data representing the colour of the object and the approximated calculated length. When a flag has been received indicating the coloured object has been recorded, the system will immediately update the specified object as an obstacle and will then move on to find other objects.

3.3.8 Testing

The testing of Vision could be split up into 3 stages. Basic testing is the first stage of vision testing and includes the simplest implementations to allow the rover to function (e.g. simple object detection, simple communication, simple distance calculation etc). Most of basic testing is implementation of basic functionalities and can be either observed on the nios2-terminal or the camera feed. Intermediate is the second stage of testing and builds on the basic functionality that Vision already has implemented. This

includes improving filtering, and incorporating multiple filters to improve object detection, as well as improving distance calculations by using offsets on focal length or filtering of noisy values. Most of intermediate testing involved making small incremental changes and observing any improvements made. Since compile times in Quartus are very long, whenever possible Icarus-Verilog was used as a Verilog simulator to speed up testing. Advanced testing is the last stage of vision testing and builds on both the basic functionalities and its intermediate improvements. Advanced testing is mainly used for robustness and ensures the Vision subsystem has the best possibility of success under real-time working conditions. Some examples of advanced features include testing multiple filters to achieve the best lightweight filter, picking the most optimised communication interface between Vision and Control, and lastly tuning the values to detect objects under multiple lighting conditions.

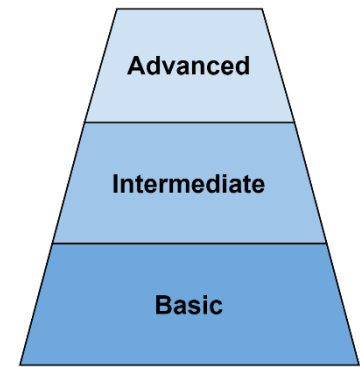


Figure 3.42 - Vision Testing Hierarchy

3.4 Drive

The main aim of the Drive subsystem is to provide distance measurements, direction and speed control, and ensuring optimal overall mobility, including turning methods for different angular displacements. The subsystem comprises the use of the DC motors, the switch mode power supply (SMPS) to generate the variable voltages to control the Rover speed, the optical flow sensor for distance measurement and the Arduino. The result is a single code that can fully operate and control the Rover mobility.

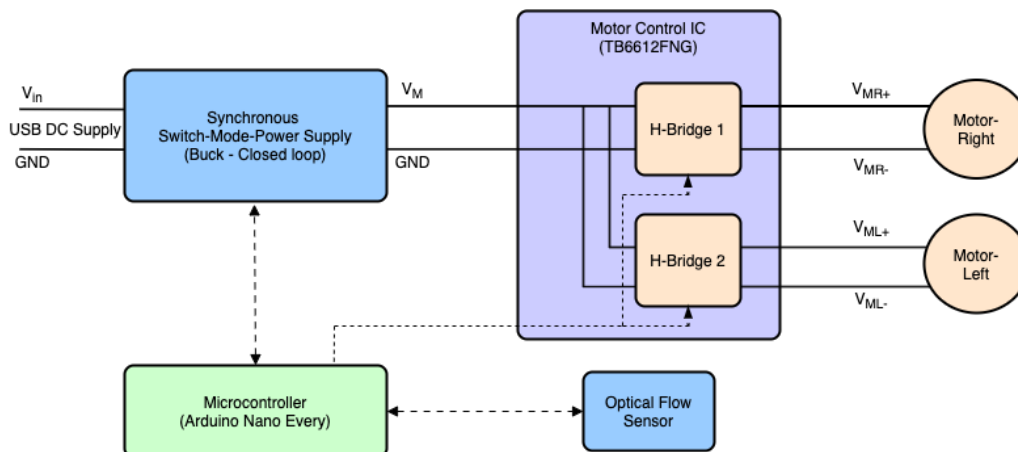


Figure 3.43 - Drive Structural Diagram

The final goal of the rover is to move differently depending on the operating mode:

- In Remote control mode the rover moves accordingly to the commands sent by the user via the web server.
- In Exploration mode the rover receives direct commands from the Vision subsystem and can avoid obstacles identified by the camera.
- In Coordinate mode the rover can drive autonomously, rotating by precise angles and moving by precise distances depending on the coordinates provided by the path finding algorithm.

In each of these modes the coordinates and the angles are tracked and sent to the ESP32 and consequently are displayed on the web server.

3.4.1 The Design Process

The initial step consisted of testing the provided codes for the motors and for the optical sensor to identify any software or hardware constraints to consider during the design. This allowed the correction of a bug at an early stage related to the distance measurement of the optical sensor which was wrapping around approximately every 20 cm due to an incorrect conversion from counts per inch to mm. It was

also noted that the sensor does not provide absolute x and y coordinates, but rather provides positions relative to the orientation of the rover, a factor that had to be taken into consideration when computing the value of the current position to send back to the server and when calculating the current angle trigonometry.

3.4.2 Position Control

The original plan was to aim for a PI controller for both distance and rotation as a final design. To begin with, a gradual approach was preferred, which resulted in first creating a closed loop (CL) controller for distance and an open loop (OL) controller for rotations (time-based). The CL controller for forwards and backwards movements uses the error between the measurement provided by the optical sensor and the desired distance. Depending on the magnitude of the error, a forwards or backwards compensation is applied. To ensure that the target is precisely reached, the motors are automatically slowed down when in the range of 1.5cm from the target, avoiding any overshoots and/or oscillations. The OL controller for rotation was designed by measuring the time the rover needed for specific angulations and stopping when the right angle had been reached. However, this method was inaccurate and dependent on the motor speed, making it an impractical solution.

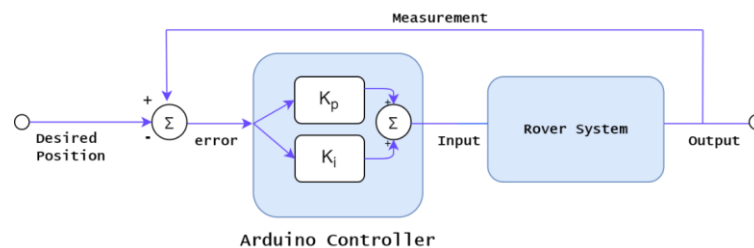


Figure 3.44 - Closed-loop controller

The final PI controllers for distance and rotation are both based on the same principle of increasingly reducing the error every loop iteration by using a proportional and integral gain. The implemented PI controller is incremental meaning that the algorithm computes the time differences of the output and adds up the increments. The PI gains have been tuned in order to minimise overshoot and oscillations and have been found by trial-and-error method: $K_p = 0.25$ and $K_i = 0.58$.

Additionally, the motor speed is decreased when the target destination is in the ± 5 cm and is further decreased when in the ± 5 mm range for distance measurement. For angle measurement the speed is initially decreased when in the $\pm 5^\circ$ and is further decreased when in the $\pm 2^\circ$. Speed control has been inserted in the PI controllers to always ensure, regardless of the initial speed of the motors, that the rover stops in the correct position. Therefore, the maximum distance error is of ± 2 mm while the maximum angle error is $\pm 1^\circ$. One of the greatest advantages of the use of the PI controllers instead of the previously designed controllers, is the ability to correct any eventual movement error due to the asymmetrical weight distribution of the power bank and any imparity in the motors. The final implementation for these controllers can be seen below:

```
//PI controller for Distance
float pidDistance(float pidDistance_input){

    float e_integration;
    e0Distance = pidDistance_input;
    e_integration = e0Distance;

    delta_uDistance = kpDistance * (e0Distance - e1Distance) + kiDistance * Ts * e_integration;
    u0Distance = u1Distance + delta_uDistance; // this time's control output

    u1Distance = u0Distance; // update last time's control output
    e1Distance = e0Distance; // update last time's error

    return u0Distance;
}
```

Figure 3.45 - PI distance controller

```

bool move_PIcontroller(int desired_x, int desired_y){
  Serial.print("ENTERING DISTANCE MOVEMENT FORWARD");
  float error_distance = sqrt(pow((current_x-desired_x),2) + pow((current_y-desired_y),2));
  bool forward = (((current_y + error_distance * cos(current_angle) < desired_y + 5) \
    && (current_y + error_distance * cos(current_angle) > desired_y - 5))) ? true : false;

  float cDistance = pidDistance(error_distance);
  Serial.println("error distance is " + String(error_distance));
  if(error_distance > 100){
    pwm_modulate(abs(cDistance)*0.01+0.5);
    Serial.print("Modulating it to 0.6");
  }
  else if(error_distance > 50){
    pwm_modulate(0.35);
    Serial.print("Modulating it to 0.45");
  }
  else{
    pwm_modulate(0.25);
    Serial.print("Modulating it to 0.35");
  }
  if(error_distance > 5 && forward){
    DIRRstate = LOW; //goes forwards
    DIRLstate = HIGH;
  }
  else if(error_distance > 5 && !forward){
    DIRRstate = HIGH; //goes backwards
    DIRLstate = LOW;
  }
  else{
    stop_Rover();
    return true;
  }
  digitalWrite(DIRR, DIRRstate);
  digitalWrite(DIRL, DIRLstate);

  float measuredDistance = sqrt(pow(dy_mm,2) + pow(dx_mm,2));
  int constant = -1;
  if (forward){
    constant =1;
    current_angle += 0.0104533;
  }
  else{
    current_angle -= 0.0104533;
  }
  current_y = current_y + (constant * measuredDistance) * cos(current_angle);
  current_x = current_x + (constant * measuredDistance) * sin(current_angle);
  return false;
}

```

Figure 3.4636 - Forward/Backwards function with controller

```

bool Turn_PIcontroller(float desired_angle, float dy_val, float dx_val, float *current_ang){
  Serial.println("Desiredangle should have been 45 but is " + String(desired_angle));
  Serial.println("dy_val is " + String(dy_val) + "dx_val is " + String(dx_val));
  float error_angle = desired_angle - toDegrees(current_angle);
  float cDistance = pidDistance(error_angle);
  if(error_angle < -10 || error_angle > 10){
    pwm_modulate(abs(cDistance)*0.01+0.3);
  }
  else if(error_angle < -5 || error_angle > 5){
    pwm_modulate(0.3);
  }
  else{
    pwm_modulate(0.25);
  }
  if (abs(error_angle) >= 3 && error_angle <= -3){
    DIRRstate = LOW; // turns Left - rotates anticlockwise
    DIRLstate = LOW;
  }
  else if (abs(error_angle) >= 3 && error_angle >= 3){
    DIRRstate = HIGH; // turns right - rotates clockwise
    DIRLstate = HIGH;
  }
  else{
    return true;
  }
  digitalWrite(DIRR, DIRRstate);
  digitalWrite(DIRL, DIRLstate);

  float measuredDistance = sqrt(pow(dy_val,2) + pow(dx_val,2));
  float alpha = asin(measuredDistance/(2*r)) * 2 ;
  float angleval = *current_ang;
  angleval = (dx_val < 0) ? (angleval + alpha) : (angleval - alpha);
  angleval = (angleval > 3.14159265359) ? (angleval - 3.14159265359) : angleval;
  angleval = (angleval < -3.14159265359) ? (angleval + 3.14159265359) : angleval;
  Serial.println("current angle is " + String(toDegrees(angleval)));
  *current_ang = angleval;
  return false;
}

```

Figure 3.35 - Turning function with PI controller

3.4.3 Speed Control

It is possible to operate the rover at different speeds due to the use of the SMPS as the DC-DC interface between the battery and the motors. In the proposed implementation, the SMPS has been set to synchronous CL Buck mode to maximise the power efficiency with respect to the asynchronous mode, in which a MOSFET rather than a diode is used for the path in the off state. The voltage range that can be provided to the motors is 0-5V, with 0V meaning that the rover does not move and 5V indicating that the rover moves at its maximum possible speed. During the final testing of the rover, a power bank rather than a wall socket was used to power up both the SMPS and the ESP32, and by measuring the input voltage this was found to be 4.76V. Regardless of the fluctuations in input voltage, the advantage of the SMPS is providing a constant output voltage to power the motors, so any decrease in input voltage does not affect the rover performance at the desired speed.

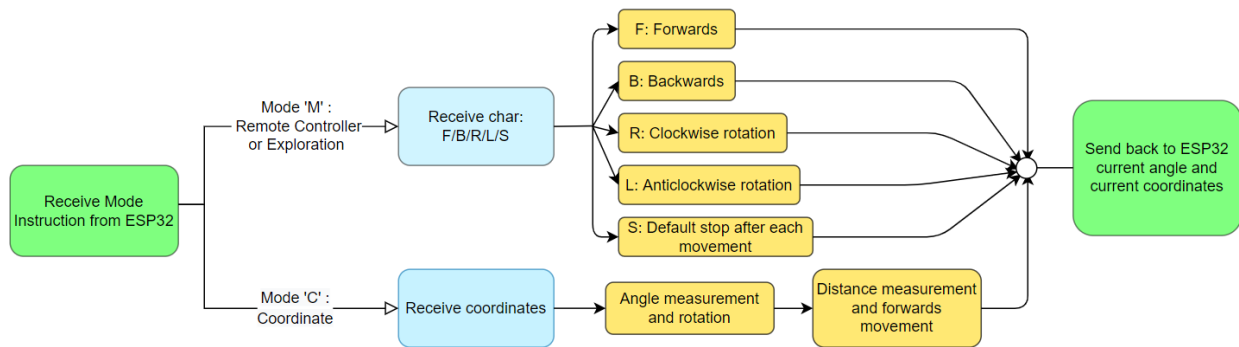
To implement speed control using the SMPS, the reference to the potentiometer (**analogReadA2**) was removed from the motor code sampling function. It was instead substituted with a function with different values between 0 and 1023 which represent the voltage between 0 and the analogue reference that is 4.096V. This method effectively allowed the change of the rover speed without needing to tweak the potentiometer. Alternatively, inside of specific functions such as the PI controllers, the **pwm_modulate()** function has been called to write directly on pin 6 of the Arduino, which is in charge of the pulse width modulation (PWM) outputted by the SMPS. This function was not employed to stop the motors because the SMPS output does not instantaneously turn 0, while it is optimal to slowly decrease the motor speed. On average, the voltage value that has been used during testing is 2.5V (**sensorValue2** = 624) which allowed the rover to cover a distance of 1m in approximately 6.13s. Therefore, the average speed was computed to be $v = 0.163 \text{ m/s}$ and this has allowed to measure the motor's rpm to be:

$$rpm = \frac{60}{2\pi \cdot radius_{wheel}} \cdot v =$$

$$= \frac{60}{2\pi \cdot 0.0325m} \cdot 0.163ms^{-1} = 47.89 rpm$$

3.4.4 Code Structure

The following diagram illustrates the structure of the code and the functionalities it provides depending on the characters received from the ESP32:



3.4.5 Controlling movement in different modes

To operate Exploration mode and Remote controller mode, the rover efficiently uses the same Drive code

Figure 3.4837 - Drive functional diagram

by receiving the character 'M' transmitted by the ESP32. Depending on the next sent character, provided by either Control or by the Vision subsystem, the rover moves accordingly as seen in the diagram above.

When the rover receives a 'C', it enters Coordinate mode. While in this mode, it receives pair of coordinates one at a time generated by the path finding algorithm and computes the correct angle and distance needed to reach them.

The computation of the correct angle that the rover needs to face depends on the current angle, the desired angle given by the specific coordinates, and the quadrant that the rover is currently in. For the first quadrant, the desired angle is found by calculating $\tan^{-1} \left(\frac{x_{desired} - x_{current}}{y_{desired} - y_{current}} \right)$ and then inserting it directly into the PI controller for rotation. For the other three quadrants, the signs of $x_{desired}$, $x_{current}$, $y_{desired}$, $y_{current}$ are changed accordingly to respect the sign of the angle.

After the rotation has been completed, the rover reaches the desired destination with the distance PI controller. This method allows to successfully reach all the coordinates provided by the path finding algorithm ensuring that the rover avoids the obstacles, as well as sending updates to the Control subsystem every 200ms with the rover's current position and the angle its' facing.

The diagram below on the left shows the possible rotations that the rover needs to achieve depending on where the coordinates are positioned in the cartesian plane. The diagram on the right illustrates an example of how the rover would adjust its orientation based on the provided coordinates.

Initially, the position of the rover was calculated using the following heuristics:

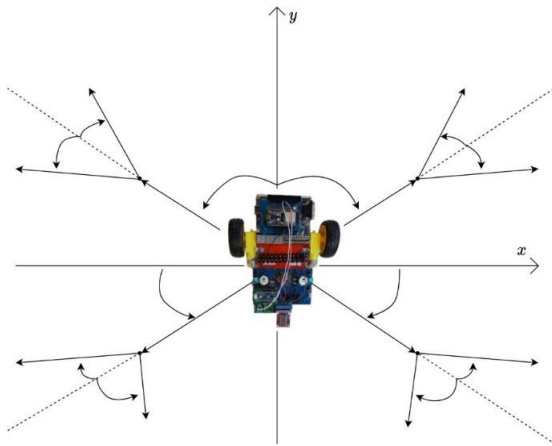


Figure 3.49 - Possible rotations to reach destination

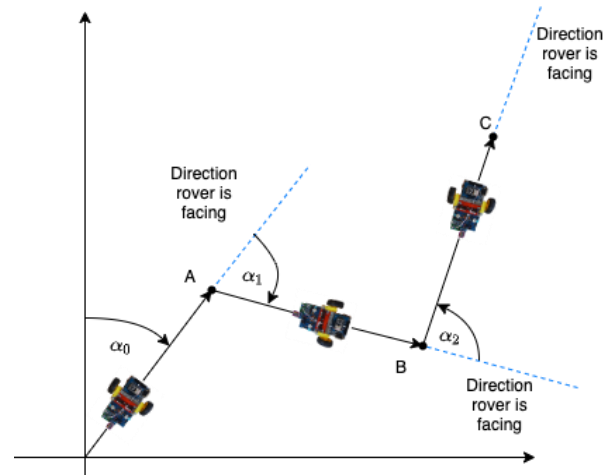


Figure 3.50 - Chaining coordinates together

The rover's angle doesn't change during forwards / backwards movement, allowing the use of simple trigonometry based on the change in y-value of the optical sensor to determine the change in xy-coordinates.

The rover's xy-position doesn't change during rotations, allowing arc-length calculations based on the changes in sensor xy-values to be used to determine the change in angle.

However, it was noted during testing that for all three modes there is a delay between the end of the rover's movement and the catching up of the measurement provided by the optical sensor. Fixing this issue was of particular importance because it could cause an accumulation of errors affecting the rover's trajectory and the tracking of its position. An initial attempt to fix this delay was to include a counter of 30 loops that would force the rover to wait and continue to perform the position calculations outlined above while the rover was stopped, giving the sensor measurements time to catch up. This implementation worked correctly for forwards and backwards movements, but it was lacking accuracy for rotations. Furthermore, the delay in sensor readings was highly variable, with 5-17 iterations worth of delay, meaning that very often part of the 30-loop wait time was wasted, leading the rover to be less efficient. Another attempt to solve this issue involved tracking the previous movement during a stop, and continuously updating the position based on that movement. This was because in the times the rover would be halted between movements, the sensors would have enough time to catch up without having to add an artificial delay. However, this still led to some inaccuracies in tracking the position of the rover due to the assumptions made for position calculation.

The final, best working solution was to operate the tracking of the position independently to the specific movement of the rover. This is done by taking a ratio between dx and dy changes and depending on that ratio one would be able to deduce what movement the rover is currently executing. After numerous testing, it was deduced that the rover is moving forward/backwards whenever the magnitude of dy is approximately greater than 4 times the magnitude of dx regardless of motor speed. This is because the change in x and y both increase linearly as motor speeds increase implying that this method takes various speeds into consideration. Anything outside this region will be an angle calculation. In testing, it was noted that imbalances in the motors would cause the rover to shift to the left. To compensate, whenever the rover was moving forward, an angle offset was added to ensure the PI controller would be able to accurately keep the rover facing towards its destination.

3.5 Power

The Energy subsystem is an essential element of the rover design as it provides power as well as managing the recharging of the battery and long-term health of the cells. To do so, it is required to closely monitor the characteristics of the cells to ensure the rover can function for the longest possible time. The system needs to charge the battery using solar panels as the source of energy and monitor the state of charge while the rover is operational to ensure that the rover is returned to charge when necessary. The process of designing a system to meet the criteria is an iterative one and requires continuous testing and improvement.

The charging station can be both on-board or static. Although an on-board energy system would allow the rover to charge from anywhere and therefore have 'infinite' range, it would drastically increase the weight and risk the solar panels being damaged during transit. It was therefore decided to design a static charging station to which the rover could return to for charging. Whilst this may restrict the rover's range to a circumference around the station, it ensures the long-term safety of the solar panels, which is of critical importance as these may need to be deployed in difficult to access environments. In addition, due to low solar irradiance on mars (590W/m^2) [98], the rover would be restricted to the equatorial regions regardless of this decision.

3.5.1 Cell Characterisation

The first step was to characterise the batteries using the first test code provided. All graphs of voltage over time had a similar shape as shown in Figure 3.39. The cell capacity can be found by measuring the total time for discharge and multiplying by current $\text{cell capacity} = t_d * I$. When tested, the capacities of the batteries used ranged from 497mAh to 517.3mAh. This highlights the importance of cell balancing; when being charged together, the 497mAh cell (lower capacity) will charge first and, if charged beyond the maximum, the cell will become damaged. Cell balancing is the process by which the individual cells in the battery are managed such that they charge up to the same level at the same time. This maximises the capacity of the battery by maintaining equivalent state of charge for every cell and prolongs the useful life of the cell, allowing it to be charged for the greatest number of cycles.

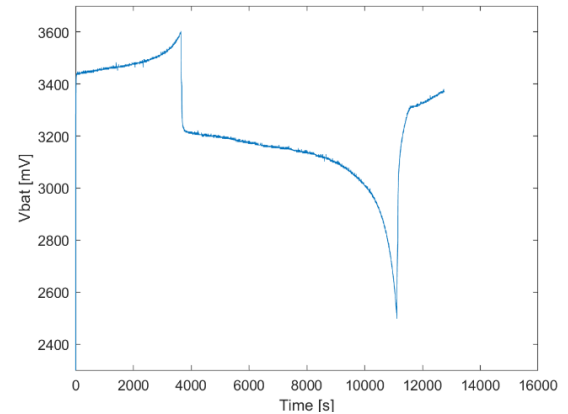


Figure 3.381 – Battery characterisation

3.5.2 State of charge (SOC)

The state of charge gives the level of charge of an electric battery relative to its capacity, expressed as a percentage

$(SOC(t) = \frac{Q(t)}{Q_n} * 100)$ [99]. This cannot be measured directly at any instant as this would require very specialist equipment, but it can be estimated based on calculation. Several methods were researched before reaching a final decision, two of which are outlined below.

Voltage method – perhaps the simplest method, this involves measuring the open circuit voltage (OCV) and using a known discharge curve of the battery to convert the voltage reading to an SOC value. The problem with this method is how voltage is significantly affected by current which is, in turn, affected by temperature. This method is highly effective for cells which have a linear discharge curve such as lead-ion however, the cells in use are lithium-ion which have a non-linear discharge curve. In this region, voltage changes are exceedingly small and do not truly reflect the cells state of charge. A further downside to voltage lookup is how, even when used correctly, it has an accuracy of 10-15%. This could

become very problematic for the rover during operation, for example, the SOC may read much greater than the true value and this could mean the rover runs out of power before it can trigger recharge.

Current integration - Current integration, or coulomb counting, is the most common technique for SOC used in portable devices. This method involves integrating the current over time, therefore requiring monitoring of the in-and-out flowing current. To do so, one must have an initial SOC measurement and be able to measure the battery current. The main issue with this method is how it does not consider losses during charge / discharge which would give less releasable charge than expected. In addition to this, there is also a need for regular recalibration due to the decreasing state of health of the battery as it ages. To calculate SOC at a time t , we can use this equation $SOC(t) = SOC(t - 1) + \frac{i(t)}{capacity} \Delta t$ [100] and then simply scale by SOH (see SOH) to correct for the degrading battery health.

Both methods have merits however, as highlighted above, the discharge graph of the lithium-ion batteries has a non-linear region in which voltage lookup cannot be used. This means the main method of SOC monitoring will be coulomb counting. This is made possible by the current sensing device on port B of the SMPS. The one issue with this method was the requirement of initial SOC but this can be found using a voltage lookup table which has been created using the discharge cycle. To obtain the graph of SOC in figure 3.52, code was added which calculated the SOC using the equation above and obtaining a value as a percentage. In the process, a voltage lookup table could also be created which related certain voltages to a SOC. During charging and usage on the rover, the power system will be able to keep track of the SOC in this way and send the data as a percentage to the Control sub system. The Control sub system can thus manage the display of battery SOC. The SOC can also be used to estimate the total range of the rover.

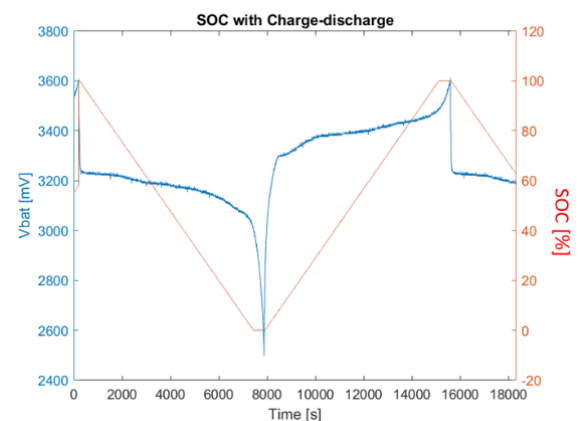


Figure 3.402 Graph of charge cycle with SOC measurement

3.5.3 State of health (SOH)

Over time, as the lithium-ion cell is used for more cycles, the total cell capacity will degrade. The extent to which the battery has deteriorated can be quantified using the notion of SOH. The SOH can be defined as $SOH = \frac{total\ capacity\ today\ (Ah)}{BOL\ capacity\ (Ah)} * 100$ where BOL means beginning of life. One simple way to monitor this would be to keep track of the total capacity of the cell. This can be done by integrating the current-time graph during discharge; the same calculation that was used to find the cell capacity in the first tests. In the short term, SOH does not vary greatly. This means a good method to keep track of it would be to run a controlled discharge cycle at regular intervals at the charging station. This is the most accurate method to calculate SOH since the discharge curve during operation of the rover may not be linear (e.g. as rotation and straight motion draw different amounts of current) leading to an inaccurate value. The final power system would therefore check the SOH at an interval of once per day by running a controlled discharge cycle at the charging station.

3.5.4 Solar panel characterisation

The solar panels provided have a nominal maximum current of 230mA with a voltage of 5V. This power output is not, however, guaranteed as it is heavily dependent on the intensity of sunlight available. When operating with the halogen lightbulb, a single solar panel was found to give an open circuit voltage in the

range 5.3-5.45V however, the short current was incredibly low at only 20-50mA. The current improved when used in direct sunlight, achieving values closer to 190mA. As the standard charging current is 0.5C (=250mA), a conclusion can be made that, to charge the cells at the required constant current, the panels are best arranged in parallel. This means the individual currents will sum to give enough current to charge at 250mA.

Furthermore, another important characterisation method measures the maximum power point. The data below was collected by sweeping the current using the Arduino and obtaining values of current and voltage from the solar panels whilst in a parallel connection and in a Boost setup. The IV curve highlights how, at high voltages (>4.8V), the current drops very quickly so the aim should be to operate in the linear region where the current remains constant. The PV curve shows that the maximum power of 1.46W is obtained at approximately 4.6V. When charging with constant voltage, we will need to maintain a voltage of 3.6V across the batteries. This voltage avoids the large current drop, but it compromises on the maximum possible power (which occurs at 4.6V).

3.5.5 Maximum power point tracking (MPPT)

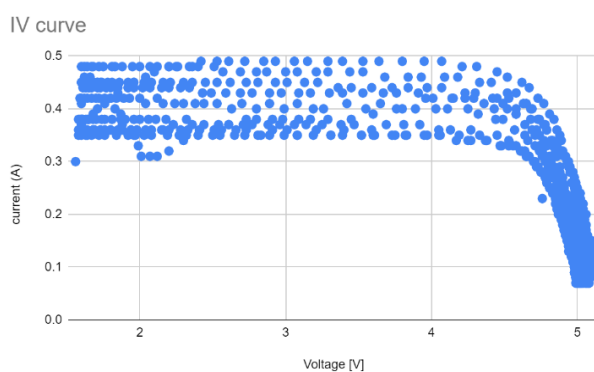


Figure 3.53 IV characteristic of PV panels

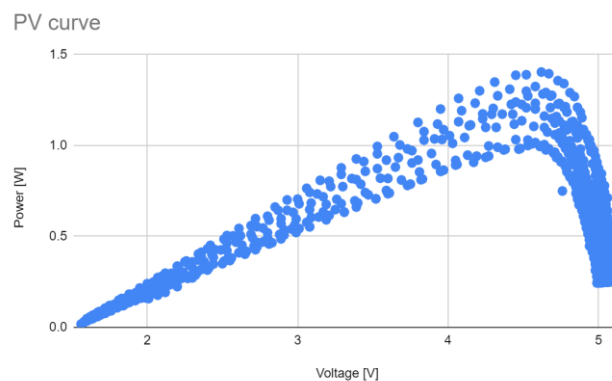


Figure 3.54 Power against Voltage for PV panels

MPPT works to ensure the solar panels are working at maximum power in varying solar irradiance and temperature. One implementable algorithm is 'Perturb and Observe' in which the operating point is continuously altered, and the power is observed until the maximum power point is met. In this situation, this would mean altering the PWM duty cycle to sweep the current as performed in the previous experiment. The main conflict here is that the battery also requires PWM control to maintain its own current. As a result, it is not possible to implement MPPT and constant current charging at the same time. One solution to this would be to only use MPPT when the current is detected to be extremely low (far right of IV curve) and the MPPT algorithm would be used to shift the operating point so as to increase the current.

3.5.6 Charging method and balancing

The method for charging used in the first tests relied on constant current. This is not ideal and, as seen in the rest states, the voltage will drop gradually due to the internal resistance of the battery. To overcome this issue, the cells will be charged in two stages, in accordance with the industry standard for lithium-ion cells:

Constant current (CC) – In this stage a constant current is applied at 0.5C to 1.0C to charge up to approximately 80% of full capacity

Constant voltage (CV) – A constant voltage is applied to the terminals to charge up to 100%.

This method will ensure that the maximum capacity of the cells is reached during charging and will also prevent voltage drops during any rest periods.

The individual cells can be arranged in parallel or series, both systems with advantages and disadvantages. A parallel combination will simplify the operation of the CV charging mode; however, the splitting of current depends on the individual equivalent series resistance (ESR) so is therefore uncontrollable. A series combination allows for more controlled current during CC mode but, as the SMPS applies voltage at the two ends, the voltage in CV mode may not be the same for both. Further disadvantages of the series combination include the requirement of potential dividers to take measurements of voltage across port B of the SMPS and the inconvenience of breaking the charging circuit in order to take measurements. For these reasons it was decided that the battery would function best with parallel cells.

The next caveat involves the number of cells to use in the battery. Due to the limited number of ports in the Arduino, it was only possible to test with 2 cells as any more would require the removal of LED indicators which were important to use in the testing process. The charging method could easily be extended to 3 cells by adding extra checking states.

Figure 3.55 represents the FSM which was coded into the Arduino. It consists of two loops for CC and CV with transition from one to the other only when the voltage is at 3.48V (approximately 85% charged).

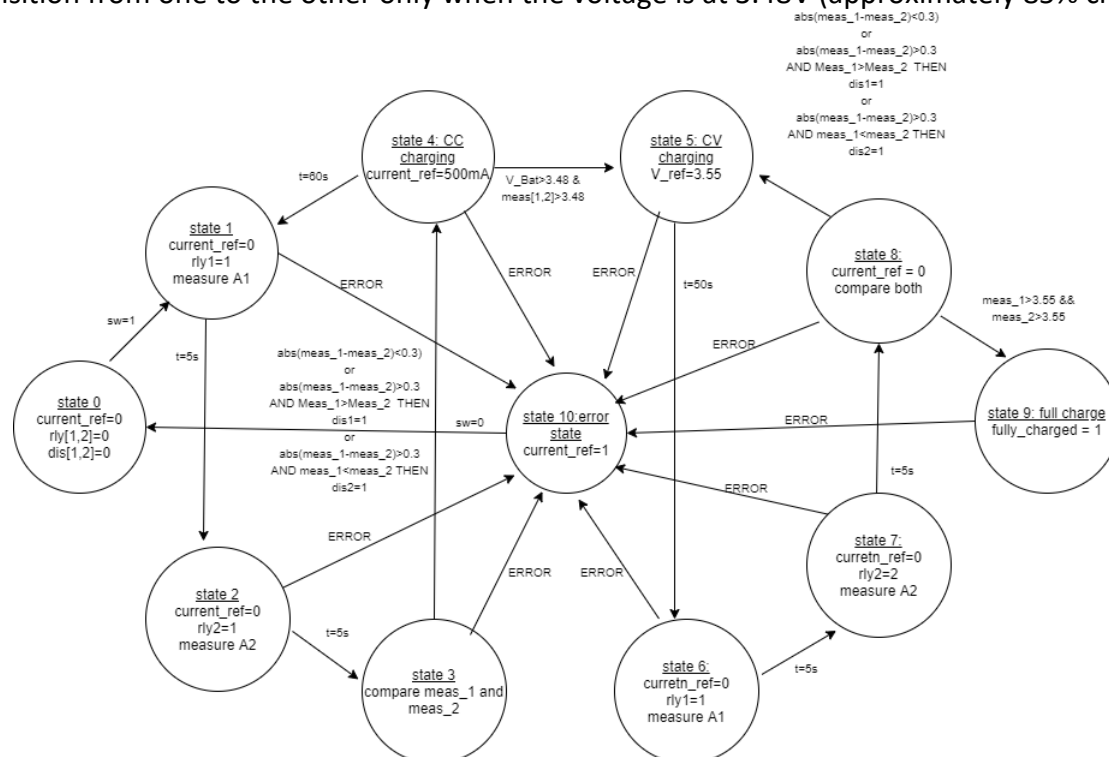


Figure 3.55 battery charging FSM

The CV loop will then continue until the battery is fully charged. During the testing process, the program was run several times with different values for wait time in the 'checking' states and conditions for transition. This included altering the time for which the battery stayed in CC mode before returning to state 1. This tuning was necessary as in order to compromise between total charging time and battery balancing. For example, if two unequal cells were held in a charging mode for too long, the difference in charge would be amplified: potentially damaging the cell.

The SMPS was used in Buck mode as the voltage only needed to be held at a maximum of 3.6V and the solar panels are able to provide more than this. This also allows for easy current measuring due to the placement of the current measuring resistor on port B. Current and voltage control was achieved using a

PID controller with values tuned for best control. To do this, conditions were needed in the fast loop to activate the correct PID controller depending on the present state.

One intended extension of this design, which was halted by cell faults, would be the integration of a maximum power point tracking (MPPT) system. This would only be used to bring the solar panels into the correct operating point when the current was found to drop significantly. This compromise is required due to how both CC and MPPT require PWM modification.

3.5.7 Overall integration of system

The Energy subsystem has been designed as a static charging station with the intention that the Control subsystem is able to monitor the SOC during discharge and tell the rover to return to the station when the SOC is at a critically low level. The battery is arranged in parallel so will have a nominal voltage of 3.2-3.6V and this is required to power the Drive unit which draws 0.011A and 4.3V from the source. Although, the Drive SMPS steps down the voltage to 2.5V, the cell voltage should still be boosted as, when at lower SOC the batteries could be at a voltage below 2.5V which would not be enough to drive the motors. One solution to this problem would be to include another SMPS that will first boost the voltage to a level similar to the USB source (~5V). This would also mean that the battery would be connected to port B allowing for easy current measurement.

The ability to measure current from the cells is also beneficial in SOC monitoring during motion. The Drive subsystem is able to use the coulomb counting method to keep track of the SOC during operation by using the initial SOC value of 100% (initial SOC when leaving charging station) and measuring the decrease during operation. The current can also be used to give an estimate of range of the rover using the equation $range = \frac{\mu_n * SOC * SOH * 3600}{current\ drawn} * speed\ of\ rover$, where the cell capacity, μ_n , is multiplied by 3600 to obtain a value in seconds. This gives an estimate of the distance the rover is able to travel in meters given the SOC and SOH.

Another important aspect of the power system's integration is its communication with the Control module. While charging at the station, the power system will send a packet of data through UART consisting of two 3-digit numbers representing SOC and SOH. From this the charge progression can be displayed along with the health of the battery.

3.6 Improvements

Some further improvements to the overall system could include:

- Adding a login page to limit the number of users who can control the system.
- Additional optimisations to the pathfinding algorithm to account for cases where the heuristics fail.
- Using more advanced equipment to measure SOH by recording the increase of ESR in batteries

4 Conclusion

Over the course of this project, Debonair has grown into a system which meets all the requirements that were initially set. With full support for 3 different functional modes, this rover is equipped to handle a variety of different situations, all paired with a robust and secure backend communication platform which seamlessly integrates with an enticing and intuitive front-end UI. Additionally, fully fledged power infrastructure has been designed with integration support, which although untested due to COVID limitations, would enable the system to be self-sustainable when deployed into a remote environment.

Although currently limited to a 4m² environment, changes could be quickly made to significantly increase the range that the rover could explore, requiring only a simple update to the relative coordinates on the web-app. The scalable and modular nature of the system guarantees that the number of nodes in the system could be increased with ease, and that any additional functionality could be incorporated quickly.

5 Bibliography

- [1] NASA, "NASA Science Mars 2020 Mission Perseverance Rover," NASA Science, [Online]. Available: <https://mars.nasa.gov/mars2020/spacecraft/rover/communications/>. [Accessed 12 6 2021].
- [2] CPrime, "What is agile?," 2021. [Online]. Available: <https://www.cprime.com/resources/what-is-agile-what-is-scrum/>. [Accessed 12 5 2021].
- [3] SNDK corp, "Amazon Elastic Cloud Compute (EC2)," 2021. [Online]. Available: <https://www.sndkcorp.com/amazon-ec2/>. [Accessed 9 6 2021].
- [4] M. Serozhenko, "MQTT vs. HTTP: which one is the best for IoT?," 2017. [Online]. Available: <https://medium.com/mqtt-buddy/mqtt-vs-http-which-one-is-the-best-for-iot-c868169b3105>. [Accessed 17 5 2021].
- [5] C. Bernstein, K. Brush and A. Gillis, "MQTT (MQ Telemetry Transport)," 2021. [Online]. Available: <https://internetofthingsagenda.techtarget.com/definition/MQTT-MQ-Telemetry-Transport>. [Accessed 17 5 2021].
- [6] MQTT, "MQTT: The Standard for IoT Messaging," 2021. [Online]. Available: <https://mqtt.org/>. [Accessed 17 5 2021].
- [7] ScalAgent, "Benchmark of MQTT servers," 2015. [Online]. Available: http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf. [Accessed 25 5 2021].
- [8] R. Harper, "Duck DNS," 2021. [Online]. Available: <https://www.duckdns.org/domains>. [Accessed 24 5 2021].
- [9] NginX, "NGINX Documentation," 2021. [Online]. Available: <https://docs.nginx.com/nginx/admin-guide/installing-nginx/installing-nginx-open-source/>. [Accessed 28 5 2021].
- [10] M. Dabbs, "The Fundamentals of Web Application Architecture," 2019. [Online]. Available: <https://reinvently.com/blog/fundamentals-web-application-architecture/>. [Accessed 17 5 2021].
- [11] MuleSoft, "What is a REST API," 2021. [Online]. Available: <https://www.mulesoft.com/resources/api/what-is-rest-api-design>. [Accessed 17 5 2021].
- [12] Red Hat, "What is a REST API," 2020. [Online]. Available: <https://www.redhat.com/en/topics/api/what-is-a-rest-api>. [Accessed 18 5 2021].
- [13] S. Santos, "How to use ESP32 Dual Core with Arduino IDE," 2018. [Online]. Available: <https://randomnerdtutorials.com/esp32-dual-core-arduino-ide/>. [Accessed 4 6 2021].
- [14] Yida, "UART vs I2C vs SPI - Communication Protocols and Uses," 2019. [Online]. Available: <https://www.seeedstudio.com/blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/>. [Accessed 20 5 2021].
- [15] TotalPhase, "Understanding the Difference Between UART vs SPI," 2016. [Online]. Available: <https://www.totalphase.com/blog/2016/06/spi-vs-uart-similarities-differences/>. [Accessed 20 5 2021].
- [16] Espressif Systems, "SPI Master Driver," 2021. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_master.html. [Accessed 20 5 2021].
- [17] J. Mallari, "How to use SPI communication on the arduino," 2020. [Online]. Available: <https://www.circuitbasics.com/how-to-set-up-spi-communication-for-arduino/>. [Accessed 20 5 2021].
- [18] Espressif Systems, "SPI Slave Driver," 2021. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/spi_slave.html. [Accessed 21 5 2021].
- [19] N. O'Leary, "Arduino Client for MQTT," 2020. [Online]. Available: <https://pubsubclient.knolleary.net/>. [Accessed 20 5 2021].
- [20] Wikipedia, "Serial Peripheral Interface," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Serial_Peripheral_Interface. [Accessed 19 5 2021].
- [21] React, "React," 2021. [Online]. Available: <https://reactjs.org/>.
- [22] D. Abramov, "Stack Overflow," 03 2017. [Online]. Available: <https://stackoverflow.com/questions/42438171/wheres-the-connection-between-index-html-and-index-js-in-a-create-react-app-app>. [Accessed 11 06 2021].

- [23] WebSolutions, "React Navbar Tutorial - Build a Responsive Navigation Bar | Animated Responsive Navbar React Tutorial," 03 Mars 2021. [Online]. Available: <https://www.youtube.com/watch?v=1fAvRKCjdc0>. [Accessed 11 06 2021].
- [24] C. Dev, "Create Dynamic Form Fields in React," 19 July 2020. [Online]. Available: https://www.youtube.com/watch?v=zgKH12s_95A. [Accessed 11 06 2021].
- [25] M. UI, "Text Field React component - Material-UI," n/a. [Online]. Available: <https://material-ui.com/components/text-fields/>. [Accessed 11 06 2021].
- [26] V. Anushka, "Difference between Fetch and Axios.js for making http requests," GeeksforGeeks, 31 August 2021. [Online]. Available: <https://www.geeksforgeeks.org/difference-between-fetch-and-axios-js-for-making-http-requests/#:~:text=Axios%3A%20Axios%20is%20a%20Javascript,the%20ability%20to%20cancel%20requests.> [Accessed 11 June 2021].
- [27] J. Lewis, "HTTP Requests Compared: Why Axios Is Better Than Node-Fetch (Automatic Transformations, More Secure, Can Handle Errors Better, Interceptor Support, And More Browser Friendly),," Medium, 19 July 2017. [Online]. Available: <https://medium.com/@jeffrey.allen.lewis/http-requests-compared-why-axios-is-better-than-node-fetch-more-secure-can-handle-errors-better-39fde869a4a6>. [Accessed 11 June 2021].
- [28] F. Kelhini, "How to make HTTP requests with Axios," Logrocket, 12 January 2021. [Online]. Available: <https://blog.logrocket.com/how-to-make-http-requests-like-a-pro-with-axios/>. [Accessed 12 June 2021].
- [29] J. Watmore, "React + Axios - HTTP POST Request Examples,," 17 July 2020. [Online]. Available: <https://jasonwatmore.com/post/2020/07/17/react-axios-http-post-request-examples>. [Accessed 11 June 2021].
- [30] M. UI, "Material Icons - Material-UI," n/a. [Online]. Available: <https://material-ui.com/components/material-icons/>. [Accessed 11 June 2021].
- [31] F. Awesome, "All Awesome Icons," n/a. [Online]. Available: <https://fontawesome.com/v5.15/icons?d=gallery&p=2&q=arrows>. [Accessed 11 June 2021].
- [32] Golang Programs, "onMouseDown and onMouseUp Event handling in ReactJs," 2021. [Online]. Available: <https://www.golangprograms.com/onmousedown-and-onmouseup-event-handling-in-reactjs.html>. [Accessed 28 5 2021].
- [33] Plus2Net Team, "Moving image across screen from left to right and top to bottom," 2000. [Online]. Available: https://www.plus2net.com/javascript_tutorial/image-move1.php. [Accessed 3 6 2021].
- [34] J. Carter, "4x4 Grid," 2018. [Online]. Available: https://www.researchgate.net/figure/4x4-grid-scenario-Maximum-PAI2-on-the-fixed-grid-is-4-1-2-of-crime-captured-divided-by_fig3_327057084. [Accessed 5 6 2021].
- [35] ESA, "The fractured features of Ladon basin," 2012. [Online]. Available: http://www.esa.int/Science_Exploration/Space_Science/Mars_Express/The_fractured_features_of_Ladon_basin. [Accessed 5 6 2021].
- [36] S. Eschweiler, "Getting Started With Axios," Medium, 7 Mars 2017. [Online]. Available: <https://medium.com/codingthesmartway-com-blog/getting-started-with-axios-166cb0035237>. [Accessed 12 June 2021].
- [37] whoisryosuke, "React Hooks - Track user mouse position - via: <https://twitter.com/JoshWComeau>," Github, 2018. [Online]. Available: <https://gist.github.com/whoisryosuke/99f23c9957d90e8cc3eb7689ffa5757c>. [Accessed 12 June 2021].
- [38] DUST, "Sci-Fi Short Film "Planet Unknown" | DUST," DUST, 19 January 2017. [Online]. Available: https://www.youtube.com/watch?v=B4JY_JeodKw&t=10s. [Accessed 12 June 2021].
- [39] T. Capan, "Why use Node.js," 2013. [Online]. Available: <https://www.toptal.com/nodejs/why-the-hell-would-i-use-node-js>. [Accessed 17 5 2021].
- [40] npm, "npm," 2021. [Online]. Available: <https://www.npmjs.com/>. [Accessed 17 5 2021].
- [41] OpenJS Foundation, "Express - Node.js Web Application," 2017. [Online]. Available: <https://expressjs.com/>. [Accessed 20 5 2021].
- [42] R. Wieruch, "How to create a REST API with Express.js in Node.js," 2020. [Online]. Available: <https://www.robinwieruch.de/node-express-server-rest-api>. [Accessed 20 5 2021].

- [43] R. Agarwal, "Why use ExpressJS over NodeJS for Server-Side Development?," 2014. [Online]. Available: <https://www.algoworks.com/blog/why-use-expressjs-over-nodejs-for-server-side-coding/>. [Accessed 20 5 2021].
- [44] M. Hossain, CORS in Action, Manning, 2014.
- [45] T. Goode, "cors - npm," 2018. [Online]. Available: <https://www.npmjs.com/package/cors>. [Accessed 25 5 2021].
- [46] J. Kasun, "Handling CORS with Node.js," 2020. [Online]. Available: <https://stackabuse.com/handling-cors-with-node-js>. [Accessed 25 5 2021].
- [47] A. Rudd, M. Collina, M. Agor and S. Buntsevich, "mqtt - npm," 2020. [Online]. Available: <https://www.npmjs.com/package/mqtt#handleMessage>. [Accessed 21 5 2021].
- [48] MongoDB, "The database for modern applications," 2021. [Online]. Available: <https://www.mongodb.com/>. [Accessed 4 6 2021].
- [49] MongoDB, "MongoClient or how to connect in a new and better way," 2021. [Online]. Available: <http://mongodb.github.io/node-mongodb-native/driver-articles/mongoclient.html>. [Accessed 8 6 2021].
- [50] MongoDB, "Atlas M0 (Free Tier), M2, and M5 Limitations," 2021. [Online]. Available: <https://docs.atlas.mongodb.com/reference/free-shared-limitations/>. [Accessed 8 6 2021].
- [51] A. Ludovici and P. Moreno, "Latency evolution according to the payload size," 2013. [Online]. Available: https://www.researchgate.net/figure/Latency-evolution-according-to-the-payload-size-The-performance-of-HTTP-increases_fig3_276039057. [Accessed 24 5 2021].
- [52] N. Harutunyan, "Graceful shutdown in NodeJS," 2021. [Online]. Available: <https://hackernoon.com/graceful-shutdown-in-nodejs-2f8f59d1c357>. [Accessed 6 6 2021].
- [53] Linuxize, "How To Use Linux Screen," 2021. [Online]. Available: <https://linuxize.com/post/how-to-use-linux-screen/>. [Accessed 28 5 2021].
- [54] A. Gimeno, "Node.js multithreading: What are Worker threads, and why do they matter?," 2019. [Online]. Available: <https://blog.logrocket.com/node-js-multithreading-what-are-worker-threads-and-why-do-they-matter-48ab102f8b10/>. [Accessed 8 6 2021].
- [55] nodejs, "C++ addons," 2021. [Online]. Available: <https://nodejs.org/api/addons.html>. [Accessed 8 6 2021].
- [56] E. Scully, "JavaScript vs C++: Differences and Similarities," 2019. [Online]. Available: <https://careerkarma.com/blog/javascript-vs-cplusplus/>. [Accessed 8 6 2021].
- [57] M. Cieřlar, "A complete guide to threads in Node.js," 2019. [Online]. Available: <https://blog.logrocket.com/a-complete-guide-to-threads-in-node-js-4fa3898fe74f/>. [Accessed 8 6 2021].
- [58] nodejs, "Node-API," 2021. [Online]. Available: <https://nodejs.org/api/n-api.html>. [Accessed 8 6 2021].
- [59] K. M. Farish, "How to Write Node.js Addons using C++ and N-API for Beginners," 2019. [Online]. Available: <https://morioh.com/p/e1a6f79af449>. [Accessed 9 6 2021].
- [60] Chromium, "node-gyp - Node.js native addon build tool," 2021. [Online]. Available: <https://github.com/nodejs/node-gyp>. [Accessed 9 6 2021].
- [61] N. D. Gobbo, "node-addon-api Setup," 2021. [Online]. Available: <https://github.com/nodejs/node-addon-api/blob/main/doc/setup.md>. [Accessed 10 6 2021].
- [62] G. Morrison, "Input lag: How important is it?," 2013. [Online]. Available: <https://www.cnet.com/news/input-lag-how-important-is-it/>. [Accessed 13 6 2021].
- [63] F. Ipar, "Monitoring MongoDB Response Time," 2016. [Online]. Available: <https://www.percona.com/blog/2016/02/26/monitoring-mongodb-response-time/>. [Accessed 10 6 2021].
- [64] W. Wu, "What OSI Layer does TLS Operate and Why?," 2020. [Online]. Available: <https://wentzhu.com/2020/08/21/what-osi-layer-does-tls-operate-and-why/>. [Accessed 13 6 2021].
- [65] J. Fruhlinger, "What is SSL, TLS? And how this encryption protocol works," 2019. [Online]. Available: <https://www.csoonline.com/article/3246212/what-is-ssl-tls-and-how-this-encryption-protocol-works.html>. [Accessed 23 5 2021].
- [66] Cloudflare, "What happens in a TLS handshake? | SSL handshake," 2021. [Online]. Available: <https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/>. [Accessed 24 5 2021].

- [67] Chromium, "Chrome Root Program," 2020. [Online]. Available: <http://www.chromium.org/Home/chromium-security/root-ca-policy>. [Accessed 26 5 2021].
- [68] Mozilla, "CA/Included Certificates," 2021. [Online]. Available: https://wiki.mozilla.org/CA/Included_Certificates. [Accessed 26 5 2021].
- [69] Internet Security Research Group, "Let's Encrypt," 2021. [Online]. Available: <https://letsencrypt.org/>. [Accessed 27 5 2021].
- [70] L. Community, "Which browsers and operating systems support Let's Encrypt," 2017. [Online]. Available: <https://community.letsencrypt.org/t/which-browsers-and-operating-systems-support-lets-encrypt/4394>. [Accessed 27 5 2021].
- [71] A. Rawdat, "Update: Using Free Let's Encrypt SSL/TLS Certificates with NGINX," 2021. [Online]. Available: <https://www.nginx.com/blog/using-free-ssl-tls-certificates-from-lets-encrypt-with-nginx/>. [Accessed 27 5 2021].
- [72] nodejs, "HTTPS | Node.js v16.30 documentation," 2021. [Online]. Available: <https://nodejs.org/api/https.html>. [Accessed 2 6 2021].
- [73] IEEE and The Open Group, "crontab," 2018. [Online]. Available: <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/crontab.html>. [Accessed 3 6 2021].
- [74] T. v. Eicken, "arduino-esp32/libraries/WifiClientSecure," 2021. [Online]. Available: <https://github.com/espressif/arduino-esp32/tree/master/libraries/WiFiClientSecure>. [Accessed 24 5 2021].
- [75] Tech It Yourself, "How to use Arduino ESP32 MQTTS with MQTTS Mosquitto broker (TLS/SSL)," 2017. [Online]. Available: <http://www.iotsharing.com/2017/08/how-to-use-esp32-mqtts-with-mqtts-mosquitto-broker-tls-ssl.html>. [Accessed 30 5 2021].
- [76] OpenSSL, "OpenSSL - Cryptography and SSL/TLS Toolkit," [Online]. Available: <https://www.openssl.org/>. [Accessed 27 5 2021].
- [77] J. Nguyen, "OpenSSL Certificate Authority," 2015. [Online]. Available: <https://jamielinux.com/docs/openssl-certificate-authority/index.html>. [Accessed 27 5 2021].
- [78] J.-P. Mens, 2020. [Online]. Available: <https://github.com/owntracks/tools/blob/master/TLS/generate-CA.sh>. [Accessed 28 5 2021].
- [79] notion, "SSL - node.js mqtt client using TLS," 2016. [Online]. Available: <https://stackoverflow.com/questions/40018804/node-js-mqtt-client-using-tls>. [Accessed 30 5 2021].
- [80] Arduino, "Serial - Ardunio Reference," 2021. [Online]. Available: <https://www.arduino.cc/reference/en/language/functions/communication/serial/>. [Accessed 30 5 2021].
- [81] Arduino, "Serial.readStringUntil() - Ardunio Reference," 2021. [Online]. Available: <https://www.arduino.cc/reference/en/language/functions/communication/serial/readstringuntil/>. [Accessed 30 5 2021].
- [82] MDN Contributors, "Working with JSON," 2021. [Online]. Available: <https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON>. [Accessed 2 6 2021].
- [83] MDN Contributors, "JSON.parse() - JavaScript," 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse. [Accessed 2 6 2021].
- [84] Intel, "Video and Image Processing Suite User Guide," 2021. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug_vip.pdf. [Accessed 18 5 2021].
- [85] R. Fisher, S. Perkins, A. Walker and E. Wolfart, "Digital Filters," 2003. [Online]. Available: <https://homepages.inf.ed.ac.uk/rbf/HIPR2/filtops.htm>. [Accessed 24 5 2021].
- [86] "Why do we use the HSV colour space so often in vision and image processing?," 2012. [Online]. Available: <https://dsp.stackexchange.com/questions/2687/why-do-we-use-the-hsv-colour-space-so-often-in-vision-and-image-processing>.
- [87] "Colour Conversions," [Online]. Available: https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html. [Accessed 15 05 2021].
- [88] W. Green, "Division in Verilog," 2021. [Online]. Available: <https://projectf.io/posts/division-in-verilog/#:~:text=FPGAs%20are%20different%3B%20Verilog%20can,cycles%20for%2032%2Dbit%20numbers..>

- [89] Wikipedia, "Salt-and-pepper noise," [Online]. Available: https://en.wikipedia.org/wiki/Salt-and-pepper_noise. [Accessed 24 05 2021].
- [90] Y. Wang, "Image Filtering: Noise Removal, Sharpening, Deblurring," 2006. [Online]. Available: https://eeweb.engineering.nyu.edu/~yao/EE3414/image_filtering.pdf. [Accessed 24 05 2021].
- [91] Wikipedia, "Gaussian Noise," 2021. [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_noise. [Accessed 25 05 2021].
- [92] Wikipedia, "Median Filter," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Median_filter. [Accessed 25 06 2021].
- [93] "Edge-Preserving Smoothing Filters," [Online]. Available: https://link.springer.com/content/pdf/10.1007%2F978-1-4471-6684-9_17.pdf. [Accessed 25 05 2021].
- [94] S. E. Lo, "Mode Filter," 2020. [Online]. Available: https://warwick.ac.uk/fac/sci/statistics/staff/research_students/ip/postphd/. [Accessed 26 05 2021].
- [95] R. Brewster, "Paint.NET," dotPDN LLC, 2021. [Online]. Available: <https://www.getpaint.net/>. [Accessed 23 5 2021].
- [96] A. Rosebrock, "Find distance from camera to object/marker using Python and OpenCV," 2015. [Online]. Available: <https://www.pyimagesearch.com/2015/01/19/find-distance-camera-objectmarker-using-python-opencv/>. [Accessed 27 05 2021].
- [97] FPGA4Fun, "SPI - A simple implementation," [Online]. Available: <https://www.fpga4fun.com/SPI2.html>. [Accessed 06 06 2021].
- [98] D. R. Williams, "Mars fact sheet," NASA, Greenbelt, 2020.
- [99] H. Abdi, "Energy Storage Systems," Razi University, Kermanshah, 2017.
- [100] R. K. Chauhan, "Coulomb counting method for SOC Estimation," 2020. [Online]. Available: <https://www.youtube.com/watch?v=svELMZ-7uqM>. [Accessed 26 5 2021].

6 Appendix

6.1 MQTT Custom Certificate

```

44  const char* ca_cert = \
45  "-----BEGIN CERTIFICATE-----\n" \
46  "MIIFtTCCA52gAwIBAgIUkdBx7TTn8jGu+U1mGwEtDoeXaLIwDQYJKoZIhvcNAQEN\n" \
47  "BQAwaJEXMBUGA1UEAwOQW4gTVFUVCBicm9rZXIxZjAUBgNVBAoMDU93bLRyYWNr\n" \
48  "cy5vcmcxFDASBgNVBAsMC2dlbmVyYXRLLUNBMSEwHwYJKoZIhvcNAQkBFhJub2Jv\n" \
49  "ZHLAZXhxbXBsZS5uZXQwHhcnMjEwNTI0MTgxNTA1WjBq\n" \
50  "MRcwFQYDVQDDA5BbiBNUVRUIGJyb2tldjEwMBQGA1UECgwNT3duVHJhY2tzLm9y\n" \
51  "ZzEUMBIGA1UECwwLZ2VuZXJhdGUtQ0ExITAfBgkqhkiG9w0BCQEWEm5vYm9keUBl\n" \
52  "eGFtcGxLLm5ldDCCAiIwDQYJKoZIhvcNAQEBBQADggIPADCCAgoCggIBALF78P+s\n" \
53  "oQxJa6my4J+43d2RqkGjUXpYj7jZ9dT+0t/Bh5JxdP4fJ83y0a+2V9vAeJaZjPJK\n" \
54  "HoI4yT/Jn29f4aMy7pUvMdzTWC9q5f5VLI9rpsfk72wSCR+GENEWMiWSDnXlybkt\n" \
55  "2Wf1hWBRlg3V0hBABc24Zr56K1NeUE9WDvEH8zg01a477TtEj7fST81xiq33cuSV\n" \
56  "r3LFbyskbC5cMqHg8bEYS0H9/loTP3ul9Ib5Zq90YZB6VB3It7R5BIq/oJbrYsw1\n" \
57  "UAQZ5H5JrjYBjV9SRYEgfxr/w2UJZATffzpvUSL3L8CJIK9uk70Q442vvwN78p+9\n" \
58  "1sgt3ZI9BEEY7jNUSKNdi4L2pt3PaeHnDunCabXV0MLMDi8qRoINX7rgHky0529z\n" \
59  "ccjtwwhyELnho7h1J95/d/w9u0HqYlTzx19R7GhNSZs/Kr3AgknKeAu8VwvuH/gtP\n" \
60  "+P4E4Bd5HhUsLvZvuq7xViFGLC3Ad8DCI8bnppE/cVAZMfiLy7EPPb91nMEGmvus\n" \
61  "4TgR4eGIHZZHqHkMVJSqK1STW09e2x3wQJ7veGpVsIKD4HcgjfqDoD4mjC+fu9pq7\n" \
62  "X4QuYdEziNGYwYqkMy00eAfUA09nCj02mrZ6PJc4kvWxKg6JdeLa+g/UNytEveAw\n" \
63  "FKh8FzM40fUT07EcsTGwUmnTD3dha5zvmX35AgMBAAGjUzBRMB0GA1UdDgQWBBTa\n" \
64  "Uan1Hgwwrgk0JLLFh6Nj89EBRzAfBgNVHSMEGDAWgBTaUan1Hgwwrgk0JLLFh6Nj\n" \
65  "89EBRzAPBgNVHRMBAf8EBTADAQH/MA0GCSqGSIb3DQEBAQUAA4ICAQCI7D/e+4uH\n" \
66  "F4KUqgJfXb0SPkZHCBAlyqQmFHaD+2ohGUhso/X+sSFkwL92XZ6HRfqgBSJrMBS2\n" \
67  "dZAlWBh6U/75ecQPUgLXQ5B5opgZtI7gXx034Gn4D7F6hbqYNNtTvt8D6Bk0CvKS\n" \
68  "fxFOorf0gh8EH2zteeNuIqcPML9BxU/7SSpbUxNhywsvXXzqwQ9xzHXuSX429QC\n" \
69  "8n3q0+2+il+n6er0SPh9/XLiaaXw9kU039e0dyeeawLHzMGwje9hj6ajd7wpNWa+\n" \
70  "f7GZGZHWZVmirc5MZggXMkL4fMIqfgxEsyZKTz7vfiegVS+xZE13PMtVBkCeKmHR\n" \
71  "rrpTgHxN70x1ifQ35giZDBAh2px4yzSDvVvqxoD6I3mh20oF6+pfcZhh0TWiFYBI\n" \
72  "BhHSgbTj90W3N1DLcZINx5uKwGtVyvA1eg0gi6a546cWfdeXC4vq2TinMBcjaWf/\n" \
73  "7U0sIo0YN0BJRBu8IgenbcVI9UnDSk0ryVeKzDieoor70LLY30wkfSSiCU4Z8Qkh\n" \
74  "8vPGucuGX2Gcoayp9yMfUWQ5aek5ZXj80pw0MVQFsI24z+TLu9RJ0/nP3bE3ADo\n" \
75  "Lhz27tN2lpVZiqrwtl5yaKZXpmnJgsam5lWho5B8BRj6Au7RGp6Bcipj0zhfI1e\n" \
76  "KKAH0SCVC/XiMCCdR/dFBKwUK0oEfibqXQ==\n" \
77  "-----END CERTIFICATE-----\n";

```

6.2 External Assets used in Web-App

Perseverance background picture on Home page: NASA, “Mars 2020 Perseverance Rover,” NASA, Mar-2020. [Online]. Available: <https://mars.nasa.gov/mars2020/>. [Accessed: 12-Jun-2021].

Circle of Remote Control: “Circle,” Wikipedia, 17-May-2021. [Online]. Available: <https://simple.wikipedia.org/wiki/Circle>. [Accessed: 12-Jun-2021].

Rover picture: real_name_hidden, *Live, draw, and enjoy on GetDrawings.com. Get free drawings, vector graphics and how to draw tutorials*, 06-Sep-2020. [Online]. Available: <http://getdrawings.com/get-icon#rover-icon-8.png>. [Accessed: 12-Jun-2021].

Pink circle: C. Denyer, “Pink Circle Free Stock Photo - Public Domain Pictures,” *Free Stock Photo - Public Domain Pictures*. [Online]. Available: <https://www.publicdomainpictures.net/en/view-image.php?image=301177&picture=pink-circle>. [Accessed: 12-Jun-2021].

Orange circle: “orange-circle-background,” *DishwasherHero*, 26-Jan-2020. [Online]. Available: <https://www.dishwasherhero.com/select-intro/orange-circle-background/>. [Accessed: 12-Jun-2021].

Blue circle: I. Петков, “File:Light Blue Circle.svg,” *Wikimedia Commons*, 16-Jun-2018. [Online]. Available: https://commons.wikimedia.org/wiki/File:Light_Blue_Circle.svg. [Accessed: 12-Jun-2021].

Green circle: John3, “obs transparent webcam - green circle overlay PNG image with transparent background png - Free PNG Images,” *TopPNG*, 08-Oct-2019. [Online]. Available: https://toppng.com/obs-transparent-webcam-green-circle-overlay-PNG-free-PNG-Images_233919. [Accessed: 17-Jun-2021].

Black circle: M. Pastor, “That Fucking Little Black Ball,” *Medium*, 07-Nov-2018. [Online]. Available: <https://medium.com/@mathiaspastor/that-fucking-little-black-ball-c98c7d642898>. [Accessed: 17-Jun-2021].

Dots for PFA: Iconspng.com, “mono point PNG icons,” *Free PNG and Icons Downloads*, 06-Apr-2018. [Online]. Available: <https://www.iconspng.com/image/25423/mono-point>. [Accessed: 17-Jun-2021].

Logo: “Logo Maker: Used By 2.3 Million Startups,” *Logo Design & Brand Identity Platform for Entrepreneurs*, 06-Jun-2021. [Online]. Available: <https://looka.com/editor/67326525>. [Accessed: 17-Jun-2021].

Reset and Start Button Styling: *CSS Buttons*. [Online]. Available: https://www.w3schools.com/css/css3_buttons.asp. [Accessed: 17-Jun-2021].