Imperial College London

SpaceX++

EEE/EIE 2nd Year Group Project Report

Sam Albert-Antwi (01707164), Ignacio Bricchi (01719233), Kacper Neumann (01731179),
Nicholas Pfaff (01709264), Benjamin Ramhorst (01715381), Bradley Stanley-Clamp (01545990)

Department of Electrical and Electronic Engineering
Imperial College London
27th October 2021

Contents

1 Abstract **3**

2 Introduction and Project Management **3**
2.1 Group Structure . . . 3
2.2 Project Requirements . . . 3
2.2.1 Command . . . 3
2.2.2 Control . . . 3
2.2.3 Drive . . . 4
2.2.4 Energy . . . 4
2.2.5 Vision . . . 4
2.2.6 Integration . . . 4
2.3 Timeline . . . 4
2.4 Meeting Minutes . . . 5
2.5 High Level Diagram . . . 5
2.6 Intellectual Property . . . 5

3 Command Module **6**
3.1 Design Process . . . 6
3.1.1 Start: High level approach . . . 6
3.1.2 Development . . . 6
3.1.3 Finalisation . . . 7
3.2 Implementation . . . 8
3.2.1 Website . . . 8
3.2.2 Server . . . 8
3.2.3 Database . . . 8
3.3 Security . . . 10
3.3.1 Transport Layer Security . . . 10
3.3.2 Username and Password Credentials . . . 10
3.3.3 Server Firewall . . . 10
3.3.4 DB Backups . . . 10
3.4 Navigation, Localisation, and Mapping . . . 10
3.4.1 Map . . . 11
3.4.2 Shortest Path Algorithm . . . 12
3.4.3 Path to Drive Instructions Conversion . . . 12
3.4.4 Automatic Path Recalculation . . . 12
3.4.5 Automation Algorithm (Automatic Next Destination Algorithm) . . . 12
3.5 Testing . . . 13
3.5.1 Unit Testing for Go Algorithms . . . 13
3.6 Future work . . . 14

4 Control Module **14**
4.1 Arduino IDE vs ESP-IDF . . . 14
4.2 Task-Based Programming . . . 15
4.3 Choosing of Wireless IP Communication Protocols . . . 15
4.4 Choosing of Wired Communication Protocols . . . 15
4.5 Server Communication . . . 16
4.5.1 Data Model and Data Encoding . . . 16
4.5.2 Security . . . 16
4.6 Drive Communication . . . 16
4.6.1 Data Model and Data Encoding . . . 17
4.7 Vision Communication . . . 18
4.7.1 Data Model and Data Encoding . . . 18
4.8 Energy Communication . . . 18
4.8.1 Data Model and Data Encoding . . . 18
4.8.2 Energy to Host Computer Communication . . . 18
4.9 Testing . . . 19
4.9.1 Simulation Tasks . . . 19
4.10 ESP32 Custom Memory Partition and Memory Utilization . . . 19
4.11 Future Work . . . 19

5 Drive Module **20**
5.1 Development . . . 20
5.2 Structure . . . 20
5.3 Instruction Set and Communication with ESP32 . . . 20
5.3.1 Instruction Set . . . 20
5.3.2 Communication with ESP32 via UART . . . 20
5.3.3 Executing an Instruction . . . 22
5.4 Movement . . . 22
5.4.1 Translational Motion . . . 22
5.4.2 Rotational Motion . . . 23
5.4.3 Distance Measurements . . . 23
5.5 Speed and Voltage Control . . . 24
5.5.1 Speed and Voltage Controller . . . 24
5.5.2 Time Measurements . . . 25
5.6 Reflection and Future Work . . . 25

6 Energy Module **25**
6.1 Development . . . 25
6.2 Cell Characterisation . . . 25
6.3 Measuring SOH and SOC . . . 27
6.3.1 SOC . . . 27
6.3.2 SOH . . . 28
6.4 Battery Pack Configuration and Cell Balancing . . . 29
6.5 PV Panel Characterisation . . . 30
6.6 Integration with Communication . . . 30
6.7 Future work . . . 30

7 Vision Module **30**
7.1 Basic Goals . . . 30
7.2 Structure . . . 30
7.3 Development . . . 30
7.4 Video Data Reader . . . 31
7.5 Color Detector . . . 31
7.5.1 Development Process . . . 31
7.5.2 Testing Methodology . . . 32
7.5.3 Verilog Implementation . . . 33
7.6 Blob detection . . . 33
7.6.1 Development Process . . . 34
7.6.2 Testing Methodology . . . 35
7.6.3 Verilog Implementation . . . 36
7.7 Communication with ESP . . . 37
7.8 Final Design . . . 38
7.9 Future Work . . . 38

8 Integration Module **38**
8.1 Role of Integration in Project . . . 38
8.2 Rover Modifications . . . 38
8.3 Testing Methodology . . . 39
8.4 Adding Energy to Rover and Future Work . . . 39

A Appendix: Meeting Minutes **44**
A.1 12th May . . . 44
A.2 17th May . . . 44
A.3 24th May . . . 45
A.4 31st May . . . 45
A.5 7th June . . . 46
A.6 10th June . . . 46
A.7 14th June . . . 46

1 Abstract

This report describes the client requirements, design process, methodology, and final implementation of a "Mars Rover" made as part of a second year end-of-year remote group project in the summer term of the 2020-21 academic year. This 4-week project involved members from the Electronic and Information Engineering (EIE) and Electrical and Electronic Engineering (EEE) degree streams with the objective of designing the software (except for the Energy module which involved hardware design) for an autonomous rover able to perform high-level commands sent to it from a remote control station based on a hardware implementation provided. The rover was broken up into six broad modules to aid the design process - Energy, Vision, Integration, Control, Drive, and Command - which were used to meet the requirements outlined in the brief (for details see Section 2.2).

2 Introduction and Project Management

2.1 Group Structure

The project's group was comprised of six members. The group members and their component's main responsibilities are listed below. However, this structure was taken with flexibility, due to the interconnected nature of the project, leading to members helping each other with their components from day one.

- **Bradley Stanley-Clamp** (EIE, Command) - design and implementation of the server, database, and website including the visuals on the front-end and the functionality on the back-end, as well as the communication between them. This is used to control the rover remotely.
- **Nicholas Pfaff** (EIE, Control) - implementation of communication protocols between different elements of the rover and the external server using an ESP32 mounted on the rover, recording the demo video.
- **Benjamin Ramhorst** (EEE, Drive) - design of the Arduino Nano Every code (in C++) on the SMPS responsible for controlling the motors' direction of spin and voltage applied.
- **Sam Albert-Antwi** (EEE, Energy) - design of a bank of batteries (charged using photovoltaic (PV) panels) able to supply the rover with sufficient power during operation as well as idling.
- **Ignacio Bricchi** (EIE, Vision) - design of an obstacle and coloured ball recognition system using the DE10-Lite FPGA and a D8M-GPIO camera, recording the demo video.
- **Kacper Neumann** (EEE, Integration) - performing any testing required by the group on the rover with individual or all components involved, organisation and minuting of meetings, recording and editing the demo video, designing integration of Energy into the finished rover.

2.2 Project Requirements

The client requirements set out in the project brief were summarised and broken up into smaller goals to make designing decisions easier. These were broken up into each module and are summarised below. They are all functional unless specified otherwise.

2.2.1 Command

- Sending basic movement commands to rover.
- Receiving and displaying data from the rover (battery level, position, heading, status etc.).
- Path finding algorithms for navigation.
- Interactive map for displaying recorded data and giving high level (final destination) commands.
- Algorithms for completely autonomous rover control and mapping.
- Ability to store data received by the rover in a database.
- Interface for exploring past missions via database look-ups (interactive maps, recorded images, etc.).
- Implement a level of security for the system to prevent cyber threats.
- (Non-functional) Neat and easy layout.
- (Non-functional) A current 2D map of the rover's path during a session.
- (Non-functional) Minimise algorithmic time complexity (Responsive rover control).

2.2.2 Control

- Receiving data from Drive, Command, Energy, and Vision.
- Sending data to Drive and Command.
- Obstacle avoidance based on input from Vision (Stopping decisions).
- Security for command internet communication.
- (Non-functional) Fast implementation - parallelism through tasked based programming.

2.2.3 Drive

- Moving the rover forwards and backwards for a certain distance
- Moving the rover forwards and backwards for a certain amount of time
- Turing right, i.e. clockwise by 90 degrees
- Turning left, i.e. anticlockwise by 90 degrees
- Accurately measuring the position of the rover relative to its origin
- Communicating with the ESP32 so that the desired motion is achieved
- Stopping abruptly if the rover "sees" an obstacle (e.g. ball) and moving back to its original point
- Speed and voltage control based on the current rover position and its target
- Writing modular, clean and maintainable code so it is easier to debug and integrate with other modules.

2.2.4 Energy

- Characterise the individual cells
- Extract individual cell capacity.
- Measure cell state of health (SOH).
- Measure cell state of charge (SOC).
- Balance cells in chosen configuration (parallel or series).
- Characterise PV panels and decide on configuration.
- Use the SMPS to step down/step up voltage of the PV panels and battery pack.
- Integrate energy module with communication sub-module to stream SOH, SOC and PV voltage to web app.

2.2.5 Vision

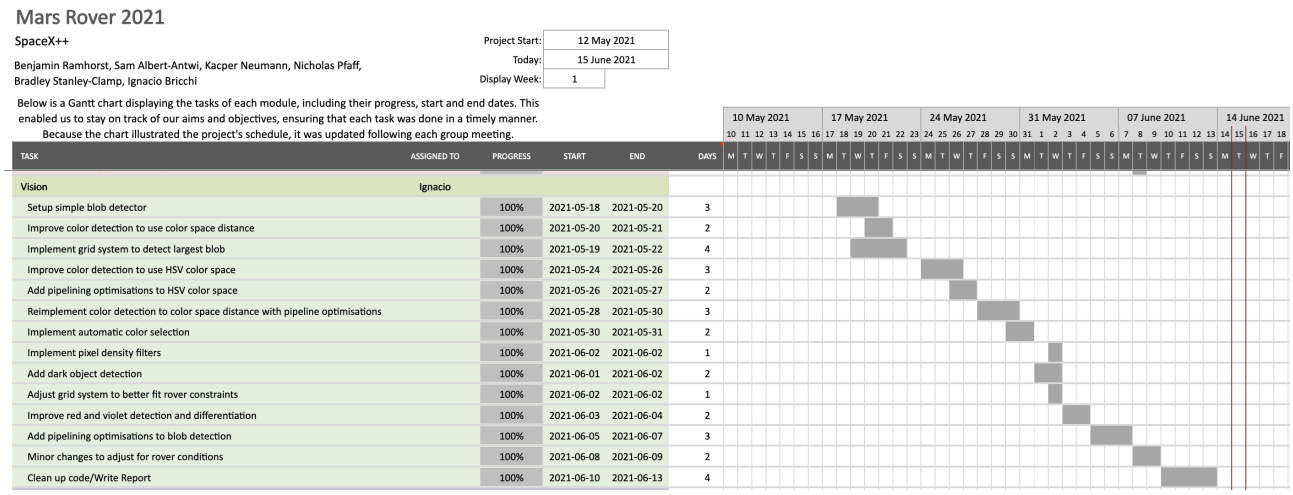
- Identification of obstructions and coloured balls in noisy backgrounds.
- Obstacle avoidance by relaying needed changes to Control.
- Identifying distance to ball.
- Identifying distance to general obstructions (e.g. dark cardboard).

2.2.6 Integration

- Smooth operation of all subsystems together.
- Build terrain map for testing (multiple scenarios).
- Method for cable holding that causes minimal interference with rover movement.
- Recording of overview video (accessible on https://youtu.be/T8w77XG1Lxw).

2.3 Timeline

The following fig. 1 shows a snippet of the project's Gantt chart (shown is only the Vision module). It enabled the team to stay on track of all their tasks and made project management easier. The full document was submitted together with this report. For a detailed description, please refer to said document.



Mars Rover 2021

SpaceX++

Benjamin Ramhorst, Sam Albert-Antwi, Kacper Neumann, Nicholas Pfaff, Bradley Stanley-Clamp, Ignacio Bricchi

Project Start: 12 May 2021
Today: 15 June 2021
Display Week: 1

Below is a Gantt chart displaying the tasks of each module, including their progress, start and end dates. This enabled us to stay on track of our aims and objectives, ensuring that each task was done in a timely manner. Because the chart illustrated the project's schedule, it was updated following each group meeting.

TASK	ASSIGNED TO	PROGRESS	START	END	DAYS
Vision	Ignacio				
Setup simple blob detector		100%	2021-05-18	2021-05-20	3
Improve color detection to use color space distance		100%	2021-05-20	2021-05-21	2
Implement grid system to detect largest blob		100%	2021-05-19	2021-05-22	4
Improve color detection to use HSV color space		100%	2021-05-24	2021-05-26	3
Add pipelining optimisations to HSV color space		100%	2021-05-26	2021-05-27	2
Reimplement color detection to color space distance with pipeline optimisations		100%	2021-05-28	2021-05-30	3
Implement automatic color selection		100%	2021-05-30	2021-05-31	2
Implement pixel density filters		100%	2021-06-02	2021-06-02	1
Add dark object detection		100%	2021-06-01	2021-06-02	2
Adjust grid system to better fit rover constraints		100%	2021-06-02	2021-06-02	1
Improve red and violet detection and differentiation		100%	2021-06-03	2021-06-04	2
Add pipelining optimisations to blob detection		100%	2021-06-05	2021-06-07	3
Minor changes to adjust for rover conditions		100%	2021-06-08	2021-06-09	2
Clean up code/Write Report		100%	2021-06-10	2021-06-13	4

Figure 1: A small snippet of the project's Gantt chart showing the Vision module

2.4 Meeting Minutes

Meetings were held at least weekly to check up on the team's progress and outline the next steps that need to be taken. These were scheduled more frequently if deemed necessary. Table 1 gives the dates of each formal meeting and the section number of the appendix where the relevant minutes can be found. Meetings that did not involve all group members were not minuted as these were often more technical than organisational. Minutes refer to group members by their initials.

Meeting Number	Meeting Date and Time	Appendix Section
1	10:00, 12th May	A.1
2	11:30, 17th May	A.2
3	11:30, 24th May	A.3
4	10:30, 31st May	A.4
5	16:30, 7th June	A.5
6	14:00, 10th June	A.6
7	16:00, 14th June	A.6

Table 1: Details of all formal meetings conducted during the project

2.5 High Level Diagram

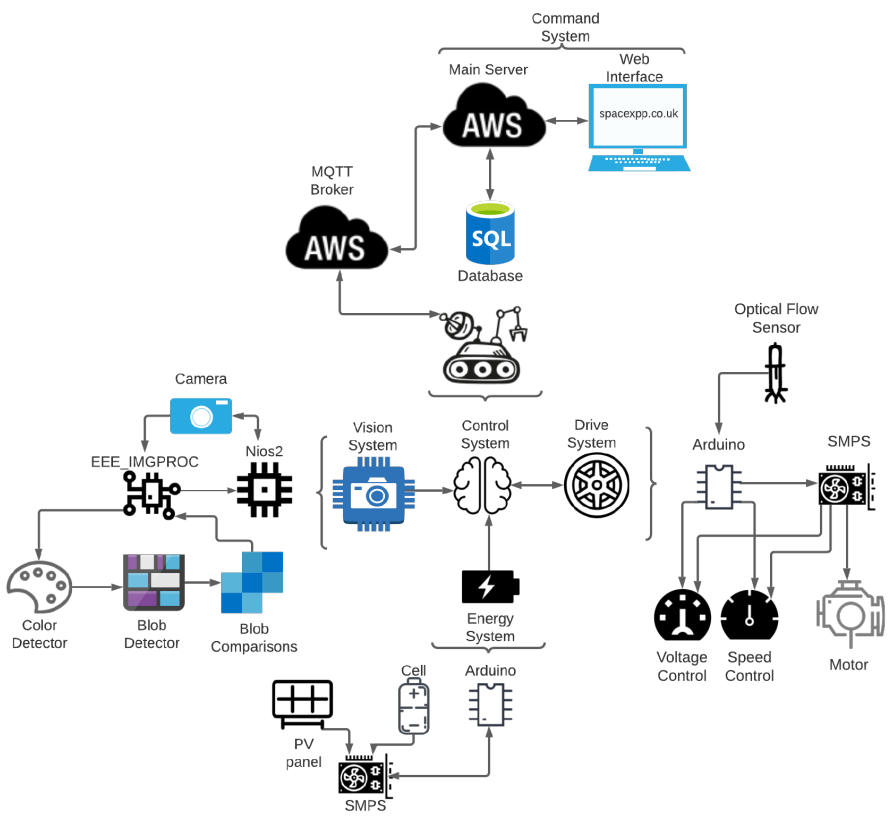


Figure 2: High level diagram of rover processes

2.6 Intellectual Property

As with most software, copyright is granted automatically at the time of writing the code. To obtain copyright, each team member identified themselves as the author of the given code and included the date of writing. While certain developers decide to make their code available under a public license (Creative Commons License, GNU General Public License), the SpaceX++ team decided this was not necessary; the code serves a specific purpose and is not intended as a reusable library or API for other developers, even if working on similar projects.

The Mars Rover 2021 cannot be granted design protection, as it was designed by the Electrical and Electronic Engineering Department, Imperial College London. Furthermore, the rover cannot be used to apply for a patent –

the first space rover was designed and built more than 50 years ago [1] and therefore does not represent an invention. Furthermore, none of the sub-modules comprising the rover are patent-worthy, as they make use of well-known algorithms and electrical engineering knowledge, such as switch mode power supplies, shortest path algorithms, web development, information processing and similar. If, however, the team found a novel way to improve the performance of a certain sub-module, the rover could be theoretically patented. Examples such as these include significantly improving the efficiency of switch mode power supplies or solar panels, breakthroughs in computer vision or improving graph algorithms the rover uses for exploration. However, it is important to note that such a breakthrough is unlikely in the relatively short period assigned for the project – 4 weeks.

Finally, the team could apply for trademark protection; however, this usually requires registering as a company, which, besides the rover and technical knowledge, requires funding, legal support as well as mass-production of the rover itself, if the product becomes viable. While achievable, it is unlikely to be done with the rover designed for this project as it has not been tested extensively and, as explained below, has various constraints depending on the environment it is operated in. Furthermore, the field of space exploration is ever-expanding and very competitive, making achieving a trademark even harder if that was to be pursued.

3 Command Module

3.1 Design Process

The overall aim of the command section was to create a platform (a website or mobile app) that provides the user with both control over the rover and information about the rover. In addition, as mentioned in section 2.2.1, the command system should meet a set of functional and non-functional requirements. This includes the ability to build a map of its terrain and store it in a database. It should also be able to provide algorithms that make the rover fully autonomous without the requirement for user input.

3.1.1 Start: High level approach

Initially, a high level approach was taken to get an understanding of where to begin. As can be seen in fig. 3, there will be a form of communication between the rover and a server. The server will have access to a database for data storage and it will also have communication to a front-end, either a website or a mobile app. The front-end should display data from the rover including an interactive map. It should also contain a command station to control the rover.

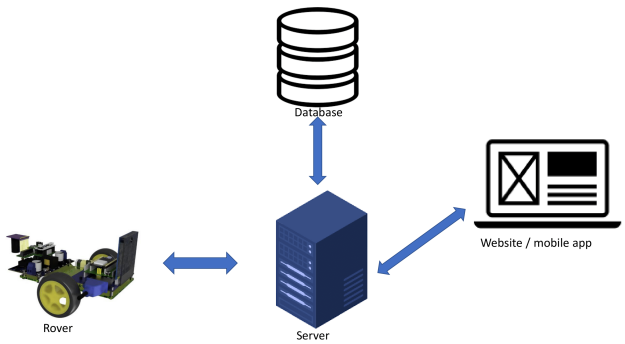


Figure 3: Diagram showing a high level design of the command subsystem

An important requirement is the design of the user interface. As there was no information about the client requirements, the design must be informative, simple and ergonomic to allow a user with varied knowledge to be able to understand and control the rover. The front-end is the only information input and output to the rover so it must provide as much information as possible without becoming overwhelming to understand.

The command system is the most vulnerable system to cyber threats as it will have some sort of communication over the Internet. Therefore, it is important to have a layer of security in accessing the website and server, as well as a level of security over the data being sent between the rover, website and server.

3.1.2 Development

The choice between an app and a website was heavily contested. To name a few considerations, an app with touch screen abilities could provide an advantageous interface with a more intuitive control, however, it would be less widely compatible. Similarly, it was not specified which device the client would be using, which could cause serious compatibility issues. On the other hand, if the website is built correctly it can be used on all modern devices. In addition, to make it widely available a mobile app requires extra steps of publishing which would have increased the task length.

Choosing a database starts with choosing the type of database. Some common examples include relational, hierarchical and network [2]. From a high level perspective, the data stored will be a map which will hold information at certain coordinates such as the location of the rover and obstacles [3]. Therefore, if a hierarchical database was used, each map could be the first children at level 1 and then each child of the map could be a specific coordinate that in turn has children describing the data at that specific coordinate. This design is good as it has a simple structure to implement, but as connections between children are not permitted and some coordinates will probably have the same data, it would be an inefficient method of storing a map. On the other hand, relational databases allow for a similar setup where there is one table that contains maps, another table that contains coordinates and a final table that contains type of data available at a given coordinate. These three tables could be linked via relations and accessed using joins [2]. This design is simpler and more efficient than the hierarchical design, with less data needed to be stored. Furthermore, relational databases can have a single entry edited without having to change the full body. In comparison, a hierarchical database would require the full database to be updated [2]. The next step is choosing the type of relational database. Firstly, the database must be free, therefore an open-source database such as MySQL and SQLite would work. The size of the database is also important. For this project the size of the data stored will be relatively low as a map can be represented efficiently and therefore will have low data storage requirements. In addition, the database will only be accessed by the server instead of multiple devices. These two factors mean the data should be stored locally on the server instead of using a cloud data storage system [4].

The design of the database is also an important consideration. Normalisation of the database will improve the design by further removing redundant data and anomalous queries, this would be done by splitting tables up and linking them via relationships [5].

Another aim of the command system is to control the rover. Firstly, the rover could be given low level commands via a remote control style interface; the same approach is taken when controlling an a RC (remote control) vehicle. This is good as it allows for maximum control over the rover's movements. However, unlike an RC vehicle, there will be an approximate 20 minute delay for any feedback assuming the rover is on Mars and has similar communication speeds and technologies to that of NASA's Mars Rover [6]. Similarly, the user will not have direct vision of the rover. Therefore, low level commands would be relatively useless as moving from one location to the other would take a long time due to delays of sending a single low level instruction then waiting to receive feedback and repeating. Another approach would be to give the rover a desired location and let it make its way there, adjusting to obstacles in its path along the way. A third approach would be where the rover is not fed any instructions but instead explores its terrain in an ordered and efficient manner, and when complete, returns a map and data of its environment. This is the most efficient approach as little communication is required between the rover and the user, however, all computation is done without the user's control, which could be problematic.

Additionally, the location of the heavy computation could be either on the server or on the rover. This would also determine the estimated location of the server. If the computation is on the rover, all data processing is local and communication to the server and website can be less frequent with less data being sent. This is beneficial as communication between the rover and the server would be expensive in terms of energy usage. In this case, it would be advantageous to locate the server on Earth as having a satellite orbiting Mars would be significantly more expensive and complex. Moreover, having the heavy computation on the server would free up computation 'space' on the rover. This is because the rover will already have a large amount of computation due to the drive, vision and control systems. Removing unnecessary computation would in turn lower the latency of the rover, particularly the control subsystem.

Regarding the map design, there were two viable options. The first approach was to program the map from scratch and directly design it to meet all functional and non-functional requirements. The second approach was to develop a preexisting open source design. Designing it from scratch allows for a more simple, specific and integrated map, however, this would be more time consuming and may be harder to develop and add extra functionality later on. In comparison, a preexisting design would allow for a more advanced map and command system to be created.

The first priority of the user interface is to identify what data should be presented on the page. Most importantly is the status section; this should provide an updated status on the connections to the server and the rover. This is vital as the user may be unaware of whether data they are seeing or trying to send is correct/working. Next, is data from the energy system, this would include state of charge, health, and error in cells. There should also be an up to date map and a terminal like section which provides general feedback on internal activity. There should also be a data input section to control the rover and access extra functionalities of the system. The implementation of this depends on the level of drive control. Similarly, the interface design must be clear and concise in terms of understanding what each component of the web-page does. To ensure the user's ease of use and to maintain simplicity, all components of the web-page should be visible at the same time without the need for scrolling.

3.1.3 Finalisation

For the front-end, a website was chosen due to its compatibility. The database was hosted on the general server using a local SQLite3 database. This is because it was lightweight and portable because the database was contained in a single file [7]. It was also easy to back up using software such as CronJob. The database aimed to be in Third Normal Form to improve efficiency of design [5]. A mid to high level set of controls was available on the website with all heavy computation occurring on the server. Finally, the map was developed from a preexisting open source design.

3.2 Implementation

3.2.1 Website

The website is made up of HTML, CSS, and JavaScript code. It has two pages, the main page contains the latest information from the rover, a live map, and controls, as seen below in fig. 4. The second page was used to load old data from previous missions, including the local terrain, the rover's final position and the set of instructions that were successfully accomplished. The website communicated to the server via HTTPS GET and POST requests.

We set up a skeleton structure between the website, server, rover and database, then developed each part individually. For the website, a very simple HTML local web-page was created with an input to simulate controls to the rover and an output to simulate data from the rover. Initially, HTTP requests were used due to its simplicity over HTTPS. To do this, a JavaScript file was used with a GET request for receiving rover instructions and a POST for sending rover commands. Once successful communication between the web-page and the server was established, the web-page was hosted using Amazon AWS S3, where features were added such as manual control and connection status as well as CSS to improve visual experience. Additional features were also added to the web-page such as a map and security which are mentioned in section 3.4.1 and section 3.3 respectively. The web-page was moved from Amazon AWS S3 to GitHub Pages as it had a much simpler updating process.

A second web-page was added to allow clients to view old mission data that had been saved to the database. Having this on a separate web-page maintained ease and simplicity of the website. Complicating the web-page would not only make the back-end of it complex but also would make the user experience less enjoyable as too much information would be displayed.

After testing it was noted that to reset the map the server would need to be restarted, which was inefficient and time consuming. Therefore, a reset button was added to reset the map and the rover's position on the map. A feedback terminal was also added to provide useful information that could not be displayed elsewhere on the page.

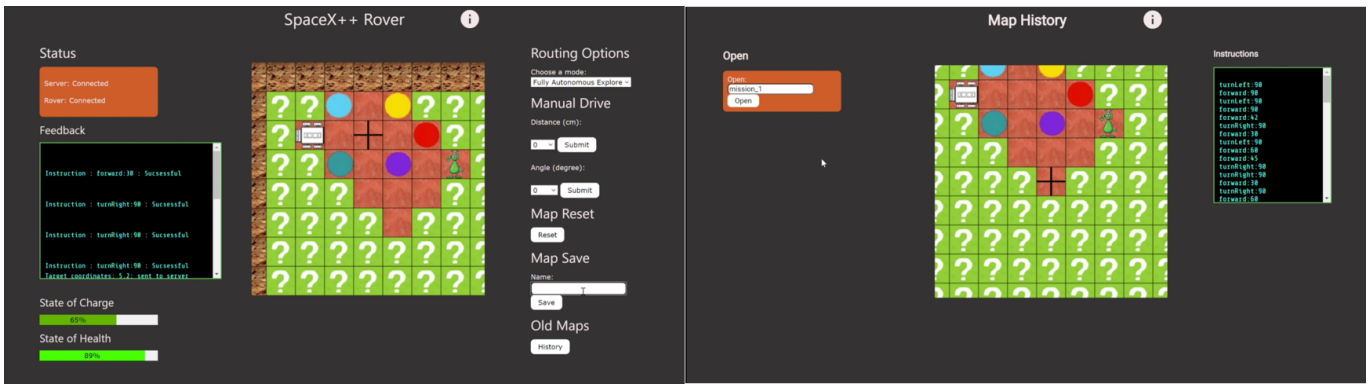


Figure 4: Diagram showing both pages of the website

3.2.2 Server

The server was hosted on an AWS Free Tier T2 Micro instance. It runs both an HTTPS API and a SQLite3 database. Due to its fast, efficient and simple design the server was written in Golang [8]. The HTTPS API was used to handle HTTPS GET and POST requests from the website and included communication to the ESP32 via MQTT protocol. It also does the heavy computation as described in section 3.4.

This project utilised skills learnt while developing the Year 2 Information Processing Project API, resulting in a sometimes similar coding style [9].

3.2.3 Database

The database can be accessed via the API on the server. The final design can be seen in fig. 5 overleaf. The map is represented as a single array of integers throughout the command system. Initially, we proposed to store the map as a single array of integers, however, this caused two issues. Firstly, SQLight3 did not support the storage of arrays. Secondly, this design did not support the development of the database as it would not be possible to store more information about individual map coordinates. Consequently, this would cause a bottle neck for future improvements. To overcome the first problem an option would be to create an encoding to store the data in a type that is available in SQLite3. For example, looping through all map coordinates and storing them as consecutive numbers in one long string. However, this did not help the second problem. Therefore, we used a separate table to store each individual tile and related it back to the map table using a foreign key for mapID. This allowed for extra data to be added to each tile as other areas of the rover were developed. The information for each tile for a given map was then able to be queried easily using SQL joins. Furthermore, the database is in Third Normal Form [5]:

- Each table cell contains a single value.

- Each record is unique, this is guaranteed from the use of primary keys.
- Data in tables can be uniquely identified from their primary keys and foreign key links.
- Has no transitive functional dependencies.

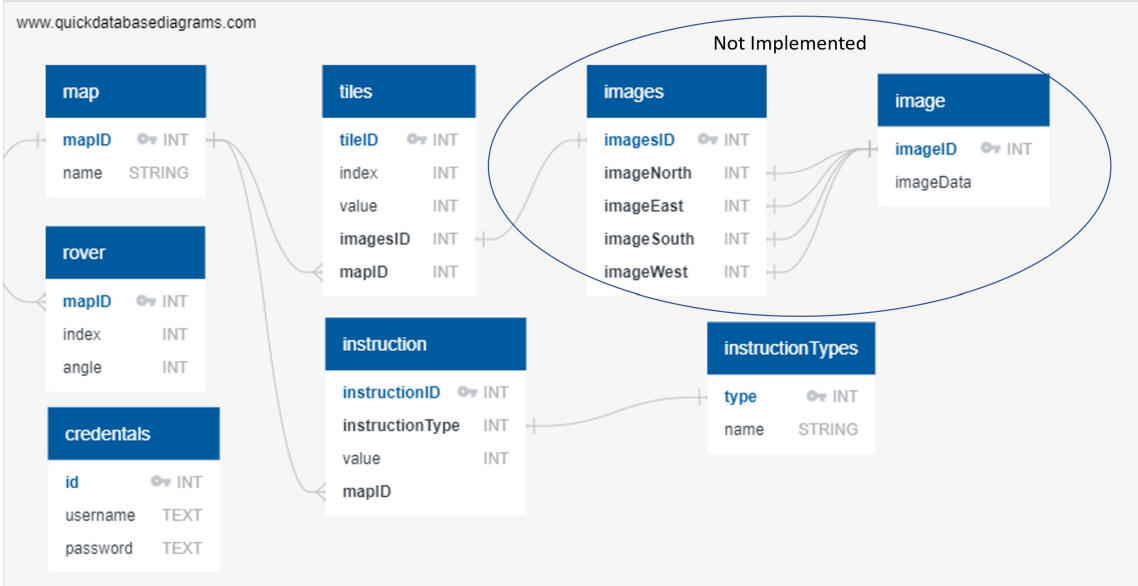


Figure 5: Diagram showing a high level design of the database

Note, although the database was designed with the extra functionality of images, they were not implemented into the system, this is discussed further in section 3.6.

Atomic database transactions were used to improve data consistency. Atomic transactions ensure that "either all occurs, or nothing occurs" [10]. Often, updating a database partially is worse than not updating the database at all. An example is inserting individual map tiles into the "tiles" table: We either want to insert all the tiles associated with a map or none of them when an error occurs. If an error occurred after half of the tiles had been inserted, a database containing only half of the tiles would be undesirable. Half of the map tiles are not enough for displaying a map and hence, this might cause additional errors later on when trying to load the current map. Or, it could cause errors when we re-attempt the operation but fail because there are now already tiles with the same name that we want to insert. These transactions were implemented by wrapping database operations inside the function shown in code 1 as the "f()" function and executing them using "tx.ExecContext()".

```go
func (s *SQLiteDB) TransactContext(ctx context.Context, f func(ctx context.Context, tx *sql.Tx) error) (err error) {
	tx, err := s.db.BeginTx(ctx, nil)
	if err != nil {
		return fmt.Errorf("server: SQLGeneral: failed to begin a transaction: %w", err)
	}

	defer func() { // Deferred anonymous function
		if err != nil { // Checking named return type
			if err := tx.Rollback(); err != nil {
				s.logger.Error("server: SQLGeneral: transaction rollback failed")
			}
			return
		}

		err = tx.Commit() // Changing value of named return type just before it is actually returned
	}()

	return f(ctx, tx) // Function used as a wrapper for database operations (e.g. multiple map tile insertions)
}
```

Code 1: Go Database Atomic Transaction Context

The function shown in code 1 combines Go's named return types and the "defer" keyword. Deferring the anonymous function shown causes it to be executed after the surrounding function, here "TransactContext()", returns. However, "returns" in this context means that the return value, here the named return value "err", has been evaluated but that it has not yet been returned to the caller of the function. This allows the deferred anonymous function to make use of the named return value by reading and modifying it. The final value returned by "TransactContext()" is the "err" value, checked and modified by the deferred anonymous function. "TransactContext()" starts by beginning a new transaction. This transaction is then passed to "f()", which is the function that is used, by the caller, for wrapping

the database operations. After executing "f()", the deferred function checks the returned error value. Something went wrong when the error is not "nil", resulting in a rollback of the transaction (Case of nothing occurring). Everything went fine when the error is "nil", resulting in the transaction being committed (Case of all occurring). Committing could fail. Hence, the returned "err" value is modified with the error returned by "tx.Commit()" and is left for the caller to handle.

3.3 Security

3.3.1 Transport Layer Security

Communication over the internet might be read or modified by a third party. For example, a third party could modify the drive instructions send from the web interface to the server. This might enable this third party to seize control over the rover. It is essential to protect against such attacks due to a Mars Rover being a high-value target. Consequently, we decided to encrypt the data flow from the web interface to the server, using TLS (Transport Layer Security). The process of achieving this mirrored the process used for encrypting the MQTT data flow. See section 4.5.2 for details.

3.3.2 Username and Password Credentials

To access the web interface, a user must first login with a username and password. These strings are immediately combined and stored as an encoded string in base-64. This is sent to the server on a partially public GET request. The request is partially public as it does not require an authorisation header to work, but the website it is sent from must have the correct authority certificate, as mentioned in section 4.5.2. In addition, it is necessary to encode the username and password in this format regardless of the TLS encryption as it is required for the HTTP authorisation header [11]. On the server side when the GET request is called, the received string is decoded and compared with credentials in the database. The server responds to the GET request with a boolean value, which is true when the credentials are valid and false otherwise. If valid, the website is loaded and all private HTTPS requests are given the correct authorisation encoding in their headers, using basic HTTP authentication format. This allows communication between the server and web interface [11]. If the credentials are not valid the login page reloads and the loop repeats.

Passwords are stored in encrypted form to protect against database breaches. bcrpyt is used as the password-hashing function. It was chosen as it automatically incorporates a salt when hashing passwords [12]. This is desirable as it reduces the risk of leaked passwords; a password hashes into a different string when the salt is different. Go's bcrypt implementation provides functionality for comparing plain-text passwords with bcrypt hashed passwords, allowing for user authentication.

HTTP middleware is used for authenticating "private" HTTPS requests. All API endpoints are "private", meaning that they must contain credentials, apart from the one used for checking user credentials for initial web interface logins. This authentication middleware is called before the API request is served, leading to a potential rejection of the request when the credentials are invalid.

3.3.3 Server Firewall

The server is an AWS Free Tier T2 Micro instance, therefore, regarding security, it comes with a large amount of control. To reduce the chance of the server being breached, all ports were closed apart from 3000 and 22 for inbound data. The HTTPS API is hosted on port 3000, and port 22 is needed to ssh onto the server. Port 8883 is the only port available for server output and this is used to communicate with the MQTT broker. To ssh onto the server, a key is required. This was kept secure and locally to reduce the risk of a leak.

3.3.4 DB Backups

Backups of the database are made daily using cron (time-based job scheduler). These backups are named by date and are compressed to save space. This ensures that valuable past mission data can be recovered, when the live database gets accidentally corrupted. Mars missions are expensive, which makes the data collected especially valuable.

3.4 Navigation, Localisation, and Mapping

A new mission starts with a completely unknown map, which contains the rover in its centre. This represents the rover just having landed on Mars, without knowledge about its environment. The rover's task is to explore this initially unknown terrain and produce a map that contains both empty space and different kinds of obstructions. Obstructions can be both coloured balls and unknown general obstructions. A mission is completed once all coloured balls have been identified and located on the map.

Navigation should guarantee the following: the map's centre should always be the rover's initial starting point, the rover should always be on a deterministic map location (centre of a map tile), map updates should be performed both during turning and moving forward, the map should be updated when the terrain has changed (e.g. when a new obstruction appeared), and it should be possible to navigate the map using different kinds of inputs: Manual drive, map destination selection, and fully autonomous. Manual drive is entering a single low level instruction such as turning right for a specified angle. Map destination selection is using the map cursor to select the rover's next destination,

without specifying how the rover should get there. Fully autonomous means that the rover explores the map and identifies all coloured balls without any user input (apart from starting this mode). The rover should automatically find a new path when its original path is no longer feasible due to new obstructions being discovered.

3.4.1 Map

The map is a grid-like structure that shows the type of terrain, location of the rover, and any obstacles identified by the rover. It has a cursor that allows the user to navigate around the map using the arrow keys and select locations they wish the rover to go to using the enter key. Starting with the source code from GitHub user belen-albeza [3], the map was developed to allow all functional and non functional requirements to be met. For example, only fixed shift movements of the cursor were allowed. This meant each square on the map could be a fixed size defined by the server in agreement with the Drive leader. Therefore, the rover could maintain accurate movements without losing vision on the square in front. In addition, extra controls were added to allow target coordinates to be sent to the server that would in turn be converted into drive instructions for the rover. Moreover, a cancel control that would stop the rover from undertaking any more drive instructions was added. Extra tiles were added to account for all obstacles that may occur as well as more relevant graphics for terrain. The already implemented collide functionality was developed to allow for more obstructions. A second layer of the map was used to represent the location of the rover.

To reduce the level of computation on the website, it would only store a cached set of data that included the map, the rover's location, and orientation. This was updated every 3 s from the server. This also meant that multiple users could view the map at the same time from different devices which is beneficial when working with a team as most space agencies do.

The map is represented as an array of integers where each index corresponds to a square on the map. The value of the integer determined the visual image displayed in that square, it also showed whether it was an obstacle or not. If it was an obstacle, the cursor would be unable to select that location as a target. When calculating optimum paths it would be identified as a an inaccessible location by the rover, and therefore the path would go around that obstacle.

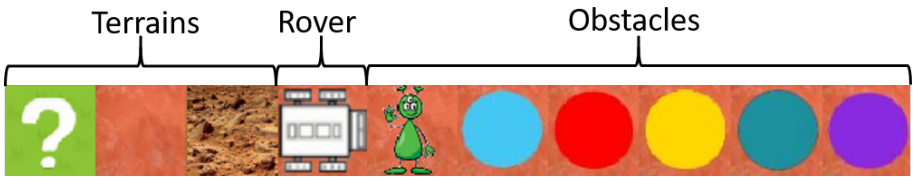


Figure 6: Diagram showing available images for the map tiles

The map is updated via a feedback path. The ESP32 sends drive instructions simultaneously to both the server and drive module. If the instruction is a drive instruction as defined in table 2, the instruction and value related to it are stashed in a global variable on the API. When the next instruction is sent, if it is another drive instruction or a finish instruction, the stashed instruction is executed on the map. This includes updating the path of the rover with the driven terrain instead of the question mark terrain, as seen in fig. 6, updating the rover's location, and orientation. The new instruction is then stored in the stashed instruction. If a finish instruction is executed on the map, no changes occur and a finish message is sent to the feedback terminal on the website. The one instruction delay caused by the stashing means that the obstacle avoidance information from the vision module can be sent to the server before executing the instruction on the map. This prevents incorrect data from being shown to the user. If the rover identifies an obstacle it will first send the stop instruction with a single character describing the obstacle identified. If the current stashed instruction is "forward", more information is needed to identify how far the rover can travel before identifying the obstacle. This information is provided on the next instruction sent by the rover "Stop Data". If the rover is turning and a stop instruction is received, the obstacle is not in the way of the rover but just identified whilst the rover is turning. Therefore, the obstacle is updated on the map and the full stashed turn instruction is executed using a "getOneInFront(changeInRotation)" function. This function identifies where the rover was facing and the location of one square in front of it.

Instruction Type	Instruction Name	Value Type	Encoding, where V is value
Drive	Forward	Distance in cm	F:V
Drive	Turn Left	Angle in degrees	L:V
Drive	Turn Right	Angle in degrees	R:V
Stop	Stop	Single character	S:V
Stop	Stop Data	Distance in cm	SD:V
Finish	Finish: no more instructions	N/A	X:V

Table 2: table showing the different types of instructions the server can receives from the rover

3.4.2 Shortest Path Algorithm

An algorithm for finding the shortest path on a tile map from start to destination, while considering obstructions, was required. This algorithm should be efficient to reduce the delay between a user selecting a command and the rover starting to move. A smaller delay means better user experience. In a Mars scenario, this delay would not be as important as it would be unnoticeable in comparison to the communication delay from Earth to Mars. Nevertheless, it was important for our Earth testing for this delay to be minimal.

Research yielded that a common, greedy graph algorithm, called A*, would be perfect for this task. The A* algorithm is an informed version of Dijkstra's algorithm: An additional heuristic function, which estimates the distance from each node to the destination node, was considered [13]. The A* algorithm is equal to Dijkstra's algorithm when this heuristic function is set to zero. The algorithm works by always visiting the node from the "open set" with the lowest "f score" next. The "open set" contains nodes that could be visited next. The "f score" is an estimated distance from the start node to the destination node and is the addition of the "g score" and the "heuristic function". The "g score" is the current minimum distance from the start node to the current node (used as the cost function is Dijkstra's) and the "heuristic function" estimates the shortest distance from the current node to the destination node.

The A* algorithm is optimal if its heuristic function is admissible [13]. It was decided to use the Manhattan distance as the heuristic function as it is both admissible and consistent. Admissible implies that "it never overestimates the cost of reaching the goal" [14]. Consistent implies that the "estimate is always less than or equal to the estimated distance from any neighbouring vertex to the goal, plus the cost of reaching that neighbour" [15]. This is the case as we are restricted to vertical and horizontal movements on the tile map. The heuristic being consistent implies that a path to a node is optimal once the node is removed from the "open set". This increases efficiency as nodes do not need to be re-added to the "open set".

A custom min-heap [16] was implemented and used to get fast access to the node with the lowest "f score", leading to an overall time complexity of $O(n \log(n))$, where n is the number of tiles in the map ($n = \text{cols} * \text{rows}$). The space complexity is $O(n)$, as all nodes are stored in memory.

3.4.3 Path to Drive Instructions Conversion

An algorithm was written to convert a path from the "Shortest Path Algorithm" into low-level drive instructions. The algorithm uses the following drive instructions: forward for a distance, turn left by 90°, and turn right by 90°. Multiple forward instructions are grouped into a single instruction if possible. Turning instructions are not grouped together as this allows for easier and more accurate map updates when turning.

There are three possible traversal modes: "Simple", "Full Discovery", and "Destination Discovery". "Simple" is following the path as is, while minimizing the number of drive instructions required to do so. "Full Discovery" is following the path, while inserting a turn left and turn right instruction after each single tile forward instruction (Forward instructions are not grouped). This maximizes the number of newly discovered tiles but also results in the largest number of instructions for a given path. "Destination Discovery" is like "Simple" but fully discovers the neighbours of the destination tile by inserting a turn left and a turn right instruction at the end.

3.4.4 Automatic Path Recalculation

In the beginning of every mission, all map tiles are "unknown". The "Shortest Path Algorithm" considers these "unknown" fields as empty tiles that can be traversed. Hence, it often occurs that a calculated path goes through some kind of obstruction that can not be traversed. Once such an obstruction is identified, the rover stops, the drive instruction buffer on the ESP32 is flushed, the map is updated with the obstruction, and the server calculates a new path to the desired destination and the corresponding drive instructions. This process repeats until the destination is reached. The rover stops and waits for further user input if obstructions make it impossible to reach the desired destination tile, for example when an obstruction is identified on the destination tile.

3.4.5 Automation Algorithm (Automatic Next Destination Algorithm)

The only remaining part of a fully autonomous rover was an algorithm for selecting the next destination coordinates, for whenever the rover arrived at its previous destination. This algorithm could then be called inside a loop until all balls are found or until the whole map is discovered. Multiple solutions for such an algorithm were considered. An optimal next destination would be in the middle of a field of "unknown" map tiles. This would allow the rover to discover as many "unknown" tiles as possible when being operated in full discovery mode. Hence, an optimal algorithm would produce a "worst-fit" style solution.

The first algorithm considered was a graph traversal algorithm for finding the biggest field of connected "unknown" map tiles: Two for loops would iterate over the map and call a depth-first traversal method on each not yet visited tile. The destination coordinates would be the centre of this field. This method works well for cases when all fields of "unknown" tiles have rectangular shapes. However, it works terribly when they have, for example an L-shape. Such an L-shape could result in the centre being on an already visited tile, resulting in the algorithm getting stuck. Moreover, choosing the destination on this L-shape would also be sub-optimal as there might not be any surrounding "unknown" fields, not fully utilizing full discovery mode.

This algorithm led to the conclusion that the biggest rectangular shape of "unknown" tiles, that could be fitted into the map, would be optimal for finding a desirable next destination. Such an algorithm was implemented using a linked list for storing "unfinished rectangles" and an array for storing "finished rectangles". Rectangles were identified and completed by scanning the map row by row and keeping track of a current "unfinished rectangle". All "unfinished rectangles" were iterated through while a row was traversed. This allowed for finding all possible rectangles while visiting every map tile only once. A version of this algorithm called "getUndiscoveredRectangles" still exists. However, this algorithm currently only works for non-touching rectangles and hence is not very useful for its desired purpose. It was attempted to extend it for touching rectangles, but this is not yet finished and would require more time for dealing with all possible edge cases.

It was decided to choose a simpler algorithm and go back to the rectangle algorithm when the simpler algorithm turned out to be too inefficient. The final "simple" algorithm works by iterating over the map and choosing the first destination that has four undiscovered neighbors (allowing all to be discovered using full discovery or destination discovery mode). If no such destination exists, the algorithm attempts to find one with three undiscovered neighbors and so on.

3.5 Testing

For all components of the command system from the HTTP API to small functions with simple methods, similar approaches were taken and the general structure of this testing can be seen in fig. 7. To begin with, research on the component was done - this was to find information on different methods. The aim was to prevent jumping straight into coding and wasting time producing inefficient code. Next, designing the code component was a quick procedure often comprising of drawing diagrams or writing pseudocode to help with the thought process. The research and design steps were often logged in a OneNote document and occasionally discussed with other group members.

The skeletal structure was then created. This involved creating the necessary structures for the code such as defining functions, HTTPS addresses and structs. However, they were left with minimal functionality giving only necessary code to allow the structures to be tested together. Testing consisted of compiling followed by testing via manual input. If testing was unsuccessful, code was evaluated and errors were identified. If there was a more significant issue with the design of the skeletal structure, redesigning of the implementation was necessary. Once there were no issues with the code, the next step was to add on functionality.

Adding functionality comprised of building up the skeletal structure with computational code. This helped reach the aims of the component. This was then tested separately from the overall system where possible. Testing was a series of manual inputs of main and edge cases. If unsuccessful, code was evaluated and fixes were made.

The final step was to implement the component into the full system. It was then tested using the testing script mentioned in section 4.9.1, which provided a large random assortment of tests that would test full functionality of the command system.

Keeping these three coding steps separate significantly reduced debugging time, as it was much easier to identify the location of where errors could occur.

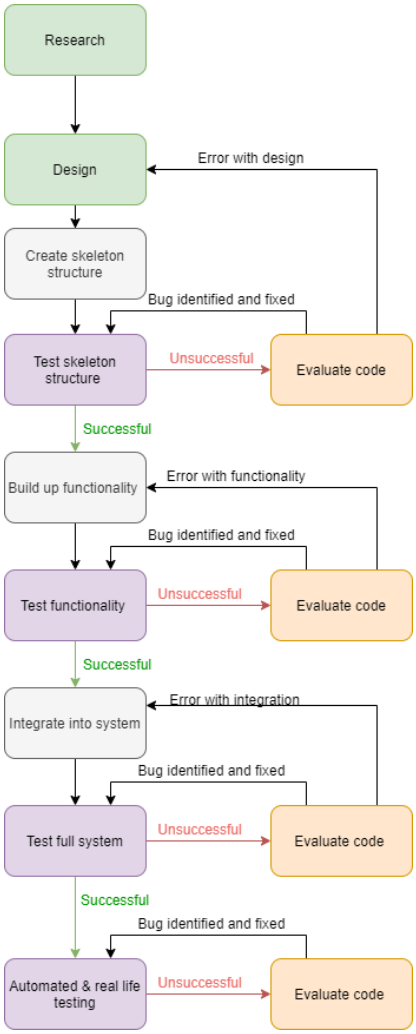


Figure 7: Flowchart of the testing and design process of the command system

3.5.1 Unit Testing for Go Algorithms

Unit tests were written for all more complex Go algorithms. Go has a built-in unit test framework which can be called using "go test -v". Unit tests made it easy to trace errors to specific test cases and hence identify where to look for the mistake. Moreover, they allowed quickly checking if a modification to an algorithm broke existing functionality. It was attempted to write extensive test cases before starting to implement an algorithm. This allowed for fixing the requirements and identifying edge cases at the start. They could then be run during the development process to help tracking progress.

An example of such a test case is shown in code 2. This test case is taken from "TestPathToDriveInstructions" and enabled us to identify an initial error that resulted in small paths not being converted correctly into drive instructions.

```
{
        path: [][]int{
                {0, 1},
                {0, 2},
        },
        tileWidth:        10,
        initialDirection: north,
        traverseMode:     simple,
        expectedInstructions: []driveInstruction{
                {
                        Instruction: "turnRight",
                        Value:       90,
                },
                {
                        Instruction: "forward",
                        Value:       10,
                },
        },
},
```

Code 2: Go Test Case Example

3.6 Future work

Further work that could be done on the command system would be adding the functionality of images. This would include implementing the already designed database functionality for images shown in fig. 5. In addition, adding functionality to the map would allow us to show if there are images available for a specific tile and from which angle images were taken from. Moreover, adding an extra control, such as a mouse click, would allow the user to open the images of the tile the mouse is currently selecting. Research and implementation of image data transfer would also be necessary in future work to identify how the web-page could access images via HTTPS protocol as well as how the server could obtain the image data from the rover. This is also mentioned in section 4.11.

4 Control Module

Figure 8 shows the communication protocols that connect different components together as well as the direction of data flow. Moreover, some example data packets with their data encoding are shown in orange. The actual energy to host computer to MQTT broker connection has been omitted from the diagram.

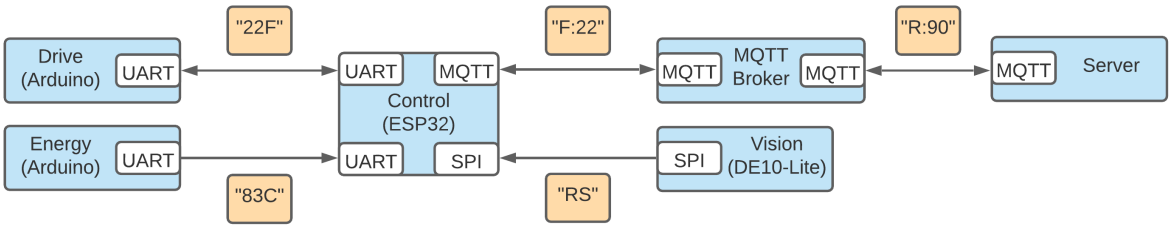


Figure 8: High-level Communication Flow Design

4.1 Arduino IDE vs ESP-IDF

The Arduino IDE can be used to program the ESP32 with Arduino C++ code. The benefit of using the Arduino IDE is, that it is easier to get started as there are tons of Arduino tutorials available for free in the internet. Moreover, the Arduino libraries are much simpler to understand than the more extensive native ESP-IDF libraries. However, using the ESP-IDF has numerous benefits: it gives more control over the ESP32's peripherals and functionalities, the libraries are more extensive than the Arduino ones, and it runs FreeRTOS by default which makes things like task-based multi-core programming easier [17]. Using these considerations, it was decided to program the ESP32 using the ESP-IDF with the Visual Studio Code PlatformIO extension. This decision turned out to be a good idea. The learning curve of the ESP-IDF was steep as expected, however, after getting used to the ESP-IDF documentation and FreeRTOS, programming the ESP32 became much easier. An extensive book about the ESP32 from Neil Kolban [18] helped greatly in this process. It would not have been possible to achieve what has been achieved without using the ESP-IDF (e.g. custom memory partitions, task parallelism, fast SPI slave implementation, etc.).

4.2 Task-Based Programming

The ESP32 has two identical cores that share the same memory. It was identified early in the design stage, that using these two cores in parallel could significantly speed up the ESP32's responsiveness. Research yielded that it is common practice to build task-based applications with FreeRTOS [19]. Each task is an infinite loop that is registered using "xTaskCreate" and a priority. Messages can be passed between tasks (e.g. using FreeRTOS queues). FreeRTOS then schedules these tasks automatically. The two cores allow two tasks to be run in parallel. Tasks are paused when they are not used (e.g. when waiting for a new message over UART from the Drive). FreeRTOS sees these paused tasks and switches to a task that is not currently active and has the highest priority.

The final implementation resulted in three (four) tasks: one for Drive communication, one for Vision communication, and one for Energy communication. Server communication works through an "event loop", which is just another task under the hood. This means that two out of four tasks can run at any one time. It was decided that Vision should have the highest priority and Energy the lowest priority. Hence, FreeRTOS always tries to run the Vision task. This is important as the rover must react rapidly to "stop" commands from Vision to not crash into obstructions. Latency in the Energy task is not too important as it is mainly responsible for occasional battery status updates. Moreover, in practice Energy was integrated via a host computer and hence the Energy task can be commented out for the current system's testing, resulting in three tasks remaining.

4.3 Choosing of Wireless IP Communication Protocols

Multiple wireless IP-based communication protocols for communicating with the server were researched and considered. They include raw TCP, WebSocket, ESP32 as HTTP client, ESP32 as HTTP server, and MQTT. Only connection oriented protocols were considered as data reliability and ordering were deemed important. Ideally, a full-duplex communication between ESP32 and server was desired; the ESP32 needs to send status updates when they are ready and the server needs to send commands when they are available from user input. It was decided that continuous HTTP GET and POST requests, that would be necessary due to HTTP being (generally) half-duplex, would result in too much overhead. A continuously open raw TCP connection would result in the least overhead possible and hence in the lowest latency possible. However, it was excluded due to the difficulty and development time required for dealing with low-level TCP connections and adding data encryption. This left the competition between WebSocket and MQTT. WebSocket has generally lower latency than MQTT [20], as is expected due to MQTT's additional "broker", however, the latency difference is single digit milliseconds. Furthermore, WebSocket provides a real full-duplex channel while MQTT's broker architecture results in two clients never being directly connected. However, MQTT was specifically designed for battery constraint IoT devices and for multiple clients [21]. This would make adding additional clients such as a separate battery charging station or a Mars weather station trivial. Overall, WebSocket was deemed more appropriate if rover control should be low-level and happen in real-time, requiring low-latency. MQTT is better for adding more clients in the future and for controlling the rover using more high-level commands. Therefore, it was decided to use MQTT for communication between ESP32 and the server. The plan was to control the rover using high-level commands and Energy would potentially need to be added as a separate client.

4.4 Choosing of Wired Communication Protocols

Wired communication was needed for communication between Control and Drive, Vision, and Energy. A wired communication protocol might work well for one of these communications but not for another. Three wired communication protocols were considered: I2C, SPI, and UART [22], [23], [24]. The main relevant differences are summarised in table 3. Throughput's "About" refers to common upper limits.

	UART	SPI	I2C
Throughput	Device dependent. About 115 kbps to 460 kbps.	Device dependent. About 10 Mbps to 80 Mbps.	Generally 3.4 Mbps in high-speed mode.
Number of Pins	2	4 (More if multiple slaves)	2
Communication Type	Asynchronous	Synchronous	Synchronous
Full/Half-Duplex	Full-Duplex	Full-Duplex (Master must start communication)	Half-Duplex

Table 3: Wired Communication Protocol Comparison

Additionally, UART is the simplest one to setup for communication with the ESP32 (Based on initial proof-of-concept setups). Overall, UART seemed like the best solution for simple full-duplex communication, when throughput is less important. This was deemed to be the case most of the time as the majority of the data would be in the form of small instructions and status updates (all of these ended up being 1-2 bytes). The only exception to these small data packets are images; a possible extension of displaying obstruction/ball images on the webpage was considered (see section 4.11 for more detail). This would require an image to be transferred from the FPGA to the ESP32 and then to the server, whenever an obstruction is identified. This image transfer would need to be fast enough to not block further "STOP" instructions from Vision, leading to potential rover crashes. Images are $640 * 480 = 307200$ pixels in

size. Each pixel has three colours with 8 bits each, resulting in 24 bits per pixel. Overall, this results in an image being about 7.4 Mbs. Even when neglecting ESP32 and FPGA memory issues and the time taken to start sending the image from the FPGA, transferring such an image would take too long with UART. Too long means that the rover would need to be stopped during an image transfer as blocking the Vision to ESP32 communication for multiple seconds while driving is too risky. This issue remains even after significant image compression before transmission. The only considered communication protocol that could deal with transferring such images rapidly is SPI. Hence, it was decided to use UART for both Drive and Energy communication and SPI for Vision communication. I2C is great for many devices communicating with each other using few pins, however, a lack of pins is not an issue in this project. Moreover, UART is simpler to implement when compared with I2C due to the absence of addressing and it being full-duplex. Hence, I2C was deemed to yield no benefit for the kind of communication required.

4.5 Server Communication

As mentioned in section 4.3, MQTT was chosen for communicating with the server. Mosquitto running on an AWS EC2 instance was chosen as the MQTT broker. Mosquitto is a commonly used and free open source MQTT broker.

The ESP32 subscribes to the topic "/drive/instruction" for receiving drive instructions from the server: a sequence of drive instructions is published by the server one by one. The ESP32 then stores these instructions in a FreeRTOS FIFO queue (see section 4.6 for more detail). MQTT QoS (quality of service) 2 was chosen for this communication; it is very important that drive instructions arrive in order and without errors. Latency is less important for this communication.

The ESP32 publishes, and the server subscribes to, the topics "/feedback/instruction" and "/energy/status". The topic "/feedback/instruction" is used for sending drive instructions to the server, at the same time that they are sent to the drive. This allows the server to keep track of the rover's current map position and hence update the map accordingly, without having to wait until the rover finished the whole instruction sequence (see section 3 for more details). QoS 2 was chosen for "/feedback/instruction" for the same reasons that it was chosen for "/drive/instruction". The topic "/energy/status" is used for sending battery status updates to the server. QoS 0 was chosen for "/energy/status" as data integrity/ordering is not too important for these status updates and hence it is preferred to minimize additional network traffic.

4.5.1 Data Model and Data Encoding

It was decided to use a very lightweight data encoding for wired communication with Drive, Vision, and Energy in order to minimize decoding time on the microprocessors (see sections 4.6, 4.7 and 4.8 for more detail). However, a more verbose data encoding was chosen for communication between ESP32 and the server. Both the AWS server and ESP32 are powerful when compared to the Arduino Nano Every and Nios 2 processor and hence can deal with slightly harder to decode data formats without causing delays, hence readability and maintainability of code was deemed more important than efficiency in this case. The ESP32-server data encoding follows the following format: "Meaning:Value". For example, a drive instruction for moving 22 cm forward would be encoded as "F:22".

4.5.2 Security

There is no need to encrypt local wired communication: no one from the outside is able to access these transmissions, however, wireless communication over the internet might be read or modified by a third party. For example, a third party could intercept wireless MQTT communication and change a message from "R:90" to "F:300". This might result in the server's map becoming invalid, leading to the next paths becoming invalid. A Mars Rover is a high-value target and hence it is essential to protect against such attacks by using encryption and authorisation.

Consequently, it was decided to use MQTT over TLS (Transport Layer Security) instead of TCP (default). This change involved creating a key and certificate for the MQTT broker. OpenSSH was used for creating X.509 (certificate standard) and SHA-256 (Cryptographic Hash Algorithm) certificates. Moreover, the broker certificate had to be signed by a certificate authority. It was decided to self-sign the certificate by first creating a certificate for the "SpaceXpp CA" certificate authority. This certificate authority certificate could then be used on both the ESP32 and the server for authentication and encryption.

A third party that knows the MQTT topics used and manages to steal the self-signed certificate authority certificate could send additional messages to these topics. This again could lead to the map becoming invalid. Hence, it was decided to add two-factor authentication by additionally requiring a username and password. A certificate is something that you have while a password is something that you know. It is harder to steal passwords than certificates as passwords are usually stored in encrypted (hashed and salted) form.

4.6 Drive Communication

As mentioned in section 4.4, UART was chosen for the communication between ESP32 and Drive. The absence of a master-slave architecture for UART proved especially useful as both ESP32 and Drive must be able to start communication, and continuous polling (sampling status of external device) would take up unnecessary processing power. A baud rate of 115200 was chosen. A significantly higher baud rate could have been achieved [25], [19],

however, 115200 is fast enough for the small, infrequent data packets that are used and is slow enough to not cause any problems (a standard Arduino baud rate).

The ESP32 stores a sequence of received drive instructions in a FreeRTOS FIFO queue. This queue is used both for passing the instructions from one concurrent task to another (MQTT event loop to drive UART task) and for feeding the instructions one at a time to the Drive (first in first out property of a FIFO queue). The Drive's Arduino sends a "ready" signal whenever it is ready to receive another instruction. This occurs when the previous instruction is completed. The ESP32 sends the next instruction to the Arduino if a "ready" signal has been received and the instruction queue is not empty. This "ready" communication acts as a flow-control mechanism to prevent the Arduino from running out of memory when the instruction sequence is big. Also, it prevents the Arduino from having to process new incoming instructions and execute the current instruction at the same time. This is good as the Arduino Nano Every has only a single core. Moreover, this makes it easy for the ESP32/server to keep track of the current instruction and hence update the map incrementally (no additional feedback from Drive is necessary). Next to drive instructions, the ESP32 can also send a "stop" instruction to the Drive's Arduino. This "stop" instruction means that the Drive must stop instantly to avoid colliding with a newly identified obstruction. Hence, the ESP32 does not wait for a "ready" signal before sending "stop". After stopping, the Arduino sends the distance travelled forward before stopping to the ESP32. This is necessary as the rover must move backwards into the centre of the closest square before continuing. The ESP32 uses the map tile width, the stop distance received from Drive, and the total forward distance of the previous instruction to calculate the backward distance. This backward distance is then send to the Drive.

4.6.1 Data Model and Data Encoding

As mentioned in section 4.5.1, a lightweight data encoding was designed for wired Drive communication. The ESP32-Drive data encoding follows the following format: "ValueInstruction". Value is either nothing or a one byte integer. Instruction is a single uppercase character. For example, a drive instruction for moving 22 cm forward would be encoded as "22F". The benefit of this encoding is, that the instruction can be easily popped of the received string without having to iterate over the entire string. Hence, obtaining the instruction from this data encoding is a constant time O(1) operation as can be seen in code 3. This single character instruction can then be used efficiently inside a switch statement.

```
// Read instruction for use in a switch statement
const char instruction = rx_data[strlen((char*)rx_data)-1];
// Remove instruction from data, leaving a value that can be cast into an integer
rx_data[strlen((char*)rx_data)-1] = '\0';
```

Code 3: Instruction Decoding

As mentioned in section 4.5.1, the downside of this encoding is the resulting lack of readability and maintainability of code. Hence, a struct type serving a similar purpose to an enumerated type was implemented as can be seen in code 4. This struct type shows two drive instructions that have two characters instead of one character, appearing to violate the drive data encoding. However, these two instructions are never send to the Drive but represent more specific cases of the "stop" instruction. This is used by the ESP32 to keep track of why it decided to stop and hence determine whether moving backwards is necessary or not. For example, it is not necessary to move backwards if it stopped while turning, with the next instruction being forward towards the obstruction, and hence the rover is still in the middle of a map tile. See section 4.7 for more detail.

```
// In header file:
typedef struct DriveEncoding{
    const char* forward;
    const char* backward;
    const char* turnRight;
    const char* turnLeft;
    const char* stop;
    const char* stopFromForward; // Only used internally by ESP32
    const char* stopFromTurn;    // Only used internally by ESP32
} DriveEncoding;
// In source file:
const DriveEncoding driveEncoding = {"F", "B", "R", "L", "S", "SF", "ST"};
// Usage example:
currentDriveInstruction = driveEncoding.stopFromForward;
```

Code 4: Drive Instruction Encoding Struct

Testing revealed that the Arduino Serial library sometimes struggles with reading multi-byte data correctly. Hence, it was decided to add extra redundancy to the data sent from the ESP32 to the Drive's Arduino. A start

marker '<' and an end marker '>' were added to the data packets sent. An example data packet would be "<22F>". This idea was inspired by user Robin2 [26]. The Arduino reads from its UART buffer one byte at a time. It switches into a receiving state once it finds the start marker '<' and saves all following bytes into a received buffer, until it finds the end marker '>'. Finding the end marker results in the received buffer being null terminated and processed as a received instruction string. All data that is not encapsulated by start and end markers is ignored. The ESP32 is better at receiving multi-byte data and hence does not need this redundancy.

4.7 Vision Communication

As mentioned in section 4.4, SPI was chosen for the communication between ESP32 and Vision. The master-slave architecture for SPI worked well as data flows only from Vision to ESP32 and not the other way around, hence it was decided to configure the vision DE10-Lite FPGA as the SPI master and the ESP32 as the SPI slave. This means that Vision can send a "stop" instruction without having to wait for the ESP32 to start a transmission. Configuring the ESP32 as the SPI master would require the ESP32 to start SPI transmissions constantly for receiving vision data without delays. This would result in one of the two cores being unnecessarily and constantly occupied by the vision task. The ESP32 uses the "spi_slave_transmit()" ESP-IDF method for receiving data from Vision. The blocking nature of this method works well with the task-based program; FreeRTOS notices the blocking operation and switches to another task until something is received. The maximum ESP32 SPI slave clock rate (corresponds to bit rate due to SPI being synchronous communication) is 10 MHz [19]. This is below the FPGA's maximum rate of 25 MHz (as the system clock frequency is 50 MHz) [27]. However, setting the SPI frequency to the 10 MHz ESP32 limit "*can lead to random errors*" [19]. Hence, it was decided to use 8.333 MHz, which is the next highest possible SPI frequency (based on SPI master system clock divisor of 50/6).

Vision sends a "stop" signal, with the kind of obstruction, whenever it identifies an obstruction. To minimize unnecessary communications, Vision does not send "stop" multiple times in a row for the same obstruction. There is only one ball of each colour and hence it is not necessary to inform the server of that obstruction twice as the map will have already been updated with that obstruction, leading to the next path considering that obstruction. After receiving the "stop" signal, the ESP32 decides what to do based on the current (currently executed by Drive) and next (top of instruction queue) drive instruction. It is only necessary to stop the rover if the rover is currently moving forward or if the rover is currently turning and the next instruction is forward. All other scenarios would not lead to a collision and hence the only action is a map update. If the ESP32 decides to stop, it sends a "stop" command to the drive and flushes its instruction queue. Moreover, it informs the server of the stop and the obstruction type, which in turn updates its map and recalculates a path to the original destination, based on the updated map.

4.7.1 Data Model and Data Encoding

The lightweight vision data encoding is similar to the drive encoding mentioned in section 4.6.1. It follows the same format: "ValueInstruction". However, value can either be a character or a number (one byte integer for drive). Again, this encoding was chosen for its low decoding cost. Currently, the only data flow fully implemented is "stop" data. This data is sent when Vision detects an obstruction and follows the format "ObstructionS". For example, stopping due to a red ball results in "RS". Other data packets for image transfers were considered but not fully implemented (see section 4.11).

4.8 Energy Communication

As mentioned in section 4.4, UART was chosen for the communication between ESP32 and energy. A baud rate of 115200 was chosen for the reasons mentioned in section 4.6. Communication only flows from Energy to Control and not the other way around. Energy sends battery status updates to the ESP32, which then forwards them to the server. Energy aims to limit the frequency of status updates sent as battery status does not change rapidly. Sending unchanged status updates continuously to the ESP32 would result in one of the two cores being (unnecessarily) constantly occupied by the energy UART task. Three different energy status types exist: state of charge, state of health, and cell error.

4.8.1 Data Model and Data Encoding

The energy data encoding is identical to the drive encoding mentioned in section 4.6.1. The three status types are encoded as "C" for state of charge, "H" for state of health, and "E" for cell error. Moreover, an identical struct type solution for increasing code readability and maintainability was used.

4.8.2 Energy to Host Computer Communication

As it was not possible to fully integrate the energy system with the rover, a system was set up to allow the energy system to communicate with the server via a host computer. This would at least allow for some level of integration with this system. The Energy module outputs data through the Arduino's serial port. To access this on the computer the steps described by S. Hanselman [28] were followed. Minicom was the tool used, and it provided an interface of

communication with the Arduino's serial port which could be read from. This system was kept open as a co-routine in a bash script which would constantly check for any new data. Once data was detected, it was passed into a small Python script that did some minor formatting for the server, and then published a signal via the MQTT broker for the web server to detect. Because battery data is not something that requires very accurate and up to date detail, it was decided that a signal would only need to be sent every 15 seconds as to not overwhelm the server with data.

4.9 Testing

Communication over wired communication protocols with Drive, Vision, and Energy was tested thoroughly in collaboration with Integration. These tests generally involved Teams calls and debugging using an oscilloscope. Common errors were incorrect pins and attempting to use "Serial" instead of "Serial1" for UART communication with the Arduino Nano Every. MQTT communication was tested by subscribing and publishing to topics directly on the MQTT broker using Mosquitto's (Open source MQTT broker) client tools.

4.9.1 Simulation Tasks

Testing the navigation, localisation, mapping, and stopping due to obstruction logic required the whole rover to be complete. This caused problems as it was desirable to test these without Drive and Vision being fully functional. Waiting for Drive and Vision to be fully functional would have postponed testing this complex logic until last minute, not leaving enough time for identifying and fixing bugs. Consequently, it was decided to implement simulated tasks for both Drive and Vision. Simulating the Drive and Vision tasks made it possible to continuously test the ESP32, server, and web application logic. Hence, we could be certain that this part of the rover already worked seamlessly once we started testing the rover as a whole.

The Drive and Vision tasks were simulated using random number generation and probabilities. This randomness allowed for a huge variety of map terrains and navigation flows that include scenarios that are unlikely to come up from normal user input. Moreover, creating simulated versions of the already existing Drive and Vision tasks allowed smooth switching to/from simulated mode by simply commenting out either the real or the simulated tasks in the main function. Furthermore, the similarity to the real tasks allowed testing for potential timing issues caused by multiple tasks running concurrently. The downside of this randomness was that it made it hard to test for specific scenarios; one had to keep testing until the desired scenario came up. However, writing specific test cases turned out to be cumbersome due to the many components and timing factors involved.

4.10 ESP32 Custom Memory Partition and Memory Utilization

A custom memory partition table, as shown in code 5, was used for increasing program storage space inside the ESP32's flash memory (called "factory" in code 5). Moreover, the default memory partition allocated space to unused functions such as two OTA (Over the Air Programming) definitions.

```
# Name,   Type, SubType, Offset,  Size,
nvs,      data, nvs,     0x9000,  0x6000,
phy_init, data, phy,     0xf000,  0x1000,
factory,  app,  factory, 0x10000, 3M,
```

Code 5: Custom ESP32 Memory Partition Table

PlatformIO was used for providing the following memory utilization statistics: 164.7 KB of RAM used (51% of usable RAM), 813.1 KB of flash memory used (26% of flash defined in the custom partition). This shows that it would have been possible to do significantly more computation on the ESP32. However, it was decided to move as much of the computation onto the server side. Firstly, the AWS EC2 instance's CPU is more powerful than the ESP32's CPUs. Secondly, it is desirable for the ESP32's cores to be idle as this decreased the response time to any incoming messages from the other subsystems. For example, it is essential to respond rapidly to a "stop" signal from Vision without first completing some optimal path finding algorithm.

4.11 Future Work

The main future work idea affecting control is image transfer from Vision to the server. It would be nice to be able to view images of identified balls on the website. As discussed in section 4.4, a raw uncompressed image is about 7.4 Mbs. The selected SPI frequency, of 8.333 MHz (see section 4.7), would allow the image to be transferred to the ESP32 in less than a second. Moreover, it would be possible to send the image to the server in one MQTT message [29], however, sending the image in one go to the ESP32 would be difficult. The ESP32 has only 520 KB of RAM (less is actually available for data) and SPI buffers must be placed into RAM [19]. Storing images on the ESP32 is not a problem as it has 4 MB of flash memory [19] and a custom memory partition inside that flash memory could be created for storing images. It was decided that splitting images before transmission and reconstructing them on the server would be reasonable. A data format for this splitting was defined, following the vision "ValueInstruction" encoding. Instruction "B" for beginning a new image, with the value containing the first part of the image. "I" for

image data, with the value being a part of the image. "E" for end of image, with the value being the last part of the image. This leads to an image containing one "B", multiple "I", and one "E" instruction packets. Small delays between multiple packets is likely and hence ordering should not be an issue. However, it might be necessary to stop the rover during image transmission if the whole process ends up taking too long; it would not be desirable to start sending packets of the next obstruction image before the current one is finished.

5 Drive Module

The aim of the Drive module is to allow the Mars Rover to move in various directions with different speeds by receiving controls from the server. Furthermore, the rover should immediately stop any motion if an obstacle is detected (for detailed requirements see section 2.2.3).

5.1 Development

Development was done on an Arduino Nano Every in C/C++. The majority of functions were written as C++ libraries, in pairs of header and source C++ files and imported into the final code, drive.ino [30]. This allowed for easier testing and integration, as well as easier pull request reviews by other team members. The voltage for the motors was regulated using a non-synchronous, closed-loop Buck SMPS, which controlled the speed of the rover. The position of the rover was measured using an optical flow sensor, which is controlled from the Arduino. Finally, the motors and the rover motion were controlled by a motor control IC. Prior to starting the project, template code was provided for the following functions: SMPS control, optical flow measurements, and basic motion, which were further modified to fit the project requirements.

5.2 Structure

The final code consisted of the following libraries/classes:

- **Optical flow** - measures the current position of the rover and returns it to the main program, so it can be compared to the desired position;
- **High-level movement** - allows the rover to move forwards or backwards for a certain distance or amount of time, as well as turn right or left and stop once the desired position is reached;
- **Low-level movement** - controls the output signals to the motors, causing the rover to move or rotate in a specific direction;
- **UART** - allows bi-directional communication with the ESP32. The ESP32 can send a specific movement instruction to the Drive module, as well as access certain information about the current position of the rover;
- **Speed control** - calculates the desired speed from the displacement error and from the desired speed, the desired SMPS voltage is calculated using a proportional controller;
- **SMPS** - Ensures that the SMPS operates at the desired voltage to achieve the desired speed, as well as staying below its maximum operating current and voltage, to ensure safe operation.

5.3 Instruction Set and Communication with ESP32

5.3.1 Instruction Set

The Drive module consists of the following instructions:

- **doNothing** - the rover is stationary;
- **forwardForTime** - causes the rover to move forwards for a certain amount of time t [ms];
- **backwardForTime** - causes the rover to move backwards for a certain amount of time t [ms];
- **forwardForDistance** - causes the rover to move forwards a certain distance d [cm];
- **backwardForDistance** - causes the rover to move backwards a certain distance d [cm];
- **turnR** - causes the rover to rotate clockwise by 90 degrees;
- **turnL** - causes the rover to rotate anticlockwise by 90 degrees;
- **stopAbruptly** - cause the rover to stop immediately (due to an obstacle being detected), regardless of the currently executed instruction.

To allow easier addition of further instructions, as well as testing and code reviews, these instructions were stored in a C++ enum. [31].

5.3.2 Communication with ESP32 via UART

As explained in section 4.6, the desired instruction is transmitted from the server to the ESP32. This instruction is then transmitted from the ESP32 to the Arduino microcontroller via the UART protocol, printed to Serial1 of the Arduino in the following format '< receivedUARTChars instructionKey >' (excluding the spaces), where receivedUARTChars represents the distance or angle the rover should move and instructionKey is the direction: F for forwards, B for

backwards, R for turning right, L for turning left, and S for stopping abruptly. This communication can be seen in the code below:

```
// Receving new instruction from UART
void UART::recvUARTWithStartEndMarkers() {
  static boolean recvInProgress = false;
  static byte idx = 0;
  const char startMarker = '<';
  const char endMarker = '>';
  char received;
  while (Serial1.available() && !newUARTDataAvailable) {
    received = Serial1.read(){
    if (recvInProgress) {
      if (received != endMarker) {
        receivedUARTChars[idx++] = received;
        if (idx >= numUARTChars) {
          idx = numUARTChars - 1;
          Serial.print("ERROR: UART character buffer not large enough\n");
        }
      } else {
        receivedUARTChars[idx] = '\0'; // terminate the string
        recvInProgress = false;
        idx = 0;
        newUARTDataAvailable = true;
      }
    } else if (received == startMarker) {
      recvInProgress = true;
    }
  }
}

// Decoding the current instruction alongside its paramater
void UART::processNewUARTData() {
  if (newUARTDataAvailable) {
    // Read and pop instruction key
    const char instructionKey = receivedUARTChars[strlen(receivedUARTChars) - 1];
    receivedUARTChars[strlen(receivedUARTChars) - 1] = '\0';
    switch (instructionKey) {
      case 'F':
        instruction = forwardForDistance;
        break;
      case 'B':
        instruction = backwardForDistance;
        break;
      case 'R':
        instruction = turnR;
        break;
      case 'L':
        instruction = turnL;
        break;
      case 'S':
        instruction = stopAbruptly;
        break;
      default:
        Serial.print("INVALID INSTRUCTION KEY\n");
        break;
    }
    newUARTDataAvailable = false;
  }
}
```

While the above format is simple, it is very versatile and extendable. Instructions such as rotating a certain angle or increasing speed can be encoded into a similar format and added to the enum of previously defined instructions. Once an instruction is completed and the desired position is reached, it is sufficient to print 'R' (ready) to Serial1,

thus letting the ESP32 know that the Drive module is ready for the next instruction. An early design choice was whether multiple instructions would be buffered on the Arduino or the ESP32. The final decision was that the ESP32 would send one instruction at a time to the Arduino, and once completed, the next instruction is transmitted. As the ESP32 is more powerful than the Arduino, the goal was minimise memory usage and computational requirements on the Arduino and shift them to the ESP32.

5.3.3 Executing an Instruction

Once an instruction is received and decoded, the current position of the rover is noted using the optical flow sensor and the desired position is calculated from the current position, the "receivedUARTChars" (see section 5.3.2) and the instruction to be executed. The output signals to the motors are are set to achieve the desired motion (forwards, backwards, or rotation). This process is repeated every 800μs (control frequency of the SMPS) until the desired position is reached. While this can cause an overshoot, it is compensated for by reducing the speed of the rover when it is closer to the desired position (see section 5.5). Once the desired position is reached, the H-bridge is disconnected immediately, causing the rover to stop almost instantaneously.

5.4 Movement

Functions controlling the rover are split into two components - low-level functions controlling the output signals to the motors and high-level functions (making use of the low-level functions) to reach the desired target based on the current position. This improves the readability and flexibility of the code.

5.4.1 Translational Motion

Once a move forwards instruction is received, the current position of the rover is noted, from which the desired position is calculated. The rover moves forward until the current position matches the desired position as can be seen in code 6. The error is repeatedly calculated so that speed can be controlled. The boolean "done" is set to "true" once the rover reached its target. The function "forwards()" directly controls the output signals to the motors. Backwards motion is executed in a similar manner, by varying the output signals to the motors.

```
/*
    Movement is a C++ struct containing a float 'e', which stands for position error, and a
    boolean 'done', which is set to true once the desired position is reached.
    The float d is the distance the rover is required to move, in centimeters
    The floats initialY and currentY are the initial y-position of the rover once the instruction
    is received and the current (during instruction execution) y-position, respectively.
*/
movement moveForwardForDistance(float d, float initialY, float currentY) {
  movement ans;
  if (currentY < (initialY + d)) {
    forwards();
    ans.e = initialY + d - currentY;
    ans.done = false;
  }else{
    ans.e = 0.0;
    ans.done = true;
  }
  return ans;
}

/*
    MR (5) and ML (9) are the pins connecting or disconnecting the H-bridge;
    when both MR and ML are 'HIGH' the rover moves, when they are set to low,
    rover is stationary.
    DIRR (21) and DIRL (20) are the pins controlling the direction of movement
*/
void forwards() {
  digitalWrite(MR, HIGH);
  digitalWrite(ML, HIGH);
  digitalWrite(DIRR, HIGH);
  digitalWrite(DIRL, LOW);
}
```

Code 6: Moving Forward A Certain Distance

5.4.2 Rotational Motion

Similar to translational motion, rotational motion is achieved by calculating the desired position. This is done using triangle geometry, namely:

- The distance from the centre of motion (mid-point between the wheels) to the optical flow sensor is 16.2 cm (this can be changed in the code to calibrate the optical flow sensor to specific light conditions).
- The distance between the initial position (when the rotational instruction was received) of the rover and the current position as the distance between two position vectors in the x-y plane.
- Making use of Pythagoras' theorem, the rover rotated by 90 degrees when the distance between the two points is equal to hypotenuse of an isosceles right-angle triangle with a side of 16.2 cm.
- This can be extended to rotate any angle (rather than just 90 degrees) by using the Law of Cosines; however this would increase the computational complexity.

Code 7 shows this procedure implemented in C++. Similar to the previous section, the first function is the high-level function calculating the error and 'requesting motion' from the motors, while the lower function ("anticlockwise()") causes the rover to move.

```
movement turnRight(float currentX, float currentY, float initialX, float initialY) {
  float currentHypotenuse = (currentX - initialX) * (currentX - initialX)
                            + (currentY - initialY) * (currentY - initialY);
  // distance is 16.2 [cm]
  float desiredHypotenuse = 2 * distanceFromRoverCentreToSensor * distanceFromRoverCentreToSensor;
  movement ans;
  if (currentHypotenuse < desiredHypotenuse) {
    anticlockwise();
    ans.e = desiredHypotenuse - currentHypotenuse;
    ans.done = false;
  } else {
    ans.e = 0.0;
    ans.done = true;
  }
  return ans;
}

void anticlockwise() {
  digitalWrite(MR, HIGH);
  digitalWrite(ML, HIGH);
  digitalWrite(DIRR, LOW);
  digitalWrite(DIRL, LOW);
}
```

Code 7: Turning right

5.4.3 Distance Measurements

Distance measurements are done using an optical flow sensor (ADNS-3080 High-Performance Optical Mouse Sensor). The sensor is fairly accurate and when used for testing the Drive module, the rover moved the expected distance, however, it is worth noting the several bugs with the optical flow sensor:

- The optical flow sensor performs poorly in very lit rooms; through various testing it was noted that the sensor performed best in darker environments. This might be a problem, given that the Vision module performs poorly in very dark environments. Furthermore, it might not always be possible to control the lighting of the environment, as it would be the case if the rover operated on Mars.
- When tested on a carpet, the accuracy of the optical flow sensor was very low. Consequently, further testing was done on a wooden floor. Similar to above, this might pose a problem, as it seems that the accuracy of the sensor depends on the terrain; since the rover was only tested inside, on wooden floors (due to cable connections), it is impossible to predict how the sensor would perform on Mars.
- The optical flow sensor inverts measurements in certain setups. Two different optical flow sensors (one from the Drive module and one from the Integration module) produced opposite signs for the same direction of movement. This might be a manufacturing or mounting error. To compensate, the code for measuring displacements contains two extra lines for inverting the measurements, based on which sensor is used; these lines were commented out as needed.

5.5 Speed and Voltage Control

5.5.1 Speed and Voltage Controller

Controlling the speed of the rover is necessary for minimising overshoot. The rover's speed can be adjusted by varying the voltage applied to the motors, which is set by the operating point of the SMPS. A proportional controller minimizes overshoot while also also being easy to implement and tune. Furthermore, a proportional controller avoids oscillation, unlike a PID controller. The controller can be seen in code 8 and can be explained as follows:

1. The position error is calculated, as the difference between the desired position and the current rover position.
2. From the position error, the reference speed is calculated using a proportional controller. The coefficient for calculating the speed from the error is set to 1.5. This value was obtained experimentally by measuring the speed of the rover and the time it takes to slow down. The maximum speed of the rover was measured to be around 15 cm/s and it was decided that the rover should start slowing down around 10 cm prior the desired position. This ensures that the rover moves at maximum speed when the error is larger than 10 cm, while still having sufficient time to slow once close to the desired position.
3. Using the reference speed, the operating point of the SMPS is calculated and fed into the PID controller used in the EEE Power Lab [32], ensuring that the SMPS is operating at the desired voltage. The coefficient for calculating reference voltage from the reference speed is set to 0.25 as the maximum voltage is around 4V, which corresponds to a speed of 15 cm/s.
4. As the rover gets closer to the target, the speed and voltage are reduced, thus minimising overshoot.
5. To ensure frequency separation and allow the voltage to settle at its required opting point, the speed is recalculated at a ten times lesser frequency than the voltage, using a counter. The counter is incremented every time the voltage PID controller is invoked and once the counter reaches 9, a new reference speed is calculated and the counter is reset.

```
float SpeedController::calculateVoltageReference(float e){
   // (float) e is the position error; difference between desired position and current position, in cm
   /*
    (int) counter ensures frequency separation by calculating the desired speed at a ten times
    lesser frequency than the voltage controller
   */

  if (counter == 9) {
    // Proportional controller; first calculate speed from the error and then the voltage from the speed
    float referenceSpeed = distanceToSpeedCoefficient * abs(e);
    referenceVoltage = min(speedToVoltageCoefficient * referenceSpeed, maximumVoltage);

    // Reset counter
    counter = 0;
  } else {
    // During these states the spped controller is idle - reference voltage does not change
    counter++;
  }
  // return so it can be used in the voltage PID controller used for running the SMPS
  return referenceVoltage;
}
```

Code 8: Speed and Voltage Controller

Figure 9 overleaf shows a high-level representation of the Drive module. First the position error is calculated and fed into the speed controller to obtain the speed at which the rover should move. From the reference speed, the reference voltage is calculated and set as the operating point of the buck SMPS. The output voltage of the buck SMPS is used for driving the motors, alongside output signals from the Arduino, which control the direction of movement.

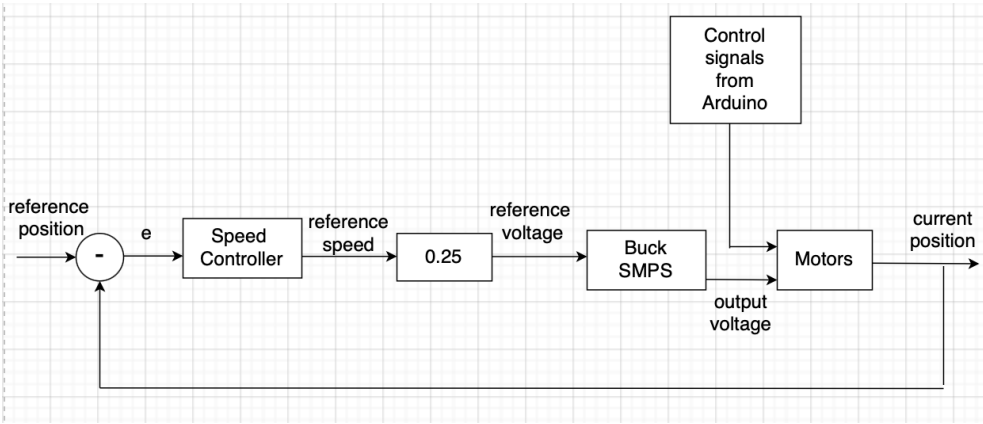


Figure 9: Diagram showing overall controller and drive system

5.5.2 Time Measurements

An error that was discovered in the early stages of the project were false time measurements using the "millis()" function on the Arduino microcontroller. This error was due to the Timer A being enabled, which was used for setting the frequency of the SMPS PID controller. This error was fixed by removing Timer A and replacing it with the "delay()" function with the same period. This allowed for accurate time measurements, which are needed for instructions depending on the time elapsed. Furthermore, having accurate time measurements can later be used to set the speed at which the rover moves. Prior to applying this fix to the rover itself, the code was tested on an SMPS to verify functionality as well as safety, by checking the output voltage and current.

5.6 Reflection and Future Work

While all the previously mentioned targets have been met, there is room for future work. The Drive module needs to be tested on various terrains with different lighting conditions. During initial testing, it was noted that rover would not move on some carpets. The accuracy of the optical flow sensor depends on both the terrain and the lighting conditions, as explained in section 5.4.3. Easy additions would be calculating the total distance travelled by the rover, as well as calculating the current rover speed and transmitting them to the server. As explained in section 5.4.2, a further addition would be causing the rover to rotate a certain angle, rather than a fixed 90 degrees. While an easy addition, which can be achieved by extending the turning function and using the Law of Cosines, it would provide little benefit to the overall project. This is due to the navigation and mapping design that only supports 90 degree turns. Finally, both a beneficial and lengthy addition would be full speed control. Such control would allow the server to impose a certain speed to the rover, causing it to move at the given speed before automatically decreasing it prior to reaching the target. This would require extension on both the Command and Control modules, as well as the current speed/voltage controller.

6 Energy Module

The overall aim of the Energy module was to enable the Mars Rover to operate without the need of wired connections to a power source by designing an energy storage system and enabling charging capability via photo-voltaic (PV) panels (for detailed requirements see section 2.2.4).

6.1 Development

Development was done using the Arduino Nano Every to send instructions to the SMPS. The SMPS was in buck mode for cell testing and in boost mode for PV panel testing. To characterise the cells, the provided test script was used (see section 6.2) and to characterise the PV panels, the same PID parameters from the provided script were used in a boost closed-loop configuration (see section 6.5).

6.2 Cell Characterisation

Initially, each cell had been tested for performance by cycling through the different states in the test script. The aim was to run each cell through 2 charge and discharge cycles to extract the cell capacity more accurately, however, even when running some cells for 40 hours, there was no complete discharge to then restart the charge and discharge cycle. Regardless of how long each individual cell was cycled for, none reach the minimum voltage threshold for which they would be forced to be charge again at the constant 250 mA, however, during and after the rest state (state 2; see fig. 10), a plateau region was identified as the measured current decreased to minuscule values between the

range of 1-2 mA. The cells remain in this range until manually disconnected. This could have been testament to the durability of the individual cells. The flat discharge curves and extended periods of use to achieve no full discharge raised concerns, however, research indicated that because of the chemistry of LiFePO4 cells flat discharge curves are normal [33]. In the end, it was discovered that the cell characterisation was conducted in the wrong SMPS mode - working in asynchronous rather than synchronous buck mode. The flat discharge curves were explained to be the reason why methods based on voltage estimations would prove difficult for SOC estimations later on in the project.

Once it had been discovered that the SMPS was operating in the wrong mode, the jumpers J1 and J2 on the SMPS were moved to the left to ensure synchronous operation. The cells were once again connected to port B of the SMPS to characterise. However, when initiating each test, the cells entered the error state upon connection, indicated by the LEDs connected to the Arduino. With assistance from colleagues, the process of debugging began to address why this was occurring. The following steps and outcomes were recorded:

1. Use the multimeter to manually read open-circuit voltage of the cells.
   (a) The only reason why the error state would occur according to the test script was if the cells were not within the readable range of 2.4V - 3.6V.
   (b) The multimeter confirmed that the cells were all in working order, none suffering from overcharge or being completely depleted.
2. Connect the Arduino via a USB cable to a laptop port to verify that the Arduino's voltage sensor is not misreading the cell voltage.
   (a) As the SMPS and the Arduino had been setup incorrectly initially, this test was conducted to verify that the 1.03 correction for measurement error worked in the correct configuration.
   (b) Using the Serial Monitor on the Arduino IDE it was confirmed that the multimeter readings mirrored that of the Arduino.
   (c) From this test it was discovered that the problem was that the cells started in the error state for 1 second and then went to the zero state.
   (d) From the state diagram in fig. 10, it can be seen that the '0' state indicates that the SMPS switch was in open-loop (OL) configuration.
   (e) This posed another problem in the process of debugging because the switch on the SMPS was actually switched on the closed-loop (CL).
3. The OL/CL switch was manually adjusted on the SMPS to see if trial and error would cause the Arduino to respond as expected.
   (a) After multiple attempts of switching the CL/OL switch it was concluded that the issue was the detection of this signal.
4. The colleague responsible for the Integration module ran the test script on their Arduino.
   (a) The lines in code pertaining to SD card initialisation were commented out to allow for testing with the SMPS as the Integration module is not equipped with an SD card reader.
   (b) The Integration Arduino was able to detect the switch in closed-loop and open-loop on the SMPS.
   (c) This indicated that the 'INPUT_PULLUP' function was faulty on the Arduino being used on the Energy module.
   (d) The test script was then analysed to see if the digital pin used for OL/CL switch could be bypassed/reassigned.
   (e) As the pin had no external port to bypass this and the pin used had been integrated into the PCB it was concluded that the Energy module Arduino was no longer fit for use.
   (f) Luckily, the parties responsible for the Energy and Integration module were both in London so the functional Arduino was borrowed to continue cell characterisation until it was needed for filming the project video.

The finalised results for a single cell are shown in fig. 11 overleaf. To extract the cell capacity, one can mathematically integrate the discharge section of the curve of 'measured current against time' [34]. As the rated capacity of the test cells is 500mAh, one would expect discharging at 250mA to last for 2 hours based on the units of measure. By truncating the measured current array only to read values of current measured when the current reference was -250 mA, capacity was calculated to be 479.38mAh by reusing the trapz() function in MATLAB and dividing the cumulative time in hours and the measured current. This effectively provides the normalised capacity of the cell.

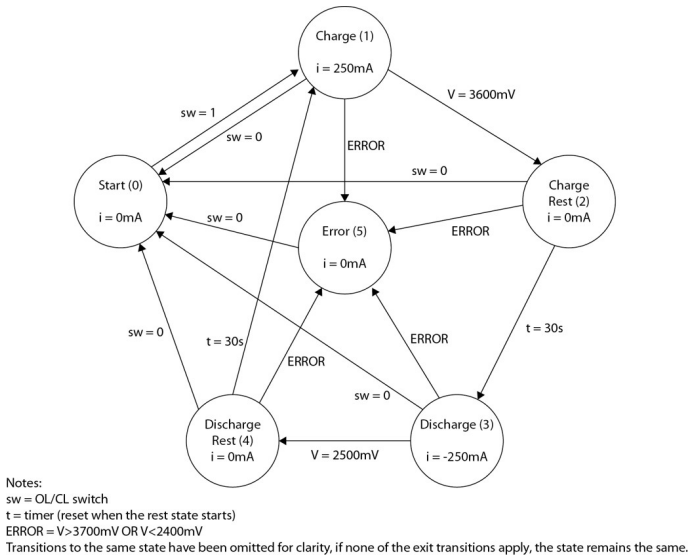
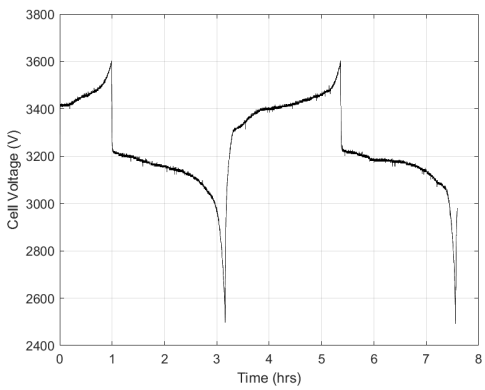
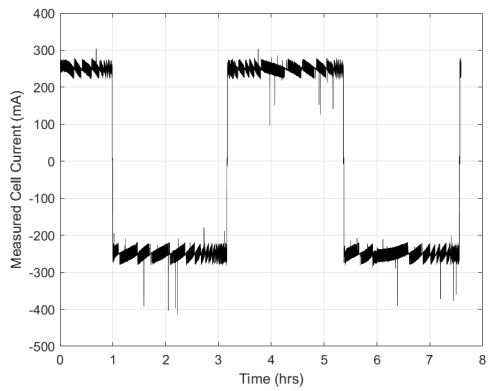


Figure 10: State diagram of the state machine in the battery characterisation code provided



(a) One cell's voltage retention over 7.5 hours



(b) One cell's current profile over 7.5 hours

Figure 11: Analysis of charging and voltage retention of one cell

6.3 Measuring SOH and SOC

State of health (SOH) and state of charge (SOC) both pertain to cells and are measures of their performance. Sate of health pertains to the long term performance, whereas state of charge pertains to short term performance of cells. In order to measure both functions, algorithms must be implemented to measure cell characteristics such as cell impedance, current, and voltage. [35] The various algorithms for the functions have been stated in section 6.3.1 and the reasoning for the chosen method have been provided.

6.3.1 SOC

State of charge is defined as the ratio of a cells current capacity to the nominal capacity - meaning that it is the ratio comparison of the remaining charge in a cell with the maximum amount of charge that can be stored in the cell. A reading of 100% would indicate a fully charged cell, whereas a reading of 0% would indicate a cell with no charge [36]. There are 4 main classifications of SOC measurements which are implemented more generally in the industry [37] and for small-scale domestic activities. These consist of (with increasing complexity):

1. Direct Measure
    - Open circuit voltage measure (OCV measure)
    - Terminal voltage measure
    - Impedance measure
    - Impedance spectroscopy method
2. Indirect Measure
3. Adaptive Measure
4. Hybrid Measure

The open circuit voltage (OCV) method is a direct method of SOC measure which entails the measurement of the cells' voltage in the no-load state. No external circuitry which draws power from the cell must be connected in this setup so that measurement occurs while the cell is in chemical equilibrium. This often requires the cells to be disconnected for over 2 hours to ensure internal chemical equilibrium is met. This is a simple and effective method for small-scale projects which do not require frequent updates of the cell SOC [36]. As the initial cell characterisation step required no load to measure cell voltage, the Arduino script could be edited to simply read and record the cell voltage before use, however, as one would be guessing whether the cell had reached chemical equilibrium, this method was deemed unsuitable. Additionally, as the method requires isolation from external load, it would not be suitable for integration with the Mars Rover as real-time performance of the cells is imperative to fulfil recharging objectives. The downsides of the method are: great sensitivity to precision of the equipment being used to sense voltage, the open-loop nature of the method which cannot be automated, unsuitability for cells with flat SOC-OCV characteristics (unsuitable for the LiFePO4 cells under testing).

The terminal voltage method is a direct method of SOC measurement which operates similarly to the OCV method in regards to terminal readings being required but has subtle differences. During the use of the cell, terminal voltage drops are recorded to estimate the variations in internal resistance values to in turn gauge the cell EMF [37], [39]. The EMF is proportional to the terminal voltage and follows an approximately linear relationship with SOC as seen in fig. 12. Similar to the OCV method, this is an inexpensive and quick method to estimate SOC, however, studies have found that timing this method is pivotal [33]. This is because, as cell discharge comes to a halt, the potential for error in the estimated SOC increases due to the sudden fall in terminal voltage at the end of discharge.

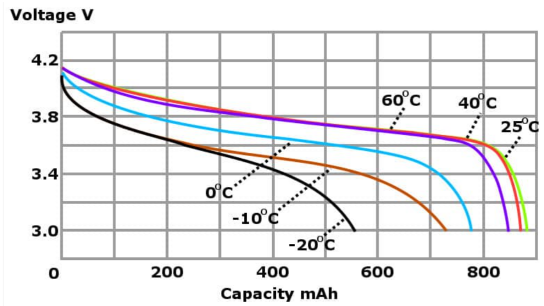


Figure 12: A typical voltage vs SOC curve [38]

As both of the aforementioned methods rely solely on voltage measure for SOC estimation, it must be noted the difficulties posed by single parameter methods such as these. Cell voltage curves are non-linear parameters which are affected by other parameters such as cell temperature which are not being measured. More specifically for LiFePO4 cells, as their voltage-SOC curves are typically flat [40], it is difficult to deduce a difference between charge levels for which there is not much great difference between the voltage value. At the extremes of the charge and discharge curve is where this information is better ascertained.

Impedance spectroscopy is a direct method of SOC which particularly pertains to estimating cell impedance directly. Impedance spectroscopy works by applying an AC voltage across the cell - the AC voltage used is small and parameters of the AC signal including frequency and amplitude are known. In order to measure the cells' impedance, the in-phase AC current response is logged and the impedance is determined by employing Ohm's Law. The impedance is calculated as a ratio between the AC voltage and the in-phase AC current response. This is a non-destructive technique that is dependent on the perturbation of an equilibrium state. This method requires more equipment to implement and assumes that AC signal used to excite the cell and the in-phase AC response have a linear relationship.

A method more suited to the project to measure SOC, is an indirect method called 'coulomb counting'. This method is used to measure the charge entering the cell and leaving the cell [38]. One could use a shunt resistor in series with a cell to measure the current flow, however, as current measuring capabilities have already been enabled on the Arduino, using the analogue input pins would be best to deduce this. The mathematical integration of the current expended against time can be used to compare against the rated capacity of the cells and gauge a percentage value of SOC. As the opted method for extracting cell capacity in the cell characterisation stage does not deviate from this, one could save time by implementing this method. The reasons why one may opt not to use this method is that it is a single parameter method which does not take into account cell efficiency and limitations in measuring capabilities of the equipment.

6.3.2 SOH

State of health is a measure of cell longevity in comparison with their 'fresh state' having not been used before. It is used to indicate the overall condition of the cell - defined as a ratio comparison of a cells instantaneous capacity to the capacity of a new cell [41].

State of health can be determined by measuring the parameters of the cells that suffer significant degradation with age. One could keep a log of cell impedance at the beginning of each use or similarly measure cell conductance, as single parameters, however the use of multiple parameters leads to greater accuracy. Additional ways to gauge SOH include: self-discharge rate, cell power density, and cell capacity comparison to rated capacity [37].

Cell impedance and cell conductance measures use the same methodology to derive their values - both work in the same way as impedance spectroscopy measure for SOC measure [39]. The parameters are chosen to be measured as with age, cell impedance is expected to increase and conductance is expected to decrease. Factors that have consequential effects on cell impedance and conductance are mainly to do with the chemical reactions within the cells and are influenced by the following [42]:

- Electrolyte impedance - this increases over time from dendrite growth.
  - Electrolytic impedance is characterised by the amount of ionic charge carriers and their ability to move through the electrolyte.
  - With age, the concentration of the electrolyte decreases as some of the electrolyte is consumed in the corrosion byproducts of the electrodes and through secondary reactions.
- Rate of reaction at the electrodes - reaction capability decreases with age.
  - As cells age, the porosity, crystalline structure and chemical composition are degraded.
  - Cells with Lithium in their chemical composition, typically suffer collapse of the receptor channels in their electrodes or get blocked with lithium byproducts of corrosion.
- The mobility of ions over time.

Cell impedance and conductance measurements are extracted by exciting the cell with a small-signal AC signal to measure the in-phase response current. The impedance is derived from Ohm's Law as a ratio between the applied AC voltage and the in-phase AC current response, assuming that the relationship is linear. The conductance is the mathematical reciprocal of the impedance [41]. For this to be implemented, an initial measure of impedance/conductance must be recorded to set a 'standard' for comparison. This is effective as the process can be executed whilst the cell is in operation. For this project, one could use the AWG generator to excite the cells when in use, and the current measurement capabilities of the Arduino could be used to read in-phase current response. Phasor calculations would have to be implemented in the Arduino script as imaginary values are expected. Despite the simplicity and inexpensive nature of this method, one must note that in fully integrating the Energy module with the Mars Rover, the AWG generator would have to be accounted for when considering the cell configuration for the final battery pack design. The AWG generator draws 5A via a USB connection, whereas the other modules have an upper bound of 3A current being drawn.

6.4 Battery Pack Configuration and Cell Balancing

Cell balancing is implemented to ensure that individual cells within a battery pack do not unequivocally charge or discharge as this leads to issues regarding efficiency. A series cell configuration or a mixture of both series and parallel is chosen to form a battery pack from cells of the same cell chemistry, same rated capacity and operating voltage. Despite these similarities in cell composition, issues arises when they are combined to be cycled through charging and discharging through use [43]. Without conducting cell balancing on the battery pack, issues may occur.

- Cell Degradation
  - Premature ageing and cell composition breakdown.
- Incomplete Battery Pack Charging
  - Through energy taxing charging and discharging, each cell will degrade differently and weaker cells will charge and discharge at a faster rate than neighbouring cells.
  - Once a singe cell achieves maximum charge, the charging process will stop and stronger cells will remain undercharged.
- Incomplete Battery Pack Discharge
  - Weaker cells within the battery pack will discharge at a faster rate than healthier cells.
  - As the weaker cells will deplete to minimum voltage quicker, in the case of the Mars Rover, a signal would be sent from the Arduino to enable recharging.
  - This would pose an issue for the healthier cells as they would neither discharge fully nor charge fully.
  - The full potential of cells would not be realised and premature charging will occur in the healthier cells which are not fully discharged.
- Thermal Runaway of Cells
  - Thermal runaway is the process in which a cell operating at a higher voltage than the rest of the cells goes by undetected and overcharges further.
  - For example, if a cell was operating at 3.6V and the other cells in the pack were operating and 2.8V - if those cells required charging to 3.6V the single cell sitting at 3.6V would be charged further by that 0.8V.

In order to balance cells, there are various techniques such as: passive balancing, active balancing, lossless balancing and redox shuttle method [43]. Of the aforementioned methods, the one most suitable to being integrated with the Mars Rover would be the active cell balancing using the SMPS. Theoretically, a higher voltage cell would be discharged stepped-down via buck mode to provide charge to the lower voltage cell. This method does have its shortfalls as charge can only be transferred from cells with greater terminal voltage readings. There are also losses that occur due to switching as well as the requirement of multiple SMPS units and an awkward cell configuration.

6.5 PV Panel Characterisation

The PV panels were initially tested manually using the lamp and multimeter to measure the delivered current and voltage. After leaving the PV for a sufficient amount of time, the maximum readable voltage from the panel's terminal fluctuated between 4.8 and 5 V. Operating within this range, the delivered DC current was 0.098 A. In the event that the PV panels would be integrated into the final rover, the series configuration was agreed upon as one could expect a series voltage ranging between 19.2 - 20 V (see section 8.4). However, as the Arduino being used was borrowed and quite a lot of time had been taken by initialising battery cells in the wrong configuration, the full characterisation of the PV panels using the SMPS could not be realised in due course. The proposed steps (had time permitted) have been outlined below:

1. Characterise PV panels by sweeping duty cycle of Boost SMPS in synchronous mode to generate I-V curves.
2. Employ the MPPT algorithm in which either load values are manually changed or voltage perturbation [32] occurs to gauge the maximum power point.
3. After finding the MPPT for one angle, employ the same MPPT algorithm for the other angle.

6.6 Integration with Communication

For the communication with the ESP, the UART connection was chosen, the connection process and reasoning for the choice of using this interface is described in more detail in section 4.8. The Arduino is the component for which all analogue data is sent to - the analogue input pins receives the necessary measurement signal. This data is dependant on which component is connected to port B of the SMPS. The proposed algorithms implemented in the Arduino IDE would calculate SOH, SOC and MPPT and send the necessary data back to the ESP32.

6.7 Future work

If it had been made possible to integrate the energy module with the rest of the Mars Rover, further design alterations would have to be implemented. An initial implementation that was discussed used two SMPS circuits (not possible to use a single SMPS to alternate between buck and boost mode which is required for the cells and PV panels). The battery pack could be split into two separate battery banks that would be connected via a micro-controller similar to the Arduino Nano Every. In this configuration, one battery bank would be electrically isolated from the Mars Rover using a switch but would be charging via the PV panels whilst the in-use battery pack delivers sufficient energy to the rest of the rover. Once an agreed threshold is passed, the battery packs role will be switched to ensure the rover is constantly running with sufficient power without the need to 'dock' to recharge. The PV panel and SMPS circuit would also be restricted by voltage/current thresholds written onto the Arduino.

This configuration formed the basis of further development of ideas on how to integrate Energy into the rover. The final solution proposed is covered in section 8.4 which varies slightly (the final configuration of parallel cells was used with an added redundancy due to the environment in which the rover would operate).

7 Vision Module

7.1 Basic Goals

For the vision module there were a few original goals that were set out. The primary focus was to detect the five coloured balls provided (red, yellow, teal, blue, and violet). Additionally communication between vision and control was required (for details see section 2.2.5). On top of this a few extra goals were added. One was to detect unknown objects. This would be used to help generate a more interesting map comprising of more than just colored balls. Distance detection was important as the rover should only be alerted if the detected ball was close enough to the rover. A final soft constraint was noise tolerance: small background objects with similar colors to the balls, or just random noise should be ignored.

7.2 Structure

The structure of the vision module can be separated into four main sections. The first reads video data, such as pixel color, pixel position, and packet information. There is a color detector sub-module, which takes in the pixel color data and returns what possible colors it could belong to. A blob detector sub-module is used to detect blobs of pixels of similar colors. A communication module is used to relay information to control.

7.3 Development

Developing directly on the FPGA was deemed unfeasible; Quartus takes too long to compile and run, which is unreasonable for small changes. Instead, where possible, another language such as JavaScript was used to prototype the vision system first. This code was then translated into Verilog for final testing.

7.4 Video Data Reader

This section was provided to us by the EEE department [44]. The section manages all of the setup required for the camera connection to be made and processed. It reads data from the camera to detect packets and filters out any non image packets, based on the specifications provided by Altera [45] and the camera manufacturer OmniVision [46]. For image packets it reads out the pixel colors, and keeps track of how many pixels have been received for computing the x and y position of the currently processed pixel.

The pixel counting works because the image's dimensions are known, and pixels stream into the module row by row, from the top row of the image to the bottom. The x and y position of the pixels are kept track of by incrementing the x position counter on every new pixel. When the x counter reaches the size of the image width, it is set to zero and the y counter is increased. At the beginning of every frame, the x and y counters are reset to zero.

7.5 Color Detector

The color detector is the first phase of the ball detection algorithm. It reads single pixel information and uses this to decide which category the pixel belongs to: red, yellow, teal, blue, and violet balls, or unknown objects.

7.5.1 Development Process

Two different methods for color detection were considered. The first was color distance, where the RGB values are used as coordinates in a 3D space. Given two different colors, we can use the 3D space to compute the distance between them, using Pythagoras' Theorem. The smaller the distance the more similar the colors. This method is known to have certain problems in color detection since the color palette is not perceptually linear [47], hence, attempting to use a linear comparison for the distance in RGB space will not always work accurately.

Because of the RGB color space not having a perceptual linearity [47], a second possible color space was researched: HSV. Instead of representing color using differing amounts of red, green, and blue light, this method uses hue, saturation, and value. Hue is represented as an angle around a circle of all possible color hues. Saturation is a percentage which defines how much intensity the colour has, where 0% is white, and 100% is fully colored. Brightness (Value) is a percentage describing how bright the colour is, where 0% is black, and 100% is full brightness.

Using the HSV colour space it was possible to sample the hue of every pixel, and use that to determine the pixel's color. Value and saturation could be used for additional checks, ruling out whites and blacks. This method has the benefit of being closer to human color perception.

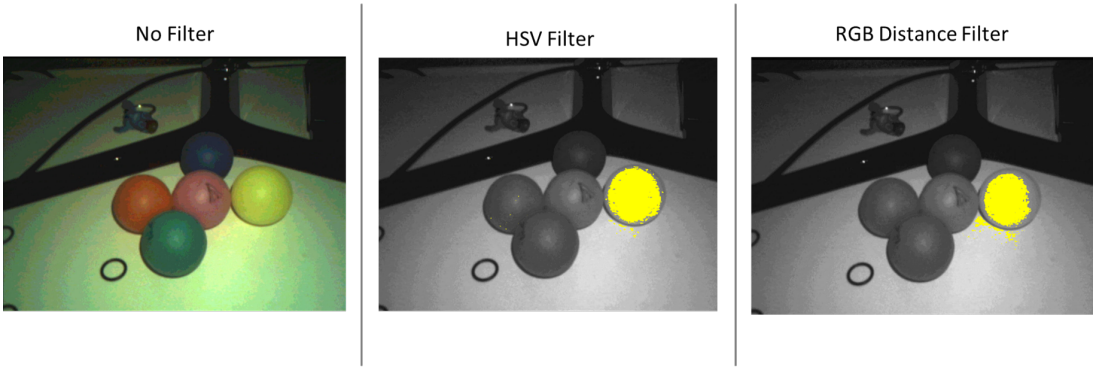


Figure 13: Comparison between HSV colour detection and RGB distance detection

Both methods were first tested in JavaScript. Figure 13 shows the test results for the yellow ball. It can be seen that HSV seems to produce a marginally better results. This however did not translate to the FPGA. The conversion algorithm used for the color space translation is a modified version of the algorithm described by user David H [48]. In preliminary testing this seemed like the better method. hence, a Verilog translation was developed and implemented. However, we quickly ran into a timing problem; the conversion algorithm requires using multiplications and divisions, both being relatively slow. Because the FPGA has built in multiplication modules, multiplication can be executed quite quickly as long as only a certain amount of them are used. Division on the other hand does not have dedicated hardware support, so a divider needs to be synthesised. This ends up being quite slow.

When testing on the FPGA the dividers caused timing problems. Sometimes wrong HSV values would be produced for a given RGB input, as the data would not have enough time to propagate through the entire module before being read. Attempts were made to optimize the module to improve timing. First, the algorithm was changed to only include one multiplication and two divisions, the minimum possible. However, the problem persisted. Pipelining was added to the module, to only require one heavy computation at any given clock cycle. On its own this was quite successful, however, when combined with the blob detector module, the combination of both was unstable.

Looking back at the results of fig. 13 we can see that the RGB distance is not that much worse than the HSV detection. The benefit of RGB distance is that it only requires multiplication and no division. This method worked well without any of the optimisations described above. Nevertheless, pipelining was still added to ensure safe margins

for when additional modules were added later on. Additionally, apart from the color distance, extra checks where added to improve the detection. These are mainly comparisons, such as ensuring r > g + b for red. This eliminates some of the problems with color perception, while being less computationally expensive than other solutions, providing accurate enough detection of the balls.

7.5.2 Testing Methodology

As mentioned in section 7.5.1, testing was done in JavaScript. The p5.js library [49] was used. The specific detection algorithms were both implemented in JavaScript. Parameters, such as the hue range for HSV and the color distance threshold for the RGB distance method were tuned using this setup. Because color distance was the final method chosen the testing methodology for that method is discussed in more detail.

Firstly, the FPGA was connected to the camera, and then connected via VGA to a computer. This was used to take a few pictures of the balls under different lighting conditions. These were then loaded into JavaScript and each pixel was analyzed as it would be in the FPGA.

In total there are three main parameters which can be tuned for any given color: the reference color, the color distance, and extra checks. For the primary colors: red, green, blue, and yellow the ball colors are almost exact. Because of this, the color detection for these balls works best when the reference color was the most intense possible color. For example, for yellow we use #ffff00, red would be #ff0000, and so on. The violet ball is not exactly perfect violet of #ff00ff. After some testing, and using some color picking tools on the ball images the best reference color for violet was found to be #ff00b4. Finally, it was decided that unknown objects would have to be dark objects. Consequently, it was decided to use black as the reference color #000000 for them.

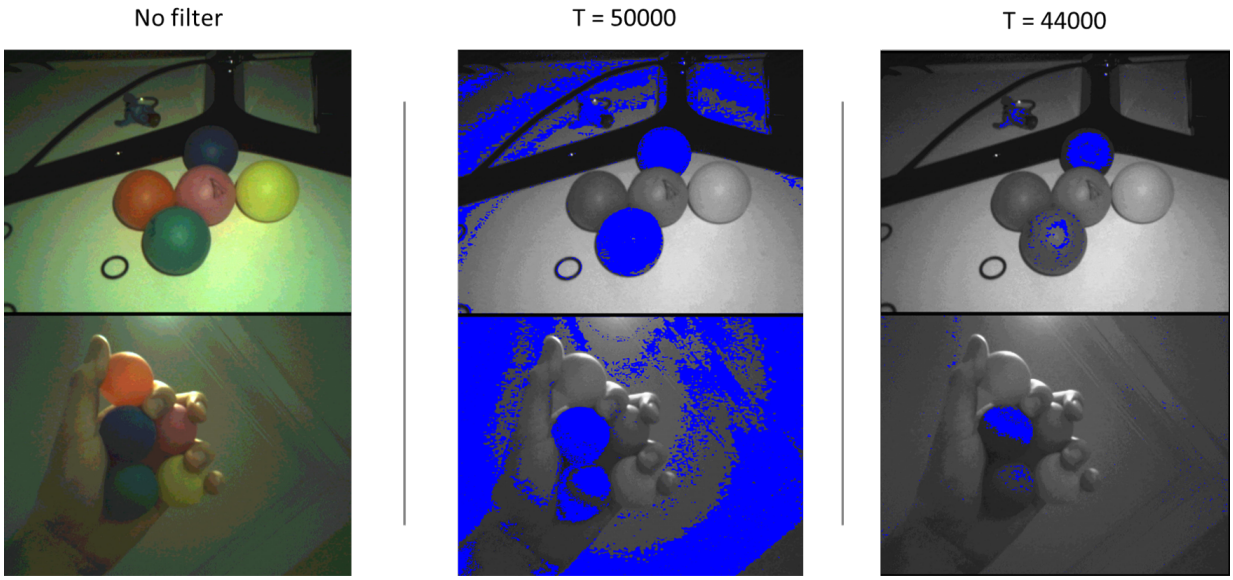


Figure 14: Comparison between different thresholds for RGB distance detection for the blue ball

Once these reference colors were analysed, a threshold distance was found for each color independently. This was done by first setting the distance threshold very high, and decreasing it until parts of the ball stopped being detected. The threshold was fixed to this point. This was done with images in different lighting conditions. Afterwards, a value in the range of chosen thresholds was picked such that enough pixels were selected under all lighting conditions. In fig. 14 we can see the difference tuning the distance threshold can have on colour detection: blue detection using a distance threshold of 50000 units and 44000 units can be seen. The detection is poorer in low light conditions, however, it was deemed good enough for the rover's purposes.

With distances we can get fairly accurate results. Nevertheless, adding some extra low computation checks, such as comparisons, can provide additional accuracy. These do not have a straightforward process to work out, hence, some analysis on the color of the balls was done, and some heuristics were developed. One example with blue, comparing fig. 14 with fig. 15, using the same threshold but improved extra checks fully removes any incorrectly detected pixels on the green ball. Here we used the extra check of $b > g$ & $g > r$. The analysis for this was done by looking at what pixels where erroneously detected. Firstly, the pixels in the green ball were analysed. It had to be ensured that there was more blue than green, this led to the $b > g$ part of the condition. Secondly, we looked at the dark shadowy areas that got picked up. They seemed to contain more red on average than green, so the metric $g > r$ was added. This eliminates almost all noise under most conditions. The final parameters for each of the objects are shown in table 4 overleaf. This table completes the development of the color detection module.

T = 44000 with extra checks

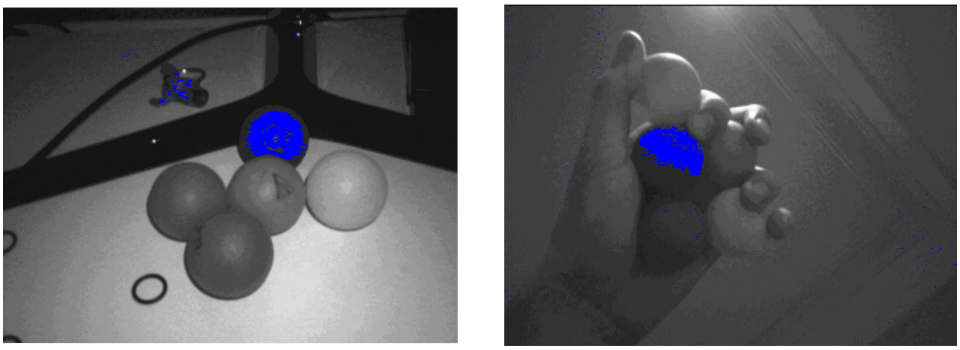


Figure 15: Blue ball detection using $T = 44000$ from fig. 14 with the addition of an extra condition $b > g$ & $g > r$

Object	Reference Colour	Distance Threshold	Extra checks
Red Ball	#ff0000	30000	$r < g + b$
Yellow Ball	#ffff00	20000	$g > r - 50$
Teal Ball	#00ff00	35000	$g > b$ & $g > r$ & $b > r + 10$
Blue Ball	#0000ff	44000	$b > g$ & $g > r$
Violet Ball	#ff00ff	33000	$(r < b + g + 20)$
Unknown Object	#000000	2000	N/A

Table 4: Colour detection parameters used

7.5.3 Verilog Implementation

The colour detection system requires multiplications. As mentioned previously, this requires using pipelining to improve timing constraints. A three stage pipeline was used. The first stage is the input stage, where the input data is read. The second is where all multiplication occurs, and the final is the comparison stage where the conditions are tested. Because of pipelining, the colour detector requires inputs: clk, reset_n, r, g, b, x, and y. The x and y are required because the delay caused by the pipeline means that the output of the module is not the output of the current x and y value, but rather the x and y of two pixels before.

The outputs of the module, are a list of bits for each of the colour modes, where high is detected, and low is not detected, as well as and x_out, and y_out which define what x and y value corresponds to the current color detection output. The module's instantiation was defined by code 9 below.

```
module SXPP_COLOR_DETECT(
    clk,
    reset_n,

    r, g, b,
    x_in, y_in,

    is_red, is_yellow, is_green, is_blue, is_pink, is_unknown,
    x_out, y_out
);

input logic clk, reset_n;
input logic [7:0] r, g, b;
input logic [10:0] x_in, y_in;

output logic is_red, is_yellow, is_green, is_blue, is_pink, is_unknown;
output logic [10:0] x_out, y_out;
```

Code 9: Colour Detector Module Instantiation

7.6 Blob detection

Being able to identify what pixels belong to a specific colour is not the whole story. From here we need to be able to group clumps of blobs together. This is required to properly identify if an object is a coloured ball, or just a spec of a colour in the background, or camera noise. This is what the blob detector is for. For the method used, inspiration

was taken from a video tutorial from "The Coding Train" [50]. However, many changes where made to better suit the scope of the project.

The original algorithm works by instantiating an empty list of blobs. Each pixel is scanned one by one, and if the blob matches a colour it compared to each item in the blob list. If it is near none of the current blobs (or no blobs exist yet) a new blob is instantiated into the blob list. If it is near one of the blobs, it is added to that blob which is expanded. This is repeated until all of the pixels have been scanned. This can be done for each of the colours used.

7.6.1 Development Process

The algorithm described in section 7.6 above works well in normal programming languages, however, in hardware the idea of creating a new blob does not really exist; hardware to store all the information must be pre-defined. Instead of starting with a list of no blobs, our system begins with a fixed list of n blob detectors, ordered in a grid. Each detector only stores information about one blob and is reset at the beginning of every frame. The grid acts as an initialisation area, so each blob must be initialized by a pixel in that region.

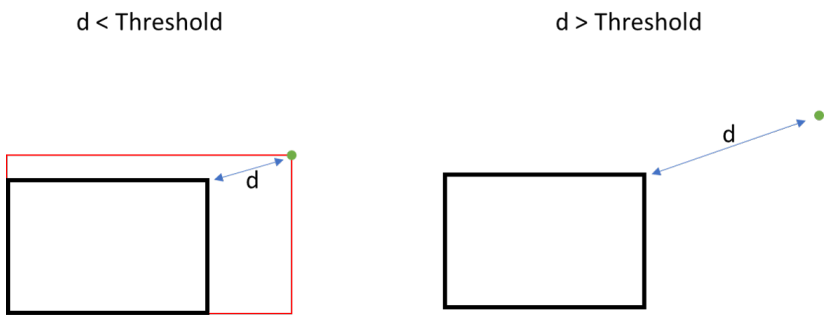


Figure 16: Demonstration of blob detector threshold

The first development step was to create a simple, singular blob detector. The way this worked was to have each output of the colour detector routed into a new blob detector module. The module would take only one of the colour inputs as an input to alert the detector that the current pixel should be used for detection. From here if the detector has not yet been initialized it defines its left, right, top, and bottom bounds as the current pixel's x and y values respectively. From here on out, it will run a check on each new valid coloured pixel; if the distance between the center of the pixel and a radius, calculated as the distance from the center of the blob's bounds to one of the corners, is below a certain threshold then the blob's bounds are expanded to include this pixel. A visual explanation can be found in fig. 16. On the left, a new pixel is included, leading to an increase in the blob bounds. On the right, the pixel is outside of the range, leading to it being ignored.

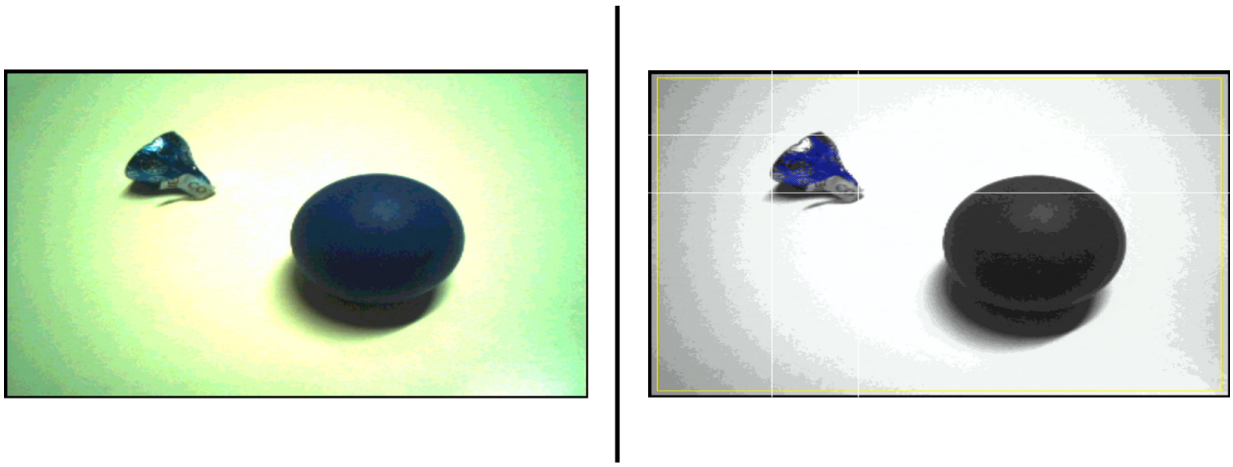


Figure 17: Example of the detector selecting the first object it finds

This current system has one big problem that needed to be fixed: having only one detector meant that if a speck of a colour is detected at any point before a pixel actually pertaining to the ball is then the detector would be initialized to that spec, and would ignore the rest of the image. An example of this can be seen in fig. 17. Here, the "noise" is a large object for demonstration purposes. However, if it was even a single pixel of blue it would be detected first, leading to the blue ball being ignored.

To work around this, a grid system is used. Each detector is passed a parameter defining a range of x and y values, which define where a pixel must be located to initialize the detector. After initialisation, detected pixels can lie anywhere, but in the beginning it must lie in the predefined range. Each detector is given a different range in the grid, increasing the chance of having at least one of the detectors pick up the ball. After an entire frame is processed,

the radius of each blob is compared and the detector with the largest blob is chosen to be the true blob. This could theoretically be expanded as to be able to detect the n largest blobs to detect more than one object of a given colour but this was not useful for the scope of this project: only one ball of each color exists.

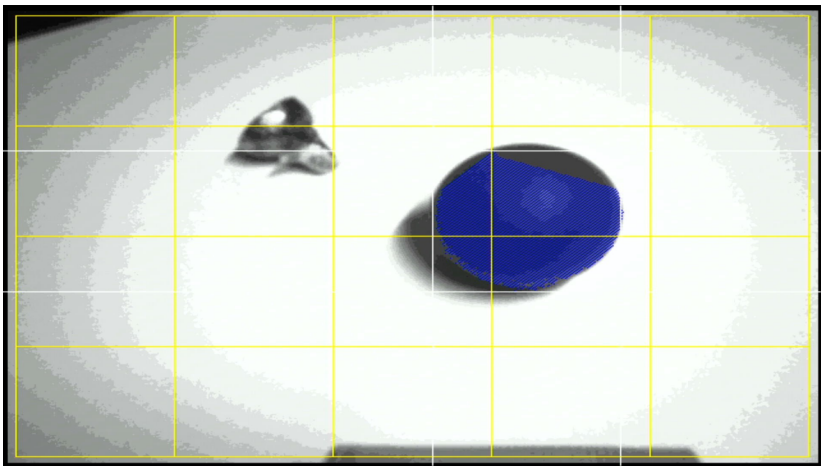


Figure 18: Example of the grid system solving the problem shown in fig. 17

The grid system is good at ignoring noise as shown in fig. 18; even a large disturbance can be ignored. Additionally, because there is no storing of pixels for later analysis, everything was running as each pixel was being read so the tracking of motion of the balls is very smooth. Furthermore, since the detection is using a bounding box to represent an area, and the objects detected are balls, which appear as circles on the camera, the radius of the objects can be approximated thought the equation $(width + height)/4$. This can be used directly to approximate the distance of a ball from the camera.

Now that the grid system was in place, several grids could be allocated. One for each object requiring detection, and to determine exactly which ball was the true blob being detected, the comparison was set up to loop through each detector for each grid.

After testing, some bugs still remained. For example, red balls under certain lighting conditions would display a violet outline. This would be detected as a perfectly valid blob by the detector. This would also happen with green noise that would occur in low light conditions with the camera. These problems occurred because the violet outline was one connected line, which meant all the pixels were close and thus in the valid range. The green noise was sparse but close enough to cause the same problem. Both of these were solved with a similar solution: each blob would keep track of how many pixels were included in the blobs region. This would then be used as part of the comparison stage to rule out areas with too low pixel density. Now using radius, pixel count, and the grid system combined, the detector provided enough accuracy that even large amounts of noise could be dealt with.

7.6.2 Testing Methodology

The basic blob detector was the first system designed and, naively, it was done directly in Verilog. This lead to a lot of hours of bug fixing which could have been avoided if a testing methodology, like that in the colour detector, were used. That said, the testing was mainly done by writing the code in Verilog, then first analysing it in a separate compiler for syntax errors, and then compiling it in Quartus. Initially, rather than using the actual coloured balls, small visual programs were written in JavaScript. This allowed for precise tuning of colours. This was required as the colour detection was only checking to see if the colours were pure. Pure red would be close to #ff0000, pure blue #0000ff, etc. Using this system allowed to test more interesting combinations of colours easily, such as creating specs of background colours to test the grid system, and easily moving the balls around in patterns to test the tracking.

Many visual helpers were added as debugging features. A few of these included: using the switches to enable and disable certain debugging features, adding a grid overlay to properly see the position of each detector, having modes for displaying only the largest blob and it's appropriate colour on a gray-scale background, showing all colour detected pixels in different modes, and a few others. Additionally Quartus' Signal Tap Analyzer was a helpful tool for reading the inputs and outputs of particular Blob detectors. This tool allows for reading the value of certain pins at run time to observe how the components are interacting.

After combining the blob detector with the colour detection system, ball detection was tested using different lighting conditions. This revealed some of the bugs described above, such as with the violet outline red balls sometimes had, or the green tinting the camera had in lower light conditions. This led directly to improvements.

In the detector there were a few parameters which needed tuning, the three most impact-full ones were the distance radius detection, the distance threshold for accepting a new pixel, and the grid size and position. The radius detection is about deciding how to determine the radius of the blob, this would have to be an approximation since the blob itself occupied rectangular bounds and not circular ones. The rectangular bounds were chosen because it provided simple calculations for expanding the area. Three possible systems where attempted for approximating the

radius of the bounds: the first was using the same formula as for the radius which would be output as the radius of the ball, $(width + height)/4$. The second was to divide the max of the two dimensions by two $max(width, height)/2$. The third was to use the distance from the center to the corner $\sqrt{width^2 + height^2}$ (we would not actually use the "sqrt" but use compare against the square of the distance threshold). In the end, the last one was chosen as it provided the most accurate results. The first sometimes underestimated the size of the blob and missed many pixels. The second caused thin lines of noise to become large blobs. To further optimize, since the width and height of bounds should end up being approximately the same, the expression was simplified to $widht * height$. This reduces the number of multiplications and hence saves on area.

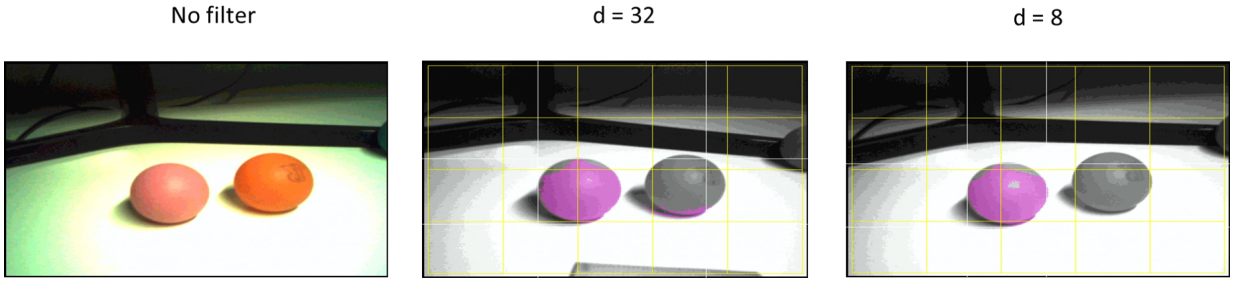


Figure 19: Blob detector running using different distance thresholds for inclusion in the blob

The second parameter which needed tuning was the distance threshold, which defines how far away from the radius can the distance be and still be considered part of the blob. This was tested by setting a value, and using the switches to change that value as to obtain a distance which was not too large as to include noise, but not too small as to have problems with small gaps between pixels. In the end, it the distance was selected to be 8 pixels. In fig. 19 it can be observed how increasing the distance too much can cause some erroneous pixels to be picked up as part of the same object: when the two objects are detected separately the FPGA can detect that the pink pixels on the red ball are noise and discards them, but if they are considered as a single unit it cannot do that.

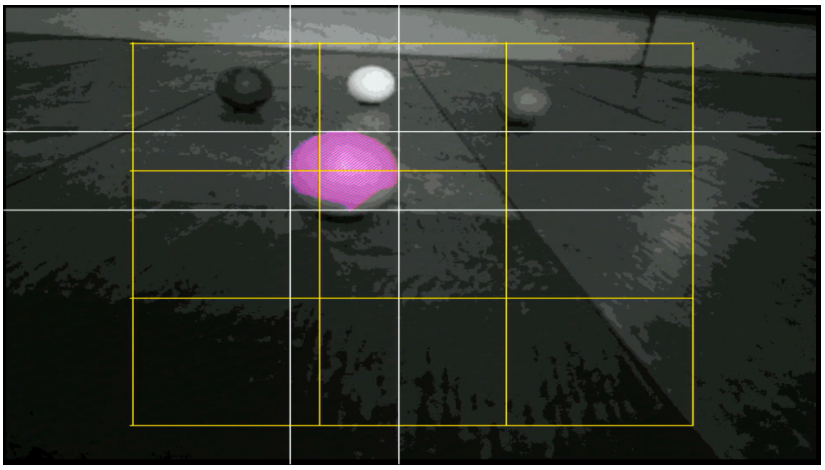


Figure 20: Final grid layout for the rover

For the grid size and position, the tuning was done in testing with the entire built system. The grid was made as small as possible as to avoid having to analyse areas of the vision which were redundant to the system. The size was made as to be accurate enough to detect balls, but not too large as to use unnecessary hardware. This required a lot of trial and error, leading to the final grid as shown in fig. 20. It has a dimension of 3x3 and only occupies the center most 440x400 pixels. Knocking off 200 pixels on the width was done because the way the camera was set up on the rover meant that the data it would need to observe would most likely be directly in front of it. This way we could have a more packed grid in the area that mattered without having to spend resources on adding more detectors for places where detection would never occur. For the height, some margins were added to avoid some of the noise that occurs on the outer most parts of the camera. It was found that the camera had strong green tints on the outermost parts of the image, and although many tools were in place to mitigate the effects of this, the best solution was to simply ignore these areas.

7.6.3 Verilog Implementation

As with the colour detection module, this module uses multiplication, which is a slow operation, even with the dedicated hardware. Hence, the multiplication step has been pipelined into three stages. First the input is stored, then it is multiplied, and then comparisons are made which either increase or do not increase the blob's bounds.

The module is shown in code 10. The inputs of the module are "clk" and "reset_n", an additional reset for resetting at the beginning of every frame. The x and y value of the current pixel data, whether the pixel was or was not detected as the appropriate colour. For outputs, "valid" contains data about whether a pixel was included in a blobs detected region. Additionally the radius, and pixel count were output, along side the bounding box information. In a fully optimised system, the only necessary outputs are "rad" and "pixel_count" which are used in the comparison stage to determine the largest blobs. The other outputs are for the visual debugging.

```
module SXPP_BLOB_DETECT (
    // global clock & reset
    clk,
    reset_n,

    // setup
    reset,

    // input pixel data
    x_in, y_in,
    is_valid_color

    // output data
    valid,

    rad,

    pixel_count,
    minx, maxx, miny, maxy
);

// inputs
input logic clk;
input logic reset_n;

input logic reset;

input logic[10:0] x_in, y_in;
input logic is_valid_color;

// outputs
output logic valid;

output logic[31:0] rad;
output logic [31:0] pixel_count;

output logic[10:0] minx, maxx, miny, maxy;
```

Code 10: Blob Detector Module Instantiation

7.7 Communication with ESP

For the communication with the ESP the SPI pins where used, the connection process and reasoning for the choice of using SPI interface is described in more detail in section 4.7.

To get the data from the EEE_IMPROC module into the Nios 2 processor the pre-existing communication module, FIFO, was expanded to be able to send more data at a time, as the defaults were not enough. Once the data about the detected ball was in the Nios 2, some light processing was done to check if the radius of the current detected blob was above a minimum threshold, this was the distance detection. For balls, the radius threshold was set to 30px and the pixel threshold was set to 1000px. For unknown objects only the radius threshold was used at 200px. Additionally, a check was put in place to ensure that the currently received detection occurred for a few frames in a row, and was not just a random spike due to noise. Once a detection was deemed to be valid for long enough, it would be sent via the SPI interface. A system kept track of the last message sent to ensure that the same colour could not be sent more than one time consecutively. This prevents problems such as overloading the ESP32, and accidentally misinterpreting the same ball as being in multiple places at once.

Template Design

Revision Name	DE10_LITE_D8M_VIP
Top-level Entity Name	DE10_LITE_D8M_VIP
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	10,214 / 49,760 (21 %)
Total combinational functions	8,603 / 49,760 (17 %)
Dedicated logic registers	6,532 / 49,760 (13 %)
Total registers	6599
Total pins	171 / 360 (48 %)
Total virtual pins	0
Total memory bits	1,341,624 / 1,677,312 (80 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Final Design

Revision Name	DE10_LITE_D8M_VIP
Top-level Entity Name	DE10_LITE_D8M_VIP
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	37,077 / 49,760 (75 %)
Total combinational functions	35,321 / 49,760 (71 %)
Dedicated logic registers	15,176 / 49,760 (30 %)
Total registers	15243
Total pins	171 / 360 (48 %)
Total virtual pins	0
Total memory bits	1,093,114 / 1,677,312 (65 %)
Embedded Multiplier 9-bit elements	288 / 288 (100 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 21: Comparison between template and final design of the vision module

7.8 Final Design

With all the design decisions taken and implemented, everything was put together. Overall, quite a bit of logic was added to the FPGA. Compared to the template provided the total logic elements used increased from 10214 to 37077. All of the data relevant to the difference between the template and final design can be observed in fig. 21.

7.9 Future Work

The current system works well in lit rooms, and suffers a little as the light becomes very intense, or if the light becomes very low. In extremely bright conditions the camera reads everything as white, and it is difficult to detect the saturated colours. In dark environments there exists a lot of blue and green noise, and the dark colours are too far away from the pure colours to be picked up. Increasing the range of lighting condition under which the vision module works would be one of the most important future developments.

The module when in its working range of lighting conditions is accurate both in detecting the ball's exact position as well as its radius. Because the balls are of a known size, these accurate values could be processed to determine the balls exact position in 3D space. This would allow for more detailed maps to be generated. A lot of work would be required from many of the other modules to support such a change.

One idea which was discussed but never managed to get fully implemented was the ability to send pictures from the FPGA to the server when a ball was detected. This would allow the ground control team of the rover to see exactly what the rover detected. This feature was partially supported by the Control module, but was never developed in the Vision and Command modules, it would probably be the first extension tackled if more time was provided. See section 4.11 for more details regarding this extension.

8 Integration Module

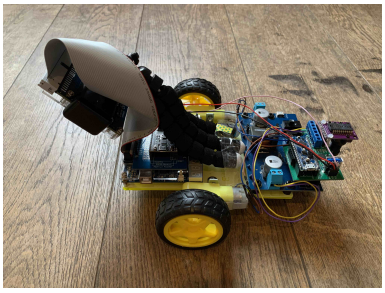
8.1 Role of Integration in Project

The role of the Integration module was to ensure a smooth operation of the rover through enabling easy testing to other group members on the fully assembled vehicle. This proved to be more difficult than expected if done fully remotely so towards the end, we opted out to meet in person and test the rover that way (in compliance with COVID-19 restrictions at the time). As such, there are no major implementation details to be covered in this section that have not already been described in the previous ones.

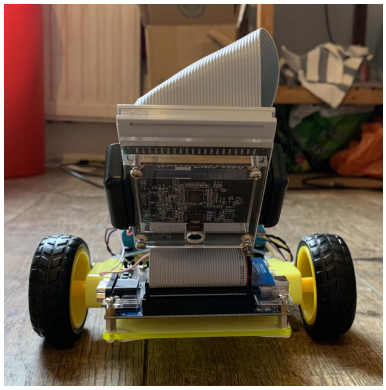
8.2 Rover Modifications

Only two small modifications were done to improve the rover's performance: adding a camera mount and a counterweight. Fig. 22 overleaf shows these modifications. During testing we discovered that the default position of the camera was inadequate as it meant that the size of the squares on the map (see section 3.2.1) was over 80 cm - far too large for a practical application of effective ball detection. One way of decreasing this size was to enable the rover to see objects right in front of it which would allow the square size to be exactly the same as the rover's dimensions. This was achieved by utilising a simple phone stand to hold the camera in a position where it would face slightly down (shown in fig. 22b, NB: the camera position is slightly different in the video to allow for a demonstration of the ball detection algorithm).

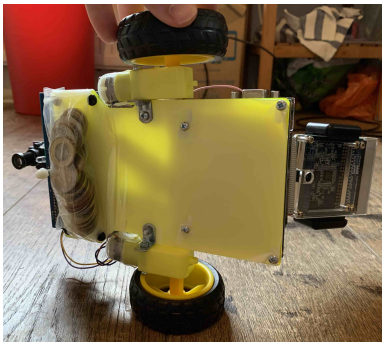
This modification came with a disadvantage though - the rover was now very front-heavy and would easily tilt forwards during backwards motion. The tilt meant that the optical flow sensor was lifted too high and the rover could no longer measure distances accurately. To mitigate this, we needed to add a counterweight at the back of the rover. This was done using a very simple solution - a bag full of coins - which worked perfectly. They were the most accessible small and heavy objects at the time, but a more professional solution could be developed if required.



(a) Rover as seen from the top



(b) Rover as seen from the front



(c) Rover as seen from below

Figure 22: Pictures of assembled rover

8.3 Testing Methodology

The testing approach used in the project was very much one of small scale tests - lots of refining code on the go and retesting as well as extensive tests of each module alone before integration (for details, see the other modules' sections on "Testing Methodology"). This worked effectively due to the fairly small scale of the project and the ease with which the code could be edited on the fly. Utilizing MS Teams and screen sharing, we were able to have the person who originally wrote the code monitor and advise on any needed modifications and aid with debugging.

The module that required the most editing on the go was the Drive. This can be attributed to both the sensitivity of this module due to the rover's differential drive, but also the fact that the components available to the Drive module leader excluded the FPGA and camera at the front. This discrepancy meant that the rover's weight balance was slightly different which affected the Drive's performance. The other modules (Command, Control and Vision) could be tested beforehand with relative ease without the need to run everything on a full rover.

Nevertheless, the role of the Integration module leader was to aid the others in any design problems they might be facing as well as ensuring that all modules work together well. To achieve this, regular meetings were held with other module leaders to test individual functionality before trialing everything together on the complete rover (see section 2.4). This saved a lot of time that the team would have otherwise spent debugging when the whole rover was assembled. In the end, the results of the final tests can be seen in the project video.

8.4 Adding Energy to Rover and Future Work

In the beginning, we established that the Energy module could be integrated into the rover design if it provided a constant 5 V output at a maximum of 3 A current being drawn from it. This meant that no physical testing needed to be done on the assembled rover when it came to adding the batteries to it as long as the condition above was met (which could be verified using a simple 1.67 Ω resistor). We initially planned on filming the rover being powered by the batteries, but due to the ban on working on them that appeared towards the end of the project, this was not carried out. The ban also meant that the Energy module was not finished completely, but this is described in the form submitted alongside this report.

A method of connecting the batteries and solar panels to the rover was theorised and can be seen in fig. 23. This is only a simplified version meant to act as an illustration of our ideas. The batteries would be split into two sections: a main bank (3 cells) and a secondary bank (2 cells). The "SMPS Cascade or LT3652" block has two possible implementations:

1. Two SMPS circuits - a boost and a buck connected in series. The first boost SMPS could be used to implement the MPPT algorithm, however, because of the changing duty cycle, the output voltage of the converter would also change. This means that another buck SMPS would be needed to take the varying output of the boost and convert it to a steady current to charge the batteries. A potential divider (in parallel with panel) and a sensing resistor (in series with panel) would need to be added at the input of the boost to enable reading both the current and voltage of the solar panel and allow for computation of the panel's output power. The buck SMPS would have a closed-loop controller to maintain a constant current at its output.

2. An LT3652 chip [51] could be with the associated circuit shown in the datasheet. This is a dedicated chip used to charge batteries from a solar panel that would work well in this proposed circuit given its maximum ratings.

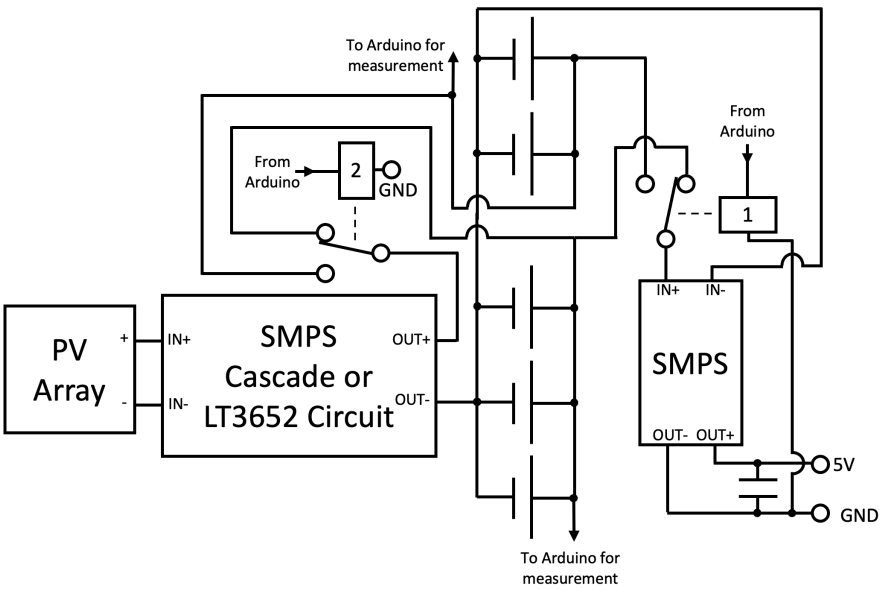


Figure 23: A circuit schematic of the proposed Energy module assembly

In the event of a complete discharge of the main bank (monitored by the Arduino as a fall below the minimum battery voltage), the Arduino would switch relay 1 to use the secondary bank. This configuration means that if a short across the either bank occurs and the batteries are inoperable, the other bank can be used - it adds redundancy to the rover which is necessary in an extremely remote environment where no repairs can be made on the go such as Mars. Relay 2 is switched as well when a short occurs to protect the charging circuit and PV array. A capacitor on the output with a sufficiently large value would preserve the supply of power to the brains of the rover during this switching.

The rover could also have an emergency backup mode, in which the circuits would be optimised to draw as little power as possible by adding an optoisolator for the power supply to Vision, essentially restricting supply only to the Arduino and the ESP32 for communication and recharging (a message about this could be sent to ground control).

References

[1] Wikipedia, *Rover (space exploration)*, English, Jun. 2021. [Online]. Available: https://en.wikipedia.org/wiki/Rover_(space_exploration) (visited on 12/06/2021).

[2] *Types of Database Management Systems*. [Online]. Available: https://www.c-sharpcorner.com/uploadfile/65fc13/types-of-database-management-systems/ (visited on 10/06/2021).

[3] *Mozdevs/gamedev-js-tiles*, original-date: 2015-09-28T13:46:48Z, Jun. 2021. [Online]. Available: https://github.com/mozdevs/gamedev-js-tiles (visited on 14/06/2021).

[4] *Relational Databases - How to Choose |// SQL Tutorial |// SQL for Beginners*, en-GB. [Online]. Available: https://www.youtube.com/watch?v=WzfDLqt-WIg (visited on 13/06/2021).

[5] *What is Normalization in SQL? 1NF, 2NF, 3NF, BCNF Database (DBMS) Example*. [Online]. Available: https://www.guru99.com/database-normalization.html (visited on 13/06/2021).

[6] mars.nasa.gov, *Moving around Mars*. [Online]. Available: https://mars.nasa.gov/mer/mission/timeline/surfaceops/navigation/ (visited on 13/06/2021).

[7] *SQLite Home Page*. [Online]. Available: https://www.sqlite.org/index.html (visited on 14/06/2021).

[8] *The Go Programming Language*. [Online]. Available: https://golang.org/ (visited on 14/06/2021).

[9] IBricchi, *IBricchi/GamblingFPGAs*, original-date: 2021-03-01T14:46:34Z, Jun. 2021. [Online]. Available: https://github.com/IBricchi/GamblingFPGAs (visited on 15/06/2021).

[10] Wikipedia, *Atomicity (database systems)*, English, Mar. 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Atomicity_(database_systems)&oldid=1013005337 (visited on 11/06/2021).

[11] *Authorization - HTTP | MDN*, en-US. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Authorization (visited on 14/06/2021).

[12] ——, *Bcrypt*, en, Page Version ID: 1025246756, May 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bcrypt&oldid=1025246756 (visited on 13/06/2021).

[13] ——, *A* search algorithm*, English, Page Version ID: 1025834072, May 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=A*_search_algorithm&oldid=1025834072 (visited on 08/06/2021).

[14] ——, *Admissible heuristic*, English, Page Version ID: 1019263756, Apr. 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Admissible_heuristic&oldid=1019263756 (visited on 08/06/2021).

[15] ——, *Consistent heuristic*, English, Page Version ID: 1026861844, Jun. 2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Consistent_heuristic&oldid=1026861844 (visited on 08/06/2021).

[16] *Algoexpert*, English. [Online]. Available: https://www.algoexpert.io (visited on 08/06/2021).

[17] Morkelon, *Why idf vs Arduino IDE?*, Reddit Post, Oct. 2019. [Online]. Available: www.reddit.com/r/esp32/comments/dk3rsn/why_idf_vs_arduino_ide/ (visited on 08/06/2021).

[18] N. Kolban, *Kolban's book on ESP32*. Sep. 2018. [Online]. Available: https://leanpub.com/kolban-ESP32.

[19] *API Reference - ESP32 - — ESP-IDF Programming Guide latest documentation*. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/index.html (visited on 08/06/2021).

[20] G. M. B. Oliveira, D. C. M. Costa, R. J. B. V. M. Cavalcanti, J. P. P. Oliveira, D. R. C. Silva, M. B. Nogueira and M. C. Rodrigues, 'Comparison between mqtt and websocket protocols for iot applications using esp8266', in *2018 Workshop on Metrology for Industry 4.0 and IoT*, 2018, pp. 236–241. DOI: 10.1109/METROI4.2018.8428348.

[21] *FAQ*. [Online]. Available: https://mqtt.org/faq/ (visited on 08/06/2021).

[22] cdaviddav, *UART Tutorial for Arduino and ESP8266*, en-US, Jan. 2020. [Online]. Available: https://diyi0t.com/uart-tutorial-for-arduino-and-esp8266/ (visited on 10/06/2021).

[23] *UART vs I2C vs SPI – Communication Protocols and Uses*, en-US, Sep. 2019. [Online]. Available: /blog/2019/09/25/uart-vs-i2c-vs-spi-communication-protocols-and-uses/ (visited on 10/06/2021).

[24] *UART vs SPI vs I2C | Difference between UART,SPI and I2C*. [Online]. Available: https://www.rfwireless-world.com/Terminology/UART-vs-SPI-vs-I2C.html (visited on 10/06/2021).

[25] Microchip Technology Inc., *ATmega809/1609/3209/4809 – 48-pin Data Sheet*, English, 2019. [Online]. Available: https://content.arduino.cc/assets/Nano-Every_processor-48-pin-Data-Sheet-megaAVR-0-series-DS40002016B.pdf.

[26] *Serial Input Basics - updated*, en, Apr. 2016. [Online]. Available: https://forum.arduino.cc/t/serial-input-basics-updated/382007/2 (visited on 11/06/2021).

[27] Altera Corporation, 'SPI Core', English, in *Quartus II Handbook*, vol. 5, May 2007. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/qts/n2cpu_nii51011.pdf.

[28] S. Hanselman, *Connect to a device over Serial COM Port on Windows 10 with WSL1 TTY devices with Windows Terminal and minicom*, Blog. [Online]. Available: https://www.hanselman.com/blog/connect-to-a-device-over-serial-com-port-on-windows-10-with-wsl1-tty-devices-with-windows-terminal-and-minicom.

[29] Steve, *MQTT Client and Mosquitto Broker Message Restrictions*, en-US, Jan. 2020. [Online]. Available: http://www.steves-internet-guide.com/mqtt-broker-message-restrictions/ (visited on 11/06/2021).

[30] *How to write your own arduino libraries – simple arduino libraries for beginners*, English. [Online]. Available: https://www.instructables.com/Simple-Arduino-Libraries-for-Beginners/ (visited on 28/05/2021).

[31] *C++ enumeration*, English. [Online]. Available: https://www.programiz.com/cpp-programming/enumeration (visited on 31/05/2021).

[32] T. Green, *Power electronics and power systems module notes (elec50012)*, 2020.

[33] EmbedGyan, *Battery management system : How to measure state-of-charge*. [Online]. Available: https://embedgyan.wordpress.com/2020/06/25/battery-management-system-how-to-measure-state-of-charge/ (visited on 14/06/2021).

[34] user5139637 (https://electronics.stackexchange.com/users/82414/user5139637), *Area under voltage vs time discharge curve*, Electrical Engineering Stack Exchange, URL:https://electronics.stackexchange.com/q/207774 (version: 2017-04-12). eprint: https://electronics.stackexchange.com/q/207774. [Online]. Available: https://electronics.stackexchange.com/q/207774 (visited on 10/06/2021).

[35] W. C. Ltd, *Battery testing*. [Online]. Available: https://mpoweruk.com/testing.htm#impedance (visited on 14/06/2021).

[36] T.-H. Cho, H.-R. Hwang, J.-H. Lee and I.-S. Lee, 'Comparison of intelligent methods of soc estimation for battery of photovoltaic system', *(IJACSA) International Journal of Advanced Computer Science and Applications*, vol. 9, no. 9, 2018. [Online]. Available: https://thesai.org/Downloads/Volume9No9/Paper_7-Comparison_of_Intelligent_Methods.pdf#:~:text=Open%20Circuit%20Voltage%20Method%20The%20open%20circuit%20voltage,data%20must%20be%20constructed%20through%20SOC-OCV%20relationship%20experiments..

[37] Electricalvoice, *State of charge (soc) & state of health (soh)*, Oct. 2017. [Online]. Available: https://electricalvoice.com/state-of-charge-soc-state-of-health-soh/ (visited on 11/06/2021).

[38] Ashish, *What are the different methods to estimate the state of charge of batteries?*, Nov. 2019. [Online]. Available: https://www.scienceabc.com/innovation/what-are-the-different-methods-to-estimate-the-state-of-charge-of-batteries.html (visited on 12/06/2021).

[39] W.-Y. Chang, *The state of charge estimating methods for battery: A review*. [Online]. Available: https://www.hindawi.com/journals/isrn/2013/953792/ (visited on 12/06/2021).

[40] M. A. Roscher and D. U. Sauer, 'Dynamic electric behavior and open-circuit-voltage modeling of lifepo4-based lithium ion secondary batteries', *Journal of Power Sources*, vol. 196, no. 1, pp. 331–336, 2011, ISSN: 0378-7753. DOI: https://doi.org/10.1016/j.jpowsour.2010.06.098. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378775310010852.

[41] A. Mangler, O. Kanoun and T. Günther, *Determining the state of health of batteries quickly and precisely*, Sep. 2017. [Online]. Available: https://www.rutronik.com/article/detail/News/determining-the-state-of-health-of-batteries-quickly-and-precisely/ (visited on 09/06/2021).

[42] PowerStream Technology, *Battery impedance and resistance*, Oct. 2019. [Online]. Available: https://www.powerstream.com/internal-resistance.htm (visited on 12/06/2021).

[43] A. Raj, *Cell balancing techniques and how to use them*, Feb. 2019. [Online]. Available: https://circuitdigest.com/article/cell-balancing-techniques-and-how-to-use-them#:~:text=Cell%20Balancing%20Techniques%20and%20How%20to%20Use%20Them,3%20Lossless%20balancing.%20...%204%20Redox%20Shuttle.%20 (visited on 10/06/2021).

[44] E. Stott, *Edstott/EEE2Rover*, English, original-date: 2021-05-06T08:55:49Z, Jun. 2021. [Online]. Available: https://github.com/edstott/EEE2Rover.

[45] Altera, *Video and Image Processing Suite UserGuide*, English, Oct. 2016. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/archives/ug-vip-16.1.pdf.

[46] OmniVision, *OV8865 Datasheet (rev 1D)*, English, Feb. 2014.

[47] R. Thiadmer, *Colour metric*, English. [Online]. Available: https://www.compuphase.com/cmetric.htm (visited on 20/05/2021).

[48] S. Leszek, *C++ - Algorithm to convert RGB to HSV and HSV to RGB in range 0-255 for both*, English, Nov. 2019. [Online]. Available: https://stackoverflow.com/questions/3018313/algorithm-to-convert-rgb-to-hsv-and-hsv-to-rgb-in-range-0-255-for-both (visited on 20/05/2021).

[49] *Processing/p5.js*, original-date: 2013-02-26T03:14:47Z. [Online]. Available: https://github.com/processing/p5.js.

[50] *11.7: Computer Vision: Blob Detection - Processing Tutorial*, English. [Online]. Available: https://www.youtube.com/watch?v=ce-2l2wRqO8 (visited on 06/06/2021).

[51] L. Technology, *Lt3652 power tracking 2a batterycharger for solar power datasheet*. [Online]. Available: https://www.analog.com/media/en/technical-documentation/data-sheets/3652fe.pdf.

A Appendix: Meeting Minutes

A.1 12th May

TO DO:

- KN, BS, SA to send GitHub usernames to IB.
- Go over report specs when they come out.
- Everyone do general research in branch of project they are assigned.

Source Control
Will use GitHub with one repo and multiple folders (one for each branch).
Boxes
Only BS got his box. Everyone else didn't get one yet.
Schedule
Don't have specifications yet so cannot start planning project timeline. BS suggests waiting for the weekend until everyone got their things and we received specifications. 4 weeks to complete project so spend this week researching and initial planning.
Specification
We still don't know what the specs are. Can start brainstorming ideas (streaming camera to app etc.). NP suggests having a website is easier to control rover. May later on develop an app afterwards if we have time. BS can start working on his part because he already got his box.
Communication
Will use Teams as primary measure of contact. WhatsApp as a secondary. NP suggests we need to keep detailed minutes for the report. 4/5 people are in London so can potentially meet up in person at some point in the future (if restrictions allow).
Report
Need to choose between LaTeX and Word for report. Still a bit early to decide.

A.2 17th May

TO DO:

- Have a basic functionality of rover by Monday 24/05.
  - BR to implement basic drive functionality.
  - SA to fully set up energy supply to motors and other components.
  - BS and NP to polish website and figure out communication.
- KN will translate requirements into one document when they come out.
- Everyone to begin filling out their module timelines on the chart.

Integration
KN to (maybe) call NP and discuss connectivity between modules. KN working on assembling rover (done by the end of today).
Drive
BR will implement basic movements. Depending on accuracy of motors, we can implement a 2D map on the command side (website), but this is a stretch goal.
Energy
SA went over the files provided. Will start working through it today. As a stretch goal we could implement a solar panel interface.
Command and Control
BS set up website that is now up hosted externally. Need to figure out connectivity between backend and rover. Need to decide what information needs to be transferred between rover and command website.
Vision
IB has been playing around with examples from files given.
Requirements
Don't really know what requirements are there. Perhaps more clear requirements will be released tomorrow. Should work towards a functional rover by next Monday (24/05). Implement basic commands and obstacle avoidance by then.
Boxes
Everyone received their boxes.

A.3 24th May

TO DO:

- IB to improve colour detection for other balls and communication out to the ESP32.
- BR to work on implementing further commands.
- KN to help SA with the SMPS later on today.
- BS and NP will work on webpage further to set up communication with rover.

Command
BS polished up the webpage and set up the basic communication framework with the rover. Security needs to still be added for the communication (already set up from rover to the server but still needs to set up from server to webpage). Perhaps think about implementing a log in option as security.
Drive
BR gauged the accuracy of his components, namely the optical flow sensor and improved it. All low-level rover functions are implemented (change speed, turn etc.). BR is now working on more advanced functions like move x cm forward/backward.
The rover can rotate but not by an angle yet. Next steps are controlling based on speed i.e., move at x m/s and the circuit figures out what voltage to apply to the motors. BR is also thinking of implementing an instruction buffer, but it still needs to be decided whether we chose to have the buffer on the ESP32 or the Arduino. So far BR has implemented move x meters or for x seconds or turn at x degrees or for x seconds.
Control
NP defined the communication protocol between ESP32 and Arduino. So far there is not much to do until other modules have completed their implementation (namely Vision, Drive and Energy).
Energy
KN to help SA with the SMPS later on. SA has been working on setting up solar panels (wired in series). The current step is implementing Arduino code to control the SMPS.
Need to combine codes from Drive and Energy on the Arduino at the end so both modules need to look into condensing code into functions and including them into a large file as libraries (.h files). So far BR and SA will focus on implementing functionality and combine the codes into one file at the end.
Vision
IB managed to implement an algorithm that detects the largest ball of a single colour for now. It should be fairly simple to have all the colour detections running all the time. The camera outputs 8-bit depth but IB is only able to retrieve 4-bits so far. It may be possible to get the full camera quality but that is an optional functionality that can be looked at later on.

A.4 31st May

Control
NP added SPI connection to FPGA from ESP32 and has also been helping other group members with communication to the ESP32. Currently working with IB to figure out if images can be sent back to command periodically.
Energy
SA ran tests on all batteries, and it turns out that they last for quite a while. SA also looked into SoC and SoH algorithms and will aim to get something together soon. A theoretical approach is to have to battery banks that switch between supplying the rover with energy and charging from the solar panels. SA started to record the Energy video.
Vision
IB has been trying to figure out a different colour detection method but that didn't provide results, so he refined the current algorithm to be quite accurate. The vision can also predict how far the balls are from the rover fairly accurately.
Sending images to the command side may be a bit difficult but NP is optimistic that it may be possible. This is a side objective and will not be crucial. IB will focus on obstacle recognition.
Command
BS has been working on the website and polishing it up. Basic map has been implemented to display a 2D map of the rover's surroundings. BS may start setting up image receiving from the rover regardless of if we do it or not as it shouldn't take too long and may be useful if we do figure out image transmission.
Drive
BR finished the basics of the drive. The rover is now able to move based on a range of input commands and can turn as well. The current control mechanism overshoots the desired position which is just a simple "if not at position, then move" controller but can be replaced by a PID controller which is the next step to refine the motion of the controller.
Testing
Aim to run a full test of rover tomorrow (Tuesday 01/06/2021) to put everything together and see the rover in action.

A.5 7th June

TO DO:

- Everyone to write your section of the report by Thursday 10th June.
- SA to finish implementing SoH algorithm.
- KN, SA, IB and NP to meet on Wednesday to film the video.
- KN to edit video (with help from BS if needed).

Velocity Control
It is a requirement for drive. The computation of velocity should be on the rover and the speed will be controlled via a proportional controller i.e. move faster if far away, move slower if closer.
Report
IB already wrote a draft of his section of the report for vision. Report length is max 40 pages. Should now aim to write individual sections by Thursday. IB set up an Overleaf document for the report where we can all collaborate on one LATEX document.
Video
IB, KN and NP will meet on Wednesday 9th June to record the video. Will probably be sped up to fit more content in.
Energy
SoH algorithm still needs to be implemented. SA is working on that. Will try to integrate the batteries with the rover alongside filming the video on Wednesday.

A.6 10th June

Report
IB finished (need to put figures in). NP done most but still things to add. BS done most but still has more to write about and research. BR stil has some sections to finish. SA is getting along with report writing. He will produce a block diagram for the energy module integration into the final rover. IB will add a diagram of the full structure of the rover in the first section of the report.
Video
KN, IB, and NP spent yesterday recording the video. KN is now editing it. Will produce outline of script and send to NP and BS once editing is done to record a voiceover.
Gantt Chart
Fill out chart with your final changes. Will be needed for final submission.
Oral
Schedule oral relatively early as some people are not available towards the end of next week.
IP Essay
Whoever is done first (probably IB and BR) will write that and add to report.

A.7 14th June

Video
KN completed the video with BS and NP's voiceover. Available on YouTube as unlisted in case the examiners need it in this format under https://youtu.be/T8w77XG1Lxw.
Report
Everyone finished their report sections except for SA. KN and IB will try to aid with putting all the necessary items into the report.
Oral
IB moved the oral to the afternoon on Friday to fit all group members.
IP Essay
BR finished it and advised everyone to include specific clauses in the code to comply with the IP lecture content.
Repo
IB and BR will make a readme file on there as a finishing touch.
End of Project
Great job everyone!