

ELEC50008 – Engineering Design Project

Mars Rover

15th June 2021

Group 20 - Wireless Access Point

Name	Username	CID
Adam Horsler	ah2719	01738592
Pranav Viswanathan	pv319	01744658
Mihai-Catalin Stretcu	ms5919	01784126
Youngmin Song	ys2319	01733645
Sungjun Han	sh1319	01708685
Yeong Kyun Han	ykh19	01702061

Word Count: 8643

Contents

Figures	3
Tables.....	3
Abstract	4
1. Introduction.....	5
2. Design	6
2.1 Design Process	6
2.2 Project management	7
2.3 Structural Design of the Mars Rover	7
2.4 Functional Design of the Mars Rover	9
2.5 Inter-module Communication	10
2.5.1 Vision and Control	10
2.5.2 Command and Control	11
2.5.3 Drive, Energy and Control.....	11
3. Implementation	12
3.1 Implementation of subsystems	12
3.1.1 Vision	12
3.1.2 Drive.....	15
3.1.3 Command	21
3.1.4 Energy:.....	27
3.1.5 Control	32
3.3 Evaluation of Design	34
3.4 Intellectual Property	35
4. Conclusion	36
5. Appendix.....	37
5.1 Appendix A: Gantt Chart.....	37
5.2 Appendix B: Github Repo	37
5.3 Appendix C: HSV Colour Detection Code Example	37
5.4 Appendix D: UART Communication Code Example	37
5.5 Appendix E: State Machine Code	38
5.6 Appendix F: Quartus Compilation Summary	38
5.7 Appendix G: MIT Licence Example	39

Figures

Figure 1 - Subsystem Technologies Overview	6
Figure 2 - Design Overview	8
Figure 3 – Structural Diagram	9
Figure 4 – Functional Diagram	10
Figure 5 -Receiving Information from Server	11
Figure 6 - Coloured Ping Pong Balls	14
Figure 7 – Instruction Processing	17
Figure 8 – H-bridge Circuit	18
Figure 9 – Initialisation and Setup Code of Pins	18
Figure 10 - Motor Direction and Rotation Control	19
Figure 11 – Modified Sampling Function	20
Figure 12 – Position Data from Optical Flow Sensor	21
Figure 13 – Server and Client Communication Diagram	22
Figure 14 - theoretical I-V graph of a PV panel	28
Figure 15 - experimental I-V graph of a PV panel	28
Figure 16 - Power-voltage graph of PV panel showing the MPP	29
Figure 17 - Flowchart of P&O algorithm	29
Figure 18 - Arduino code of P&O algorithm	30
Figure 19 - MPPT test with duty cycle against time	30
Figure 20 - State diagram of charging and discharging a single cell	31
Figure 21 – Transition Between Charging and Discharging	32
Figure 22 - Control Dataflow	33

Tables

Table 1 – Subsystem Roles Overview	6
Table 2 - Colour Conversion Boundaries	13
Table 3 - Drive Instructions	15
Table 4 - Instructions to Behaviour	16
Table 5 - Motor Speed Specifications	16
Table 6 – Azure Server Overview	22
Table 7 - TCP/IP Server Connections	23
Table 8 - Connection Test	23
Table 9 - Drive Instructions Movement Code	25
Table 10 - Testing Rover and User connection	26
Table 11 - ESP32 Connection	27
Table 12 - Control Communication	33
Table 13 - Control Functions	34

Abstract

This project aims to build a Mars Rover, which detects the obstacle in front of the vehicle, measures the distance, and avoids it. This rover should also be controllable through a remote server. To effectively establish all functionalities, the whole design is divided into six different subsystems. Command subsystem allows the user to send instructions to the main processor remotely. Drive subsystem receives the instruction and alters the speed or direction of motion. Meanwhile, Vision subsystem observes the front and detects any obstacles; it calculates the area and sends the data to Command to decide the next instruction. Energy subsystem is also required to analyze the battery life. Every data transfer is governed by Control subsystem as it utilizes various data protocols between interacting subsystems. Finally, all parts are assembled under Integration subsystem. The resulting rover enables easy control and swift transition of motion. It produces promising data from obstacle detection as well. However, there are still limitations of some functionalities not being fully implemented.

1. Introduction

Mars Rover is a useful vehicle that travels through harsh environment of Mars. It scans its surroundings and explores the area without on-site control. It can also move around to find a place to charge its PV panel for power supply. Even the motor control is commanded by the user staying on Earth. Every operation is done remotely and automatically, and such a feature makes the rover design complex and multidimensional. Therefore, it is important to split the overall structure into distinct subsystems.

The main objective of this project is to design a vehicle that can imitate the autonomous feature of Mars Rover. Six subsystems are introduced: Energy, Drive, Vision, Control, Command, and Integration. Members work in their own development suite to implement functionalities delegated to each subsystem. In the final design, the following core hardware are assembled to handle the rover's operation:

- Optical flow sensor to measure how far the rover has moved.
- D8M-GPIO camera and FPGA board to detect the obstacle in front.
- ESP32 processor to implement various protocols for communication.
- SMPS and Arduino board to supply power and run the motor.

To achieve wireless communication between the user and the rover, a remote server for ESP32 processor is set up to exchange instructions and status data. For transferring data between rover hardware, ESP32 simply uses UART by specifying RX and TX pins on neighboring boards.

This report begins from project planning stage, providing details about design process and project management. It then elaborates on design specification from structural and functional point of view. Interaction between subsystems follows. In the next section, the realized design is evaluated based on test results and functionality of subsystems. Finally, the outcome's limitation is addressed, and possible improvements are discussed.

2. Design

2.1 Design Process

The problem specification states that “the aim of this project is to design and build an autonomous rover system that could be used in a remote location without direct supervision.” To achieve this, the task is split into six subsystems: Integration, Control, Drive, Energy, Vision and Command. Figure 1 below illustrates each subsystem along with its associated technologies.

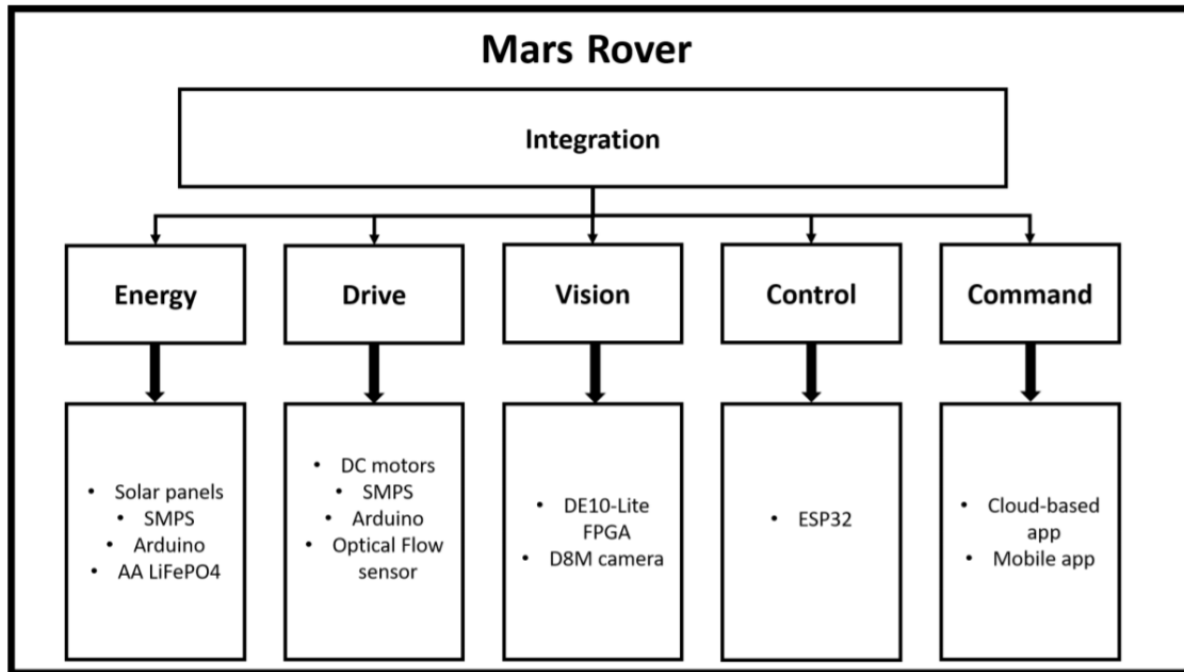


Figure 1 - Subsystem Technologies Overview

The role of each subsystem is shown in Table 1 below.

Subsystem	Role
Energy	Providing charged batteries to the rover
Drive	Movement of the rover
Vision	Detecting and avoiding obstacles
Control	Communication between the different subsystems and some basic calculations
Command	Remote control of the rover and offsite storage
Integration	Connection and integration of all the other modules

Table 1 – Subsystem Roles Overview

Splitting the design process into six key areas has various advantages over other approaches. Firstly, it allows for modular testing to ensure functional correctness before integration. This simplifies the testing and debugging process. Secondly, it defines roles and responsibilities of each group member, making teamwork more efficient and provides accountability. Lastly, due to the different expertise in the group, it allowed those that already have experience with certain technologies to focus on them.

To achieve each subsystems functional goals, work was primarily conducted separately, with weekly group meetings and ad-hoc meetings if further work between specific subsystems was required (for example, control and vision). This is further explained in the project management section.

2.2 Project management

As the rover is a multidisciplinary project, and not all team members are located within commuting distance of each other, project management was very important to achieving the project goals.

The main project management tool the team used was a Gantt Chart implemented in excel. Each subsystem had their own section, with target goals and milestones. A separate comments sheet was used to track group meeting notes as well as resolve problems/confusion when implementing features requiring multiple subsystems. The Gantt Chart can be found in Appendix A.

Group meetings occurred once a week, while subsystem-specific meetings were made on an ad-hoc basis. Thus, the comments section provided a way of keeping all members up to date with each other's subsystem even if they were not required in the ad-hoc meetings.

Microsoft Teams was used for file transfer between subsystems. For the vision and control subsystems, GitHub was used for version control and management of progress/problems through using different branches and commit logs. A link to the team's GitHub repositories can be found in Appendix B.

2.3 Structural Design of the Mars Rover

For the rover to function correctly, each subsystem must communicate with the control subsystem, which will then optionally modify and pass on the information to another subsystem. The entity-relationship diagram below shows a high-level view of the information sent between subsystems.

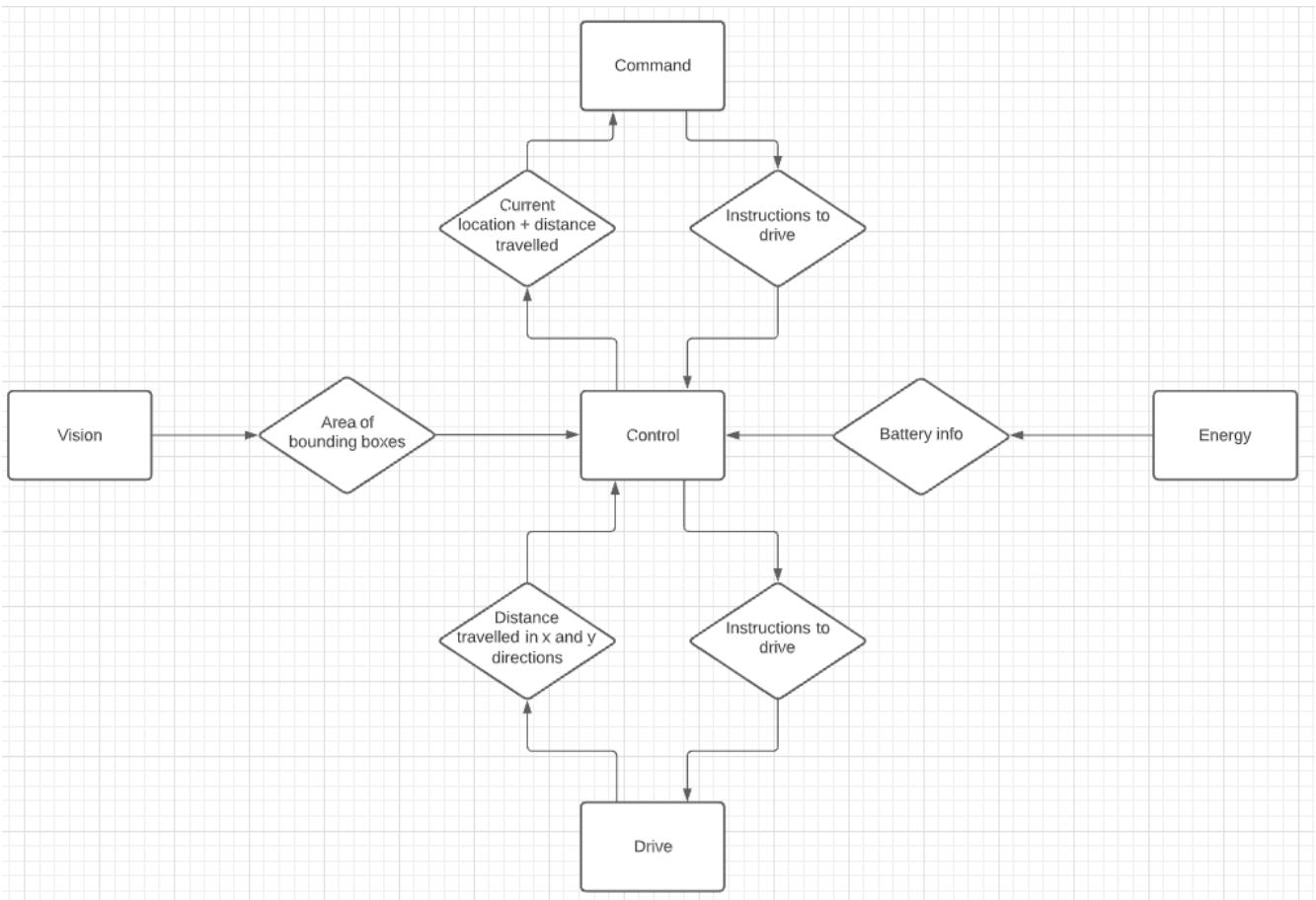


Figure 2 - Design Overview

From the diagram, Control is the central communication subsystem in the design. This is because the ESP32 can implement several communication protocols, for example SPI and UART. As it has an IP address, it can send and receive information from the Command module.

Vision detects the color of obstacles seen through the camera. It can locate the obstacle using bounding boxes and convert the data into a coordinate. The area covered by boxes is calculated as well. At every instant, Vision monitors the change of this bounded area to see if the rover is moving toward or away from the object. All coordinate data are sent to Control for processing via UART.

Energy manages battery information. It estimates the lifetime of the battery and sends it to Drive. Control also receives and passes the data to provide the user the current status of battery. Ideally, PV panels should be used to charge and supply power. However, in this project, the final outcome does not include physical implementation of PV panels. The rover is run by USB power supply, and the ideal battery configuration using PV panels is explored only during design stage.

Drive is a key to move the rover. Drive processes the instruction by receiving 8-bit serial data. The first two bits tell the vehicle to move forward or backward. The third and fourth bits manage rotational movement, clockwise or counterclockwise. The final four bits determine the speed of the motors.

Drive also uses an optical sensor to record the position and the distance travelled. These data are again delivered to Command to let user monitor the movement.

Command interacts with Control to communicate with all the other subsystems. The main information sent by Command is the instruction to drive the rover. Control receives the initial instruction via a remote server and passes the information to drive via UART. TX and RX pins are specified inside both the Arduino and ESP32. Meanwhile, battery status, obstacle detection data, and the rover's displacement data are all sent to Command through Control for monitoring purposes.

A summary of the structural design is shown below in Figure 3. This details the technical specifications used in the Rover for each subsystem.

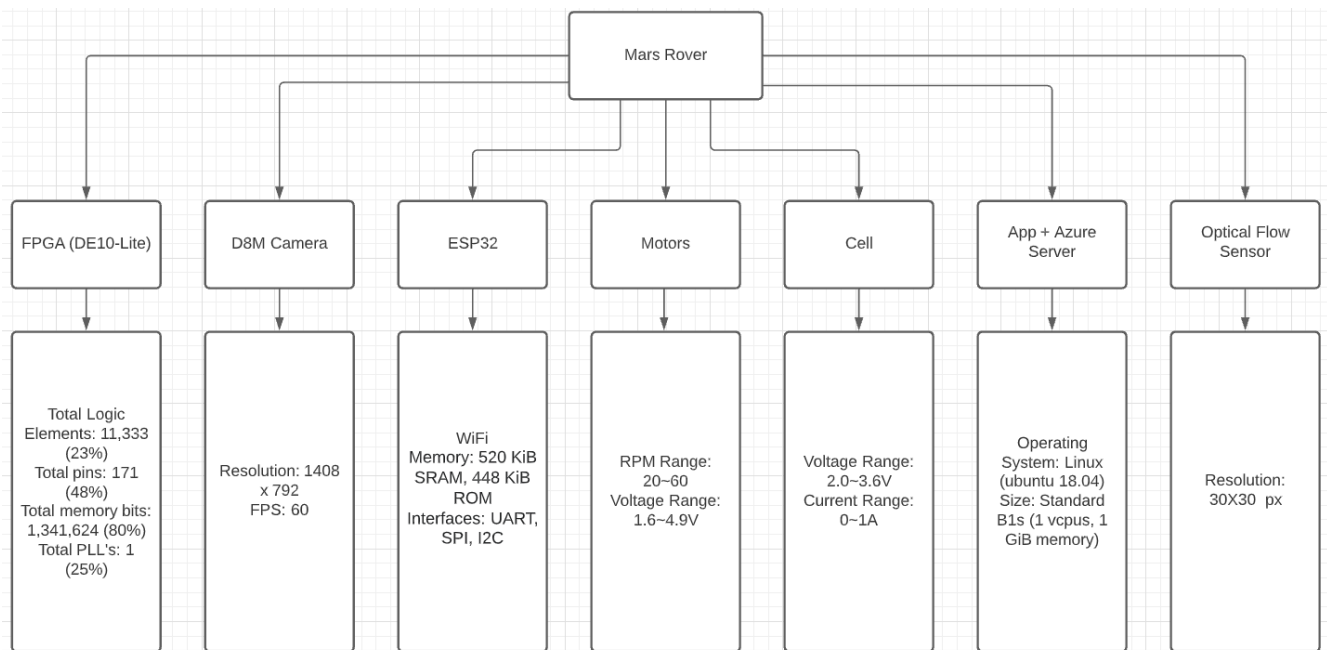


Figure 3 – Structural Diagram

2.4 Functional Design of the Mars Rover

A Use Diagram has been constructed to summarize some of the relationships between use cases, the external environment, and the Mars Rover. The diagram is shown below in Figure 4. On the left are the external actors that can impact the Rover, which include the App User, ping pong balls and the external environment in general. The App User can directly send movement commands to the Rover. The coloured ping pong balls are considered obstacles by the Rover and will need to be semi-autonomously avoided. Whether they are identified by the algorithm developed on the FPGA using the D8M Camera will be affected by external factors surrounding the rover, for example the brightness.

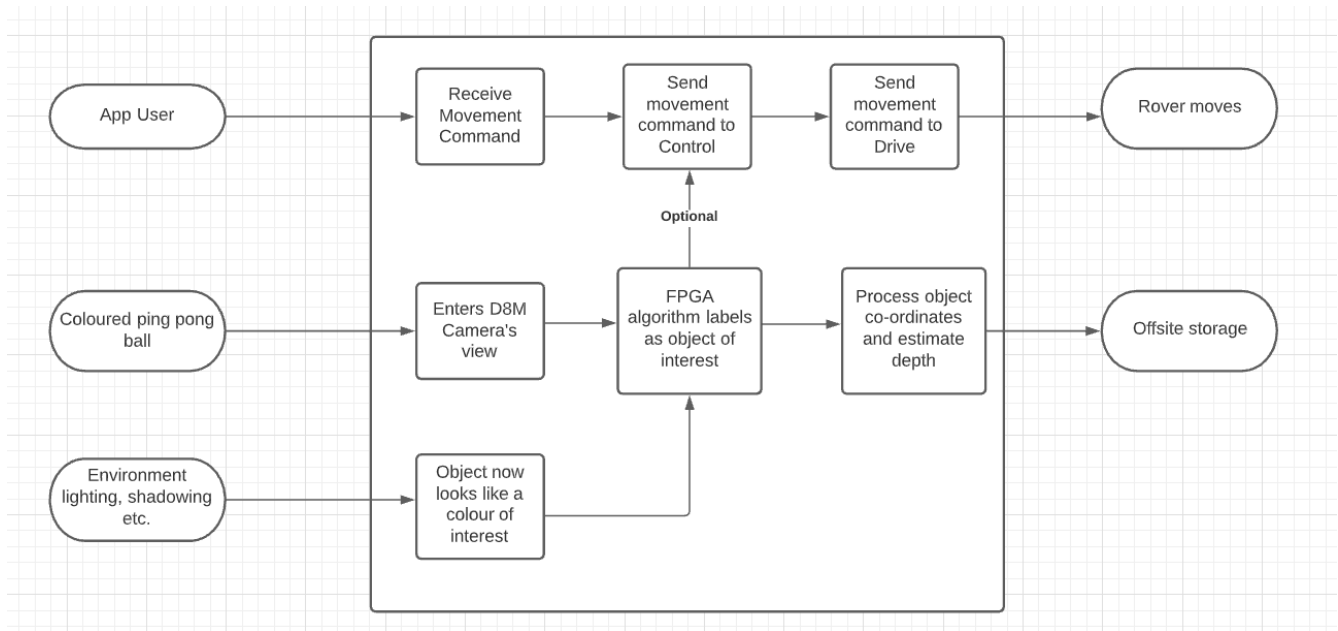


Figure 4 – Functional Diagram

Using the actors as shown in the use case diagram, the system will have several potential responses. Firstly, if there are no obstacles in its path, and it receives no commands from the app user, it will remain stationary. If the app user sends commands, the rover should move according to those commands. Similarly, if it believes it will collide into an obstacle, which includes the colored ping pong balls, then it can optionally divert its path. In addition, the system should be able to build up a rough picture of its surroundings which it can save in offsite storage.

2.5 Inter-module Communication

2.5.1 Vision and Control

After trialing communication using the SPI protocol, it was decided that a more reliable and simpler method would be to use a Software UART connection. On the FPGA side, this involved adding the UART RS-232 Serial Port IP Core to the Qsys model and doing the necessary connections. An advantage of using the UART core is that the Altera software provides abstractions from the protocol for developers, using their Hardware Abstraction Library (HAL). This enables the UART hardware to be accessed using C functions from the stdio.h library, for example `fprintf`. On the other hand, implementing an SPI protocol in Verilog, while providing more customizability, requires hardware design and manipulation of the modules to ensure correct functionality.

Using the UART's full functionality requires significant memory usage, and thus it must be used sparingly. In addition, the code base of the software was kept at a minimum. To differentiate between the different colors and the bounding box minimum and maximum **x** and **y** values, the color code parameters are hardcoded in the message buffer data, for the ESP-32 to decode accordingly. As the state machine advances, each color's values are sent to the write buffer.

A code snippet of the UART implementation in the NIOS II processor is shown in Appendix D. A file pointer is created and written to from the FIFO buffer. The file pointer is opened and closed once in the main function to reserve memory space.

On the ESP32 side, although UART Serial was decided as the transfer medium. It was soon found that the 3 HW Serial Ports on the ESP32 were already occupied, one being for Drive, another for Energy and the last serving as the Serial0 for USB. Thus, the team resorted to using a Software serial, which emulates the UART process through software, and has transmit and receive ports that can be defined as any GPIO pin. The team setup the ESP32 and the FPGA in a one-way orientation, from the FPGA to the ESP32 only. A 32-bit wide number would be sent, and calculations (detailed above) would be performed on the ESP32. This 32-bit wide number warranted 4 packets to be sent via the Serial connection which was configured as 8 bits wide. On the ESP32 side of code, the value would be read using the `Serial.parseInt()` function and stored in an 8-byte wide integer for later calculation.

2.5.2 Command and Control

The connection between command and control was crucial for the rover to be remotely controlled by a server. The team decided to use WiFi as the primary medium for data transfer. Since the ESP32 had to connect to an offshore webserver, the team had to consider what transfer protocol to use. HTTPS, MQTTS and TCP/IP socket transfer were all considered, and the team decided upon TCP/IP socket transfer, favoring its ease of implementation and the ability to send and receive individually without the need for a handshake. It also took up the least space as a function on the ESP32 and did not require any new library. It did, however, introduce many complications regarding the way information was then transferred between command to drive. For instance, to receive information from the server, the team had to make use of `strcpy()` to copy the instruction from the server as a string to the ESP32, and then `atoi()` to convert that to an int for sending via Serial to the Arduino Nano for drive:

```
Serial.println(buf);  
  
//copy out of this "instruction" variable to get the integer number that represents the instruction  
strcpy(instruction,buf);
```

```
Serial.println(buf);  
tempdriveinstr = Serial.readString();  
newdriveinstr = atoi(tempdriveinstr.c_str());
```

Figure 5 -Receiving Information from Server

In the same way, the ESP32 was forced to send information as a string to the server, and so to display parameters like the battery level or the drive distance command was forced to convert from string back to an integer.

2.5.3 Drive, Energy and Control

The drive module and energy modules are meant to share battery and SMPS data to make efficient use of the battery power available. Certain drive instructions would have been adjusted according to current battery charge status or current battery life cycle. For example, an instruction which controls

the speed of the motors would have had to consider the whole range of possible output voltages from the battery. Unfortunately, due to the energy module students not being able to work on batteries any longer, the final version of the rover does not make use of the battery, all SMPS control will have to be performed with input from a power supply only.

To receive instructions from, and to upload the battery information to command for drive and energy respectively, the team made use of UART serial communication through the ESP32. This was set up by defining certain pins for transmission and initializing serial communication:

```
//pin def for UART for energy
#define RXD2 16 //8
#define TXD2 17 //9
//pin def for UART for drive
#define TXD1 18 //9
#define RXD1 19 //10
//pin def for UART for vision
#define TXD3 15 //14
#define RXD3 2 //15

void setup() { //setups ESP32 controller connections: Wifi, SPI and UART
  Serial.begin(115200);
  while(!Serial)
  {}
  Serial.println("starting up");
  initWifi();
  Serial.println("wifi connected");
  serverconnect();
  Serial.println("connected to server");
  Serial.println("setup finished");
  Serial2.begin(9600, SERIAL_8N1, RXD1, TXD1); //for energy, may need to change baud rate
  Serial1.begin(9600, SERIAL_8N1, RXD2, TXD2); //for drive, figure out arduino uno max baud rate
  Serial3.begin(9600, SWSERIAL_8N1, RXD3, TXD3); //For Uart FPGA connection
```

Figure 6 -Pin Definition and Serial Initialisation Code

the same was done on the Arduino nano. To begin, the Arduino nano would send information regarding the distance travelled in 2 axes as a uint8_t integer. This decision was made to keep the number of packets being sent across UART the same, as the team used the default SERIAL_8N1 Transfer protocol. The same was done for the instructions being sent to the Arduino, and the instruction integer was capped at 256 so as to not exceed 8 bits. By ensuring only one packet is send, the process of communication from drive to control to command and back is homologated, and so delays may be omitted altogether. The same was maintained for communication from Energy to Control to Command.

3. Implementation

3.1 Implementation of subsystems

3.1.1 Vision

3.1.1.1 Preprocessing

Due to noise in the video feed, it was decided that a filter would be useful to increase the accuracy of the colour detection and bounding box calculations. The group experimented with a few filter algorithms – convolutional, gaussian and median filters. As all three methods would need the same window buffer (in our case 3x3), the median filter was implemented due to its relatively simple calculations when compared to other options.

The median filter would store a 3x3 window of the image feed, comparing each element and sorting them in increasing order, then finally choosing the median value. This would then be used as the final RGB value before colour conversion. This was programmed on the FPGA due to its parallel computation capability and low latency from the input feed.

3.1.1.2 Colour Conversion

The initial video packet is received in Red, Green and Blue (RGB) colour format. This is displayed using a 24-bit register, 8 bits for each of the 3 colours, respectively. However, the RGB colour format is not well suited for object detection as it does not separate image intensity from colour information. This will result in incorrect results in the presence of even slight light changes, shadows etc. Upon testing of the bounding box detection algorithm using the RGB colour format, it was found that these limitations were too high for accurate results. The Hue, Saturation and Value (HSV) colour format does enable this separation, and so it is better suited for the system. Subsequently, the vision subsystem has chosen to convert the original RGB colour into HSV for object detection.

The algorithm for converting 24bit RGB to 25bit HSV is well documented. The 9 most significant bits (MSB's) of the HSV format correspond to the hue, which takes a value of 0-360, the next 8 bits are for the saturation and the final 8 bits are the value, both taking a value of 0-255. This is implemented as an asynchronous algorithm on the FPGA using Verilog. The code, along with the test bench, can be seen in HSV.v and HSV_TEST.v respectively.

To configure the HSV boundaries for the various object colours, a combination of trial-and-error testing and the in-built Quartus HSV format tool was used. The Quartus tool can be found in Tools -> Options -> Colors. From this, the following boundaries were used to distinguish the necessary object colours:

Object Colour	Hue	Saturation	Value
Red	< 20 or > 340	> 100	> 80
Green	< 140 and > 100	> 120	> 100
Blue	< 240 and > 150	> 80 and < 180	> 100
Orange	> 20 and < 80	> 120	> 100
Pink	> 300	> 60	> 80

Table 2 - Colour Conversion Boundaries

The boundaries are hardcoded in the top-level module `EEE_IMGPROC.v` file, an example of which is shown in Appendix C. In this code snippet, the saturation and value of red are shown – when specific saturation and value boundaries are required, the corresponding colour is named in the wire. The approach taken to red shown in the code snippet is the same as for the other colours.

An example test picture of coloured ping pong balls is shown below. The setup is a black background with a light source directly above the objects and camera.

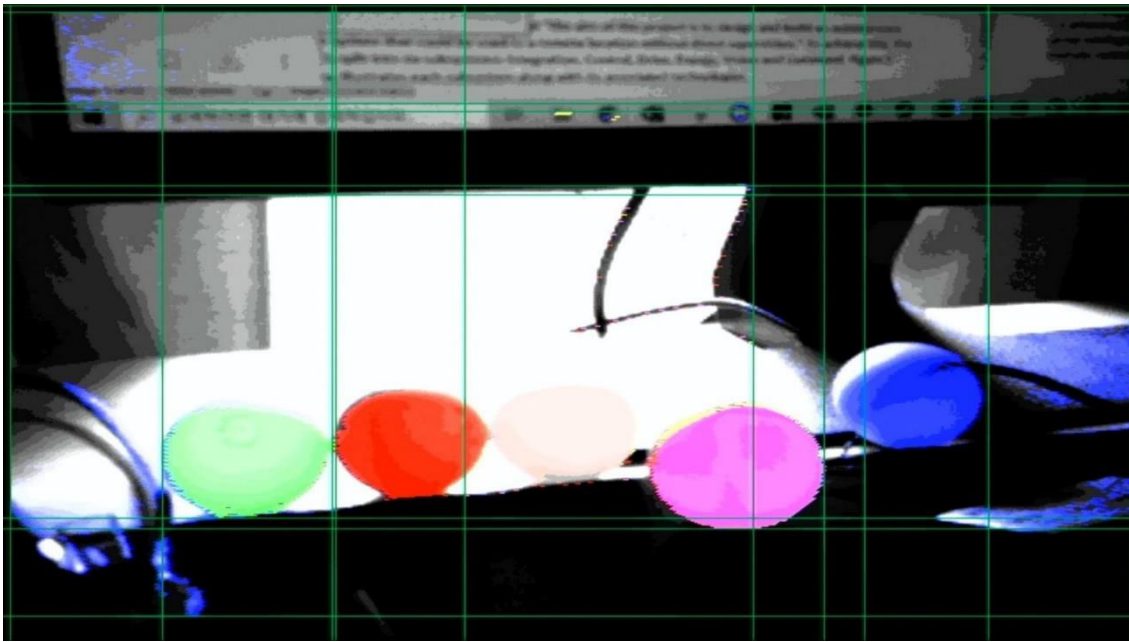


Figure 7 - Coloured Ping Pong Balls

3.1.1.3 Bounding Box Calculation

The input from the video feed is scanned from left to right, top to bottom. If a colour is detected in a valid packet, then the x and y bounds are updated and stored until the colour is no longer detected. For debugging purposes and as a visual aid, the values are latched to display the box formed from these x and y values on the video feed. Simultaneously, the updated x and y corner values (i.e., the minimum and maximum values of each box found) used to calculate the area of the object.

Upon calculation, a simple state machine is implemented to ensure every color's area is processed and sent along with the identifier of that colour. A counter is added to delay state machine advanced to ensure all the processing requirements are done. The state machine is reset when the frame count is 0 and the message buffer size is below a constant value. The state machine code is shown in Appendix E.

For testing and debugging purposes, visual boxes are formed around objects of interest and can be viewed on the live video feed. This is shown in Figure 16, where boxes are formed around the ping pong balls. The corners of these boxes are their respective x and y values.

3.1.1.4 Communication with ESP-32

For the rover to semi-autonomously navigate its surroundings, the bounding box x and y values are not sufficient without further calculations. The approach that the group took was to send the x and y values produced by the FPGA to the ESP-32 for the further calculations due to its higher memory capability over the NIOS II processor. There were two methods of communication trialed, described below.

The first communication method was using the SPI protocol directly from the FPGA hardware to the ESP-32, where the ESP-32 would act as the master and the FPGA would act as the slave. In this case,

the NIOS II processor would simply act as a debugging tool and isn't required for functionality. The second method is using the UART protocol. In this case, the NIOS II processor would be responsible for sending the data to the ESP-32 by receiving the x and y values from the message buffer and sending them bit by bit over UART.

After various attempts and testing, it was decided that the UART method of communication would be pursued and implemented. The reasoning and a more in-depth explanation of the communication functionality is described in the "Inter module communication" section.

3.1.1.5 ESP-32 Vision Calculations and Off-Site Storage

As the ESP-32 receives the x and y values minimum and maximum values, it calculates the area of each color's bounding box in view. The current area is then compared with the previously calculated one. If the area has increased, the rover is moving in the direction of the object and so will eventually collide (provided the area of the bounding box of the object continues to increase). Thus, this area comparison is useful in determining whether the rover needs to change direction.

3.1.1.6 Resource Utilization

The Quartus compilation report allows users to analyze the resource utilization on the FPGA of the team's design. The compilation flow summary, along with the Fmax summary, is shown in Appendix F. To summarize, the group used 23% of the available logic elements, 48% of the total pins and 80% of the total memory bits.

3.1.2 Drive

3.1.2.1 Overview

The drive module is designed to reliably provide the entire range of movement for the rover. The functional design of this module is centered around three main problems: defining a suitable instruction set, controlling the SMPS to operate the motors and returning an accurate value of the distance travelled from the optical flow sensor.

3.1.2.2 Instruction Set

Instructions are received from the 8-bit serial communication with the control module. The focus here is to create a simple but competent instruction set that can reliably be used by the command module through the control module. If we were to translate the chosen instructions into decimal data, they would become numbers in the range 0-255. From this perspective, we decide to formally define the set of instructions as \overline{xyz} decimal numbers with $x, y \in \{0,1,2\}$ and $z \in [0; 9]$, $x, y, z \in \mathbb{Z}$. Here, z defines the speed of the motors, while x and y define the functions represented in the table below.

	0	1	2
x	no linear movement	move forward	move backwards
y	no rotational movement	rotate clockwise	rotate counterclockwise

Table 3 - Drive Instructions

The instruction set translates seamlessly to physical movement of the rover. Some useful examples of the rover behavior for different instructions are described below.

Instruction	Behavior
101	Rover moves forward with speed 1 out of 9.
209	Rover moves backwards with speed 9 out of 9.
015	Rover rotates clockwise with speed 5 out of 9.
023	Rover rotates counterclockwise with speed 3 out of 9.
112	Rover moves forward with speed 2 out of 9. Rover ignores rotational command.
009	Rover doesn't move.
200	Rover doesn't move. Speed is 0 out of 9.
0	Rover doesn't move.

Table 4 - Instructions to Behaviour

Some important design choices are exemplified above. To begin with, an overlapping linear and rotational movement instruction as seen in the 112 instruction case above will only result in linear movement. The rotational component of the instruction will be treated as input error and therefore it will not result in rotational movement of the rover. Another design choice is the existence of multiple possible instructions for the rover to become stationary. Any instruction containing 0 for the speed component will result in no movement from the rover. Any instruction with 0 for both the linear and rotational movement components will also result in no movement from the rover.

Because we are receiving and sending 8-bit data through the serial communication with control, it may be useful to look at the instructions from a bitwise perspective. This is analyzed by the functional design processes of the command module and can be read in the specific section below.

3.1.2.3 Motor Control and the SMPS

The motor control part of the drive module can be split into two main components. One of the components is the directional control for each of the motors, which is done through the two H-bridge circuits on the Motor Control IC. The other is speed control and has to be implemented by creating a suitable framework to adjust the SMPS voltage.

The H-bridge circuits serve the function of switching the polarity of the two motor terminals, thus making each wheel turn in the direction specified by the current instruction.

The SMPS serves as a speed controller for the two motors. Through the instruction set, we have defined 10 possible speeds and thus 10 possible output voltages from the power supply. Their exact specification is shown in the table below.

Speed Level	0	1	2	3	4	5	6	7	8	9
Wheel RPM	0	20	23	26	31	36	42	47	52	60
Output Voltage (V)	0	1.69	2	2.3	2.69	2.88	3.19	3.48	3.79	4.82

Table 5 - Motor Speed Specifications

One important functional requirement here is for the voltage to be provided accurately and reliably to the motors in order to obtain the desired movement. The implementation section of the report

covers how the functional design described above is translated into software specification and hardware operation of the SMPS.

3.1.2.4 Position Data

Position Data is obtained using information from the optical flow sensor. The sensor measures the displacement of individual pixels of different brightness levels in time and uses this value to compute a set of distances for the movement of the entire rover assembly.

The main functional design requirement for the sensor is to output the distance travelled in the x and y directions relative to the initial position of the rover. This information is sent back to the control module through the existing serial connection.

3.1.2.5 Instruction Reading

The 8-bit instruction is read from Serial and split into the three components. From the speed component (last digit), a reference value for SMPS control is calculated and kept into the global variable *sensorValue2*. In the regular closed-loop control of the SMPS, this variable would have stored the reading from the potentiometer situated on the SMPS board. For our purpose, we simulate a sensor reading that puts the SMPS in the right operating region for each speed. The figure below shows how this is implemented in code.

```
if (Serial.available())
{
    last_instr = Serial.read();
    Serial.println(last_instr);
}

int instr_combined = last_instr;
int instr_speed = instr_combined % 10;
instr_combined /= 10;

int instr_rot = instr_combined % 10;
instr_combined /= 10;

int instr_dir = instr_combined;

//speed control
sensorValue2 = 350 + ((1023-350) * instr_speed / 9);
```

Figure 8 – Instruction Processing

To note here is the range of values of the *sensorValue2* variable. The maximum possible value is 1023, while the theoretical minimum is 0. In practice, we choose to bias the 0 speed instruction by 350 and space the 10 possible speed values evenly between 350 and 1023. For lower bias levels, the weight of the rover assembly and cables would overwhelm the torque produced by the motors and the rover would not move.

3.1.2.6 Motor Control Implementation

Two H-bridge circuits are used to switch the polarity on each of the motors in order to control the general direction of movement for the rover. An H-bridge circuit consists of four transistors connected around a motor as shown in the simplified figure below.

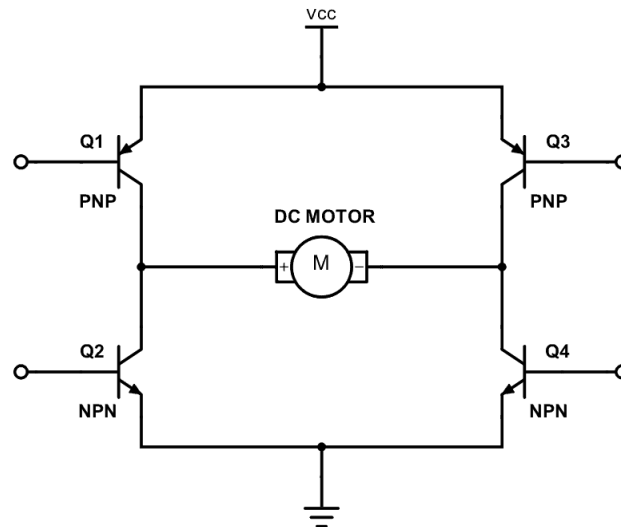


Figure 9 – H-bridge Circuit

Our Motor Control IC contains two of these H-bridge circuits on a single chip. The chip also incorporates the control logic circuitry necessary for each of the bridges.

To control each of the motors from code, we require one direction pin and one PWM pin going from the Arduino Nano Every to the H-bridge circuit. These pins are defined in code and initialized at setup as shown below.

```
int DIRRstate = LOW;           //initializing direction states (these mean backwards)
int DIRLstate = HIGH;

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

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

```
pinMode(DIRR, OUTPUT);
pinMode(DIRL, OUTPUT);
pinMode(pwmr, OUTPUT);
pinMode(pwml, OUTPUT);
digitalWrite(pwmr, LOW);       //setting right motor as off
digitalWrite(pwml, LOW);       //setting left motor as off
```

Figure 10 – Initialisation and Setup Code of Pins

The motor states are updated at each main loop iteration and follow the specification of the last instruction received. In case the connection to serial is broken, the last instruction will stop being followed and both motors will stop running instead.

```

if (instr_dir)
{
    if (instr_dir == 1)
    {
        //front
        DIRRstate = HIGH;
        DIRLstate = LOW;
        digitalWrite(DIRR, DIRRstate);
        digitalWrite(DIRL, DIRLstate);
        digitalWrite(pwmr, HIGH);
        digitalWrite(pwml, HIGH);
    }
    else
    {
        //back
        DIRRstate = LOW;
        DIRLstate = HIGH;
        digitalWrite(DIRR, DIRRstate);
        digitalWrite(DIRL, DIRLstate);
        digitalWrite(pwmr, HIGH);
        digitalWrite(pwml, HIGH);
    }
}

else
{
    if (instr_rot)
    {
        if (instr_rot == 1)
        {
            //clockwise
            DIRRstate = HIGH;
            DIRLstate = HIGH;
            digitalWrite(DIRR, DIRRstate);
            digitalWrite(DIRL, DIRLstate);
            digitalWrite(pwmr, HIGH);
            digitalWrite(pwml, HIGH);
        }
        else
        {
            //anti-clockwise
            DIRRstate = LOW;
            DIRLstate = LOW;
            digitalWrite(DIRR, DIRRstate);
            digitalWrite(DIRL, DIRLstate);
            digitalWrite(pwmr, HIGH);
            digitalWrite(pwml, HIGH);
        }
    }
    else
    {
        //stop motors
        digitalWrite(pwmr, LOW);
        digitalWrite(pwml, LOW);
    }
}

```

Figure 11 - Motor Direction and Rotation Control

The speed control of the motors is done through the closed-loop SMPS control structure. The PID controller that was already in use for the sample code and other previous SMPS uses is still in charge of controlling the duty cycle and thus the output voltage. However, to make this work with the speed defined in the instruction, we have to remove the reading of *sensorValue2* from the sampling algorithm and replace it with the virtual value calculated for the speed instruction.

```

void sampling(){

    // Make the initial sampling operations for the circuit measurements

    sensorValue0 = analogRead(A0); //sample Vb
    //sensorValue2 = analogRead(A2); //no longer using potentiometer reading
    sensorValue3 = analogRead(A3); //sample Vpd
    current_mA = ina219.getCurrent_mA(); // sample the inductor current (via the sensor chip)

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

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

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

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

```

Figure 12 – Modified Sampling Function

3.1.2.7 Position Data from Optical Sensor

A first step in providing accurate position data for x – axis and y – axis movement is manual calibration of the optical flow sensor. This implies two physical adjustments to the sensor module. For one, the led that provides the light that will reflect into the optical sensor from the ground has to be pointing at the center of the area that the optical sensor covers. The second physical adjustment implies modifying the distance between the lens and optical sensor such that the sensor can focus clearly on the ground. The code used for calibration is provided in the documentation. The functions that accompany the documentation proved to be of great use and are included in the main loop as the backbone of the position data measurement. As the accuracy of these measurements proves to be satisfactory, the data structures and functions of the sample code are also used here.

```

//Position Feedback from Optical Sensor
int val = mousecam_read_reg(ADNS3080_PIXEL_SUM);
MD md;
mousecam_read_motion(&md);

distance_x = md.dx;
distance_y = md.dy;

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

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

```

```

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

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

```

Figure 13 – Position Data from Optical Flow Sensor

3.1.3 Command

3.1.3.1 Overview:

The command subsystem was tasked to design a web or mobile platform for remote control the rover by connecting to the control subsystem. The specifications for command were moderately lenient compared to those of the other subsystems, and multiple options were given to communicate with the rover. The option the team chose was the first option, which was “A command-line interface that connects to a remote server over TCP to provide real-time logging and control of the system”.

3.1.3.2 Remote Server:

The server used was the Azure virtual machine server. The operating system used was Ubuntu 18.04, and the location of the server was set at East US, as it had the lowest cost. Further detail regarding the server is given in the table below.


Azure Server			
Resource group (change) : MarsRover		Operating system	: Linux (ubuntu 18.04)
Status : Running		Size	: Standard B1s (1 vcpu, 1 GiB memory)
Location : East US 		Public IP address	: 104.45.192.134
		Virtual network/subnet	: MarsRover-vnet/default
		DNS name	: Not configured

Table 6 – Azure Server Overview

The initial plan of the command module was to start off with a backend connection between the ESP32 and a server, continued by uploading and receiving the data via terminal. Despite further plans to develop a frontend mobile app with buttons, the team was limited to the terminal due to time constraints.

3.2.3.3 Setting up TCP Server & Basic Server Client Communication:

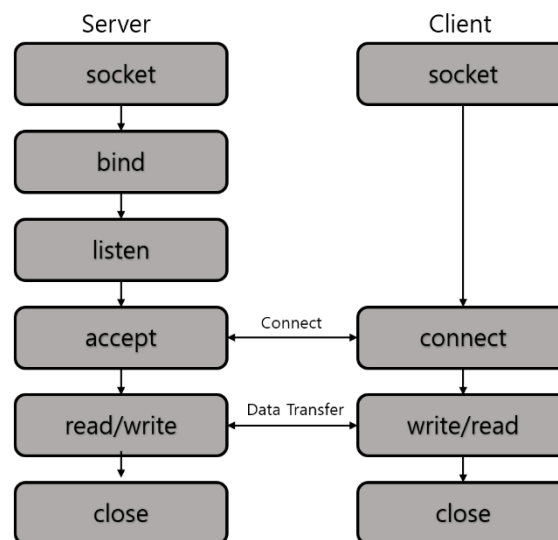


Figure 14 – Server and Client Communication Diagram

The sequence of function calls for the TCP server and the TCP client was designed as the figure above.

The setup of both the server and client were done in C, and the main libraries used were `sys/types.h`, `sys/socket.h`, and `netinet/in.h`.

Server Setup	Client Setup
<pre>#include<sys/types.h> #include<sys/socket.h> #include<netinet/in.h> #define PORTNUM 18000 struct sockaddr_in sin,cli; sd=socket(AF_INET,SOCK_STREAM,0); setsockopt(sd,SOL_SOCKET,SO_REUSEADDR,&optvalue ,sizeof(optvalue)); memset((char*)&sin, '\0', sizeof(sin)); sin.sin_family=AF_INET; sin.sin_port=htons(PORTNUM); sin.sin_addr.s_addr=htonl(INADDR_ANY); bind(sd,(struct sockaddr *)&sin,sizeof(sin)) listen(sd,5) ns=accept(sd,(struct sockaddr *)&cli,&clientlen))</pre>	<pre>#include<sys/types.h> #include<sys/socket.h> #include<netinet/in.h> #define PORTNUM 18000 struct sockaddr_in sin; sd=socket(AF_INET,SOCK_STREAM,0); memset ((char*)&sin, '\0', sizeof(sin)); sin.sin_family=AF_INET; sin.sin_port=htons(PORTNUM); sin.sin_addr.s_addr=inet_addr("104 .45.192.134"); connect(sd,(struct sockaddr *)&sin,sizeof(sin))</pre>

Table 7 - TCP/IP Server Connections

The socket used for the server was the AF_INET socket, and the team chose a TCP protocol over the UDP protocol. As seen in Figure 4, first the socket is created for both the server and client via “sd=socket(AF_INET,SOCK_STREAM,0);”. Then, the server and client binds the socket to an address by the declaration of sin.sin_family, sin.sin_port, and sin.sin_addr.s_addr followed by the bind() function. The server then listens for connections by the listen() function. Finally, the server accepts the connection with the accept() function call. This connection happens when the client calls the function connect().

Server Setup	Client Setup
<pre>write(ns,buf,strlen(buf)+1) read(ns,msg,sizeof(msg)) write(ns,buf,strlen(buf)+1)</pre>	<pre>read(sd,buf,sizeof(buf)) write(sd,msg,strlen(msg)+1) read(sd,buf,sizeof(buf))</pre>

Table 8 - Connection Test

Once the TCP server was set up, the next step was to test the connection between the server and the client. The server would send information as a string in the variable buf, and the client would send information by the string in the variable msg. Specifically, buf would carry a string containing the instruction for the movement of drive, and msg would carry a string containing the status of the rover such as the remaining battery, distance travelled, and current location of the rover.

3.2.3.4 Instruction set for Drive

The requirement received from control was that we had 8-bit space to translate the movement control of the rover from command to drive. The movements the team decided to make were

movement forward or backward, rotate clockwise or counterclockwise, and a motor power between 1-9.

The team dedicated the first 2 bits to the longitudinal movement, the next 2 bits to the rotational movement, and the final 4 bits to the motor power.

Longitudinal Movement	Rotational Movement	Motor Power
<pre> if(strcmp(move, fo)==0){ //move forward instr[0]=0; instr[1]=1; } else if(strcmp(move, ba)==0){ //move backward instr[0]=1; instr[1]=0; } else{ //dont move instr[0]=0; instr[1]=0; } </pre>	<pre> if(strcmp(rotate, cl)==0){ //rotate clockwise instr[2]=0; instr[3]=1; } else if(strcmp(rotate, ccl)==0){ //rotate counter clockwise instr[2]=1; instr[3]=0; } else{ //dont rotate instr[2]=0; instr[3]=0; } </pre>	<pre> if(10 > power > 0){ //power is power int n =power; instr[4]=0; instr[5]=0; instr[6]=0; instr[7]=0; for(int i=7;n>0;i--){ instr[i]=n%2; n=n/2; } } else{ //dont move instr[4]=0; instr[5]=0; instr[6]=0; instr[7]=0; } </pre>

Table 9 - Drive Instructions Movement Code

As shown in the table above, if the input longitudinal command contained the string “forward”, the first 2 bits were set as 01. If the string was set as “backward” the first 2 bits were set as 10, and for any other string it was set to 00. For the rotational command, if the string contained clockwise the next two bits were set as 01. Similarly, counterclockwise would result in the bits 10, and any other string would result in the bits 00. Finally, the remaining 4 bits were used to represent the power of the rover. The input was an integer value of the motor power between 1-9, which were translated into binary values between 0001 to 1001.

3.2.3.5 Test of Model Rover and User Connection:

The next step taken for the command module was to test the server communicating with the “model” rover. Here, the team sent sample command lines instructing the rover to move through the terminal and tested if the integer translation of the 8-bit instruction was successfully carried on to the client. The results are shown in the table below:

Test Server	Test Client
<pre> ubuntu@RoverServer:~\$./server waiting client... forward move 6 01000110 Status of rover: Battery remaining: x%, Distance Travelled: xm Sending result: 70 rotate clockwise 2 00010010 Status of rover: Battery remaining: x%, Distance Travelled: xm Sending result: 18 backward move 4 10000100 Status of rover: Battery remaining: x%, Distance Travelled: xm Sending result: 132 rotate counterclockwise 8 00101000 Status of rover: Battery remaining: x%, Distance Travelled: xm Sending result: 40 stop </pre>	<pre> sjhan@DESKTOP-SUNGJUN: p/marsrover\$./client 70 18 132 40 </pre>

Table 10 - Testing Rover and User connection

The left side of the table displays the input in the command line of the server, while the right side represents the integer instruction received. Taking into example the first command line which was “forward move 6”, the 8-bit translation of this command is 01000110, which is the integer 70. The server first receives the status of the rover, which contains information on the battery remaining and the distance travelled. Then, the server sends the integer 70, which was successfully received by the client as seen in the right side of the Table. The team repeatedly tested for the four different possible types of instructions and confirmed that the model rover and user connection was successful.

3.2.3.6 Connecting to ESP32 and Communicating:

Following the test of the model rover and user connection was the server connecting with the actual ESP 32. In this process, some changes had to be made to the code, as there were trouble using the original library used, which was `arpa/inet.h`, which was limited to UNIX. Instead, `netinet/in.h` was used, which was more widespread for the use of the ESP32. Like the testing in section *Test of Model Rover and User Connection*, the user would right a command line such as “forward move 1”, which should be carried over to the ESP32 as an integer of the instructions translated into 8-bits. The table below demonstrates the process:

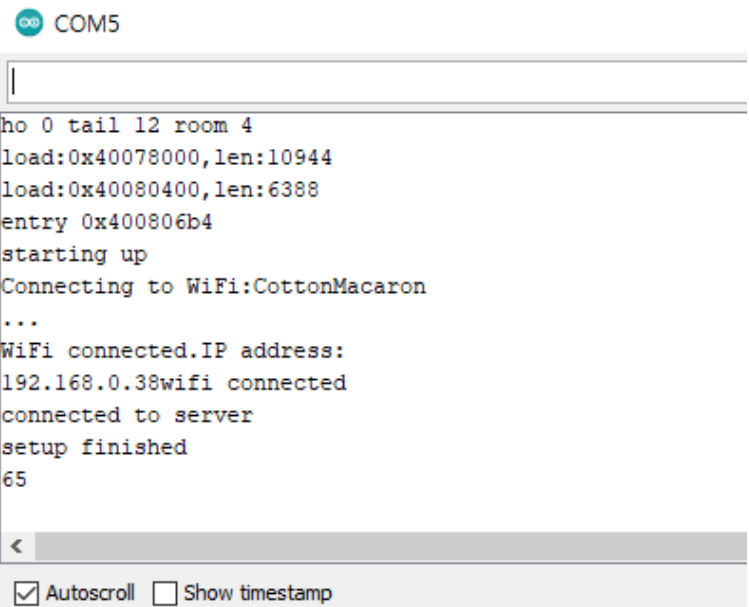
Server Terminal	ESP32
<pre> ubuntu@RoverServer:~\$./server waiting client... forward move 1 01000001 Status of rover: NA Sending result: 65 </pre>	 <pre> ho 0 tail 12 room 4 load:0x40078000,len:10944 load:0x40080400,len:6388 entry 0x400806b4 starting up Connecting to WiFi:CottonMacaron ... WiFi connected.IP address: 192.168.0.38wifi connected connected to server setup finished 65 </pre> <p><input checked="" type="checkbox"/> Autoscroll <input type="checkbox"/> Show timestamp</p>

Table 11 - ESP32 Connection

The left side of the table showing the Server Terminal again shows the command line and the input from the ESP32, and right side of the table shows the interaction of the ESP32 and it successfully receiving the integer instruction, 65.

3.1.4 Energy:

3.1.4.1 Overview

The energy module provides a battery for the rover. There are mainly two parts to the battery: the PV panels and the Lithium-Iron Phosphate cells, which are connected using the SMPS and Arduino to control their behavior.

3.1.4.2 Photovoltaic(PV) Panels:

To characterize the PV panel, a single PV panel was connected to port B of the SMPS along with a 120Ω resistor connected to port A of the SMPS. According to theory, the I-V graph of a PV panel should look like the following:

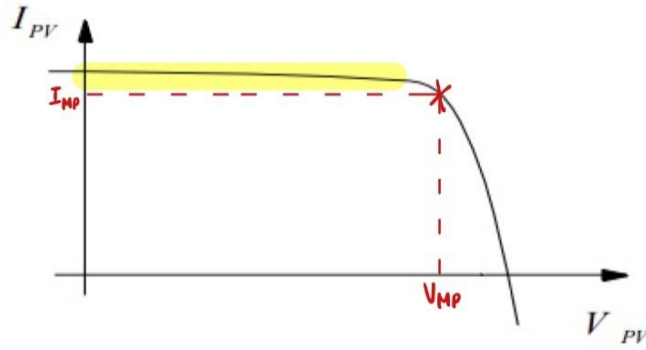


Figure 15 - theoretical I-V graph of a PV panel

V_{MP} and I_{MP} represents the voltage and current at the maximum power point. This will be visited again later.

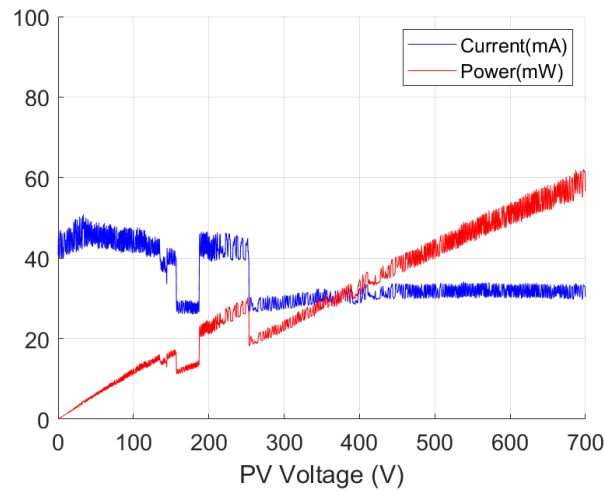


Figure 16 - experimental I-V graph of a PV panel

From the graph, the current can be said to be gradually decreasing slightly, with power obviously increasing as the voltage increases and only a bare minimum change in current is shown (in mA). The voltage range has been shortened as the voltage is out of the operation voltage of the PV panel. The current in figure 6 represents the highlighted in figure 5.

3.1.4.3 Maximum Power Point Tracking (MPPT):

Maximum power point tracking is an algorithm designed for the PV panel to operate near its maximum power point (MPP). This is where the product of V_{MP} and I_{MP} is the largest. Here is a diagram of where the maximum power point is:

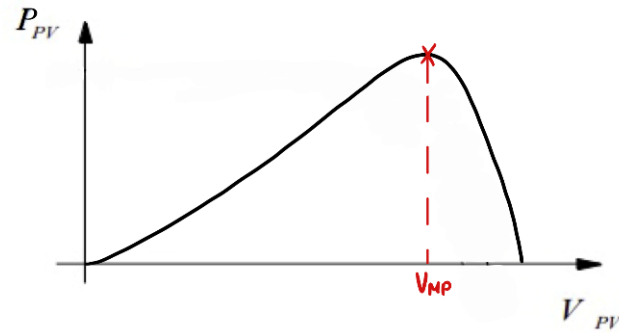


Figure 17 - Power-voltage graph of PV panel showing the MPP

There are several ways to design the algorithm for MPPT: Perturbation and observation (P&O), incremental conductance and fractional open-circuit voltage. For this project, P&O was chosen as the other two failed to be programmed.

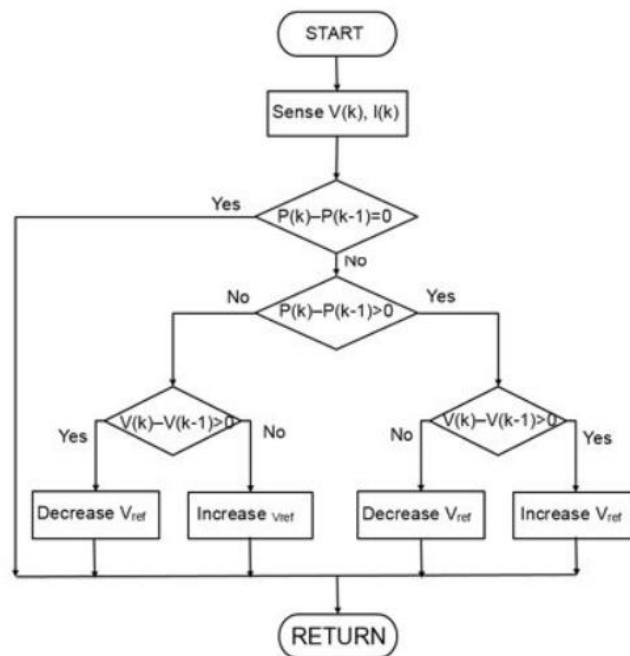


Figure 18 - Flowchart of P&O algorithm¹

¹ <https://uk.mathworks.com/solutions/power-electronics-control/mppt-algorithm.html>

This design was then implemented as an Arduino code:

```

pwm_last = pwm_temp;
pwm_temp = pwm_out;
power_last = power;
power = current_measure * V_Bat;
if(power - power_last == 0){
    digitalWrite(8, true);
}else if(power - power_last > 0){
    if(pwm_out - pwm_last > 0){
        pwm_out = pwm_out + 0.01;
    }else{
        pwm_out = pwm_out - 0.01;
    }
}else if(power - power_last < 0){
    if(pwm_out - pwm_last > 0){
        pwm_out = pwm_out - 0.01;
    }else{
        pwm_out = pwm_out + 0.01;
    }
}else{
    pwm_out = pwm_out + 0.01;
}
pwm_out = saturation(pwm_out, 0.99, 0.01);

```

Figure 19 - Arduino code of P&O algorithm

The algorithm was designed in a way that the duty cycle will vary until the MPP is reached and almost remain constant once the MPP is reached. The PV panel was tested under the halogen bulb lamp with the SMPS at buck mode, with a 10Ω resistor. To see the variation in duty cycle, the lamp was shifted every few minutes.

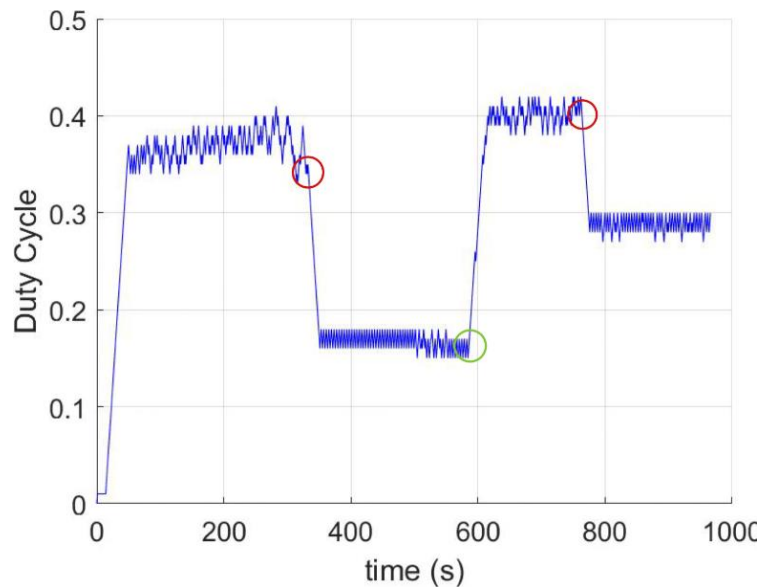


Figure 20 - MPPT test with duty cycle against time

The red circle indicates when the lamp was moved away from the PV panel and the light green circle indicates when the lamp was moved towards the PV panel. When the lamp is moved the light

intensity and temperature, which are both factors to the position of the MPP, are varied and to reach a new MPP for the following changes, the duty cycle changes.

3.1.4.4 Single Cell:

Before the cells were arranged to form a battery, a single cell was tested, using the pre-made Arduino code given to us. By connecting the cell and the 10Ω resistor to the SMPS, the results have shown that it follows the state diagram.

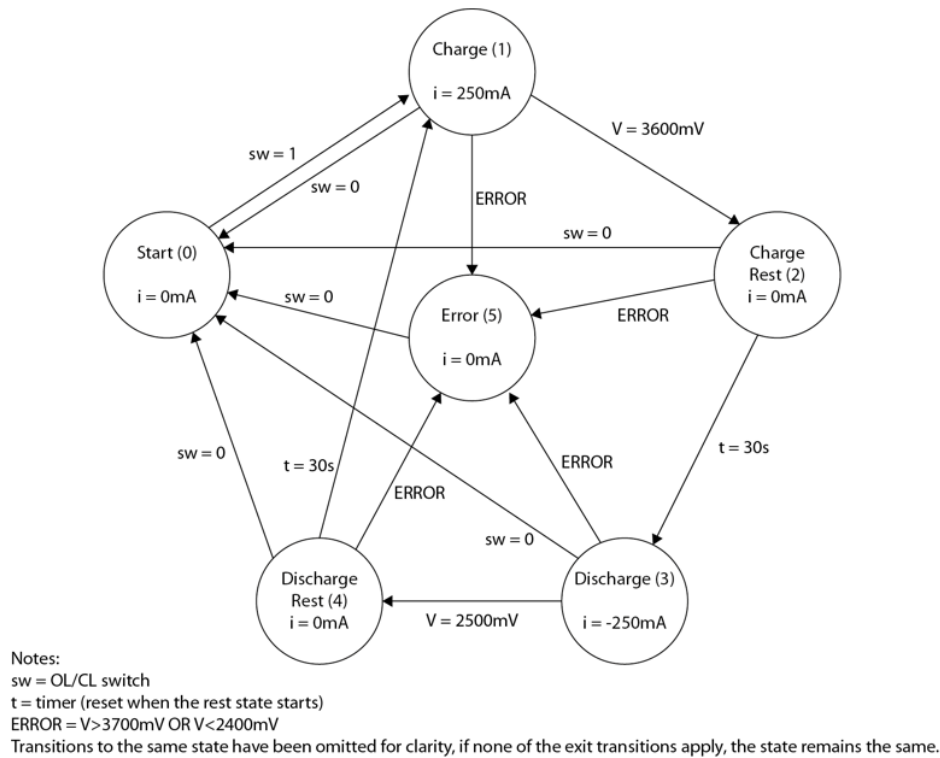


Figure 21 - State diagram of charging and discharging a single cell

The following diagram shows the transition between charging and discharging, with a 30second rest in between:

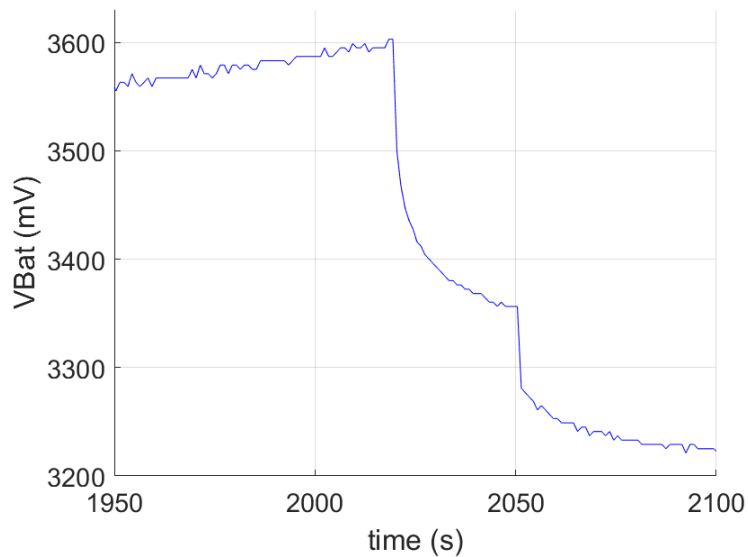


Figure 22 – Transition Between Charging and Discharging.

As followed in the state diagram, the cell stops charging as it hits 3.6V (3600mV). However, it does not give a constant voltage after the 30 seconds rest given by the rest state (state 2). It takes more than 30 seconds and another drop before it reaches a steady voltage of around 3200mV. During the rest state, several factors give an unreliable result. During the first drop in voltage at around 2020 seconds, it starts off as a steep drop. This is most probably due to a resistive element that gave extra voltage ($V = IR$) when charging since there was a current of 250mA. The steep drop is followed by a curve that reaches till around 2050 seconds, which is probably discharging of capacitive elements within the circuit that has been charged while the cell was being charged. From this it can be noticed that a single current should not be used to charge the cell.

3.1.4.5 <Battery failure>

No other experiments could be done as the energy module was told to stop working with the cells

3.1.5 Control

3.1.5.1 Overview

Before starting off with implementing various solutions for the rover, the team identified what was required for each sub-system. For Control, several tasks were identified:

1. Establish connections between all the systems and allow them to communicate without user interference.
2. Decide on and use protocols that favor a smaller delay and ease of accessibility.
3. Aid Vision in doing smaller calculations regarding boundaries.
4. Receive instructions from the user over WIFI.

The team created a small flowchart to decide what was required where for each subsystem:

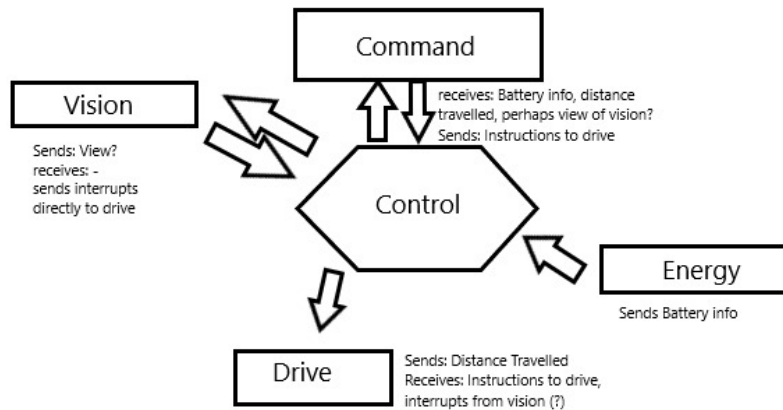


Figure 23 - Control Dataflow

To ensure functionality, the Team decided to ensure that the rover worked without autonomy for the time being.

To begin, the team assessed what transfer protocols could be used between control, drive, energy, and vision. This is examined in more detail in the inter-module communication section. These were the protocols that were decided:

Directed Sub-System	Communication Protocol
Drive	Hardware Serial UART
Vision	Software Serial UART
Energy	Hardware Serial UART
Command	TCP/IP Socket transfer

Table 12 - Control Communication

3.1.5.2 Deciding on Information flow order:

Since Control was the central hub for data flow, an order of importance first had to be established. Thus, the team decided that the following algorithm would be used:

Get battery information via UART from Energy.

1. Get Drive distance information via UART from Drive.
2. Get Vision information via Software Serial UART.
3. Perform small calculations for Vision.
4. Send interrupt (if applicable) to Drive based on Vision calculations. This overrides next steps and restarts the loop.
5. Send new instruction to Drive.
6. Connect to the server socket and send Battery information and Drive distance to the server, then receive the new instruction.

Please note that on the first start, step 6 will not actually have any information as the ESP32 does not actually connect to the server until the end of the first loop, implying that user instructions will only begin from the second loop.

The implementation of various protocols is examined in closer detail in the prior section.

3.1.5.3 Created functions:

To carry out the algorithm above, the following functions were created and used:

Function	Use Case
InitWiFi()	Used to connect the ESP32 to WiFi
Batterycheck()	Retrieves information about the battery via UART
receivedrivedist()	Retrives the distance travelled from Drive via UART
Receivevision()	Gets 4 byte word from Vision for its related calculations
Calculatevision()	Performs Vision's relevant calculations
Senddriveinstr()	Sends the new instruction from the server to Drive via Serial UART
Emergencystop()	Takes information from vision and then decides to omit user control and full emergency stop the rover
Serverconnect()	Used to connect to the command server via Sockets
Serverhandshake()	Initialized the data transfer between the ESP32 and the command server. Sends info of drive/energy and receives the new instruction

Table 13 - Control Functions.

3.3 Evaluation of Design

The evaluation of the design can be analyzed from the Rover point of view, i.e., all the subsystems working together to produce a semi-autonomous Rover. It can also be analyzed from the individual subsystems point of view, i.e., how well did each subsystem implement their functionality irrespective of the Rover (e.g., during individual testing).

The vision subsystem was tested using the camera app on a host PC. This enabled fine tuning of the colour conversion and detection algorithm. Unfortunately, the median filter was not fully integrated into the design as it had not been tested fully, thus the algorithm is susceptible to fine changes in colour in the environment. The HSV colour scheme worked well to distinguish the ping pong balls, but required a certain level of brightness in the environment.

The control subsystem was critical to operation of the rover, both autonomous and user controlled. It dictated information flow throughout the rover and had 3 main connections: Serial to Drive and

Energy, Serial to the FPGA (Vision) and TCP/IP socket transfer to Command. Unfortunately, and after multiple debugging sessions and queries to both GTAs and other groups, the team could not manage to get the Serial connections to work, alienating the Drive Arduino, and making it incapable of using user instructions from Command or interrupts from Vision. There were many functions introduced to carry out the algorithm highlighted in section 3.2.5.2, but those involving Serial Transfer were rendered unfunctional. The connection to the Command subsystem however worked flawlessly, and the ESP32 could send and receive strings with no problem. Thus, the rover was once again made only operable through direct connection to Drive's Arduino Uno. In the future, the team realized that more care should be taken for testing, and that more frequent tests should've been performed on the Control side.

The command subsystem was used to create a platform to communicate with the other subsystems via control. This subsystem was crucial in that it is the only practical way for the user of the mars rover to communicate with the rover itself, as well as receive information of the status of the rover. Most of the communication was done with the control subsystem, from sending movement related instructions to drive as well as receiving the status from drive, energy, and vision. However, due to time constraints, the front end of the command module was not completed, hence leaving the communication platform Linux terminal based.

The drive module was able to fully control the movement of the rover and provided a reliable instruction set to be used by other subsystems. Testing the drive module as part of the integration module could be done with minimal effort and the drive instructions executed as previously tested on the whole rover assembly. However, the drive module could not receive instructions from the Serial connection to the control module, even after several debugging sessions. The same instructions could be transferred over a Serial connection from the integration computer in order to aid the demonstration of the other functions of the rover.

The energy subsystem was meant to design a self-charging and discharging battery system integrated with photovoltaic cells. The battery information is supposed to provide instructions to the drive and control subsystems. However, due to the halt in any battery involved experiment to avoid any sort of danger, the battery was not complete for the rover to use.

3.4 Intellectual Property

Intellectual Property (IP) is something that is created by someone(s). There are four main types of IP protection: patents, trademarks, design and copyright. A patent consists of the protection of an invention regarding its functionality, whereas trademarks serve as a "badge of origin". A design IP application protects the appearance of an invention, while copyright protects original expression.

On the software side, there are many different IP license, however a few dominate in terms of usage, especially on Github. These include the MIT license, Apache and GPL. For the project, some source code is used from public repositories from Github under the MIT license. The MIT license is a permissive license that allows anyone to do almost anything with the code. An example of code that has been used under the MIT license is that used in the median filter implementation in the vision

subsystem. All the code produced as part of this project will be set to the MIT license; as an example the vision subsystem Github repository license text file is shown in Appendix G.

4. Conclusion

The complexity of the project structure leads to a complexity of the conclusions that can be drawn from this experience. It may be useful to compare this year's group project to the one we all took part in last year. Here, some similarities arise in the need to meet a set of functional requirements, as set by the individual specifications of each module. There are also important differences, compared to last year, as the team size doubled.

From a functional standpoint, each individual module has managed to meet the design specification. The existence of bugs is almost inevitable considering the complexity of the assembly. This is something the team has learned, and our continuous common effort is towards finding and fixing them. Possible extensions to current designs are something the team will also consider.

The team managed to work together to solve most problems that had arisen while developing each component of the rover assembly. Effective communication was key, considering the larger team size and the difference in time-zones, and the team managed to exchange ideas in an assertive and efficient manner.

Project management is also important to conclude on. Each module had to manage its own individual tasks, update their section of the Gantt chart and update the team on their current situation during our weekly meetings. All these actions have helped the team to further develop our project management skills.

Time management is an area where the project has had an influence on most of us. It has made the team significantly more aware of the importance of setting specific time constraints for tasks to be completed under.

To conclude with, the Wireless Access Point rover is proud to be able to move and see and is eager to fix its inter-module communication problems. The Wireless Access Point team members are also proud to have worked with each other and to have grown as engineers together.

5. Appendix

5.1 Appendix A: Gantt Chart

Found in Submission link

5.2 Appendix B: Github Repo

<https://github.com/pv319/WAP/Vision>

5.3 Appendix C: HSV Colour Detection Code Example

```
// Detect saturation and hue levels
assign colour_detect_s = (hsv_s > 9'd100) ? 1'b1 : 1'b0;
assign colour_detect_v = (hsv_v > 9'd80) ? 1'b1 : 1'b0;
assign colour_detect_blue_s = (hsv_s > 9'd10) ? 1'b1 : 1'b0;
assign colour_detect_pink_s = (hsv_s > 9'd60) ? 1'b1 : 1'b0;
assign colour_detect_blue_v = (hsv_v > 9'd20) ? 1'b1 : 1'b0;
assign colour_detect_orange_v = (hsv_v > 9'd120) ? 1'b1 : 1'b0;
//assign colour_detect_orange_s = (hsv_s > 9'd150) ? 1'b1 : 1'b0;

// Detect red areas
wire red_detect, red_detect_h, red_detect_hsv;
assign red_detect = red[7] & ~green[7] & ~blue[7];
assign red_detect_h = (hsv_h < 9'd20 || hsv_h > 340) ? 1'b1 : 1'b0;
assign red_detect_hsv = (red_detect_h && colour_detect_s && colour_detect_v) ? 1'b1 : 1'b0;
```

5.4 Appendix D: UART Communication Code Example

```
//Read messages from the image processor and print them on the terminal
while ((IORD(0x42000,EEE_IMGPROC_STATUS)>>8) & 0xff) { //Find out if there are words to read
    int word = IORD(0x42000,EEE_IMGPROC_MSG); //Get next word from message buffer
    if(fp){
        fprintf(fp, "%08x\n",word);
        printf("Sent word\n");
    }
    else{
        printf("Unable to connect to UART\n");
    }
}
```

5.5 Appendix E: State Machine Code

```
always@(*) begin //Write words to FIFO as state machine advances
+ case(msg_state)
+   3'b000: begin
+     msg_buf_in = 32'b0;
+     msg_buf_wr = 1'b0;
+   end
+ end*/
+   3'b001: begin
+     msg_buf_in = {8'b0, RED, red_area};
+     msg_buf_wr = 1'b1;
+   end
+   3'b010: begin
+     msg_buf_in = {8'b0, GREEN, green_area};
+     msg_buf_wr = 1'b1;
+   end
+   3'b011: begin
+     msg_buf_in = {8'b0, BLUE, blue_area};
+     msg_buf_wr = 1'b1;
+   end
+   3'b100: begin
+     msg_buf_in = {8'b0, PINK, pink_area};
+     msg_buf_wr = 1'b1;
+   end
+   3'b101: begin
+     msg_buf_in = {8'b0, ORANGE, orange_area};
+     msg_buf_wr = 1'b1;
+   end
+   default: begin
+     msg_buf_in = 32'b0;
+     msg_buf_wr = 1'b0;
+   end
+ endcase
end
```

5.6 Appendix F: Quartus Compilation Summary

Flow Status	Successful - Tue Jun 15 13:12:46 2021
Quartus Prime Version	16.1.0 Build 196 10/24/2016 SJ Lite Edition
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	11,333 / 49,760 (23 %)
Total registers	7059
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 %)

	Fmax	Restricted Fmax	Clock Name	Note
1	26.01 MHz	26.01 MHz	u0 altpll_0 sd1 pll7 clk[2]	
2	69.55 MHz	69.55 MHz	MAX10_CLK1_50	
3	80.93 MHz	80.93 MHz	MIPI_PIXEL_CLK	
4	99.78 MHz	99.78 MHz	altera_reserved_tck	•
5	112.66 MHz	112.66 MHz	u0 altpll_0 sd1 pll7 clk[3]	
6	152.42 MHz	152.42 MHz	MAX10_CLK2_50	

5.7 Appendix G: MIT Licence Example

Copyright (c) 2012-2021 Adam Horsler and others

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.