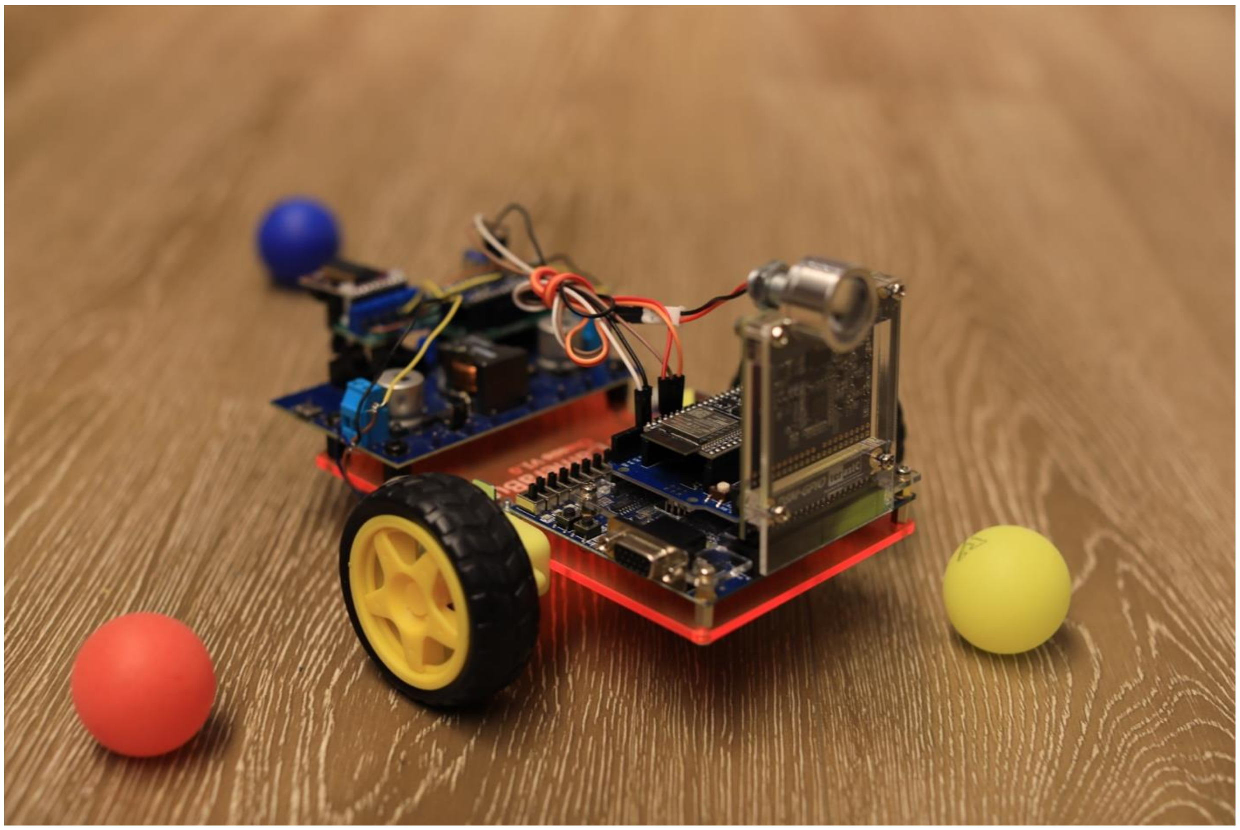
Imperial College London

ELEC50003 / ELEC50008 Computer Engineering / Design Project 2

Year 2 EEE / EIE

15 June 2021



Integration: Shuanghua Liu – sl2019
Drive: Shifan Chu – sc1019
Energy: Yingjie Qin – yq219
Vision: Alexander Pondaven – ap2619
Command: Tsz Wing Kwok – twk119
Control: Sherwin da Cruz – swd18

Table of Contents

1 Abstract

A remote-controlled and autonomous Mars rover was assembled and programmed. The rover was designed to make autonomous decisions such as avoiding obstacles and moving to a target position. This was achieved by processing camera images on a DE10-Lite FPGA to detect obstacles and sending the information to a ESP32. This microprocessor then sends movement decisions to an Arduino to move the rover and report position information. A web interface allows for monitoring the rover's position, battery usage and includes a video feed. This also allows for position remote control and adjusting object detection parameters in real time.

2 Structural Design

2.1 Data flow diagram

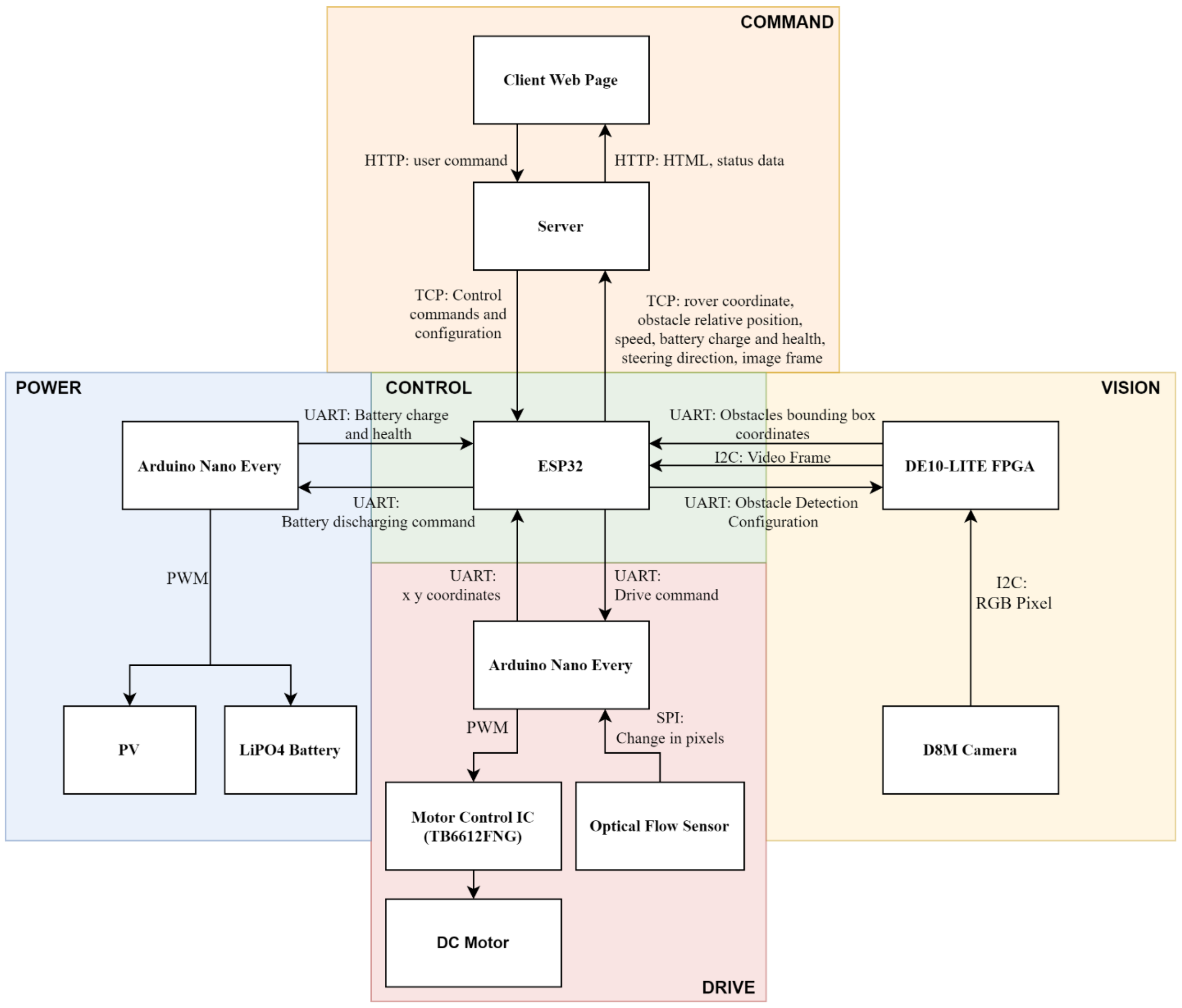

COMMAND
Client Web Page
HTTP: user command
HTTP: HTML, status data
Server
TCP: Control commands and configuration
TCP: rover coordinate, obstacle relative position, speed, battery charge and health, steering direction, image frame
POWER
CONTROL
VISION
Arduino Nano Every
UART: Battery charge and health
ESP32
UART: Obstacles bounding box coordinates
I2C: Video Frame
DE10-LITE FPGA
UART: Battery discharging command
UART: Obstacle Detection Configuration
PWM
UART: x y coordinates
UART: Drive command
I2C: RGB Pixel
Arduino Nano Every
PV
LiPO4 Battery
PWM
SPI: Change in pixels
D8M Camera
Motor Control IC (TB6612FNG)
Optical Flow Sensor
DC Motor
DRIVE

Figure 1. Structural diagram

A detailed explanation of data transfer can be found in Section 4.8, while detailed explanations on data transfer within the ESP32 can be found in Section 4.4.2.

2.2 Client Server Model

The command system follows a client-server model, with a front-end web page and back-end server. The server is hosted on a cloud IP address, accessing the specific IP address on port 3001 on the browser would direct user to the front-end web page.

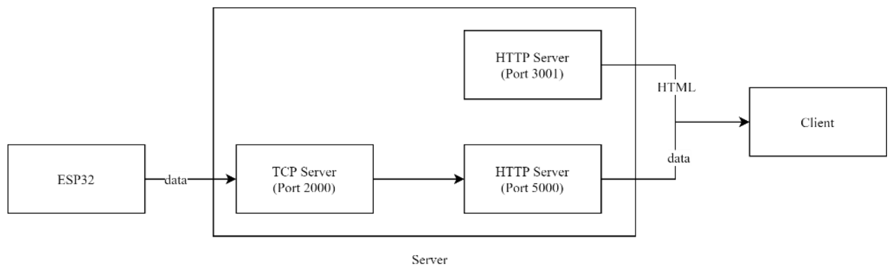


Figure 2 Client Server Architecture

The client requests data from the server and displays on the webpage. The server, on the other hand, receives message from the rover, updates its database, and sends data to clients upon request. It is possible to have more than one client, but the server is designed to only connect to one ESP32.

2.3 Webpage Design

The client webpage consists of a home page and several subpages.

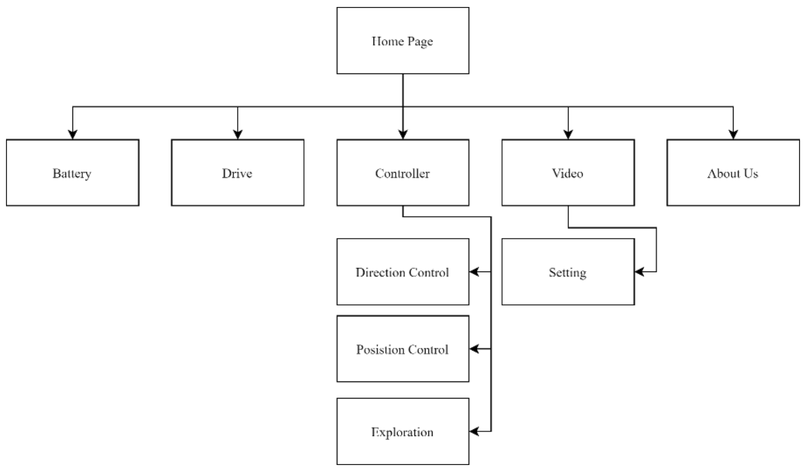


Figure 3 Web Page Site Map

Details on information displayed on each page is explained in Section 3.1.7

3 Functional Design

3.1 Functionality

3.1.1 Use Case

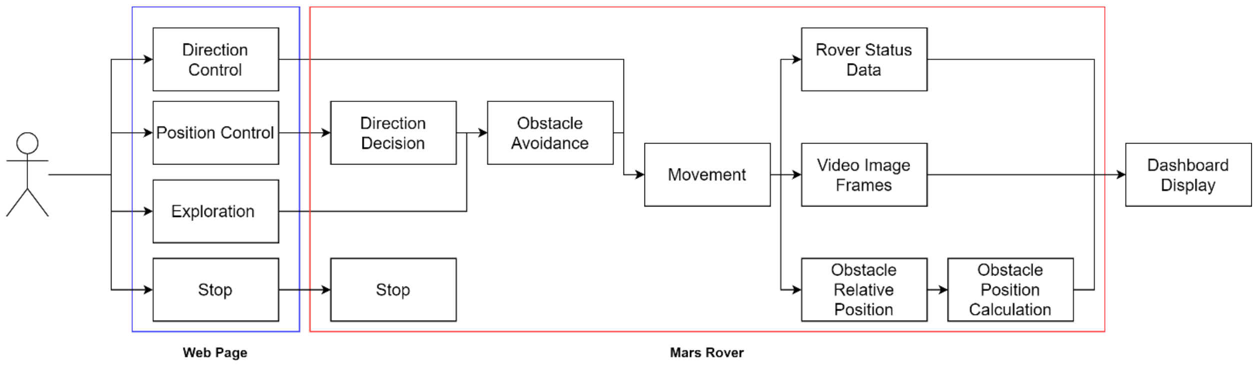


Figure 4 Use Case Diagram

3.1.2 Control Modes of the Rover

3 control modes are available for user to choose, being Direction Control, Position Control and Exploration.

In direction control, user decides the direction rover goes: forward, backward, left and right. This mode has the rover totally controlled by human. In position control, user inputs a coordinate and the rover would directs itself towards that coordinate while avoiding obstacles. Human decides on the destination of the rover, but the rover decides on the path. Last but not least, exploration mode lets the rover wanders around to explore its local working area.

Details implementation on control modes are explained in Section 4.4

3.1.3 Obstacle Avoidance

The rover can avoid obstacles in exploration mode and position control mode. The ESP32 obtains bounding box information from the FPGA and uses it to avoid obstacles while moving.

3.1.4 2D Map Construction

The rover can detect and calculate the distance of the obstacles from the rover and its angle. Alongside with position of the rover, measured by an optical flow sensor, the system could calculate the positions of the obstacles. In real-life scenario, the rover could be used to survey environments that human cannot reach.

3.1.5 Dynamic Colour Configuration

To improve accuracy of obstacles detection in different lighting environment, dynamic colour configuration on the vision subsystem is supported. User could change the gain, exposure, HSV range of each colour on the webpage, and affecting the FPGA's decision on obstacle detection.

3.1.6 Video Streaming

The rover streams colour video at QVGA resolution to the webpage, at about 0.5 FPS. This allows the user to see where the rover is and make other decisions such as how to move. Additionally, the user can visually see how well the obstacle detection is doing in the current circumstances and adjust accordingly. For example, if there is a sudden sandstorm on mars and the image becomes very dark, the user can increase the gain settings to improve obstacle detection in real time.

3.1.7 Webpage Display

A webpage dashboard is used to display rover's status data and allow users to input command.

The home page provides a brief overview of each subsystem, clicking into each card will bring the user to the respective subpage. An expandable sidebar also allows user to jump to different subpages.

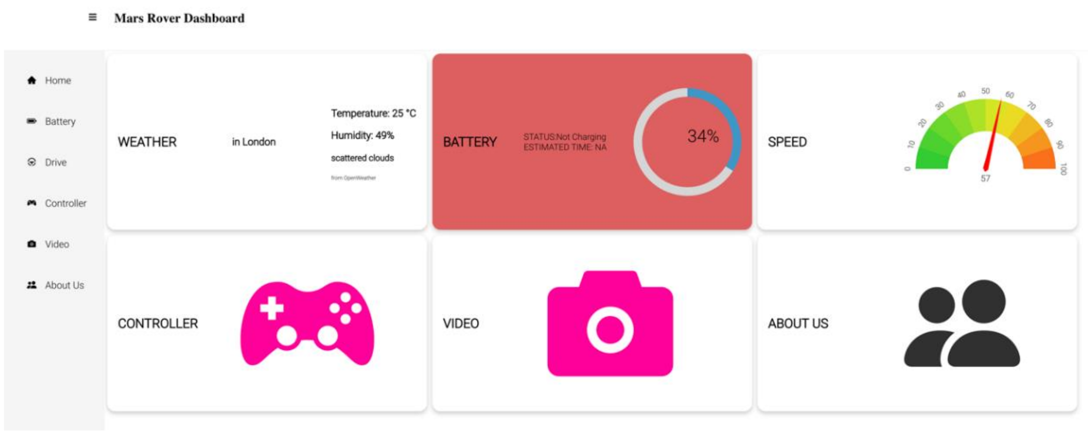
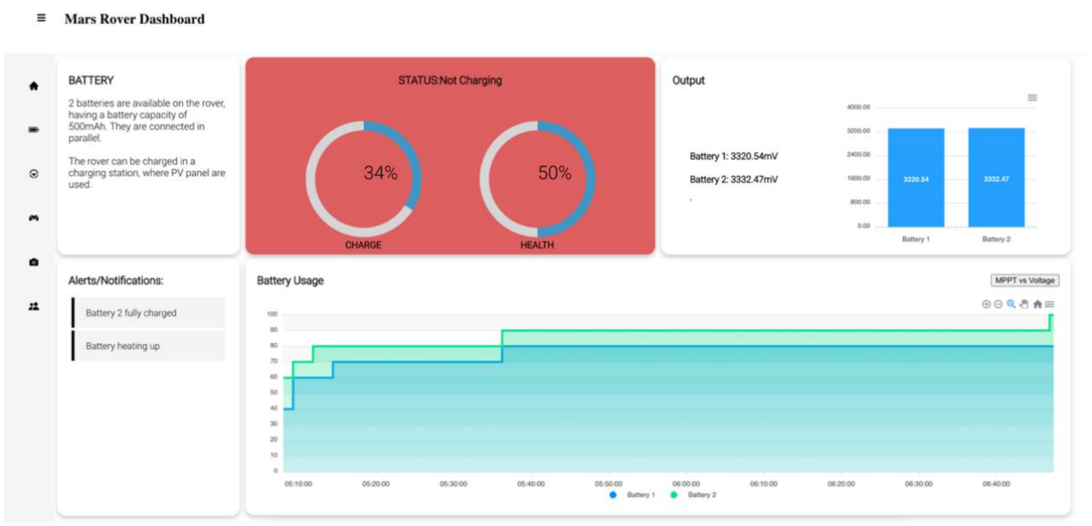


Figure 5 Web Page Home Subpage

The Battery subpage provides details on properties and information of the energy subsystem, including State of Charge, State of Health, battery usage history and power-voltage graph. It also has a notification display to alert user such as low battery charge or heating up of battery.


Mars Rover Dashboard

BATTERY

2 batteries are available on the rover, having a battery capacity of 500mAh. They are connected in parallel.

The rover can be charged in a charging station, where PV panel are used.

STATUS:Not Charging

34% CHARGE

50% HEALTH

Output

Battery 1: 3320.54mV

Battery 2: 3332.47mV

Alerts/Notifications:

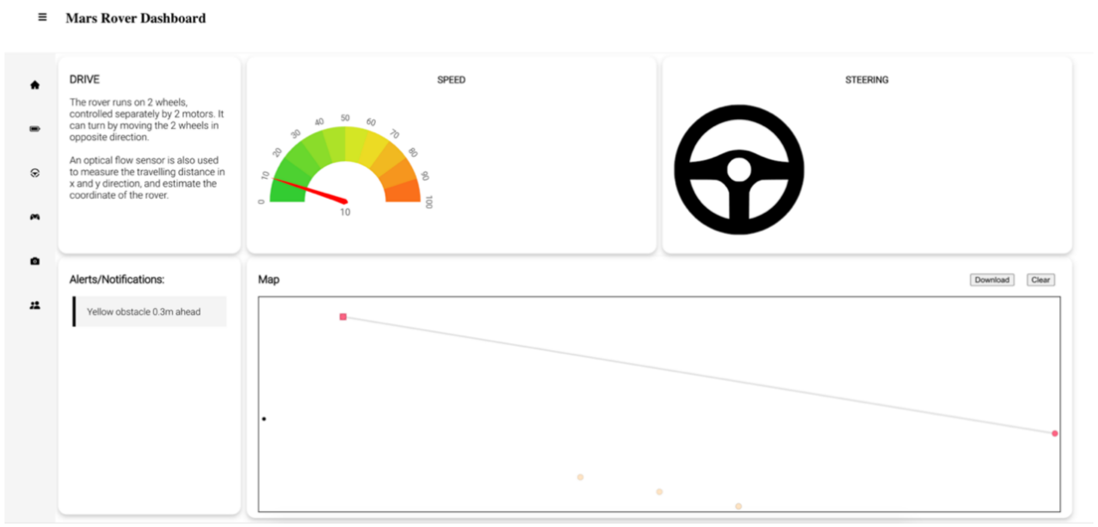
Battery 2 fully charged

Battery heating up

Battery Usage

Figure 6 Web Page Battery Subpage

The drive subpage, on the other hand, provides details of the drive subsystem. It displays the speed and steering of the rover. A 2D map, containing obstacles' positions, rover's past and current position, is also displayed on the page.


Mars Rover Dashboard

DRIVE

The rover runs on 2 wheels, controlled separately by 2 motors. It can turn by moving the 2 wheels in opposite direction.

An optical flow sensor is also used to measure the travelling distance in x and y direction, and estimate the coordinate of the rover.

Alerts/Notifications:

Yellow obstacle 0.3m ahead

Figure 7 Web Page Drive Subpage

The Video subpage provides real-time video streaming from the camera in the vision subsystem. The video is in resolution 320x240 with a framerate of around 0.5 frames per second.

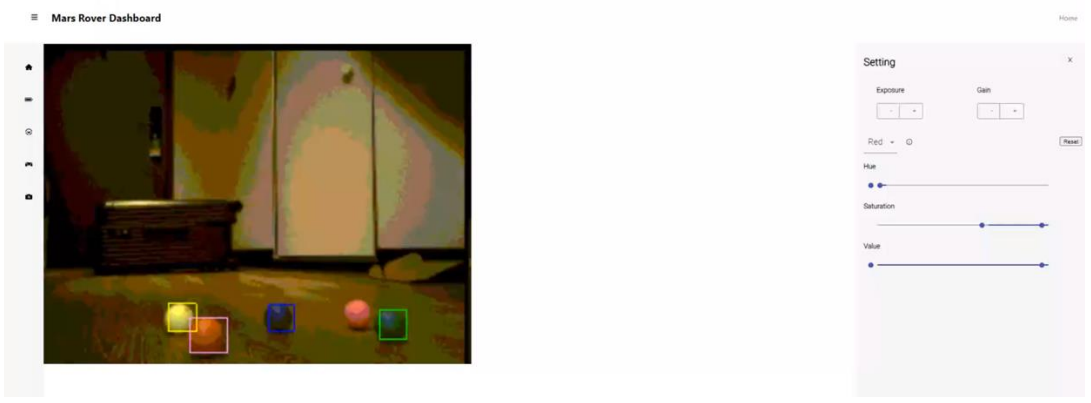


Figure 8 Web Page Video Subpage

The controller subpage allows user to remotely control the rover with 3 different control modes available.

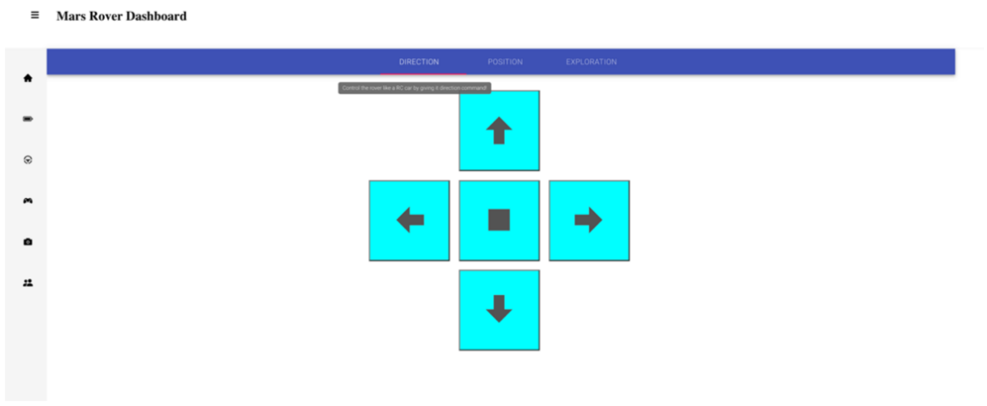

Mars Rover Dashboard
DIRECTION
POSITION
EXPLORATION
Control the rover like a RC car by giving it direction command!

Figure 9 Web Page Controller Subpage

3.2 Functional / Non-Functional requirements

3.2.1 System requirements:

Subsystem	Functional requirements	Non-Functional requirements
Vision	a. Processor should send coordinates of five bounding boxes to the ESP32, corresponding to the position of five uniquely coloured ping pong balls (red, pink, yellow, green, blue). b. If the coloured ping pong ball does not exist in the image, the coordinates for that colour should be zero. c. The hexadecimal displays, switches, and buttons must be used to allow for rapid testing and feedback. d. The image processing module must be able to output the image frame with bounding boxes around the detected ping pong balls. e. The NIOS II must be able to send data (e.g. bounding box coordinates) and read data (e.g. camera settings) from the ESP32.	a. Be able to perform well in different light conditions. b. The DE-10 Lite should read RGB pixels from the D8M camera at a rate of 60 fps. c. The output image should not have glitches due to timing issues and should look like the input image. d. The FPGA switches should enable switching between different image outputs like grey-scale image and the image with boundary boxes. e. Bounding boxes should completely surround the corresponding coloured ping pong ball
Control	a. Data sent from one subsystem to another through the ESP32 should be sent correctly b. ESP32 should be able to obtain obstacle data form FPGA and make movement decisions to avoid them.	a. Data should be sent with a minimum delay b. Certain data (ie commands for remote control) should be prioritised over other data (ie Video data) to minimise latency.
Video Streaming	a. The FPGA should be able to send images to the ESP32 which should send them to the web server for displaying	a. Video streaming should use minimal CPU processing on the Nios II, ESP32 and server. b. Video streaming should use a small network bandwidth
Command	a. Displays rover's status diagram (e.g. speed, position) b. Provide interface for user to remotely control the rover	a. Data should be displayed on web page with minimum delay
Drive	a. Optical flow sensor should work properly with motor, to achieve a better motion control. b. Position Data should be send correctly to esp32. c. Communication through serial monitor should receive a decoded data. d. Position should be controlled base on received instruction. e. Angle should be controlled base on received instruction.	a. Have less delay in the control system. b. Accurate position sensing c. Accurate position control. d. Accurate angle control.
Power	a. Balanced battery voltage and capacity remaining should constantly be monitored b. Battery charging method should be properly designed c. PV should charge battery at maximum power point	a. Battery and PV should be connected in the way to have ideal and practical performance

4 Implementation

4.1 Vision Subsystem

4.1.1 Overview

4.1.1.1 Aim

The Vision subsystem uses the video input from the D8M camera (OV8865). The DE-10 Lite FPGA aims to decrease the large data flow from the camera by finding just the coordinates of bounding boxes around ping pong balls to be sent to the ESP32 microcontroller. There are five ping pong balls that need to be identified, each having a distinct colour (red, pink, yellow, green, blue). The project code was built off Edward Stott's GitHub code [1].

4.1.1.2 Environment

The NIOS II processor was programmed on Eclipse and the Verilog image processing project was written and compiled in Quartus v16.1.

4.1.1.3 Design Architecture

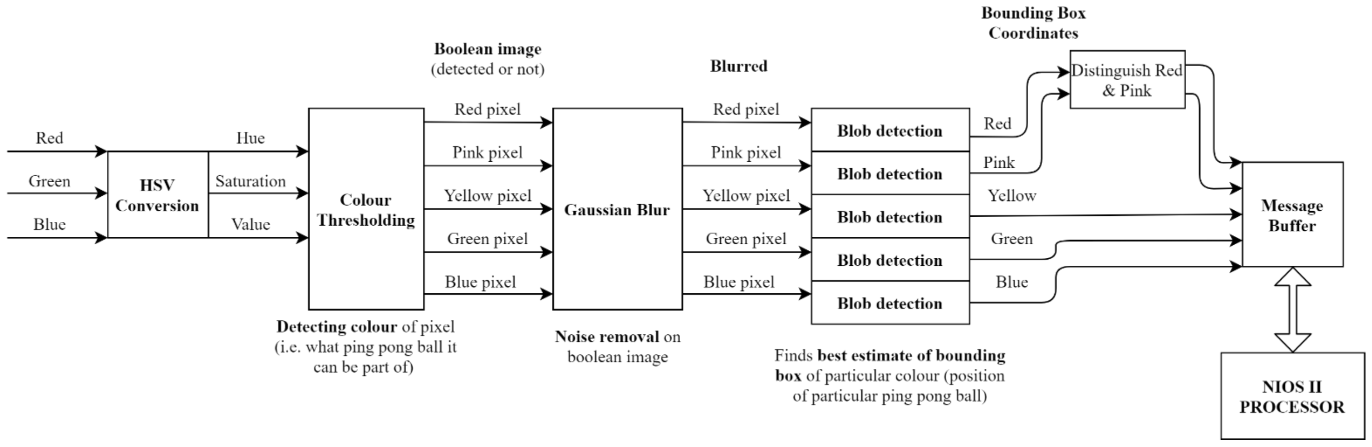


Figure 10. Image Processing (EEE_IMGPROC) Block

The overall structure of the video pipeline is seen in Figure [1]. The architecture of the image processing (EEE_IMGPROC block) done to compute the bounding box coordinates from the RGB pixels is outlined in Figure .

4.1.2 Image Processing

4.1.2.1 Pre-processing

Several pre-processing algorithms were explored to improve the performance of bounding box detection and reduce the high frequency noise that appeared on the poor quality 640x480 pixel image. Average or gaussian blur algorithms used a 3x3 window that convolved with the image pixels i.e., took a linear combination of nine values with the weights defined by the algorithm's window/kernel (demonstrated in [2] with kernel "sliding" across source image).

Image pixel data was processed one pixel at a time (no random access of pixels in frame), so as the pixels entered the processing module, the pixel values needed to be stored. For a 3x3 window, the previous two rows needed to be stored to operate on the pixels above the current pixel. Initially, the values were stored in 2D register arrays in Verilog, which acted as shift registers as they shifted the pixels upwards as new data entered the pipeline. In this method, the linear combination of values (multiplying/dividing weights and adding) was all done in one cycle, leading to timing issues due to the large critical path and glitches seen in Figure . After optimising the calculations by bit slicing (removing division) and using the fact that the matrix calculation was separable (discussed below), the 3x3 convolution worked, but 5x5 convolution increased the critical path too much again. The calculations could be pipelined, but this method was messy in terms of registers. This made it difficult to find errors, especially due to 10-minute compilation times.

A cleaner, more modular window generation method was developed to allow testing separate from the entire pipeline. This involved storing the rows in two FIFOs (first in first out) in RAM that stored the entire two previous rows. This was implemented in Verilog with the FIFO module, where FIFOs were written to initially until they had 639 entries (one row of image data), and then they were written to and read from every clock cycle. This module (seen in Figure) could then be used in a separate module to implement different algorithms, accessing all nine pixels in the same cycle. The module could easily be debugged and simulated on a stream of consecutive numbers individually to verify quickly if it worked (simulation results seen in Figure).

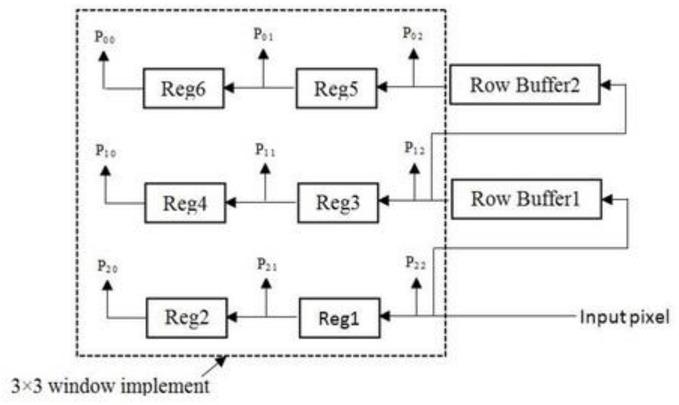


Figure 11. 3x3 Window Generation Module [3]

The row buffer logic was done using the “almost full” FIFO signal to check when the FIFO had been filled with one row of pixels (639 entries) as seen in the window module code in Figure 58. Note that the second-row buffer read and wrote if it had been filled with a row of pixels. If it had not been filled yet, but the first buffer was filled, the second buffer was only written to (not read from) so that it could be filled. Otherwise, if the first buffer was not filled either, the second buffer was not read or written to, to allow the entries to fill the first buffer first.

The 3x3 convolution algorithm was determined by the weight matrix. An average blur would just have 1/9 as the weights for each entry as it averaged all nine entries. Gaussian blur had a higher weight for the centre pixel, and lower for the outer pixels as seen in Figure [4]. The Gaussian blur window matrix is separable (separated into two different linear operations), which allowed the input pixels to be convolved horizontally (convolve three rows) and then those results to be convolved vertically (one convolution) as seen in the code in Figure 12. This reduced the complexity of the algorithm from O(n^2) (multiplying by each weight and adding all nine values) to O(n), which could help even more for larger convolutions like a 5x5 window. The 3x3 window was used in the final product rather than a 5x5 window as that would have required four FIFOs to store four rows and the memory usage needed to be minimised to fit.

```
//row convolutions
  p0 <= p00 + 2*p01 + p02;
  p1 <= p10 + 2*p11 + p12;
  p2 <= p20 + 2*p21 + p22;

  //convolve rows
  p <= p0[9:2] + 2*p1[9:2] + p2[9:2];
```

Figure 12. Gaussian blur separable code

Gaussian blur was found to produce the best results as it tended to preserve the shapes of edges better than a simple average blur. This allowed for random noise to be removed while the important edges around the ping pong balls were preserved. The blur performed best by applying it to a Boolean image (acting on only particular detected pixels e.g. only very red detected particles (explained in Section 4.1.2.2)). Edge detection or filters on the grey-scale image performed poorly due to the poor quality of the image and the large amount of high frequency noise. Applying the blur to just detected pixels (Boolean image) allowed unwanted noise to be “smoothed out” and then a threshold was applied to only detect pixels that were of high intensity after the blur. This improvement can be seen in Figure .

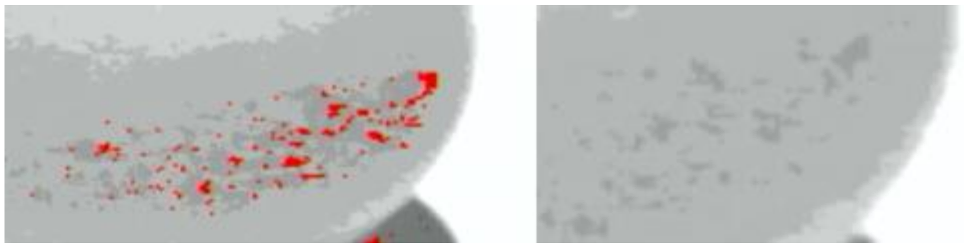


Figure 13. Before (left) and after (right) blurring detected red pixels.

Exposure and gain were very important settings that impacted detection. They often needed to be tweaked in the NIOS II code by inputting a character to either increase or decrease the values. This was then done through the webpage, so the user could set these values, as the algorithm’s performance was quite dependent on lighting conditions.

4.1.2.2 Colour detection

The aim of colour detection was to maximise the number of detected pixels on a particularly coloured ping pong ball (cluster of pixels) and to minimise the number of pixels detected in the background (false positives). These clusters of detected pixels could then be used to find the bounding box of the ball (detailed in Section 4.1.2.3).

4.1.2.2.1 Basic colour detection

Initially, each pixel from the camera has a red, green and blue value (each 8-bit). These values were very susceptible to different lighting conditions but could be used for simple detection algorithms. The basic detection used in Edward Stott's code [1] involved checking that the most significant bit of the red value was asserted, while the most significant bits for green and blue were unasserted, which detects pixels that are more red. This could be used for very basic red ping pong ball detection, to test bounding box detection and communication between subsystems early on.

This red ball detection performed badly as it could not differentiate the red ball and pink ball well. The camera also had artifacts or rings around the centre of vision with decreasing brightness, and this made it difficult for the ping pong ball to perform well in the centre of the screen (bright) as well as in its peripheral vision in the corners of the image where it appeared darker. Taking a similar approach for the other colours like yellow (high red and green values) performed even worse as the yellow light reflecting off the white table made the entire table under the ping pong balls get detected as yellow. Similarly, the blue ball appeared almost black on the image if the gain was not high enough, which made it difficult to differentiate between black items in the environment. A contrast function [5] was also implemented (see Figure) to try to make it easier to differentiate colours in the image. Although, this was still very light sensitive, and it was hard to configure the detection for all the ball colours.

4.1.2.2.2 HSV implementation

Converting from RGB to the HSV (Hue, Saturation, Value) colour space was the best way to detect the most pixels of a particular colour on a ball. The hue represents the colour, the saturation represents the amount of grey, and the value represents the brightness [6]. A RGB to HSV converter module (RGB2HSV.v) was created to be tested individually. Hue took into account all red, green, and blue values to calculate the colour, and a range of hues could be taken to select a particular colour as the ping pong ball may have had slightly different hues depending on lighting conditions. A range of saturation and values were also given to select the same colour but in different light conditions. This allowed selection of pixels of the same colour but with a different brightness, which was useful to find the ball both when it was in the centre as well as in the corners of the frame where it is darker. The selection of a range of saturation helped when differentiating between pink and red, since they both had the same hue, but pink had a lower saturation than red. Yellow also had a larger saturation than the white background, which made distinguishing them very easy by selecting a range of saturations.

The hue, saturation, and value were all made to be in the range 0-255 through the hardware implementation of the HSV equations [7]. This made it very easy to have the HSV values as output pixels, so the grey-scale image would help visualise how they change in real-time by the intensity/brightness of each pixel (very helpful for Section 4.1.2.2.3). The implementation was heavily pipelined as it used division and multiplication that caused timing problems if done in a single cycle. An overview of the hardware can be seen in Figure .

4.1.2.2.3 HSV Range Selection

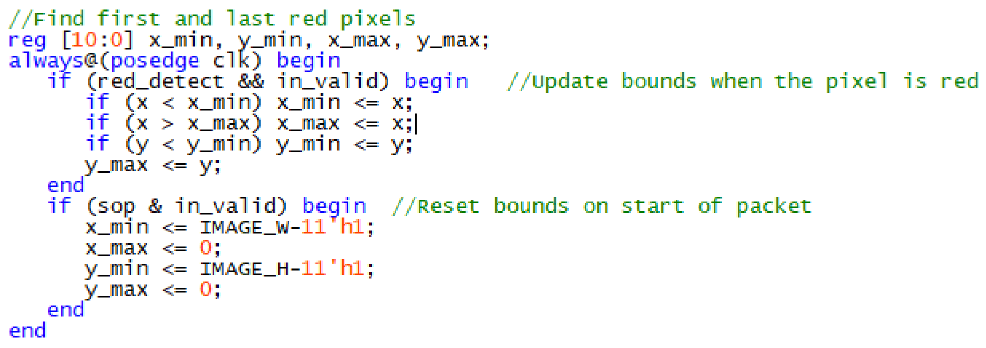
Selecting these ranges for hue, saturation, and value was done experimentally. It would have been inconvenient to try values in Verilog and compile the project each time, so memory mapped registers were used, which stored the upper and lower bounds for the HSV values. These values could be adjusted in the NIOS II code in Eclipse for rapid testing. One of the switch configurations showed the pixels that were detected given a particular set of HSV ranges, and this could be tweaked to maximise the number of selected pixels for the particular ball colour, while minimising any false positive detections.

Care had to be taken to adjust exposure and gain correctly. A very high gain would make the yellow ball appear the same as the background, although if the gain was too low, the blue ball would be indistinguishable from any black object in the frame. Different lighting conditions could require different gain and exposure, as well as slightly different HSV ranges. This was even more of an issue since a separate team member, who had very different lighting conditions (different time of day and lights), needed to test the ball detection. Therefore, the ability to adjust gain, exposure, and the minimum and maximum HSV values through the webpage was implemented (see Section 4.2) to allow the ball detection performance to be maximised in all conditions. The performance of the ball detection was mainly dependent on how well the coloured pixels were detected, so this flexibility was very important. The Mars rover must be able to work in different lighting conditions all throughout different times of the day.

4.1.2.3 Bounding box detection

4.1.2.3.1 Basic detection

Basic bounding box detection in the starting code [1] was implemented by updating the four boundaries of the box: minimum x and y coordinates, and maximum x and y coordinates. The x and y coordinates were computed as valid pixels enter the image processing in accumulators, where x increased from left to right and y increased from top to bottom, so the origin was at the top left corner of the frame. For every detected pixel, the bounding box size was increased by updating the values of the bounds as seen in Figure . At the start of the frame (sop signal asserted), the bounds are reset.



```
//Find first and last red pixels
reg [10:0] x_min, y_min, x_max, y_max;
always@(posedge clk) begin
   if (red_detect && in_valid) begin   //Update bounds when the pixel is red
      if (x < x_min) x_min <= x;
      if (x > x_max) x_max <= x;
      if (y < y_min) y_min <= y;
      y_max <= y;
   end
   if (sop & in_valid) begin  //Reset bounds on start of packet
      x_min <= IMAGE_W-11'h1;
      x_max <= 0;
      y_min <= IMAGE_H-11'h1;
      y_max <= 0;
   end
end
```

Figure 14. Basic bounding box detection [1]

The basic red detection performed badly as it updated the same bounding box for every detected pixel in the image. This meant that it was very susceptible to noise as any new detected red pixel in any part of the image would change the bounding box. It could also not differentiate different “blobs” or clusters of red pixels as it very simply put every red pixel in the frame in a large box as seen in Figure . Therefore, it would still not perform well with noise removal, as only the best estimate of the ping pong ball boundary box should be returned.

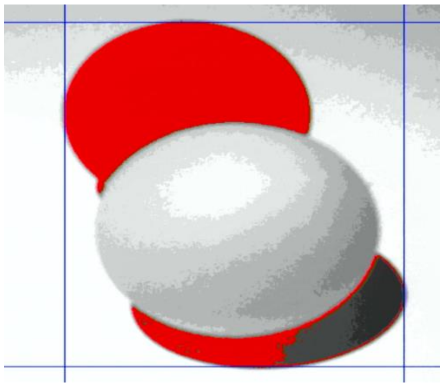
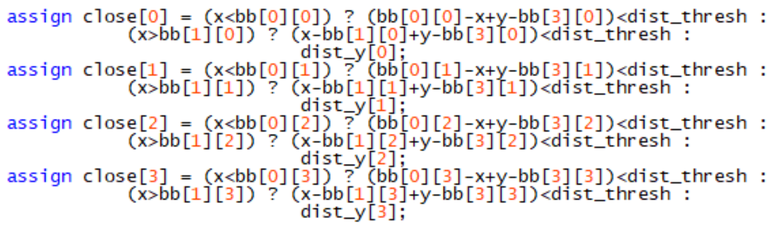


Figure 15. Basic red detection image

4.1.2.3.2 Multiple bounding box detection

To be able to differentiate clusters of detected pixels, multiple possible boundary boxes must be detected (where the best one can be selected according to specific criteria as seen in Section 4.1.2.3.3). The algorithm implemented had space for storing four different bounding boxes (as it would take a lot of memory to store all possible bounding boxes in the image).

For a given detected pixel, a new bounding box was created with the x coordinate set to x minimum and maximum and the y coordinate set to the y maximum and minimum bounds. If the next detected pixel was within a specific distance of the previous bounding boxes that had been created, it was added to the first bounding box that was within the threshold distance. This was done by changing the bounds of the bounding box to include that pixel. If the next detected pixel was not within the threshold distance to any previous created bounding boxes, a new bounding box was created at the new pixel’s coordinates. The ‘close’ signal detected if any new pixels was within the distance threshold to any of the four bounding boxes as in Figure . If the new pixel was to the right/left of the bounding box, the distance was the sum of the x-difference and y-difference to the bounding box. Otherwise, if the new pixel is below the bounding box, it was just the vertical distance from the bounding box maximum y (bottom boundary).

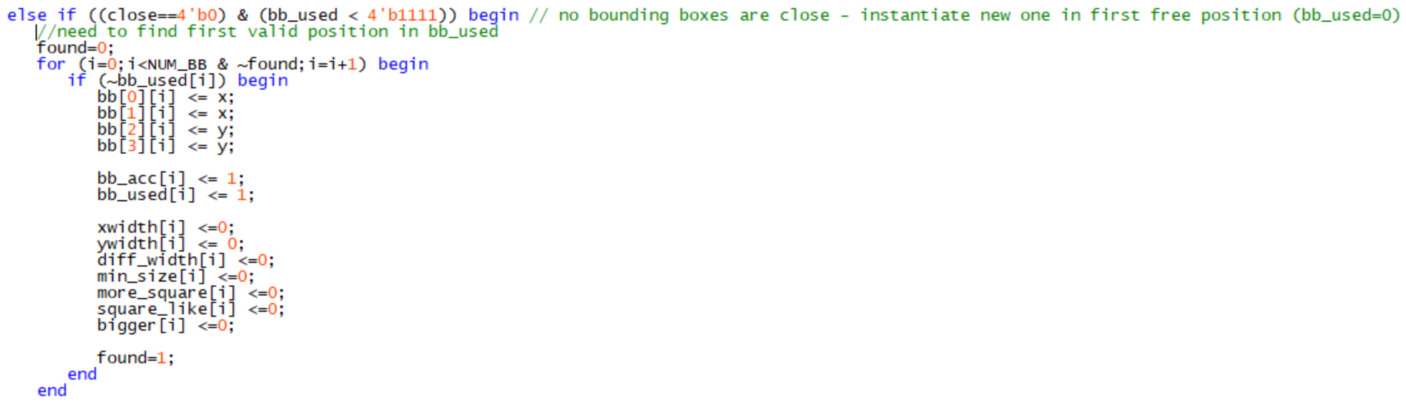


```
assign close[0] = (x<bb[0][0]) ? (bb[0][0]-x+y-bb[3][0])<dist_thresh :
                  (x>bb[1][0]) ? (x-bb[1][0]+y-bb[3][0])<dist_thresh :
                               dist_y[0];
assign close[1] = (x<bb[0][1]) ? (bb[0][1]-x+y-bb[3][1])<dist_thresh :
                  (x>bb[1][1]) ? (x-bb[1][1]+y-bb[3][1])<dist_thresh :
                               dist_y[1];
assign close[2] = (x<bb[0][2]) ? (bb[0][2]-x+y-bb[3][2])<dist_thresh :
                  (x>bb[1][2]) ? (x-bb[1][2]+y-bb[3][2])<dist_thresh :
                               dist_y[2];
assign close[3] = (x<bb[0][3]) ? (bb[0][3]-x+y-bb[3][3])<dist_thresh :
                  (x>bb[1][3]) ? (x-bb[1][3]+y-bb[3][3])<dist_thresh :
                               dist_y[3];
```

Figure 16. Distance threshold calculation code

The described approach would calculate the bounding boxes of only the upper four clusters of detected pixels, and then it would have no space to store any more. Therefore, a way to remove bounding boxes that were clearly worse than others was necessary. Removing a bounding box could only be done once the y-distance from the maximum y bound was greater than the threshold distance (as all further pixels in the frame would exceed the distance threshold and never update the bounds). At this point, the bounding box could be judged against the current best bounding box to decide if it should either be deleted or replaced as a better bounding box. The comparison between the current bounding box and the current best bounding box was done using specific criteria outlined in Section 4.1.2.3.3.

A four-bit number, bb_used, was used to show what boundary boxes are active at any given time. It started at 4'b0000 as no boundary boxes were initialised at the start of the frame. When a new boundary box was created, the first unasserted bit in bb_used was asserted e.g. 4'b1000. When a boundary box is finished (removed or moved to best bounding box), the corresponding bit in bb_used was set to zero. This freed up the space for new bounding boxes to be created by detecting any unasserted (0) bits in bb_used as seen in the code in Figure . This meant that there was a maximum of four bounding boxes that could be stored at any given time, although the pre-processing steps made sure that the number of random clusters of detected pixels was minimised (e.g. noise).



```
else if ((close==4'b0) & (bb_used < 4'b1111)) begin // no bounding boxes are close - instantiate new one in first free position (bb_used=0)
    //need to find first valid position in bb_used
    found=0;
    for (i=0;i<NUM_BB & ~found;i=i+1) begin
        if (~bb_used[i]) begin
            bb[0][i] <= x;
            bb[1][i] <= x;
            bb[2][i] <= y;
            bb[3][i] <= y;

            bb_acc[i] <= 1;
            bb_used[i] <= 1;

            xwidth[i] <=0;
            ywidth[i] <= 0;
            diff_width[i] <=0;
            min_size[i] <=0;
            more_square[i] <=0;
            square_like[i] <=0;
            bigger[i] <=0;

            found=1;
        end
    end
```

Figure 17. Create new bounding box code

Multiple bounding boxes can be seen working in Figure .

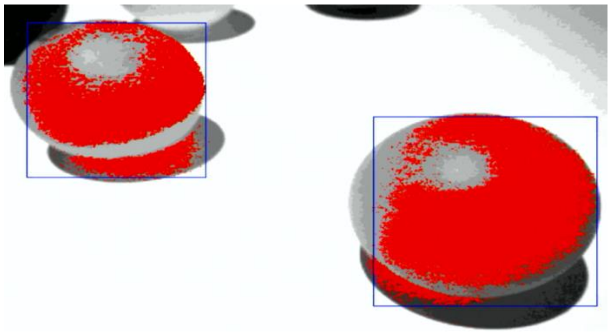
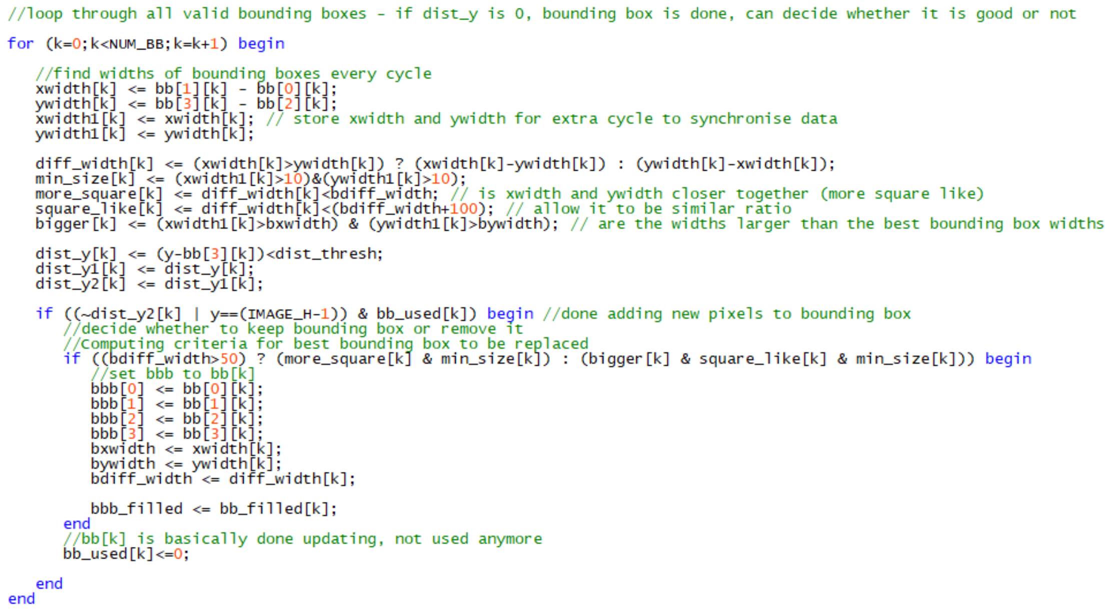


Figure 18. Multiple bounding box detection

4.1.2.3.3 Criteria for best bounding box

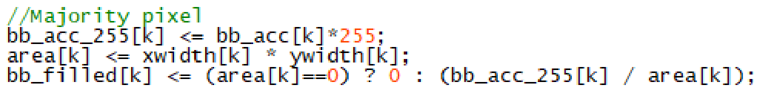
The boundary box detection module needed a way to judge whether a boundary box is "better" than the best boundary box. A more square-shaped boundary box was preferred as this tended to find the clusters of detected pixels rather than a line of connected pixels (that was often just noise). Thus, if the best boundary box was very rectangular (not square), the more square-shaped boundary box was preferred. If the best boundary box already had a square shape (within some margin), the current boundary box was decided to be "better" if it is square (or square_like in the code) and bigger as well. The algorithm aims to find the biggest and most square boundary box in the image, which performed very well given the pixels on the ball were detected properly. It also performed well in avoiding noise by ignoring rectangles and having a minimum size criterion that ensured that the widths of the box were at least 10 pixels (to avoid very small square-shaped clusters of detected pixels that were actually noise). Each step of these criteria was pipelined to minimise critical path as seen in Figure .



```
//loop through all valid bounding boxes - if dist_y is 0, bounding box is done, can decide whether it is good or not

for (k=0;k<NUM_BB;k=k+1) begin

   //find widths of bounding boxes every cycle
   xwidth[k] <= bb[1][k] - bb[0][k];
   ywidth[k] <= bb[3][k] - bb[2][k];
   xwidth1[k] <= xwidth[k]; // store xwidth and ywidth for extra cycle to synchronise data
   ywidth1[k] <= ywidth[k];

   diff_width[k] <= (xwidth[k]>ywidth[k]) ? (xwidth[k]-ywidth[k]) : (ywidth[k]-xwidth[k]);
   min_size[k] <= (xwidth1[k]>10)&(ywidth1[k]>10);
   more_square[k] <= diff_width[k]<bdiff_width; // is xwidth and ywidth closer together (more square like)
   square_like[k] <= diff_width[k]<(bdiff_width+100); // allow it to be similar ratio
   bigger[k] <= (xwidth1[k]>bxwidth) & (ywidth1[k]>bywidth); // are the widths larger than the best bounding box widths

   dist_y[k] <= (y-bb[3][k])<dist_thresh;
   dist_y1[k] <= dist_y[k];
   dist_y2[k] <= dist_y1[k];

   if ((~dist_y2[k] | y==(IMAGE_H-1)) & bb_used[k]) begin //done adding new pixels to bounding box
      //decide whether to keep bounding box or remove it
      //Computing criteria for best bounding box to be replaced
      if ((bdiff_width>50) ? (more_square[k] & min_size[k]) : (bigger[k] & square_like[k] & min_size[k])) begin
         //set bbb to bb[k]
         bbb[0] <= bb[0][k];
         bbb[1] <= bb[1][k];
         bbb[2] <= bb[2][k];
         bbb[3] <= bb[3][k];
         bxwidth <= xwidth[k];
         bywidth <= ywidth[k];
         bdiff_width <= diff_width[k];

         bbb_filled <= bb_filled[k];
      end
      //bb[k] is basically done updating, not used anymore
      bb_used[k]<=0;

   end
end
```

Figure 19. Finding best boundary box code

The accumulation of pixels in each boundary box was investigated as a criterion. This calculated the concentration of detected pixels in the box by diving the number of pixels by the bounding box area, although this resulted in a large increase in the number of resources required. The concentration of pixels was calculated as in Figure , although it did not seem to improve the algorithm due to how often bounding box change every frame, so it was not used in the final project.



```
//Majority pixel
bb_acc_255[k] <= bb_acc[k]*255;
area[k] <= xwidth[k] * ywidth[k];
bb_filled[k] <= (area[k]==0) ? 0 : (bb_acc_255[k] / area[k]);
```

Figure 20. Concentration of boundary box pixels code

Since red and pink had the same hue with different saturations, there were often detected red and pink pixels on both corresponding balls. This worked decently when both red and pink balls were in the image since most detected pixels were on the correct ball, so the best boundary box detection algorithm chose the correct ball. Although, for example if only the red ball was on the screen, the red was detected well, although there were some pink detected pixels on the red ball, so there was a small pink bounding box even when the pink ball was not there (seen in Figure). This false positive was important to fix as otherwise it looked like there is a pink ball in the distance behind the red ball, which would make the 2D map incorrect. This was fixed by checking if red and pink boundary boxes overlap within some margin and if so, only choosing the larger bounding box (code in Figure).

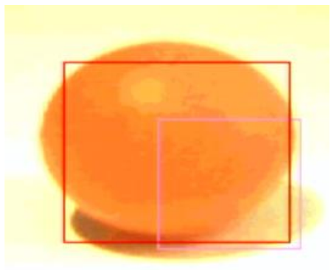


Figure 21. Red and pink differing

All boundary boxes detected can be seen detected correctly in Figure 62.

4.1.2.4 3D position of bounding box

The bounding box coordinates were used to calculate the approximate distance to the ball and the angle from the centre of the screen (described in Section 4.4.3.2). This calculation required the arctangent function and division, which made the NIOS II require too much memory, so it was decided to move these computations to the ESP32, which had its own divider hardware and more resources.

4.2 Dynamic Colour Detection Configuration

4.2.1 Aim

The aim is to improve accuracy of detecting obstacles in the vision subsystem. Different lighting conditions may vary the performance of obstacle detection. Being able to adjust the settings (exposure, gain, HSV ranges) in obstacle detection dynamically would allow the performance to be improved for all lighting environments (see Section 4.1.2.2.3).

4.2.2 Tuning Configuration from Web Page

Settings are changed by user on the Video subpage, hence user could monitor to effect with video streaming present on the page.

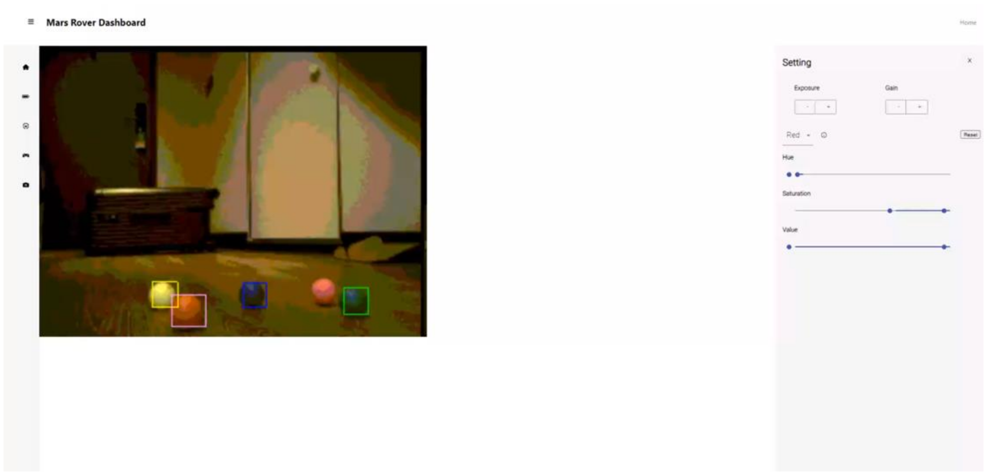


Figure 22 Video Page Configuration

Exposure and gain can be incremented and decremented by clicking on the two sets of +/- button at the top.

Each obstacle colour has its own HSV range, and it can be selected by user by the double slider. The dropdown menu allows user to choose the colour, sliders value would then change accordingly. The reset button would reset all HSV values of selected colour to the default value. Below table shows the default range for each colour:

Colour	Hue	Saturation	Value
Red	0-14	166-255	0-255
Pink	0-14	51-174	96-255
Yellow	32-53	144-255	0-255
Green	85-106	76-179	0-179
Blue	128-255	0-96	16-176

Table 1 Default HSV Ranges of Each Colour

Any change by the user would be detected by client browser and sent to the server.

4.2.3 Receiving and Sending Changes from Server

Changes received from the client are saved and queued in the server as an array of objects in the server. The server would then send the changes in a TCP packet to ESP32 upon request, possibly containing more than one change. The array of changes would then be cleared after sending out the packet, accumulates changes and waits for the next TCP request.

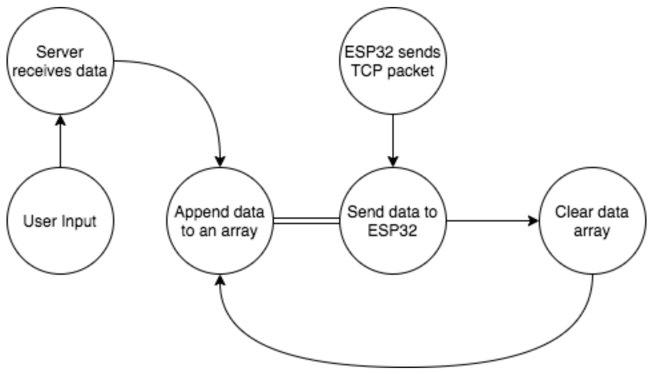


Figure 23 Timing of Data Transfer in Server

As any user inputs do not get to the rover immediately, it takes at most one TCP request cycle, changes are accumulated before sending out. For example, exposure is incremented twice and decremented once in one cycle, incrementing exposure by one will be sent to the ESP32 instead of 3 separate commands. The ESP32 task receiving TCP packets then places these requests in a FIFO queue to be read and transmitted by the FPGA UART task.

4.3 Video Streaming Subsystem

4.3.1 Aim

The aim is to enable video streaming by transmitting picture frames from the FPGA to the ESP32 before being transmitted by TCP/IP for displaying. The initial aim was to transmit RGB colour images with 4-bit colour depth (12-bit pixels) at VGA resolution (640 by 480 pixels), matching the VGA display output. Ideally minimal processing should be required by the Nios II processor, ESP32 and the Server to display the images.

4.3.2 Approach

To test that the ESP32 can correctly transmit images, the Test Pattern Generator II IP Core was used to produce a colour bar pattern (0, Figure). This allowed for testing using only the ESP32 and DE10 FPGA without the need for any camera hardware. Icarus Verilog was used to simulate Verilog designs and testbenches before use in the FPGA.

For the ESP32 to obtain an image from the FPGA, an appropriate communication protocol would be needed. The FPGA has IP blocks for UART, I2C and 3-wire SPI. However, these blocks require interfacing with the Nios II processor, which is not ideal. An Avalon-ST SPI core is also provided, however further processing of the transmitted data would be needed to separate Video and Control Avalon-ST packets, as well as remove Idle packets inserted during the SPI transfer process [8]. This block also does not support backpressure which means the Avalon-ST and SPI clocks would need to run at similar frequencies.

It was decided that the Inter-IC Sound protocol (I2S) would be the most suitable for the task. I2S was originally designed to transmit digital stereo sound. This protocol is relatively simple, consisting of a bit clock (BCLK), word select clock (WSCLK) and data line (0, Figure). This makes it simple to create hardware logic to supply the signals involved, without using the Nios II CPU. The ESP32 I2S driver uses a Direct Memory Access (DMA) controller to constantly listen for signals and read data into memory without involving the ESP32 CPU.

Although the ESP32 has a camera driver that makes use of I2S to receive camera signals [9], the driver is fixed at reading 16-bit words and requires the use of 8 GPIO pins for data. It also supports only a limited set of camera models and does not include documentation such as timing information for getting it to work with other devices. Hence using the raw I2S driver would be more suitable to read directly from the FPGA.

However, I2S data words need to be transmitted bit by bit instead of pixel by pixel. Hence extra logic is needed to read pixels and transmit them. Since the frames would likely be read by the ESP32 at a much lower rate than the stream of video packets in the vision processing pipeline, the Clocked Video Output (CVO) IP blocks are suitable since they provide pixel by pixel video output at a user-controlled clock speed.

The I2S driver allows for choosing between 8, 16, 24, 32-bit words and single or dual channel (1 or 2 words per WSCLK cycle) with a selected sampling frequency (f_s) [9]. A master clock (MCLK) frequency can be provided and is needed for the ESP32 to work in I2S 'Slave' mode [10]. f_{BCLK} would ideally be calculated as

$$f_{BCLK} = (Bits\ per\ channel) * (No\ of\ channels) * \ f_s$$

However, due to hardware constraints the ESP32 needs f_{BCLK} to be an integer divisor of f_{MCLK}, and hence readjusts f_s to allow f_{BCLK} to match the constraint. Fortunately, the CVO IP block has no lower bound for its video output frequency (f_{CVO_CLK}), and all the above values can be adjusted as appropriate. MCLK is produced by the ESP32's PLL and can only be outputted on GPIO 0, 1 or 3 (clk, Tx0 or Rx0 respectively) [11]. Only the latter 2 connect to the FPGA.

4.3.3 Using the I2S Driver

Since each pixel is 12 bits wide, a word length of 24 bits (dual channel) was selected, such that 4 pixels are sent during one WSCLK cycle. As it was not possible to use the ESP32 PLL output as input to the FPGA PLL (due to routing constraints), the required f_{BCLK} was generated with the FPGA PLL from the 50 MHz FPGA Main Clock. A value of 2.5 MHz for f_{BCLK} was chosen. This meant setting f_{BCLK} to 2.5 MHz by setting $f_{MCLK} = 20\ MHz, f_s = 48\ kHz$ (note that f_s gets readjusted to 52083 Hz). This value was chosen as it is comparable to f_{BCLK} if 24-bit stereo sound sampled at 48 kHz was being transmitted.

A Verilog module was used to take in pixels and output their individual bits, as well as generate other signals CVO_CLK and WSCLK. The ESP32 would signal the FPGA by sending a high Clear To Send (CTS) signal and the module would start sending on the next frame. Blocking the BLCK signal to the ESP32 causes it to stop reading in data, since the I2S driver essentially operates as a shift register to store bits on the rising edge of BCLK. Hence data is only sent to the ESP32 during the frame following the CTS signal, and stops during periods of horizontal and vertical blanking.

This video data is then read into a buffer by the DMA controller, and can then be read. Given the large frame size (460 kB), a portion of a frame was read and sent by TCP to a computer to be plotted with Python. To help with debugging, an image consisting of only white pixels (RGB = 255, 255, 255) was sent. The following image (Figure) was obtained.

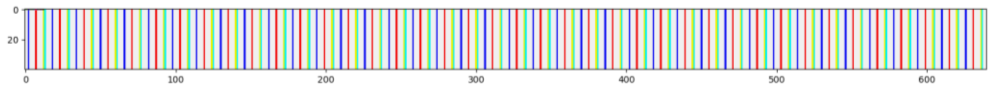


Figure 24. Image obtained when sending only white pixels.

Since the I2S data width, 24 bits does not occupy an entire 32-bit word, the I2S driver in single channel mode inserts a padding byte when storing to memory [12]. In dual channel mode the driver strangely inserts a padding byte in every 8 bytes. This causes subsequent 4-bit RGB values to be shifted, producing the colour pattern shown.

4.3.4 Obtaining Clear Images

To fix the problem, 16-bit I2S words were selected instead, and the design was modified to 'pack' four 12-bit pixels into three 16-bit words. This correctly produced a white image when white pixels were sent. When tested with the colour bar test pattern (0, Figure), the following image was produced (Figure).

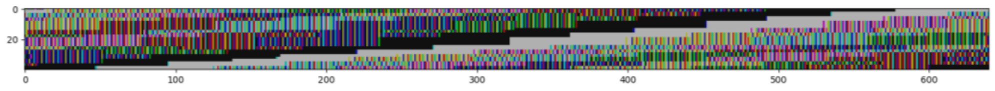


Figure 25. Image obtained when sending colour test pattern.

The black and white bars, which have uniform R, G and B values can be seen while the others do not appear clearly. This was because the ESP32, having a little-endian CPU, reorders every byte in a 32-bit word [13]. Since each byte contains 2/3 of a pixel, this reordering further shuffles pixel RGB values around and distorts colour. The best solution would be to reorder groups of 4 bytes on the FPGA before entering the ESP32, to minimize CPU processing on the ESP32 and Server. However, this would make the design more complex given that 8/3 pixels = 4 bytes. Hence it was decided to use 8-bit monochrome images instead (Figure). This would simplify the Verilog design, and prevent colour being affected should one byte be incorrectly transmitted, since now 1 pixel = 1 byte.

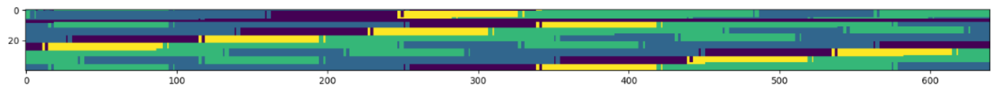


Figure 26. Visualized 8-bit monochrome image obtained. The effects of little-endian byte ordering can be clearly observed.

The byte ordering was fixed by reading from a CVO II IP block, which supports transmitting 4 pixels in parallel, and reversing the byte order within the FPGA. Note that the Colour Plane Sequencer II IP block required for converting between Avalon-ST streams with 1 and 4 pixels in parallel is only available on Quartus 16.1 and later.

Up till this point the image had a 'diagonalization' effect where the start and end of a line did not match. This was because a small fraction of the data was lost during transmission, which happened because the FPGA-generated PLL clock was not in sync with the

ESP32 MCLK. This was fixed by simply feeding the MCLK signal into the FPGA through the Rx0 pin and using a Verilog counter to produce the BCLK signal, instead of the FPGA PLL. Printing debug statements during development still works with the Tx0 pin left untouched. The final Verilog code used can be found in the appendix (0, Figure).

4.3.5 Transmitting bitmap files

At full VGA resolution, the ESP32 does not have enough RAM left to fit a full image frame. After reducing the image size to QVGA resolution (320 by 240 pixels), each frame is just 77 kB, which the ESP32's RAM can easily accommodate. This was done using the Scaler II IP Core with a simple Bilinear algorithm which consumes less logic elements. The side effect of this is that sharp lines may become blurry. However, the video stream serves to give the user a sense of the Rover's bearings for decision making, and increasing the image quality beyond a certain point brings little benefit.

When taking into account horizontal and vertical front and back porches, each frame outputted by the CVO II block is 556 by 280 pixels. Given the CVO_CLK frequency, each frame takes 0.498s to be read. Hence the video is clocked out at about 2 FPS.

Ideally colour images would be desired while maintaining 8-bit pixels, to allow the user to spot different colored obstacles. The FPGA can convert RGB values to YCbCr values with subsampling [14], where for example half of each pixel contains a luminance (Y) value, and the other half alternates between Cb and Cr chrominance values, known as 4:2:2 subsampling. Unfortunately, such 8-bit colour picture formats are uncommon, the closest being the CLRJ format [15], with no easy method of displaying images without further processing.

Fortunately, the BMP file format supports a user defined colour palette for 8-bit pixels [16]. This allowed the 3-3-2-bit RGB colour palette (0, Figure), used on some early computers, to be generated and then embedded in the bitmap file header. This increases the transmission size by 1kb but minimizes further server processing as the transmitted images can readily be displayed. The image of the test pattern obtained is shown (0, Figure).

4.3.6 Video Pipeline Integration

To integrate the Video Streaming functionality into the current Video Pipeline, a second Avalon-ST source was added to the Image Processing IP block. The Video is sent to both outputs, but will only accept backpressure from one, selected through a physical switch on the FPGA. The Video Pipeline following the image processing is shown below (Figure).

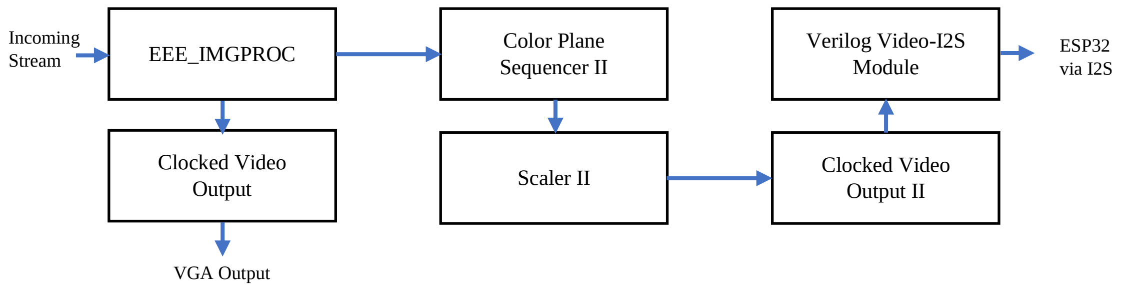


Figure 27. Video Pipeline split into two after image processing.

However, the code for video streaming did not fit with the full vision bounding box project. The largest memory usage was due to the on-chip memory, as well as the clocked video output Figure . Thus, the on-chip memory was minimised (to still allow UART) and the VGA output was removed for the video streaming section since only the clocked video output to the ESP32 was required to do video streaming. Gaussian blur also required a large amount of memory to store two rows of data for each of the five colours (see Figure). Thus, the vision subsystem was split into two projects: one with the full bounding box detection (with VGA output) and one with video streaming functionality to the website (with no blur in the detection and no VGA output).

4.3.7 Receiving Bitmap Files on the Server

As the bitmap takes up a majority of the TCP transmission bandwidth, sharing the same TCP port for communication of data and video limits the transmission speed. A separate TCP server port is created, dedicated for video streaming. However, during testing, it was found out that hosting the TCP server port in the Node.js main server alongside with other ports slows down the main server. Hence a separate Python TCP server is created.

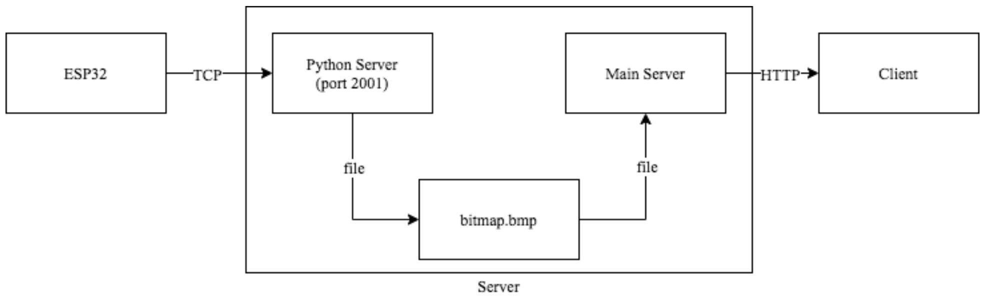


Figure 28 Video Streaming Server Structure

The server reads 77880 bytes, the size of one bitmap image, and saves as an image in the server directory. The "main" server would then reads the image and sends to client upon request. This make the video streaming server totally independent of the "main" server, avoid slowing down or fight resources between two servers.

```python
def savebitmap(data):
    f = open('bitmap.bmp', 'wb')
    try:
        f.write(data)
    finally:
        f.close()
data = connection.recv(77880, socket.MSG_WAITALL)
if len(data) == 0:
    break
print('received data of size %d', len(data))
savebitmap(data)
```

4.3.8 Performing Video Streaming on Web Page

The webpage was originally designed to use available video streaming protocols (HLS) to display real-time video from the FPGA camera. However, bottleneck sits in the TCP communication between ESP32 and server, it can only support at most 1 FPS. Utilising a video streaming protocol could not improve the video quality and even slows down the server.

An image slideshow approach is used instead to display the video. Client request new image data every 1 second and displays on the webpage. Data are sent as a Base64-encoded string as HTML supports Base64-encoded image.

As bitmap image received from ESP32 is 8-bits per pixel, while standard bitmap image is 32-bits per pixel. Most browsers do not support 8-bit bitmap, hence images are converted to 32-bits in the server before sending to client.

```javascript
var file;
try {
    file = fs.readFileSync('bitmap.bmp')
} catch (err) {
    return;
}
var bmpData = bmp.decode(file); // 8-bit image
var rawData = bmp.encode(bmpData);
base = Buffer.from(rawData.data).toString('base64')
response.json({
    bmp: base
})
```

As mentioned in the previous section, bitmap image received by the server would be stored in the directory. Hence upon client request, the server would first read the image, decode it from the 8-bit image and encode it as 32-bit. Finally, data is encoded as base64 string and sent to the client.

4.4 Control Subsystem

4.4.1 Overview

4.4.1.1 Aims and Responsibilities

The aim of the control system is to communicate with the each of the other subsystems, Drive, Power, Vision and Command. This involves using physical communication protocols over wires for the first 3 subsystems and Internet Protocols for communicating with the Command subsystem. Communication with the Power system was not implemented but is discussed in this report.

4.4.1.2 Environment

The control system is handled by an ESP32. This was programmed using the ESP-IDF framework and uploaded with PlatformIO. The ESP32 makes use of the FreeRTOS kernel, integrated into the IDF, to schedule and manage tasks. This is akin to a computer running many processes simultaneously that can communicate with each other using Queues, Semaphores and Mutexes [17].

4.4.2 Communication

4.4.2.1 Overview

The diagram (Figure) shows the data flow to/from and within the ESP32, and the different components within the ESP32 that interact with the data. The 3 UART drivers are used to communicate with Drive, Vision and Energy. The I2S driver is used to obtain video frame data, with more details given in section 4.3.

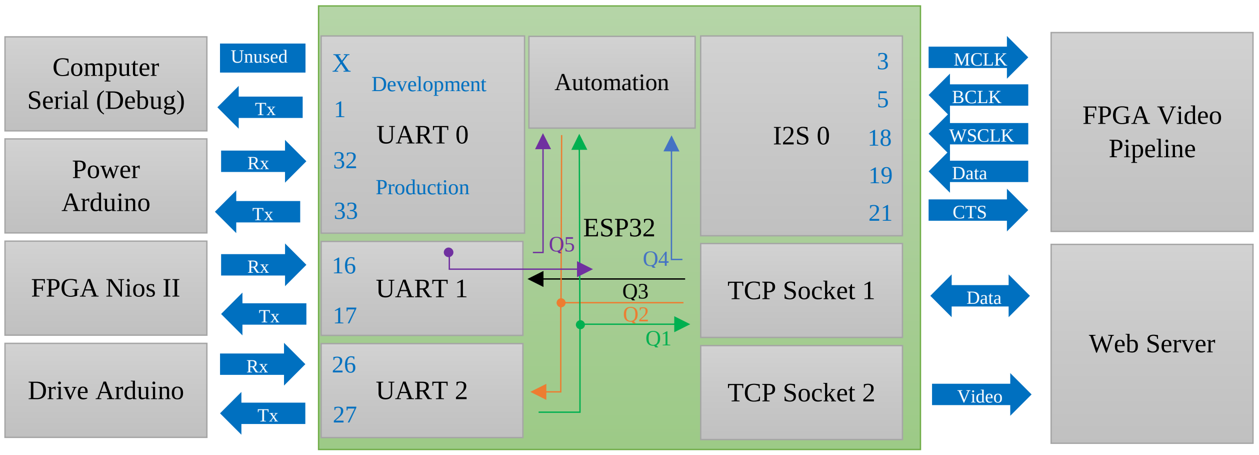


Figure 29. ESP32 Communication. Pin numbers where the ESP32 connects to other peripherals are shown. Internal communication within the ESP32 is performed with Queues, denoted Q1 to Q5.

4.4.2.2 Queue Overview

Queues are created using FreeRTOS functions to share data between tasks. They can either be FIFO queue with items placed and removed; or a mailbox with space for a single item that may not be removed when read. Video frame data is written to a buffer by the I2S driver which the TCP task accesses. A mutex controls which task is allowed to access this buffer.

Queue	Purpose	Type
Q1	Send Rover coordinates from Drive to Server and Automation	Mailbox
Q2	Send movement commands from Automation/Server to Drive	Mailbox
Q3	Send HSV adjustments from Server to FPGA	FIFO queue with 20 slots
Q4	Send target coordinates to Automation	Mailbox
Q5	Send obstacle data from FPGA to Automation and Server	Mailbox

Table 2 Queues used to transfer data within ESP32

4.4.2.3 Programming Structure

Using the FreeRTOS Kernel, different tasks can be created and assigned to run by the Kernel. Blocking events such as delays are used to transfer control back to the Kernel which can then assign a different task to unblock and run. An example of a FreeRTOS task is shown below.

```c
void read_uart(void * params) {
setup_uart();
  while (1) {
            read_uart();
            vTaskDelay( 1000 / portTICK_PERIOD_MS );
  }
}
```

This task includes a never-ending a while loop, in which the blocking statement ‘vTaskDelay’ blocks function execution until 1000ms later, during which other tasks may run. Additionally, each task is also created with a certain priority, which the kernel uses to decide which task to prioritize running. Both the delay timing and task priority allow for control over how often a certain task runs.

4.4.2.4 *Task Priorities and Periods*

The following priorities and delay intervals were set for different functions

Task	Period (ms)	Priority
Send/Receive TCP Command Packets	1000	7
Send TCP Video Packets	1000	2
Read Video frame via I2S	1000	2
UART Communication with Drive	1500	6
UART Communication with FPGA	1500	3
Automation Tasks	1000	6

Table 3 ESP32 task priorities

Communication with the web server to obtain remote commands and send them to the drive system takes higher precedence, while video streaming which uses more network bandwidth was set to be lower.

Although a function can also be set to run on a specific core, the default setting was used where kernel tasks and WiFi events run on one core while user functions run on the other. This should prevent any slowdown in TCP transmission caused by the CPU being busy with other tasks.

4.4.3 Automation

4.4.3.1 *Overview*

An aim of the rover is to be semi-autonomous, with the rover able to perform simple obstacle detection and position navigation. This makes use of available data from both the Drive and Vision subsystem, and complements the rover’s ability to allow the user make high level decisions through the video streaming and remote control.

4.4.3.2 *Obstacle Avoidance*

To avoid obstacles, the ESP32 would need to obtain the position of any obstacles in the camera’s view. The ESP32 gets coordinates of bounding boxes for each of the ping pong ball colours via UART. Since the size of the ball on the screen is inversely proportional to its distance, and each ball is of the same width, it was determined that the distance of the ping ball could be calculated as $100mm * \frac{256px}{width}$, since it was determined that a 256px length ball on the screen was 100mm away from the rover. The angle of the ball is also calculated based on its x-position. It was also determined that the rover needs to have a ball move just out of the camera view for the rover to avoid hitting it. Hence the rover will determine whether to turn left or right after considering all obstacles that are within 600mm. These commands are directly sent to the same queue that the Drive UART communication reads from.

4.4.3.3 *Position Movement*

It follows that the ESP32 can move to a certain coordinate given its current coordinate and the direction it is moving. This allows it to move to a target position while avoiding obstacles. In order to determine the direction travelled, the rover first checks to see if the last move made was either forwards or backwards, and if not makes it move as such to determine direction. As previously discussed the rover also avoids any obstacles encountered.

4.5 Command Subsystem

4.5.1 Overview

The command subsystem aims as a remote controller and dashboard monitor of the rover. It sends manual command to the rover and receives real-time data from other subsystems on board to display on the webpage. It communicates with the control subsystem via the Internet, TCP/IP is selected as the communication protocol.

4.5.2 Front-end Web Page

4.5.2.1 Web Page Components

React is chosen as the front-end framework due to its component nature and DOM property. Its largely supported libraries and wide community is also a factor for choosing this framework.

Several react components libraries are called in the webpage for easier web development, including Material-UI, React Icons, Chart.js, ApexCharts.js, etc. Line charts, sliders, buttons, dropdown menu, progress bars in the web page are made possible with the libraries.

The 2D map is made from a scatter chart, containing data points of the obstacle positions, rover past and current position. As legend and x,y axis are disabled, the origin (0,0) is shown as a black spot on the map.

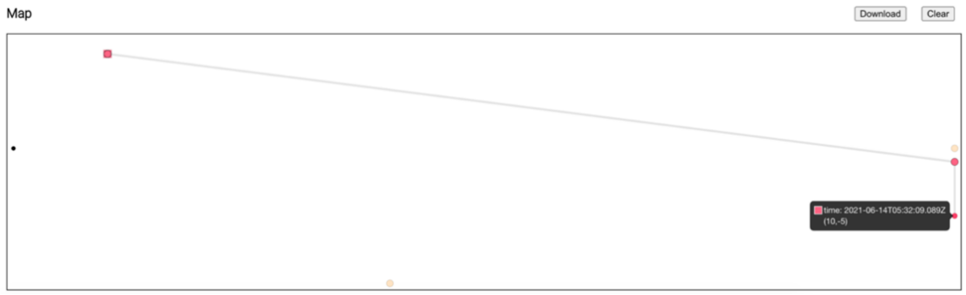


Figure 30. 2D Map on Web Page

Bisque circles represent positions of obstacles, red circles represent positions of the rover, and the red square is the current position of rover. Rover position data points are connected with a grey line, sorted by time, to represent the trail of rover. Each data point on the map also has its tooltip showing the x,y coordinates, rover positions' points also display the time data point was recorded.

4.5.2.2 Routing

React Router is used for routing between subpages, instead of fetching independent HTML pages for different URL. HTML would only fetch once within in the router [18]. It allows a more consistent UI within subpages and generally a faster response when jumping between pages. There is a home page, and 5 subpages (Battery, Drive, Controller, Video, About Us) in the dashboard which are all within the react router.

4.5.2.3 Controller

3 control modes are available for the rover, Direction, Position and Exploration. Direction gives the rover exact instruction on moving direction, Position gives the rover a target coordinate and the rover would move to it, and finally Exploration allows the rover to move randomly around. Please refer to the Control Subsystem section for detail on board implementation for each mode.

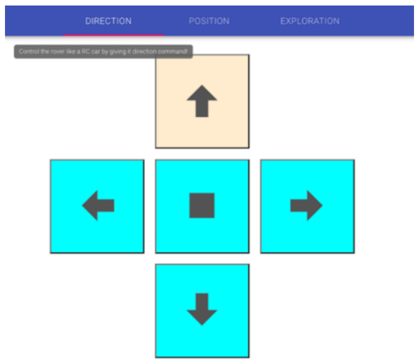
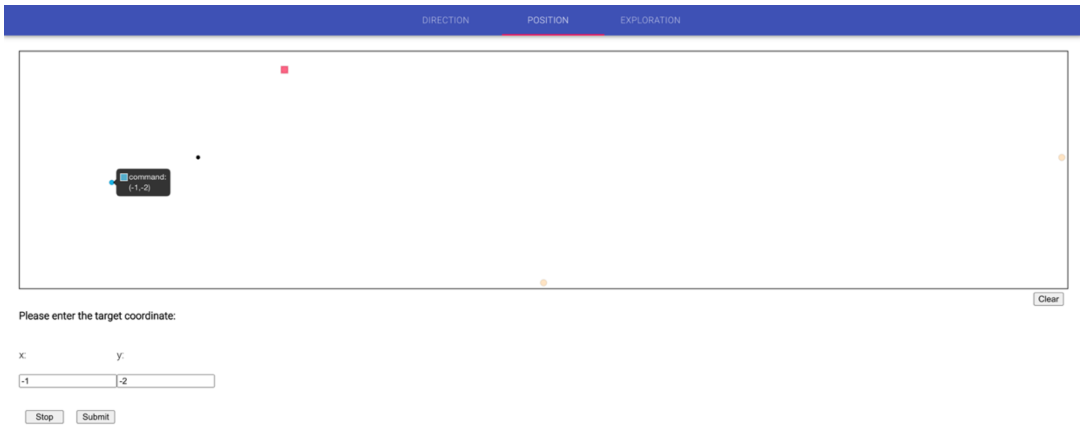

DIRECTION
POSITION
EXPLORATION
Control the rover like a RC car by giving it direction command!

Figure 31 Direction Control Controller

3 modes are available as tabs on the Control page, with simple tooltip describing what does each mode does.

In the Direction mode, 5 control buttons (left, right forward, backward, stop) indicated by icons are used to remotely control the rover. Clicking on the buttons would enable it, click again to disable it. An array of Boolean values is stored in client to keep track of which button is enabled, and shows as a BlanchedAlmond colour on the screen. If none of the buttons is enabled, it is default to have the rover stopped. In the above screenshot, the rover would be moving forward.



Please enter the target coordinate:

x: -1
y: -2

Figure 32 Position Control Controller

In the position mode, user inputs the target x and y coordinate on the form. A blue dot would be shown on the map as a preview, only upon clicking the submit button the coordinate would be sent to the server. The stop button, as suggested by the name, would stop the rover. Clicking either button triggers an alert popped out to indicate data successfully sent out.

In the Exploration mode, there is only one button that toggle between starting and stopping the rover.

4.5.2.4 Data

All data displayed on the webpage is fetched from the server using HTTP GET method. Each page fetches its data independently from different server route, such that only data needed will be fetch and allow varying fetch interval. Details on server route will be explained in the server section. Fetch interval of each page are stated in Table 4 Data Fetched in Each Web Subpage

Client data are stored as React state object, a special object type used by the React framework. When the value in the state object changes, the component will re-render, updating the webpage without reloading. State objects are updated with data received from the server [19]. Below table shows data fetched for each page.

	fetch interval (s)	Data
Home Page	10	Battery state of charge and charging state Rover speed
Battery Page	5	Battery current and historic state of charge, state of health. Charging/discharging voltage of 2 batteries Power-voltage graph data.
Drive Page	5	Rover speed, steering direction Motor notification Rover current position, obstacles' positions
Control Page	10	Rover current position, obstacles' positions (only fetch when user chooses position control mode)
Video Page	1	Image to be display as a frame in video streaming

Table 4 Data Fetched in Each Web Subpage

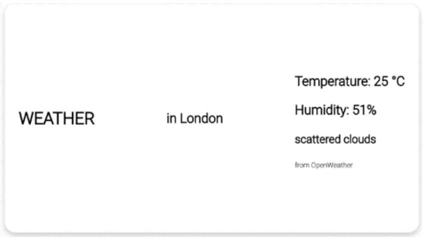
2D map on the dashboard shows past positions of the rover, allowing user to monitor the trail of the rover. All historic positions are saved in the client. Only positions after the web page has been started will be stored and displayed on the map. The 'clear' button at top right corner of the map clears all the past positions of the rover. All past positions since the server has started can be obtained as a CSV file by clicking the 'download' button. Clicking the button would activate client to fetch a CSV file from the server and download it in the browser.

On the Controller page, user commands are sent to the server on change by HTTP POST method. When user clicks a button in direction and exploration mode, change is triggered, and an update is sent to the server. On the other hand, in the position mode, data will only be posted when the stop or submit button is pressed. Posting command to server is independent of fetching, hence to ensure timeliness of commands.

Setting in the Video page, like in the Controller page, data is posted once change has occurred. Moreover, to ensure consistency of HSV values over clients, value of HSV is fetched from the server. As each colour has different value, data is fetched when user selects a different colour in the dropdown menu. Pressing the reset button would also triggers fetching of default HSV values. Details please refer to Dynamic Obstacle Detection Configuration section.

4.5.2.5 Weather

The current weather of London is shown on the home page.


WEATHER

in London

Temperature: 25 °C

Humidity: 51%

scattered clouds

from OpenWeather

Figure 33 Weather Card on Home Page

Data is obtained from Open Weather Current Weather Data API where current weather data could be obtained by fetching from a certain URL [20] .

4.5.3 Back-end Server

Node.js is chosen as the backend environment. It offers plenty of libraries and APIs on communication, and is designed for real-time web applications. The language itself being event driven makes it more ideal to handle multiple communication channels concurrently in comparison with sequential languages such as Python.

The server acts as a two-way communication channel for data from the rover be able to display on the front-end webpage and remote command from user reaching the rover.

4.5.3.1 HTTP port

2 HTTP ports are established in the server.

Port 3000 returns HTML file as client request. Data are static and should only be fetched once from the client. The HTML files are generated from above-mentioned React components.

Another port is established at 5000, client fetch data from this port. The call-and-response nature of client-server model allows the server only response with data when client requests. Several paths are available for client to only get or post data it needs. Below table shows the method, data and corresponding client page of each path.

Path	Method	Data	Client Page
/data	GET	Battery state of charge Rover speed	Home Page
/battery	GET	Battery current and historic state of charge, state of health. Charging/discharging voltage of 2 batteries Power-voltage graph data.	Battery Page
/drive	GET	Rover speed, steering direction Motor notification Rover position, obstacles positions	Drive Page Control Page (map in position control)
/drivedata	GET	CSV file on past rover positions	Drive Page (download button)
/position	POST	Any user control command e.g. moving forward	Control Page
/video	GET	Image encoded in Base64 string	Video Page
/videosetting	POST	Manually set obstacle detection configuration	
	GET	Value of each colour's HSV range	

Table 5 Response Data for Each HTTP Path

Please refer to Video Streaming section for details on implementation of video and Dynamic Obstacle Detection Configuration section for manual setting.

/videosetting GET method returns different data according to parameters, as data is fetched within each slider.

```
app.get("/videosetting/:color/:type", (request,response) => {
    var rtn = null;
    // videocolor is a local variable stored in server
    for (let i=0; i<videocolor.length; i++){
        if (videocolor[i].color === request.params.color){
            rtn = videocolor[i]
        }
    }
    if (rtn === null) return;
    rtn = rtn[request.params.type];
    if (rtn === undefined) return;

    response.json({
        value: rtn
    })
})
```

color refers to the colour user is adjusting and type refers to either hue, saturation or value.

4.5.3.2 TCP port

A TCP port at 2000 is established for communication between ESP32 and the server. Received bytes are first parsed from JSON format to an Javascript object, then update the data in server accordingly. User commands stored in server are also parsed into JSON format and send to the ESP32. Please refer to Section 4.8 for details on data being sent and received. The TCP port also follows a call-and-response structure, packet will only be sent to the ESP32 when it receives a packet. The ESP32 initiates the communication to avoid ESP32 being overloaded by data.

Another TCP port is established at 2001, dedicated for video streaming communication. Please refer to Video Streaming section for details.

4.5.3.3 Data

Data are stored as local variable in the server, updates when it receives packets from ESP32 and sends to the clients when there is a fetch request. As the server is expected to run 24/7, offline database is not set up. Data are stored locally and will be lost when the server is stopped. An exception would be information about the energy subsystem. Data is not obtained during run-time in this project due to hardware limitation, they are saved as csv file in the server beforehand. At the start of the server, it reads and parse the csv file and store as local variable.

4.5.3.4 Obstacle Position Calculation

Positions of obstacles can be calculated given the direction and angle of obstacle from the rover and the rover's current direction. The rover's facing angle also needs to be known to calculate the x,y coordinates of the obstacles to be displayed on a 2D map.

The facing angle of the rover can be calculated by obtaining the slope of two consecutives position given that the rover is moving forward. Hence calculation of rover is only done when the rover is moving forward.

Moreover, due to measurement error, the calculated position of a same obstacle might vary a bit. Hence certain error tolerance is allowed, which is a 100mm square. If a previous-calculated obstacle of same colour is within error range of the new obstacle, the previous obstacle positions would be updated with the average of these two values, else a new obstacle would be created.

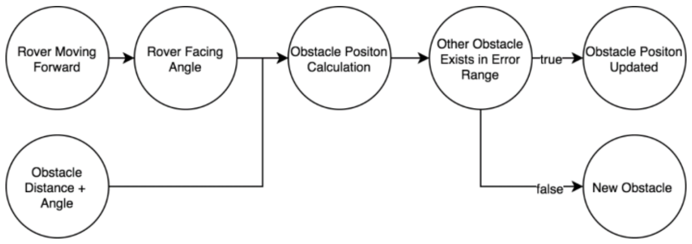

Rover Moving Forward
Rover Facing Angle
Obstacle Positon Calculation
Other Obstacle Exists in Error Range
true
Obstacle Positon Updated
Obstacle Distance + Angle
false
New Obstacle

Figure 34 Obstacle Position Flow Chart

4.6 Drive Subsystem

The purpose of the drive module is to control and measure the distances of the mars rover, and control the direction it facing to. The control module will implement the instruction received at Tx Rx ports, and send the position back. The block diagram shows below is the overall structure of the control module.

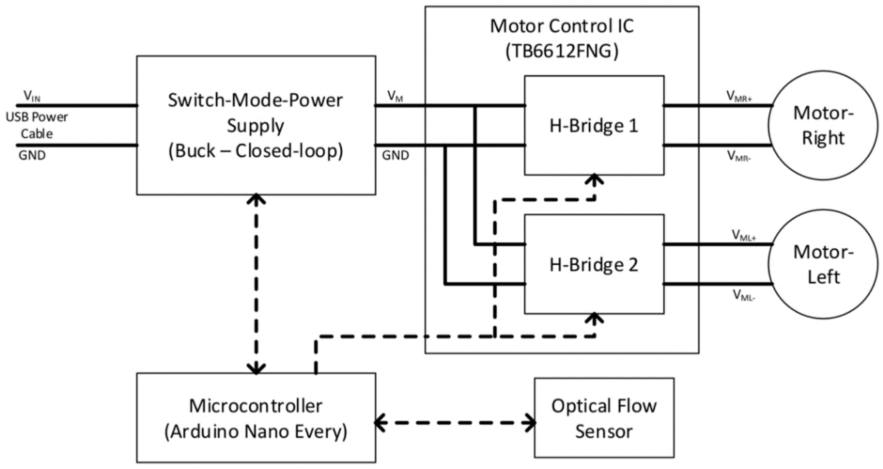


Figure 35 overall structure of drive system

4.6.1 Control Principle

The Arduino Nano Every controls the duty cycle in the SMPS, thus control the voltage input to the motor. By changing the value and sign of the voltage to H-bridge, we could achieve different mode for the rover, for example, turning left/right, moving forward/backward, with different speed.

4.6.2 Optical sensing

In this part, the Arduino will communicate with optical flow sensor through SPI port. The main function in this part is "mousecam_read_motion" which will read delta x and y in optical sensor into dx and dy in Arduino. This function will be called in each loop. By accumulate dx and dy, the total_x and total_y can be found. Later on, this information will be send to other microcontrollers through serial communication port(Tx, Rx, GND). Codes can be found in appendices 10.4.1.

4.6.3 Difficulties *in optical sensing:*

After integrating the optical and drive codes, we find that the current and voltage PID controller perform badly, which turn to have a noticeable delay and large oscillation. After investigation, we find that this is caused by the time delay in optical flow sensor, and time delay in Arduino main loop. Since the optical sensor have a fixed delay in each read and write, the only way to cancel the delay is to cancel the delay in the main loop. After doing this, the controlled duty cycle perform better and faster, and there are no noticeable oscillation appears.

Also, the range for total_x and total_y originally is about -200 to 200. This is because that the data type of total_x1 and total_y1 are int. The size of int in Arduino nano only have 16 bits, so the range for total_x1 and total_y1 are -32768 to 32767 [21]. Thus, total_x and total_y is restricted between -20(-32768/157) and 20(32767/157). This problem can be solved by changing the data type from int to long, which have 32 bit size. [22]

4.6.4 Drive control

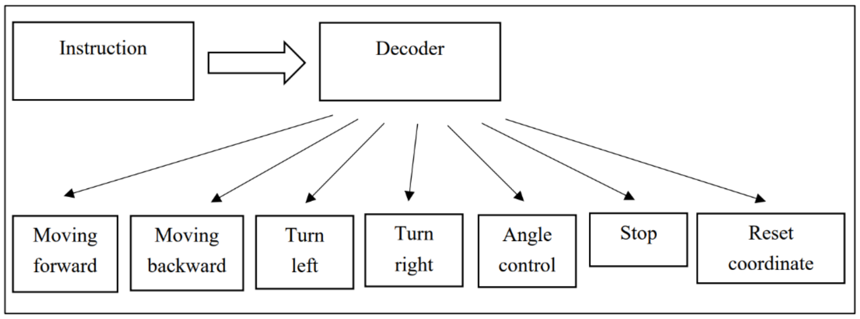


Figure 36 motor code structure

The overall structure of the distance control can be seen in the graph above. There are 7 instructions can be read from serial communication port. For example, when the decoder received a stop instruction, the voltage in the motor will be set to zero, so the mars rover will stop. When the decoder received a turn_left instruction, DIRRstate and DIRLstate will be set to LOW, so the mars rover turning in anticlockwise.

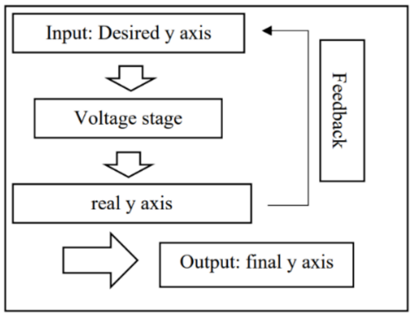


Figure 37 logic of moving forward method

4.6.4.1 Moving forward:

One most important instruction is "Moving forward". Since the mars rover can only moving in one direction, we just need to consider the forward scenario, then manipulate the angle to cover other directions. The main logic for this instruction is shown in the right figure above. When decoder received instruction such as "Moving forward 30 cm", the desired distance will be recorded. By comparing the current location with the desired one, the ey (difference between desired y and current y) will be recorded. When ey is large, the voltage of the motor should be high, so it will approaching the desired location faster. When ey is small, the voltage of the motor should be low, so the mars rover can have a better control of the speed, and not missing the target y.

Following this logic, serial several voltage stage can be set below.

Voltage mode(stage)	Range of ey	Voltage (Max 1024)	Action
1	0 to 1	0	Moving forward
2	1 to 10	400	Moving forward
3	10 to infinite	500	Moving forward
4	-2 to 0	400	Moving backward
5	-infinite to 2	500	Moving backward

Figure 38 Voltage Stages for Moving Forward

Since there are delay in data transmission from optical sensor to Arduino nano, if set the voltage to 0 only when ey=0, the mars rover will keep oscillating and never stop. So we need to set an acceptable error for ey, in order to avoid oscillation. In this case, Voltage will be set to 0 when ey smaller than 1cm. If it miss the desired y due to inertia, the motor will moving backwards slowly instead, until ey smaller than 1.

The codes in Appendices 0 shows the application of those 5 modes in this process.

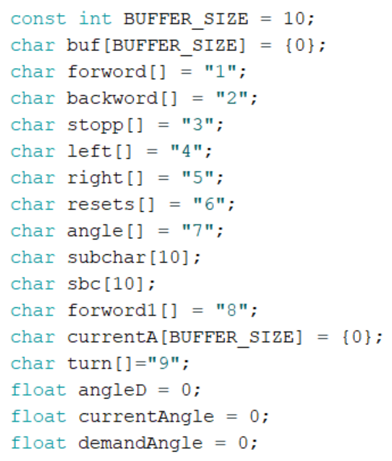
4.6.5 Angle control

Controlling the direction of the mars rover is very similar to controlling the distances of the mars rover, except that the "current angle" feedback come from the camera, not optical sensor. The camera will send the current angle and desired angle together with the turning instruction, the difference between those two angles is noted as angleD. Thus, according to the vslur of angleD, the voltage stage can be set as follow:

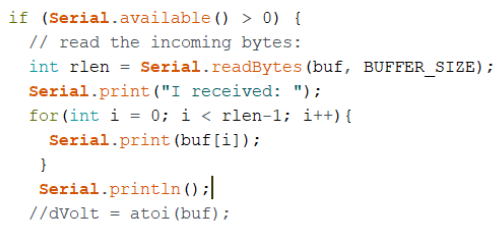
Voltage mode(stage)	Range of angleD	Voltage (Max 1024)	Action
1	0 to 5	0	Turning anticlockwise
2	5 to 20	400	Turning anticlockwise
3	20 to 90	500	Turning anticlockwise
4	90 to infinite	600	Turning clockwise
5	-0 to -5	0	Turning clockwise
6	-5 to -20	400	Turning clockwise
7	-20 to -90	500	Turning clockwise
8	-90 to -infinite	600	Turning clockwise

Figure 39 Voltage stages for Angle Control

The codes in appendices 10.4.3 shows the application of those 5 modes in this process. The graph bellow is the codes for angle control. There are also other instructions such at turn left and turn right, reset coordinate. Which just simply set DIRRstate and DIRLstate to LOW or HIGH without feedback control, and reset the total_x and total_y to zero.



```
const int BUFFER_SIZE = 10;
char buf[BUFFER_SIZE] = {0};
char forword[] = "1";
char backword[] = "2";
char stopp[] = "3";
char left[] = "4";
char right[] = "5";
char resets[] = "6";
char angle[] = "7";
char subchar[10];
char sbc[10];
char forword1[] = "8";
char currentA[BUFFER_SIZE] = {0};
char turn[]="9";
float angleD = 0;
float currentAngle = 0;
float demandAngle = 0;
```



```
if (Serial.available() > 0) {
  // read the incoming bytes:
  int rlen = Serial.readBytes(buf, BUFFER_SIZE);
  Serial.print("I received: ");
  for(int i = 0; i < rlen-1; i++){
    Serial.print(buf[i]);
   }
  Serial.println();
  //dVolt = atoi(buf);
```

Figure 40 initial decoder key map

One way to decode the instruction is passing different opcodes or specific number in char type to each instruction. A very simple decoder key map is list below.

4.6.6 Communication

When serial monitor receives an instruction with the first char called “1”, the Arduino will notice that it is an “moving forward” instruction. “findsubchar” function will return the rest number after “1” which is the value of desired distance, thus, ey can be found. The code on the left is how this decoder applied.

4.7 Power Subsystem

4.7.1 Aim

The aim of Power subsystem is charging the batteries by PV panels, and in case battery data, e.g., battery voltage, the State of Charge and State of Health are constantly being monitored. The system then should be connected with Drive for supplying power to the whole rover, and connected with control for data transmission. However, these cannot be physically implemented.

4.7.2 Overview

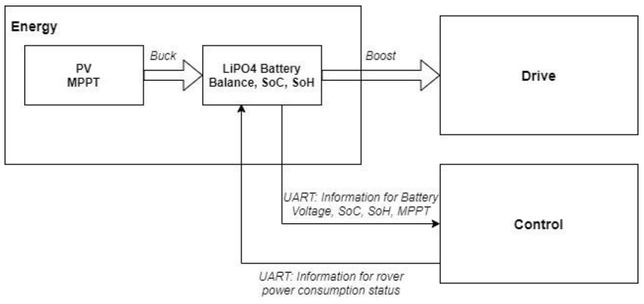

Energy
PV MPPT
Buck
LiPO4 Battery Balance, SoC, SoH
Boost
Drive
UART: Information for Battery Voltage, SoC, SoH, MPPT
Control
UART: Information for rover power consumption status

Figure 41 Power subsystem overview

By considering the fact that the space is limited on rover, the energy module cannot be physically integrated and tested by integration, as well as lack of sensors to implement more accurate measurement methods, the following integration method has been implemented. Figure The PV will perform as a charging station using MPPT algorithm to charge the parallel balanced batteries. The OCV of batteries will then be measured, together with the SoC which is achieved by constructing a lookup table from data collected in previous coulomb counting method. During charging process, the actual capacity available will also be recorded, together with number of charging cycles to determine the SoH status of battery.

After parallel balanced batteries have been charged, the SMPS at energy side will be set by mode Boost, connecting to Buck mode SMPS at Drive side to supply power for the rover.

The details of design and implementation will be discussed below.

4.7.3 Photovoltaics Panel

The Photovoltaics performs as charging station for batteries due to physical space limitation on Rover to add in another SPMS, as well as the fact that batteries cannot charge and discharge simultaneously to provide power (since current provided by 4 parallel connected PV is too small to operate the rover), making it less sensible to integrate PV with batteries onto the Rover.

4.7.3.1 Characterization

Here is the characterization IV curve for PV Figure. We implement this by using Synchronous Buck mode with duty cycle sweep from 0 to 1. The result shows the characteristic PV panel performance, with maximum power output around the curvature. It is worth noted that the part below 4V does not show the characteristic behavior. This is due to measuring voltage through potential divider at port A will also draw some current, i.e., having the voltage drop, which in case causes PV voltage cannot decrease below this point. The MPPT algorithm then applied to the PV panel to keep power output at the maximum point.

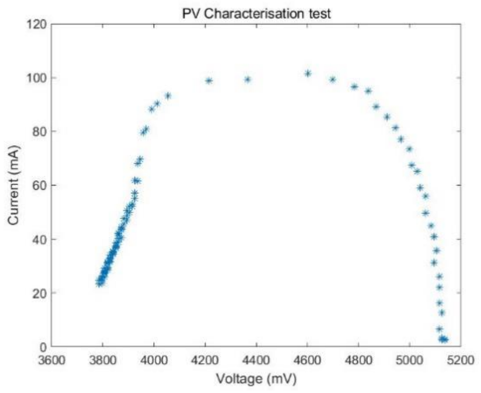


Figure 42 Plot for PV characterization

4.7.3.2 MPPT

```
if(input_switch == 1 && i < 1.00){ //Start condition
   if(power > power_previous){ // Compare power
      if(V_pd >= V_pd_previous){  // Compare voltage
        i -= 0.01;} //increase voltage if on LHS of MPP
      else{
      i += 0.01;} //decrease voltage if on RHS of MPP
    }
    else{
      if(V_pd <= V_pd_previous){
      i -= 0.01;} //increase voltage if on LHS of MPP
      else{
      i += 0.01;} //decrease voltage if on RHS of MPP
    }
}
```

Figure 43 Code for MPPT

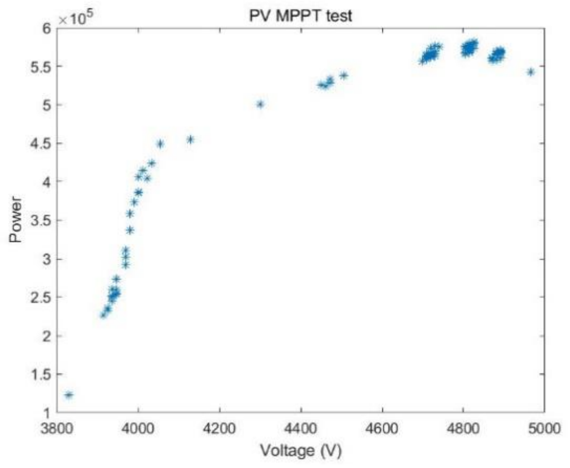


Figure 44 Plot for MPPT curve

The MPPT algorithm is implemented by comparing the current power with the previous one, together with voltage comparison to locate the current operating point of PV. If the current power is greater than the previous one, and the same applies to the voltage, then that means we are at the left-hand side of maximum power point. On the other hand, if in this case voltage is less than the previous one, this means we are at the right-hand side of maximum power point and the corresponding adjustments to the duty cycle are applied. The result shows the PV panel power varies around the maximum power point [23] [24].

However, there are limitations in our method. Since the IV curve is influenced by the irradiance level and temperature, the lack of sensors to measure those factors may result in inaccurate IV curves compared with the environment on the Mars. In addition, both PV voltage and current measuring methods are indirect, and values are corrected according to SMPS component parameters, which would result in inaccurate recorded values.

For PV charging with batteries, it is hard to implement CC or CV method in this case due to the conflict in PWM generated by PID controller and MPPT. Also, the small amount of current supplied by 4 parallel connected PV panels making it less sensible to implement CC or CV methods.

4.7.4 LiPO4 Battery

4.7.4.1 Characterization

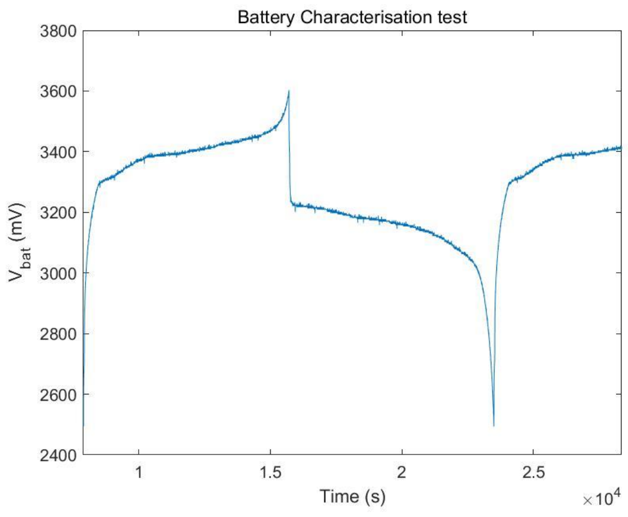


Figure 45 Plot for Battery Characterization

The entire charging and discharging process for battery is shown in figure. The rough battery capacity estimation can be achieved by calculating total discharging amount in discharging state, through multiplying constant discharging current with the time in discharge state.

Moreover, it is worth noted for the chemical reaction inside the battery. LiFePO4 as cathode (+) and Graphite as anode (-). During discharging process, the Li+ ion will move from anode to cathode through electrolyte, and electron from anode will move to cathode through the outer electric circuit. During charging process, the Li+ ion will move from cathode to anode through electrolyte, and electron from cathode to anode through the circuit [25].

4.7.4.2 Balance

The batteries are balanced in parallel. Due to the limitation of discharge circuit within battery board, in which voltage decreases very slowly and it is hard to use that for discharging lower capacity cells. Instead, we are using active balance method, in which we stop charging or discharging whenever the lowest capacity cell is fully charged or discharged.

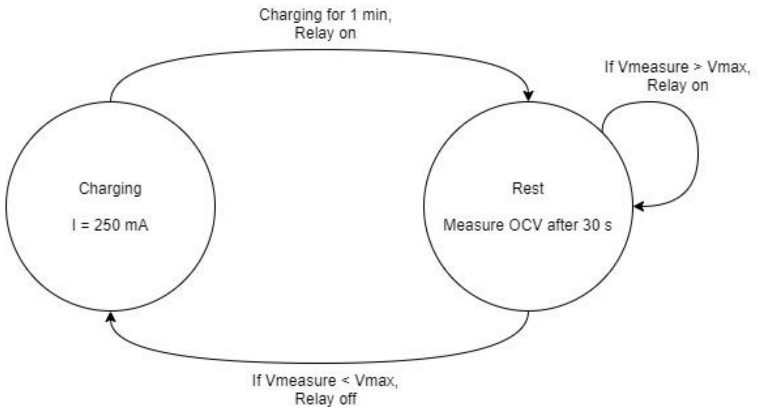


Figure 46 Figure Design for battery balance algorithm

Figure 76 Code for Parallel Balance

In this case, the charging process is not continuous as the voltage of individual battery have to be constantly monitored. The battery is firstly charged for 1 minute, followed by 30s rest before having a relatively more accurate OCV measure. In addition, due to parallel balance nature, we cannot implement Coulomb counting method to monitor its State of Charge, while only implementing the voltage look-up table method instead, which would be a relatively inaccurate method. However, if more equipment such as current sensor is provided for individual battery board, the parallel balance SoC estimation would be more accurate.

In this case, the reason for us to use parallel balance instead of serial balance is mainly due to the power supplied by PV panel. Since the 4 parallel connect PV with Halogen as light source can only supply current with maximum 70mA, it's even less effective for us to change the configuration into serial battery connection in Boost mode.

4.7.4.3 Charging Methods

Since the LiFeO4 batteries cannot be fully charged by simply using constant current charging method, and only use CC method may have negative effect on battery state of health, the pre-charging state and constant voltage charging state are also applied here under USB power situation [26] [27].

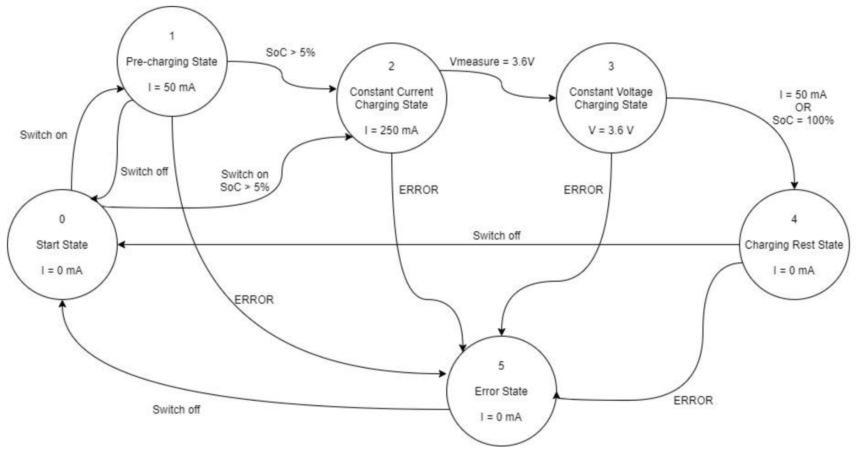


Figure 47 Design for CC and CV charging

Figure 77 Code for CC/CV Charging

In the charging FSM, we firstly pre-charge the batteries to around 5% of total capacity with small current for 50mA. Since charging with large current at low SoC of Li cells may damage its chemical structure, in this way we can effectively increase the life of battery thus improve State of Health. On the second stage, we charge cells by constant current until OCV 3.5 V, which is the maximum charging voltage for the battery. In this case, we can no longer charge the cell by large current as this brings the potential risk of damaging the cell. On the other hand, constant voltage charging is applied to fix the charging voltage at 3.5 V. As the potential difference between the inner part of battery and outer 3.5V applied becomes smaller, the current flow into the cell will become smaller. When the current is small enough e.g., 50mA, the charging process is considered to be completed.

However, it is also suggested that tracing the sudden change point of polarized voltage of Li battery and decrease the charging current accordingly would be a better approach than constant voltage charging method, for which it takes into account for both charging efficiency and SoH [28].

The voltage controller from Power Lab is used here for voltage control. The current controller is cascaded after it. The output from PID voltage controller becomes the input of current controller before passing through the saturation function. In this case, the voltage is regulated around voltage reference value, and the current is supplied according to the charging status of battery.

4.7.4.4 State of Charge

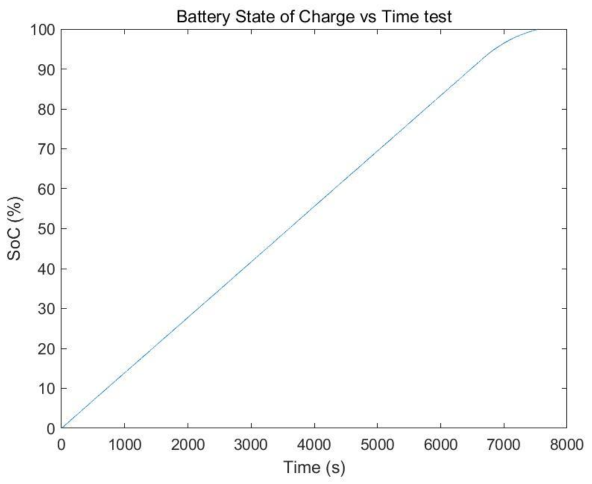
The state of charge of batteries are measured by coulomb counting and voltage look up table methods [29] [30].

Voltage (mV)	**SoC (%)**
2500	**2**
2900	**5**
3100	**10**
3200	**15**
3230	**20**
3250	**25**
3280	**30**
3300	**40**
3310	**50**
3320	**60**
3330	**70**
3340	**80**
3350	**90**
3360	**100**

Table 6 Design for look-up table

```c
float integral(float current_measure, float current_previous){
  float trapezoidal = (1.00/(2.00 * 3600.00)) * (current_measure + current_previous); //Use trapezoidal method for integration
  return trapezoidal;}
//Here below are in SLOW LOOP (1Hz)
soc = soc + (integral(current_measure, current_previous))/(500);
current_previous = current_measure;
```

Figure 48 Code for Coulomb Counting



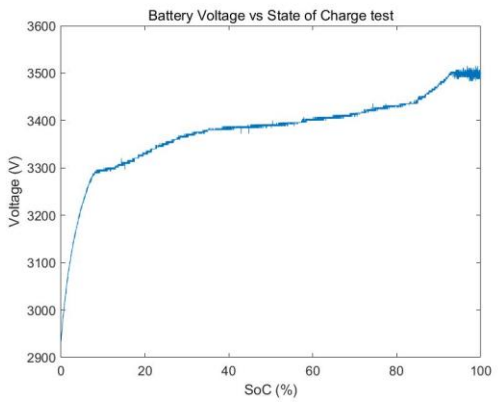


Figure 49 Plot for Coulomb counting with time using CC and CV charging method

Figure 50 Plot for Coulomb counting with voltage using CC and CV charging method

Since for state of charge measurement, we have the formula $SoC_{current} = SoC_{previous} + \frac{n\int I(t)\,dt}{Q_{total}}$, where n is coulomb efficiency and here approximated to be 1 [31] [32]. In the formula the current SoC is the sum of previous SoC and the increase in SoC, which is represented by increased amount of charge over the total charge of battery. We can integrate current here by trapezoidal method in Arduino, and the previous SoC value is read and write into SD card for each charging time.

The SoC figure demonstrates the SoC increase with time linearly in CC mode, and its gradient decreases in CV mode as current decreases gradually in this mode. We can also observe that the OCV is fixed at 3.5V for above 90% SoC, which meets our expectation.

A more complex approach for SoC estimation would be using Kalman Filter [33].

4.7.4.5 State of Health

There are lots of factors influence the state of health of battery. The most significant one would be its total capacity available divides its nominal capacity. The batteries are designed to have some more capacity than labelled when they are newly manufactured, in case is around 580 mAh. However, as battery has been charged by multiple times, due to chemical reactions inside the battery, the actual capacity available will decrease.

In addition, the number of charging and discharging cycles should also be noted. The battery is designed to have 1000 cycles. As number of cycles increases, the impedance of battery will also increase, in case the internal resistance of battery may also be the indication for state of health of battery. This can be measured by Ohm's law using $R_b(SOC,T) = \frac{OCV(SOC,T) - V_{bat}(SOC,T)}{I_{pulse}}$ [34].

Besides, the temperature of battery is also a significant factor for SoH. Under normal circumstances, the LiPO4 batteries we use should have temperature between 10 to 40 [35]. The bad weather on the Mars may heavily influence the life cycle of battery.

4.7.5 Communication with other modules

4.7.5.1 With Drive

Since 5V is required for rover operation, the parallel balanced batteries will connect with drive SMPS in Buck mode through energy SMPS in boost mode. The current and voltage required are pre-calculated and discharges accordingly. However, since it is impossible to test the implementation, the discharging process is for reference only. The rover range estimation will also be based on comparing the data of current power consumption and current distance travelled to estimate remaining range available.

4.7.5.2 With Control

The discharging process are monitored by control according to the status of rover, whether the rover is at rest, operating with light off or fully operating. The entire charging and discharging process relevant information are sent to control through UART port, and this is then displayed at command side.

4.7.6 Implementation on Charging and Discharging

The charging for battery with PV Figure 78 Plot for Charging MPPT(Starting with Figure 78) and the remote battery discharging process (Starting with Figure 83) can be found in appendix

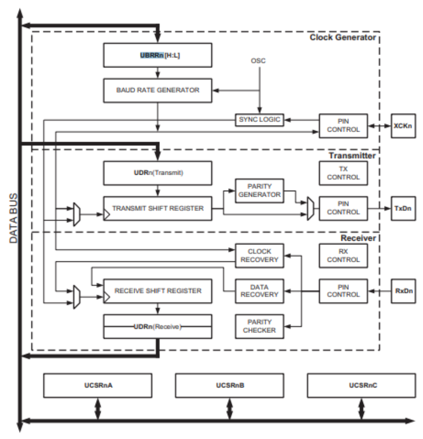
4.8 Inter-module communication (integration)

4.8.1 ESP32-Drive Arduino UART

The communication protocol is set to be UART because the Arduino Navy Nano has an embedded UART port, it is sufficient for transmitting the opcode and the Rover's coordinates. The configuration of the UART is "SERIAL_8N1", which has 1 start bit, 8 data bits, none parity bit and 1 stop bit. The baud rate is set to be 115200 bps as this is the default baud rate of ESP32, and also the limit of Serial monitor of the Arduino IDE. Theoretically it can be improved because the ESP32 UART port support up to 5Mbps,

Baud rate selection

Figure 20-1. USART Block Diagram(1)



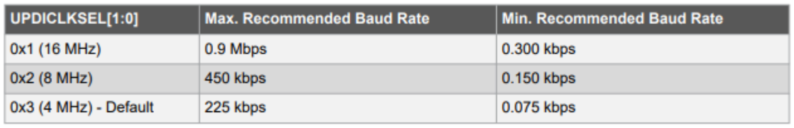
Note: 1. Refer to Figure 1-1 on page 12 and Table 14-9 on page 97 for USART0 pin placement.

Figure 51. ATmega4809 Data Sheet *[36]*

The baud rate is determined by writing to the registers UBRRn[H:L], it can be described by the following equation,

$$\mathrm{UBRRn[H:L]} = \frac{\mathrm{f_{OCS}}}{16\mathrm{BAUD}} - 1$$

Given the clock frequency to be 16MHz, the maximum baud rate possible is 1 Mhz.



UPDICLKSEL[1:0]	Max. Recommended Baud Rate	Min. Recommended Baud Rate
0x1 (16 MHz)	0.9 Mbps	0.300 kbps
0x2 (8 MHz)	450 kbps	0.150 kbps
0x3 (4 MHz) - Default	225 kbps	0.075 kbps

Figure 52. ATmega4809 Data Sheet *[36]*

Json Struct

Json Format (Syntax) is a light weighted design, the "ArduinoJson" library has been used enforced this feature. By comparing to the official "Arduino_Json" libaray, the "ArduinoJson" is twice smaller in size, consume 10% less RAM and runs 10% faster [37] . The drive Arduino only sending/receiving data passively. Hence the commination interval is purely depends on ESP32, this enable the ESP32 to perform task prioritising.

4.8.2 ESP32-Server communication

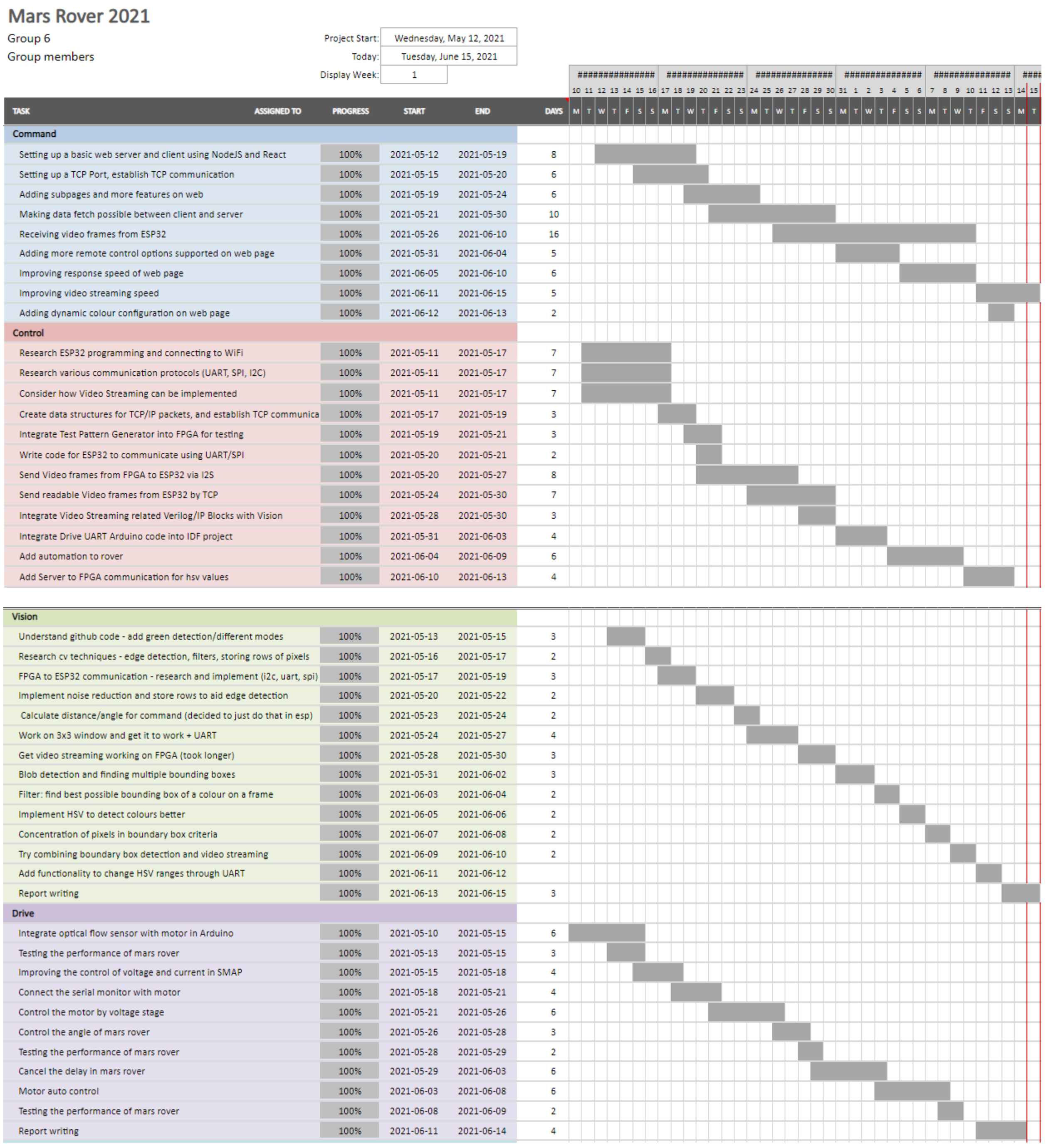
Several application IP protocols were considered for communication between the ESP32 and the web server. HTML, MQTT and WebSockets were considered with example CC0-licenced code already available from Expressif [38]. However, it was decided that raw TCP sockets would suffice as they would still be able to still transmit information reliability with less overhead, provided error checks are performed.

5 Project Management

5.1 Progress Checking

Regular team meeting were held twice a week at the start of the project to keep track of progress and ensure everyone in the team was on track. Meetings were held in Microsoft Teams and offline communication was mainly done via WhatsApp.

As this project involves quite a few people and several systems, it was important to be able to keep track of the number of tasks. A Gantt chart was utilised to ensure tasks were completed on schedule.



Mars Rover 2021

Group 6

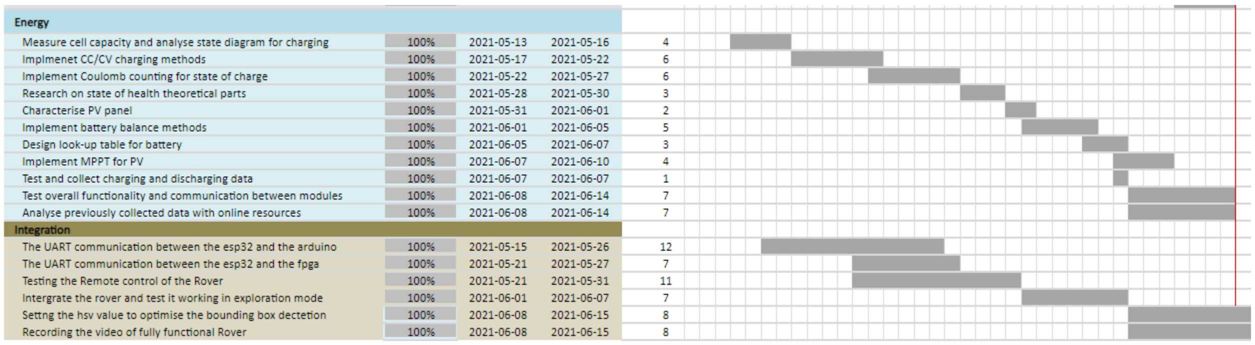
Group members

Project Start: Wednesday, May 12, 2021

Today: Tuesday, June 15, 2021

Display Week: 1

TASK	ASSIGNED TO	PROGRESS	START	END	DAYS
Command					
Setting up a basic web server and client using NodeJS and React		100%	2021-05-12	2021-05-19	8
Setting up a TCP Port, establish TCP communication		100%	2021-05-15	2021-05-20	6
Adding subpages and more features on web		100%	2021-05-19	2021-05-24	6
Making data fetch possible between client and server		100%	2021-05-21	2021-05-30	10
Receiving video frames from ESP32		100%	2021-05-26	2021-06-10	16
Adding more remote control options supported on web page		100%	2021-05-31	2021-06-04	5
Improving response speed of web page		100%	2021-06-05	2021-06-10	6
Improving video streaming speed		100%	2021-06-11	2021-06-15	5
Adding dynamic colour configuration on web page		100%	2021-06-12	2021-06-13	2
Control					
Research ESP32 programming and connecting to WiFi		100%	2021-05-11	2021-05-17	7
Research various communication protocols (UART, SPI, I2C)		100%	2021-05-11	2021-05-17	7
Consider how Video Streaming can be implemented		100%	2021-05-11	2021-05-17	7
Create data structures for TCP/IP packets, and establish TCP communica		100%	2021-05-17	2021-05-19	3
Integrate Test Pattern Generator into FPGA for testing		100%	2021-05-19	2021-05-21	3
Write code for ESP32 to communicate using UART/SPI		100%	2021-05-20	2021-05-21	2
Send Video frames from FPGA to ESP32 via I2S		100%	2021-05-20	2021-05-27	8
Send readable Video frames from ESP32 by TCP		100%	2021-05-24	2021-05-30	7
Integrate Video Streaming related Verilog/IP Blocks with Vision		100%	2021-05-28	2021-05-30	3
Integrate Drive UART Arduino code into IDF project		100%	2021-05-31	2021-06-03	4
Add automation to rover		100%	2021-06-04	2021-06-09	6
Add Server to FPGA communication for hsv values		100%	2021-06-10	2021-06-13	4
Vision					
Understand github code - add green detection/different modes		100%	2021-05-13	2021-05-15	3
Research cv techniques - edge detection, filters, storing rows of pixels		100%	2021-05-16	2021-05-17	2
FPGA to ESP32 communication - research and implement (i2c, uart, spi)		100%	2021-05-17	2021-05-19	3
Implement noise reduction and store rows to aid edge detection		100%	2021-05-20	2021-05-22	2
Calculate distance/angle for command (decided to just do that in esp)		100%	2021-05-23	2021-05-24	2
Work on 3x3 window and get it to work + UART		100%	2021-05-24	2021-05-27	4
Get video streaming working on FPGA (took longer)		100%	2021-05-28	2021-05-30	3
Blob detection and finding multiple bounding boxes		100%	2021-05-31	2021-06-02	3
Filter: find best possible bounding box of a colour on a frame		100%	2021-06-03	2021-06-04	2
Implement HSV to detect colours better		100%	2021-06-05	2021-06-06	2
Concentration of pixels in boundary box criteria		100%	2021-06-07	2021-06-08	2
Try combining boundary box detection and video streaming		100%	2021-06-09	2021-06-10	2
Add functionality to change HSV ranges through UART		100%	2021-06-11	2021-06-12	
Report writing		100%	2021-06-13	2021-06-15	3
Drive					
Integrate optical flow sensor with motor in Arduino		100%	2021-05-10	2021-05-15	6
Testing the performance of mars rover		100%	2021-05-13	2021-05-15	3
Improving the control of voltage and current in SMAP		100%	2021-05-15	2021-05-18	4
Connect the serial monitor with motor		100%	2021-05-18	2021-05-21	4
Control the motor by voltage stage		100%	2021-05-21	2021-05-26	6
Control the angle of mars rover		100%	2021-05-26	2021-05-28	3
Testing the performance of mars rover		100%	2021-05-28	2021-05-29	2
Cancel the delay in mars rover		100%	2021-05-29	2021-06-03	6
Motor auto control		100%	2021-06-03	2021-06-08	6
Testing the performance of mars rover		100%	2021-06-08	2021-06-09	2
Report writing		100%	2021-06-11	2021-06-14	4



Energy				
Measure cell capacity and analyse state diagram for charging	100%	2021-05-13	2021-05-16	4
Implmenet CC/CV charging methods	100%	2021-05-17	2021-05-22	6
Implement Coulomb counting for state of charge	100%	2021-05-22	2021-05-27	6
Research on state of health theoretical parts	100%	2021-05-28	2021-05-30	3
Characterise PV panel	100%	2021-05-31	2021-06-01	2
Implement battery balance methods	100%	2021-06-01	2021-06-05	5
Design look-up table for battery	100%	2021-06-05	2021-06-07	3
Implement MPPT for PV	100%	2021-06-07	2021-06-10	4
Test and collect charging and discharging data	100%	2021-06-07	2021-06-07	1
Test overall functionality and communication between modules	100%	2021-06-08	2021-06-14	7
Analyse previously collected data with online resources	100%	2021-06-08	2021-06-14	7
Integration				
The UART communication between the esp32 and the arduino	100%	2021-05-15	2021-05-26	12
The UART communication between the esp32 and the fpga	100%	2021-05-21	2021-05-27	7
Testing the Remote control of the Rover	100%	2021-05-21	2021-05-31	11
Intergrate the rover and test it working in exploration mode	100%	2021-06-01	2021-06-07	7
Settng the hsv value to optimise the bounding box dectetion	100%	2021-06-08	2021-06-15	8
Recording the video of fully functional Rover	100%	2021-06-08	2021-06-15	8

After each meeting, each member filled in a document describing what needed to be done before the next meeting. This was important to make sure everyone was on track. Initially, the goal was to create a working prototype of the Mars rover as soon as possible, which involved closely working with Integration throughout the project to work out how to communicate between the subsystems. The remote rover was fully working by two weeks before the deadline, which allowed us to build functionality incrementally rather than trying to integrate everything at the end.

5.2 Testing Process

Due the limitation of remote working, only one person of the team was able to have full access to the rover. Hence, local testing within each subsystem was prioritized over inter-module testing.

In the Vision subsystem, compiling the Quartus project took a long time, so development/testing was sped up in several ways. Firstly, more modes or switch configurations were implemented to test multiple outputs at the same time e.g. different steps in the pipeline (visual testing). To test HSV range values, memory mapped registers were added (e.g. HSV bounds) to experimentally determine the right values using the NIOS II, so that the project could be built and ran quickly on Eclipse rather than having to recompile the entire Verilog project every time. Some algorithms were also tested on MATLAB beforehand to make sure the algorithm made sense before converting it into Verilog or C code for the actual project.

To develop and test video streaming, the Test Pattern Generator II IP Block was used in place of the camera to generate a static test pattern. A simple python script was used to obtain image data via TCP and display them. This meant that the ESP32 and Verilog code could be developed independently and tested to ensure that images were correctly sent from the FPGA video pipeline to the ESP32 to the server via TCP, with minimal integration testing until the feature was ready to be integrated.

In the Command subsystem client-side, React has its own local development server. The page will reload if there is any edit, avoiding restarting the server all the time. To test the server, a simple program is written to simulate TCP packets sending from the ESP32. This allows testing on server without ESP32 to happen. At the same time a C++ program was created to send dummy data by TCP to test sending JSON data.

As only person responsible for integration has the hardware needed for testing, inter-module testing were done in Microsoft Teams call to allow person-in-charge explains the technical details.

6 Evaluation

Subsystem	Evaluation
Vision	**Functional:** The vision module was able to successfully send coordinates of each of the five coloured ping pong balls through the message buffer, to be received by the ESP32. Several modules were created to ensure the correct ping pong balls were detected and to prevent false positive results, where the bounding box was detected even if that coloured ping pong ball was not in the frame. There were four switches used in the project to switch between modes to output and test code visually. One of these switch configurations could output the input image with bounding boxes, which helped determine how optimisations improved the performance of the detection. Reading and sending data from the NIOS II was optimised by using fopen, fwrite, and fread functions by opening a port to the UART core (uart_esp), which was much more reliable than stdin/stdout when testing with the FPGA. Although, this did mean that the small C library could not be used to minimise resource utilisation on the on-chip memory. This on-chip memory used the largest proportion of memory and M9K memory blocks, which was the limiting factor when trying to fit video streaming in the same project as the bounding box detection. The minimum memory size that allowed UART to work was 101600 bytes, which used 128 M9K blocks as seen in Figure 66. On-chip memory resourcesFigure .

	Non-functional: HSV ranges allowed for colour detection that was less sensitive to light compared to the basic detection. Adjusting HSV bounds through the web server allowed for rapid testing, although it still required some fine tuning to get colour detection to detect the boundary boxes. In the final video, the HSV ranges were adjusted, although it was done quickly, and it was getting darker in the room over time. If more time was spent adjusting these values, the boundary boxes would have performed even better. Thus, the subsystem performed well as long as the colour detection was tuned correctly, as the bounding box detection relied heavily on what pixels were detected. The DE-10 Lite always showed a rate of 60-61 fps on the hexadecimal display. The output image also did not have any glitches or timing issues due to a large critical path. This was achieved by pipelining the computation heavy algorithms, such as the RGB to HSV converter, which required multiplication and division. A clear output image was more important than any shift in the output pixels due to the result only being available several clock cycles afterwards due to pipelining). The display could have been shifted, but the inaccuracy of the distance calculation (as it was dependent solely on the size of the bounding box) had a much larger effect on avoiding/displaying the ping-pong balls than any shift of the output pixels. Thus, time was spent improving the bounding box detection rather than fixing the small shift in output pixels.
Control	**Functional:** The ESP32 had met all functional requirements. The programming strategy of using FreeRTOS tasks and queues allowed data to be transmitted between subsystems. The ESP32 is able to successfully obtain bounding boxes from the FPGA and decide how to avoid obstacles. **Non-functional:** The ESP32 can send data with minimal delay, however the presence of Video Streaming had a sizeable impact on the delay.
Video Streaming	**Functional:** The rover can send colour images to the web server to be displayed, at a frame rate of about 0.5 FPS. **Non-Functional:** The whole video streaming process barely uses any CPU processing. The only time any CPU time is used is when the ESP32 sends out a video frame through TCP and when the server resaves the image as a modern 32-bit bitmap for displaying.
Command	**Functional:** The webpage could receive status data from the rover and displays them with a nice user interface. It can also reads in user command and successfully sends to the rover. **Non-Functional:** The webpage has several subpages to display data more neatly. There is also a minimum delay sending user command to the rover.
Drive	**Functional:** Optical flow sensor are properly integrated with motor, although there still have some delay, it is acceptable for motion control. Position data have been sent correctly, so the map can be drawn accurately. Data can be read in the serial monitor and interact functionally with optical flow sensor and motor PCB. Position and angle have been controlled separately, based on the received instruction. **Non-Functional:** Delay in the control system is very fast, which is about 20ms delay, which is acceptable for controlling the drive system accurately. And the fluctuation in the voltage control is less, ensure the motor speed run constantly. The video quality is very high, thus the position sensing achieves the requirement Position and angle are controlled with 1cm and 5 degree error respectively without fluctuation, which is acceptable for avoiding obstacles and draw a accurate map.
Power	**Functional:** The Arduino and Battery Board can meet basic functional requirements. The battery status such as SoC and OCV can be constantly monitored. However, for more complex parts such as charging battery with CC/CV and using MPPT at the same time, or implementing coulomb counting for parallel balanced batteries will be hard **Non-Functional:** The battery charging process through PV can be done by connecting both batteries and PV panels in parallel. However, the current supplied by PV is very small. Although theoretically it is possible to test serial PV with serial battery balance connection, while in rough test the PV is not supplying ideal current, and due to the battery halt we don't have any further chance to test it.
Integration	**Functional:** The UART ports between modules are working, able to perform remote control, exploration, map drawing as well as video streaming. **Non-functional:** The energy supply of Rover come from the 5V cable supply instead of battery/PV panel. The physical movement of the Rover is limited by the wires, hence the wireless communication strength of the ESP32 hasn't been used up.

The vision hardware resource figures are in Appendix Section 10.1.2.

7 Conclusion

7.1 Future developments

7.1.1 Vision

The orientation of the rover was calculated using the direction that the x and y coordinates were moving. This meant that the orientation could not be measured accurately when just turning (as the optical sensor does not change much). The accelerometer on the DE-10 Lite could possibly have been used and the orientation could have been sent to the ESP32 to have a more accurate measure of the direction the rover is turning. The biggest limitation of boundary box detection was the requirement to adjust HSV ranges according to different lighting conditions to perform well. Another algorithm could be developed to dynamically adjust the ranges by detecting the lighting conditions automatically with the camera. Storing the entire frame of the image could help detect this. Edge detection could also be used to detect the ball's position more accurately using an algorithm like the Hough transform [39] to detect circles. This method would be plausible if more memory resources were available.

The orientation of the rover was calculated using the direction that the x and y coordinates were moving. This meant that the orientation could not be measured accurately when just turning (as the optical sensor does not change much). The accelerometer on the DE-10 Lite could possibly have been used and the orientation could have been sent to the ESP32 to have a more accurate measure of the direction the rover is turning. The biggest limitation of boundary box detection was the requirement to adjust HSV ranges according to different lighting conditions to perform well. Another algorithm could be developed to dynamically adjust the ranges by detecting the lighting conditions automatically with the camera. Storing the entire frame of the image could help detect this. Edge detection could also be used to detect the ball's position more accurately using an algorithm like the Hough transform [39] to detect circles. This method would be plausible if more memory resources were available.

7.1.2 Control

Although possible to disable video streaming at compile time, an improvement would be to disable related FreeRTOS tasks at runtime (or change their interval), which would allow for better latency with movement control when needed. An extension to the exploration mode would be to implement specific mapping algorithms for the rover to map the area.

7.1.3 Command

The current system does not have any security methods implemented. To improve security of this system, encryption could be done before sending out TCP packets. Another method is using HTTPS with SSL instead of HTTP as the communication protocol between server and web client. Authentication could also be implemented on the web page, where only users could access certain pages and send out user commands.

7.1.4 Power

The current equipment can meet the basic requirement. However, due to lack of reliable sensors, such as current measure sensor on battery board, extremely slow discharging circuit, not accurate enough voltage measurement port on board and not efficient enough PV panel, it is hard to implement more complicated balance, monitoring or charging methods.

7.1.5 Drive

The motors cause some energy loss, when one motor is turning, it will introduce a voltage that apply on the motor which stop the it moving in the same direction. The motors also can't stop immediately when it received the stop instruction due to momentum, such delay has led to in accuracy in the exploration mode. To improve this, a reverse voltage can be applied on the motors to introduce a IM torque oppose the direction of motion.

8 Intellectual Property (IP) Essay

When designing a complex system, it is important to check whether any idea being worked on has already been filed as part of an IP patent, as the patent owner can legally sue for any infringement. However, in the case of Altera's IP in Quartus, Altera will, on behalf of the license holder, defend against and pay for any damages relating to Altera's IP infringing on other patents [40]. Therefore, an advantage of licensing IP in this case is the design team does not have to worry about infringement over certain parts of the design.

In case any IP is already being used, it is important to check for the terms of use. For example, the 'IP Megafunctions' in Quartus (for example, arithmetic multiply and comparison blocks) can be freely used, altered and distributed provided they are only used to "program devices", without any license needed [40]. Conversely, the 'IP MegaCore' blocks (Nios II and DSP blocks) can only be used for free under the 'OpenCore Plus Evaluation' license with where the FPGA is tethered to the computer. A production rover using Altera IP Cores with a full license would still need an Altera FPGA.

We believe the closest patentable idea is the Verilog module that converts the clocked video pixel output into I2S signals. I2S was originally designed to transmit sound rather than video. I2S has been used to output video from a microcontroller [41], although using it to read in video frames appears to be more novel. The ESP32 already has a camera driver using I2S to take in images, though it connects directly to a camera with 8 data pins. The module developed in this project is slightly different as it uses less pins and is set to obtain smaller 8-bit colour images directly from FPGA fabric.

9 References

[1] E. Stott, "GitHub EEE2Rover," 6 May 2021. [Online]. Available: https://github.com/edstott/EEE2Rover. [Accessed 11 May 2021].

[2] B. Wicht, "Deep Learning feature Extraction for Image Processing," ResearchGate, Fribourg, 2018.

[3] J. Xu, J. Geng, X. Yan, H. Wang and H. Xia, "Implementing real time image processing algorithm on FPGA," in *Twelfth International Conference on Digital Image Processing*, Osaka, Japan, 2020.

[4] S. H. Ahn, "Convolution," 2018. [Online]. Available: http://www.songho.ca/dsp/convolution/convolution.html. [Accessed 26 May 2021].

[5] Chinmay, "The IE Blog," 19 January 2020. [Online]. Available: https://ie.nitk.ac.in/blog/2020/01/19/algorithms-for-adjusting-brightness-and-contrast-of-an-image/. [Accessed 15 May 2021].

[6] J. H. Bear, "Lifewire," 11 November 2019. [Online]. Available: https://www.lifewire.com/what-is-hsv-in-design-1078068. [Accessed 6 June 2020].

[7] M. Hanumantharaju, G. Vishalakshi, S. Halvi and S. Satish, "A Novel FPGA Based Reconfigurable Architecture for Image Color Space Conversion," in *International Conference On Recent trends in Computing, Communication and Information Technologies*, Vellore, 2011.

[8] Altera Corporation, "Avalon-ST Serial Peripheral Interface Core," November 2009. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/nios2/qts_qii55009.pdf. [Accessed 6 June 2021].

[9] Espressif Systems, "I2S - ESP32 --- ESP-IDF Programming Guide," 2021. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/peripherals/i2s.html. [Accessed 29 May 2021].

[10] Expressif Systems, "I2S Slave mode with external MCLK - ESP32 Forum," 26 June 2020. [Online]. Available: https://www.esp32.com/viewtopic.php?t=16290. [Accessed 29 May 2021].

[11] Expressif Systems, "ESP32 Technical Reference Manual," March 2021. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf. [Accessed 29 May 2021].

[12] Expressif Systems, "I2S 24bit Slave - ESP32 Forum," 5 August 2018. [Online]. Available: https://www.esp32.com/viewtopic.php?t=6650. [Accessed 29 May 2021].

[13] Expressif Systems, "Store I2S bytes in different order - ESP32 Forum," 29 January 2019. [Online]. Available: https://esp32.com/viewtopic.php?t=9028. [Accessed 30 May 2021].

[14] Altera Corporation, "Video and Image Processing Suite User Guide," 31 October 2016. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/archives/ug-vip-16.1.pdf. [Accessed 25 May 2021].

[15] fourcc.org, "CLJR yuv pixel format," [Online]. Available: https://www.fourcc.org/pixel-format/yuv-cljr/. [Accessed 29 May 2021].

[16] Microsoft Corporation, "Bitmap Header Types - Microsoft Docs," 31 May 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/win32/gdi/bitmap-header-types. [Accessed 30 May 2021].

[17] R. Barry, *Mastering the FreeRTOS™ Real Time Kernel,* Real Time Engineers Ltd, 2016.

[18] "React Router," React Training, 2021. [Online]. Available: http://reactrouter.com. [Accessed 2021].

[19] "React," Facebook, 2021. [Online]. Available: https://reactjs.org/docs/hooks-state.html#gatsby-focus-wrapper. [Accessed 2021].

[20] "Current Weather API," OpenWeather, 2021. [Online]. Available: https://openweathermap.org/current. [Accessed 2021].

[21] "int - Arduino Reference," Arduino, [Online]. Available: https://www.arduino.cc/reference/en/language/variables/data-types/int/. [Accessed 21 5 2021].

[22] "long - Arduino Reference," Arduino, [Online]. Available: https://www.arduino.cc/reference/en/language/variables/data-types/long/. [Accessed 20 5 2021].

[23] A. Smets, "Maximum Power Point Tracking," Delft University of Technology, [Online]. Available: https://www.youtube.com/watch?v=5Us5mM87PU8.

[24] S. Motahhir, "Arduino based MPPT Controller," [Online]. Available: https://create.arduino.cc/projecthub/motahhir/arduino-based-mppt-controller-0560db.

[25] T. U. o. Munich, "Discharge and Charge Process of a Conventional Lithium-Ion Battery Cell," [Online]. Available: https://www.youtube.com/watch?v=p8ecZ5oK7Fc.

[26] E. Osmanbasic, "Battery Management Systems–Part 3: Battery Charging Methods," 21 2 2020. [Online]. Available: https://www.engineering.com/story/battery-management-systemspart-3-battery-charging-methods.

[27] P. Technology, "How to charge Lithium Iron Phosphate Rechargeable Lithium Ion Batteries," [Online]. Available: https://www.powerstream.com/LLLF.htm.

[28] B. J. University, "Improved Li Battery CC-CV Charging Method". China Patent CN 101814640 B, 24 10 2012.

[29] S. S. Madani, E. Schaltz and S. K. Kær, "An Electrical Equivalent Circuit Model of a LithiumTitanate Oxide Battery," *batteries*, pp. 3-5, 13 3 2019.

[30] M. Jain and D. Martin, "State of Charge Estimation Problem," The University of Texas at Dallas, [Online]. Available: https://labs.utdallas.edu/essl/projects/state-of-charge-estimation-problem/.

[31] I. Baccouche, S. Jemmali, A. Mlayah, B. Manai and N. E. B. Amara, "Implementation of an Improved Coulomb-Counting Algorithm Based on a Piecewise SOC-OCV Relationship for SOC Estimation of Li-Ion Battery," *INTERNATIONAL JOURNAL of RENEWABLE ENERGY RESEARCH,* pp. 2-3.

[32] W.-Y. Chang, "The State of Charge Estimating Methods for Battery: A Review," pp. 2-4.

[33] Z. Yu, R. Huai and L. Xiao, "State-of-Charge Estimation for Lithium-Ion Batteries Using," *energies,* pp. 1-20, 30 7 2015.

[34] N. Noura, L. Boulon and S. Jemeï, "A Review of Battery State of Health Estimation," *World Electronic Vehicle Journal,* pp. 3-4, 16 10 2020.

[35] J. Sun, P. Yang, R. Lu, G. Wei and C. Zhu, "LiFePO4 optimal operation temperature range analysis for EV/HEV Analysis for EV/HEV," pp. 1-2.

[36] MICROCHIP, " ATmega4808/4809 Data Sheet," [Online]. Available: http://ww1.microchip.com/downloads/en/DeviceDoc/ATmega4808-4809-Data-Sheet-DS40002173A.pdf.

[37] ArduinoJson, "ArduinoJson," [Online]. Available: https://arduinojson.org/?utm_source=meta&utm_medium=library.properties.

[38] Expressif Systems, "Protocols Examples - ESP-IDF Github Repository," [Online]. Available: https://github.com/espressif/esp-idf/tree/master/examples/protocols. [Accessed 15 June 2021].

[39] H. Rhody, "Lecture 10: Hough Circle Transform," Rochester Institute of Technology, Rochester, 2005.

[40] Altera Corporation, "QUARTUS(R) PRIME LICENSE AGREEMENT VERSION 16.1," 2016. [Online]. Available: https://download.altera.com/akdlm/software/acdsinst/16.1/196/licenses/license.txt. [Accessed 7 June 2021].

[41] Hackaday, "Software Defined Television on an ESP32," [Online]. Available: https://hackaday.com/2018/02/22/software-defined-television-on-an-esp32/. [Accessed 115 June 2021].

[42] Philips Semiconductors, "I2S Bus Specification," Febraury 1986. [Online]. Available: https://www.sparkfun.com/datasheets/BreakoutBoards/I2SBUS.pdf. [Accessed 25 May 2021].

[43] R. C. Niemietz., "MSX2 Sceen Palette," 1 February 2008. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/3/30/MSX2_Screen8_palette.png. [Accessed 5 May 2021].

[44] OpenWeather, "https://openweathermap.org/current," [Online]. [Accessed 2021].

10 Appendices

10.1 Vision

10.1.1 Image processing

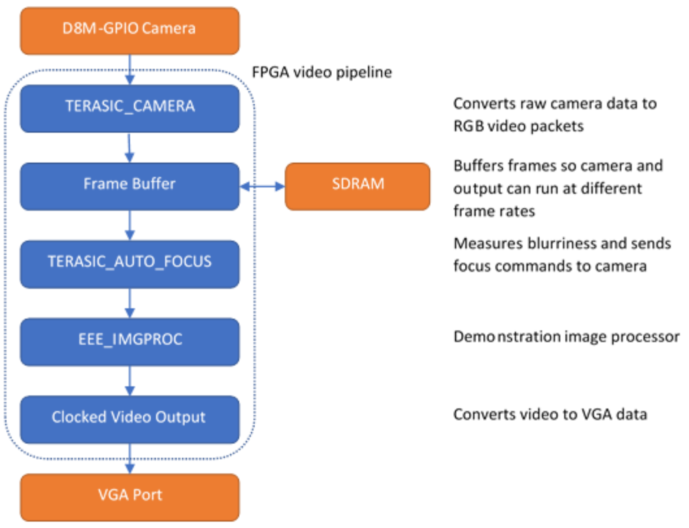

D8M-GPIO Camera
FPGA video pipeline
TERASIC_CAMERA
Converts raw camera data to RGB video packets
Frame Buffer
SDRAM
Buffers frames so camera and output can run at different frame rates
TERASIC_AUTO_FOCUS
Measures blurriness and sends focus commands to camera
EEE_IMGPROC
Demonstration image processor
Clocked Video Output
Converts video to VGA data
VGA Port

Figure 53. Streaming video pipeline diagram from Edward Stott's GitHub [1]

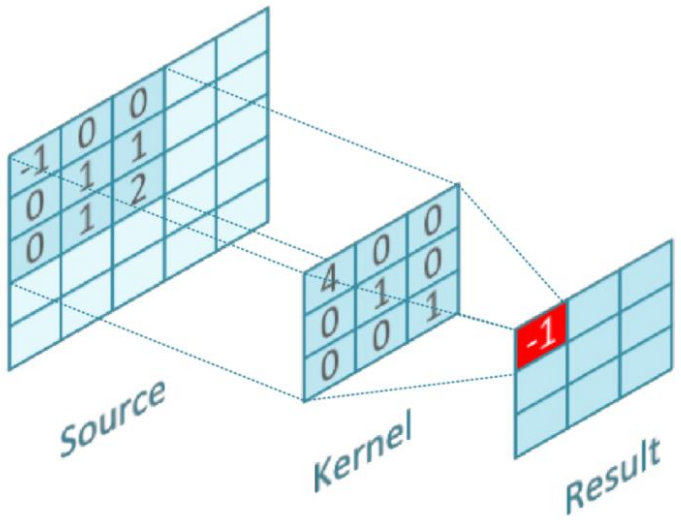


Figure 54. Demonstration of 3x3 convolution [4]

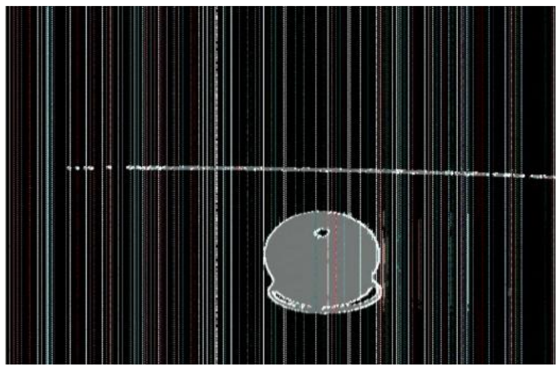


Figure 55. Glitches on image due to large critical path

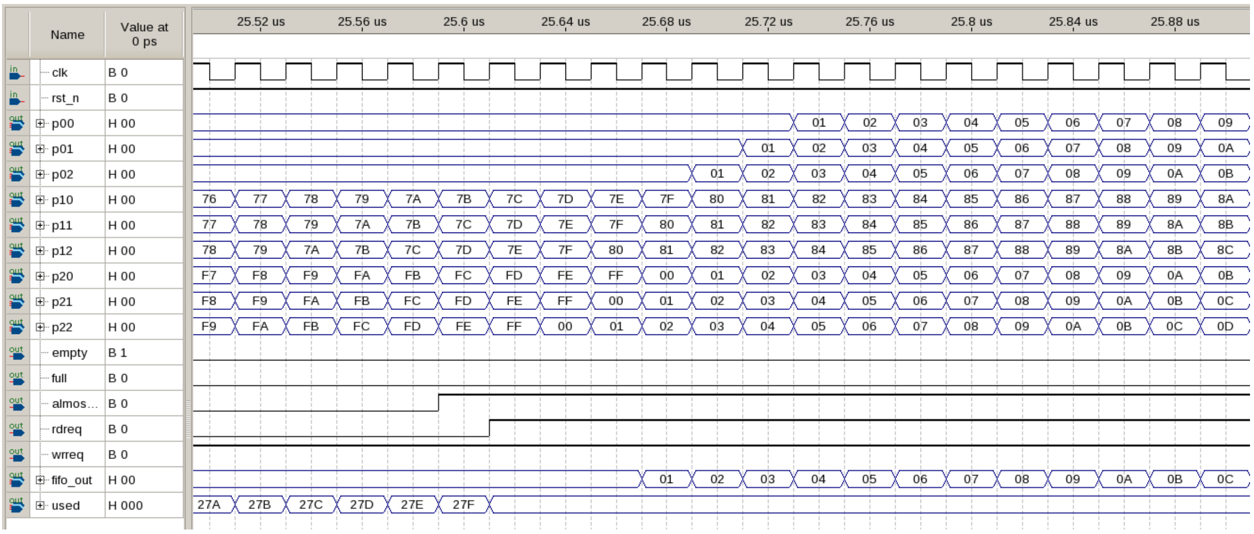


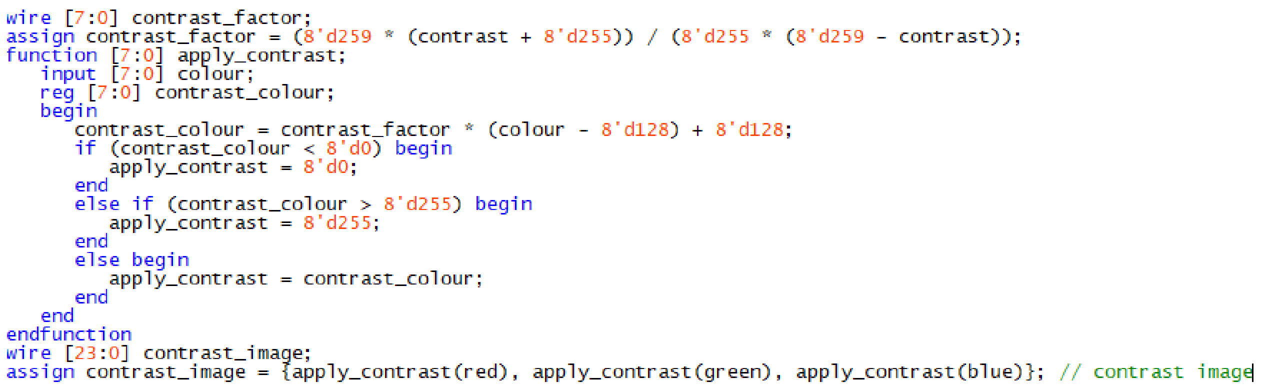
Figure 56. Row FIFO Simulation

$$\begin{bmatrix} \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \\ \frac{2}{16} & \frac{4}{16} & \frac{2}{16} \\ \frac{1}{16} & \frac{2}{16} & \frac{1}{16} \end{bmatrix} = \begin{bmatrix} \frac{1}{4} \\ \frac{2}{4} \\ \frac{1}{4} \end{bmatrix} \cdot \begin{bmatrix} \frac{1}{4} & \frac{2}{4} & \frac{1}{4} \end{bmatrix}$$

Figure 57. 3x3 Gaussian blur equation [2]

```
if (in_valid) begin
    //assigning registers
    p22 <= in_grey;
    p21 <= p22;
    p20 <= p21;
    p12 <= out_1;
    p11 <= p12;
    p10 <= p11;
    p02 <= out_2;
    p01 <= p02;
    p00 <= p01;

    // read when almost full (639 words in buffer since it takes 2 cycles to go from almost_full -> rdreq -> out)
    if (almost_full_1) begin // if buffer1 is full - read and write
        rdreq_1 <= 1'b1;
        wrreq_1 <= 1'b1;
    end
    else begin // if not full, just write
        rdreq_1 <= 1'b0;
        wrreq_1 <= 1'b1;
    end

    //row buffer 2
    if (almost_full_2) begin // if buffer2 is full - read and write
        rdreq_2 <= 1'b1;
        wrreq_2 <= 1'b1;
    end
    else if (almost_full_1) begin //if not full, only write if buffer 1 is full
        rdreq_2 <= 1'b0;
        wrreq_2 <= 1'b1;
    end
    else begin
        rdreq_2 <= 1'b0;
        wrreq_2 <= 1'b0;
    end
end
else begin
    // don't read or write if pixel is not valid
    rdreq_1 <= 1'b0;
    wrreq_1 <= 1'b0;
    rdreq_2 <= 1'b0;
    wrreq_2 <= 1'b0;
end
```

Figure 58. 3x3 Window module logic (WINDOW_3x3.v)



```
wire [7:0] contrast_factor;
assign contrast_factor = (8'd259 * (contrast + 8'd255)) / (8'd255 * (8'd259 - contrast));
function [7:0] apply_contrast;
   input [7:0] colour;
   reg [7:0] contrast_colour;
   begin
      contrast_colour = contrast_factor * (colour - 8'd128) + 8'd128;
      if (contrast_colour < 8'd0) begin
         apply_contrast = 8'd0;
      end
      else if (contrast_colour > 8'd255) begin
         apply_contrast = 8'd255;
      end
      else begin
         apply_contrast = contrast_colour;
      end
   end
endfunction
wire [23:0] contrast_image;
assign contrast_image = {apply_contrast(red), apply_contrast(green), apply_contrast(blue)}; // contrast image
```

Figure 59. Contrast function code

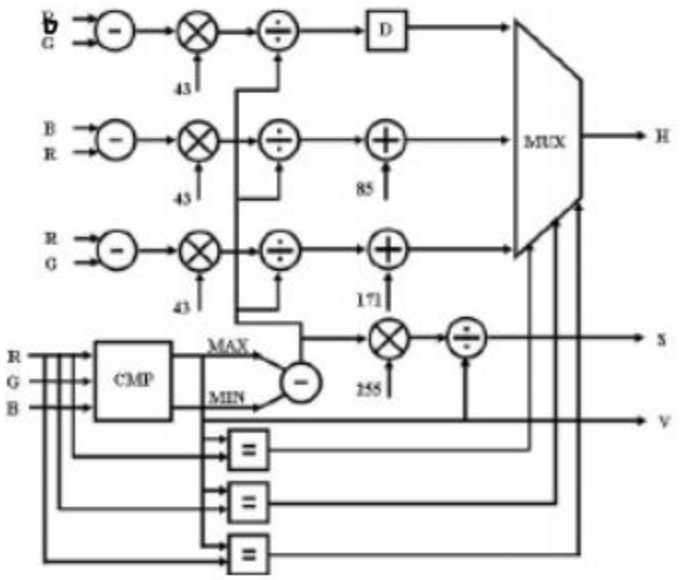
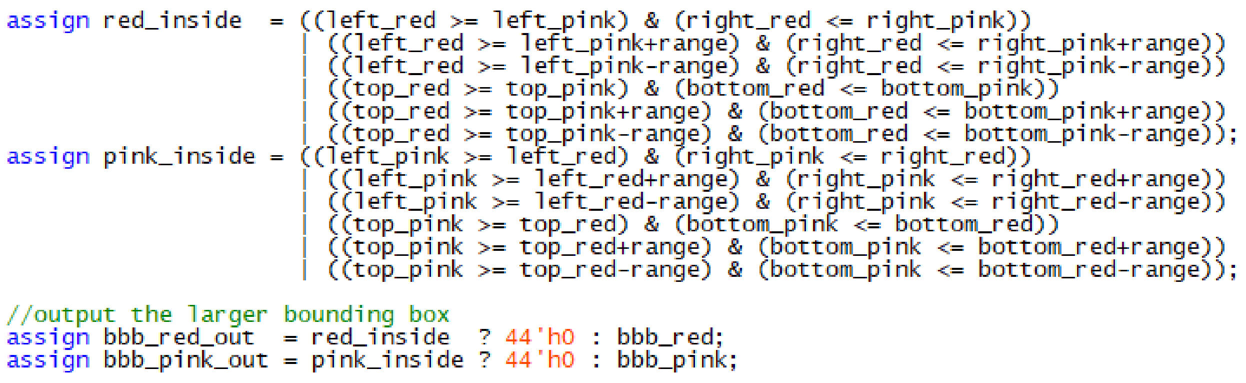


Figure 60. RGB to HSV Hardware Architecture [7]



```
assign red_inside  = ((left_red >= left_pink) & (right_red <= right_pink))
                   | ((left_red >= left_pink+range) & (right_red <= right_pink+range))
                   | ((left_red >= left_pink-range) & (right_red <= right_pink-range))
                   | ((top_red >= top_pink) & (bottom_red <= bottom_pink))
                   | ((top_red >= top_pink+range) & (bottom_red <= bottom_pink+range))
                   | ((top_red >= top_pink-range) & (bottom_red <= bottom_pink-range));
assign pink_inside = ((left_pink >= left_red) & (right_pink <= right_red))
                   | ((left_pink >= left_red+range) & (right_pink <= right_red+range))
                   | ((left_pink >= left_red-range) & (right_pink <= right_red-range))
                   | ((top_pink >= top_red) & (bottom_pink <= bottom_red))
                   | ((top_pink >= top_red+range) & (bottom_pink <= bottom_red+range))
                   | ((top_pink >= top_red-range) & (bottom_pink <= bottom_red-range));

//output the larger bounding box
assign bbb_red_out  = red_inside  ? 44'h0 : bbb_red;
assign bbb_pink_out = pink_inside ? 44'h0 : bbb_pink;
```

Figure 61. Differentiating red and pink pixels code

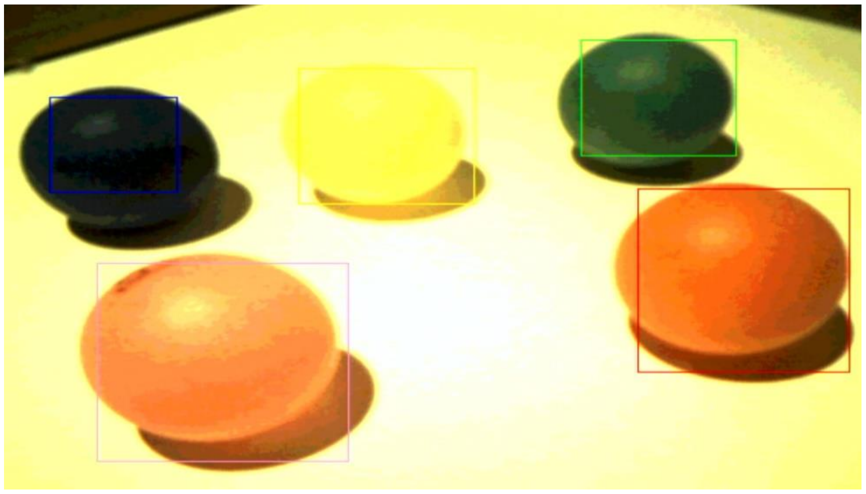
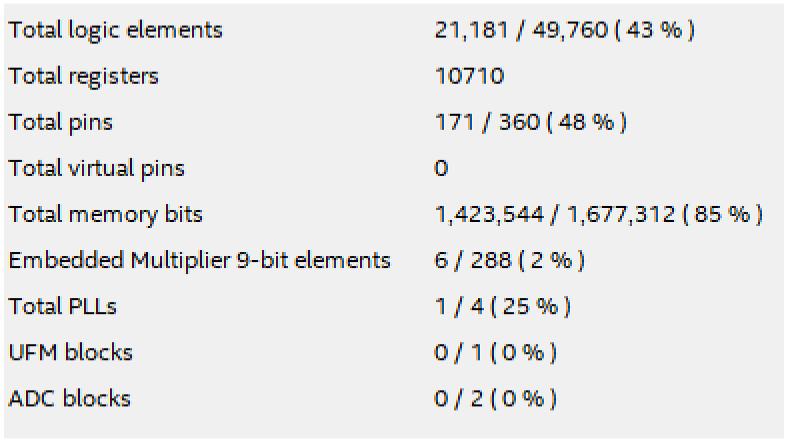


Figure 62. Bounding box detection for five ping pong balls

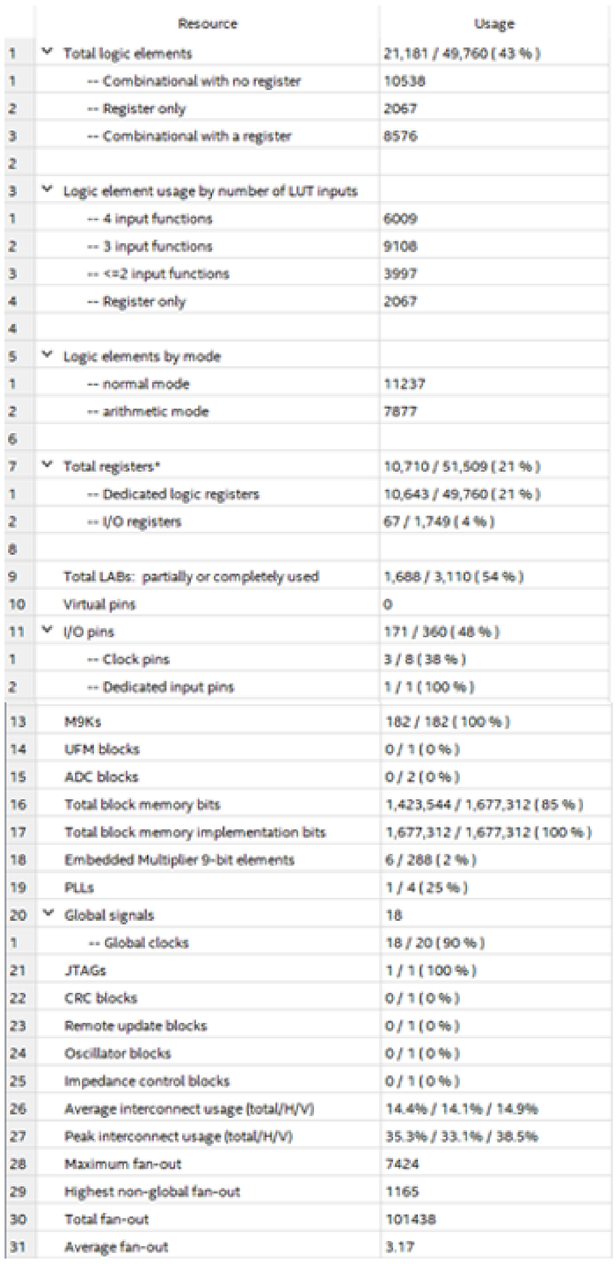
10.1.2 Hardware resources

10.1.2.1 Boundary box Quartus project



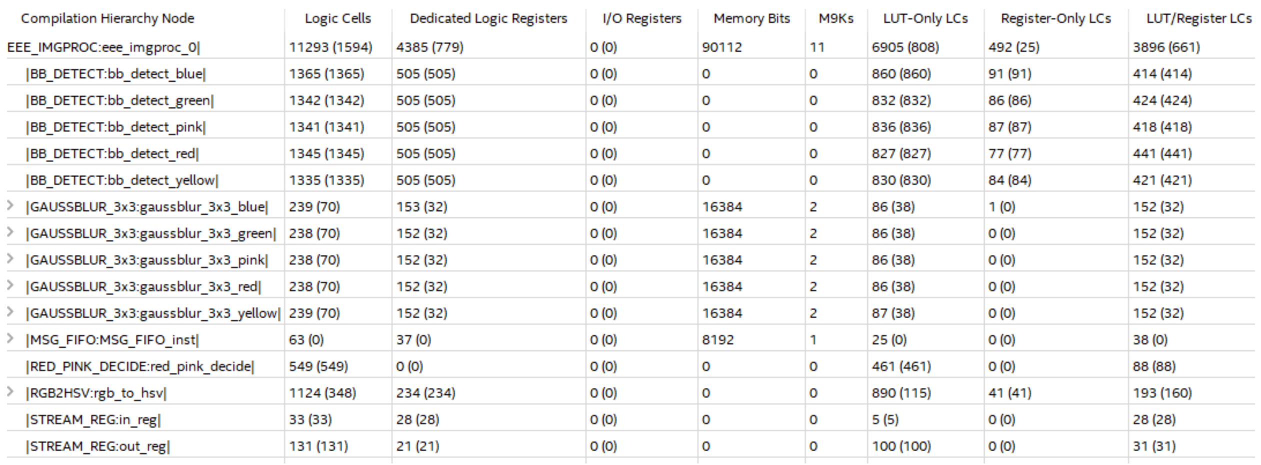
Total logic elements	21,181 / 49,760 (43 %)
Total registers	10710
Total pins	171 / 360 (48 %)
Total virtual pins	0
Total memory bits	1,423,544 / 1,677,312 (85 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 63. Compilation summary for bounding box project



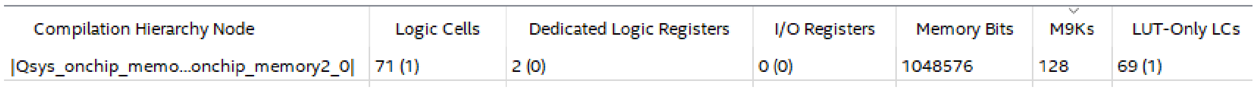
	Resource	Usage
1	Total logic elements	21,181 / 49,760 (43 %)
1	-- Combinational with no register	10538
2	-- Register only	2067
3	-- Combinational with a register	8576
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	6009
2	-- 3 input functions	9108
3	-- <=2 input functions	3997
4	-- Register only	2067
4		
5	Logic elements by mode	
1	-- normal mode	11237
2	-- arithmetic mode	7877
6		
7	Total registers*	10,710 / 51,509 (21 %)
1	-- Dedicated logic registers	10,643 / 49,760 (21 %)
2	-- I/O registers	67 / 1,749 (4 %)
8		
9	Total LABs: partially or completely used	1,688 / 3,110 (54 %)
10	Virtual pins	0
11	I/O pins	171 / 360 (48 %)
1	-- Clock pins	3 / 8 (38 %)
2	-- Dedicated input pins	1 / 1 (100 %)
13	M9Ks	182 / 182 (100 %)
14	UFM blocks	0 / 1 (0 %)
15	ADC blocks	0 / 2 (0 %)
16	Total block memory bits	1,423,544 / 1,677,312 (85 %)
17	Total block memory implementation bits	1,677,312 / 1,677,312 (100 %)
18	Embedded Multiplier 9-bit elements	6 / 288 (2 %)
19	PLLs	1 / 4 (25 %)
20	Global signals	18
1	-- Global clocks	18 / 20 (90 %)
21	JTAGs	1 / 1 (100 %)
22	CRC blocks	0 / 1 (0 %)
23	Remote update blocks	0 / 1 (0 %)
24	Oscillator blocks	0 / 1 (0 %)
25	Impedance control blocks	0 / 1 (0 %)
26	Average interconnect usage (total/H/V)	14.4% / 14.1% / 14.9%
27	Peak interconnect usage (total/H/V)	35.3% / 33.1% / 38.5%
28	Maximum fan-out	7424
29	Highest non-global fan-out	1165
30	Total fan-out	101438
31	Average fan-out	3.17

Figure 64. Resource usage summary for bounding box project



Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
EEE_IMGPROC:eee_imgproc_0|	11293 (1594)	4385 (779)	0 (0)	90112	11	6905 (808)	492 (25)	3896 (661)
|BB_DETECT:bb_detect_blue|	1365 (1365)	505 (505)	0 (0)	0	0	860 (860)	91 (91)	414 (414)
|BB_DETECT:bb_detect_green|	1342 (1342)	505 (505)	0 (0)	0	0	832 (832)	86 (86)	424 (424)
|BB_DETECT:bb_detect_pink|	1341 (1341)	505 (505)	0 (0)	0	0	836 (836)	87 (87)	418 (418)
|BB_DETECT:bb_detect_red|	1345 (1345)	505 (505)	0 (0)	0	0	827 (827)	77 (77)	441 (441)
|BB_DETECT:bb_detect_yellow|	1335 (1335)	505 (505)	0 (0)	0	0	830 (830)	84 (84)	421 (421)
|GAUSSBLUR_3x3:gaussblur_3x3_blue|	239 (70)	153 (32)	0 (0)	16384	2	86 (38)	1 (0)	152 (32)
|GAUSSBLUR_3x3:gaussblur_3x3_green|	238 (70)	152 (32)	0 (0)	16384	2	86 (38)	0 (0)	152 (32)
|GAUSSBLUR_3x3:gaussblur_3x3_pink|	238 (70)	152 (32)	0 (0)	16384	2	86 (38)	0 (0)	152 (32)
|GAUSSBLUR_3x3:gaussblur_3x3_red|	238 (70)	152 (32)	0 (0)	16384	2	86 (38)	0 (0)	152 (32)
|GAUSSBLUR_3x3:gaussblur_3x3_yellow|	239 (70)	152 (32)	0 (0)	16384	2	87 (38)	0 (0)	152 (32)
|MSG_FIFO:MSG_FIFO_inst|	63 (0)	37 (0)	0 (0)	8192	1	25 (0)	0 (0)	38 (0)
|RED_PINK_DECIDE:red_pink_decide|	549 (549)	0 (0)	0 (0)	0	0	461 (461)	0 (0)	88 (88)
|RGB2HSV:rgb_to_hsv|	1124 (348)	234 (234)	0 (0)	0	0	890 (115)	41 (41)	193 (160)
|STREAM_REG:in_reg|	33 (33)	28 (28)	0 (0)	0	0	5 (5)	0 (0)	28 (28)
|STREAM_REG:out_reg|	131 (131)	21 (21)	0 (0)	0	0	100 (100)	0 (0)	31 (31)

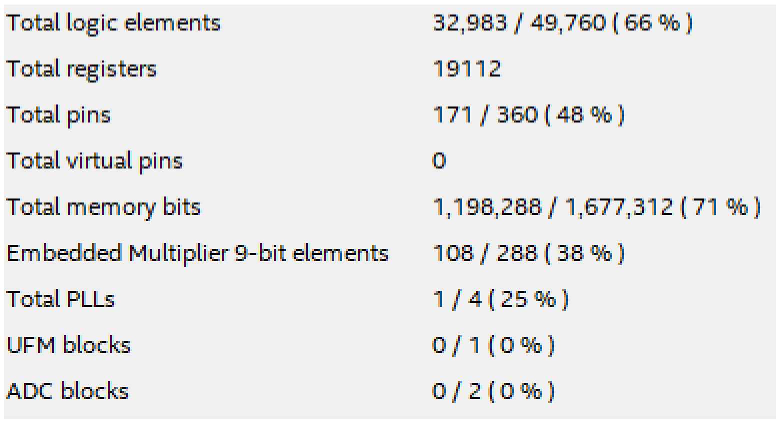
Figure 65. Resource utilisation by entity for EEE_IMGPROC in boundary box project



Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	LUT-Only LCs
|Qsys_onchip_memo...onchip_memory2_0|	71 (1)	2 (0)	0 (0)	1048576	128	69 (1)

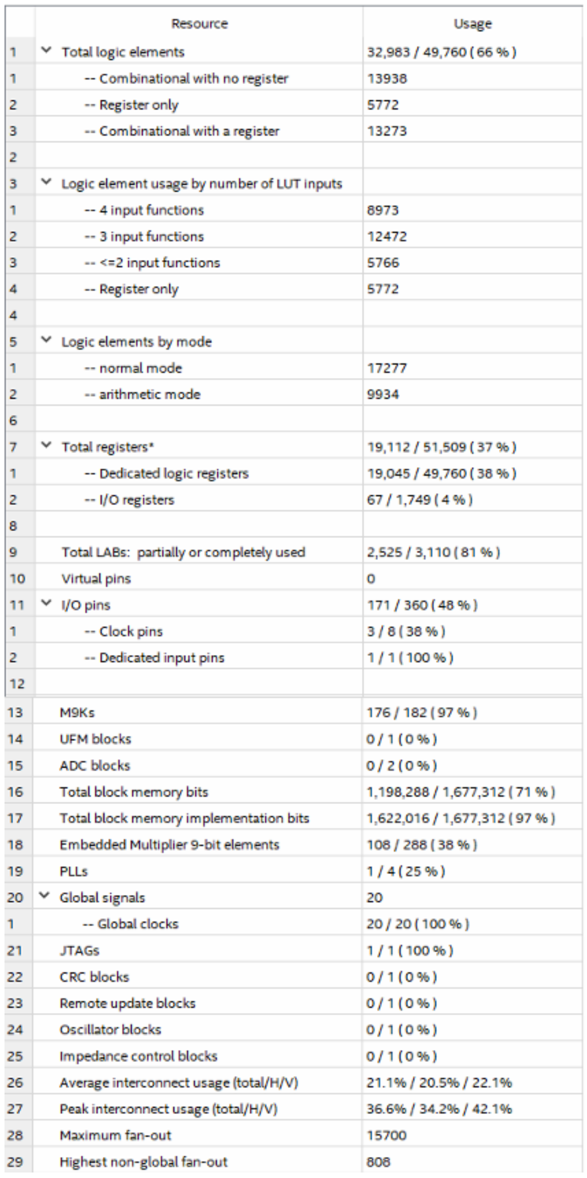
Figure 66. On-chip memory resources

10.1.2.2 Video stream Quartus project



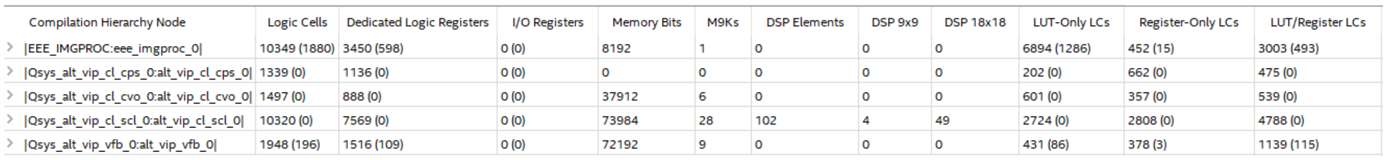
Total logic elements	32,983 / 49,760 (66 %)
Total registers	19112
Total pins	171 / 360 (48 %)
Total virtual pins	0
Total memory bits	1,198,288 / 1,677,312 (71 %)
Embedded Multiplier 9-bit elements	108 / 288 (38 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figure 67. Compilation summary for video streaming project



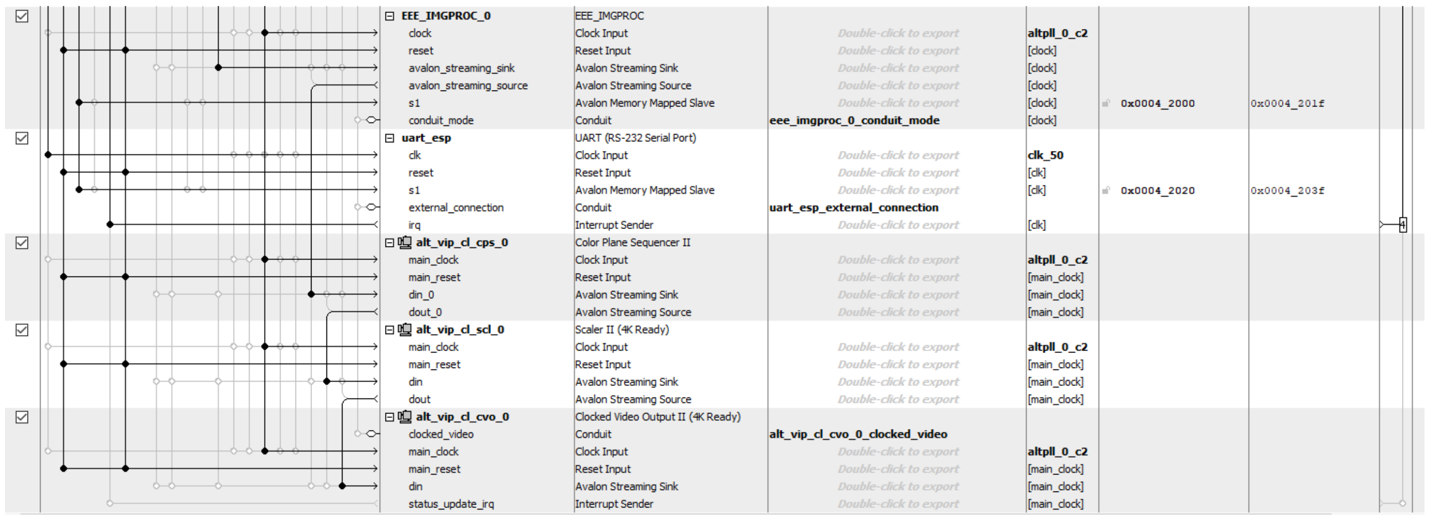
	Resource	Usage
1	Total logic elements	32,983 / 49,760 (66 %)
1	-- Combinational with no register	13938
2	-- Register only	5772
3	-- Combinational with a register	13273
2		
3	Logic element usage by number of LUT inputs	
1	-- 4 input functions	8973
2	-- 3 input functions	12472
3	-- <=2 input functions	5766
4	-- Register only	5772
4		
5	Logic elements by mode	
1	-- normal mode	17277
2	-- arithmetic mode	9934
6		
7	Total registers*	19,112 / 51,509 (37 %)
1	-- Dedicated logic registers	19,045 / 49,760 (38 %)
2	-- I/O registers	67 / 1,749 (4 %)
8		
9	Total LABs: partially or completely used	2,525 / 3,110 (81 %)
10	Virtual pins	0
11	I/O pins	171 / 360 (48 %)
1	-- Clock pins	3 / 8 (38 %)
2	-- Dedicated input pins	1 / 1 (100 %)
12		
13	M9Ks	176 / 182 (97 %)
14	UFM blocks	0 / 1 (0 %)
15	ADC blocks	0 / 2 (0 %)
16	Total block memory bits	1,198,288 / 1,677,312 (71 %)
17	Total block memory implementation bits	1,622,016 / 1,677,312 (97 %)
18	Embedded Multiplier 9-bit elements	108 / 288 (38 %)
19	PLLs	1 / 4 (25 %)
20	Global signals	20
1	-- Global clocks	20 / 20 (100 %)
21	JTAGs	1 / 1 (100 %)
22	CRC blocks	0 / 1 (0 %)
23	Remote update blocks	0 / 1 (0 %)
24	Oscillator blocks	0 / 1 (0 %)
25	Impedance control blocks	0 / 1 (0 %)
26	Average interconnect usage (total/H/V)	21.1% / 20.5% / 22.1%
27	Peak interconnect usage (total/H/V)	36.6% / 34.2% / 42.1%
28	Maximum fan-out	15700
29	Highest non-global fan-out	808

Figure 68. Resource usage summary for video streaming project



Compilation Hierarchy Node	Logic Cells	Dedicated Logic Registers	I/O Registers	Memory Bits	M9Ks	DSP Elements	DSP 9x9	DSP 18x18	LUT-Only LCs	Register-Only LCs	LUT/Register LCs
|EEE_IMGPROC:eee_imgproc_0|	10349 (1880)	3450 (598)	0 (0)	8192	1	0	0	0	6894 (1286)	452 (15)	3003 (493)
|Qsys_alt_vip_cl_cps_0:alt_vip_cl_cps_0|	1339 (0)	1136 (0)	0 (0)	0	0	0	0	0	202 (0)	662 (0)	475 (0)
|Qsys_alt_vip_cl_cvo_0:alt_vip_cl_cvo_0|	1497 (0)	888 (0)	0 (0)	37912	6	0	0	0	601 (0)	357 (0)	539 (0)
|Qsys_alt_vip_cl_scl_0:alt_vip_cl_scl_0|	10320 (0)	7569 (0)	0 (0)	73984	28	102	4	49	2724 (0)	2808 (0)	4788 (0)
|Qsys_alt_vip_vfb_0:alt_vip_vfb_0|	1948 (196)	1516 (109)	0 (0)	72192	9	0	0	0	431 (86)	378 (3)	1139 (115)

Figure 69. Resource utilisation by entity for video streaming project



☑	**EEE_IMGPROC_0**	EEE_IMGPROC			
	clock	Clock Input	Double-click to export	**altpll_0_c2**		
	reset	Reset Input	Double-click to export	[clock]		
	avalon_streaming_sink	Avalon Streaming Sink	Double-click to export	[clock]		
	avalon_streaming_source	Avalon Streaming Source	Double-click to export	[clock]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clock]	0x0004_2000	0x0004_201f
	conduit_mode	Conduit	**eee_imgproc_0_conduit_mode**	[clock]		
☑	**uart_esp**	UART (RS-232 Serial Port)				
	clk	Clock Input	Double-click to export	**clk_50**		
	reset	Reset Input	Double-click to export	[clk]		
	s1	Avalon Memory Mapped Slave	Double-click to export	[clk]	0x0004_2020	0x0004_203f
	external_connection	Conduit	**uart_esp_external_connection**			
	irq	Interrupt Sender	Double-click to export	[clk]		
☑	**alt_vip_cl_cps_0**	Color Plane Sequencer II				
	main_clock	Clock Input	Double-click to export	**altpll_0_c2**		
	main_reset	Reset Input	Double-click to export	[main_clock]		
	din_0	Avalon Streaming Sink	Double-click to export	[main_clock]		
	dout_0	Avalon Streaming Source	Double-click to export	[main_clock]		
☑	**alt_vip_cl_scl_0**	Scaler II (4K Ready)				
	main_clock	Clock Input	Double-click to export	**altpll_0_c2**		
	main_reset	Reset Input	Double-click to export	[main_clock]		
	din	Avalon Streaming Sink	Double-click to export	[main_clock]		
	dout	Avalon Streaming Source	Double-click to export	[main_clock]		
☑	**alt_vip_cl_cvo_0**	Clocked Video Output II (4K Ready)				
	clocked_video	Conduit	**alt_vip_cl_cvo_0_clocked_video**			
	main_clock	Clock Input	Double-click to export	**altpll_0_c2**		
	main_reset	Reset Input	Double-click to export	[main_clock]		
	din	Avalon Streaming Sink	Double-click to export	[main_clock]		
	status_update_irq	Interrupt Sender	Double-click to export	[main_clock]		

Figure 70. QSYS project video streaming IP connections

10.2 Video Stream

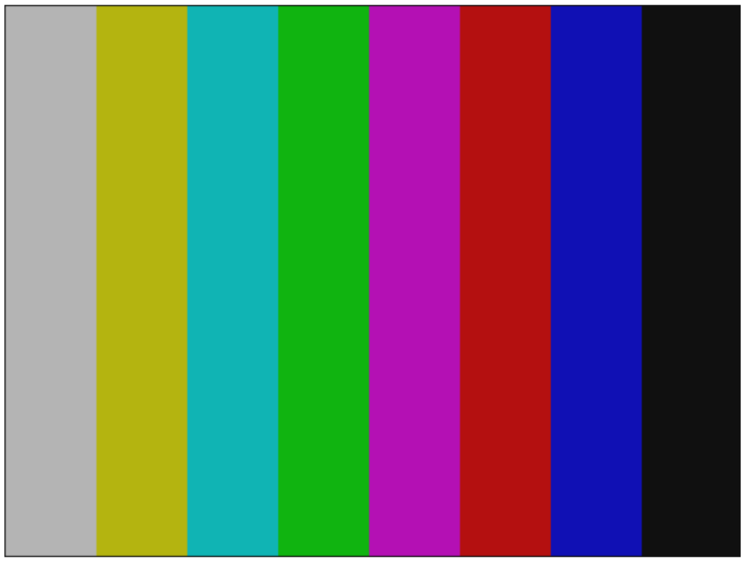


Figure 71. Color bar test pattern produced by Test Pattern Generator II IP Core [14].

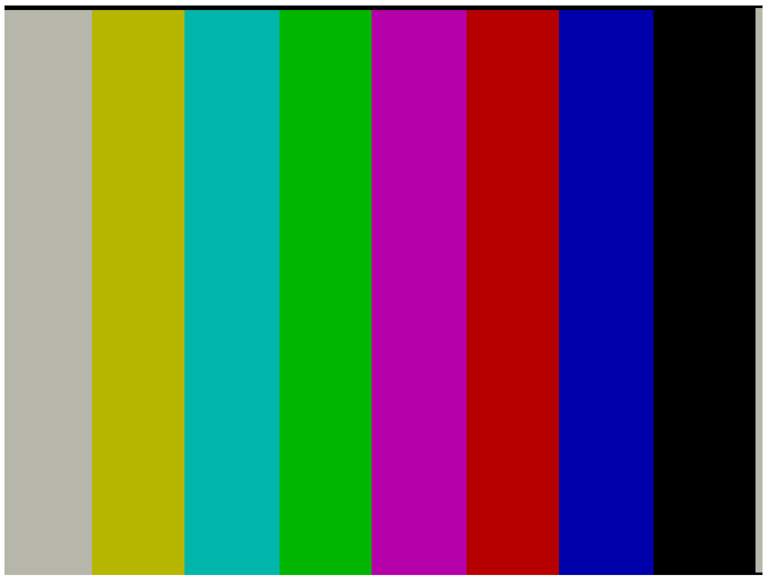


Figure 72. Bitmap image obtained from ESP32

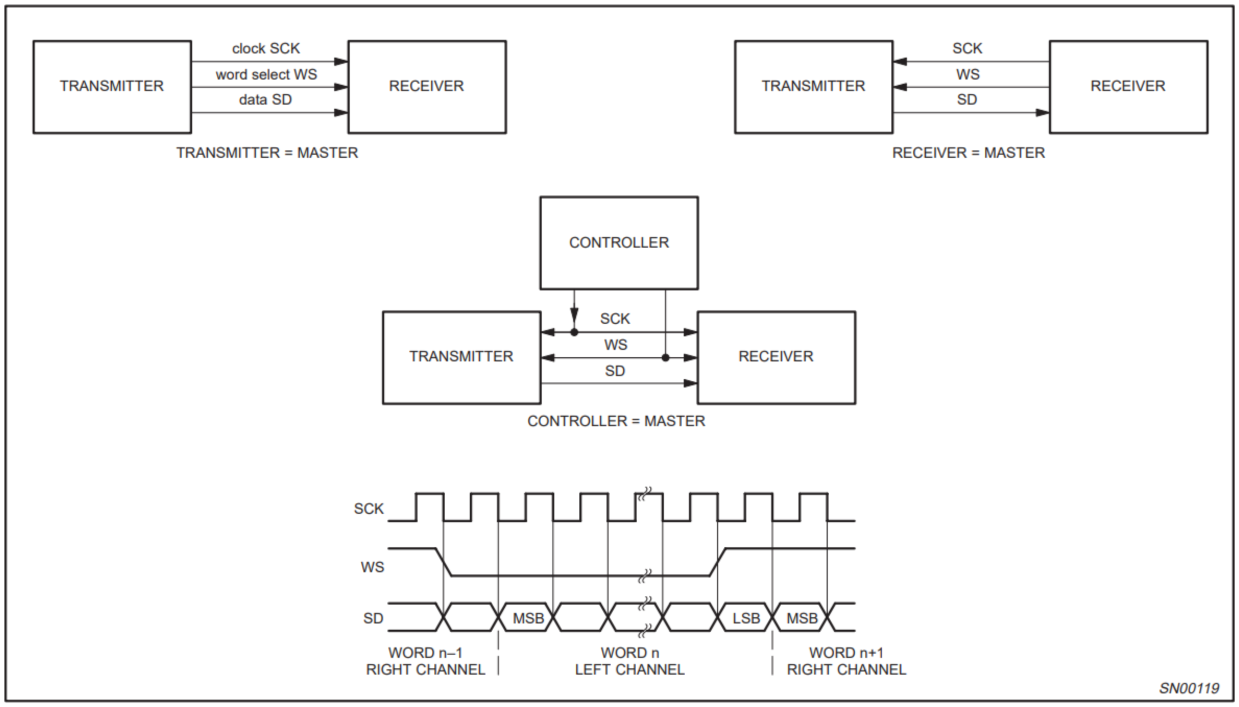


Figure 73. I2S Protocol timing and configurations [42]. Note that the original specification is shown where a new data word is sent on the clock cycle following the WS transition, compared to on the same clock cycle which is used in this project.

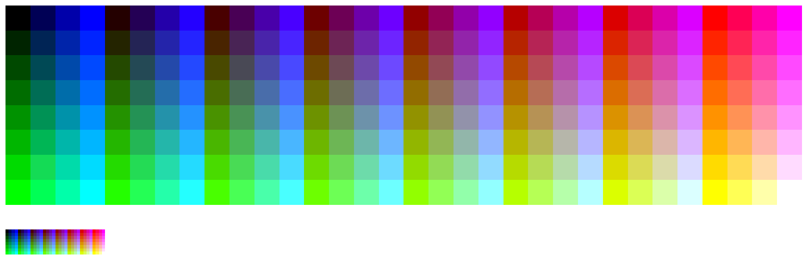
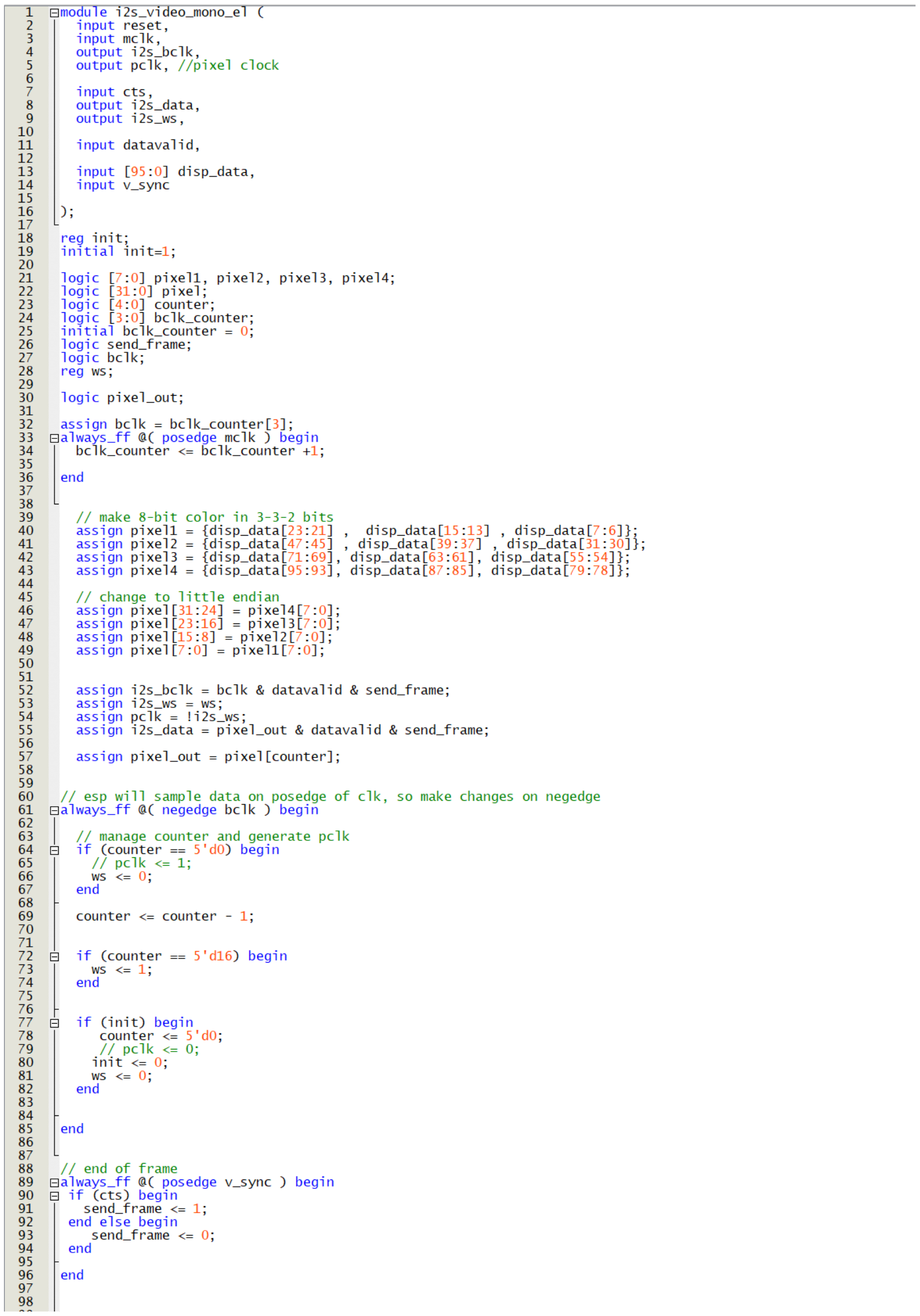


Figure 74. The 3-3-2-bit RGB color palette [43].



```verilog
module i2s_video_mono_el (
  input reset,
  input mclk,
  output i2s_bclk,
  output pclk, //pixel clock

  input cts,
  output i2s_data,
  output i2s_ws,

  input datavalid,

  input [95:0] disp_data,
  input v_sync

);

reg init;
initial init=1;

logic [7:0] pixel1, pixel2, pixel3, pixel4;
logic [31:0] pixel;
logic [4:0] counter;
logic [3:0] bclk_counter;
initial bclk_counter = 0;
logic send_frame;
logic bclk;
reg ws;

logic pixel_out;

assign bclk = bclk_counter[3];
always_ff @( posedge mclk ) begin
  bclk_counter <= bclk_counter +1;

end


  // make 8-bit color in 3-3-2 bits
  assign pixel1 = {disp_data[23:21] ,  disp_data[15:13] , disp_data[7:6]};
  assign pixel2 = {disp_data[47:45] , disp_data[39:37] , disp_data[31:30]};
  assign pixel3 = {disp_data[71:69], disp_data[63:61], disp_data[55:54]};
  assign pixel4 = {disp_data[95:93], disp_data[87:85], disp_data[79:78]};

  // change to little endian
  assign pixel[31:24] = pixel4[7:0];
  assign pixel[23:16] = pixel3[7:0];
  assign pixel[15:8] = pixel2[7:0];
  assign pixel[7:0] = pixel1[7:0];


  assign i2s_bclk = bclk & datavalid & send_frame;
  assign i2s_ws = ws;
  assign pclk = !i2s_ws;
  assign i2s_data = pixel_out & datavalid & send_frame;

  assign pixel_out = pixel[counter];


// esp will sample data on posedge of clk, so make changes on negedge
always_ff @( negedge bclk ) begin

  // manage counter and generate pclk
  if (counter == 5'd0) begin
    // pclk <= 1;
    ws <= 0;
  end

  counter <= counter - 1;


  if (counter == 5'd16) begin
    ws <= 1;
  end


  if (init) begin
     counter <= 5'd0;
     // pclk <= 0;
    init <= 0;
    ws <= 0;
  end


end


// end of frame
always_ff @( posedge v_sync ) begin
 if (cts) begin
   send_frame <= 1;
 end else begin
    send_frame <= 0;
 end

end

```

Figure 75. Verilog Code for I2S transmission

10.3 Power

10.3.1 Balance

```
case 2:{ //Measurement Stage

  current_ref = 0; //Stop charging during measurement

  if (rest_timer_rest < 30) { // Rest battery to measure OCV

    next_state = 2; //Stay here until 30s

    rest_timer_rest ++;}

  else{ //Actual OCV measure

    digitalWrite(5, true); //Turn on relay on battery 1

    digitalWrite(4, true); //Turn on relay on battery 2

    if(V_measure_1 < V_max){

      digitalWrite(5, false); //Turn off relay if below 3.4V

      next_state = 1;} //Keep charging

    else{

      digitalWrite(5, true); //Keep relay on if fully charged

      Serial.println("Battery 1 finished!");}

    if(V_measure_2 < V_max){

      digitalWrite(4, false);

      next_state = 1;}

    else{

      digitalWrite(4, true);

      Serial.println("Battery 2 finished!");}


  if(V_measure_1 >= V_max || V_measure_2 >= V_max){

  //Stop charging for all batteries if anyone fully charged

    next_state = 3;}

  rest_timer_rest = 0;}
```

Figure 76 Code for Parallel Balance

10.3.2 Charging Methods

```
//Here below are in FAST LOOP (1kHZ), PID parameters are used as the same from Power Lab

if (voltage_ref == 0){

    error_amps = (current_ref - current_measure) / 1000;

    pwm_out = pidi(error_amps);

    pwm_out = saturation(pwm_out, 0.99, 0.01);

    analogWrite(6, (int)(255 - pwm_out * 255));}

else{//PID for voltage controller cascaded with current controller

    error_volts = voltage_ref - V_Bat;

    current_volts = pidv(error_volts);
```

```
     current_volts = saturation(current_volts, 0.25, 0);

     //Voltage controller output as current controller input

    error_amps = current_volts - (current_measure/(1000.0));

    pwm_out = pidi(error_amps);

    pwm_out = saturation(pwm_out, 0.99, 0.01);

    analogWrite(6, (int)(255 - pwm_out * 255));}

//Here below are in SLOW LOOP (1Hz)

case 2:{ //Constant voltage (CV) charging state (3.6V)

    voltage_ref = 3500;

    if (current > 50){//Keep charging for current above 50mA

        next_state = 2;

        digitalWrite(8,true);} //LED on

    else{

        next_state = 3;

        digitalWrite(8,false);}

     if(input_switch == 0){

        next_state = 0;

        digitalWrite(8,false);}

    break;}
```

Figure 77 Code for CC/CV Charging

10.3.3 Charging

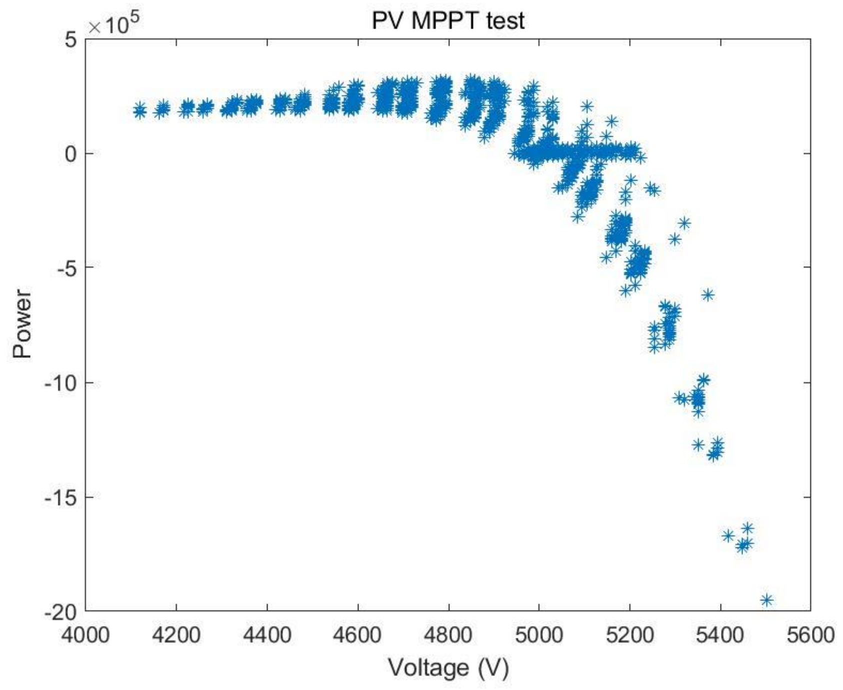


Figure 78 Plot for Charging MPPT

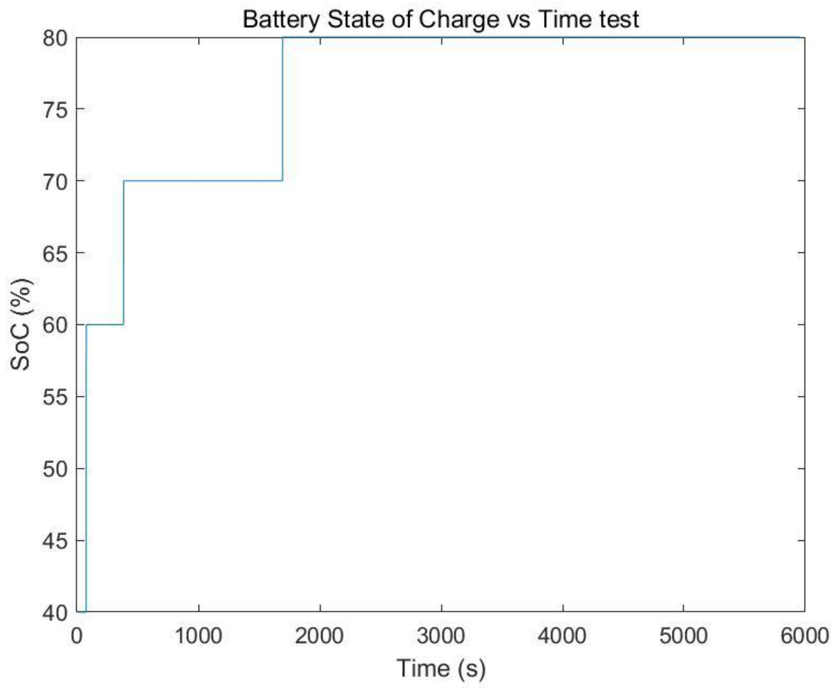


Figure 79 Plot for Battery 1 SoC

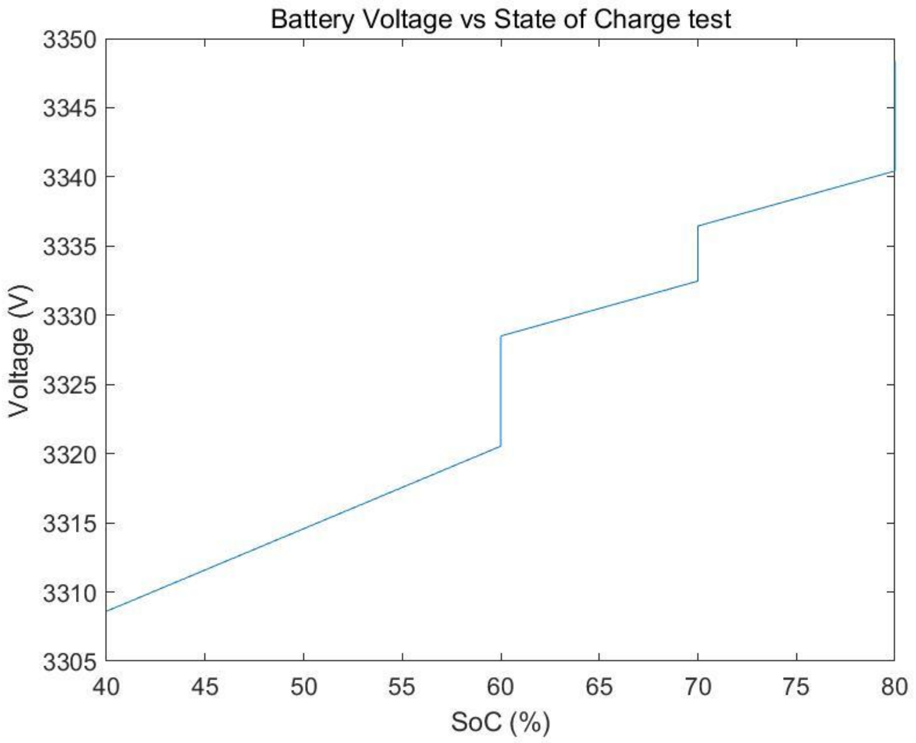


Figure 80 Plot for Battery 1 SoC

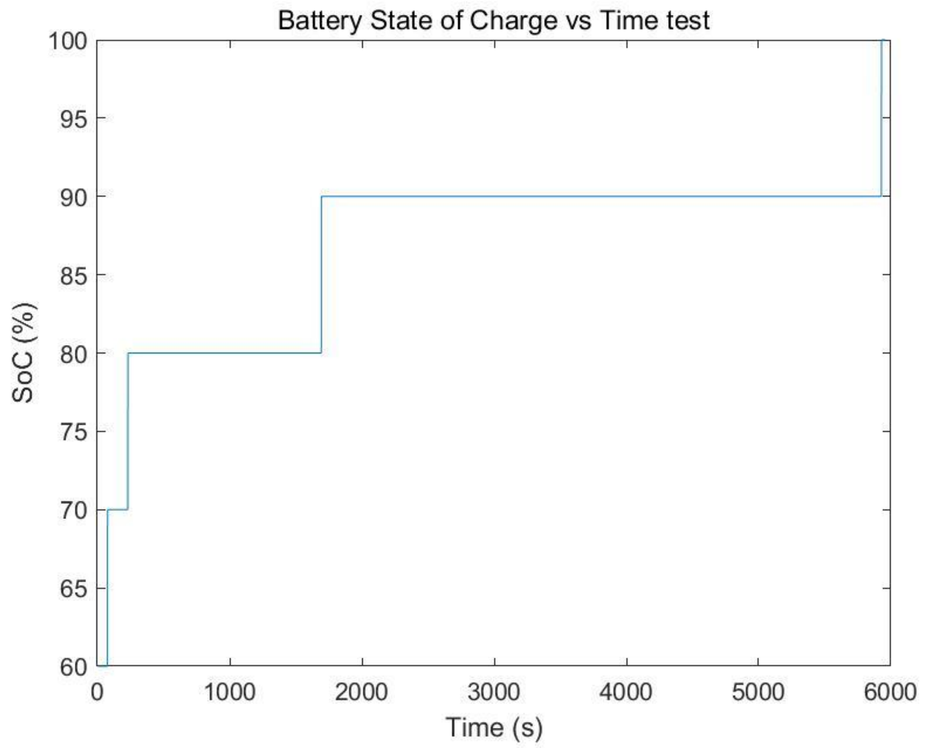


Figure 81 Plot for Battery 2 SoC

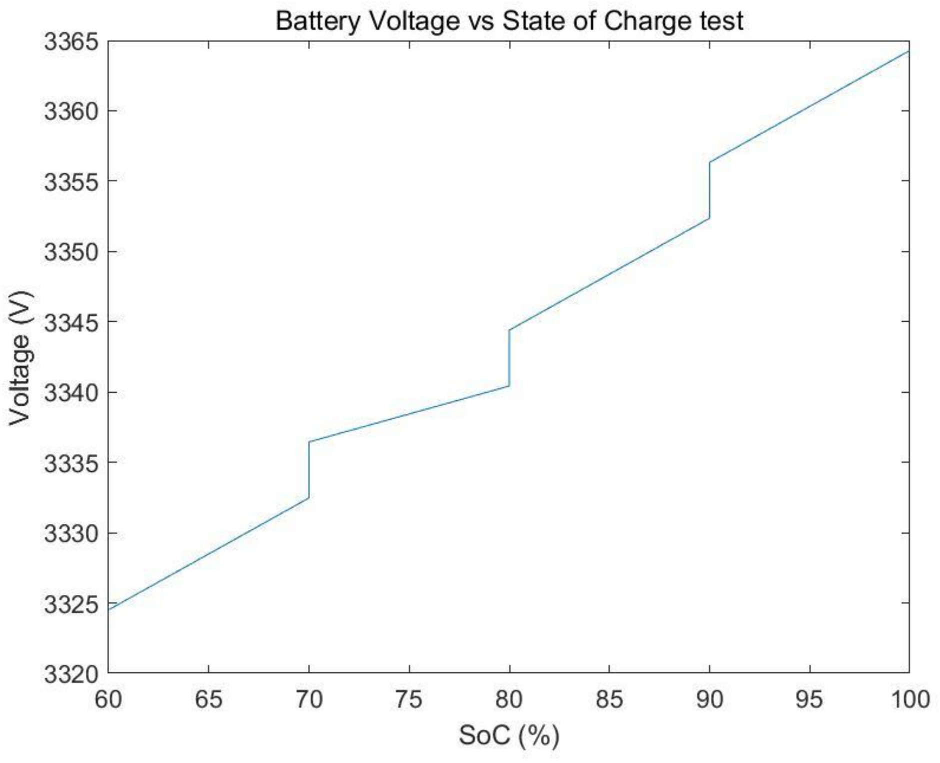


Figure 82 Plot for Battery 2 SoC

10.3.4 Discharging

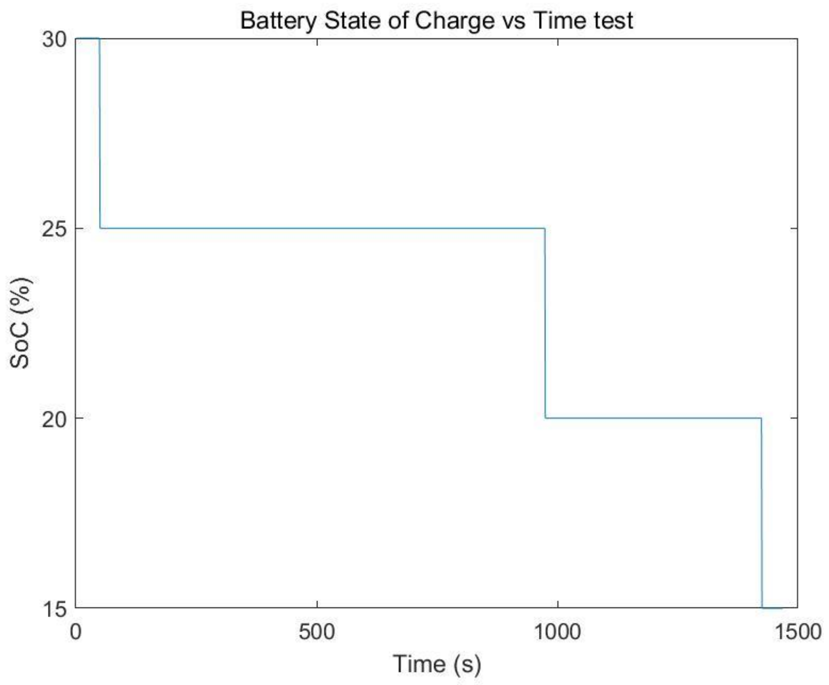


Figure 83 Plot for Battery 1 SoC

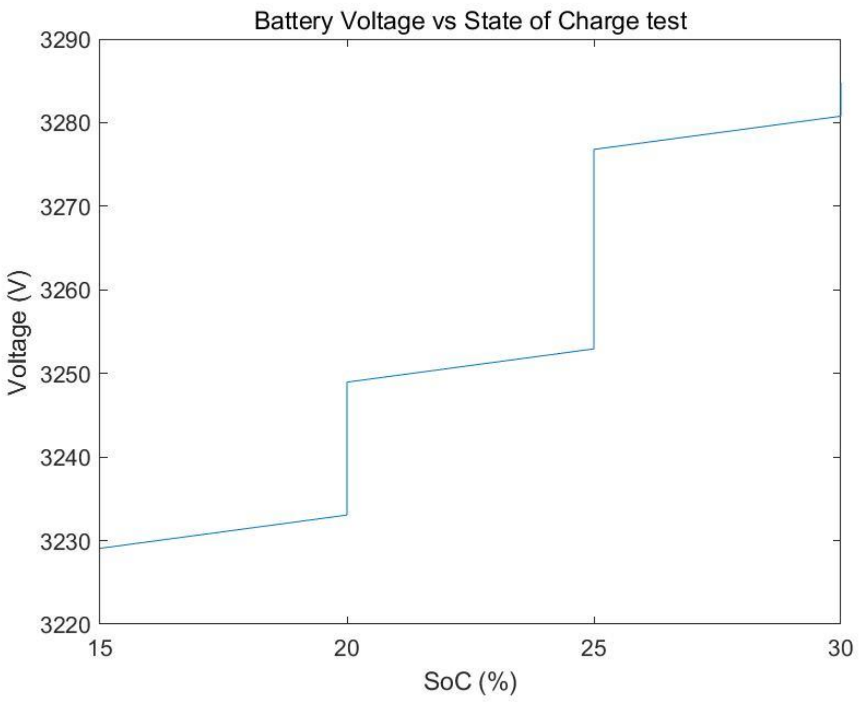


Figure 84 Plot for Battery 1 SoC

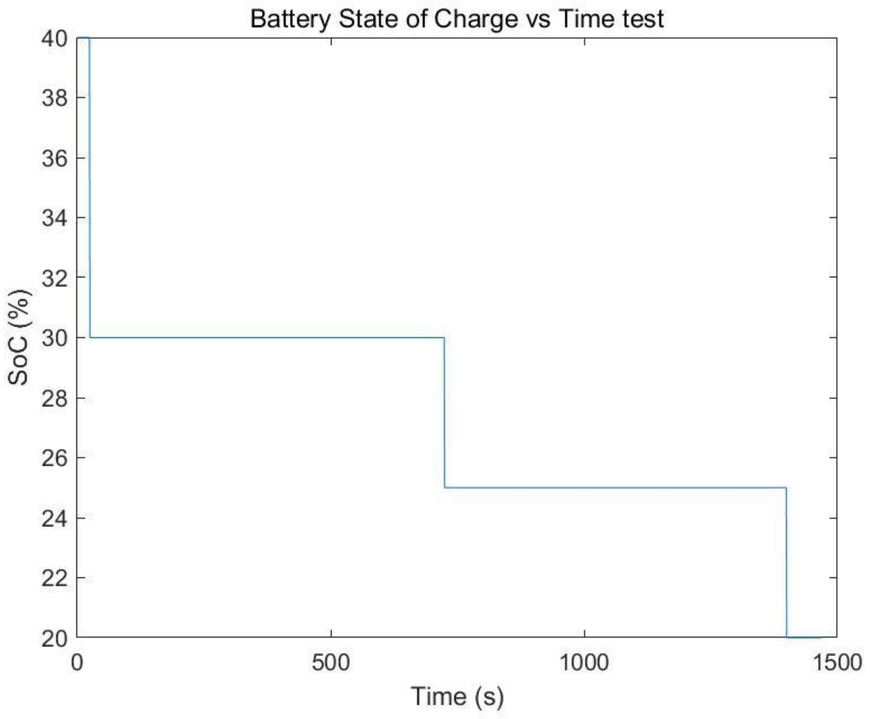


Figure 85 Plot for Battery 2 SoC

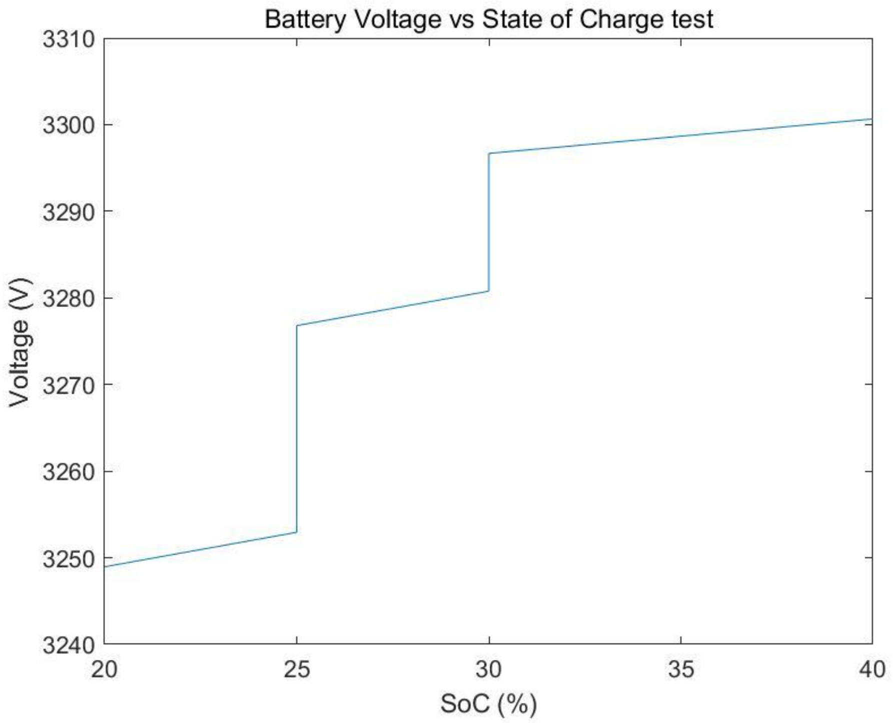
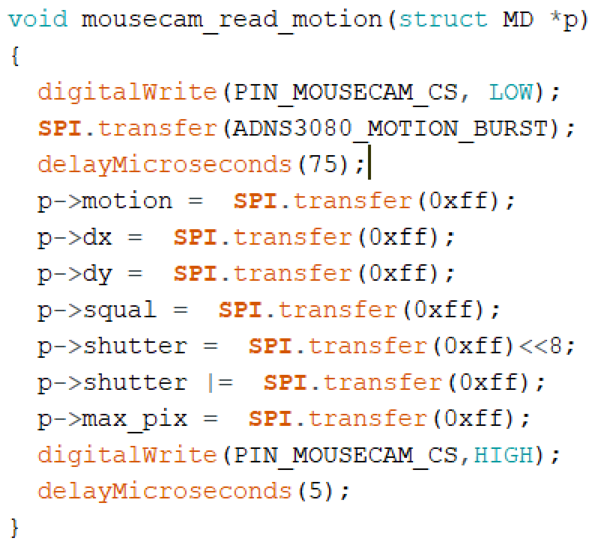


Figure 86 Plot for Battery 2 SoC

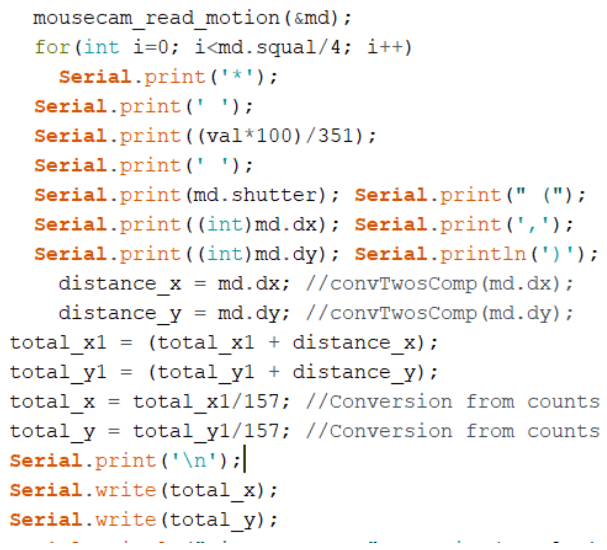
10.4 Drive

10.4.1 Arduino nano codes for communication with optical flow sensor



```
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 88 optical flow sensor communication part1

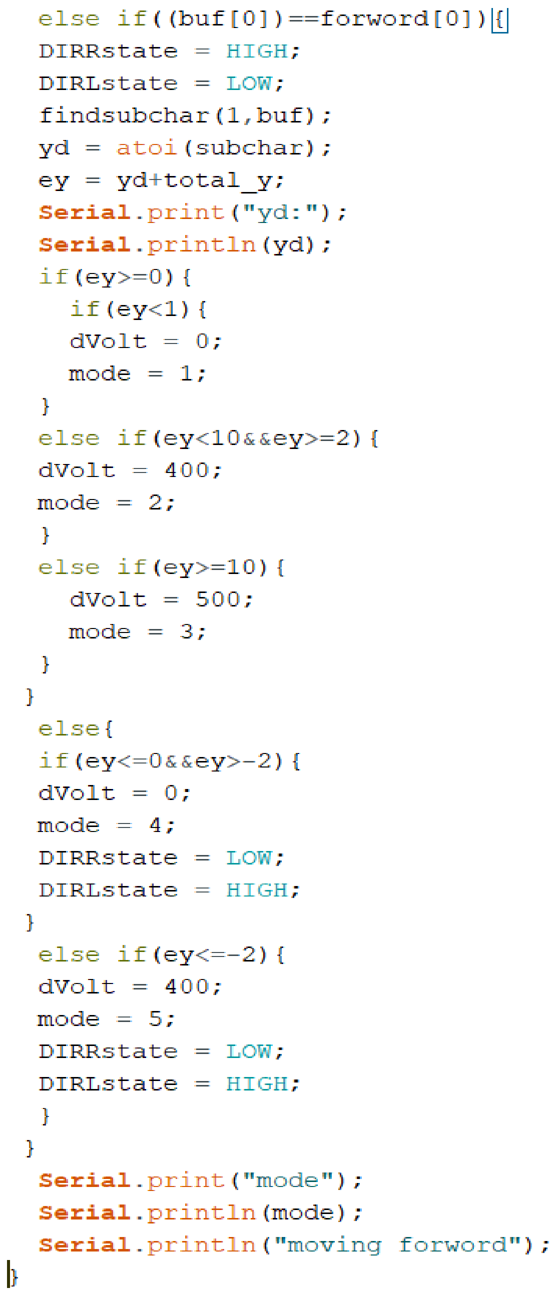


```
  mousecam_read_motion(&md);
  for(int i=0; i<md.squal/4; i++)
    Serial.print('*');
  Serial.print(' ');
  Serial.print((val*100)/351);
  Serial.print(' ');
  Serial.print(md.shutter); Serial.print(" (");
  Serial.print((int)md.dx); Serial.print(',');
  Serial.print((int)md.dy); Serial.println(')');
    distance_x = md.dx; //convTwosComp(md.dx);
    distance_y = md.dy; //convTwosComp(md.dy);
total_x1 = (total_x1 + distance_x);
total_y1 = (total_y1 + distance_y);
total_x = total_x1/157; //Conversion from counts
total_y = total_y1/157; //Conversion from counts
Serial.print('\n');
Serial.write(total_x);
Serial.write(total_y);
```

Figure 89 optical flow sensor communication part2

Figure 87 codes of forward moving

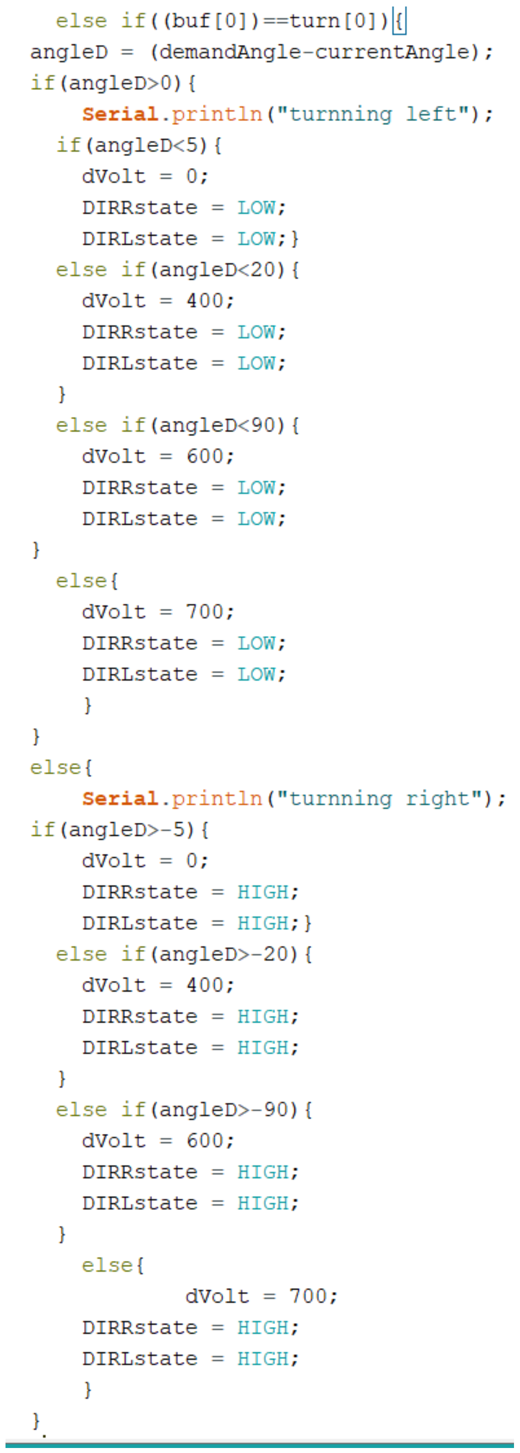
10.4.2 Arduino nano codes for moving forward method



```
    else if((buf[0])==forword[0]){
    DIRRstate = HIGH;
    DIRLstate = LOW;
    findsubchar(1,buf);
    yd = atoi(subchar);
    ey = yd+total_y;
    Serial.print("yd:");
    Serial.println(yd);
    if(ey>=0){
      if(ey<1){
      dVolt = 0;
      mode = 1;
    }
    else if(ey<10&&ey>=2){
    dVolt = 400;
    mode = 2;
    }
    else if(ey>=10){
      dVolt = 500;
      mode = 3;
    }
   }
    else{
    if(ey<=0&&ey>-2){
    dVolt = 0;
    mode = 4;
    DIRRstate = LOW;
    DIRLstate = HIGH;
   }
    else if(ey<=-2){
    dVolt = 400;
    mode = 5;
    DIRRstate = LOW;
    DIRLstate = HIGH;
    }
   }
    Serial.print("mode");
    Serial.println(mode);
    Serial.println("moving forword");
}
```

Figure 90 codes of forward moving

10.4.3 Arduino nano codes for angle control



```
  else if((buf[0])==turn[0]){
angleD = (demandAngle-currentAngle);
if(angleD>0){
    Serial.println("turnning left");
  if(angleD<5){
    dVolt = 0;
    DIRRstate = LOW;
    DIRLstate = LOW;}
  else if(angleD<20){
    dVolt = 400;
    DIRRstate = LOW;
    DIRLstate = LOW;
  }
  else if(angleD<90){
    dVolt = 600;
    DIRRstate = LOW;
    DIRLstate = LOW;
}
  else{
    dVolt = 700;
    DIRRstate = LOW;
    DIRLstate = LOW;
    }
}
else{
    Serial.println("turnning right");
if(angleD>-5){
    dVolt = 0;
    DIRRstate = HIGH;
    DIRLstate = HIGH;}
  else if(angleD>-20){
    dVolt = 400;
    DIRRstate = HIGH;
    DIRLstate = HIGH;
  }
  else if(angleD>-90){
    dVolt = 600;
    DIRRstate = HIGH;
    DIRLstate = HIGH;
  }
    else{
            dVolt = 700;
    DIRRstate = HIGH;
    DIRLstate = HIGH;
    }
}
```

Figure 91 codes of angle control