Technical Report – Team 4

Team 4
Abdelrahman Moataz Noureldin Ali
Omar Hesham Ahmed Zaitoun
Ibraheem Osama
Youssef Tarek Hussein
John Ayman Ibrahim Gaied

Table of Contents

1. Introduction

The Aquaphoton 2024 Mega Project objective is to simulate the electrical systems at work in an ROV, including:

- Internal Electrical PCB Systems
- Control Station GUI
- Computer Vision

The Mega Project main requirement is to design and harmoniously implement the above 3 systems on a small car. It is a test of the technical and cooperative skills of the hardware, firmware and software sub teams.

This report aims to show the culmination of the hard work done by the 5 ambitious team members of team 4, in their journey to succeed in the Aquaphoton trials.

Division of labour:

1. **Hardware:** Abdelrahman Moataz, Ibraheem Osama, Omar Zaitoun
2. **Firmware:** Abdlerahman Moataz
3. **GUI:** John Ayman
4. **Computer Vision**: Youssef Tarek

2. Electrical System

I. Hardware

Microcontroller:

The team decided on using the ESP8266 microcontroller, specifically the ESP8266 12E chip. Despite its cheap price, the advantages are numerous:

- Built-in Wi-Fi module
- PCB Antenna
- 4MB Flash memory
- Digital PWM available for all GPIO (except GPIO16)

Figure 1: ESP8266 12E

The reason this microcontroller was chosen was because of its inbuilt Wi-Fi capabilities. A main task of this project was for the microcontroller to be able to wirelessly communicate with GUI's computer, which the Wi-Fi handles seamlessly. Using a single Wi-Fi chip is significantly cheaper than using an ATmega microcontroller with a Bluetooth module.

The ESP8266 unfortunately comes with the following disadvantages:

- Only 11 GPIO that can be used. This includes 2 serial transmission pins (GPIO 1 = TX, GPIO 3 = RX) and 3 boot related pins (GPIO 0, GPIO2, GPIO15), meaning that only half of the 11 can be used without any restrictions.
  NOTE: The restrictions and distribution of each GPIO will be discussed in the firmware section.
- ADC is only connected to 1 pin (A0) and so can only have 1 analogue input.
- Require a 3.3V supply, not many regulators that can provide this voltage are in stock in Alexandria. It is also a low sourcing voltage.
- Digital pins are rated at 3.3V, 12mA. Analog pin can only handle voltage in 0 to 1V range.

These obstacles however, were cleared using various techniques that will be showcased in the PCB design section.

The hardware team decided on designing 2 PCBS:

1. ESP8266 microcontroller bare circuit
2. Main circuit

The main circuit consists of the main power source (12V), all the sensors, indicator LEDS, RGB LED for speed indication, and finally the motor driver IC L293d.

ESP8266 Microcontroller Bare Circuit:

Figure 2 shows the schematic of the microcontroller bare circuit:

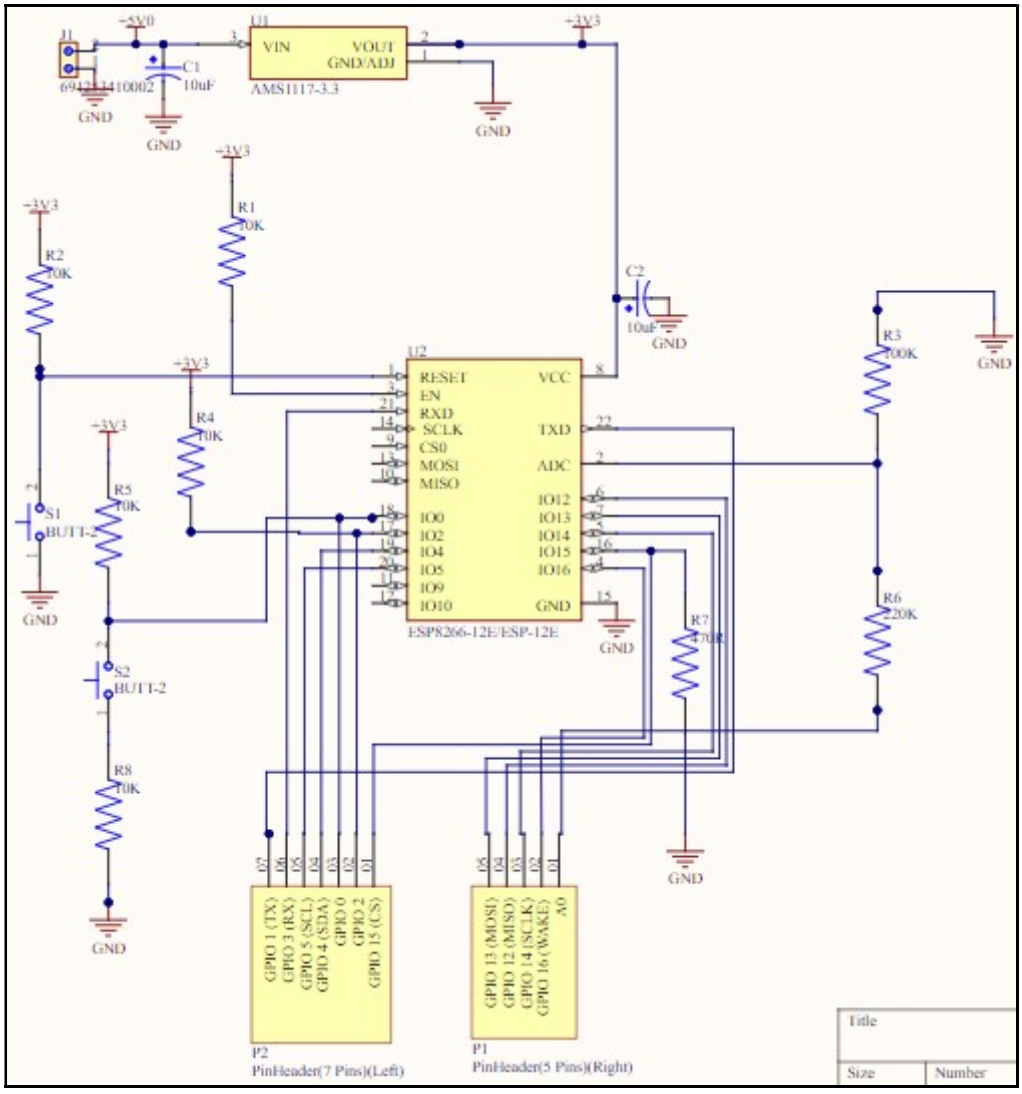

Figure 2: ESP8266 Bare Circuit Schematic

A 5V source powers up the 3.3V LDO regulator AMS1117. Decoupling capacitors of 10 uF are used as seen in the typical configuration of its datasheet

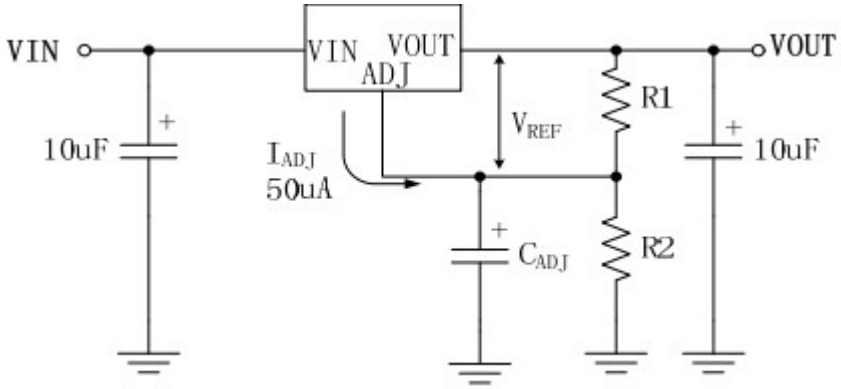

Figure 3: The AMS1117 circuit in datasheet

The ESP has a built in indicator LED and so an external one is not required to indicate power is supplied.

The following is done to make the chip work:

1. EN pin is pulled high to enable the chip
2. To prevent random reset of controller, the RESET pin is pulled high and is connected to a GND switch.
3. GPIO0 and GPIO2 are pulled high (at boot) to select boot mode “Boot sketch” as seen in Figure 3. This mode executes the program stored in the chip memory.
4. GPIO15 is pulled low to turn off SD mode.
5. GPIO0 is connected to a GND switch which is required when programming the chip (reference Figure 3).

GPIO15	GPIO0	GPIO2	Mode
0V	0V	3.3V	Uart Bootloader
0V	3.3V	3.3V	Boot sketch (SPI flash)
3.3V	x	x	SDIO mode (not used for Arduino)

Figure 4: Boot Modes of ESP8266

The ADC pin is connected to a 220k/100k voltage divider, numbers used from the nodeMCU development board schematic. This increases the input range of the ADC from 0 – 1V to 0 – 3.3V, which will be needed for the current and voltage sensors.

The GPIOs are then connected to pinheaders which will be used to connect to the other PCB. The seemingly arbitrary order of the GPIO labels on the pinheaders is used to simplify the actual PCB design.

Figure 5 shows the PCB of the above schematic:

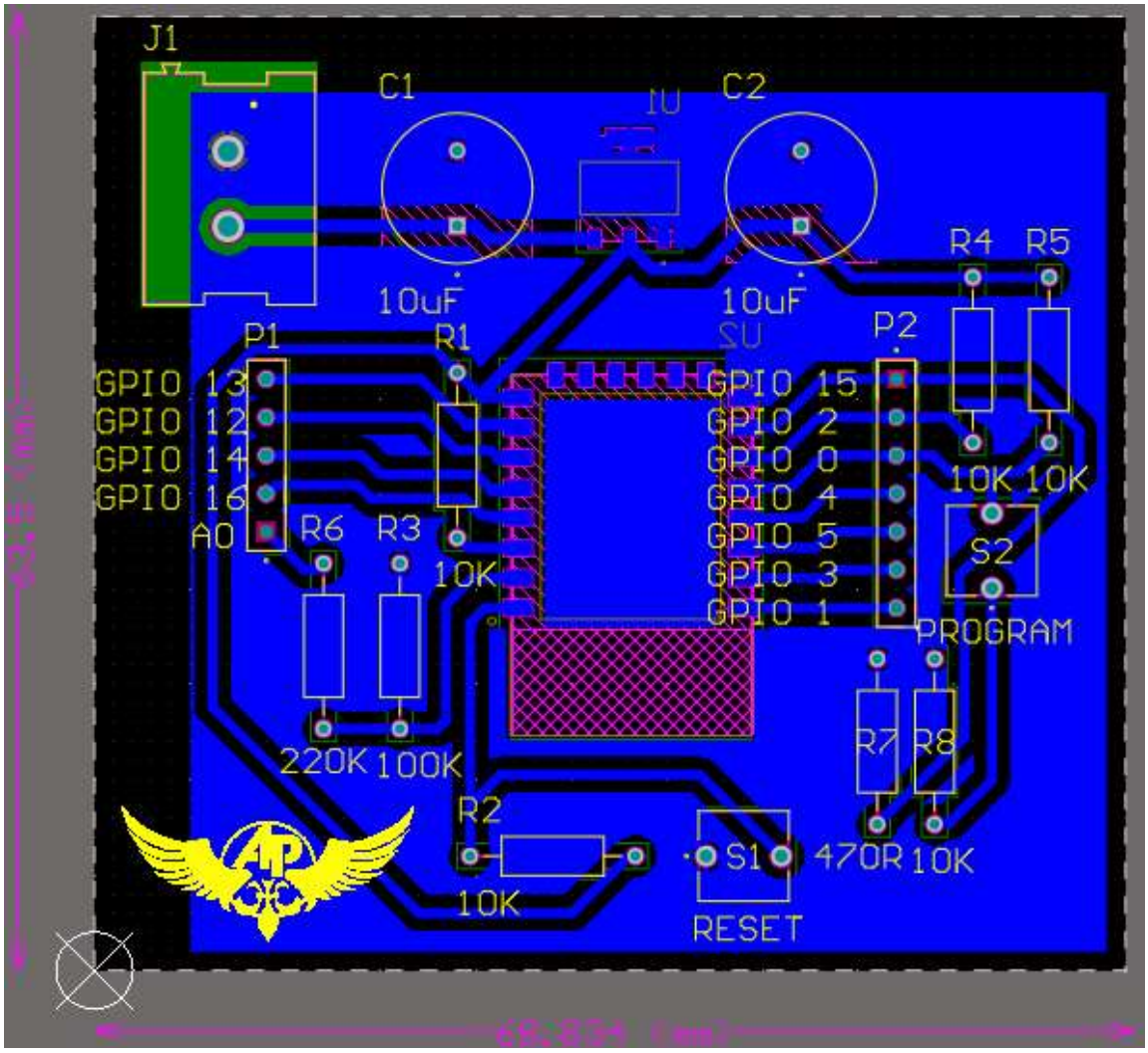

Figure 5: ESP8266 Bare Circuit PCB

The design rules were as follows:

Table 1: Design rules of ESP PCB

Clearance	**Dimension (in mm)**		
Track to Track, Polygon to rest	0.8		
SMD to Track	0.5		
SMD to SMD, THT to rest	1		
Track Width:	0.5	0.8	1.2

The decoupling capacitors were placed as close as possible to the regulator and the microcontroller chip to reduce noise entering each.

Some tracks used the minimum width of 0.5mm to stay within reason of tolerances. Every other track is 1mm thick and, with the help of polygon pour, give a lot of area for heat dissipation.

The final dimensions of the PCB are 69 x 64 mm, with enough space given for the overlay

Main Circuit:

Figure 6 shows the schematic of the main circuit:

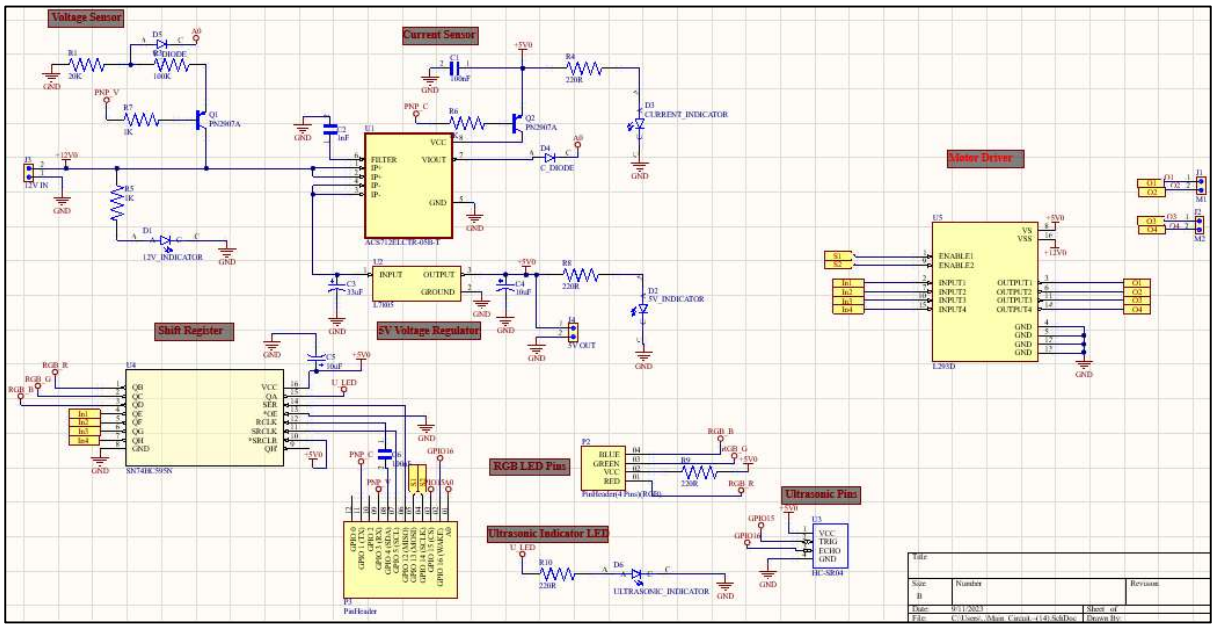

Figure 6: Main Circuit Schematic

Since the main circuit contains everything other than the microcontroller, we will start from left to right studying each section:

Power IN + Voltage Sensor

A 12V battery powers up the whole system through the terminal block.

In parallel to the power supply is a voltage divider circuit 100k/20k which makes sure to not take much current from the main circuit. A 5:1 voltage divider is used so 20% of the voltage reaches the ADC. This is done so the signal reaching the microcontroller is less than the 3.3V maximum. The the maximum voltage that will be received by the ADC is 12/5 = 2.4 V.

As said above, a major problem of using an ESP8266 is the lack of ADC pins. The solution used was to switch the sensor being actively read by the ESP every specific time interval. This is done by turning OFF the sensor not being read while turning ON the other.

This is done by using a PNP transistor (acting as an electronic switch) and a normal diode. When the signal from the microcontroller is HIGH, the transistor will switch off creating an open circuit, preventing the 12V from reaching the sensor.

A transistor acting as an electronic switch can be from the VCC side (PNP) or the GND side (NPN). If an NPN transistor was used, when the sensor is turned off, the voltage divider ceases to work, causing the voltage in signal path to be 12V instead of the 2.4V maximum, breaking the MCU. A PNP transistor is used so that when it is switched off, the voltage divider will be connected to GND and so will not harm the MCU.

The diode is useful in this case because it allows current to only flow in a single direction (from sensor to MCU). This is done so when the sensor is turned off, no current will flow from the other sensor into this one, isolating the two sensors.

The voltage drop due to the diode is about 0.7V to 0.8V (found through testing) is added in the Arduino code to get an accurate reading.

In series to the power supply is a 1k resistor and an indicator LED. The maximum current the LED can handle is 20mA, and the voltage drop is about 2.1V for a red LED. Calculating the current yields:

$$\frac{12V_{in} - 2.1V_{drop}}{1000\ \Omega} = 9.9\ mA$$

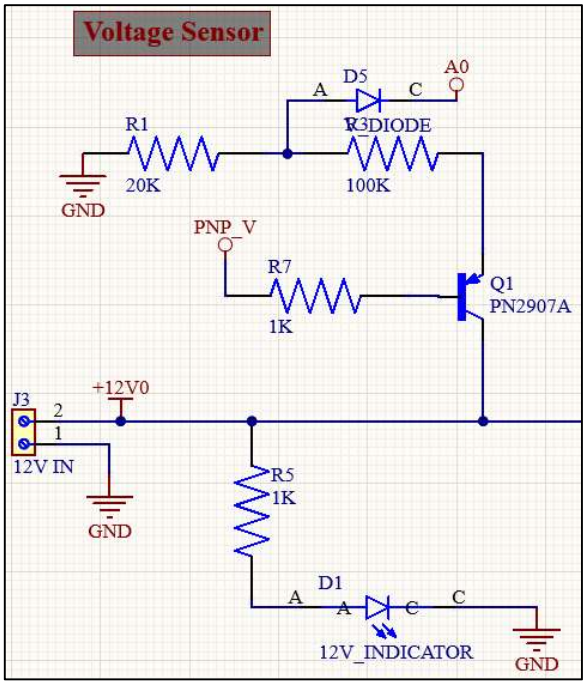

Figure 7: Power supply + voltage sensor

ACS712 Current Sensor

The current sensor circuit is the same as the schematic of the ACS712 module made by SparkFun (Figure 8).

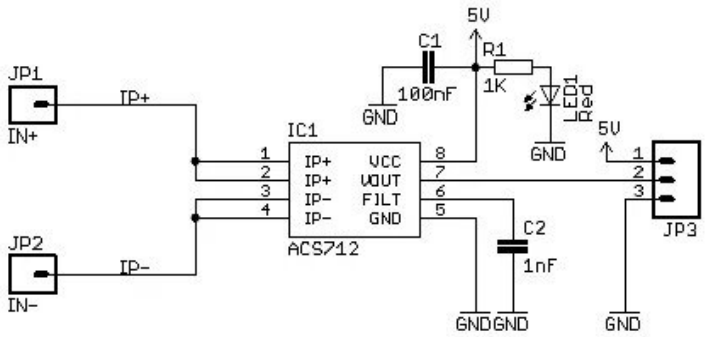

Figure 8: ACS712 module schematic

The 1nF capacitor is used to reduce noise and 100nF capacitor is used for decoupling.

The ACS712 is connected directly in series to the power supply before any other component, to measure total current going to circuit. The 10mA current going to the indicator LED will be added to the current reading through Arduino code. (The current going to the voltage sensor is negligible)

The same transistor / diode circuit discussed for the voltage sensor is used here.

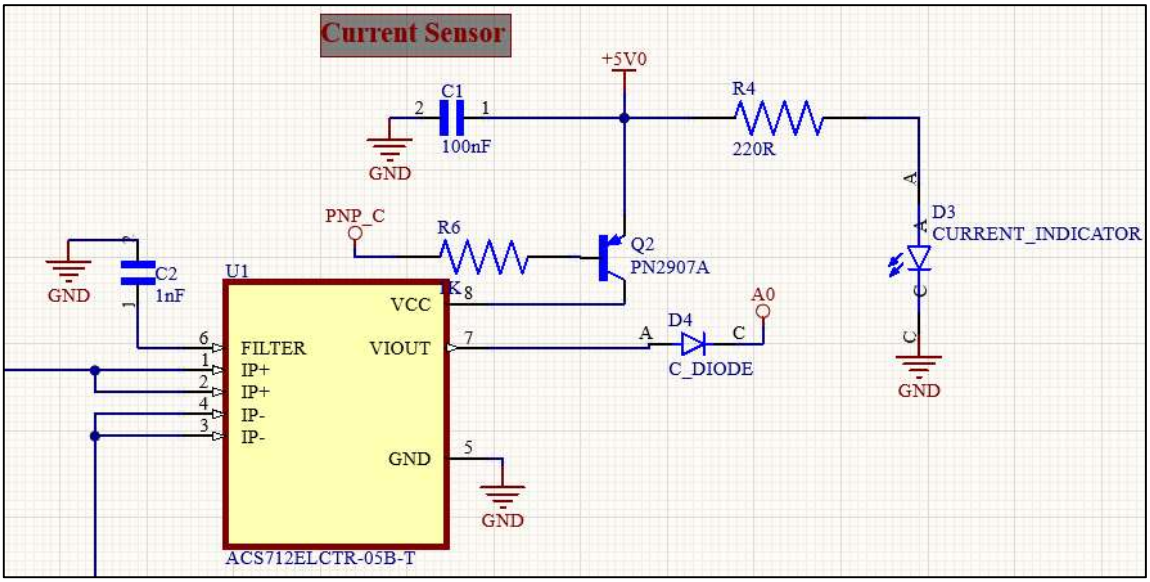

Figure 9: Current sensor

L7805 Voltage Regulator

In series to the current sensor is the L7805 linear regulator which powers:

- Its own indicator LED
- RGB LED
- Current sensor (+ its indicator LED)
- Ultrasonic sensor
- Output terminal block (for the MCU)
- L293d motor driver IC
- 74HC595 shift register

The current pulled from the regulator is as follows (following datasheet values):

Table 2: Current consumption of components powered by L7805

Component	Current(mA)
2x indicator LED (5 – 2.1 / 220)	26.4
RGB LED (when red is on is max current for voltage drop of 2.1)	13.2
ACS712 current sensor (at 5V supply)	10(typ.)
HC-SRO4 ultrasonic sensor	15
ESP8266 microcontroller	70(typ.)
L293d motor driver	35(typ.)
74HC595 shift register (for 6V VCC)	0.008(max.)

The total current is approximately equal to 170 mA (typ.). Figure 10 shows the thermal characteristics of the regulator:

3 Maximum ratings

Table 1. **Absolute maximum ratings**

Symbol	Parameter		Value	Unit
V_I	DC input voltage	for V_O= 5 to 18 V	35	V
		for V_O= 20, 24 V	40	
I_O	Output current		Internally limited	
P_D	Power dissipation		Internally limited	
T_{STG}	Storage temperature range		-65 to 150	°C
T_{OP}	Operating junction temperature range	for L78xxC, L78xxAC	0 to 125	°C
		for L78xxAB	-40 to 125	

Note: Absolute maximum ratings are those values beyond which damage to the device may occur. Functional operation under these condition is not implied.

Table 2. **Thermal data**

Symbol	Parameter	D²PAK	DPAK	TO-220	TO-220FP	Unit
R_{thJC}	Thermal resistance junction-case	3	8	5	5	°C/W
R_{thJA}	Thermal resistance junction-ambient	62.5	100	50	60	°C/W

Figure 10: Thermal characteristics of L7805

The L7805 is of the package TO – 220. The power output of the regulator is:

$$P = IV = 0.170A * 5V = 0.85W$$

$$\therefore T = PR_{\theta JA} = 0.85W * 50\ ℃/W = 42.5\ ℃$$

Which is lower than the maximum operating temperature of 125 °C and so, it doesn't require a heat sink.

Decoupling capacitors were added according to datasheet, but the values used were multiplied by 100 to use electrolytic capacitors.

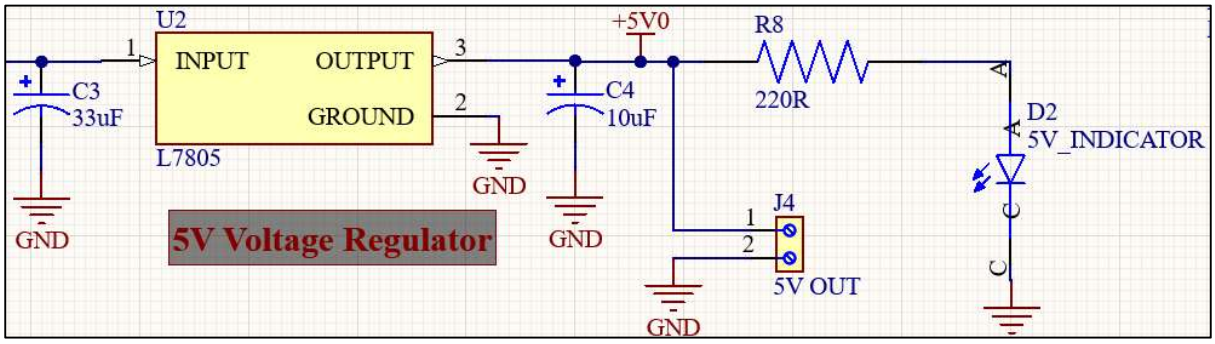

Figure 11: L7805 circuit

Ultrasonic Sensor + RGB LED

Female pinheaders were put in the PCB so soldering won't be done on its own pins. The ultrasonic indicator is an LED that is used in the autonomous mode of the car. In the autonomous mode, if the distance of the car from the wall is the same as that given by GUI, it lights up.

The RGB LED used is common anode, meaning that all LEDs inside share VCC and are controller through their cathode connection. Although the microcontroller cannot output 5V, when the control pins are high the potential difference across the LEDs is only 1.7V. The blue and green LEDs have a forward voltage drop of 2.8V and the red LED has a voltage drop of 2.1V and so the diodes will not turn on. The LEDs are active low, requiring a LOW signal from microcontroller to light up.

Since each LED shares same anode, and only one LED should light up in any scenario, only one resistor is used and it is connected to VCC.

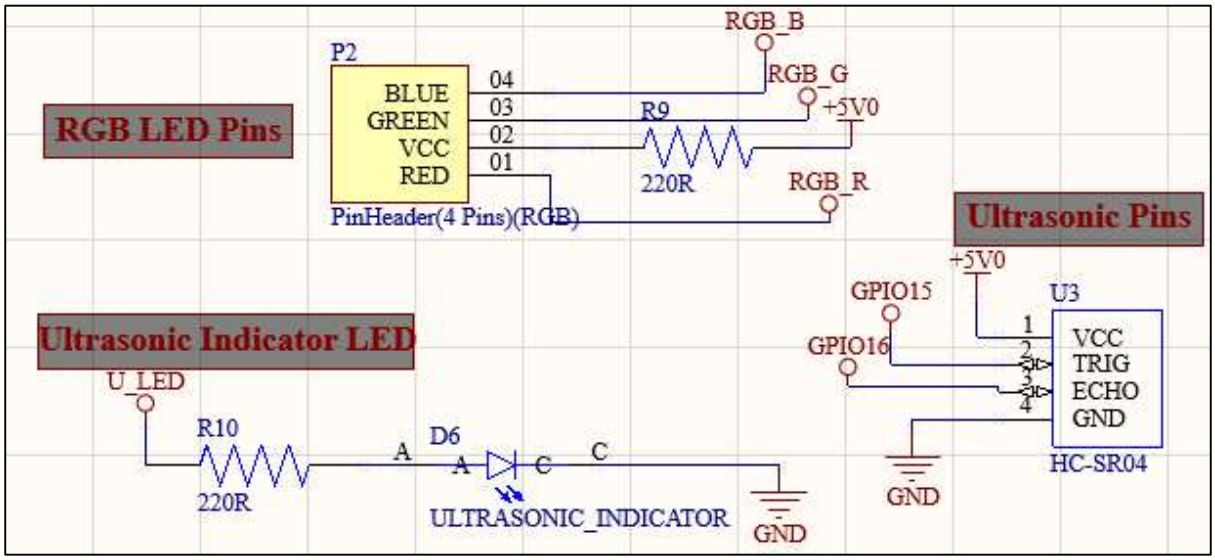

Figure 12: RGB LED and ultrasonic sensor + LED circuits

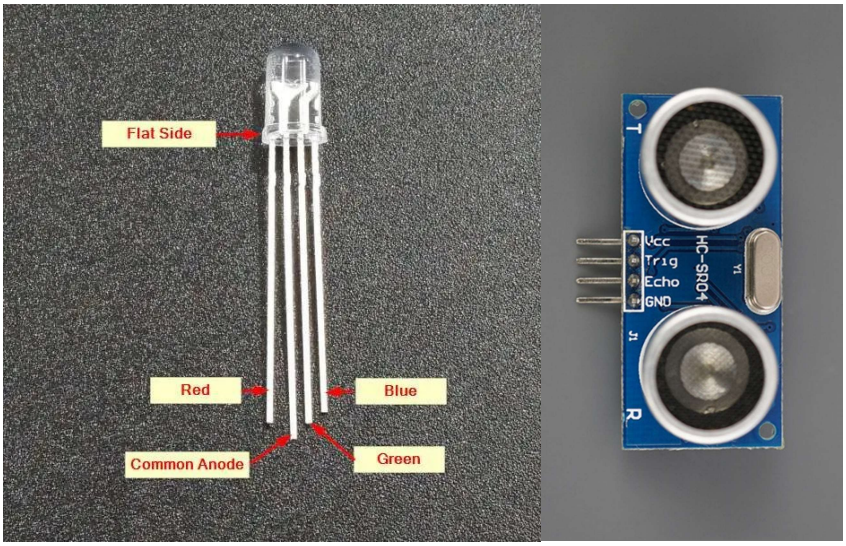

Figure 13: RGB LED and HC - SO4

L293d Motor Driver

The L293d is a dual channel motor driver IC that can output max continuous current of 600mA per channel, and can take supply voltage up to 36V. Each channel has one enable pin and two logic input pins. The truth table for bidirectional control of 1 motor is as follows: (as seen in datasheet)

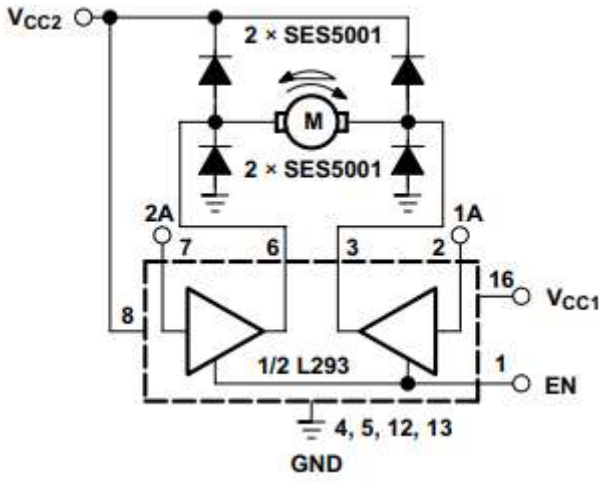

Figure 14: Bidirectional DC motor control

Table 3: Bidirectional DC Motor Control

EN1	IN1(1A)	IN2(2A)	Result
HIGH	LOW	LOW	Fast motor stop
HIGH	LOW	HIGH	Turn right
HIGH	HIGH	LOW	Turn left
HIGH	HIGH	HIGH	Fast motor stop
LOW	X	X	Free-running motor stop

The enable pins can take a PWM signal to control the speed of the motor rotation.

The motor used is a DC gear motor that can take up to 12V and has max current consumption of 250 mA (at stall). Rotation speed (at no load) is approximately 600 rpm.

Maximum junction temperature of the IC is 150°C. The junction to ambient thermal resistance of the IC is 36.4°C/W. If the IC is driving two motors at stall (500 mA) then the temperature of the IC will reach 218.4°C which is much higher than maximum allowable temperature. However, the car is not heavy enough to put the motors at stall and so, won't reach these temperatures.

The output pins of the IC is connected to terminal blocks which will connect to motors using jumper wires.

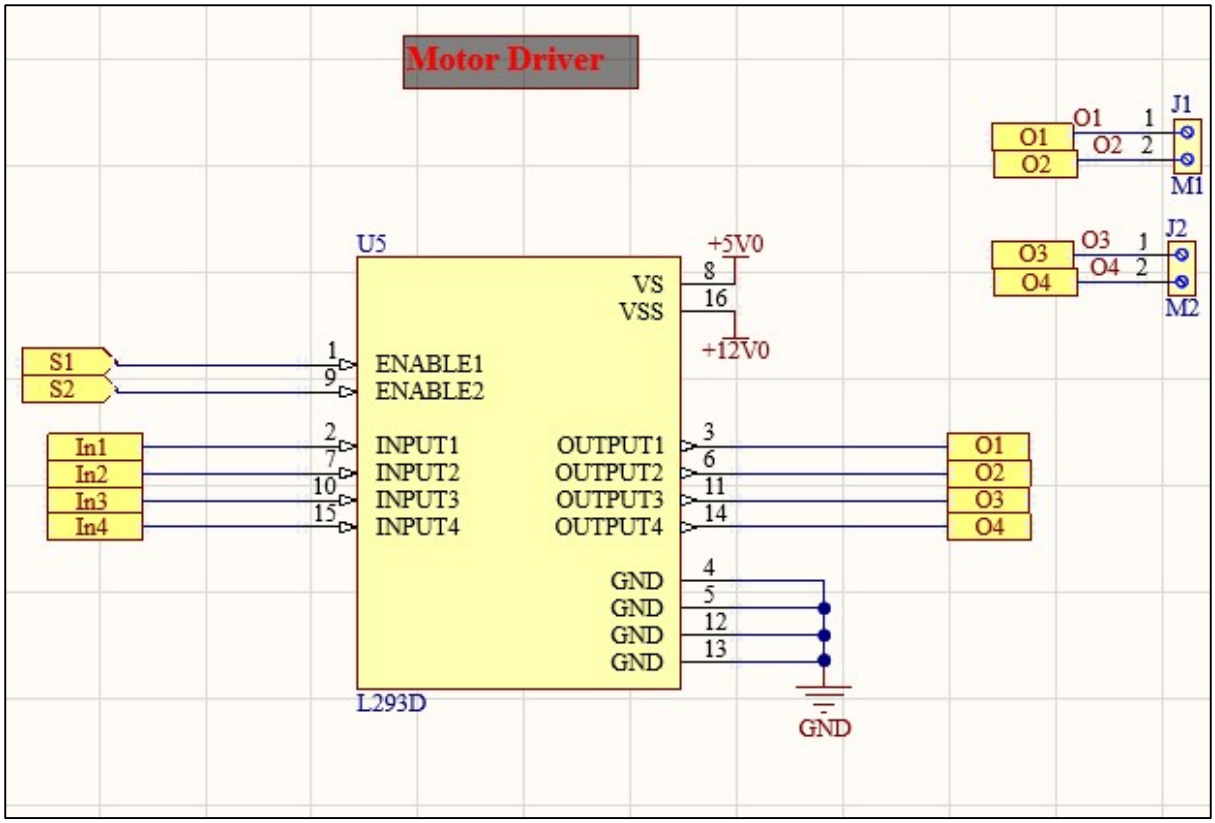

Figure 15: L293d motor driver circuit

74HC595 Shift Register

To end off the main circuit, the final component added is the shift register. The shift register used is the 74HC595 SIPO 8 bit shift register. This means that it takes a serial input and gives 8 parallel lines as output.

The 74HC595 is made up of two 8 bit registers. The first is called the shift register and the second is called the storage or latch register.

The shift register is the one that receives that serial data through the SER (data) pin. Every time a clock pulse is received in the RCLK (clock) pin (during the rising edge), the current state of the SER pin is written into bit 0 of the shift register. Either 1 for SER = HIGH or 0 for SER = LOW. The data does not overwrite what was in bit 0, instead the data is shifted to next bits to make space for new data received. The data in the final bit (bit 7) is discarded every clock pulse (because it was shifted out).

When the SRCLK (latch) pin receives a pulse (specifically the rising edge), the data in the shift register is copied to the storage/latch register. Each bit of the storage register is connected to one of the output pins of the IC, with the LSB connected to QA and MSB connected to QH.

The shift register is the solution to the limited number of pins of the ESP8266, because it effectively works as port expander. It only takes 3 inputs (the data, clock, and latch pins) to get 8 different outputs, and it is one of the fastest methods of port expansion due to the 29MHz frequency of the clock (35.5 ns per bit).

The output pins of the shift register can only provide digital HIGH or LOW (with HIGH being based on the input VCC) and so, cannot be used for PWM functions such as motor speed control.

The VCC pin connected to a decoupling electrolytic capacitor of 10 uF and the latch pin is connected to a ceramic 100nF capacitor to prevent flickering of the latch signal

The *OE pin is connected to GND to enable the IC. The *SRCLK pin resets the data in the latch register when sent a LOW signal. It is connected to VCC to prevent random resets of latch register.

Finally, pin headers are added to connect the main circuit to the MCU PCB.

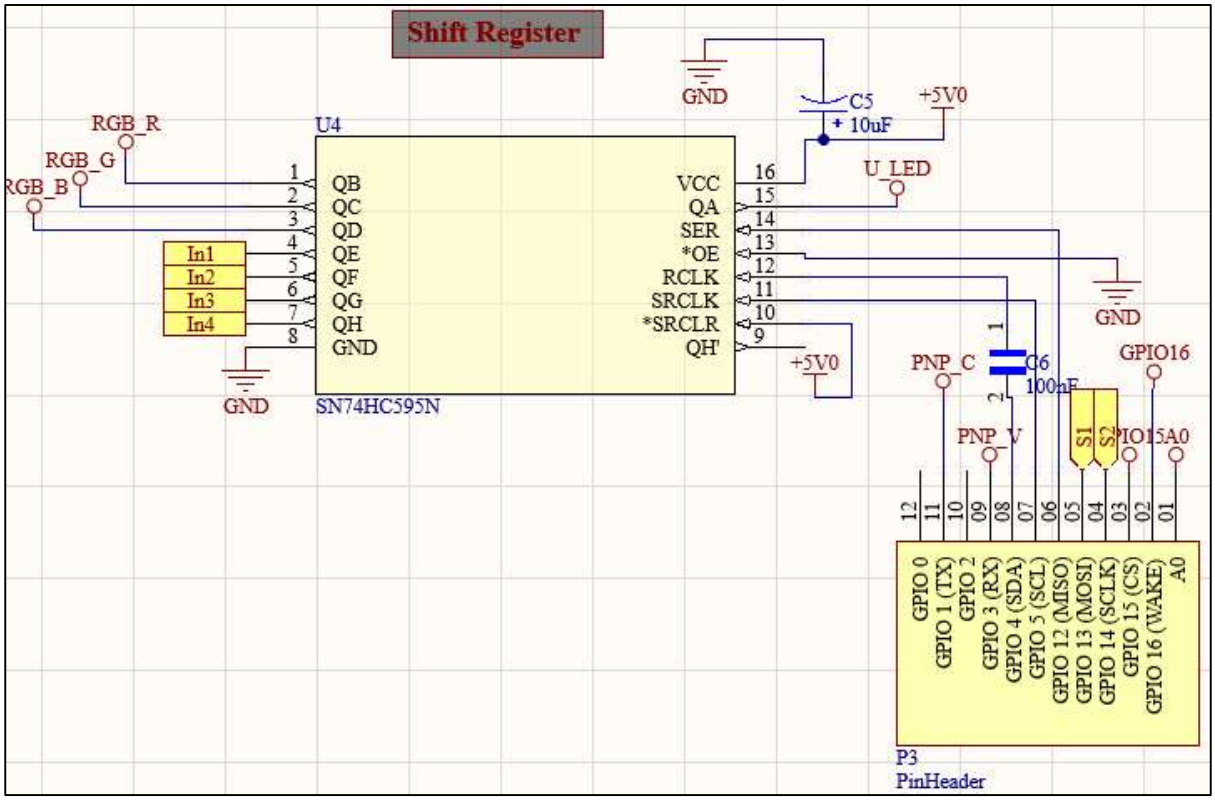

Figure 16: 74HC595 shift register + pinheaders for MCU

Custom Motor Driver (Scrapped)

Figure 17 shows the schematic of the motor driver:

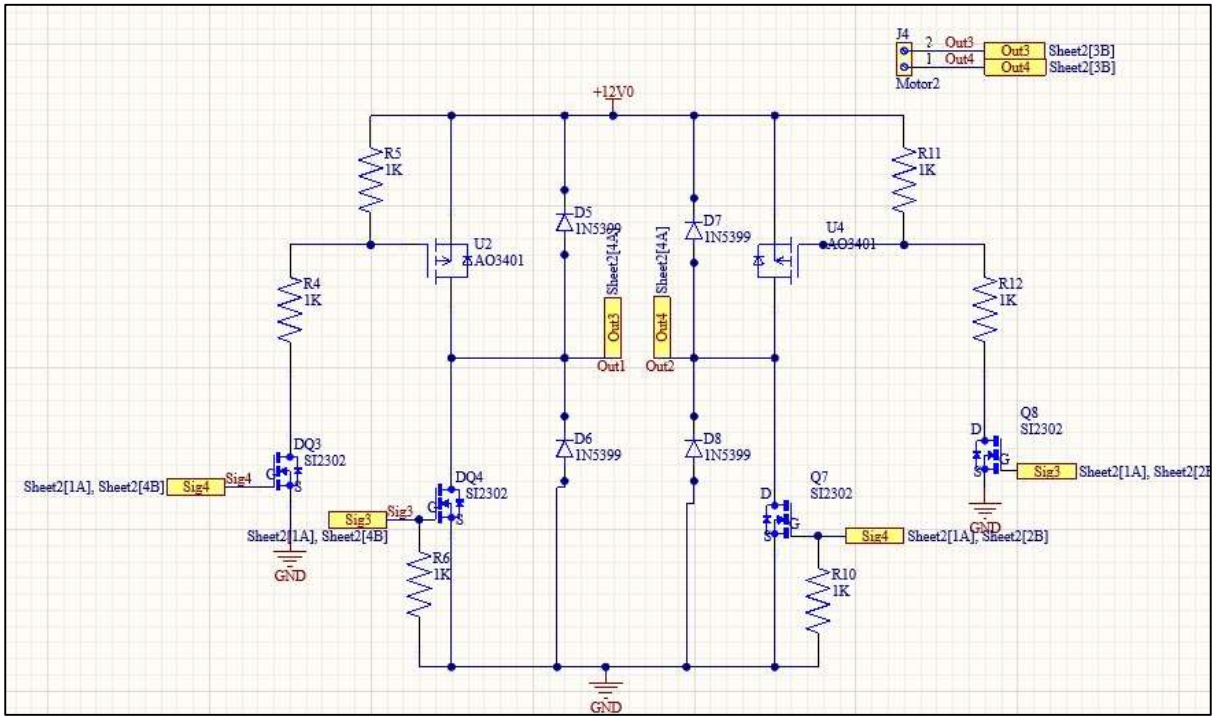

Figure 17: One half of the motor driver

The upper p channel MOSFETs are driven by other n channel MOSFETs which when open, no current passes and so the volt on source is same as gate.

When the n channel is closed a current passes and hence a voltage drop is applied on the resistor making the volt on the p channel gate lower than that of the source and so gate is enabled.

The lower n channel MOSFETs are driven directly by the microcontroller by applying volt on the gate making it have higher volt than the source as it is connected directly to ground.

This driver has 4 PWM controlled pins 2 for each motor. The following is the truth table of the driver:

Table 4: Motor driver truth table for one motor

IN1	IN2	Result
LOW	LOW	Motor turn off
LOW	HIGH	Motor forward
HIGH	LOW	Motor backward
HIGH	HIGH	**SHORT CIRCUIT – TO BE AVOIDED**

Dead band is added in Arduino code when switching between forward and backward motion to avoid the instance where all switches are closed momentarily during switching.

NOTE: It was very difficult for the team to solder the micro SMD MOSFETS and diodes into the PCB. The THT alternatives were unfortunately too expensive to use and so the team decided on using an L293d motor driver IC instead.

Figure 18 shows the final PCB designed for the main circuit:

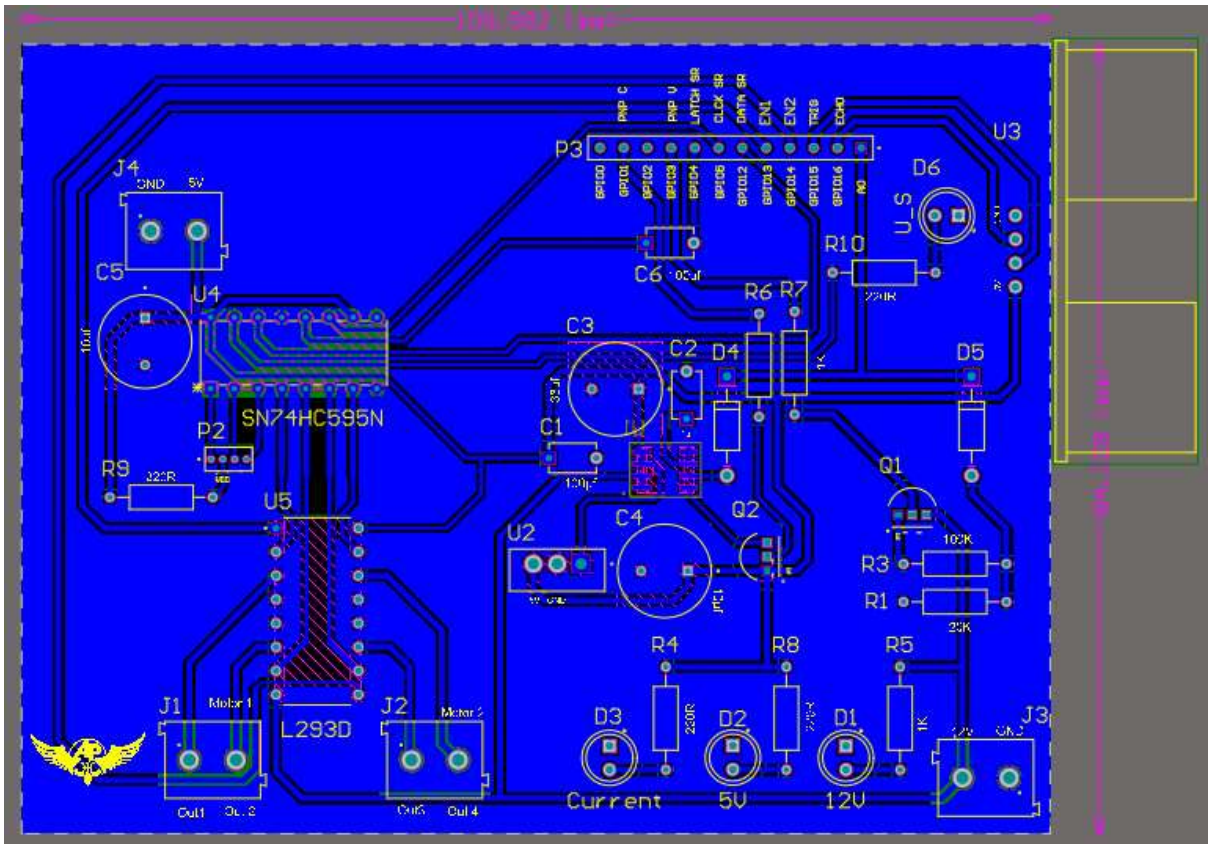

Figure 18: Main circuit PCB

The design rules were as follows:

Table 5: Main circuit PCB design rules

Clearance	**Dimension (in mm)**		
Track to Track, SMD to Track, SMD to SMD	0.6		
THT to rest	0.254		
Copper to Track, Copper to rest	0.6, 0.4		
Track Width:	0.5	0.8	1

Like the previous PCB the decoupling capacitors we put as close as possible to the corresponding IC. The indicators LEDs are lined up neatly and close to the 12V in terminal block. The ultrasonic indicator LED was put close to the ultrasonic sensor. The motor output terminal blocks are put close to the motor driver IC to reduce track length and hence noise going to motors.

II. Firmware

The ESP8266 is compatible with the Arduino IDE that were designed for an Atmega, and can use almost all the same functions. However, the ESP8266 is not compatible with all the available Arduino libraries, requiring some of them to be ported to be useable.

The Arduino code, based on modular programming, was divided into two files:

- The main ESP8266 code
- The ESP8266 Wi-Fi code (in a .h file)

Let's start by taking a look at the Wi-Fi side of things.

Essence of Communication Protocol

Connecting the GUI and ESP

Important terminology to be defined first:

Station: Any device that is connected to Wi-Fi network is called a station

Access Point (AP): A device that provides Wi-Fi connections to a hub of stations is called an AP. Examples include router and MODEM.

The ESP8266 can act as both a station and an AP. However, since it can only act as an AP through wireless connection and not Ethernet, it is called a soft AP. Whenever two stations are connected to Wi-Fi network, they both are also part of a local area network (LAN). Device in a LAN can communicate directly with each other without requiring an active internet connection.

For the GUI and ESP to communicate, it is required for both share a Local Area Network (LAN). This can be done by using both as stations and connect them to the same Wi-Fi network. Better yet, using the ESP as a soft AP means that the computer can directly connect to the ESP's network, cutting the Wi-Fi middle man and making the project completely independent, only requiring the computer and ESP to be within wireless connection range.

Wireless Communication through a Website

Now that the ESP and GUI are in the same LAN, how do they communicate? The main idea to allow communication between the ESP and GUI is to use a webserver (website) as a host for communication. All data is sent to this webserver and can then be read by either the ESP or GUI.

Any device connected to a LAN has its own webserver that can be accessed through its own IP address. The communication will be done through the ESP's webserver where it can write any information there. The GUI, being inside same LAN, can connect to that server to read the data inside the server.

Using an IP address as the URL of the website is cumbersome and not practical, especially because it is not a constant for each device like MAC addresses. The

solution to this problem is the Domain Name System (DNS) which assigns a domain (i.e a website's IP address) a constant domain name. However, normal DNS is not usable in a LAN, which is rectified by using multicast DNS (mDNS).

The mDNS uses domains with the .local suffix. Whenever a device in the LAN requests a domain with .local, it will send a multicast query (i.e send request to all devices in network at same time) to all devices in LAN that support mDNS. The device with the right name will send back another multicast with its own IP address and so, communication is successful.

How HTTP is Utilised as a Channel of Communication

Since communication will be done through a website, writing to (or reading from) the website will be done through HTTP, mainly HTTP GET requests.

HTTP: HyperText Transfer Protocol is a text-based protocol used to communicate with webservers. This is done through HTTP requests, with the two main requests being the GET or POST request.

For example, if you want to access google images through a web browser, the URL used will be https://www.google.com/imghp. Accessing this website requires the web browser to send a GET request so that the webserver returns the wanted data (in this case, google images site). The GET request header will consist of the following:

```
GET /imghp HTTP/1.1
Host: www.google.com
```

And other irrelevant data. The 3 important things here are the request type (GET), the path of wanted request (/imghp) and finally the host of the wanted request (domain name). Using these 3 components is the core of the communication protocol used by the team. Notice how in the first line, a whitespace exists before and after the path. This is important and will be used in the Arduino code.

Now we will see how this information is implemented in the Arduino code.

Server Control

All functions related to Wi-Fi were defined in a separate header file called "ESP8266_Server_Control.h".

All functions are explained in the following flowcharts:

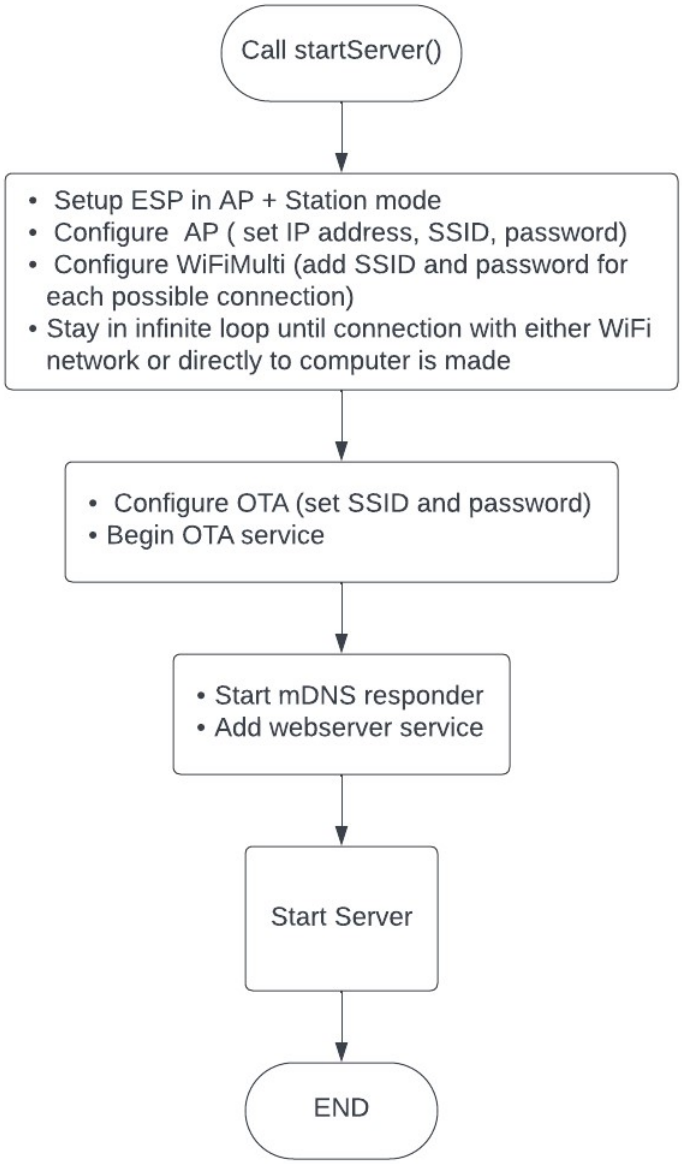

Figure 19: startServer function

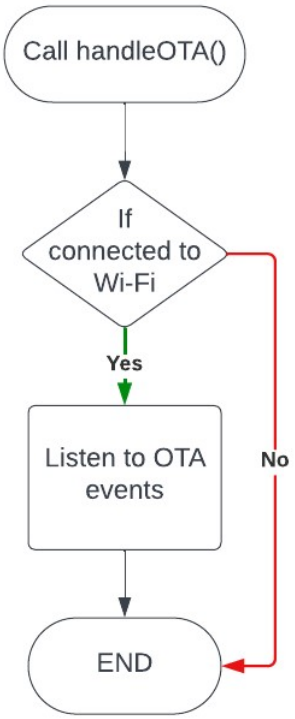

Figure 20: handleOTA function

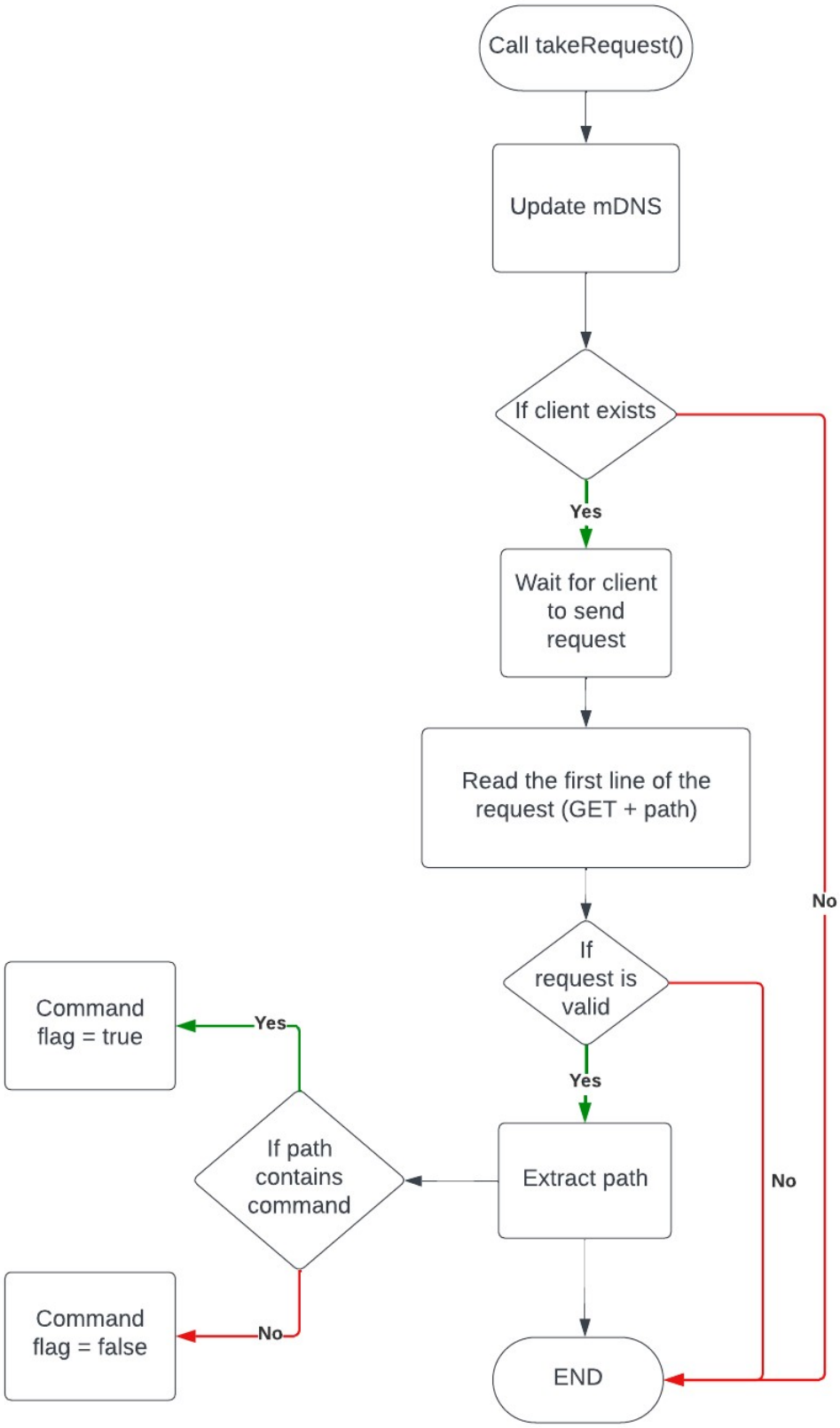

Figure 21: takeRequest function

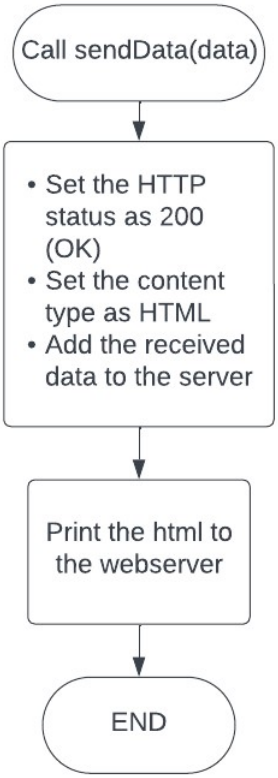

Figure 22: sendData function

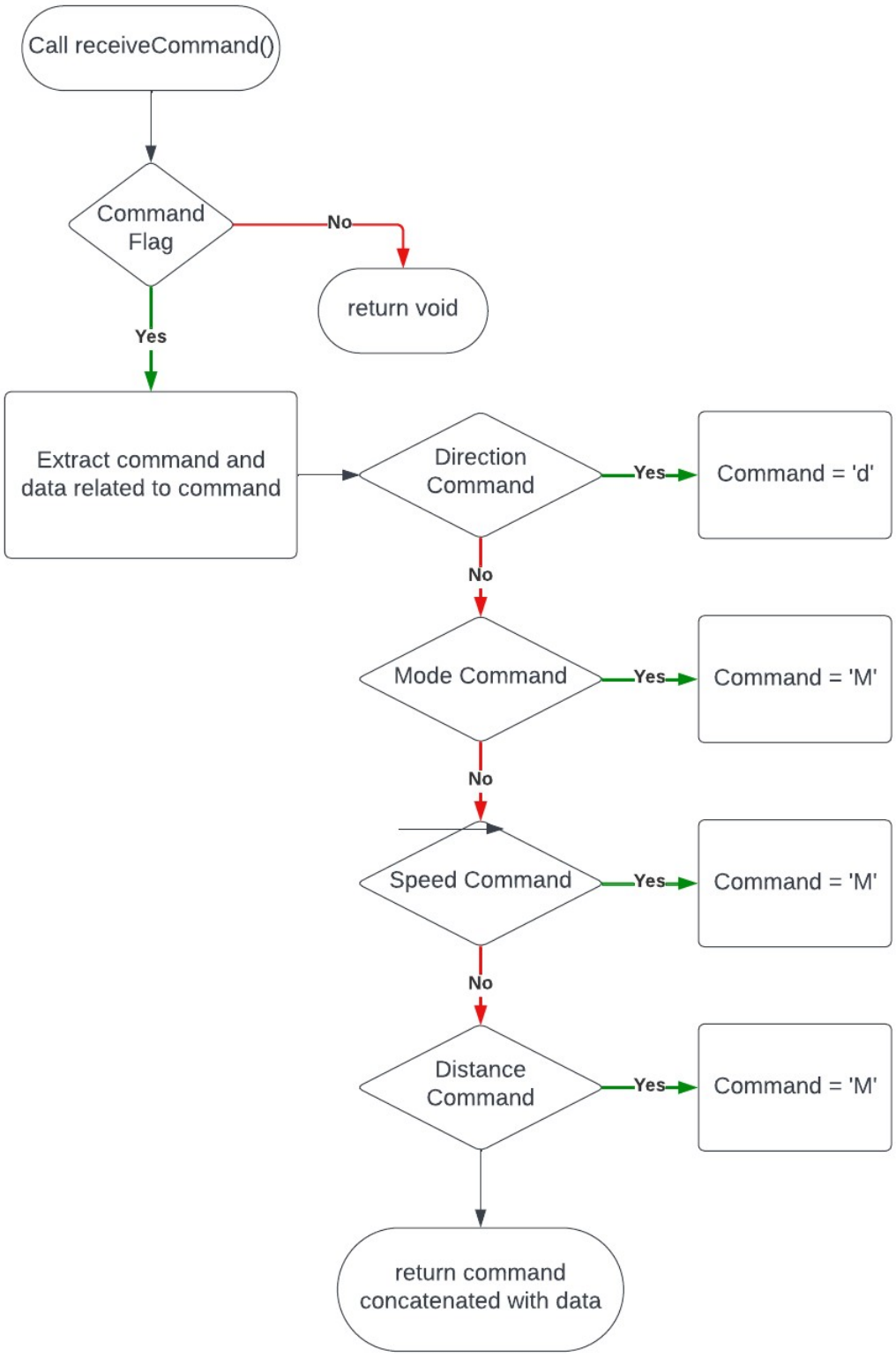

Figure 23: receiveCommand function

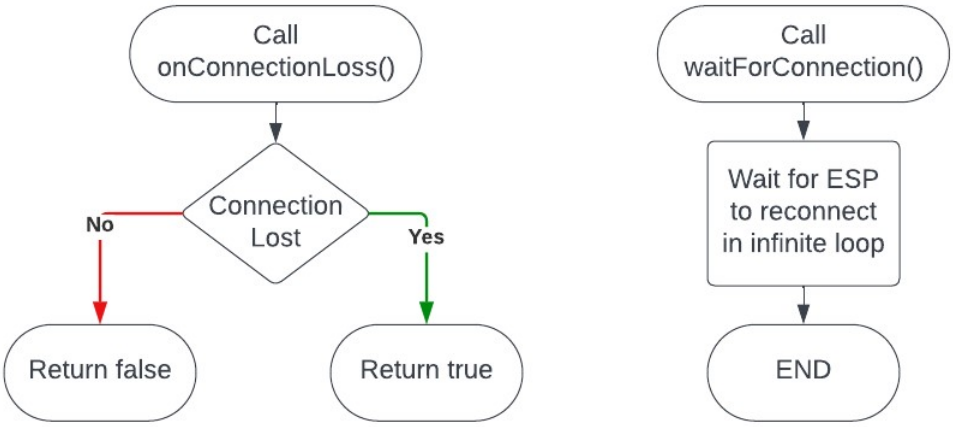

Figure 24: Connection functions

System Control

The following are functions that deal with the whole system. They control the shift register, RGB led, motors, and all sensor readings.

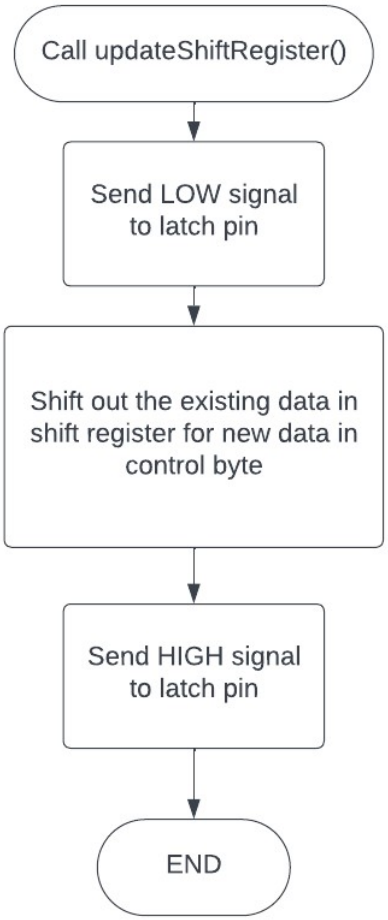

Figure 25: updateShiftRegister function

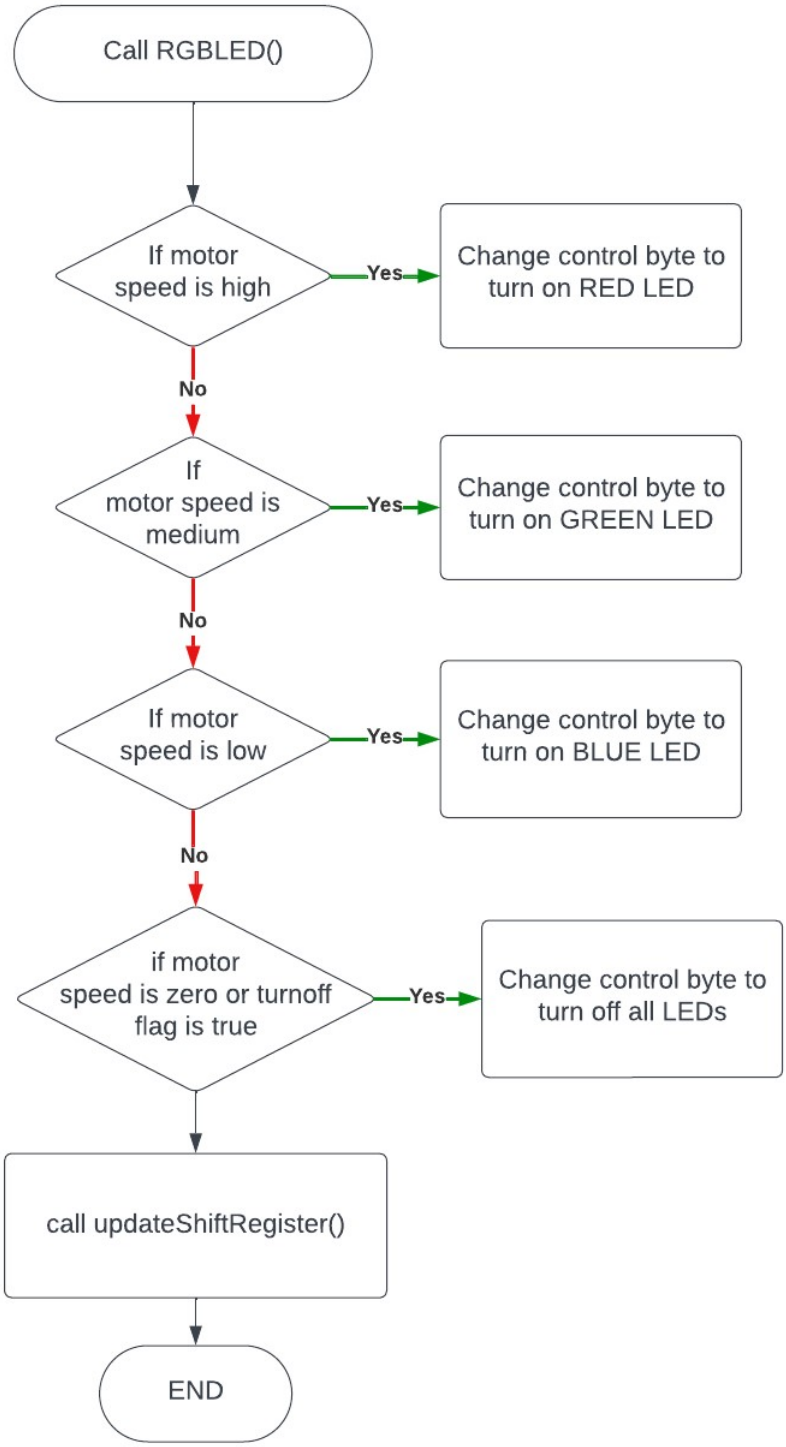

Figure 26: RGBLED function

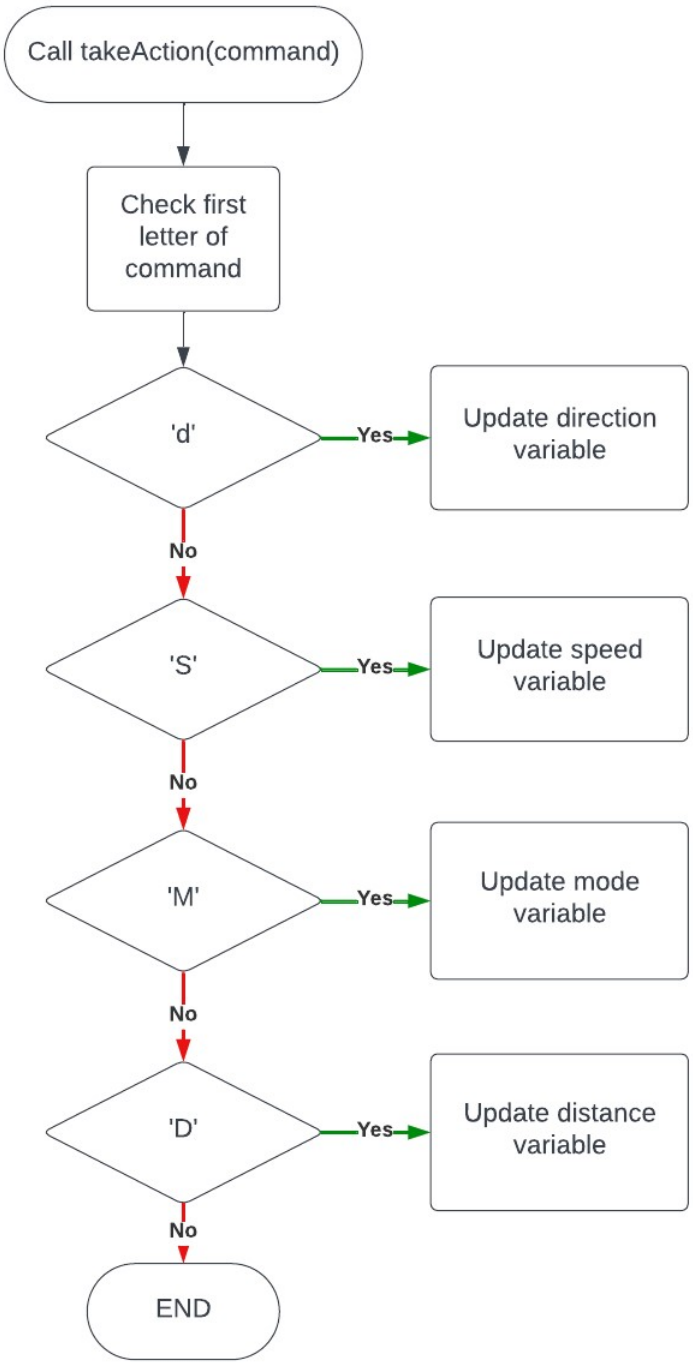

Figure 27: takeAction function

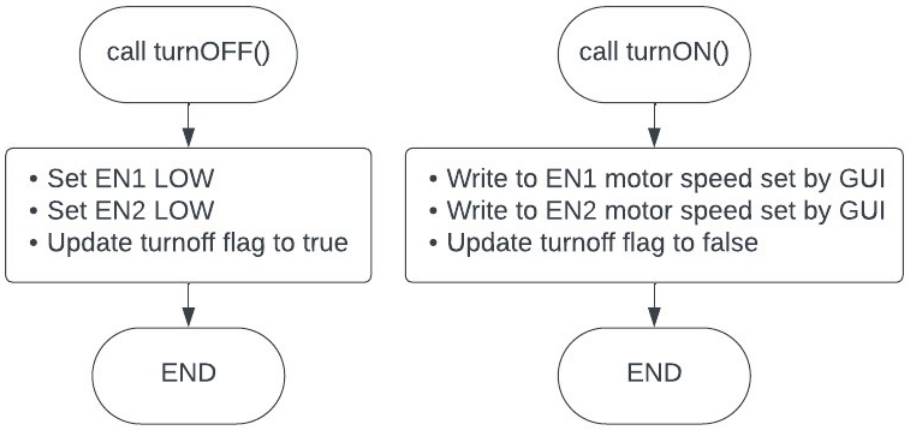

Figure 28: turnOFF and turnON function

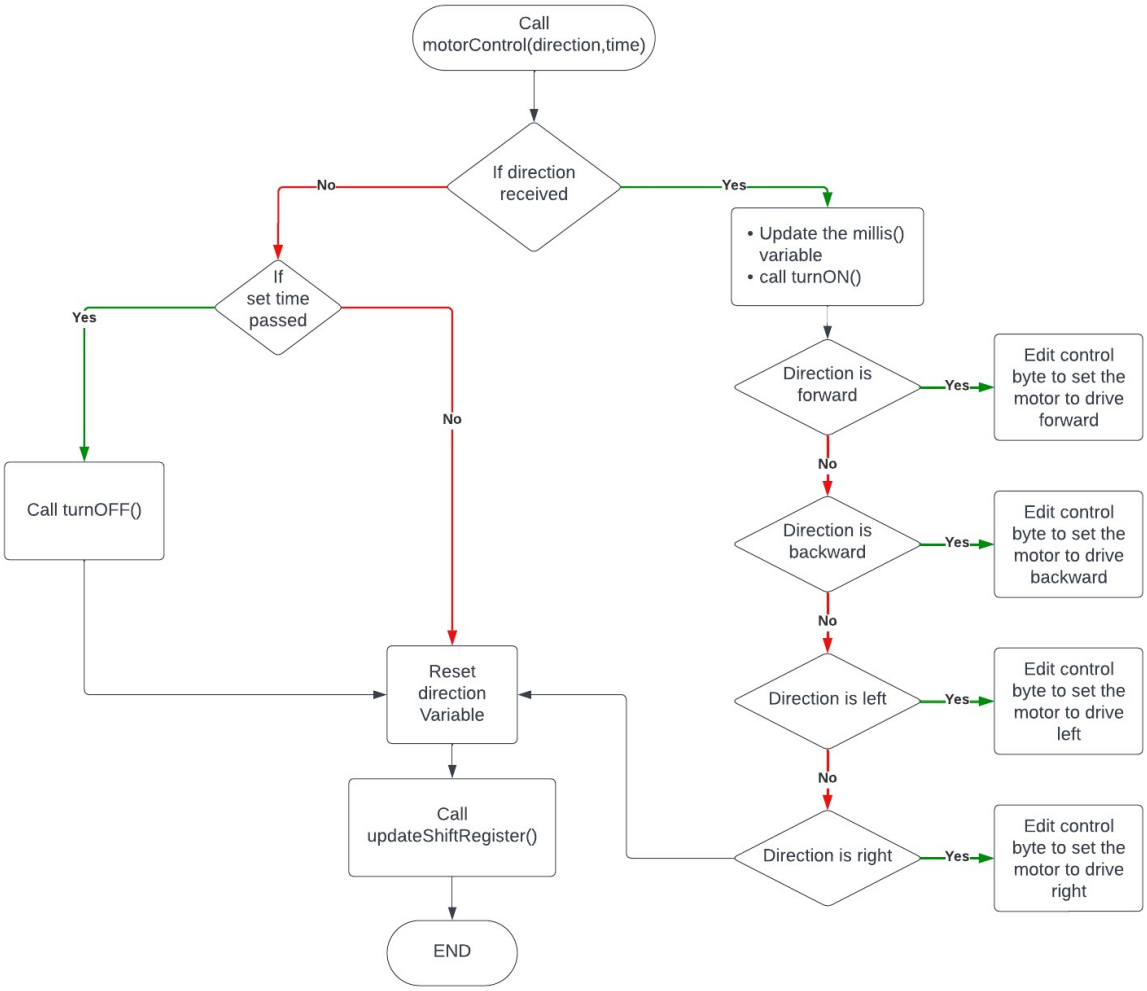

Figure 29: motorControl function

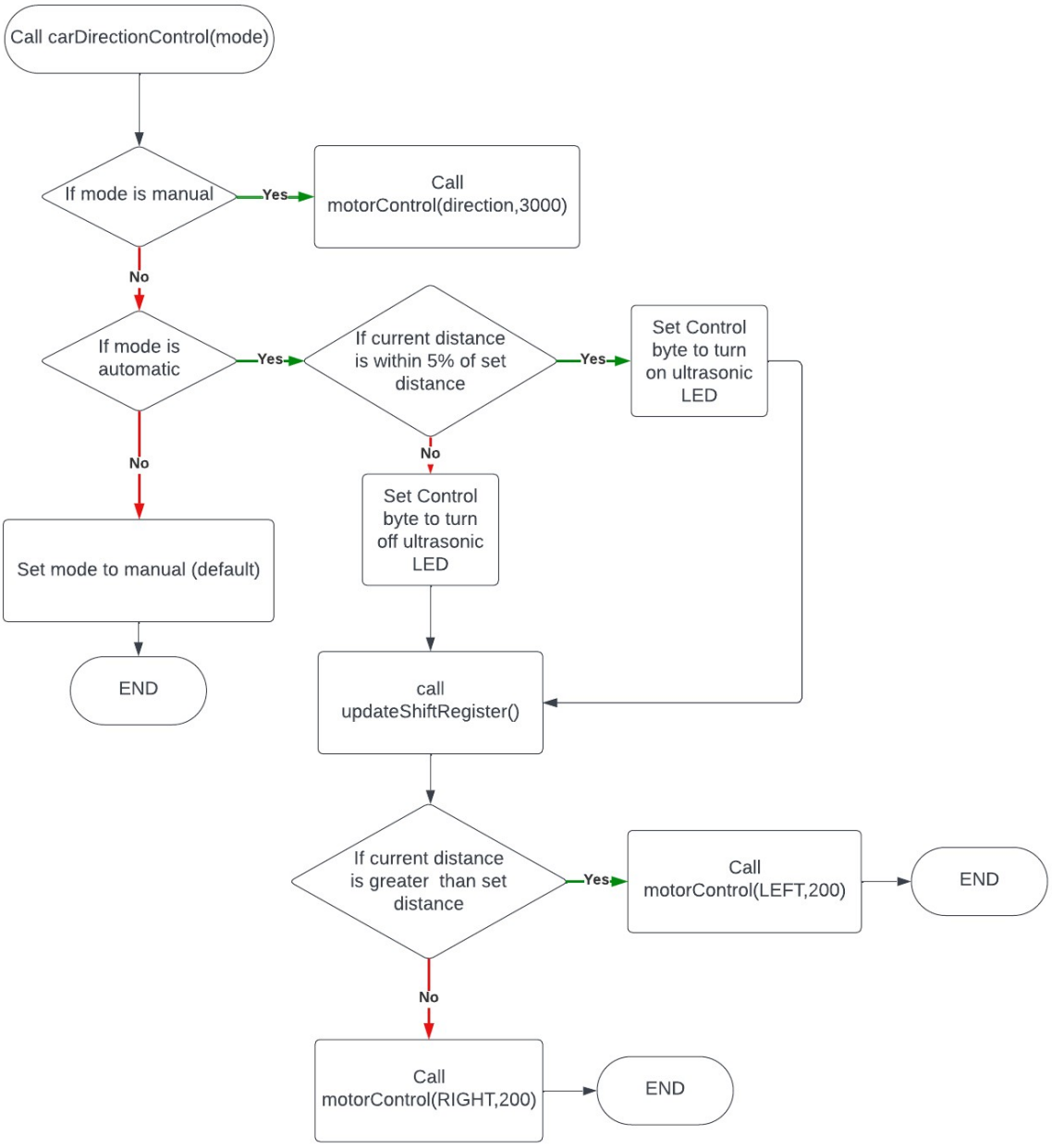

Figure 30: carDirectionControl function

Main Code

First we will map out the pin usage of the ESP:

Table 6: ESP8266 pin restrictions and mapping

PIN	Restriction	Mapped to
GPIO0	Boot mode select pin, pulled high and can only work as active LOW	Not Used
GPIO1 (TX)	Serial transmission pin. Cannot be used if serial communication needed. HIGH at boot.	Current sensor transistor switch
GPIO2	Boot mode select pin, pulled high and can only work as active LOW	Not Used
GPIO3 (RX)	Serial transmission pin. Cannot be used if serial communication needed.	Voltage sensor transistor switch
GPIO4 (SDA)	No Restrictions	Latch pin of shift register
GPIO5 (SCL)	No Restrictions	Clock pin of shift register
GPIO 6 - 11	Reserved for Flash memory. Cannot be used	Not Used
GPIO 12 (MISO)	No Restrictions	Data pin of shift register
GPIO 13 (MOSI)	No Restrictions	L293d: EN1
GPIO 14 (SCK)	No Restrictions	L293d: EN2
GPIO 15 (SS)	Boot mode select pin, pulled LOW and can only work as active HIGH	Ultrasonic sensor trigger pin
GPIO 16	Cannot be used as interrupt or for PWM	Ultrasonic sensor echo pin

Table 7: Shift register output mapping

Shift Register PIN	Mapped to
QA	Ultrasonic indicator LED
QB	RGB LED : red
QC	RGB LED : green
QD	RGB LED : blue
QE	L293d : IN1
QF	L293d : IN2
QG	L293d : IN3
QH	L293d : IN4

The code runs as follows:

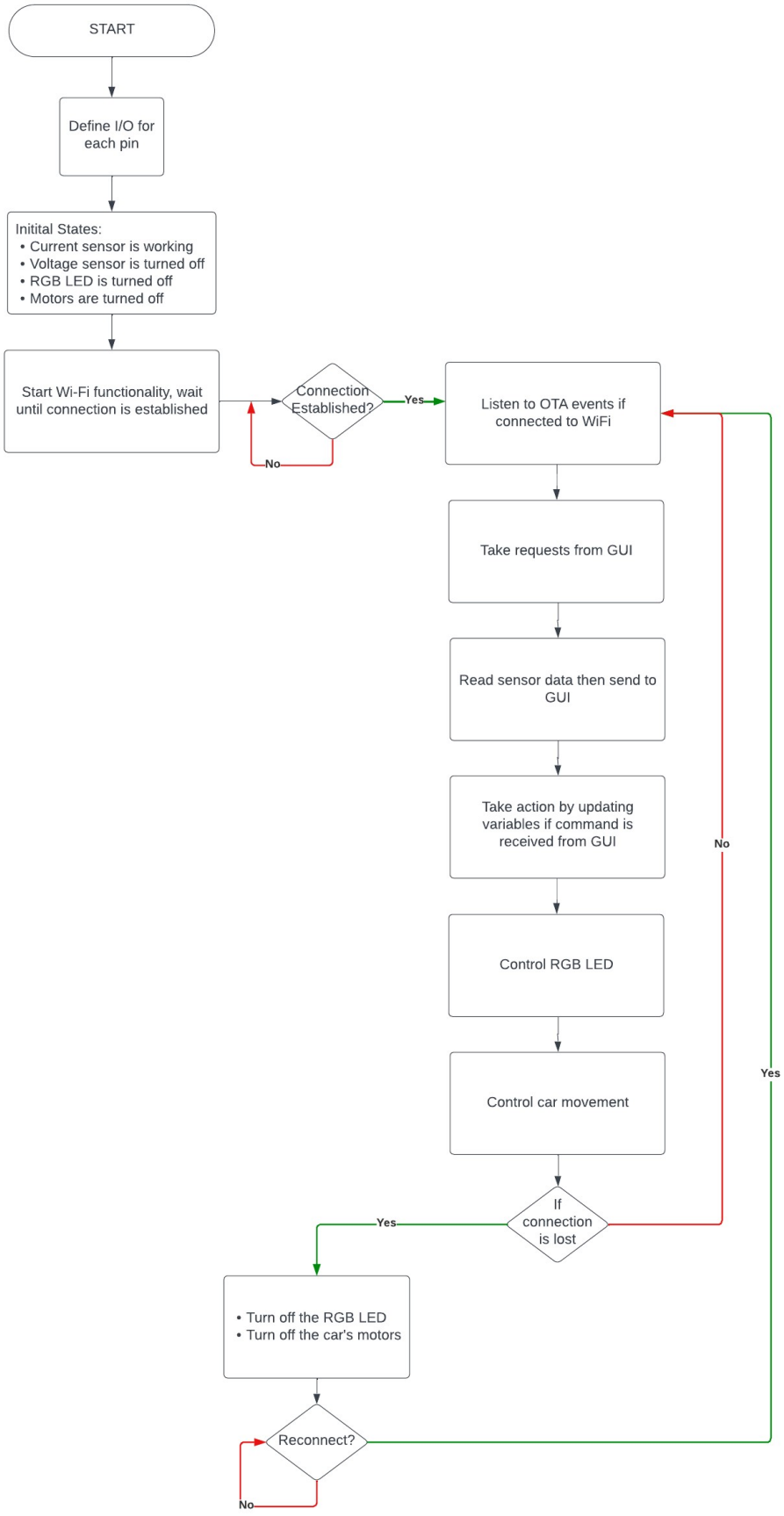

Figure 31: Setup and Loop of main code

3. GUI

I. Flowchart

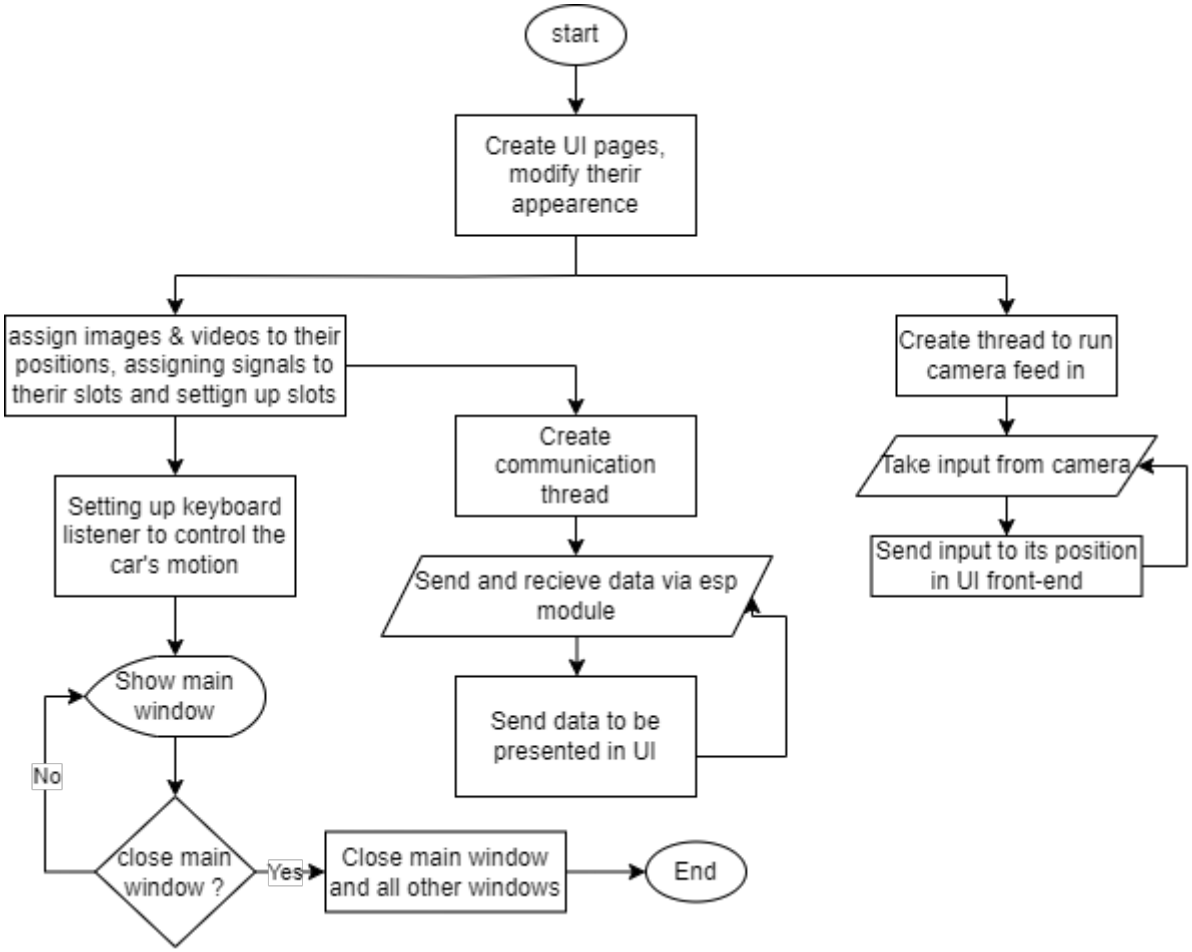

Figure 32: GUI flowchart

II. Main + Sub Window Features

Our UI is made using Qt designer IDE a free and open-source widget toolkit for creating UIs framework along with PyCharm IDE to write the backend and add few modifications to the front end. It consists of main tab with few normally closed widgets, each widget is for an image processing task. The main tab displays a live camera feed with the ability to screenshot the capture art any time. It also presents sensors readings from the car. It also gives the user the ability to control the car's motion using control buttons in the UI (for a detailed step motion) or keyboard buttons (for long distances motion). Users also can control the speed of the car using pre-specified modes or set the speed themselves using the speed control box. It also offers the ability to control the car manually or autonomously by specifying a

constant distance from the car to the wall. It also shows the connectivity mode of the UI with the car.

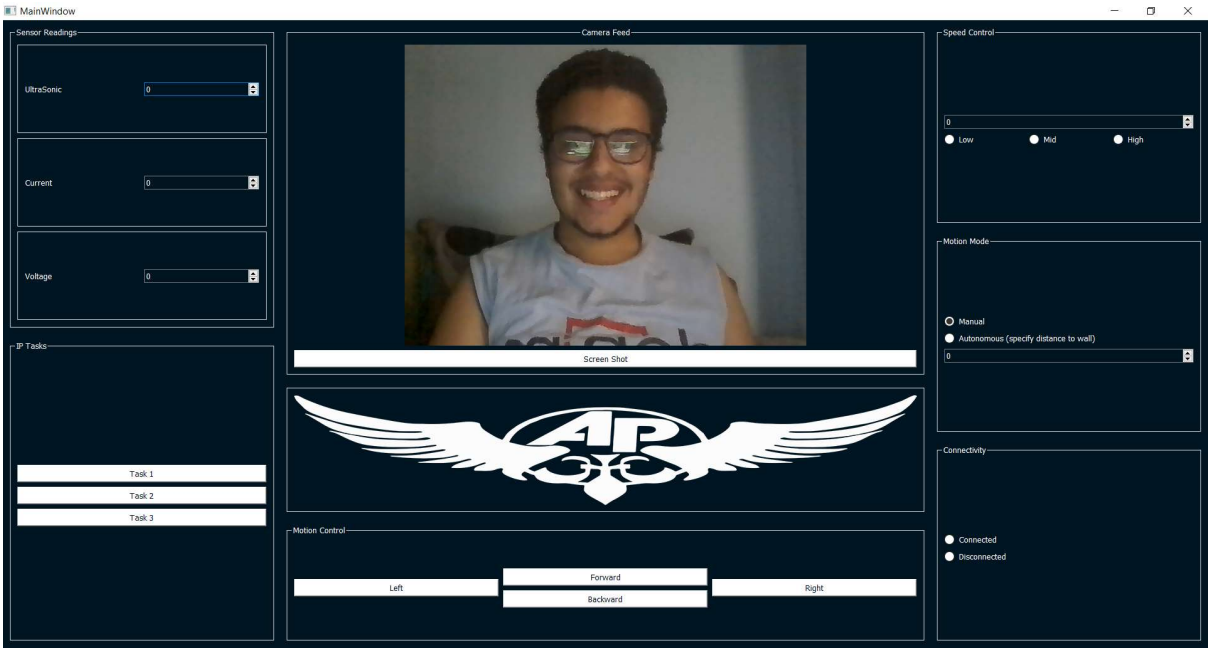

Figure 33: Main window

The UI also provides the ability to see the ins and outs of image processing tasks.

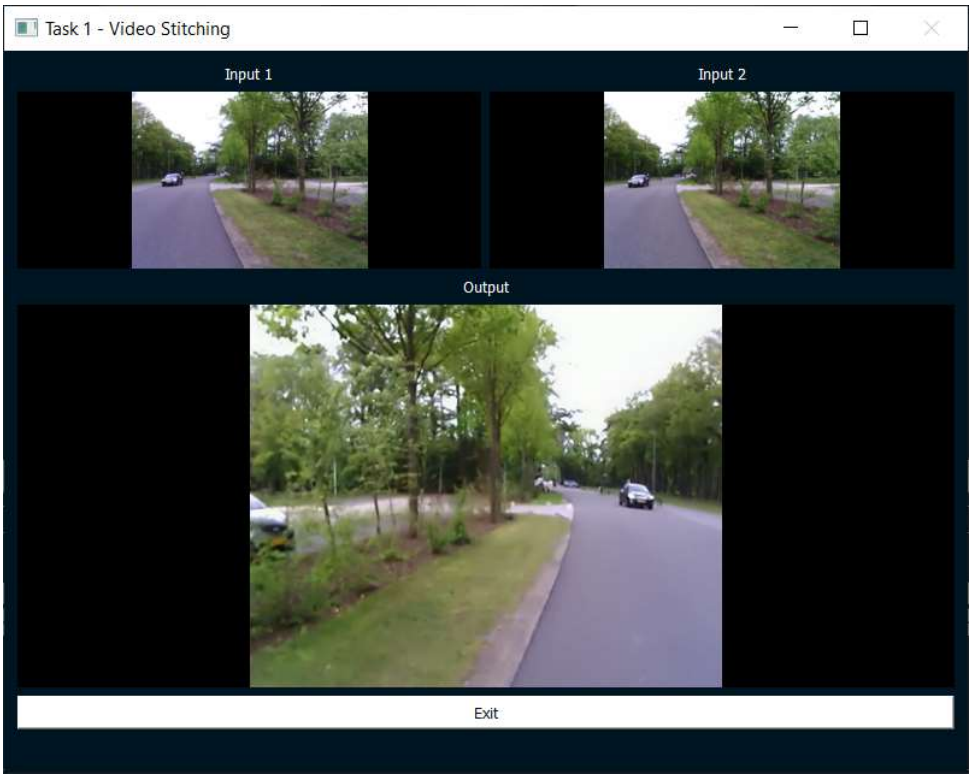

Figure 34: Video Stitching window

Our backend was written in python, seeking the highest performance and ease in programing, and modifying. Meanwhile, our frontend was written using XML language.

4. Computer Vision

I. Task 1: Video Stitching

In this task, we are required to stitch two videos together, I'll illustrate the code step by step until we obtain the output video.

1. First, we will take in the videos using cv2.VideoCapture() function and we will create an object of cv2.VideoWriter() to save our output video.
2. Next, we will enter a while loop in order to read the frames of each video, we will use .read() function to read each frame and we will save the frames in two variables which are frame1 and frame2. Before moving onto the next step, we will check if the videos didn't end where the .read() function returns two parameters , one of them is a parameter which indicates the video ended or not.
3. We will create an object of a built-in class using cv2.Stitcher_create(), add the two frames in a list then use the object we created to stitch these two frames together, now we have a panorama frame but it has a black border which makes noise and distortion in the video.
4. In order to remove this black frame, we will take a copy of our image but in gray scale then we will transform it into a binary image, then by taking contours for the maximum area which is our main frame, we can make a bounding rectangle around our image and then remove the outer noise from the final image.
5. Now, we obtained a clear image without the black bounding frame, we will add this image to the object we created in the first step which saves an output video.
6. That's it, the previous steps will be repeated until one of the videos ends, you will find the output stitched video saved in the project file.

II. Task 2: Stereo Vision

In this task, we are required to calculate the length of an object in a certain photo, I followed the pipeline instructions, the following steps explain the code:

1. The Scale-Invariant Feature Transform (SIFT) algorithm is used for feature detection and description. Keypoints and descriptors are extracted from both 'image1.png' and 'image2.png'. A Brute-Force Matcher (cv2.BFMatcher) is used to match descriptors between the two images, and a ratio test is applied to filter out good matches.

2. The Fundamental Matrix is estimated using the matched keypoints from both images. RANSAC (Random Sample Consensus) is employed to filter out outliers in the matches.

3. We identify the intrinsic parameters such as calibration matrices for both cameras ('cam0' and 'cam1'), focal length, camera center coordinates, baseline, image dimensions, and disparity range.

4. Then the Essential Matrix (E) is computed by multiplying the calibration matrices with the Fundamental Matrix (F). The Essential Matrix is then decomposed into the relative rotation matrix (R) and translation vector (t) using cv2.recoverPose().

5. Rectification transformation matrices (R1 and R2) are calculated for both images. Rectified images are generated using cv2.remap() to ensure corresponding points are aligned horizontally.

6. Epipolar lines are drawn on the rectified images corresponding to the matched keypoints. These lines help visualize the epipolar geometry between the two images.

7. A stereo matching algorithm (StereoBM) is used to compute the disparity map between the rectified images. The disparity map is normalized and shown.

8. The depth map is calculated from the disparity map using the given calibration parameters. It represents the depth of each pixel in the scene.

9. Object detection is performed using a pre-trained MobileNet model. Objects are detected using forward pass through the network. Bounding boxes are drawn around detected objects, and regions of interest (ROIs) are extracted.

10. For each ROI, the 3D coordinates of pixels are calculated using the depth map. The Euclidean distance between the first and last pixel in each ROI is computed to determine the length of the detected object in 3D space. The length is then converted to centimeters.

Just to be clear, the code works well till we reach the length calculation part, don’t know what is causing the issue for now, but I’ll try to solve this problem before the project’s discussion, regardless of this problem, the code performs all of the other steps without facing any difficulties.