Autonomous Navigation of UGV/UAV

Sachinkumar Omprakash Dubey

A thesis Submitted to

Indian Institute of Technology, Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of M.Tech. in Communication & Signal Processing

Department of Electrical Engineering

June 2022

Abstract

This report is intended to serve as a concise and to-the-point report of my work in the domain of autonomous navigation of UGVs (Unmanned Ground Vehicles) and UAVs (Unmanned Aerial Vehicles). A driver or operator is generally required to operate a ground vehicle (car, bus, etc.) or an aerial vehicle (quad-copter, etc.). These operators/drivers must be trained, and obtaining a license is also a tedious process (especially for aerial vehicles). In reality, these factors prevent us from utilizing these machines to their full potential and exploring their wide array of valuable applications.

By automating ground vehicles, we could save on fuel, reduce accident-related costs, and improve overall transportation efficiency. An autonomous aerial vehicle, on the other hand, can be used for many purposes, including: Food and medical delivery in remote areas, Public safety in calamities, Bridge and power lines inspection, Agriculture, Road accident report, Traffic monitoring, and Transportation.

This report presents implementation of various application of UGV/UAV kit which forms a base for more complex autonomous navigation application.

Contents

List of Figures

Chapter 1

Introduction

1.1 Controllers and their specifications

1.1.1 ESP32 Micro-controller

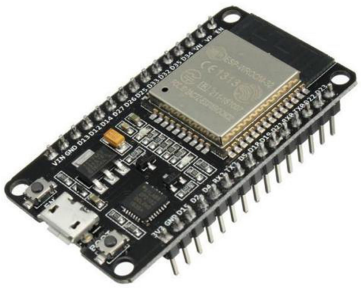


Figure 1.1: ESP-WROOM-32 Dev Kit

ESP32 is among the popular micro-controllers. The ESP32 series utilizes Tensilica's Xtensa LX6 microprocessor, or the Xtensa LX7 dual-core microprocessor or a single-core RISC-V microprocessor. It includes built-in antenna switches, RF balun, power amplifier, low-noise receive amplifier, filters, and power-management modules. Unlike Arduino Uno it provide on-board WiFi and Bluetooth functionalities. It supports various interfaces such as SD card, UART, SPI, I2C, LED/Motor PWM module, GPIO, IR, ADC/DAC, etc. It operates on voltage supply of 3-3.6V. It is a successor to the ESP8266 micro-controller.

1.1.2 Vaman Pygmy BB4

Vaman is a development board based around the Pygmy Stamp module, and brings on-board WiFi/BT/BLE connectivity with ESP32, as well as an 18650 battery option, μSD Card (connected to ESP32), possibility to use EOSS3 in smartphone mode with the ESP32 as the Host, and all EOSS3 pins accessible. A BMX160 provides an AMG IMU, supplemented by a BNO055 smart fusion sensor. A DPS310 provides Pressure, Humidity and Temperature monitoring.

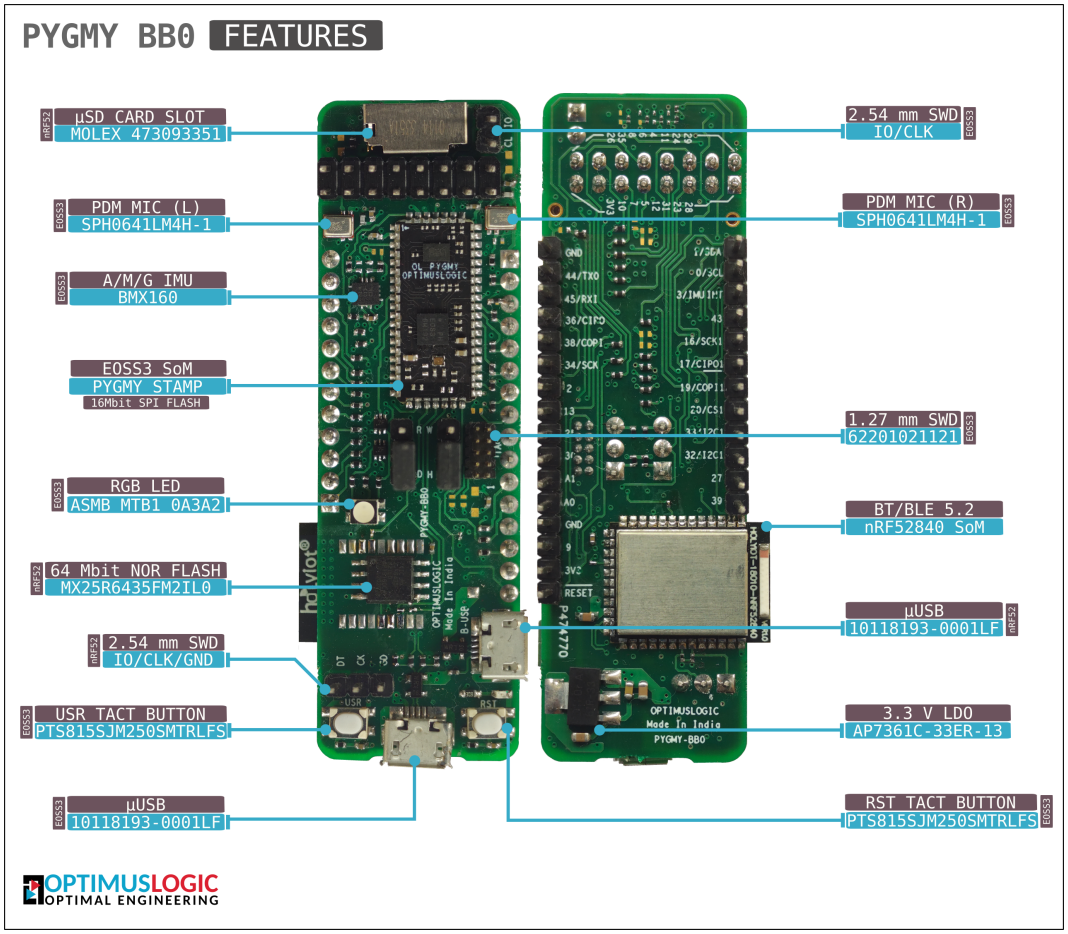


Figure 1.2: Vaman (Pygmy BB4)

1.1.3 Arduino Uno

Arduino is an open-source development board ideal for handling a variety of applications. It is inexpensive as well as feature-packed. It is compatible with a variety of daughter boards that can be attached to it. The low-power Bluetooth shield, along with the Wi-Fi and Ethernet shields, allows increased communication functionalities.

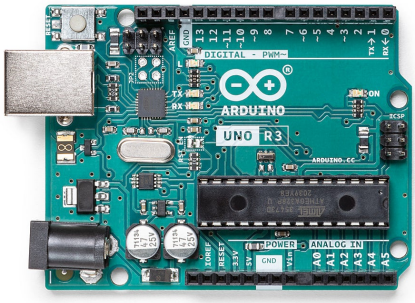


Figure 1.3: Arduino Uno R3

1.1.4 Raspberry Pi 3B

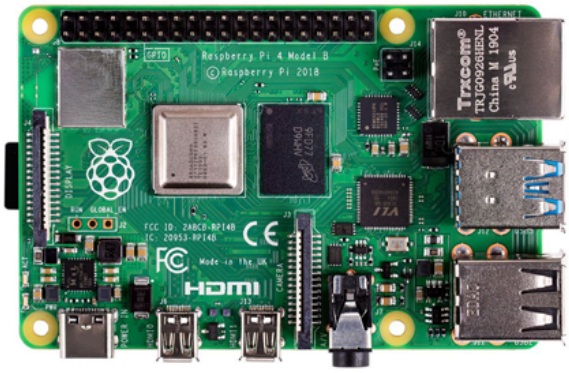


Figure 1.4: Raspberry Pi 3B

Raspberry Pi (RPi) is an on-board computer and it is among the most popular development boards. The Raspberry Pi runs a customized version of Linux called Raspberry Pi OS. It has all the features of a personal computer. This board has USB interfaces that allow it to connect to peripherals (such as a mouse and keyboard), USB storage, etc. HDMI interface can be used to connect displays to the board having up to 4K resolution. Additionally, it has other ports/interfaces, including USB-C (power), a 2-lane MIPI CSI port (camera), a 3.5mm audio jack (audio), and RJ-45 (Ethernet). WiFi and Bluetooth 5.0 support are provided by the Raspberry Pi.

Parameters	Arduino Uno	Raspberry Pi 3B	ESP-32
Processor	ATMega328P	Quad-core Broadcom BCM2837 (4×Cortex-A53)	Xtensa Dual-Core 32-bit LX6 with 600 DMIPS
GPU	-	Broadcom VideoCore IV @ 250 MHz	-
Operating voltage	5V	5V	3.3V
Clock speed	16 MHz	1.2GHz	26 MHz – 52 MHz
System memory	2kB	1 GB	<45kB
Flash memory	32 kB	-	up to 128MB
EEPROM	1 kB	-	-
Communication supported	IEEE 802.11 b/g/n Bluetooth via Shield	IEEE 802.11 b/g/n Bluetooth, Ethernet Serial	IEEE 802.11 b/g/n
Development environments	Arduino IDE	Any linux compatible IDE	Arduino IDE, Lua Loader
Programming language	Embedded C, C++	Python, C, C++, Java, Scratch, Ruby	Embedded C, C++
I/O Connectivity	SPI I2C UART GPIO	SPI DSI UART SDIOCSI GPIO	UART, GPIO

Table 1.1: Comparison between Arduino Uno, Raspberry Pi 3B and ESP-32

Chapter 2

Basics of Motor control

2.1 PWM speed control

- A pulse width modulation speed control system works by sending a series of "ON-OFF" pulses to the motor. The frequency of square wave is kept constant while varying the duty cycle (the fraction of time that the output voltage is "ON" compared to when it is "OFF").
- By changing the width of the ON duration, one can control the average DC voltage applied to the motor.
- In other words, the wider the pulse width, the more average voltage applied to the motor terminals, the stronger the magnetic flux inside the armature windings and the faster the motor will rotate.
- Larger the duty-cycle, the faster the motor will rotate. Similarly, shorter the duty-cycle, the slower the motor will rotate.

2.2 Brushed DC motor

- The Brushed DC motor used in UGV kit is controlled using the dual motor driver module (L298N). The driver module can control the speed as well as the direction of rotation of the two DC motors
- PWM speed control principle discussed in previous section is used to control the speed of the motors. The pin out of the driver module is as shown in figure 2.2.

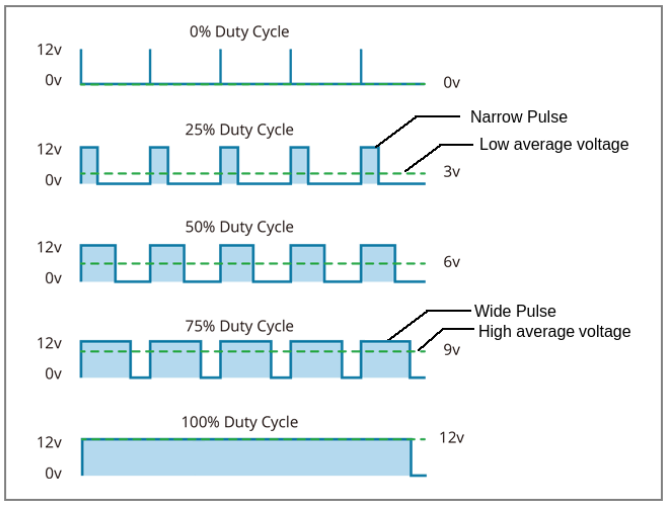


Figure 2.1: PWM speed control

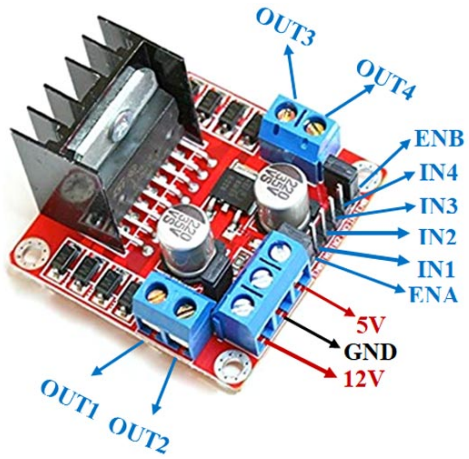


Figure 2.2: Dual motor driver module (L298N)

- The two motors are connected at the OUT1/OUT2 and OUT3/OUT4 terminals respectively.
- The ENA and ENB pins are for speed control of the two motors. PWM signal generated at the controller end is sent to these pins for motor speed control.
- The IN1 and IN2 pins are used for direction control of motor-1, while IN3 and IN4 pins are used for direction control of motor-2.
- 12V DC input is routed to the motors using H-bridge inside the L298 IC mounted on the driver module. The motor driver also has a regulator IC which produces 5V output which can be used to drive our micro-controller unit. This eliminate the need for a separate power supply for the micro-controller.

A Typical connection between the driver module and the motors is as shown in Figure 2.3. The speed and direction control signals are received from the micro-controller unit.

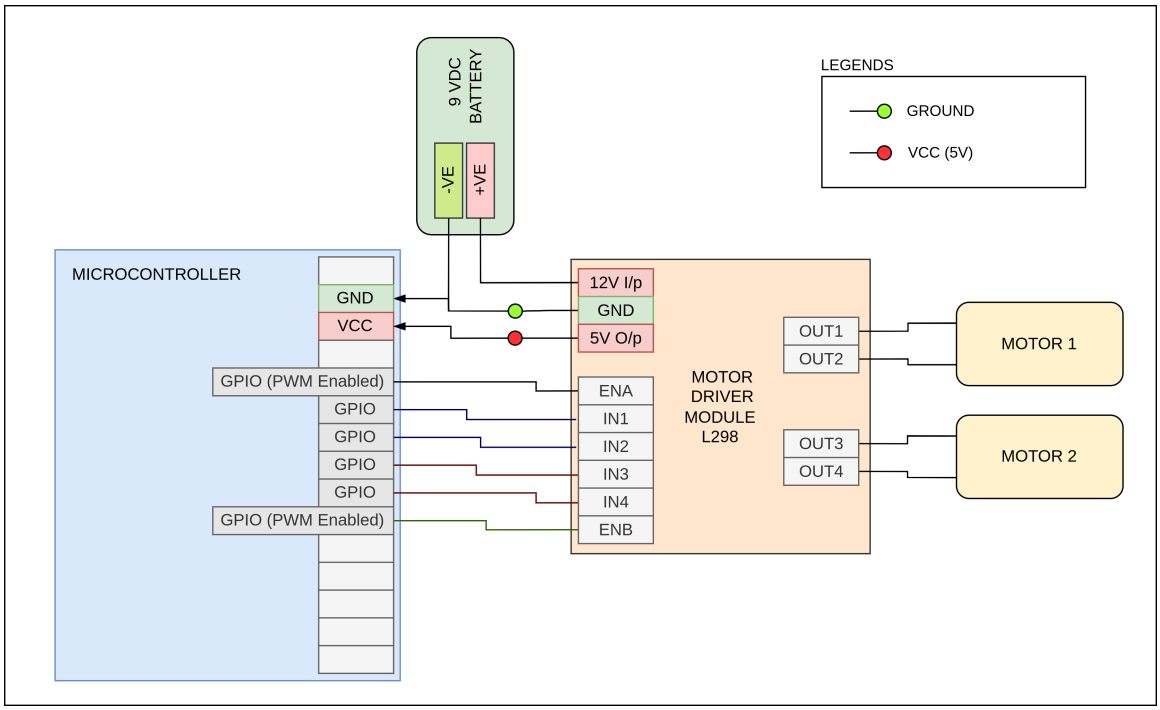


Figure 2.3: Typical connection for motor driver module

2.3 Brush-less DC motor

BLDC motors primarily consists of two main parts i.e. stator and rotor as shown in Figure xxx. Stator consist of coils made from good conducting materials (copper or aluminium), while rotor is a permanent magnet. When current is passed through coils of stator, a electromagnet is created generating a magnetic field which attracts the permanent magnet (rotor). This is basic principle which gives us rotation of BLDC motors.

Generally the opposite coils of stator winded together to provide better torque. BLDC motors comes in various configurations based on number of poles and rotor positioning. Inrunner BLDC motors have rotor part positioned inside stator part, while Outrunner BLDC motors have rotor part positioned outside the stator part as shown in figure 2.4.

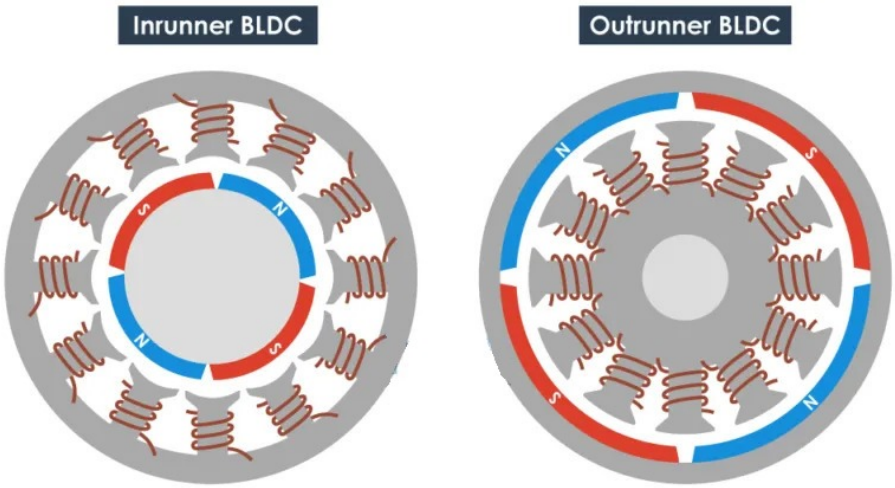


Figure 2.4: Inrunner BLDC and Outrunner BLDC motors

The Electronic speed controller is a device which rotates the BLDC motor by appropriately activating the coils using MOSFET pairs to create rotating magnetic field. Due to MOSFET the switching speed and hence the motor speed can be controlled. The connection between ESC and BLDC motor is as shown in Figure 2.6. Basically, the ESC uses two types of mechanism in order to figure out the position of the stator so that it can decide which coil should be activated for continuous rotation:

- Using Hall effect sensor
- Back EMF sensing

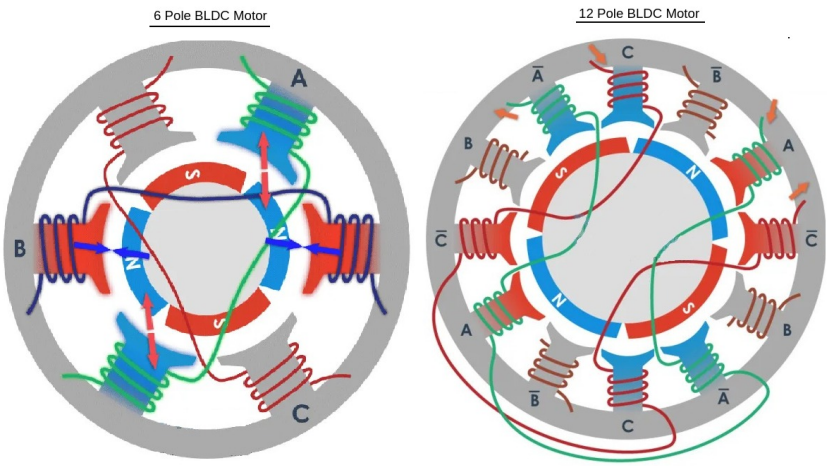


Figure 2.5: 6 Pole and 12 Pole BLDC motors

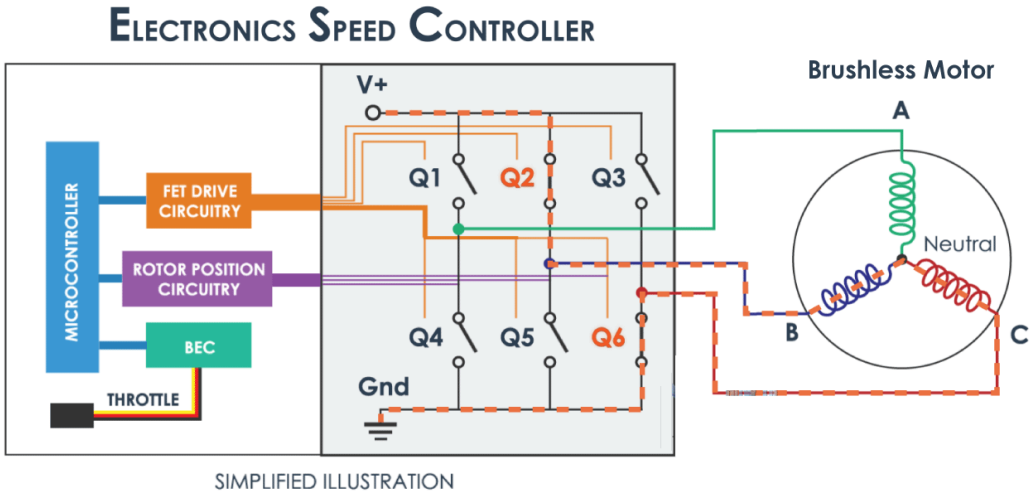


Figure 2.6: Connection between the ESC and BLDC motor

Chapter 3

Serial communication protocols

3.1 UART (Universal Asynchronous Receiver/Transmitter)

Features of UART interface:

- UART is a lower data serial communication protocol. The receiver device should know baud-rate of the transmitter device before actual communication establishment.
- UART is simple protocol, it uses start bit (before data word), stop bits (one or two, after data word), parity bit (even or odd) in its base format for data formatting. Parity bit helps in one bit error detection. UART Packet = 1 start bit(low level), 8 data bits including parity bit, 1 or 2 stop bit(high level)
- Data is transmitted byte by byte. UART does not have clock as it is asynchronous serial communication protocol. It makes use of stop and start bits to synchronise the communication process.
- For long distance communication, 5V UART is converted to higher voltages viz. +12V for logic 0 and -12V for logic 1. The figure 3.1 below shows UART interface between two devices.

3.2 SPI (Serial Peripheral Interface)

- SPI is a protocol on 4 signal lines:
  - A clock signal named SCLK, sent from the bus master to all slaves; all the SPI signals are synchronous to this clock signal;

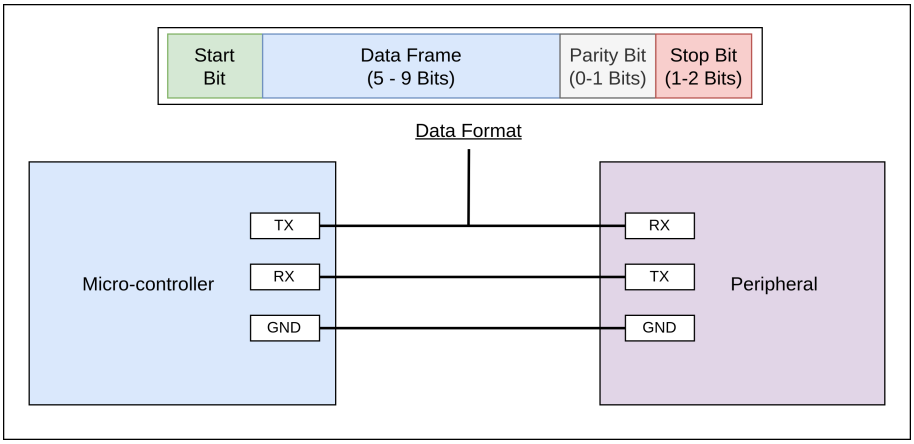


Figure 3.1: Data format and connection between two devices in UART interface

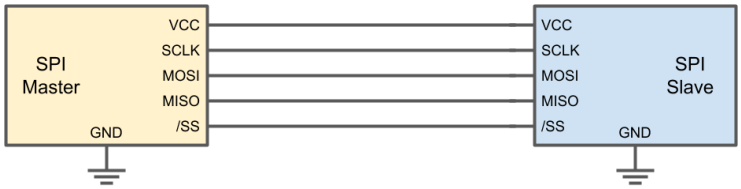


Figure 3.2: SPI point-to-point connections

  - A slave select signal for each slave, SSn, used to select the slave the master communicates with;
  - A data line from the master to the slaves, named MOSI (Master Out-Slave In)
  - A data line from the slaves to the master, named MISO (Master In-Slave Out)

- Two SPI bus Topologies:
  - SPI Master connected to a single slave (Point-to-Point topology) as shown in figure 3.2.
  - SPI Master connected to multiple slaves as shown in Figure 3.3.

3.3 I2C Protocol

Features:

- I2C is a two-wire communication protocol that is commonly used to connect low-speed devices like micro-controllers, I/O interfaces, A/D and D/A converters, EEPROMs, and other peripherals in embedded systems.

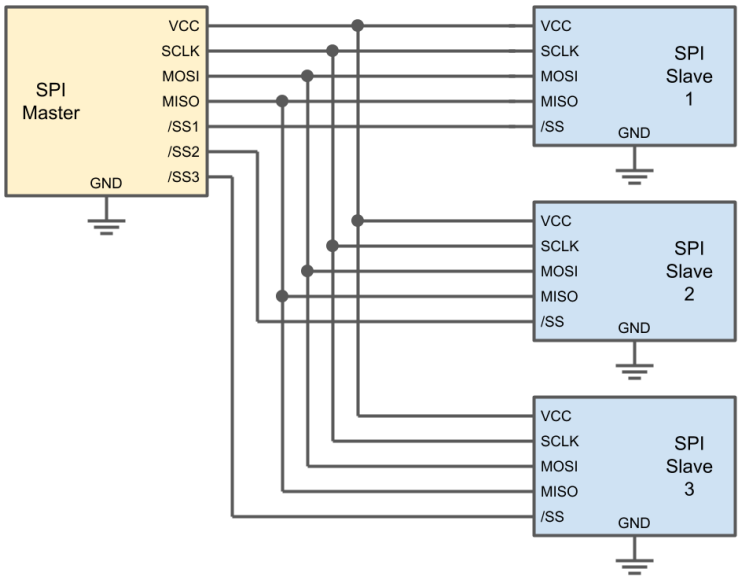


Figure 3.3: SPI connections for 3 slave devices

- One of these wires, known as SCL (Serial Clock) carries the clock signal, while the other wire, known as SDA (Serial Data) allows master and slave devices on the bus to send and receive data.

- The I2C protocol allows for multiple slave devices to be connected to a single master device, or for multiple masters controlling one or more slave devices (Figure 3.4).

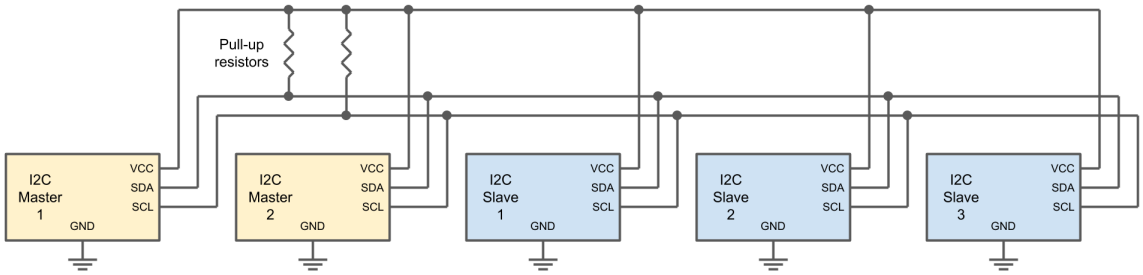


Figure 3.4: I2C connections for multiple master and multiple slave devices

Chapter 4

Implementation of various applications

4.1 UGV (Unmanned Ground Vehicle)

4.1.1 Navigation using Fly-sky Transmitter and Receiver(ESP32)

Required components/Software tools

- UGV chassis with DC motors
- ESP32 micro-controller with Type-B USB cable
- L293D Motor Driver IC
- Fly-sky Transmitter and Receiver
- Breadboard
- Jumper Wires
- Arduino IDE installed on system

Steps

- Make the connections as per the wiring diagram (Figure 4.1).
- Go to Arduino IDE and write the following program available on the code link at 4.1.1.

- Compile and upload the program to ESP32 micro-controller using the Type-C programmable cable.
- Test whether the UGV is navigating as per the command sent from the transmitter.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/ESP32/IDE/UGV_navigation_using_Flyskly/Codes
```

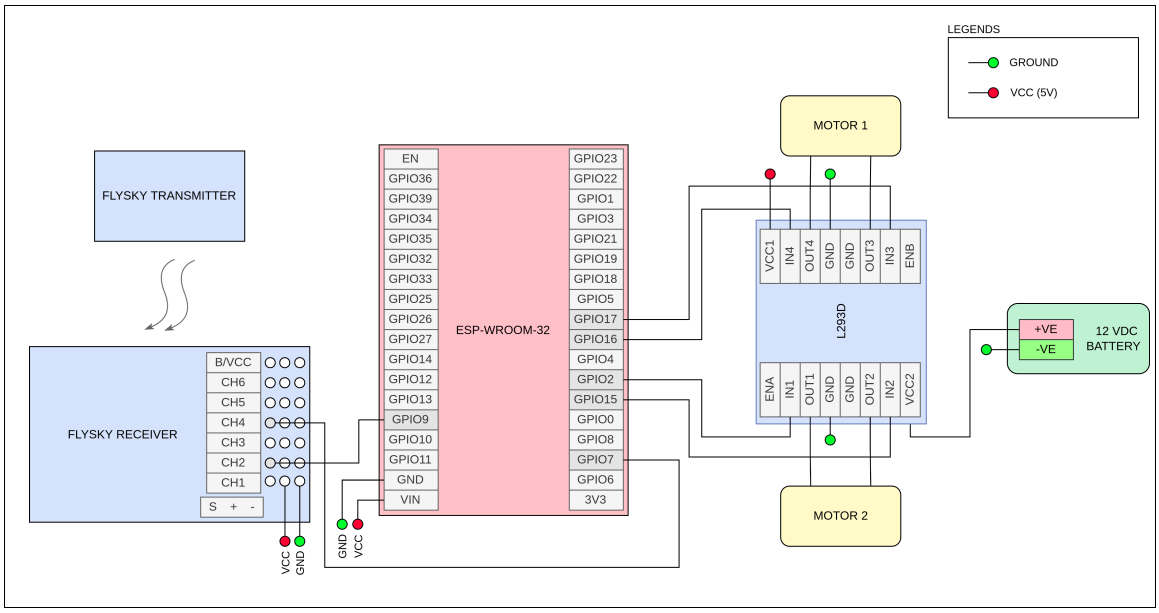


Figure 4.1: Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (ESP32)

Working

- User first give navigation input using the Throttle and Roll sticks on transmitter. This input is transmitted to the receiver wirelessly.
- Receiver generates PWM signals on channel-2 and channel-4 as per the navigation input received from the transmitter.
- These PWM signals are read by the ESP32 using pulseIn function.
- Depending on the read values the program sends suitable commands (forward, backward, left, right and stop) to the direction control pins of Motor driver IC.

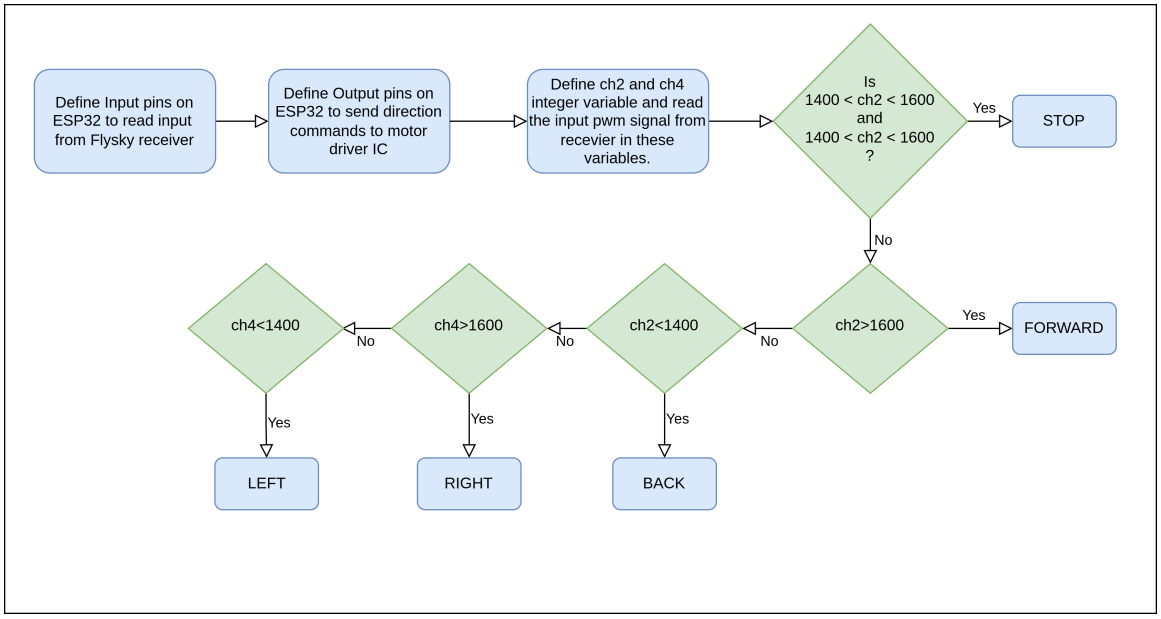


Figure 4.2: Flow Diagram for UGV Navigation using Fly-sky transmitter & receiver

4.1.2 Navigation using Fly-sky Transmitter and Receiver(Vaman)

Required components/Software tools

- UGV chassis with DC motors
- Vaman micro-controller with Type-B USB cable
- L293D Motor Driver IC
- Fly-sky Transmitter and Receiver
- Breadboard
- Jumper Wires
- QORC SDK installed on linux system with supporting tools.

Steps

- Make the connections as per the wiring diagram (Figure 4.3).
- Connect the Vaman board to Laptop/PC using Type-B USB cable.
- Run ”make” command in ”GCC project” folder to generate bin file for the ”main.c” file at the code link at (4.1.2)

- Flash the generated bin file on Vaman using the tinyfpga-programmer-gui.py tool.
- Test whether the UGV is navigating as per the command sent from the transmitter.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/Vaman/Vaman_UGV_flysky/src
```

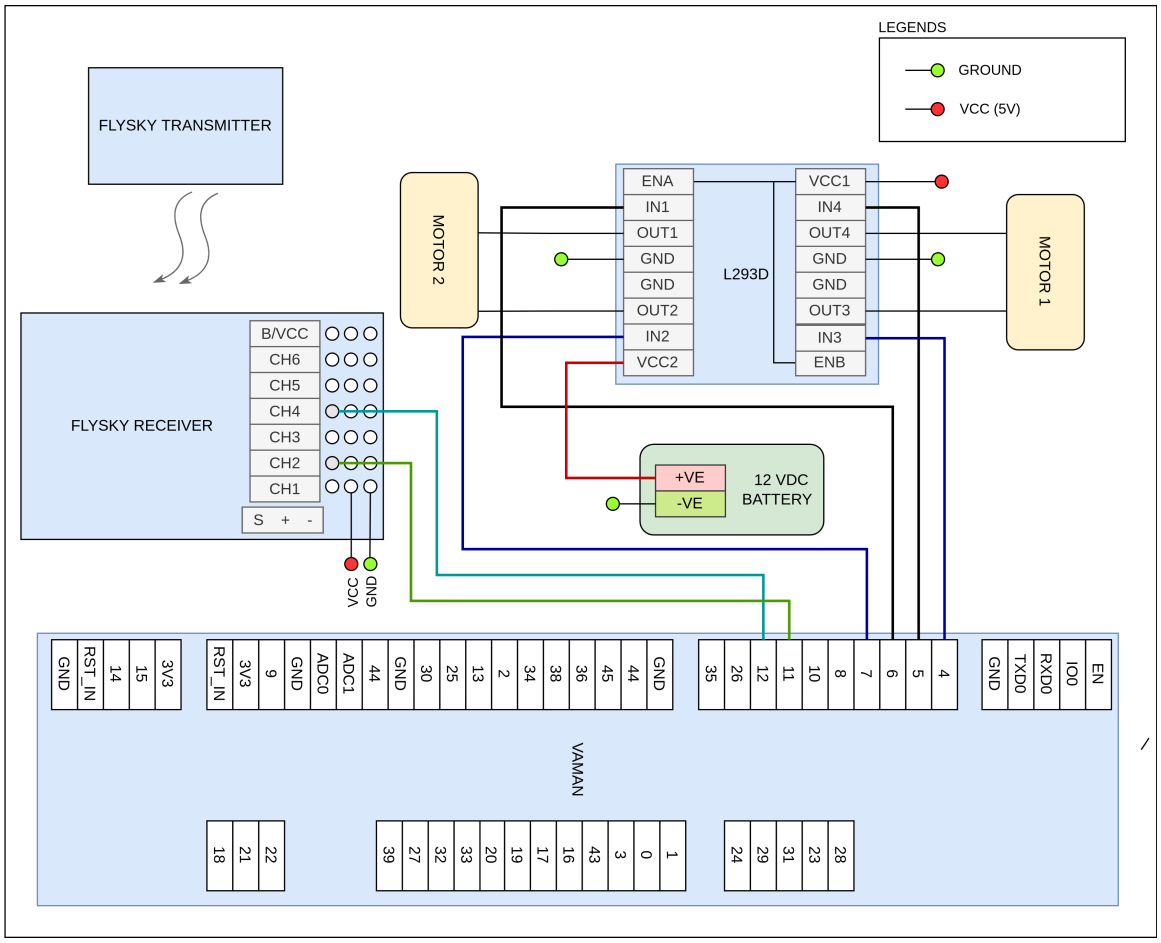


Figure 4.3: Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (Vaman)

Working

- User first give navigation input using the Throttle and Roll sticks on transmitter. This input is transmitted to the receiver wirelessly.
- Receiver generates PWM signals on channel-2 and channel-4 as per the navigation input received from the transmitter.

- These PWM signals are read by the Vaman using pulseIn function.
- Depending on the read values the C program sends suitable commands (forward, backward, left, right and stop) to the direction control pins of Motor driver IC.

4.1.3 Navigation using Android phone (ESP32)

Required components/Software tools

- UGV chassis with DC motors
- ESP32 micro-controller with Type-B USB cable
- L293D Motor Driver IC
- Breadboard
- Jumper Wires
- PlatformIO installed on VS code
- Android phone with Dabble app installed.

Steps

- Make the connections as per the wiring diagram (Figure 4.4).
- Go to PlatformIO and execute the "main.cpp" file given at code link at 4.1.3.
- Flash firmware.bin obtained upon execution of the above code to the ESP32.
- Test whether the UGV is navigating as per the command sent from the dabble app installed on Android phone.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/ESP32/IDE/UGV_navigation_using_android_phone/Codes
```

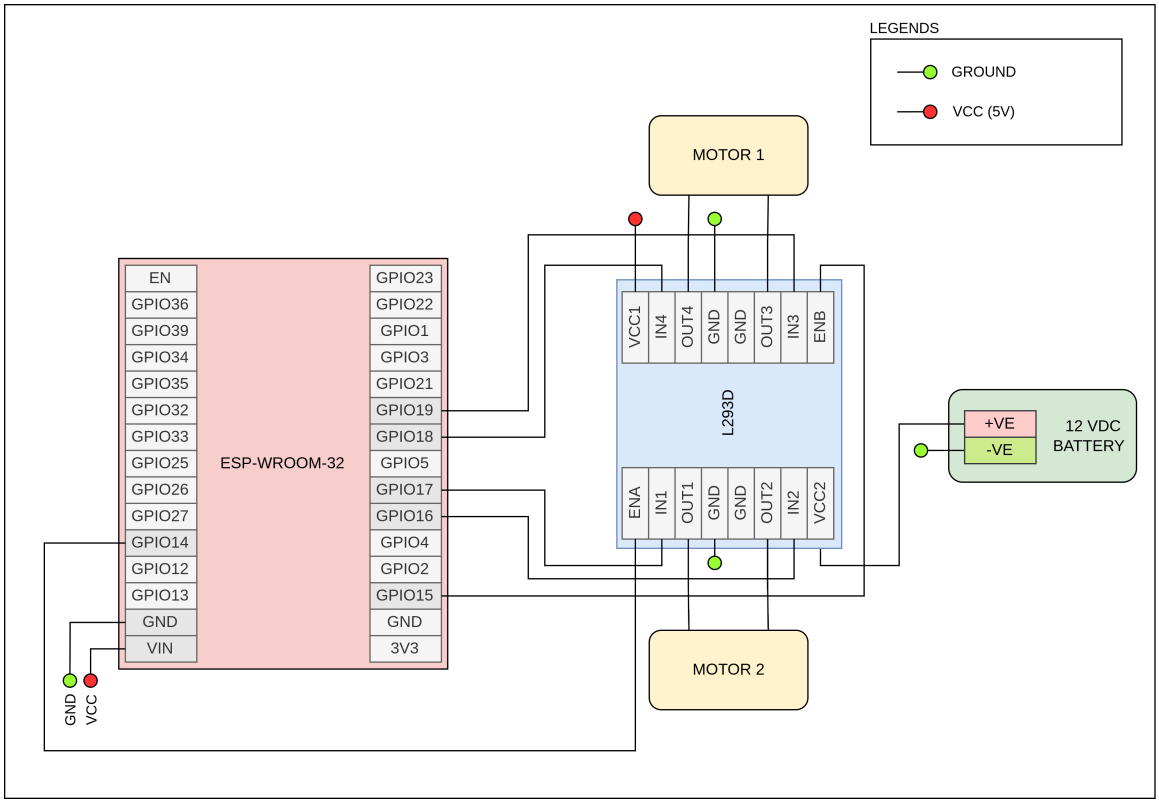


Figure 4.4: Wiring Diagram for UGV Navigation using Android phone (ESP32)

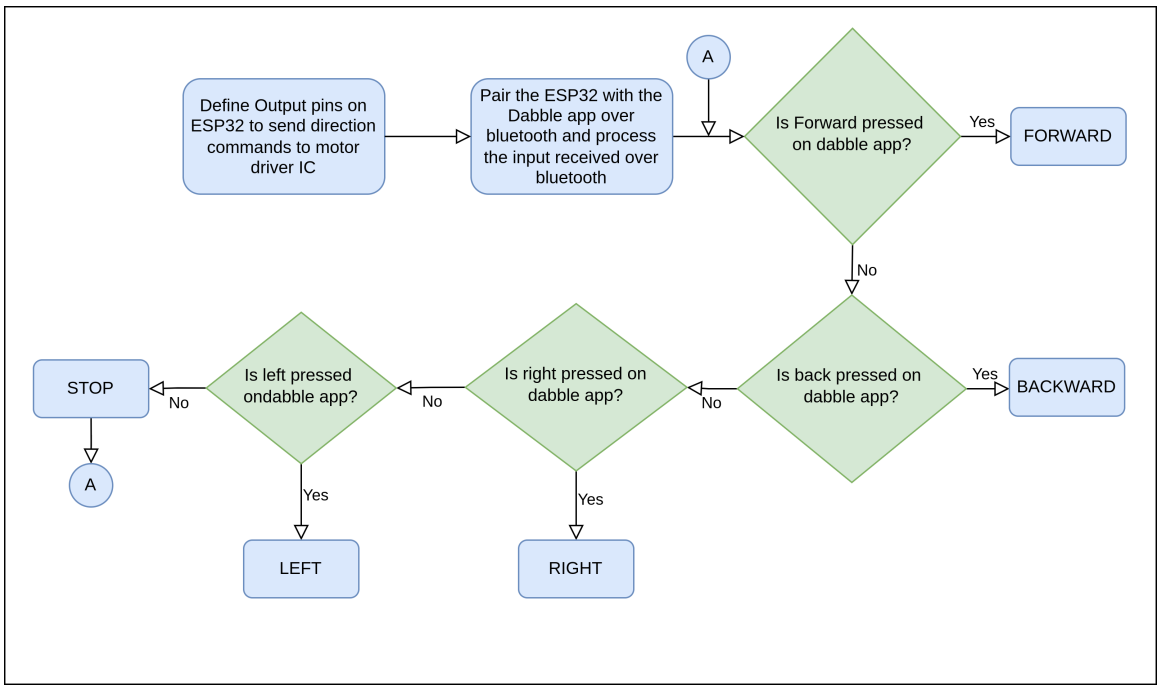
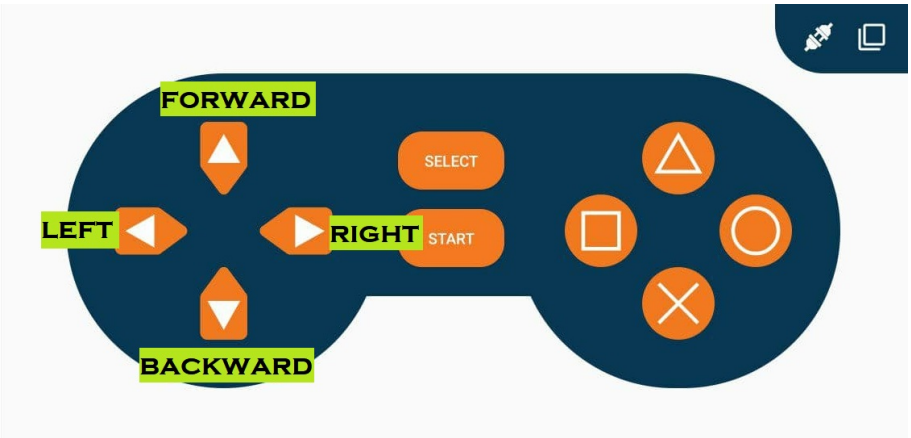


Figure 4.5: Flow Diagram for UGV Navigation using Android phone

Figure 4.6: Dabble app user interface



Working

- User gives navigation input using the dabble app installed on the android device. The dabble app communicates to ESP32 over bluetooth. The dabble app UI is as shown in figure 4.6.
- Depending on the input received over bluetooth, the program sends suitable commands (forward, backward, left, right and stop) to the direction control pins of Motor driver IC.

4.1.4 Navigation using Android phone (Vaman)

Required components/Software tools

- UGV chassis with DC motors
- Vaman development board with type-B USB cable
- L293D Motor Driver IC
- Breadboard
- Jumper Wires
- QORC SDK installed on linux system with supporting tools
- Android phone with Dabble app installed.

Steps

- Make the connections as per the wiring diagram (Figure 4.7).
- Connect the Vaman board to Laptop/PC using Type-B USB cable.

- Run "make" command in "GCC_project" folder to generate bin file for the "main.c" file at the code link (4.1.4).
- Flash the generated bin file on Vaman using the tinyfpga-programmer-gui.py tool.
- Verify that the UGV navigates as per the command received from dabble app installed on the android phone.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/Vaman/Vaman_mobile_control/src
```

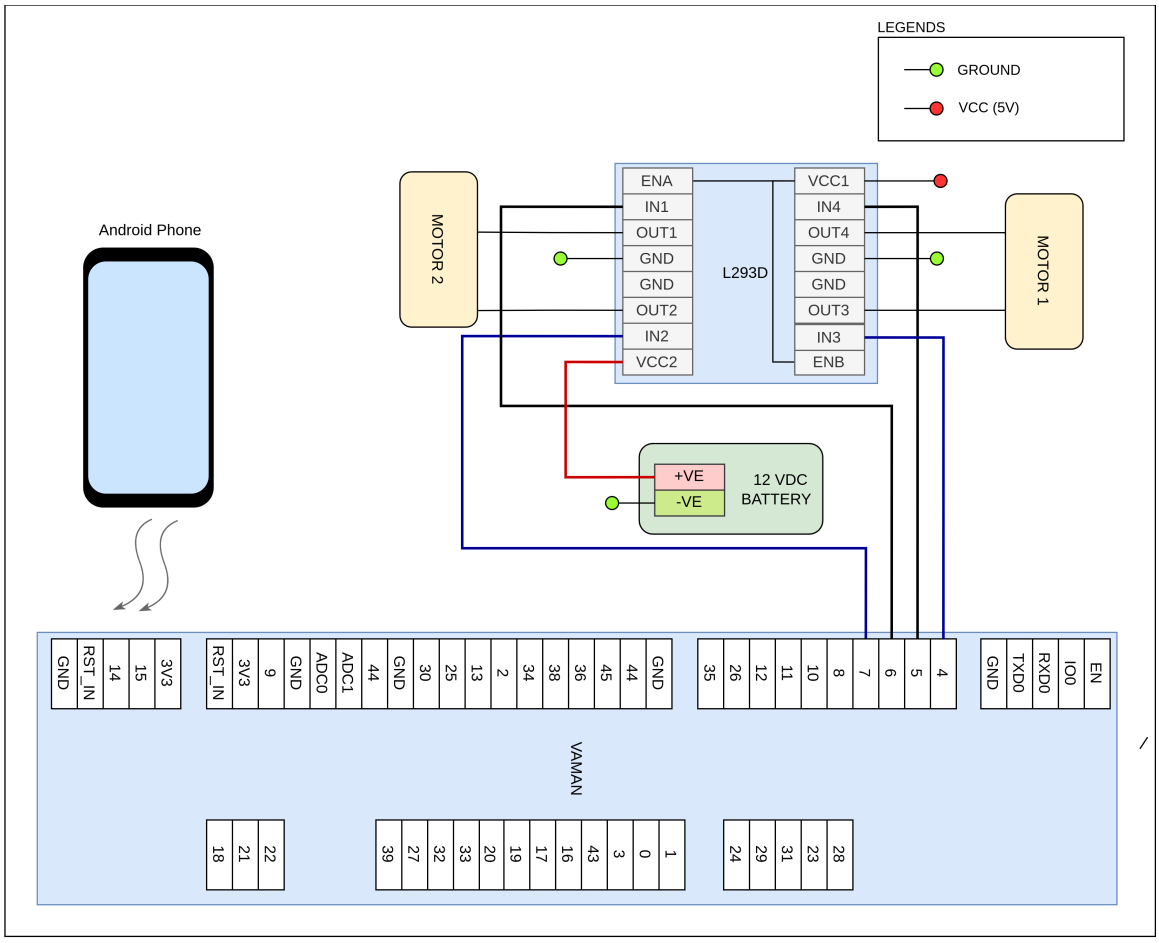


Figure 4.7: Wiring Diagram for UGV Navigation using Android phone (Vaman)

Working

- User gives navigation input using the dabble app installed on the android device. The dabble app communicates to Vaman over bluetooth. The dabble app UI is as shown in figure 4.6.
- Depending on the input received over bluetooth, the program sends suitable commands (forward, backward, left, right and stop) to the direction control pins of Motor driver IC.

4.1.5 Navigation using Speech commands

Required components/Software tools

- UGV chassis with DC motors
- ESP32 micro-controller with Type-B USB cable
- L293D Motor Driver IC
- Breadboard
- Jumper Wires
- Arduino IDE installed on system
- Android Phone

Steps

- Make the connections as per the wiring diagram (Figure 4.8).
- Go to Arduino IDE and write the following program available on the code link at 4.1.5.
- Compile and upload the program to ESP32 micro-controller using the Type-C programmable cable.
- Test whether the UGV is navigating as per the Speech command sent from the Android phone.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/ESP32/IDE/Voice_Controll_UGV
```

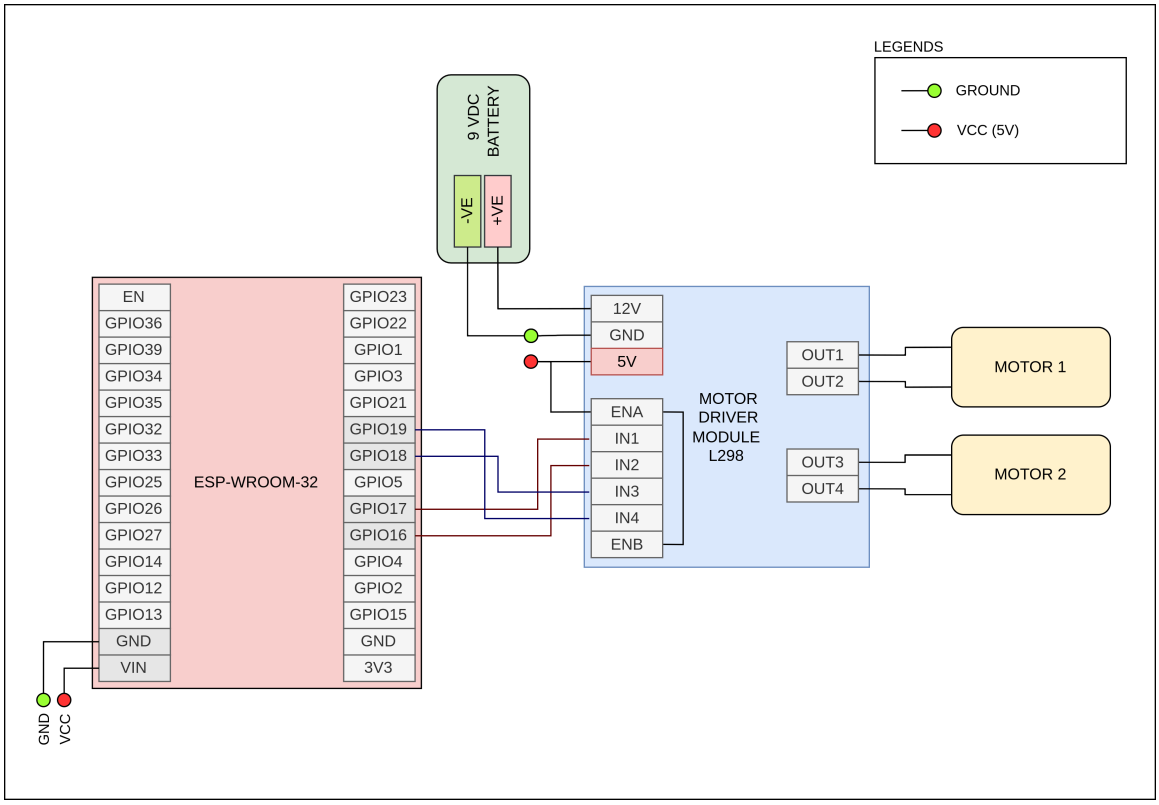


Figure 4.8: Wiring Diagram for UGV Navigation using Speech commands

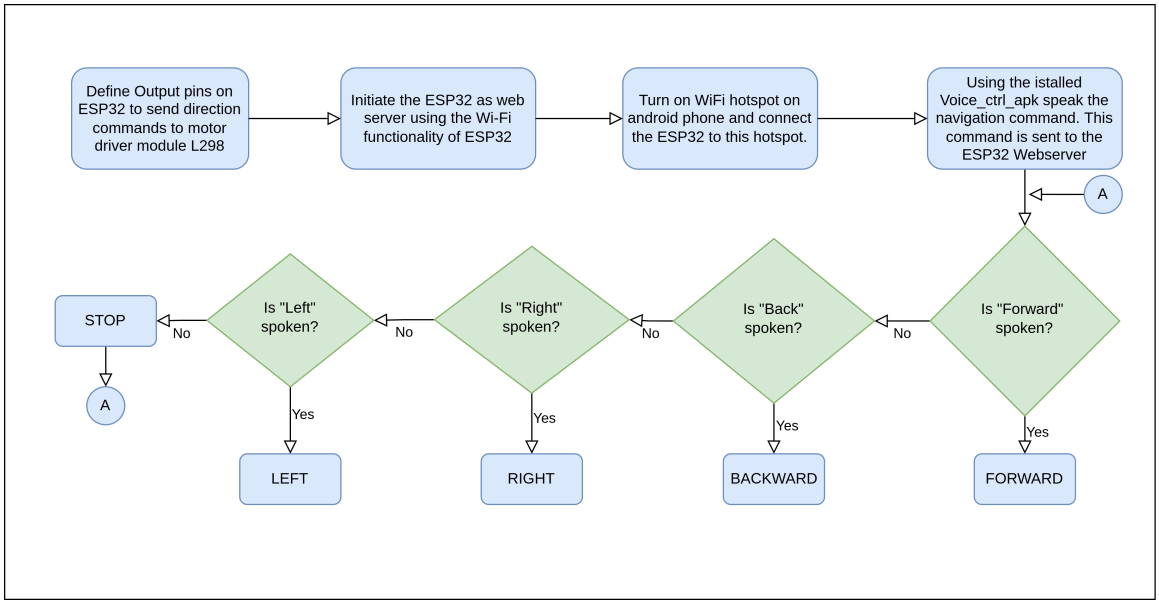
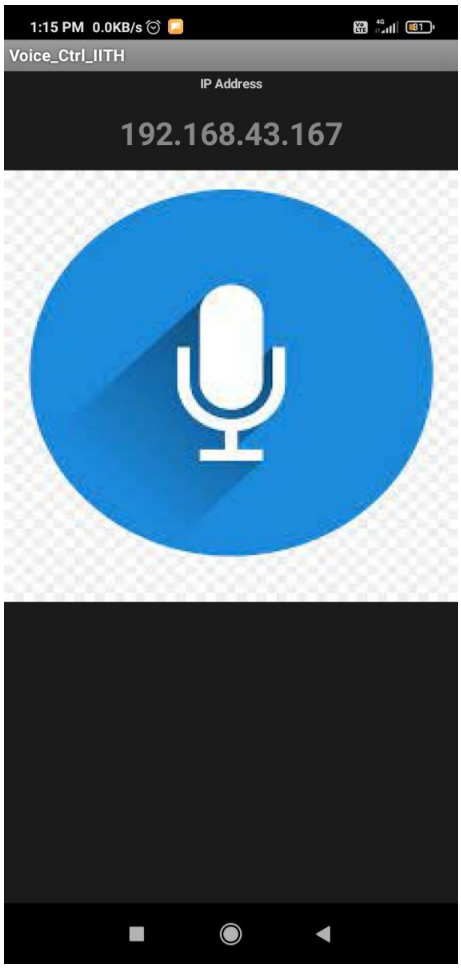


Figure 4.9: Flow Diagram for UGV Navigation using Speech commands

Working

- User first speaks into the Speech_ctrl_IITH app (Figure 4.10) installed on android phone. The speech is converted to text using Google's TTS engine working at the background.

Figure 4.10: Speech control android app created using MIT app creator



- The text is sent over Wi-Fi connection to the ESP32 web server.
- At ESP32, the program checks the command received and sends suitable commands (forward, backward, left, right and stop) to the direction control pins of Motor driver module L298.

4.1.6 Beacon Tracking using ESP32

Required components/Software tools

- UGV chassis with DC motors
- ESP32 micro-controller with Type-B USB cable
- L293D Motor Driver IC
- Breadboard

- Jumper Wires
- Arduino IDE installed on system
- Android phone used as beacon

Steps

- Make the connections as per the wiring diagram (Figure 4.11).
- Connect the ESP32 board to Laptop/PC using Type-B USB cable.
- Open the program at the code link (4.1.6) in Arduino IDE.
- From Tools menu, select suitable "Board" and "Port" for your ESP32 board.
- Compile the code by clicking on "Verify" option.
- Upload the code to ESP32 using the "Upload" option.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
UGV/ESP32/IDE/Beacon_tracking
```

Working

- Initially, ESP32 mounted on the car will read RSSI (Radio Signal Strength Indicator) levels in forward, right and left direction by suitable in-place rotation.
- Average of 20 RSSI values are taken while measuring RSSI level in a particular direction. This is done in order to read stable RSSI values.
- The car then rotates towards the direction having the highest RSSI level.
- Further, It moves forward for a certain distance towards the beacon. By repeating above steps again and again, the car navigates towards the beacon.

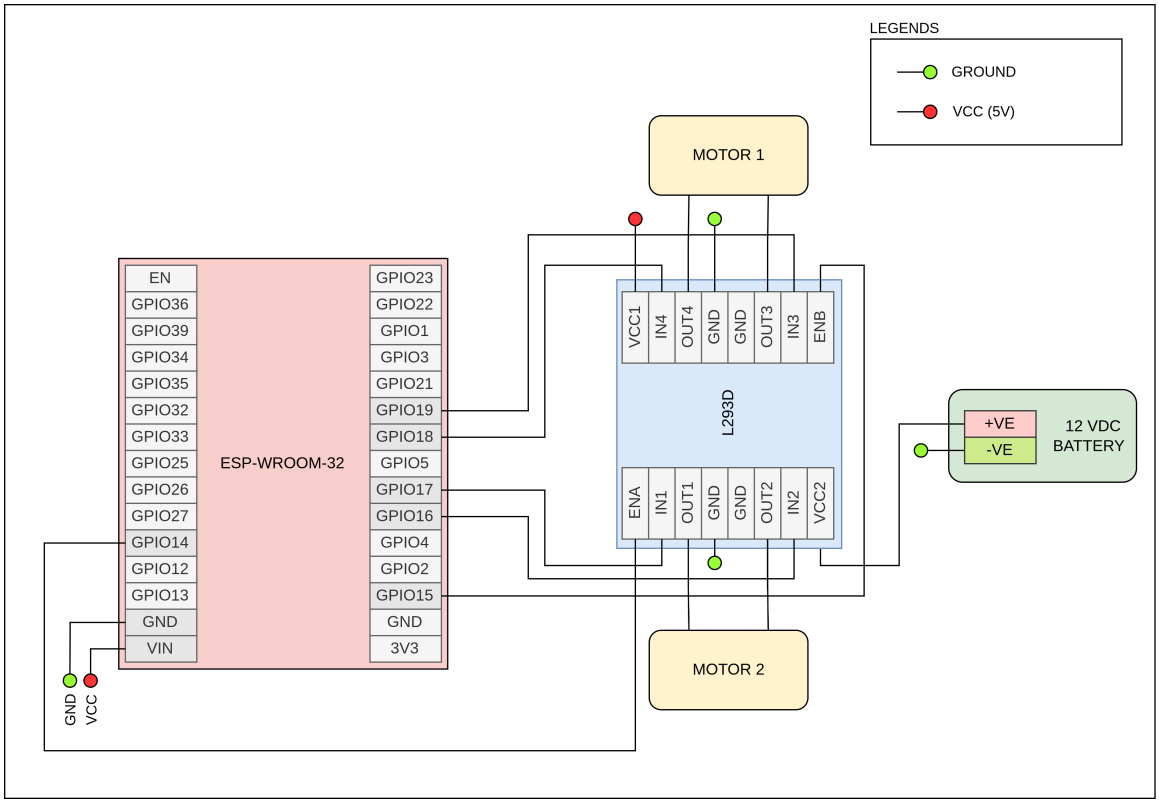


Figure 4.11: Wiring Diagram for UGV beacon tracking

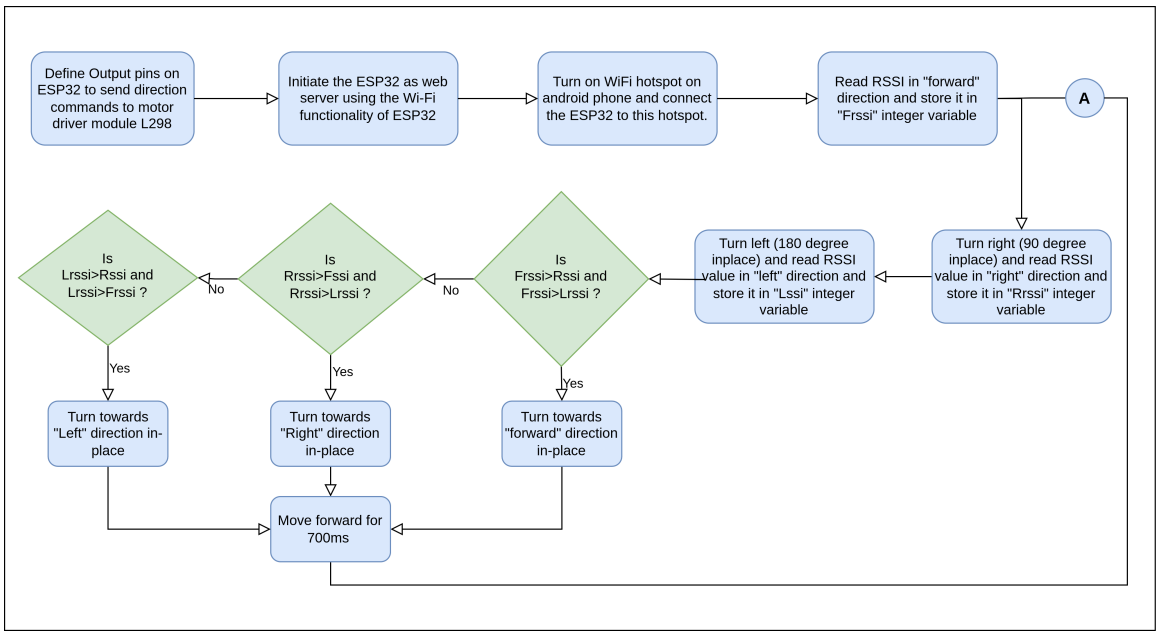


Figure 4.12: Flow Diagram for UGV beacon tracking

4.2 UAV (Unmanned Aerial Vehicle)

4.2.1 Navigation using ESP32 and Android phone

Required components/Software tools

- UAV fully assembled and configured.
- ESP32 micro-controller with Type-B USB cable for programming.
- Jumper Wires
- Arduino IDE installed on system
- Android phone with chrome web browser to access ESP32 web-server.

Steps

- Make the connections as per the wiring diagram (Figure 4.13).
- Go to Arduino IDE and write the following program available on the code link at 4.2.1.
- Compile and upload the program to ESP32 micro-controller using the Type-C programmable cable.
- Arm the drone using Yaw and Throttle slider and slowly increase the throttle slider to lift the drone vertically from the ground.
- Try navigation the UAV using the various sliders (Throttle, Yaw, Roll, pitch) on the Android phone.

Code link

```
https://github.com/sachinomdubey/Projects/blob/main/Autonomous%20Navigation/
UAV/ESP32_comm_link/pwmwebserver_esp32
```

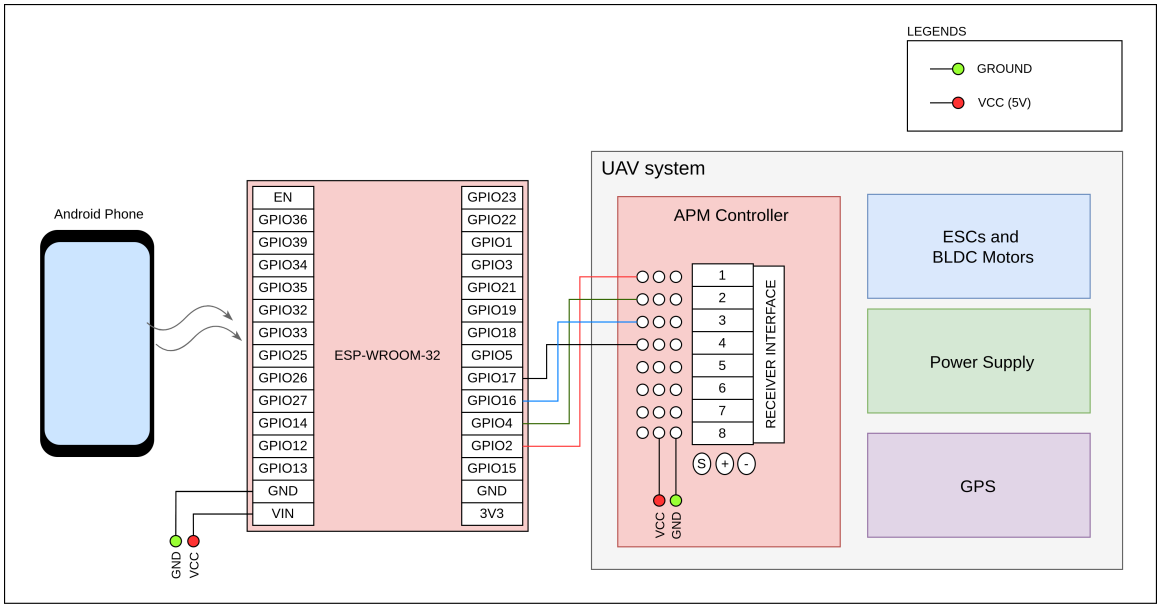


Figure 4.13: Wiring Diagram for UAV Navigation using ESP32 and Android phone

Working

- User first arms the drone using Yaw and Throttle slider (analogous to arming using fly-sky transmitter) and slowly increase the throttle slider to lift the drone vertically from the ground.
- Whenever a slider is changed by user the ESP32 reads this change and generates a PWM signal (same as generated by the Fly-sky receiver module).
- Depending on the duty-cycle of this signal the APM controller drives the four ESC modules which further controls the speech of the four BLDC motors. (Refer section 3 for details on control of speed of BLDC motors)
- Thus, the Android phone and ESP32 form a new communication link from ground operator to drone controller. The range of this link is limited by the Wi-Fi network to which the ESP32 and android phones are connected.

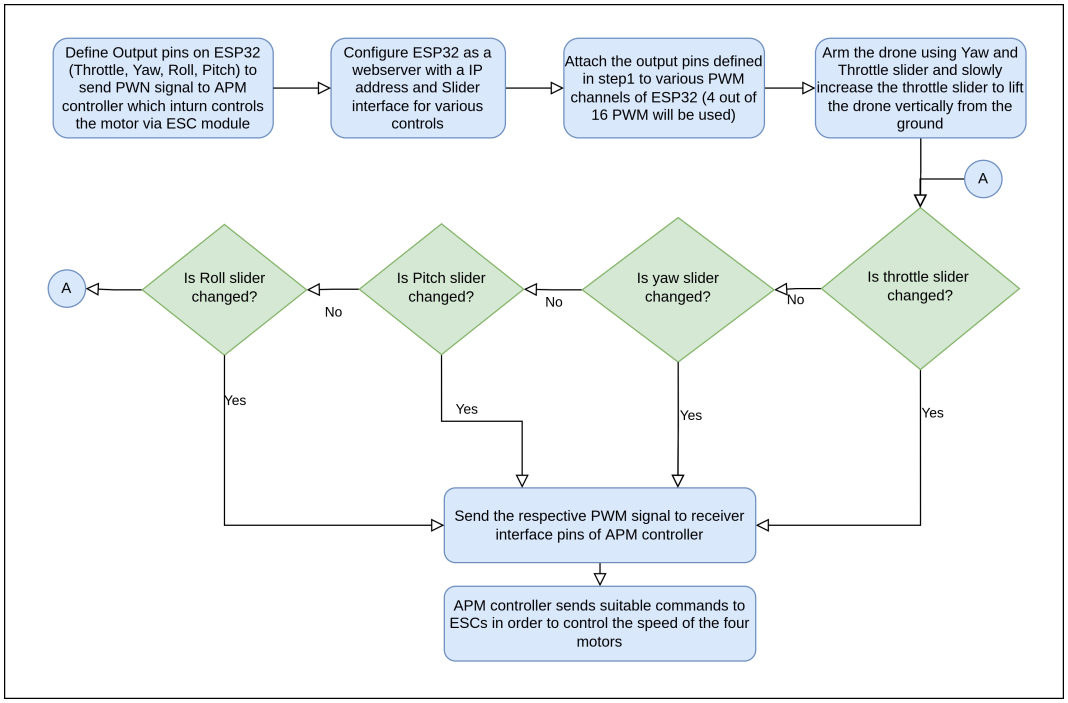


Figure 4.14: Flow Diagram for UAV Navigation using ESP32 and Android phone

4.3 Basic programs using FreeRTOS

Embedded systems uses real-time operating system. Real-time tasks are critical as timing places an important role in such systems. RTOS are made to run tasks or program with precise timings and high degree of reliability. It helps in multitasking even when the micro-controller unit has a single core to execute these tasks.

FreeRTOS is an open source RTOS which are designed, but not limited to run on small micro-controllers (8/16 bit). With basic knowledge of RTOS, we can use FreeRTOS as there is a lot of documentation available for it.

4.3.1 Running two tasks simultaneously using FreeRTOS (Arduino)

In this program, we have run two different tasks in parallel on Arduino Uno using FreeRTOS. The two tasks are as follow:

1. Blinking LED connected at pin 13 of Arduino.

2. Speeding up and slowing down a DC motor continuously in a loop.

Required components/Software tools

- Arduino Uno R3 board
- LED connected between pin 13 and GND.
- L293D motor driver IC
- 12V DC Battery for motor
- Type-B cable for powering Arduino
- Jumper Wires
- Arduino IDE installed on system

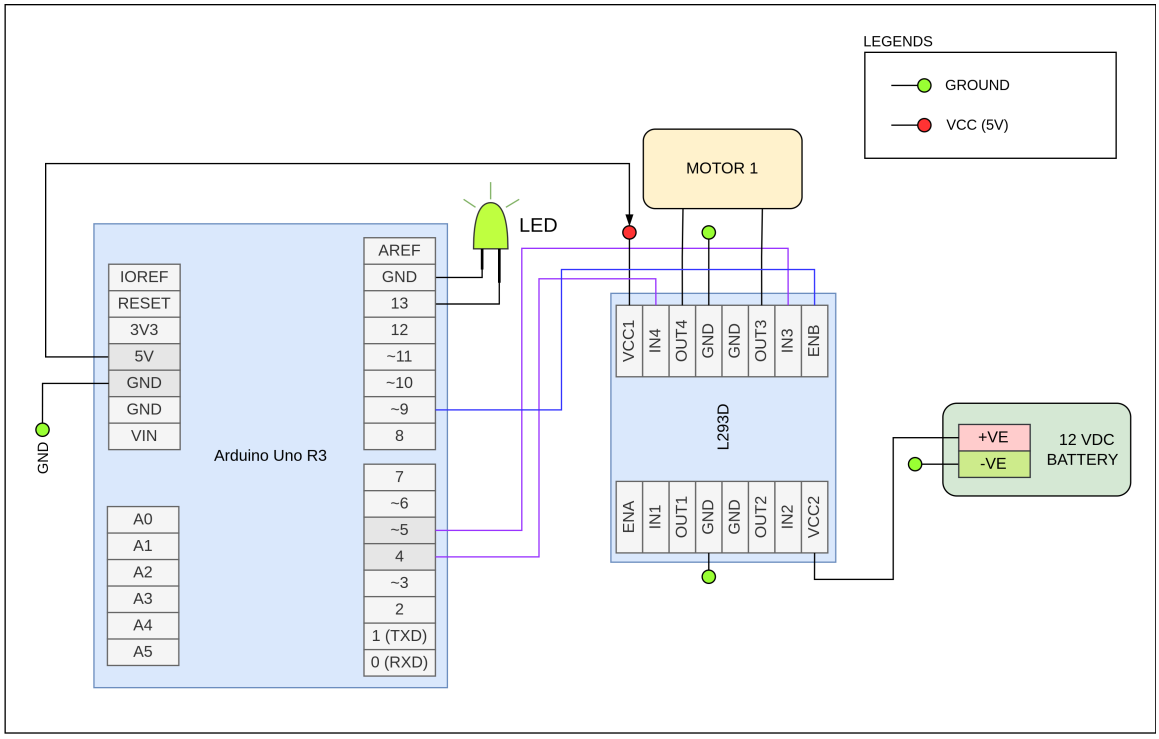


Figure 4.15: Wiring diagram for running two tasks simultaneously on Arduino Uno.

Steps

- Make the connections as per the wiring diagram (Figure 4.15).
- Connect the Arduino Uno to Laptop/PC using Type-B USB cable.
- Open the program at the code link (4.3.1) in Arduino IDE.

- From Tools menu, select suitable "Board" and "Port" for your Arduino board.
- Compile the code by clicking on "Verify" option.
- Upload the code to ESP32 using the "Upload" option.
- Verify that the LED blinks and the motor also changes speed simultaneously.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
FreeRTOS/Arduino_Blink_MotorControl/Code
```

Working

- First the FreeRTOS library is included using the below line. This header contains all the functions required to create, schedule and run tasks in parallel.

```
#include <Arduino_FreeRTOS.h>
```

- For creating task, **xTaskCreate()** API is called in setup function with certain parameters/arguments.

```
xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName,
uint16_t usStackDepth, void *pvParameters, UBaseType_t uxPriority,
TaskHandle_t *pxCreatedTask );
```

- There are 6 arguments that should be passed while creating any task. Let's see what these arguments are
  - **pvTaskCode**: It is simply a pointer to the function that implements the task (in effect, just the name of the function).
  - **pcName**: A descriptive name for the task. This is not used by FreeRTOS. It is included purely for debugging purposes.
  - **usStackDepth**: Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32-bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes) in RAM. Use this wisely because Arduino Uno has only 2Kbytes of RAM.

- **pvParameters**: Task input parameter (can be NULL).
- **uxPriority**: Priority of the task (0 is the lowest priority).
- **pxCreatedTask**: It can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task (can be NULL).

- Example of task creation

```
xTaskCreate(task1,"task1",128,NULL,1,NULL);
xTaskCreate(task2,"task2",128,NULL,2,NULL);
```

- After creating the task, start the scheduler in a void setup using **vTaskStartScheduler()** API.
- **Void loop()** function will remain empty as we don't want to run any task manually and infinitely. Because task execution is now handled by Scheduler.
- Next we define our tasks using below format and write logic for it. Everything else is taken by FreeRTOS library function and we can observe the tasks running in parallel on executing the written program.

```
void task1(void *pvParameters)
{
    while(1) {
        ..
        ..//your logic
    }
}
```

4.3.2 Running two tasks simultaneously using FreeRTOS (Vaman)

In this program, we have run two different tasks in parallel on Vaman using FreeRTOS. The two tasks are as follow:

1. Blinking LED connected at pin 13 of Arduino.
2. Speeding up and slowing down a DC motor continuously in a loop.

Required components/Software tools

- Vaman (pygmy BB4) development board
- LED connected between pin 12 of Vaman and GND.
- L293D motor driver IC
- 12V DC Battery for motor
- Type-B cable for powering and programming Vaman
- Jumper Wires
- QORC SDK installed on the Linux system along with required support tools.

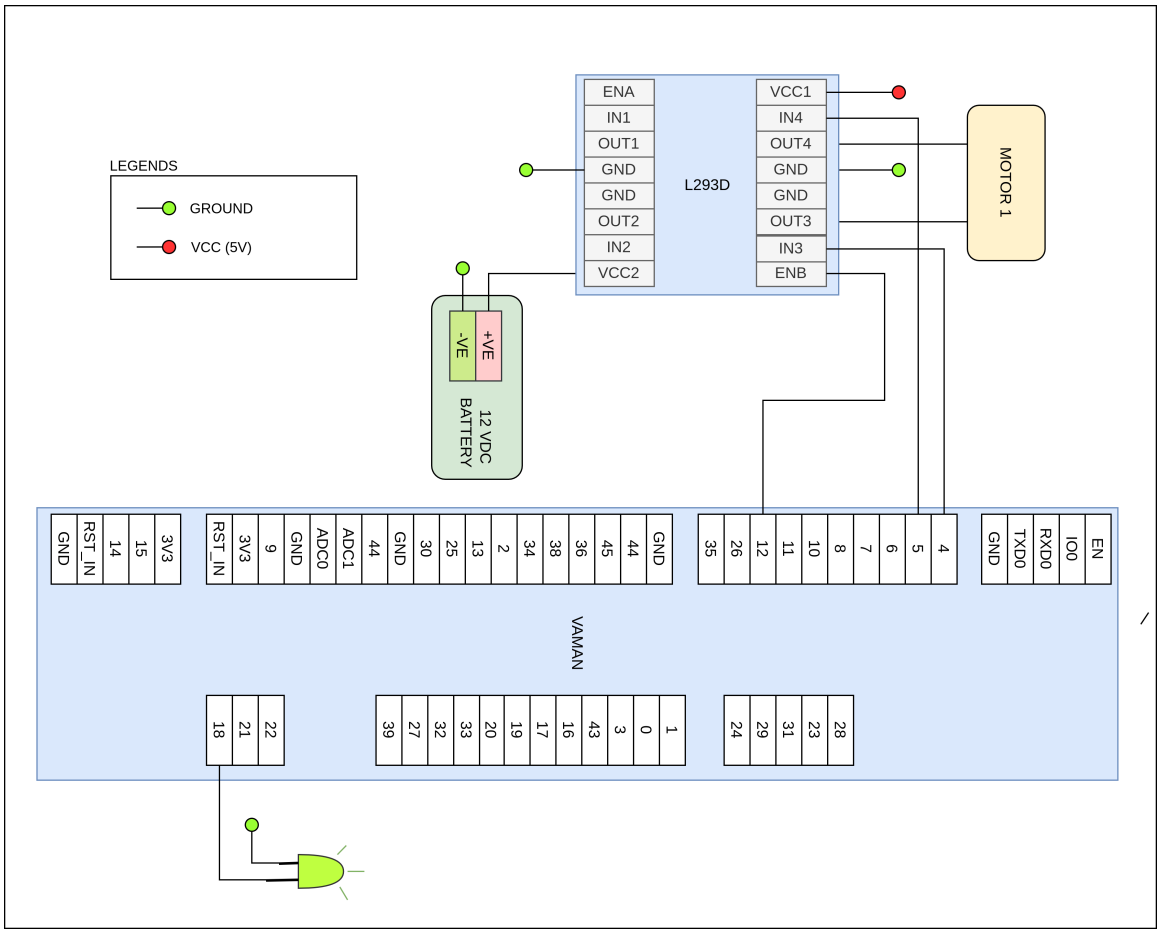


Figure 4.16: Wiring diagram for running two tasks simultaneously on Vaman.

Steps

- Make the connections as per the wiring diagram (Figure 4.16).
- Connect the Vaman board to Laptop/PC using Type-B USB cable.
- Run "make" command in "GCC_project" folder to generate bin file for the "main.c" file at the code link (4.3.2).
- Flash the generated bin file on Vaman using the tinyfpga-programmer-gui.py tool.
- Verify that the LED blinks and the motor also changes speed simultaneously.

Code link

```
https://github.com/sachinomdubey/Projects/tree/main/Autonomous%20Navigation/
FreeRTOS/Vaman_Blink_MotorControl/src
```

Working

- First the FreeRTOS library is included using the below line. This header contains all the functions required to create, schedule and run tasks in parallel.

```
#include <FreeRTOS.h>
```

- For creating task, **xTaskCreate()** API is called in setup function with certain parameters/arguments.

```
xTaskCreate( TaskFunction_t pvTaskCode, const char * const pcName,
uint16_t usStackDepth, void *pvParameters, UBaseType_t uxPriority,
TaskHandle_t *pxCreatedTask );
```

- There are 6 arguments that should be passed while creating any task. Let's see what these arguments are
  - **pvTaskCode**: It is simply a pointer to the function that implements the task (in effect, just the name of the function).
  - **pcName**: A descriptive name for the task. This is not used by FreeRTOS. It is included purely for debugging purposes.

- **usStackDepth**: Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The value specifies the number of words the stack can hold, not the number of bytes. For example, if the stack is 32-bits wide and usStackDepth is passed in as 100, then 400 bytes of stack space will be allocated (100 * 4 bytes) in RAM. Use this wisely because Arduino Uno has only 2Kbytes of RAM.
- **pvParameters**: Task input parameter (can be NULL).
- **uxPriority**: Priority of the task (0 is the lowest priority).
- **pxCreatedTask**: It can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task (can be NULL).

- Example of task creation

```
xTaskCreate(task1,"task1",128,NULL,1,NULL);
xTaskCreate(task2,"task2",128,NULL,2,NULL);
```

- After creating the task, start the scheduler in a main function using **vTaskStartScheduler()** API.
- **main()** function will remain empty as we don't want to run any task manually and infinitely. Because task execution is now handled by Scheduler.
- Next we define our tasks using below format and write logic for it. Everything else is taken by FreeRTOS library function and we can observe the tasks running in parallel on executing the written program.

```
void task1(void *pvParameters)
{
    while(1) {
        ..
        ..//your logic
    }
}
```