

Navigation and Communication for 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

Approval Sheet

This thesis entitled Navigation and Communication for UGV/UAV by Sachinkumar Omprakash Dubey is approved for the degree of M.Tech. in Communication & Signal Processing from IIT Hyderabad

(Dr. G V V Sharma) Guide

Dept. of Electrical Engineering

IIT Hyderabad

Abstract

This report is intended to serve as a concise and to-the-point report of my work on navigation and communication for UGV (Unmanned Ground Vehicles) and UAV (Unmanned Aerial Vehicles).

This thesis report titled "Navigation and Communication of UGV/UAV" consists of two parts:

- The navigation part of this thesis includes utilization of various controllers to implement applications on UGV/UAV hardware kit.
- The communication part explores SATCOM for 5G which can be utilized to communicate with UAV or UGV. It also briefly examines the use of NB-IoT.

UGV and UAV hardware kits are ideal low-cost prototype systems for testing software before scaling it up and putting it on a real ground vehicle or a more complicated UAV system.

The communication part introduces SATCOM and its need in terrestrial cellular network. It also describes various application where SATCOM can be utilized in UAV communication. This report also includes an experimental setup to demonstrate working of SATCOM with 5G network. As part of testing, A video streaming test is perform between the User Equipment (UE) and the 5G core network (5GC).

Contents

Approval Sheet	ii
Abstract	iii
1 Introduction	1
1.1 Controllers and their specifications	1
1.1.1 ESP32 Micro-controller	1
1.1.2 Vaman Pygmy BB4	2
1.1.3 Arduino Uno	2
1.1.4 Raspberry Pi 3B	3
2 Basics of Motor control	5
2.1 PWM speed control	5
2.2 Brushed DC motor	6
2.3 Brush-less DC motor	8
3 Serial communication protocols	10
3.1 UART (Universal Asynchronous Receiver/Transmitter)	10
3.2 SPI (Serial Peripheral Interface)	11
3.3 I2C Protocol	12
4 ESP32 Based Applications	13
4.1 UGV (Unmanned Ground Vehicle)	13
4.1.1 Navigation using Fly-sky Transmitter and Receiver	13
4.1.2 Navigation using Android phone	15
4.1.3 Navigation using Speech commands	18
4.1.4 Beacon Tracking using ESP32	21
4.2 UAV (Unmanned Aerial Vehicle)	23

4.2.1	Navigation using ESP32 and Android phone	23
4.3	Running two tasks simultaneously using FreeRTOS (Arduino)	25
5	Vaman Based Applications	29
5.1	UGV (Unmanned Ground Vehicle)	29
5.1.1	Navigation using Fly-sky Transmitter and Receiver	29
5.1.2	Navigation using Android phone	31
5.2	Simultaneous execution of tasks using FreeRTOS	33
6	SATCOM for UAV Communication	37
6.1	SATCOM Introduction	37
6.1.1	Application of SATCOM based UAV	39
6.2	Set-up for Demonstration of SATCOM	40
6.2.1	Free-5GC Environment	40
6.3	Video Streaming using the setup	41
6.3.1	Compilation and Execution at gNB	41
6.3.2	Compilation and Execution at UE	42
6.3.3	Emulating Satellite round-trip delay	43
6.3.4	Checking the added latency using ping	44
6.3.5	5G-NR call flow	45
6.3.6	Streaming video using the setup	46
6.3.7	Video stream result	47
7	Conclusion and Future Directions	48

List of Figures

1.1	ESP-WROOM-32 Dev Kit	1
1.2	Vaman (Pygmy BB4)	2
1.3	Arduino Uno R3	3
1.4	Raspberry Pi 3B	3
2.1	PWM speed control	6
2.2	Dual motor driver module (L298N)	7
2.3	Typical connection for motor driver module	7
2.4	Inrunner BLDC and Outrunner BLDC motors	8
2.5	6 Pole and 12 Pole BLDC motors	8
2.6	Connection between the ESC and BLDC motor	9
3.1	Data format and connection between two devices in UART interface	10
3.2	SPI point-to-point connections	11
3.3	SPI connections for 3 slave devices	12
3.4	I2C connections for multiple master and multiple slave devices	12
4.1	Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (ESP32)	14
4.2	Flow Diagram for UGV Navigation using Fly-sky transmitter & receiver	14
4.3	Wiring Diagram for UGV Navigation using Android phone (ESP32)	16
4.4	Flow Diagram for UGV Navigation using Android phone	17
4.5	Dabble app user interface	17
4.6	Wiring Diagram for UGV Navigation using Speech commands	19
4.7	Flow Diagram for UGV Navigation using Speech commands	19
4.8	Speech control android app created using MIT app creator	20
4.9	Wiring Diagram for UGV beacon tracking	22

4.10	Flow Diagram for UGV beacon tracking	22
4.11	Wiring Diagram for UAV Navigation using ESP32 and Android phone	24
4.12	Flow Diagram for UAV Navigation using ESP32 and Android phone	25
4.13	Wiring diagram for running two tasks simultaneously on Arduino Uno.	26
5.1	Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (Vaman)	30
5.2	Wiring Diagram for UGV Navigation using Android phone (Vaman)	32
5.3	Wiring diagram for running two tasks simultaneously on Vaman.	34
6.1	Conventional 5G-NR system	37
6.2	Transparent satellite based NG-RAN architecture	38
6.3	Set-up for Demonstration of SATCOM	40
6.4	Free-5GC Environment	41
6.5	Ping before adding delay	44
6.6	Ping after adding delay	44
6.7	5G-NR call flow	45
6.8	Viewing video stream at the destination	46

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 provides 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, μ SD 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.

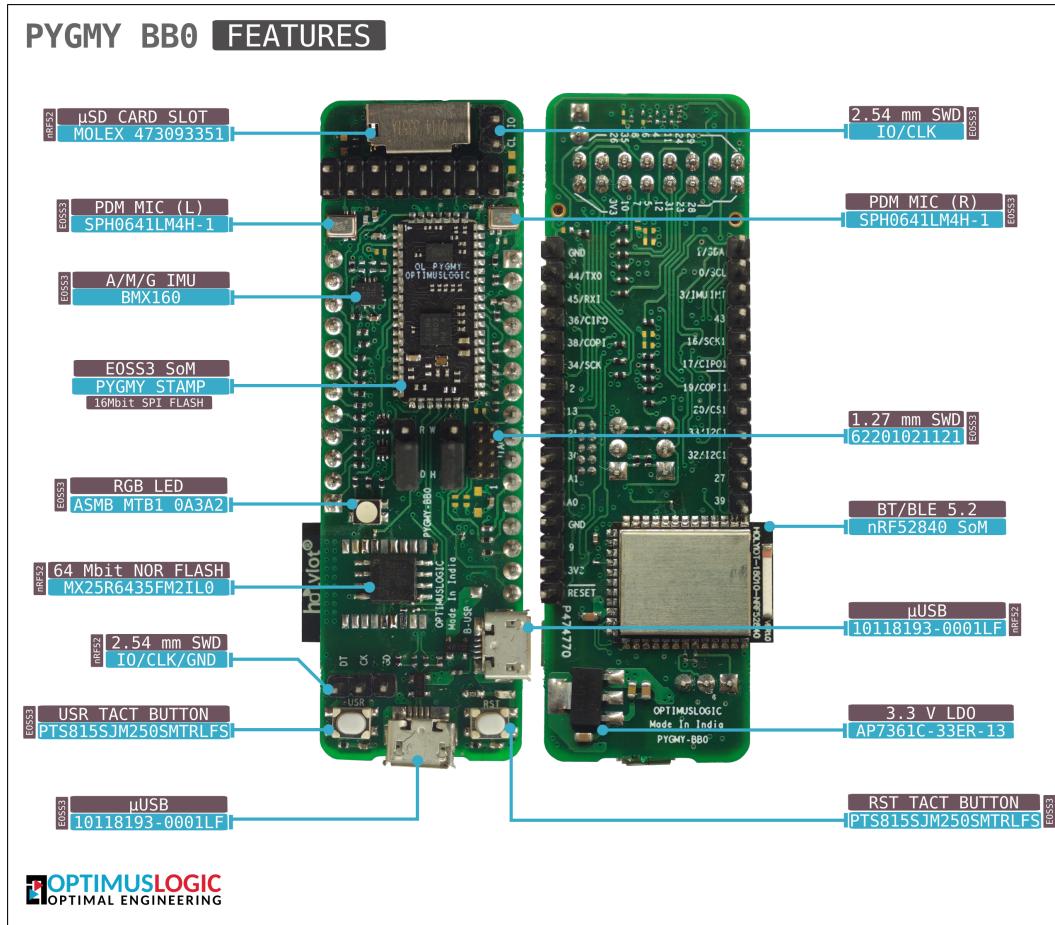


Figure 1.2: Vaman (Pygmy BB4)

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

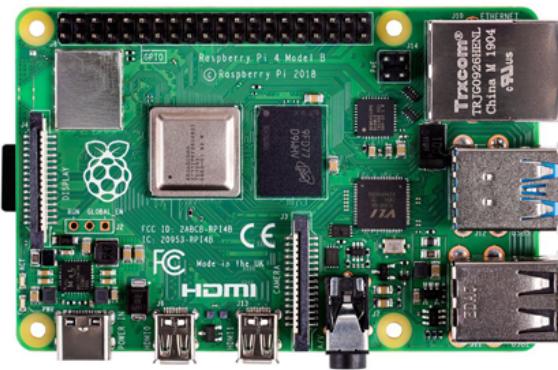


Figure 1.4: Raspberry Pi 3B

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 DS1 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. The below equation (2.1) gives the relation between the Duty cycle (D) and the average voltage:

$$\begin{aligned} V_{dc} &= \frac{1}{T} \int_0^T v_{PWM}(t) dt \\ &= \frac{1}{T} \left(\int_0^{DT} v_{\max} dt + \int_{DT}^T v_{\min} dt \right) \\ &= \frac{1}{T} (D \cdot T \cdot v_{\max} + T (1 - D) v_{\min}) \\ &= D \cdot v_{\max} + (1 - D) v_{\min} \end{aligned} \tag{2.1}$$

- Larger the duty-cycle, the faster the motor will rotate. Similarly, shorter the duty-cycle, the slower the motor will rotate.

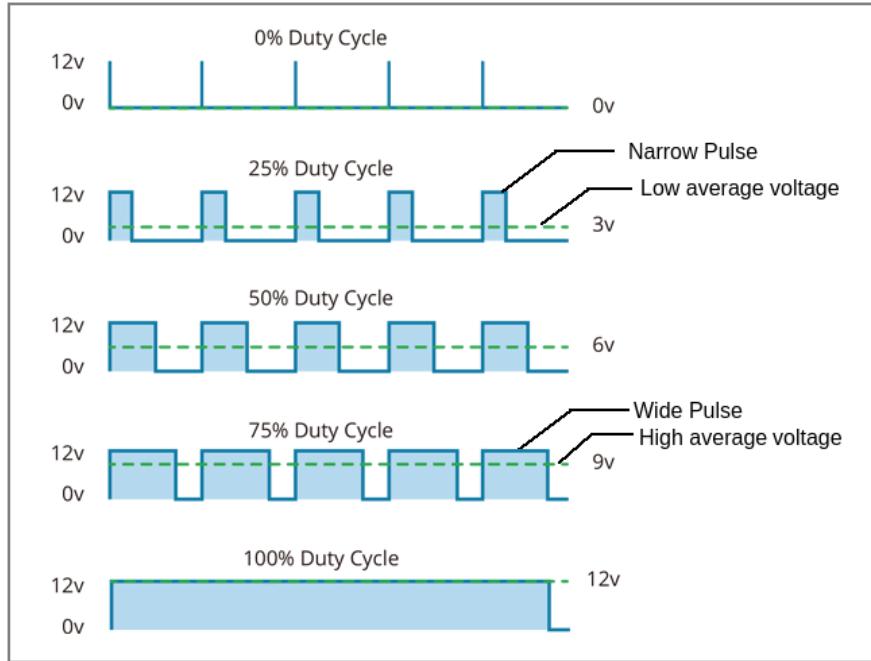


Figure 2.1: PWM speed control

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.
- 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

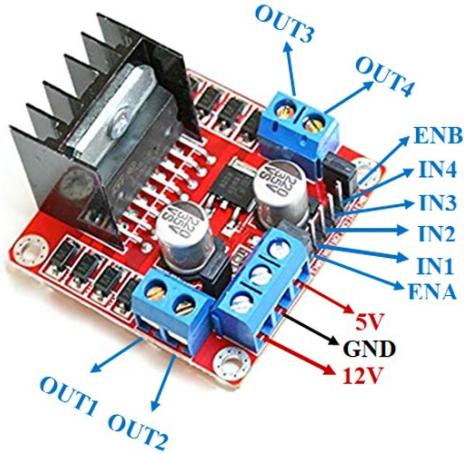


Figure 2.2: Dual motor driver module (L298N)

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.

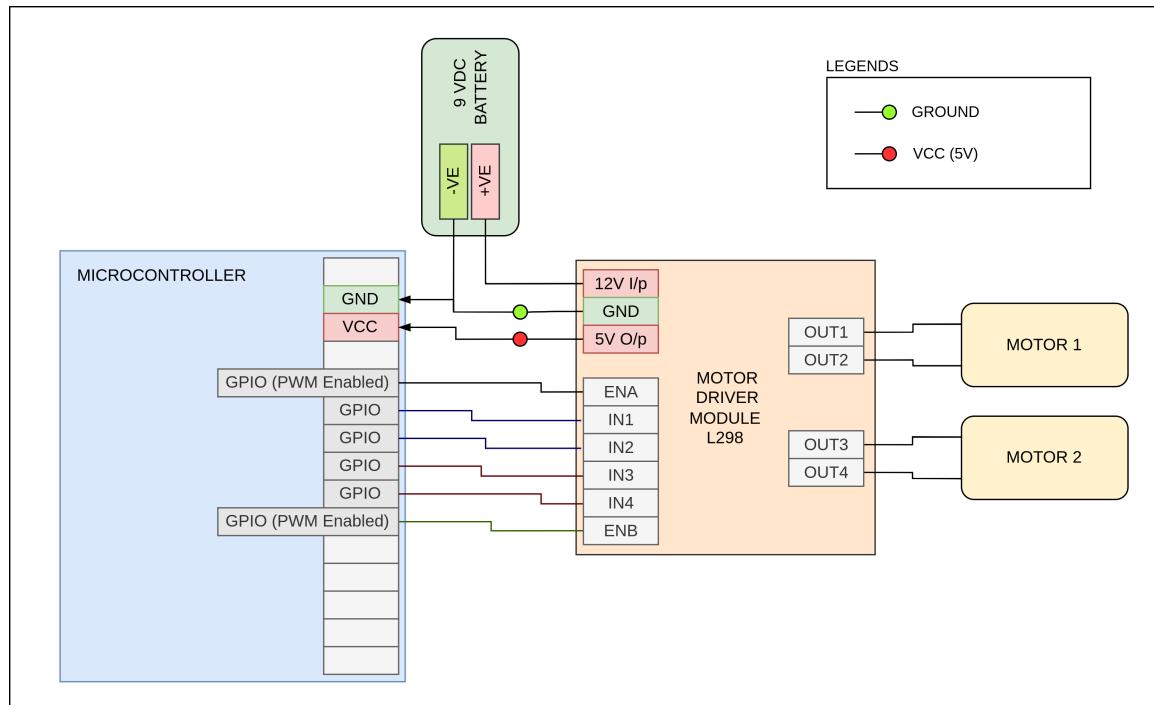


Figure 2.3: Typical connection for motor driver module

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.

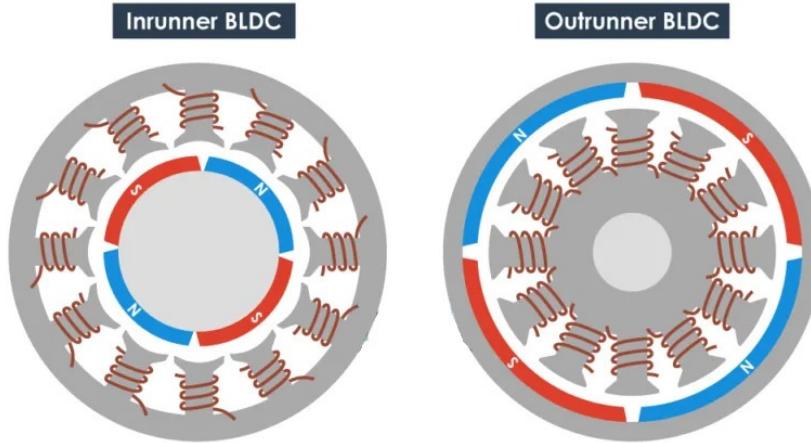


Figure 2.4: Inrunner BLDC and Outrunner BLDC motors

Generally the opposite coils of stator wined 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.

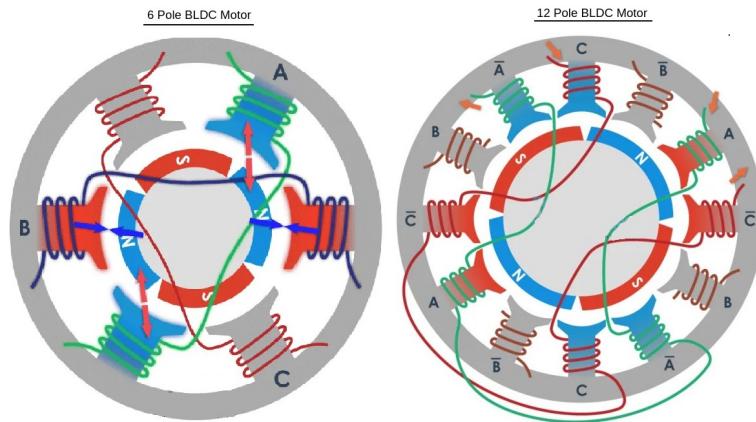


Figure 2.5: 6 Pole and 12 Pole BLDC motors

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

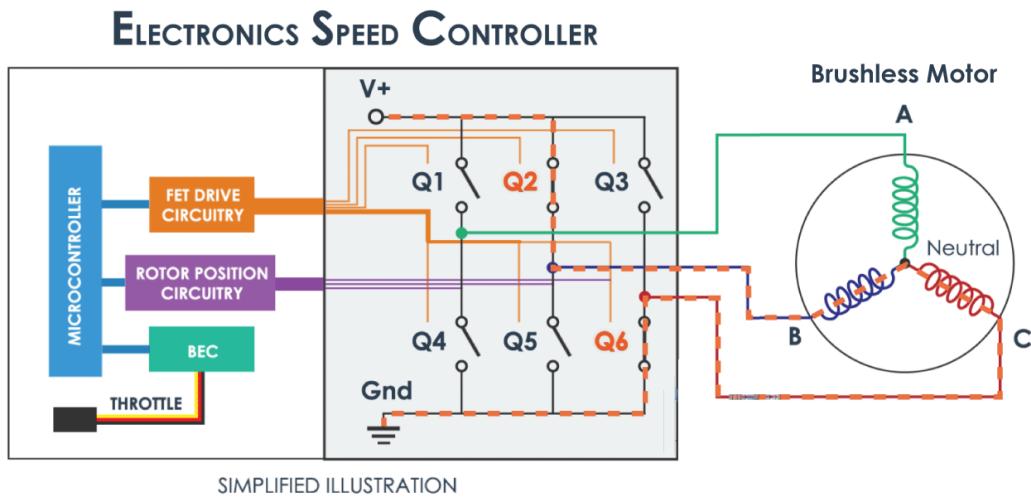


Figure 2.6: Connection between the ESC and BLDC motor

Chapter 3

Serial communication protocols

3.1 UART (Universal Asynchronous Receiver/Transmitter)

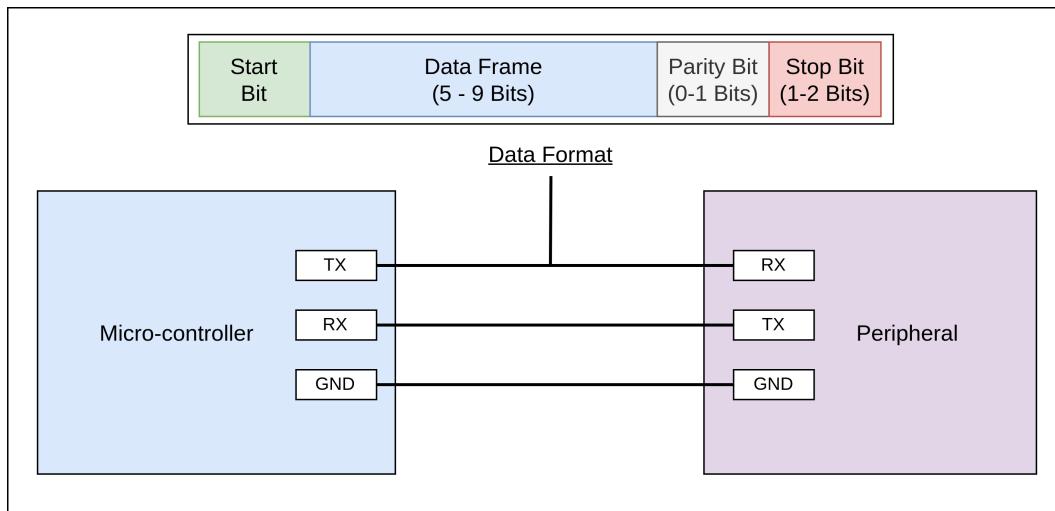


Figure 3.1: Data format and connection between two devices in UART interface

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)

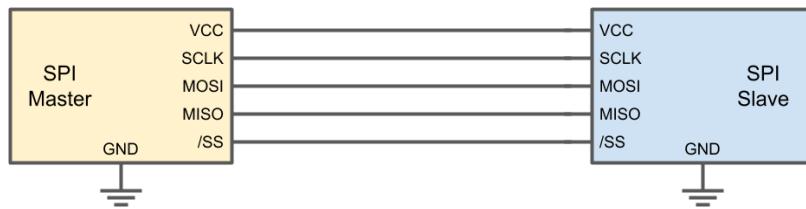


Figure 3.2: SPI point-to-point connections

- 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;
 - A slave select signal for each slave, SS_n, 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.

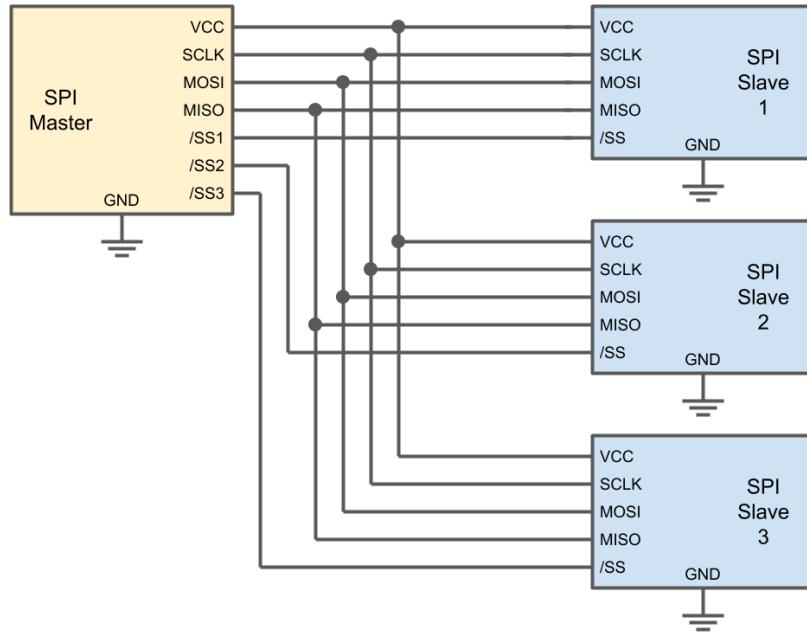


Figure 3.3: SPI connections for 3 slave devices

3.3 I2C Protocol

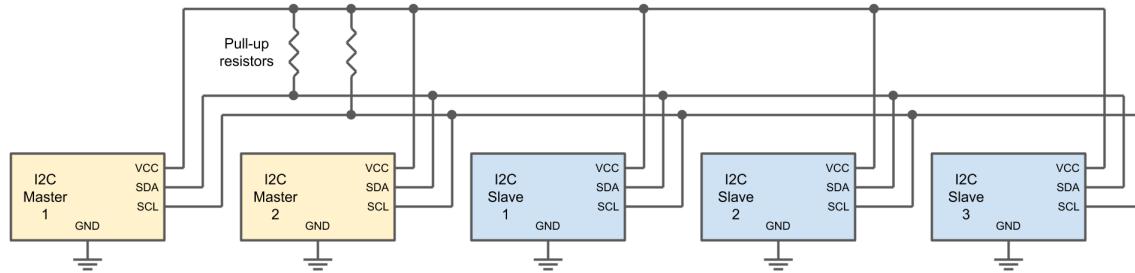


Figure 3.4: I2C connections for multiple master and multiple slave devices

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.
- 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).

Chapter 4

ESP32 Based Applications

4.1 UGV (Unmanned Ground Vehicle)

4.1.1 Navigation using Fly-sky Transmitter and Receiver

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

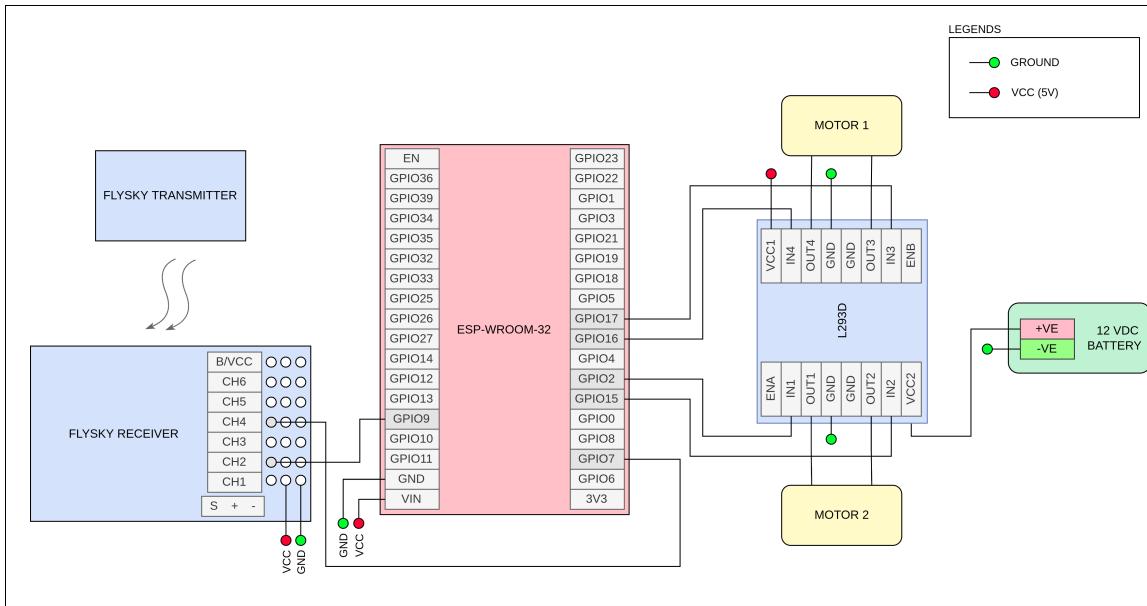


Figure 4.1: Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (ESP32)

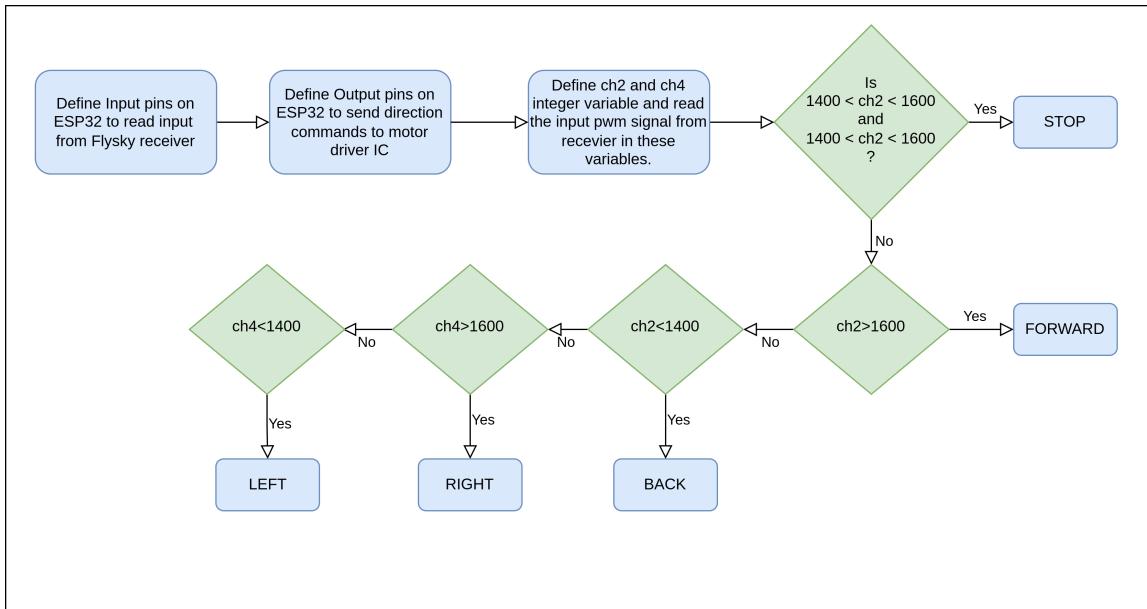


Figure 4.2: Flow Diagram for UGV Navigation using Fly-sky transmitter & receiver

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.

4.1.2 Navigation using Android phone

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 4.3).
- Go to PlatformIO and execute the "main.cpp" file given at code link at 4.1.2.
- 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

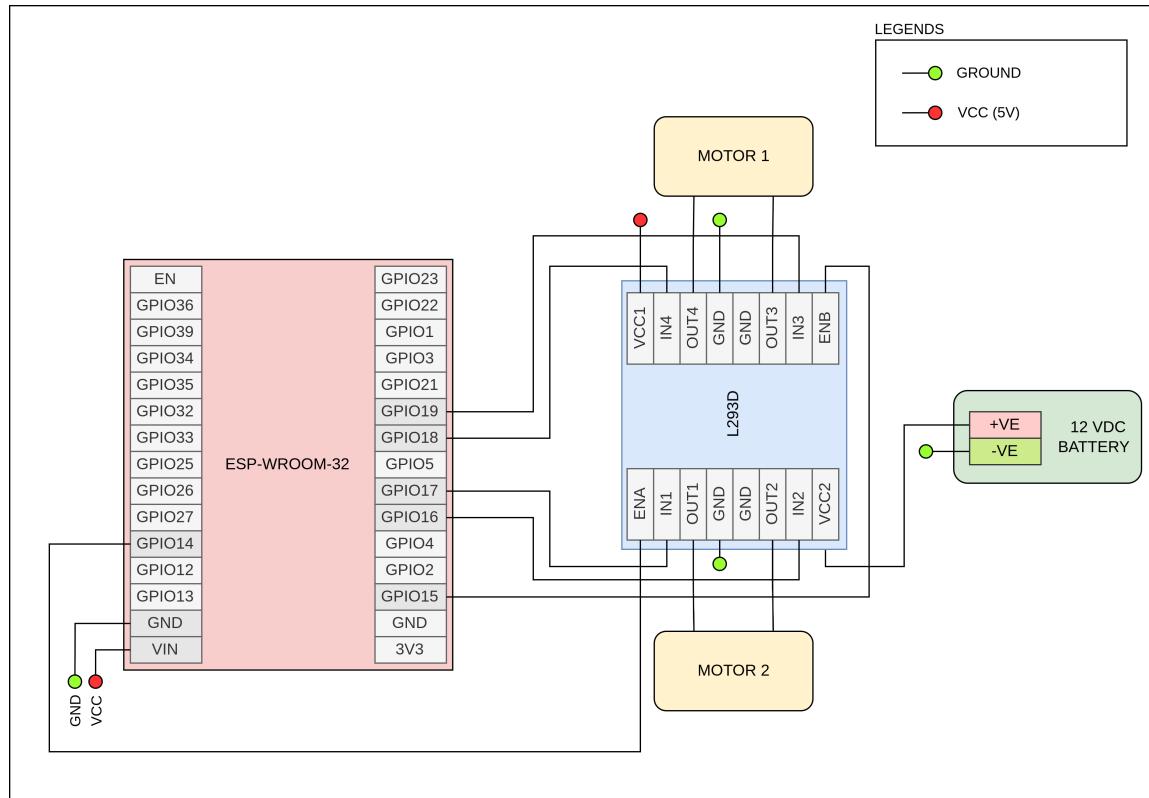


Figure 4.3: Wiring Diagram for UGV Navigation using Android phone (ESP32)

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 4.5.
- 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.

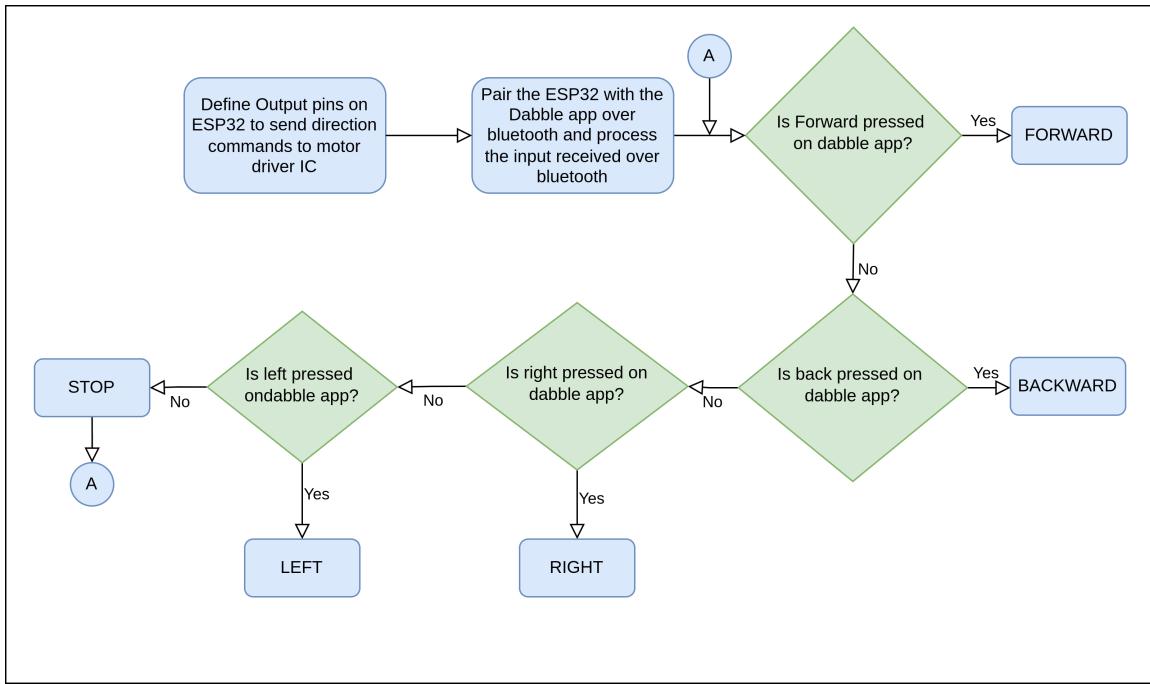


Figure 4.4: Flow Diagram for UGV Navigation using Android phone

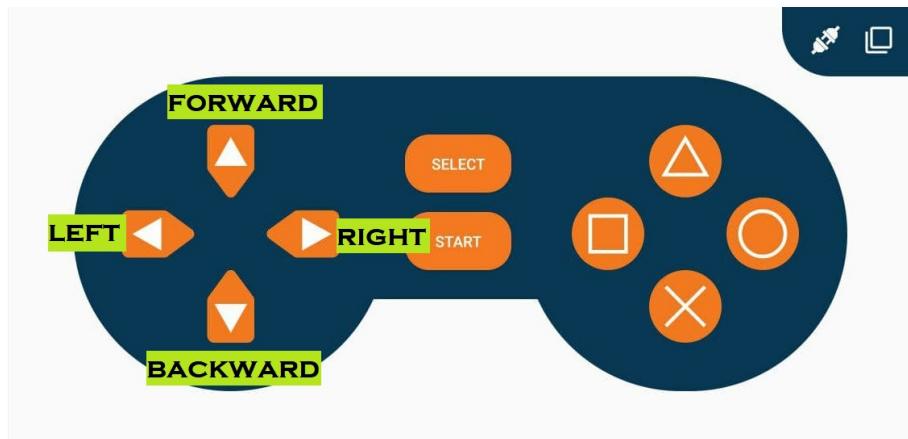


Figure 4.5: Dabble app user interface

4.1.3 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 4.6).
- Go to Arduino IDE and write the following program available on the code link at 4.1.3.
- 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\_Control1\_UGV
```

Working

- User first speaks into the Speech_ctrl_IITH app (Figure 4.8) installed on android phone. The speech is converted to text using Google's TTS engine working at the background.
- The text is sent over Wi-Fi connection to the ESP32 web server.

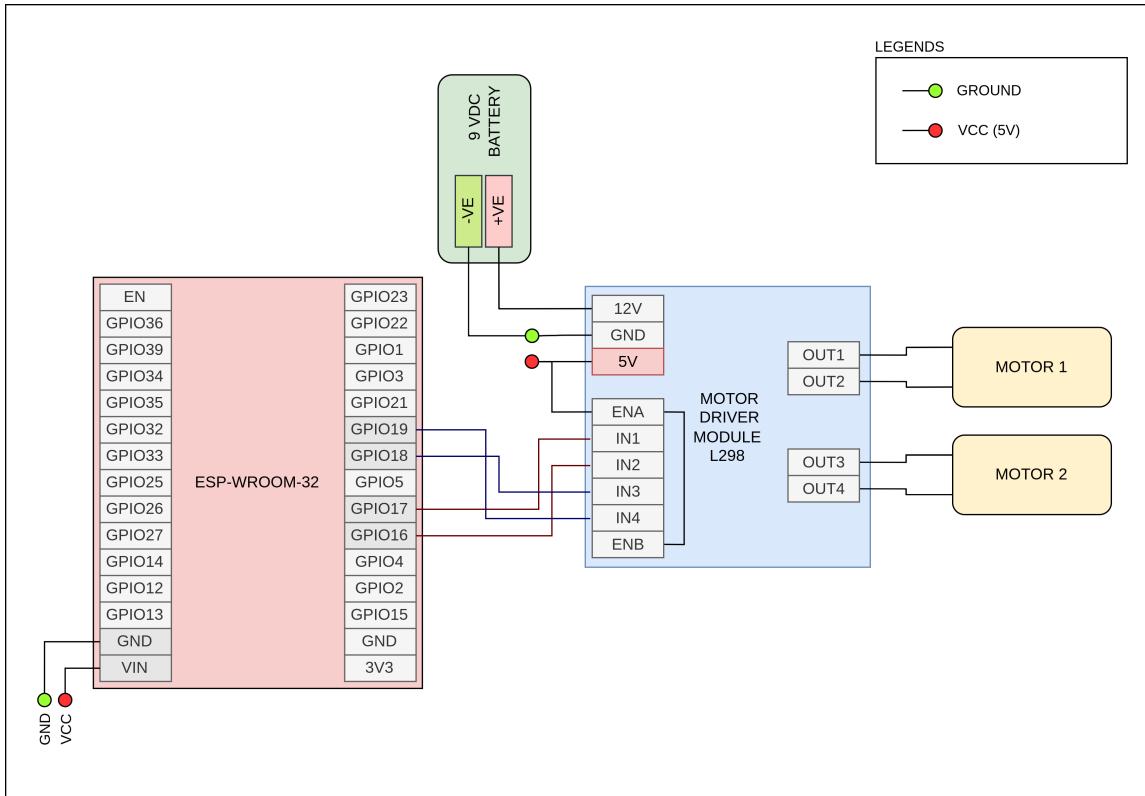


Figure 4.6: Wiring Diagram for UGV Navigation using Speech commands

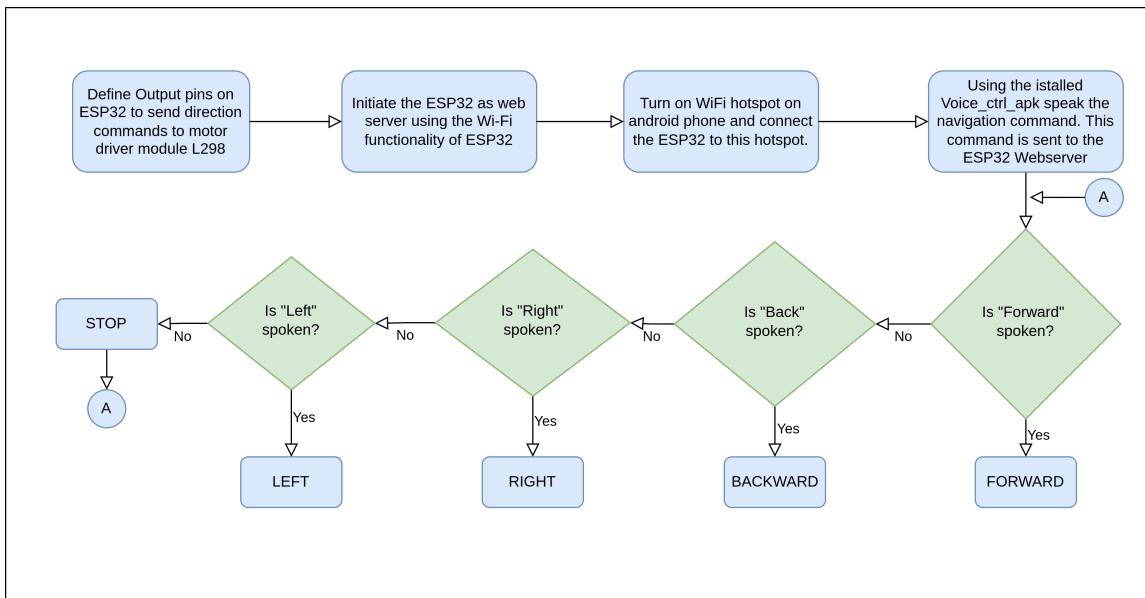


Figure 4.7: Flow Diagram for UGV Navigation using Speech commands

- 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 L298N.

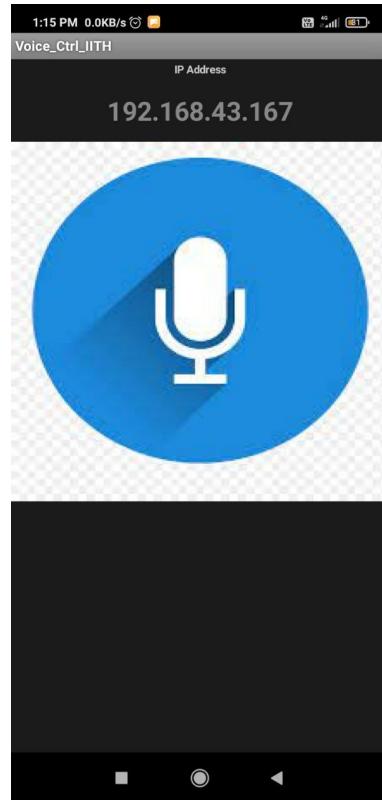


Figure 4.8: Speech control android app created using MIT app creator

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

Steps

- Make the connections as per the wiring diagram (Figure 4.9).
- Connect the ESP32 board to Laptop/PC using Type-B USB cable.
- Open the program at the code link (4.1.4) 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.

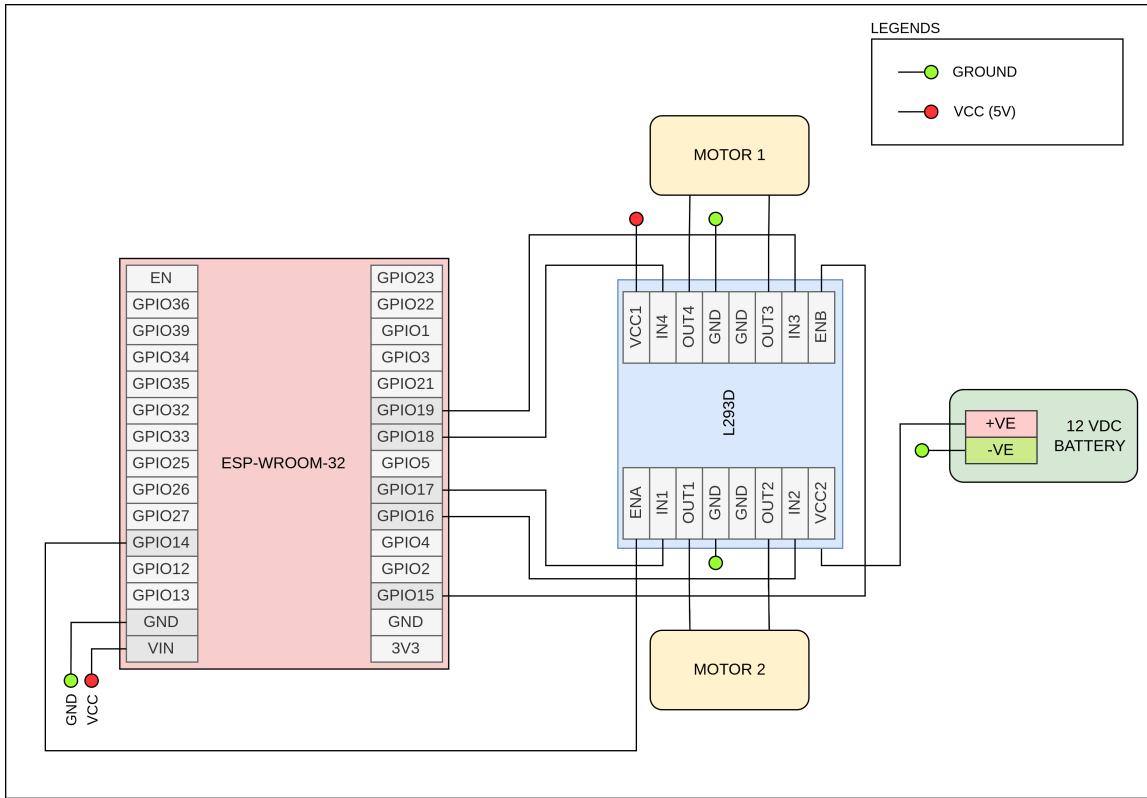


Figure 4.9: Wiring Diagram for UGV beacon tracking

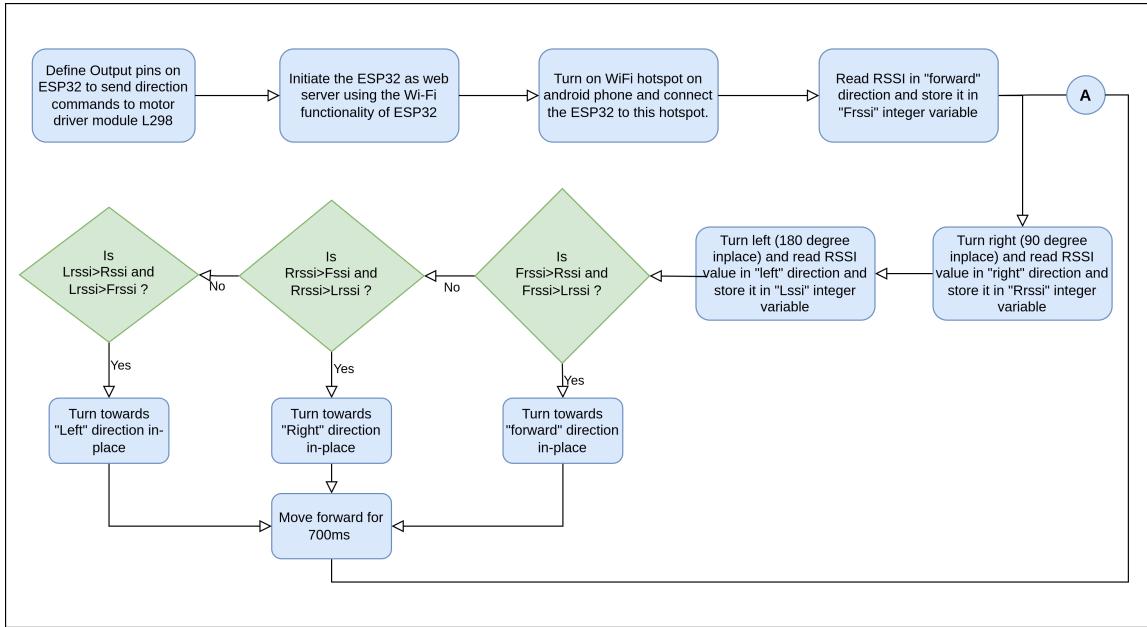


Figure 4.10: Flow Diagram for UGV beacon tracking

- Further, It moves forward for a certain distance towards the beacon. By repeating above steps again and again, the car navigates towards the beacon.

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 4.11).
- 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
```

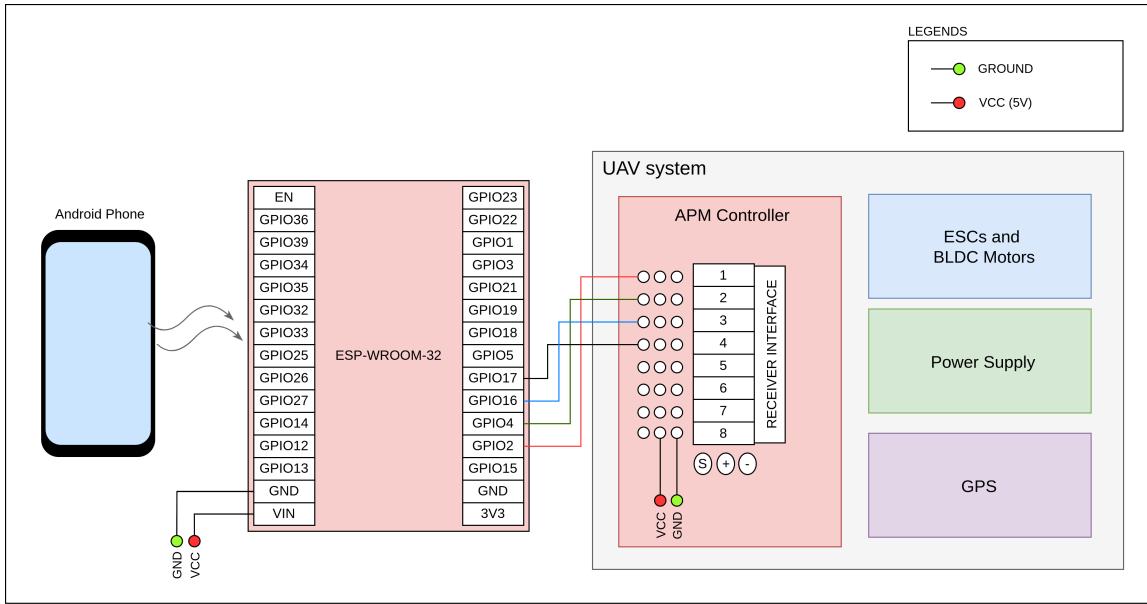


Figure 4.11: Wiring Diagram for UAV Navigation using ESP32 and Android phone

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.

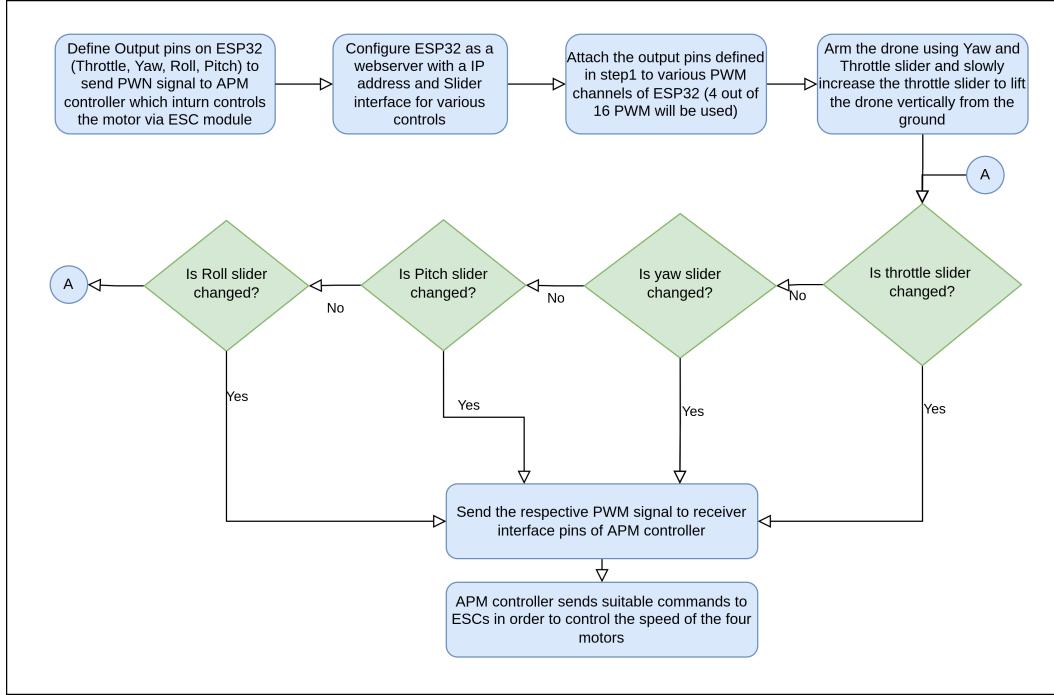


Figure 4.12: Flow Diagram for UAV Navigation using ESP32 and Android phone

4.3 Running two tasks simultaneously using FreeRTOS (Arduino)

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.

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

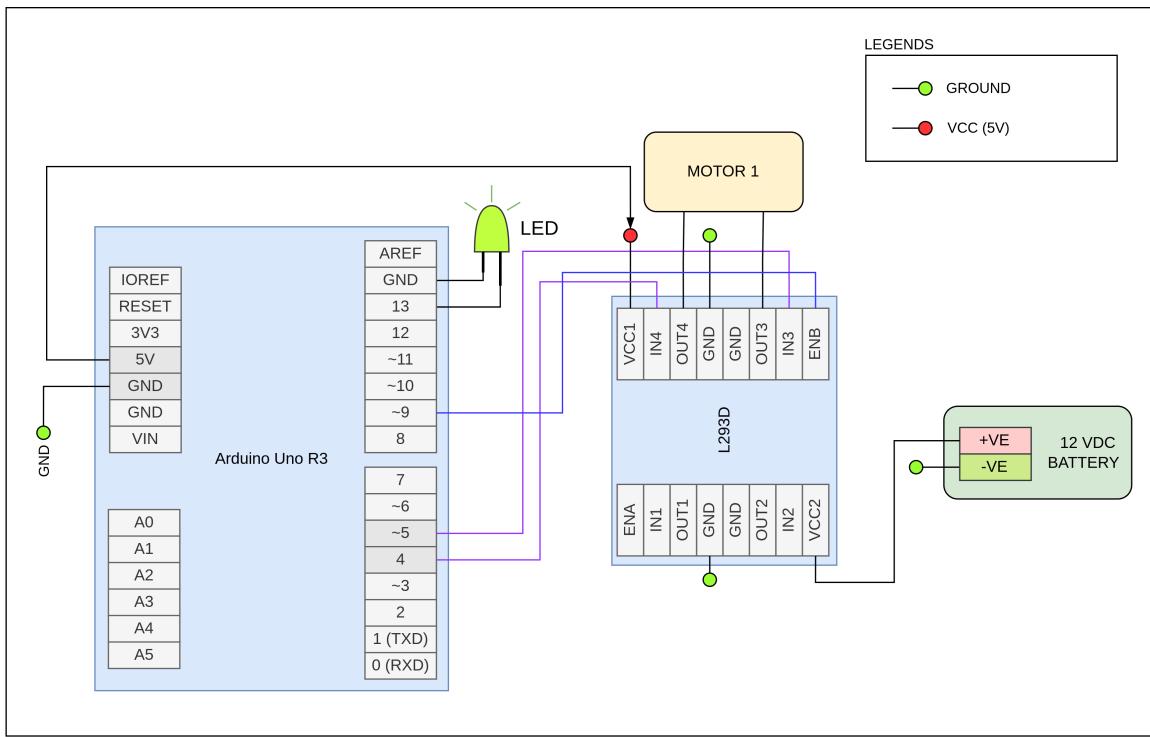


Figure 4.13: Wiring diagram for running two tasks simultaneously on Arduino Uno.

Steps

- Make the connections as per the wiring diagram (Figure 4.13).
- Connect the Arduino Uno to Laptop/PC using Type-B USB cable.
- Open the program at the code link (4.3) 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
    }
}
```

Chapter 5

Vaman Based Applications

5.1 UGV (Unmanned Ground Vehicle)

5.1.1 Navigation using Fly-sky Transmitter and Receiver

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 5.1).
- 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 (5.1.1)

- 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
```

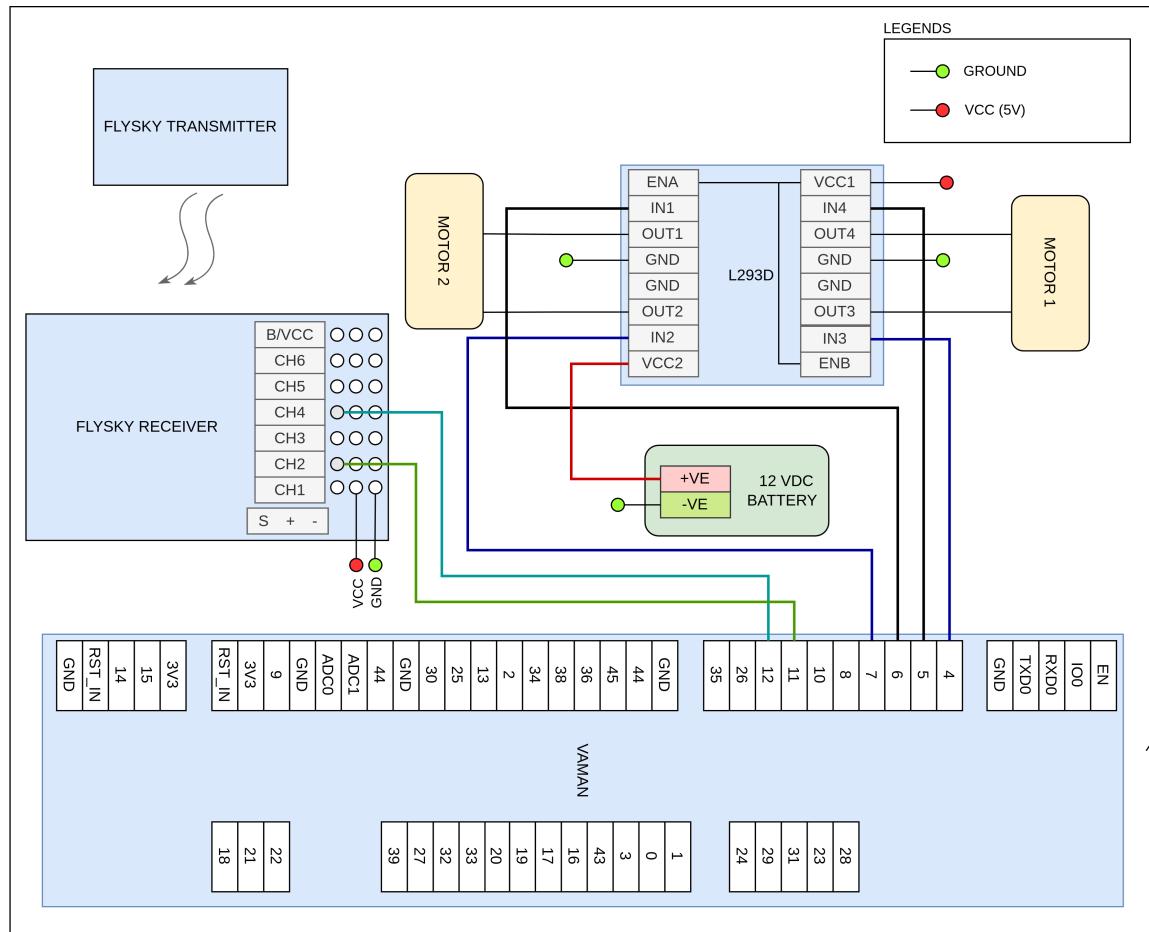


Figure 5.1: Wiring Diagram for UGV Navigation using Fly-sky transmitter & receiver (Vaman)

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.

5.1.2 Navigation using Android phone

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 5.2).
- 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 (5.1.2).
- 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

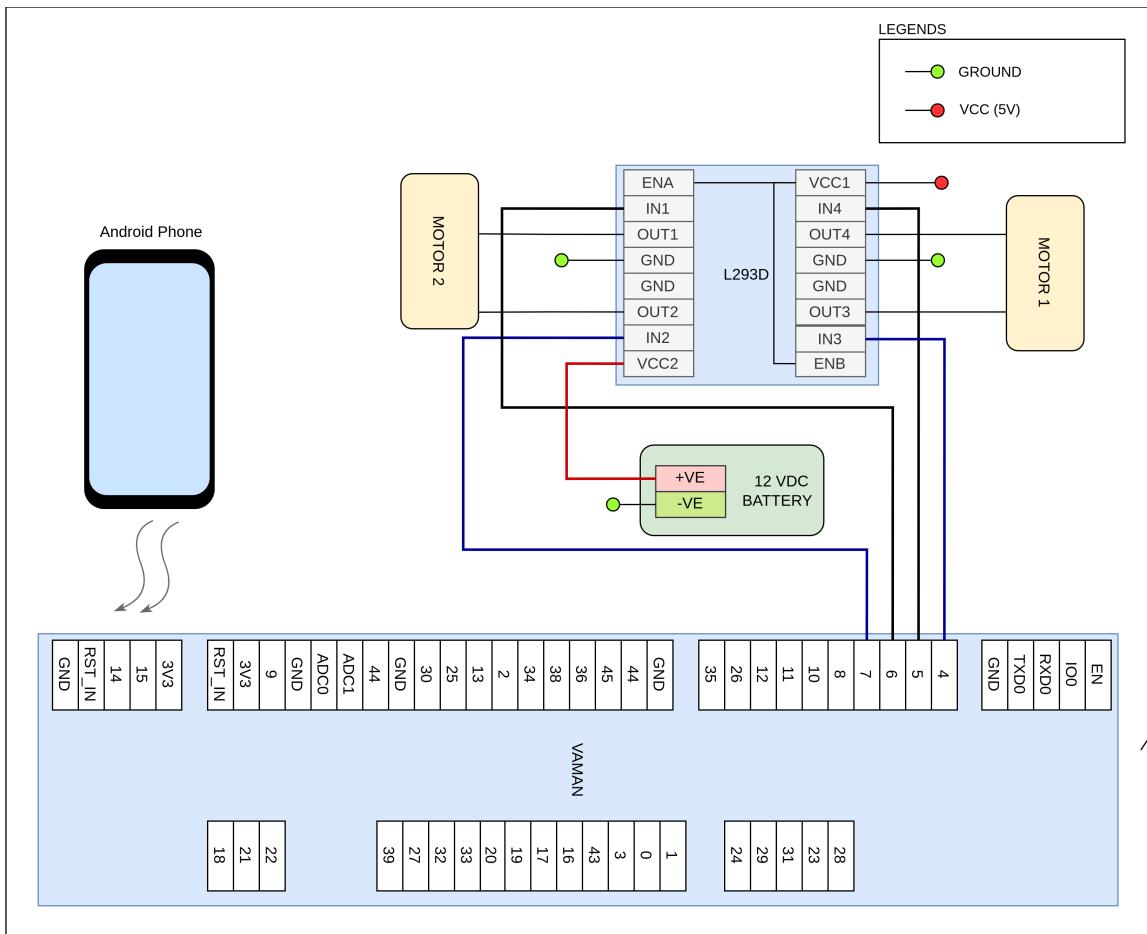


Figure 5.2: Wiring Diagram for UGV Navigation using Android phone (Vaman)

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 4.5.
- 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.

5.2 Simultaneous execution of tasks 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.

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.

Steps

- Make the connections as per the wiring diagram (Figure 5.3).
- 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 (5.2).
- Flash the generated bin file on Vaman using the tinyfpga-programmer-gui.py tool.

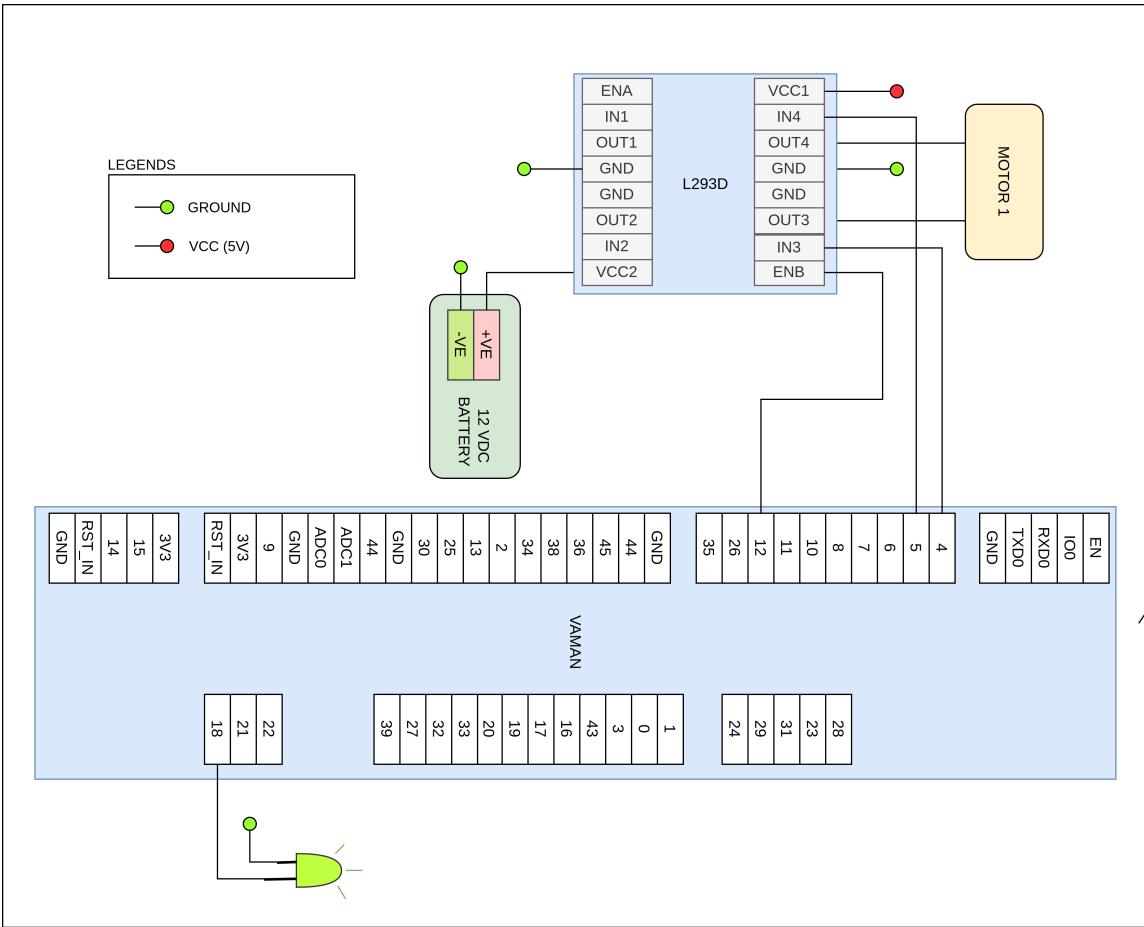


Figure 5.3: Wiring diagram for running two tasks simultaneously on Vaman.

- 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
    }
}
```

Chapter 6

SATCOM for UAV

Communication

6.1 SATCOM Introduction

Satellite Communication (SATCOM) can be utilized in applications where the user wants the data from the UAV to be accessed immediately without wanting the UAV to return. It would benefit greatly in surveillance applications in remote locations, where cellular resources are unavailable.

3GPP community is also trying to incorporate SATCOM with cellular technologies, especially 5G, given its low latency and high datarate capabilities. In a typical 5G cellular network, we have a UE (user equipment) communicating with the 5G core network via the gNB (i.e base station) to access the data and voice services as shown in the Figure 6.1.

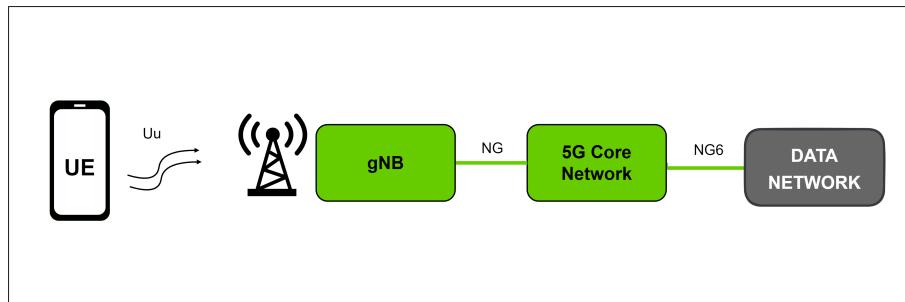


Figure 6.1: Conventional 5G-NR system

However, in case where our user equipment is an UAV, it may not be always reachable from its nearest gNB (for example in remote areas like forests, mountainous terrain, etc). This is where

SATCOM would provide a non-terrestrial network infrastructure, enabling communication with such remote devices. Furthermore, latest advancements in the satellite communications have overcome previous constraints. New generation of Low Earth Orbit (LEO) constellations have lessened satellite communications latency, and technological improvements in Geostationary (GEO) satellites have provided high throughput and increased the reliability of GEO satellites.

A SATCOM infrastructure consists of one or more satellites. Satellites are Spaceborne vehicles orbiting the earth in Low Earth Orbits (LEO), Medium Earth Orbits (MEO), or Geostationary Earth Orbit (GEO). A non-terrestrial network refers to a network, or segment of networks using RF resources on board a satellite.

A Transparent satellite based NG-RAN architecture is shown in Figure 6.2. Here, the Uu Air interface between the user equipment (i.e. UAV) and the gNB is replaced by a satellite link. This ensures connectivity everywhere on the surface which is under coverage of the satellite.

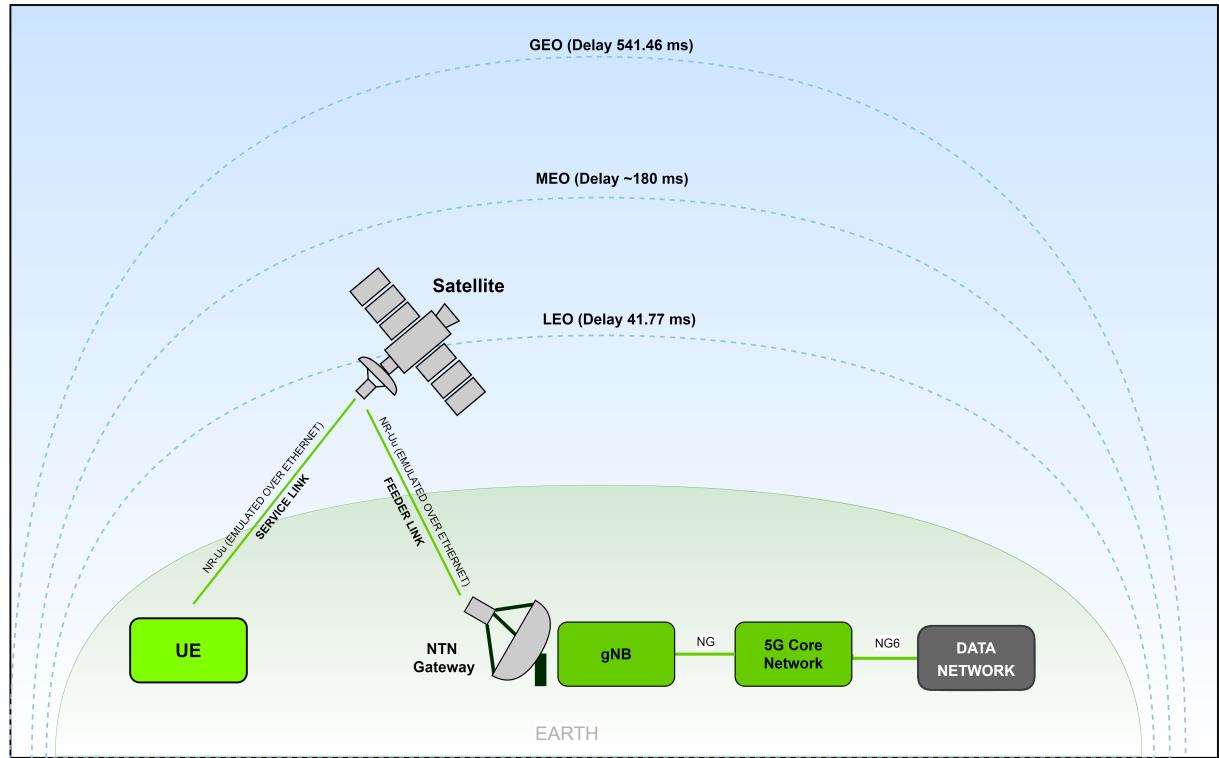


Figure 6.2: Transparent satellite based NG-RAN architecture

There can be various types of satellites, these are compared in Table 6.1:

PARAMETER	GEO	MEO	LEO
Altitude	36000 km	7000 to 25000 km	300 to 1500 km
Round trip delay in milli-seconds	High (541.46 ms)	Low (~180ms)	Very low (41.77 ms)
Earth area coverage	Very large	Large	Low
Satellites required	3 Satellites	6 Satellites	Hundreds of Satellite

Table 6.1: Comparison between GEO, LEO and MEO

6.1.1 Application of SATCOM based UAV

A UAV with SATCOM capability can take on a far wider range of tasks, restricted only by its fuel capacity. For example:

- A UAV can be used by an oil business, utility, or farming company to do field inspections or deliver crucial products to remote locations hundreds of miles away from the operator.
- From the next state, a defence or national security team can deliver a key part to reservists during a training drill or monitor the border with video or infrared cameras.
- From a remote command center, Monitoring person can get a real-time bird's eye view of a fire, accident, or medical emergency before the rescue team arrives. This will greatly help in search and rescue operations.
- From anywhere in the world, disaster relief groups may transport supplies to flood, hurricane, or earthquake victims and assess the situation through streaming video.

6.2 Set-up for Demonstration of SATCOM

This section gives details on the set-up used to emulate SATCOM for UAV using 5G system. The set-up consists of three Linux-x86 workstations connected over LAN. These workstations function as UE, gNB and 5G core respectively. Figure xx shows the three entity along with the software stack at each entity.

In a conventional 5G system, the UE and gNB communicate over wireless interface called the Uu interface. However, in our set-up this interface is emulated using raw socket connection over LAN. Further, the connection between the gNB and the 5G core is also over the raw socket. Since, we are using raw socket to transfer the control and data packets between the entities, the PHY layer which performs functions such as amplification, coding, modulation, etc is not needed. Hence, the PHY layer is not implemented in this set-up. Software stack for all other layers (MAC, RLC, PDCP, SDAP, RRC, NAS) are present at both UE and gNB. The Figure 6.3 show the setup for demonstration of SATCOM.

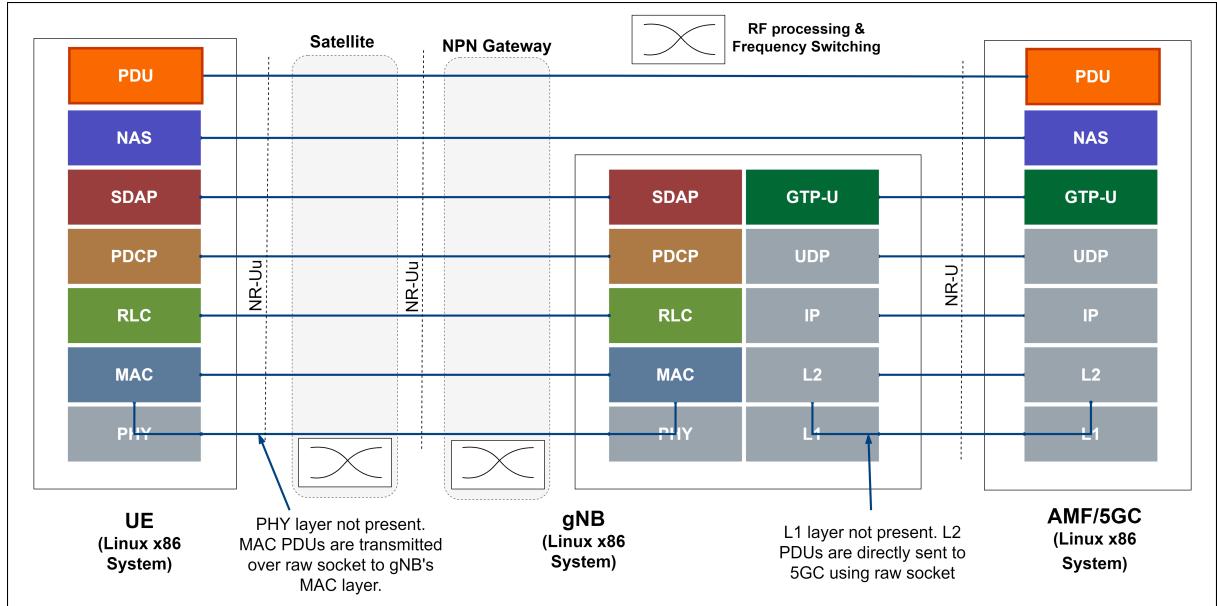


Figure 6.3: Set-up for Demonstration of SATCOM

6.2.1 Free-5GC Environment

- The Free5GC project is an open-source initiative for mobile core networks of the fifth generation (5G) aimed at construction of the 5G core network (5GC) as described in 3GPP Release 15 (R15) and further.

- It our setup it acts as our 5G core network and runs on a linux-x86 system to provides services to the UE via the gNB as defined by the 3GPP speciations.
- The main tasks of the 5G core includes : Radio resource allocation, control and initial set-up of UE, authenticates subscribers and devices, applies personalized policies, manages the mobility of the devices, etc.

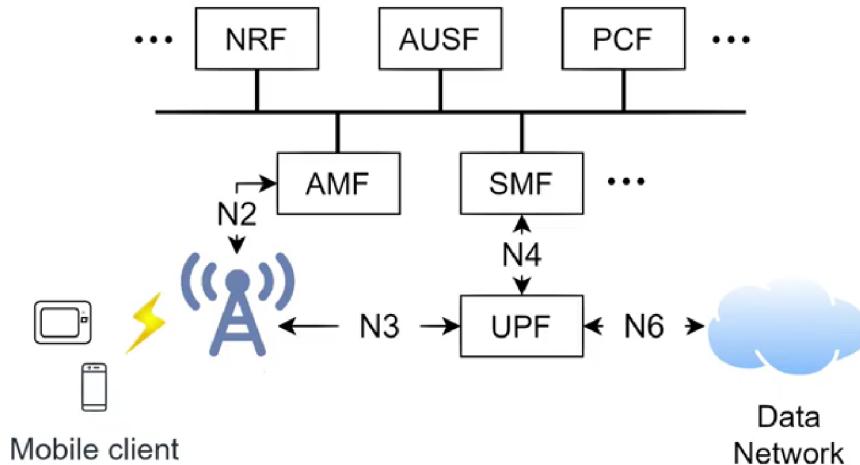


Figure 6.4: Free-5GC Environment

6.3 Video Streaming using the setup

Make sure the FREE-5GC is running. Follow this document named "Free5GC setup" to set up and run FREE-5GC.

6.3.1 Compilation and Execution at gNB

NOTE : For Pre-requisites and detailed installation steps , Please refer to the file "**UE_gNB_setup_user_guide.docx**". After pre-requisite installations (JSON, ROHC, Nettle, Libexplain, DPDK, ASN1, SCTP, AESN1 and IPSEC), Follow the below steps in gNB workstation terminal:

Exporting environment variables:

```
export RTE_SDK=<path to DPDK folder installed>
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

Loading Huge pages:

```
sudo su
echo 4096 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
exit
```

Compiling and running the gNB App:

```
cd ~/Documents/simran_wsp/bs_working/review-bs/5gnrps/src/gnbapp/test
make clean
make static -j10
sudo ./gnbapp enp1s0 -- -diersg
```

6.3.2 Compilation and Execution at UE

NOTE : For Pre-requisites and detailed installation steps , Please refer to the file **”UE_gNB_setup_user_guide.docx”**. After pre-requisite installations (JSON, ROHC, Nettle, Libexplain, DPDK, ASN1, SCTP, AESN1 and IPSEC), Follow the below steps in gNB workstation terminal:

Exporting environment variables:

```
export RTE_SDK=/home/greyteal/Documents/6G_Project/dpdk-stable-20.02.1
export RTE_TARGET=x86_64-native-linuxapp-gcc
```

Loading Huge pages:

```
sudo su
echo 4096 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
exit
```

Compiling and running the UE App:

```
cd /home/greyteal/Documents/simran_wsp/UE/5gnrps/src/ueapp/test
make clean
make CPUSOC=1 JSON=1 -j10
sudo ./ueapp enp1s0 0 -- -dierns
```

Note the PDU session IP address and open a seperate terminal to run the following commands:

```
sudo ifconfig tun00 10.60.0.1 up  
sudo ip route add 192.168.134.224 dev tun00
```

6.3.3 Emulating Satellite round-trip delay

A satellite in the GEO orbit has a round-trip signal delay of around 542ms. The entities in our setup communicate over LAN, hence we used a tool called NetEm to add the specified delay in our interface. NetEm allows linux user to add of delay, packet loss, duplication, and more to packets leaving a particular network interface.

Fixed amount of delay

```
sudo tc qdisc add dev enp2s0 root netem delay 542ms
```

Random delay

```
sudo tc qdisc add dev enp2s0 root netem delay 542ms 20ms
```

Normal delay distribution:

```
sudo tc qdisc add dev enp2s0 root netem delay 542ms 20ms distribution normal
```

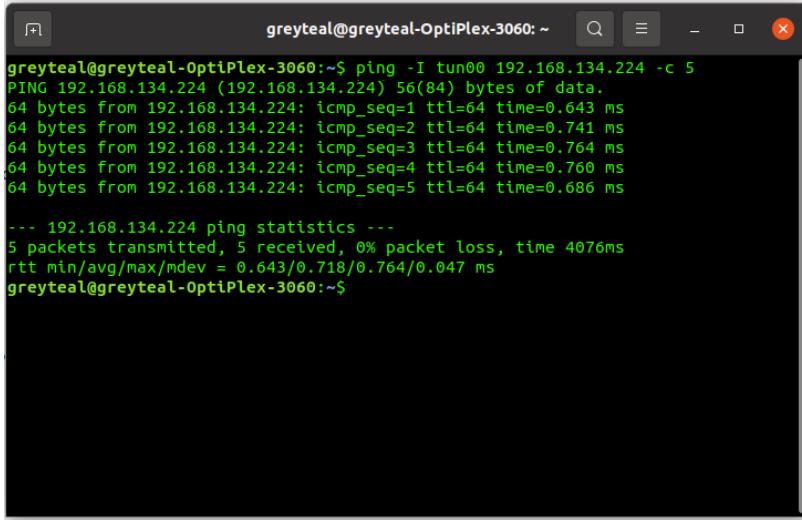
Note: Other commands to show and delete the added interface delay are as follow:

```
tc qdisc show  
sudo tc qdisc del dev enp2s0 root netem
```

6.3.4 Checking the added latency using ping

Before addition of delay as described in section Emulating Satellite round-trip delay, run the following ping command at UE terminal:

```
ping -I tun00 192.168.134.224 -c 5
```



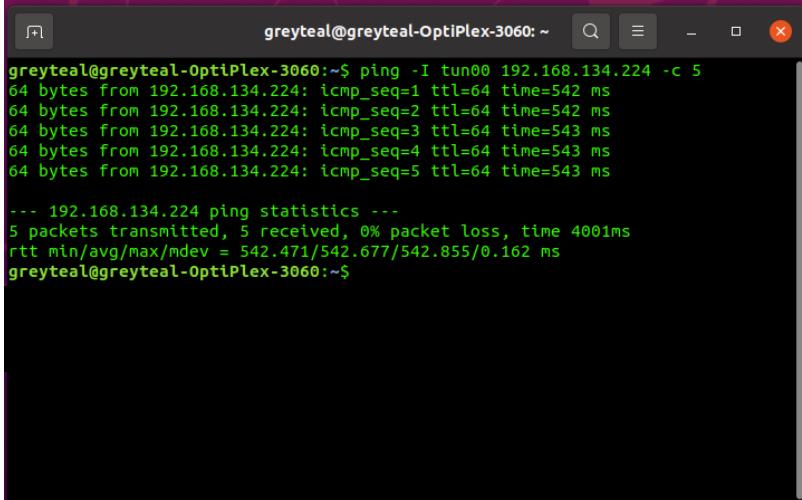
A terminal window titled "greyteal@greyteal-OptiPlex-3060:~". It displays the output of the ping command. The output shows five ICMP packets being sent to 192.168.134.224. Each packet has a sequence number (seq=1 to seq=5), TTL of 64, and a time between 0.643 ms and 0.764 ms. Below the packets, the ping statistics show 5 packets transmitted, 5 received, 0% packet loss, and an average round-trip time (rtt) of 4076ms.

```
greyteal@greyteal-OptiPlex-3060:~$ ping -I tun00 192.168.134.224 -c 5
PING 192.168.134.224 (192.168.134.224) 56(84) bytes of data.
64 bytes from 192.168.134.224: icmp_seq=1 ttl=64 time=0.643 ms
64 bytes from 192.168.134.224: icmp_seq=2 ttl=64 time=0.741 ms
64 bytes from 192.168.134.224: icmp_seq=3 ttl=64 time=0.764 ms
64 bytes from 192.168.134.224: icmp_seq=4 ttl=64 time=0.760 ms
64 bytes from 192.168.134.224: icmp_seq=5 ttl=64 time=0.686 ms

--- 192.168.134.224 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4076ms
rtt min/avg/max/mdev = 0.643/0.718/0.764/0.047 ms
greyteal@greyteal-OptiPlex-3060:~$
```

Figure 6.5: Ping before adding delay

Now, Add delay using NetEm as described in section Emulating Satellite round-trip delay and run the ping command again.



A terminal window titled "greyteal@greyteal-OptiPlex-3060:~". It displays the output of the ping command. The output shows five ICMP packets being sent to 192.168.134.224. Each packet has a sequence number (seq=1 to seq=5), TTL of 64, and a time between 542 ms and 543 ms. Below the packets, the ping statistics show 5 packets transmitted, 5 received, 0% packet loss, and an average round-trip time (rtt) of 4001ms.

```
greyteal@greyteal-OptiPlex-3060:~$ ping -I tun00 192.168.134.224 -c 5
64 bytes from 192.168.134.224: icmp_seq=1 ttl=64 time=542 ms
64 bytes from 192.168.134.224: icmp_seq=2 ttl=64 time=542 ms
64 bytes from 192.168.134.224: icmp_seq=3 ttl=64 time=543 ms
64 bytes from 192.168.134.224: icmp_seq=4 ttl=64 time=543 ms
64 bytes from 192.168.134.224: icmp_seq=5 ttl=64 time=543 ms

--- 192.168.134.224 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 542.471/542.677/542.855/0.162 ms
greyteal@greyteal-OptiPlex-3060:~$
```

Figure 6.6: Ping after adding delay

6.3.5 5G-NR call flow

After adding delay and checking the same, execute the software at gNB followed by software at UE. The following events take place in order to register the UE in the core network (Control plane function)

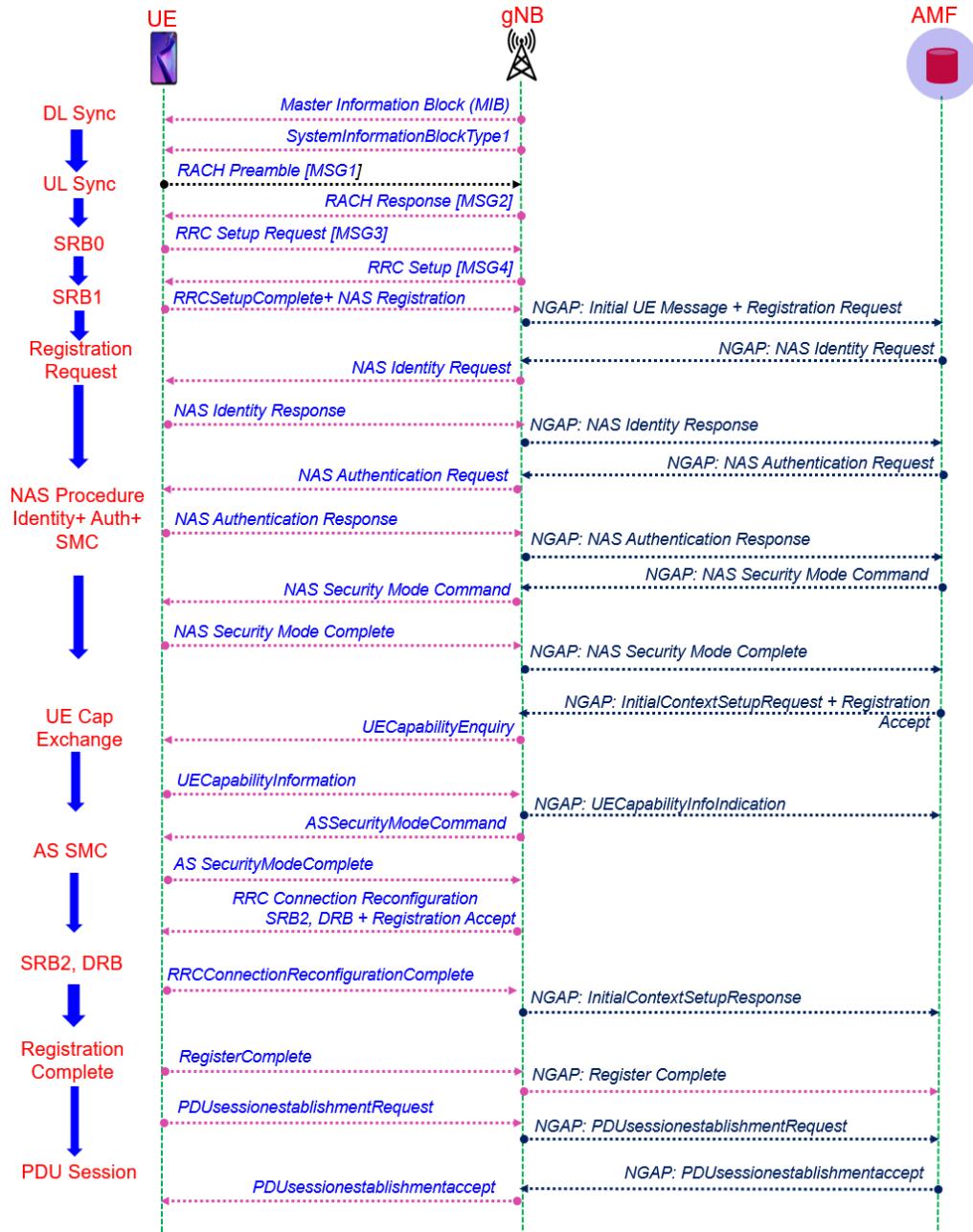


Figure 6.7: 5G-NR call flow

6.3.6 Streaming video using the setup

Creating video stream at source

To create a video stream (using file videoplayback.mp4) from 5GC to UE via gNB, run the following command in terminal of 5GC workstation. This will use UDP to stream video to a given destination IP address and port number.

```
cd ~/Documents #go to video path  
vlc videoplayback.mp4 --sout=udp://10.60.0.1:1234
```

Viewing video stream at the destination

The video packets streamed by source travel through the software stack as shown in Figure xx. To view the stream at destination, run the following command in terminal. This will read the video packets received at the tun00 interface from the SDAP layer of software stack.

```
vlc udp://@:1234 --miface=10.60.0.1:1234
```

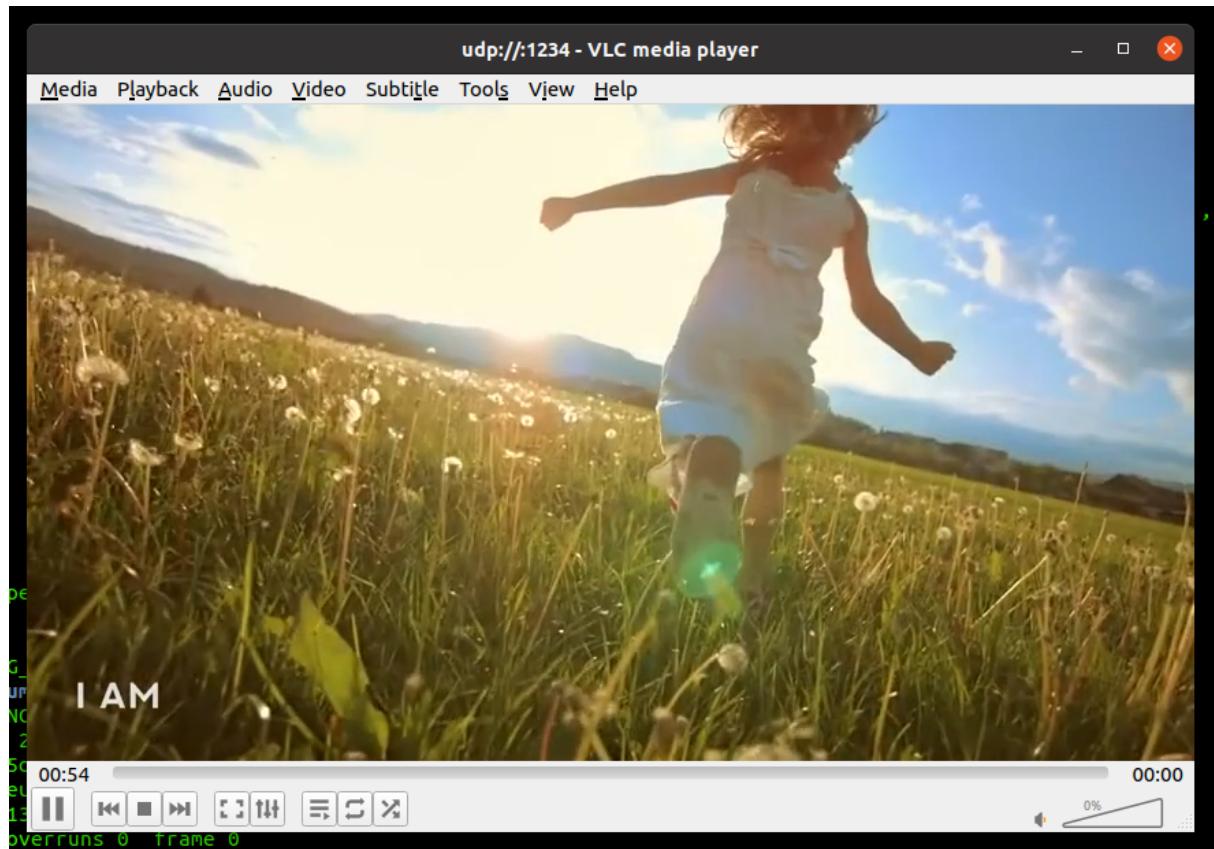


Figure 6.8: Viewing video stream at the destination

6.3.7 Video stream result

Sr No.	Delay between the play command at source and video packets received at the destination (sec)	Delay between the stop command at source and video packets halt at the destination (sec)
1	2.10	1.60
2	2.43	1.97
3	2.04	1.90
4	2.17	2.10
5	2.16	1.7
Average	2.18	1.854

Table 6.2: Result without ethernet delay

Sr No.	Delay between the play command at source and video packets received at the destination (sec)	Delay between the stop command at source and video packets halt at the destination (sec)
1	2.78	2.57
2	2.58	2.32
3	2.69	2.42
4	2.64	2.74
5	2.85	2.43
Average	2.708	2.496

Table 6.3: Result with ethernet delay of 542 ms

Chapter 7

Conclusion and Future Directions

Conclude your thesis.