ACHARYA PRAFULLA CHANDRA RAY POLYTECHNIC

188, Raja Subodh Chandra Mallick Rd, Jadavpur

QUADCOPTER USING RF TRANSMITTER-RECEIVERS AND ARDUINO NANO'S

Project report submitted in fulfillment of the requirement for **Diploma in Electronics and Instrumentation Engineering** under the **West Bengal State Council of Technical & Vocational Education and Skill Development (Technical Education Division)**

Submitted by

SHANTOM GAYEN	**D222301307**
RAKESH BHOWMIK	**D222301313**
SHIBJYOTI DAS	**D222301316**
SANJIT SAREN	**D222301320**
DIP GHOSH	**D222301323**
SAYAN BISWAS	**D222301324**
ANUVA BHATTACHARYA	**D222301326**
PRANAB MONDAL	**D222301331**

Under the supervision of

Sri. DEBASIS MONDAL

Lecturer, Department of Electronics and Instrumentation Engineering
Acharya Prafulla Chandra Ray Polytechnic

2025

A.P.C. RAY POLYTECHNIC

DEPARTMENT OF ELECTRONICS AND INSTRUMENTATION ENGINEERING

CERTIFICATE OF APPROVAL

This is to certify that the Project Report entitled "**QUADCOPTER USING RF TRANSMITTER-RECEIVERS AND ARDUINO NANO'S**" submitted by **SHANTOM GAYEN**, **RAKESH BHOWMIK**, **SHIBJYOTI DAS**, **SANJIT SAREN**, **DIP GHOSH**, **SAYAN BISWAS**, **ANUVA BHATTACHARYA** & **PRANAB MONDAL** has been carried out under my/our supervision in partial fulfillment of the requirements for the Diploma in Electronics and Instrumentation at Acharya Prafulla Chandra Ray Polytechnic, Jadavpur, Kolkata-700032 and this work has not been submitted elsewhere before for any other academic diploma.

_____

Sri. Debasis Mondal
Lecturer in Electronics and Instrumentation Engineering

_____

Head of the Department
Electronics and Instrumentation Engineering

ACKNOWLEDGEMENT

This Project Report would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable guidance and support in the preparation and completion of this.

Firstly, we want to express our sincere gratitude and thank our supervisor Sri. DEBASIS MONDAL, Lecturer in Electronic & Instrumentation Engineering A.P.C. Ray Polytechnic, Jadavpur for their unreserved help, motivation, enthusiasm and constant guidance to finish our project work step by step. Under his supervision we successfully overcame many adversities and learned a lot. We extend our deep sense of obligation and honour to him for inspiring discussions, kind cooperation and constant encouragement throughout the period of our project work which has been influential in the success of the project.

Secondly, we want to convey heartfelt thanks to all the faculty members and staff of A.P.C. Ray Polytechnic, Jadavpur and our project members for their indebted help and valuable suggestions for successful completion of the project work.

Last but not least, we would like to pay high regards to our parents, our friends and the omnipresent God for giving us strength in all the critical situations and supporting us spiritually throughout our lives.

Thanks and regards,

Name	Role
SHANTOM GAYEN	Circuit design, Leading & Management
RAKESH BHOWMIK	Miscellaneous
SHIBJYOTI DAS	Software & Report Preparation
SANJIT SAREN	Miscellaneous
DIP GHOSH	Wiring & Soldering
SAYAN BISWAS	Material Handling and Management
ANUVA BHATTACHARYA	Research & Information Gathering
PRANAB MONDAL	Hardware Preparation

CONTENTS

INTRODUCTION

A quadcopter drone is a small flying aircraft that uses four spinning propellers to lift off the ground and move through the air. Quadcopters have four smaller rotors positioned at each corner of the frame, giving them their distinctive ***quad*** (meaning 'four') name. The four propellers spin to create lift through the principles of aerodynamics. Each propeller blade is angled to push air downward as it rotates, creating an upward force that counteracts gravity. By spinning faster or slower, the drone can go up or down. The beauty of the quadcopter design lies in its ability to achieve precise movement through subtle adjustments to individual rotor speeds.

The flight control system works by varying the speed of each propeller independently. When all four propellers spin at the same speed, the drone hovers in place. To ascend, all propellers increase speed simultaneously. To descend, they all slow down together. By tilting slightly in different directions, it can move forward, backward, left, or right. This tilting motion is achieved by creating slight speed differences between opposing propellers, causing the drone to lean in the desired direction of travel.

The pilot controls all of this using a handheld remote control that sends radio signals to the drone. Controllers typically feature two joysticks that correspond to different flight functions. One joystick usually controls **throttle** (up and down movement) and **yaw** (rotation around the vertical axis), while the other manages **pitch** (forward and backward tilt) and **roll** (left and right tilt).

Quadcopters also include sophisticated stabilization systems with *gyroscopes* and *accelerometers* that help maintain steady flight even in windy conditions. These sensors constantly monitor the drone's orientation and make rapid adjustments to keep it level and stable, making them much easier to fly than earlier remote-controlled aircraft.

Quadcopters are popular for recreational flying and photography, but their applications extend far beyond hobbyist use. Many people enjoy the challenge of learning to pilot them smoothly through the air that can be both relaxing and exhilarating.

The versatility of quadcopters has led to their adoption in numerous professional fields. They're used for **agricultural monitoring**, where they can survey large crop areas and identify problems like pest infestations or irrigation issues. **Search and rescue operations** employ drones to locate missing persons in difficult terrain. **Infrastructure inspection** has been transformed by drones that can safely examine bridges, power lines, and tall buildings without putting human inspectors at risk.

The technology continues to evolve rapidly, with improvements in battery life, and autonomous flight capabilities. As regulations adapt to accommodate these versatile machines, quadcopters are likely to become even more integrated into both our recreational activities and professional industries, making them one of the most significant technological innovations of the modern era.

WORKING PRINCIPLE

The drone generates lift through four rotors arranged in a cross configuration. Each rotor consists of propeller blades that create a pressure differential by accelerating air downward, generating an upward thrust force according to Newton's third law. **The total lift must equal or exceed the drone's weight to achieve flight.**

The drone controls its orientation and movement by varying the speed of individual rotors:

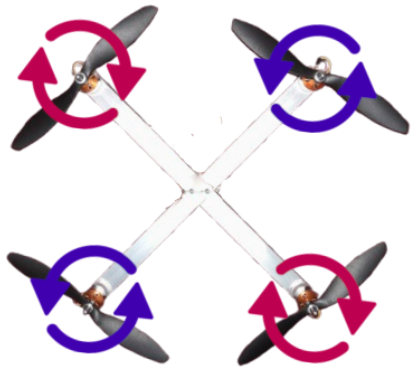
- **Roll Motion**: By spinning the left rotors faster than the right rotors (or vice versa), the drone creates unequal lift forces that generate a rolling moment about the longitudinal axis.
- **Pitch Motion**: Similarly, increasing thrust from the rear rotors while decreasing front rotor thrust creates a pitching moment about the lateral axis.
- **Yaw Motion**: Adjacent rotors spin in opposite directions to cancel out torque. By slightly increasing the speed of rotors spinning in one direction while decreasing the others, a net torque is created, causing the drone to rotate about the vertical axis.
- **Vertical Motion**: All rotors increase or decrease speed simultaneously to climb or descend.

The drone's dynamics are governed by rigid body equations of motion. The key relationships include:

- **Angular Momentum**: $\tau = I\alpha$, where *torque (τ)* equals *moment of inertia (I)* times *angular acceleration (α)*.
- **Force Balance**: The net force in each direction determines linear acceleration according to $F = ma$.
- **Complementary Filtering**: The drone combines gyroscope and accelerometer data using a weighted average:
  $Angle = \alpha \times (previous\_angle + gyro\_rate \times time) + (1 - \alpha) \times accelerometer\_angle$

The drone achieves stability through continuous feedback correction. Any external disturbance creates an angular error, which the **PID controller** detects and corrects by adjusting motor speeds. The high-frequency control loop (typically 100-1000 Hz) enables rapid corrections that maintain stable flight despite disturbances like wind gusts.

The **complementary filter** addresses the fundamental trade-off between gyroscope drift (long-term inaccuracy) and accelerometer noise (short-term inaccuracy) by trusting **gyroscopes for short-term changes** and **accelerometers for long-term reference**.



Adjacent propellers rotate in opposite directions to cancel out the reactive torque that each spinning propeller creates. This is based on *Newton's third law of motion* and the *conservation of angular momentum*. This happens because:

1. **Action-Reaction Forces**: As the motor applies torque to spin the propeller clockwise, the propeller applies an equal and opposite torque to the aircraft frame, trying to rotate it counterclockwise.
2. **Angular Momentum Conservation**: The spinning propeller has angular momentum. To maintain the system's total angular momentum, the aircraft body must develop an opposite angular momentum.
3. **Gyroscopic Effects**: The spinning propeller acts as a gyroscope, creating additional torques when the aircraft changes orientation.

To control yaw (rotation about the vertical axis), the drone deliberately creates a torque imbalance:

- **Yaw left**: Increase speed of clockwise rotors, decrease counterclockwise rotors
- **Yaw right**: Increase speed of counterclockwise rotors, decrease clockwise rotors

MATERIALS REQUIRED

TRANSMITTER

1. Arduino NANO
2. nRF24L01 + PA module
3. 7805 IC
4. 1117 IC
5. OLED display
6. Battery connector
7. 9V battery
8. Zero pcb board
9. Joystick module x2
10. 10 µF capacitor x2

RECEIVER

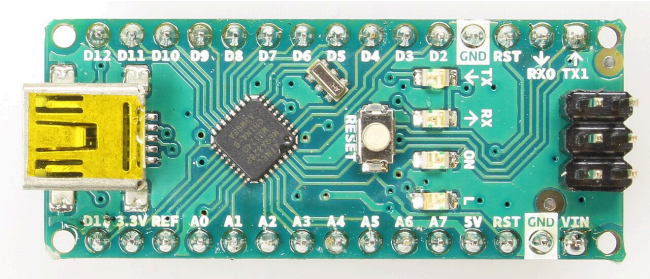
1. Arduino NANO
2. nRF24L01 module
3. 1000 kV BLDC motor x4
4. 30A ESC x4
5. 9" Propeller x4
6. 1117 IC
7. 3s Li-po battery (12V, 2200mAh)
8. XT60 connector
9. MPU6050 gyroscope accelerometer module
10. Voltage sensor
11. 10 µF Capacitor
12. Zero PCB board
13. Aluminum tube (for frame)

ADDITIONAL EQUIPMENTS

1. 12v 10 A power supply
2. 12 AWG Silicon wire
3. Arduino NANO cable

COMPONENT DESCRIPTION

Arduino NANO



Nano is a compact microcontroller board based on the **8-bit ATmega328P microcontroller**. Along with ATmega328P, it consists of other components such as a crystal oscillator, USB interface, voltage regulator, and reset button to support the microcontroller.

The Arduino Nano comes with a mini-USB interface, 8 analog input pins, 14 I/O digital ports that are used to connect with external electronic circuits. Out of 14 I/O ports, 6 pins can be used for PWM output. Its small size and full functionality make it ideal for breadboard-based projects and embedded systems.

Since its introduction, the Arduino Nano has become a popular choice among students, hobbyists, and developers working on compact and space-constrained projects. Like other Arduino boards, it is an **open-source platform**, allowing anyone to modify and customize the board and its code for specific use cases. The Arduino IDE (Integrated Development Environment) supports Nano as well and allows programming in C and C++ with minimal setup.

Arduino NANO Pin Details

PIN CATEGORY	PIN NAME	DETAILS
Power	Vin	Input voltage to Arduino when using an external power source.
	3.3V	3.3V output from on-board voltage regulator. Max current draw is 50mA.
	5V	Regulated power supply used to power the microcontroller and components.
	GND	Ground pins.
Reset	Reset	Resets the microcontroller.
Analog Pins	A0–A7	Used to provide analog input in the range of 0–5V.
Input/Output Pins	Digital Pins D0–D13	Can be used as input or output digital pins.
Serial	D0 (Rx), D1 (Tx)	Used to receive (Rx) and transmit (Tx) TTL

		serial data.
External Interrupts	D2, D3	Used to trigger an interrupt.
PWM	D3, D5, D6, D9, D10, D11	Provides 8-bit PWM output.
SPI	D10 (SS), D11 (MOSI), D12 (MISO), D13 (SCK)	Used for SPI communication.
Inbuilt LED	D13	Turns on the onboard LED.
TWI (I^2C)	A4 (SDA), A5 (SCL)	Used for TWI/I^2C communication.
AREF	AREF	Provides reference voltage for analog inputs.

Arduino NANO Specifications

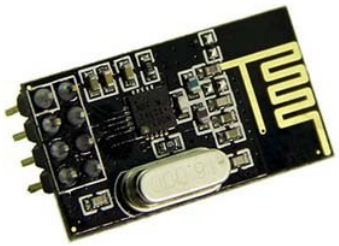
Microcontroller	ATmega328P – 8-bit AVR family microcontroller
Operating Voltage	5V
Recommended Input Voltage	7–12V
Input Voltage Limits	6–20V
Analog Input Pins	8 (A0–A7)
Digital I/O Pins	14 (Out of which 6 provide PWM output)
DC Current on I/O Pins	40mA
DC Current on 3.3V Pin	50mA (via onboard regulator, if present)
Flash Memory	32 KB (0.5 KB is used for bootloader)
SRAM	2 KB
EEPROM	1 KB
Frequency (Clock Speed)	16 MHz

Arduino IDE

The Arduino IDE is the official software platform used to write, compile, and upload code to Arduino boards. It provides a simple and beginner-friendly interface that allows users to develop programs (called sketches) using C and C++ programming languages. The IDE includes built-in libraries, code examples, and tools that make it easy to interface with sensors, modules, and other hardware components.

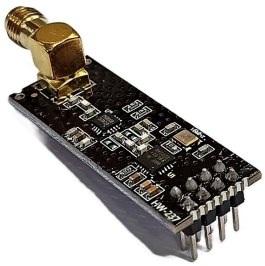
Available for Windows, macOS, and Linux, the Arduino IDE supports a wide range of boards beyond just Arduino via board manager extensions. It communicates with the board over a serial USB connection. With features like syntax highlighting, automatic formatting, and serial monitor, it is ideal for both beginners and experienced developers. The open-source nature of the IDE also allows community-driven improvements and third-party board and library integrations, making it a flexible tool for embedded system development.

NRF24L01 (With Power Amplifier)



The nRF24L01 is a low-power, highly integrated **2.4GHz RF transceiver** module designed by *Nordic Semiconductor*. It enables wireless communication between microcontrollers over short distances using the **ISM (Industrial, Scientific, and Medical) band**. It supports data rates of 250kbps, 1Mbps, and 2Mbps, making it ideal for wireless sensor networks, remote controls, and low-latency applications.

The module operates on 3.3V and uses the **SPI (Serial Peripheral Interface)** protocol to communicate with microcontrollers such as Arduino, ESP32, STM32, and others. It has an onboard antenna or external antenna (in PA+LNA variants) and supports up to **125 communication channels** and **6 data pipes**, allowing multiple devices to transmit and receive data simultaneously.



The **nRF24L01 + PA/LNA** variant is an enhanced version of the standard nRF24L01 module, featuring a **Power Amplifier (PA)** for extended transmission range and a **Low Noise Amplifier (LNA)** for better reception sensitivity. This version is capable of achieving communication ranges up to **1000 meters** (1 km) or more in open space with a clear line of sight and proper antenna setup. It still operates in the 2.4 GHz ISM band, which is globally license-free and commonly used for short-range wireless communication.

This high-power variant is especially suited for applications that require long-distance communication such as remote monitoring, wireless data logging, telemetry, and long-range IoT systems. It includes an external SMA antenna connector, improving signal strength and stability. Despite its increased range, it maintains the same SPI interface and pinout as the regular module, making it backward-compatible and easy to integrate into existing designs. For best performance, it should be powered with a stable 3.3V supply capable of sourcing at least 100–250mA, as the PA and LNA increase current draw significantly.

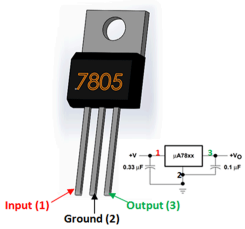
Pin Description

Pin Name	Type	Description
GND	Power	Ground connection. Connect to the GND of your microcontroller.
VCC	Power	Power supply input. Should be **3.3V** (not 5V!). Connecting to 5V will **damage the module**.
CE	Digital In	**Chip Enable**. Used to toggle between standby and active modes (transmit/receive). High to enable.
CSN (CS)	Digital In	**Chip Select Not**. Active low. Used to select the SPI device. Set low before SPI communication.
SCK	Digital In	**Serial Clock**. Clock signal for SPI communication.
MOSI	Digital In	**Master Out Slave In**. Data sent from the microcontroller to the nRF24L01.
MISO	Digital Out	**Master In Slave Out**. Data sent from nRF24L01 to the microcontroller.
IRQ	Digital Out	**Interrupt Request**. Active low. Can be used to notify the MCU about transmit/receive events. (Optional, not always needed for basic setups.)

Pin Connections

nRF24L01 Pin	Connected to
GND	GND
VCC	1117 Output
CE	D9
CSN (CS)	D10
SCK	D13
MOSI	D11
MISO	D12
IRQ	Not connected

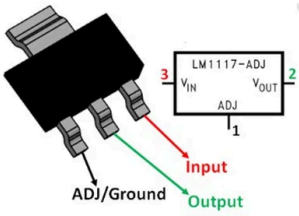
7805 IC



The 7805 is a widely used **linear voltage regulator** from the 78xx series, designed to provide a stable +5 V output from a higher unregulated input (typically between 7 V and 35 V). Because of its simplicity, affordability, and built-in protection features, the 7805 is ubiquitous in hobbyist and professional electronics projects whenever a reliable 5 V supply is needed.

Specification	Details
Output Voltage	+5V
Input Voltage Range	7V to 35V (recommended: 7V to 25V)
Output Current	Up to 1A (with proper heat sink)
Package Types	TO-220 (most common), TO-92, SMD
Internal Protection	Thermal Shutdown, Short Circuit Protection, Safe Operating Area Protection

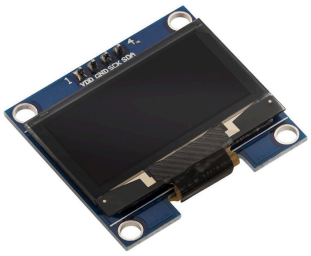
1117 IC



The AMS1117-3.3 (commonly referred to as 1117 3.3V) is a widely used **low-dropout linear voltage regulator** from the AMS1117 series, designed to provide a stable +3.3V output from a higher unregulated input (typically between 4.5V and 15V). Because of its low dropout voltage, compact form factor, and built-in protection features, the AMS1117-3.3 is extremely popular in modern electronics projects, especially those involving microcontrollers, WiFi modules, and other 3.3V digital circuits.

Specification	Details
Output Voltage	+3.3V
Input Voltage Range	4.5V to 15V (recommended: 4.5V to 12V)
Output Current	Up to 1A (with proper heat sink)
Package Types	SOT-223 (most common), TO-252, SOT-89
Internal Protection	Thermal Shutdown, Short Circuit Protection, Safe Operating Area Protection

OLED Display



An OLED display (Organic Light Emitting Diode) uses **organic compounds to emit light** when electricity is applied. Known for high contrast, wide viewing angles, and low power consumption, especially with dark content.

Unlike LCDs, OLEDs emit their own light without requiring a backlight, enabling true blacks, lower power consumption for dark screens, and faster response times.

It is ideal for embedded systems, Arduino projects, IoT devices, and any application requiring compact, high-contrast displays with efficient power usage.

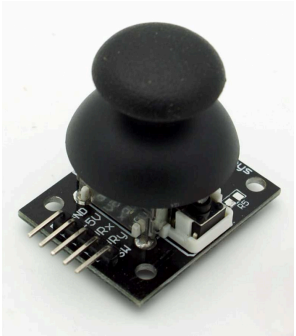
OLED Specifications

Specification	Details
Model	0.96" I2C OLED Display
Resolution	128×64 pixels
Interface	I2C
Colour	Monochrome (white, blue, or yellow)
Driver IC	SSD1306f
Supply voltage	3.3V to 5V

Wiring with Arduino

OLED	Arduino UNO
SCL	A5
SDA	A4
VCC	5V or 3.3V
GND	GND

Joystick Module



The joystick module is a fundamental input device widely used in Arduino and embedded system projects to provide **two-axis analog control** with an **integrated push-button switch**. Based on **potentiometer** technology, this module outputs variable voltage levels corresponding to the joystick's X and Y positions, making it ideal for directional control

applications. The module typically operates on 3.3V to 5V supply voltage and provides analog outputs that can be directly interfaced with microcontroller ADC pins.

In embedded systems development, joystick modules serve as intuitive human-machine interfaces for robotics, gaming controllers, remote control systems, and interactive displays. Their simple analog output interface makes them particularly suitable for Arduino projects where precise directional input is required without the complexity of digital communication protocols.

Pin Configuration

Pin	Function	Description
5V	Power supply	3.3V to 5V DC input
VRx	X-axis output	Analog voltage output (0-5V) for horizontal movement
VRy	Y-axis output	Analog voltage output (0-5V) for vertical movement
GND	Ground	Connect to system ground
SW	Switch	Digital output from push-button (active LOW)

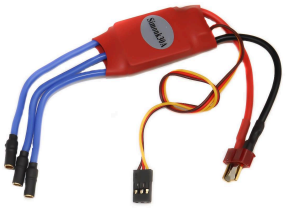
Technical Specifications

Parameter	Value
Operating Voltage	3.3 - 5V DC
Output Voltage Range	0V - VCC
Rest Position Output	~2.5V (center)
Interface Type	Analog (X,Y) + Digital (Switch)
Dimensions	~40mm x 28mm
Operating Temperature	-40°C to +85°C

BLDC Motor with ESC

Brushless DC (BLDC) motors have revolutionized embedded systems applications, offering superior efficiency, longevity, and precise control compared to traditional brushed motors. A 1000 kV BLDC motor paired with a 30A Electronic Speed Controller (ESC) represents a powerful combination ideal for high-performance applications such as quadcopters, RC vehicles, industrial automation, and robotics projects. **The kV rating indicates the motor's velocity constant** - 1000 RPM per volt applied - making this motor

suitable for applications requiring high-speed operation with moderate torque output.



The integration of BLDC motors with Arduino-based embedded systems has opened new possibilities for hobbyists and professionals alike. The ESC serves as the critical interface between the microcontroller and motor, handling the complex three-phase switching required for smooth operation while accepting simple PWM control signals from Arduino or other microcontrollers.

Motor-ESC Interface Operation

The BLDC motor and ESC work together through electronic commutation, where the ESC replaces the mechanical brushes found in traditional DC motors. The ESC contains six MOSFETs arranged in three half-bridge configurations, each controlling one phase of the motor's three-phase winding. Using **back-EMF sensing or Hall effect sensors**, the ESC determines rotor position and switches the appropriate MOSFETs to maintain continuous rotation.

The Arduino communicates with the ESC using standard servo PWM protocol: 1000µs pulse width for minimum throttle (motor stop), 1500µs for half throttle, and 2000µs for maximum throttle. The ESC interprets these signals and adjusts the motor's speed accordingly by varying the switching frequency and duty cycle of the three-phase output.

Motor Specifications

Parameter	Value	Description
kV Rating	1000 kV	RPM per volt (no-load speed)
Maximum Current	30A	Continuous current rating
Voltage Range	7.4V - 22.2V	2S to 6S LiPo compatible
Motor Type	3-Phase BLDC	Brushless, sensorless design
Pole Configuration	Typically 14-pole	High torque, smooth operation
Shaft Diameter	3.5mm	Standard propeller/gear mounting

ESC Specifications

Parameter	Value	Description
Current Rating	30A Continuous	Maximum sustained current
Burst Current	40A (10 seconds)	Short-term overload capability
Input Voltage	2S-6S LiPo	7.4V to 22.2V

Control Signal	PWM (1000-2000µs)	Standard servo PWM protocol
Refresh Rate	50Hz - 8kHz	Configurable update frequency
BEC Output	5V/3A	Built-in voltage regulator

ESC Pin Configuration

Pin	Color	Function	Connection
Positive	Red	Power Input	Battery positive terminal
Signal	White/Yellow	PWM Input	Arduino PWM pin (D3 - D6; D9 - D11)
Ground	Black/Brown	Ground	Battery negative or Arduino GND
Motor Phase A	Blue	AC Output	Motor wire A
Motor Phase B	Yellow	AC Output	Motor wire B
Motor Phase C	Red	AC Output	Motor wire C

Arduino Interface Pins

Pin	Function	Description
Digital PWM	Motor Control	Generate 1000-2000µs PWM signal
5V	ESC Power	Optional BEC power input
GND	Ground	Shared reference voltage
A0-A5	Sensor Inputs	Potentiometers, encoders, sensors

3s Li-Po Battery

The 3S Li-Po (Lithium Polymer) battery configuration represents one of the most versatile power solutions for Arduino and embedded systems applications. With a nominal voltage of 11.1V (3.7V per cell) and full charge voltage of 12.6V, this battery pack delivers reliable, high-density power storage in a lightweight package. The 2200mAh capacity provides substantial runtime for moderate power consumption projects while maintaining portability essential for mobile robotics, IoT devices, and portable instrumentation.

Li-Po batteries offer significant advantages over traditional power sources including high energy density, low self-discharge rates, and consistent voltage output throughout the discharge cycle.

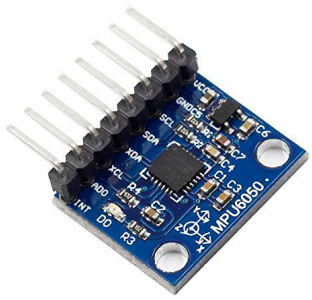
These characteristics make them particularly suitable for embedded systems requiring stable power delivery and extended operation periods without frequent recharging.

Battery Specifications

Parameter	Value
Configuration	3 cells in series
Normal Voltage	11.2V
Fully Charged Voltage	12.6V
Weigh	
Dimension	

The 12V output is ideal for powering servo motors, stepper motors, and DC gear motors commonly used in robotic platforms. The high current capability supports multiple actuators simultaneously while providing sufficient voltage for motor drivers like L298N, Electronic Speed Controller, or similar circuits.

MPU6050



The MPU6050 is a versatile **6-axis motion tracking** device that combines a **3-axis gyroscope** and a **3-axis accelerometer** on a single chip. Manufactured by InvenSense (now part of TDK), this compact sensor has become a cornerstone component in Arduino projects and embedded systems due to its exceptional accuracy, low power consumption, and ease of integration. The module operates on the **I2C communication protocol**, making it simple to interface with microcontrollers while providing precise motion sensing capabilities.

In the realm of embedded systems and Arduino applications, the MPU6050 serves as an essential building block for projects requiring orientation detection, motion sensing, and inertial measurement. Its ability to measure angular velocity and linear acceleration simultaneously makes it invaluable for robotics, drone control systems, gaming controllers, fitness trackers, and countless IoT applications where spatial awareness is crucial.

Technical Specifications

Parameter	Value
Operating Voltage	3.3V - 5V
Communication Protocol	I2C (400kHz max)

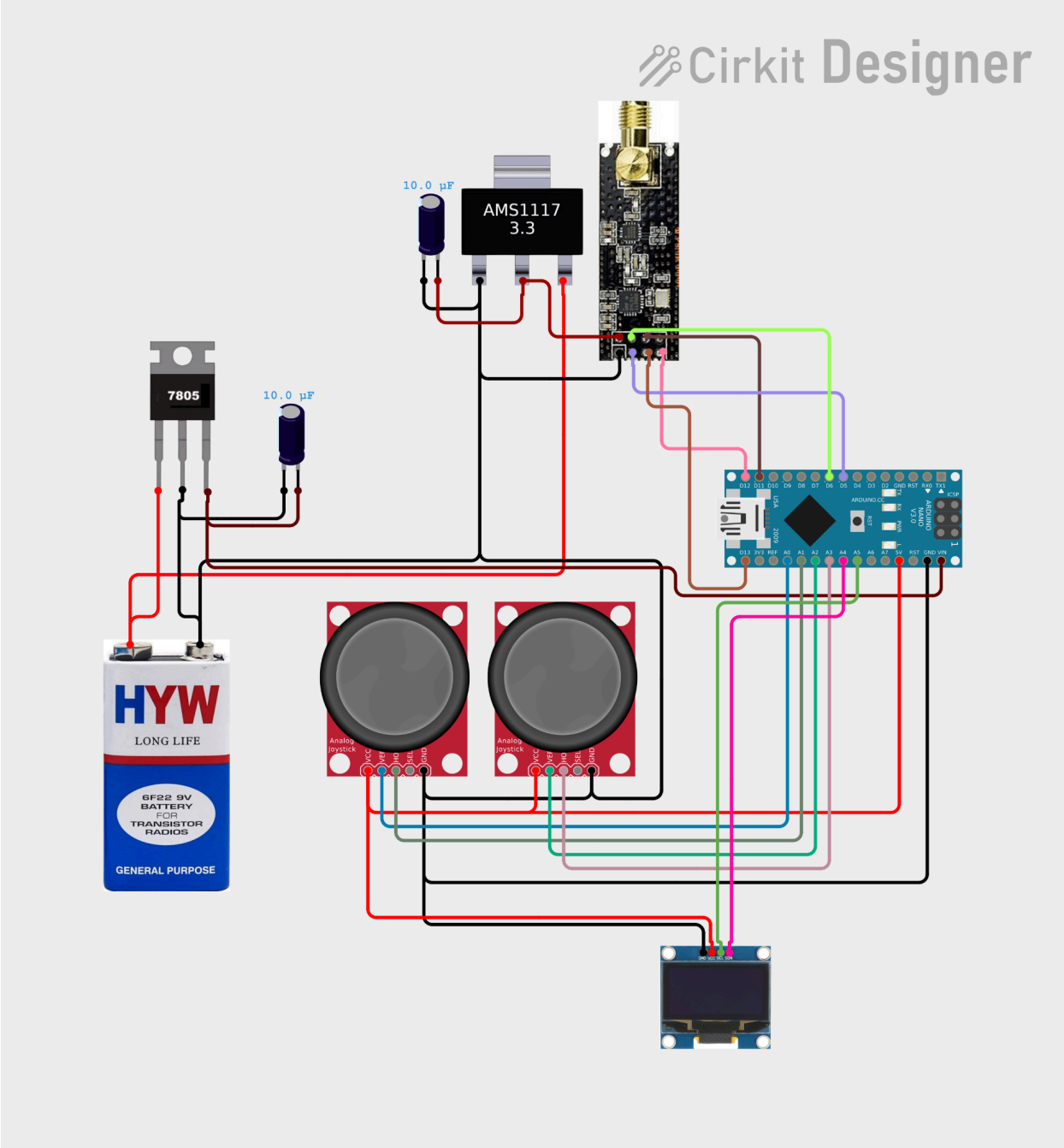
Gyroscope Range	±250, ±500, ±1000, ±2000 °/sec
Accelerometer Range	±2g, ±4g, ±8g, ±16g
Operating Temperature	-40°C to +85°C
Dimensions	4mm × 4mm × 0.9mm
Current Consumption	3.9mA

Pin Configuration

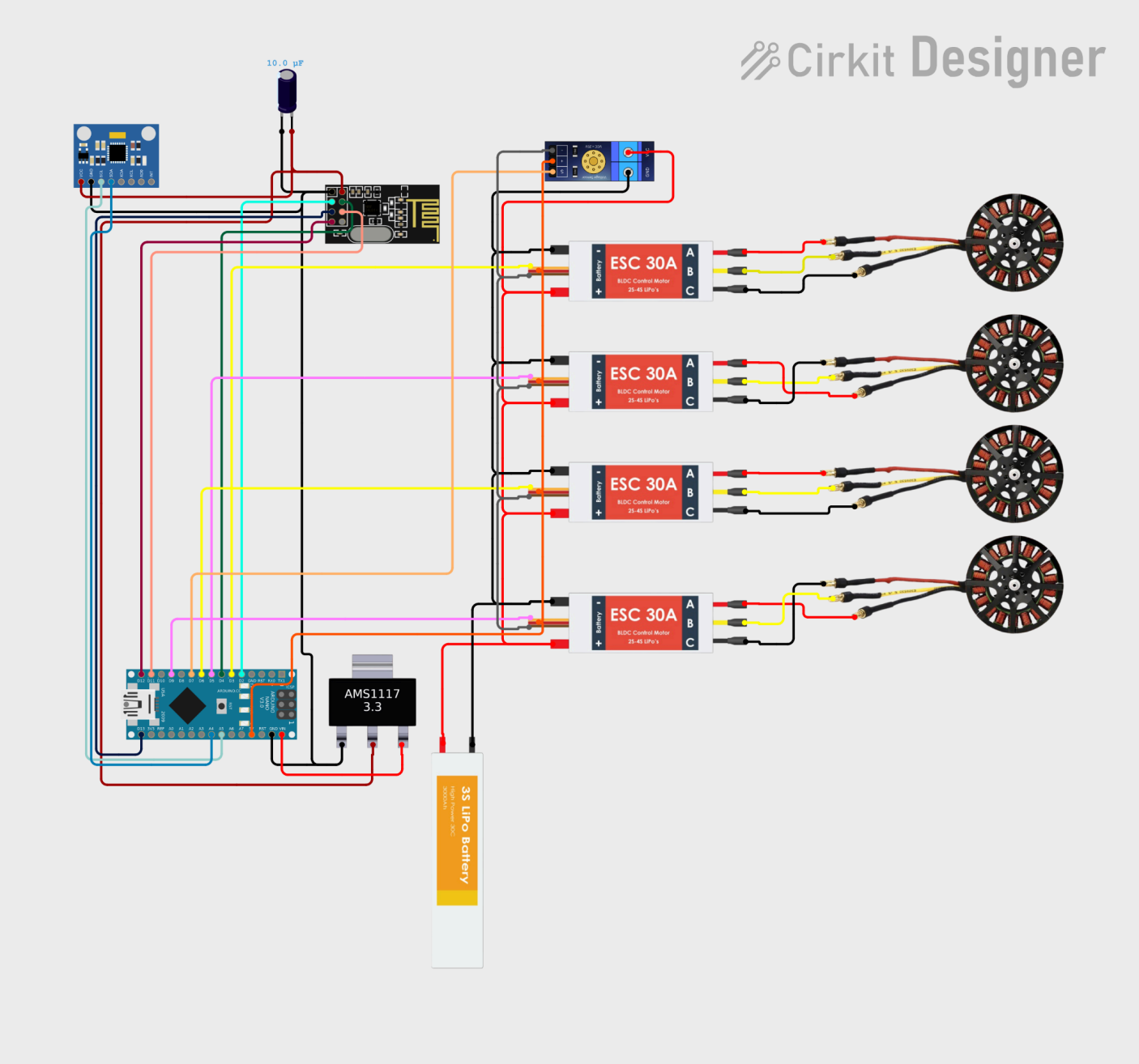
Pin	Function	Description
VCC	Power Supply	3.3V - 5V DC power input
GND	Ground	Common ground connection
SCL	Serial Clock	I2C clock line
SDA	Serial Data	I2C data line
XDA	Auxiliary Data	Auxiliary I2C data
XCL	Auxiliary Clock	Auxiliary I2C clock
AD0	Address Select	I2C address selection
INT	Interrupt	Digital interrupt output

The MPU6050 incorporates several advanced features that make it particularly suitable for embedded applications. Its Digital Motion Processor (DMP) can perform complex motion processing algorithms directly on-chip, reducing the computational load on the host microcontroller. The sensor includes a 16-bit analog-to-digital converter for each channel, ensuring high precision measurements. Additionally, it features programmable digital filters, multiple power management options, and built-in temperature compensation for enhanced accuracy across varying environmental conditions.

TRANSMITTER CIRCUIT



RECEIVER CIRCUIT



TRANSMITTER CODE

```
#include <Adafruit_SSD1306.h>
#include <splash.h>
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>

#define CE_PIN 5
#define CSN_PIN 6
#define JOY_THROTTLE A0
#define JOY_ROLL A1
#define JOY_PITCH A2
#define JOY_YAW A3
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

RF24 radio(CE_PIN, CSN_PIN);
const byte address[][6] = {"98830", "66310"};
int joystick[4];

void setup()
{
   Serial.begin(9600);
   radio.begin();
   radio.openWritingPipe(address);
   radio.setPALevel(RF24_PA_MIN);
   if (!display.begin(SSD1306_SWITCHCAPVCC, 0x3C))
   {
       Serial.println("OLED allocation failed");
   }
   beautifulInitialization();

   display.clearDisplay();
   display.setTextSize(2);
   display.setCursor(0, 16);
   display.println(" Ready to\ntake off!!");
   display.display();
}

// Beautiful animated initialization for SSD1306 OLED
void beautifulInitialization()
{
   display.clearDisplay();

   // Phase 1: Animated dots appearing
   for(int phase = 0; phase < 4; phase++){
       display.clearDisplay();

       // Draw expanding circles
       for(int i = 0; i <= phase; i++) {
           int x = 32 + i * 16;
           int y = 20;
           display.fillCircle(x, y, 3 + (phase - i), SSD1306_WHITE);
```

```
    }

    // Pulsing center logo/icon
    int pulseSize = 2 + sin(millis() * 0.01) * 2;
    display.fillRect(60, 8, 8, 8, SSD1306_WHITE);
    display.fillRect(58, 10, 12, 4, SSD1306_WHITE);

    display.display();
    delay(300);
}

// Phase 2: Progress bar animation
for(int progress = 0; progress <= 100; progress += 5) {
    display.clearDisplay();

    // Title with modern font effect
    display.setTextSize(1);
    display.setTextColor(SSD1306_WHITE);
    display.setCursor(35, 5);
    display.println("SYSTEM");

    // Stylized progress bar with rounded edges
    int barWidth = map(progress, 0, 100, 0, 96);

    // Background bar
    display.drawRect(16, 25, 96, 8, SSD1306_WHITE);

    // Progress fill with animated segments
    for(int i = 0; i < barWidth; i += 4) {
        if((i / 4) % 2 == 0) {
            display.fillRect(17 + i, 26, min(3, barWidth - i), 6, SSD1306_WHITE);
        }
    }

    // Percentage display
    display.setTextSize(1);
    display.setCursor(55, 40);
    display.print(progress);
    display.println("%");

    // Animated corner decorations
    int corner = (progress / 10) % 4;
    display.fillTriangle(0, 0, 8, 0, 0, 8, SSD1306_WHITE);
    if(corner >= 1) display.fillTriangle(120, 0, 128, 0, 128, 8, SSD1306_WHITE);
    if(corner >= 2) display.fillTriangle(128, 56, 128, 64, 120, 64, SSD1306_WHITE);
    if(corner >= 3) display.fillTriangle(0, 64, 8, 64, 0, 56, SSD1306_WHITE);

    display.display();
    delay(100);
}

// Phase 3: Success animation
display.clearDisplay();

// Final hold
delay(800);
```

```
    display.clearDisplay();
    display.display();
}

void loop()
{
    float VIN;  //Battery voltage
    float HGT;  //Height
    delay(5);
    radio.stopListening();
    joystick[0] = analogRead(JOY_THROTTLE) / 4;
    joystick[1] = analogRead(JOY_ROLL) / 4;
    joystick[2] = analogRead(JOY_PITCH) / 4;
    joystick[3] = analogRead(JOY_YAW) / 4;
    radio.write(&joystick, sizeof(joystick));
    delay(5);
    radio.startListening();

    while (!radio.available())
    {
        display.clearDisplay();
        display.setCursor(0, 0);
        display.print("NO SIGNAL");
        display.display();
    }

    radio.read(&VIN, sizeof(VIN));
    radio.read(&HGT, sizeof(HGT));
    display.clearDisplay();
    display.setCursor(0, 16);

    display.setTextSize(1);
    display.print("Battery:");
    display.setTextSize(2);
    display.print(VIN);
    display.println("V");

    display.setTextSize(1);
    display.print("\nHeight:");
    display.setTextSize(2);
    display.print(HGT);
    display.println("m\n");

    display.display();
}
```

Code Explanation

`setup()`

Purpose: Initializes all hardware components and displays startup screen

Process:

1. **Serial Communication:** Starts at 9600 baud for debugging
2. **Radio Setup:**
    - Initializes nRF24L01 transceiver
    - Opens writing pipe with address "98830"
    - Sets power to minimum (RF24_PA_MIN) for close-range operation
3. **OLED Display Setup:**
    - Initializes SSD1306 display at I2C address 0x3C
    - Checks if display allocation succeeded
    - If failed, prints error to serial
4. **Startup Animation:** Calls `beautifulInitialization()` for visual feedback
5. **Ready Screen:** Shows "Ready to take off!!" message in large text

`beautifulInitialization()`

Purpose: Creates an animated startup sequence on the OLED display

Three-Phase Animation:

1. Expanding Circles
2. Progress Bar
3. Final Hold

`loop()`

Purpose: Main control loop that handles joystick input, radio communication, and display updates

Process Flow:

1. **Joystick Reading**
    - Reads 4 analog joystick channels (A0-A3)
    - Divides by 4 to convert 10-bit ADC (0-1023) to 8-bit range (0-255)
    - Maps to: throttle, roll, pitch, yaw controls
2. **Data Transmission**
    - Switches radio to transmit mode
    - Sends 4-integer joystick array to drone
    - Array contains all control inputs in one packet
3. **Reception Mode**
    - Switches radio to receive mode
    - Blocking wait for incoming data from drone
    - Shows `NO SIGNAL` warning if no data received
    - This ensures display only updates when drone is responding
4. **Telemetry Reception**
    - Receives battery voltage (VIN) in first transmission
    - Receives height data (HGT) in second transmission
5. **Display Update**
    - Clears previous display content
    - Shows battery voltage with mixed text sizes
    - Shows height with mixed text sizes
    - Updates physical display with new content

RECEIVER CODE

```
#include <SPI.h>
#include <nRF24L01.h>
#include <RF24.h>
#include <Wire.h>
#include <MPU6050.h>
#include <Servo.h>
#include <Vcc.h>

const float VccMin = 0.0;
const float VccMax = 3.303;
const float VccCorrection = 0.98544;
const float lowBat = 12.0;
const float fullBat = 14.2;

#define CE_PIN 2
#define CSN_PIN 4
#define MOTOR1 3
#define MOTOR2 5
#define MOTOR3 6
#define MOTOR4 9
#define VSENSOR_PIN A0

#define KP 1.2
#define KI 0.02
#define KD 0.8

#define INTEGRAL_LIMIT 400.0 // Maximum integral accumulation
#define ANGLE_LIMIT 45.0     // Maximum angle in degrees
#define ALPHA 0.98 // Weight for gyro data (0.98 = 98% gyro, 2% accel)

unsigned long previousTime = 0;
float deltaTime = 0;

RF24 radio(CE_PIN, CSN_PIN);
const byte address[][6] = {"98830", "66310"};
MPU6050 mpu;
Servo esc1, esc2, esc3, esc4;

int joystick[4];
float batt[2];

float rollAngle = 0, pitchAngle = 0; // Filtered angles (degrees)
float rollRate = 0, pitchRate = 0;   // Gyro rates (degrees/sec)
float rollAccel = 0, pitchAccel = 0; // Accelerometer angles (degrees)

float errorRoll = 0, errorPitch = 0;
float previousErrorRoll = 0, previousErrorPitch = 0;
float integralRoll = 0, integralPitch = 0;

float verticalAccel = 0;          // Vertical acceleration (m/s²)
float verticalVelocity = 0;       // Vertical velocity (m/s)
float estimatedHeight = 0;        // Estimated height (meters)
float baselineAccel = 0;          // Baseline acceleration for calibration
bool heightCalibrated = false;    // Calibration flag
```

```
float heightData[2];              // Array to transmit height data [height, velocity]

#define ACCEL_SCALE 16384.0       // LSB/g for ±2g range
#define GRAVITY 9.81              // m/s²
#define HEIGHT_FILTER_ALPHA 0.85  // Complementary filter for height
#define MIN_ACCEL_THRESHOLD 0.1   // Minimum acceleration threshold (m/s²)

void heightCalc() {
   // Get raw accelerometer data
   int16_t ax, ay, az, gx, gy, gz;
   mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

   // Convert raw Z-axis acceleration to m/s²
   float rawVerticalAccel = (az / ACCEL_SCALE) * GRAVITY;

   // Calibrate baseline if not done yet
   if (!heightCalibrated) {
       calibrateHeightBaseline();
       return;
   }

   // Remove gravity and baseline offset
   verticalAccel = rawVerticalAccel - baselineAccel - GRAVITY;

   // Apply low-pass filter to reduce noise
   static float filteredAccel = 0;
   filteredAccel = HEIGHT_FILTER_ALPHA * filteredAccel + (1.0 - HEIGHT_FILTER_ALPHA) *
verticalAccel;

   // Apply dead zone to eliminate small noise
   if (abs(filteredAccel) < MIN_ACCEL_THRESHOLD) {
       filteredAccel = 0;
   }

   // Integrate acceleration to get velocity
   verticalVelocity += filteredAccel * deltaTime;

   // Apply velocity damping to prevent drift
   verticalVelocity *= 0.999;

   // Integrate velocity to get height
   estimatedHeight += verticalVelocity * deltaTime;

   // Prevent negative height (ground level)
   if (estimatedHeight < 0) {
       estimatedHeight = 0;
       verticalVelocity = 0;
   }

   // Prepare data for transmission
   heightData[0] = estimatedHeight;          // Height in meters
   heightData[1] = verticalVelocity;         // Vertical velocity in m/s

   // Transmit height data
   transmitHeightData();

   // Debug output
```

```
    Serial.print("Height: ");
    Serial.print(estimatedHeight, 2);
    Serial.print("m, Velocity: ");
    Serial.print(verticalVelocity, 2);
    Serial.print("m/s, Accel: ");
    Serial.print(filteredAccel, 2);
    Serial.println("m/s²");
}

void calibrateHeightBaseline() {
    static int calibrationSamples = 0;
    static float accelSum = 0;
    const int CALIBRATION_SAMPLES = 200;

    // Get raw accelerometer data
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    // Convert raw Z-axis acceleration to m/s²
    float rawVerticalAccel = (az / ACCEL_SCALE) * GRAVITY;

    accelSum += rawVerticalAccel;
    calibrationSamples++;

    if (calibrationSamples >= CALIBRATION_SAMPLES) {
        baselineAccel = accelSum / calibrationSamples;
        heightCalibrated = true;

        Serial.print("Height calibration complete. Baseline: ");
        Serial.print(baselineAccel, 4);
        Serial.println(" m/s²");

        // Reset height tracking variables
        estimatedHeight = 0;
        verticalVelocity = 0;
        verticalAccel = 0;
    } else {
        Serial.print("Height calibration: ");
        Serial.print((calibrationSamples * 100) / CALIBRATION_SAMPLES);
        Serial.println("%");
    }
}

void transmitHeightData() {
    // Stop listening to transmit data
    radio.stopListening();

    // Switch to transmit pipe
    radio.openWritingPipe(address[1]);

    // Transmit height data
    bool result = radio.write(&heightData, sizeof(heightData));

    if (result) {
        Serial.println("Height data transmitted successfully");
    } else {
```

```
        Serial.println("Height data transmission failed");
    }

    // Switch back to listening mode
    radio.openReadingPipe(0, address[0]);
    radio.startListening();
}

void resetHeightTracking() {
    // Function to reset height tracking (call when drone lands or resets)
    estimatedHeight = 0;
    verticalVelocity = 0;
    verticalAccel = 0;
    heightCalibrated = false;

    Serial.println("Height tracking reset");
}

void setup()
{
    Serial.begin(115200);

    // Initialize radio
    radio.begin();
    radio.openReadingPipe(0, address[0]);
    radio.setPALevel(RF24_PA_MIN);

    // Initialize I2C and MPU6050
    Wire.begin();
    mpu.initialize();

    // Check if MPU6050 is connected
    if (mpu.testConnection())
    {
        Serial.println("MPU6050 connected successfully");
    }
    else
    {
        Serial.println("MPU6050 connection failed");
    }
    calibrateGyro();

    // Initialize ESCs
    esc1.attach(MOTOR1);
    esc2.attach(MOTOR2);
    esc3.attach(MOTOR3);
    esc4.attach(MOTOR4);

    initializeESCs();
    previousTime = millis();
    calibrateGyro();

    Serial.println("Setup complete");
}

void loop()
```

```
{
   // Calculate delta time for integration
   unsigned long currentTime = millis();
   deltaTime = (currentTime - previousTime) / 1000.0; // Convert to seconds
   previousTime = currentTime;

   // Ensure reasonable deltaTime (prevent division issues)
   if (deltaTime > 0.1)
       deltaTime = 0.01; // Cap at 100ms

   heightCalc();

   delay(5);
   radio.startListening();

   if (radio.available())
   {
       voltRead();
       Serial.println("Radio available");

       while (radio.available())
       {
           radio.read(&joystick, sizeof(joystick));

           // Map joystick inputs
           int throttle = map(joystick[0], 0, 255, 1000, 2000);
           float rollTarget = map(joystick[1], 0, 255, -30, 30);
           float pitchTarget = map(joystick[2], 0, 255, -30, 30);
           int yawInput = map(joystick[3], 0, 255, -500, 500);

           // Get sensor data and apply complementary filter
           getMPU6050DataWithFilter();

           // Apply PID control with improved algorithm
           applyImprovedPIDControl(throttle, rollTarget, pitchTarget, yawInput);

           // Debug output
           Serial.print("Roll: ");
           Serial.print(rollAngle, 2);
           Serial.print("° Target: ");
           Serial.print(rollTarget, 2);
           Serial.print("° Throttle: ");
           Serial.println(throttle);
       }

       delay(5);
       radio.stopListening();

       // Battery monitoring and transmission
       transmitBatteryData();
   }
}

void calibrateGyro()
{
   Serial.println("Calibrating gyroscope... Keep drone still!");
```

```
    // Take 1000 samples to establish zero point
    long sumGx = 0, sumGy = 0, sumGz = 0;
    for (int i = 0; i < 1000; i++)
    {
        int16_t ax, ay, az, gx, gy, gz;
        mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);
        sumGx += gx;
        sumGy += gy;
        sumGz += gz;
        delay(3);
    }

    // Calculate offsets (these should be subtracted from future readings)
    // For simplicity, we're not storing offsets in this example
    // In a full implementation, you'd store these and subtract them

    Serial.println("Gyroscope calibration complete");
}

void initializeESCs()
{
    Serial.println("Initializing ESCs...");

    // Send minimum throttle to all ESCs
    esc1.writeMicroseconds(1000);
    esc2.writeMicroseconds(1000);
    esc3.writeMicroseconds(1000);
    esc4.writeMicroseconds(1000);

    delay(2000); // Wait for ESCs to recognize the signal
    Serial.println("ESCs initialized");
}

void getMPU6050DataWithFilter()
{
    int16_t ax, ay, az, gx, gy, gz;
    mpu.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    // Convert gyroscope data to degrees per second
    rollRate = gx / 131.0;
    pitchRate = gy / 131.0;

    // Convert accelerometer data to angles (in degrees)
    rollAccel = atan2(ay, az) * 180.0 / PI;
    pitchAccel = atan2(-ax, sqrt(ay * ay + az * az)) * 180.0 / PI;

    // Apply complementary filter
    // Integrate gyro data and combine with accelerometer
    rollAngle = ALPHA * (rollAngle + rollRate * deltaTime) + (1.0 - ALPHA) * rollAccel;
    pitchAngle = ALPHA * (pitchAngle + pitchRate * deltaTime) + (1.0 - ALPHA) * pitchAccel;

    // Constrain angles to reasonable limits
    rollAngle = constrain(rollAngle, -ANGLE_LIMIT, ANGLE_LIMIT);
    pitchAngle = constrain(pitchAngle, -ANGLE_LIMIT, ANGLE_LIMIT);
}
```

```
void applyImprovedPIDControl(int throttle, float rollTarget, float pitchTarget, int yawInput)
{
   // Calculate errors (target - actual)
   errorRoll = rollTarget - rollAngle;
   errorPitch = pitchTarget - pitchAngle;

   // Accumulate errors for integral term
   integralRoll += errorRoll * deltaTime;
   integralPitch += errorPitch * deltaTime;

   // Apply integral windup protection
   integralRoll = constrain(integralRoll, -INTEGRAL_LIMIT, INTEGRAL_LIMIT);
   integralPitch = constrain(integralPitch, -INTEGRAL_LIMIT, INTEGRAL_LIMIT);

   // Reset integral if error changes sign (prevents windup)
   if ((errorRoll > 0 && previousErrorRoll < 0) || (errorRoll < 0 && previousErrorRoll > 0))
   {
       integralRoll *= 0.5; // Reduce integral by half
   }
0))if ((errorPitch > 0 && previousErrorPitch < 0) || (errorPitch < 0 && previousErrorPitch >
   {
       integralPitch *= 0.5; // Reduce integral by half
   }

   float derivativeRoll = (errorRoll - previousErrorRoll) / deltaTime;
   float derivativePitch = (errorPitch - previousErrorPitch) / deltaTime;

   // Apply derivative filter to reduce noise
   static float lastDerivativeRoll = 0, lastDerivativePitch = 0;
   derivativeRoll = 0.7 * lastDerivativeRoll + 0.3 * derivativeRoll;
   derivativePitch = 0.7 * lastDerivativePitch + 0.3 * derivativePitch;
   lastDerivativeRoll = derivativeRoll;
   lastDerivativePitch = derivativePitch;

   // Calculate PID output
   float pidRoll = (KP * errorRoll) + (KI * integralRoll) + (KD * derivativeRoll);
   float pidPitch = (KP * errorPitch) + (KI * integralPitch) + (KD * derivativePitch);

   // Store current errors for next iteration
   previousErrorRoll = errorRoll;
   previousErrorPitch = errorPitch;

   pidRoll = constrain(pidRoll, -500, 500);
   pidPitch = constrain(pidPitch, -500, 500);

   if (throttle > 1100)
   {
       int motor1Speed = constrain(throttle + pidRoll + pidPitch - yawInput, 1000, 2000);
       int motor2Speed = constrain(throttle + pidRoll - pidPitch + yawInput, 1000, 2000);
       int motor3Speed = constrain(throttle - pidRoll - pidPitch - yawInput, 1000, 2000);
       int motor4Speed = constrain(throttle - pidRoll + pidPitch + yawInput, 1000, 2000);

       esc1.writeMicroseconds(motor1Speed);
       esc2.writeMicroseconds(motor2Speed);
       esc3.writeMicroseconds(motor3Speed);
```

```
        esc4.writeMicroseconds(motor4Speed);
    }
    else
    {
        esc1.writeMicroseconds(1000);
        esc2.writeMicroseconds(1000);
        esc3.writeMicroseconds(1000);
        esc4.writeMicroseconds(1000);

        integralRoll = 0;
        integralPitch = 0;

    resetHeightTracking();
    }
}

void voltRead()
{
    int i;
    float batt_volt = 0;

    for (i = 0; i < 100; i++)
    {
        batt_volt += ((analogRead(VSENSOR_PIN) * (4.46 / 1023.0)) * 5.5);
        delay(1); // Reduced delay for faster sampling
    }

    batt_volt = batt_volt / i;

    Serial.print("Battery: ");
    Serial.print(batt_volt, 2);
    Serial.print("V ");
    Serial.print(((batt_volt - lowBat) / (fullBat - lowBat)) * 100);
    Serial.println("%");
}

void transmitBatteryData()
{
    int i;
    batt[0] = 0;

    for (i = 0; i < 50; i++) // Reduced samples for faster execution
    {
        batt[0] += ((analogRead(VSENSOR_PIN) * (4.46 / 1023.0)) * 5.5);
        delay(2);
    }

    batt[0] = batt[0] / i;
    batt[1] = ((batt[0] - lowBat) / (fullBat - lowBat)) * 100;

    radio.write(&batt, sizeof(batt));
    delay(50); // Reduced delay
}
```

Code Explanation

`setup()`

Purpose: Initializes all systems before main loop

Process:

1. Starts serial communication at 115200 baud
2. Initializes nRF24L01 radio with receive address
3. Sets up I2C communication for MPU6050
4. Tests MPU6050 connection
5. Calibrates gyroscope
6. Attaches ESC servo objects to motor pins
7. Initializes ESCs with minimum throttle signal
8. Records start time for deltaTime calculations

`heightCalc()`

Purpose: Main height estimation function that runs continuously

Process:

1. Gets raw accelerometer data from MPU6050
2. Converts Z-axis acceleration from raw units to m/s²:

$$\frac{a_z}{16384} \times 9.18$$

3. If not calibrated, calls calibration function and exits
4. Removes gravity and sensor bias:

$$Vertical\ Acceleration = Raw\ vertical\ acceleration - Baseline\ acceleration - Gravity$$

5. Applies low-pass filter to reduce noise:

$$Filtered\ Acceleration = 0.85 \times old\ value + 0.15 \times new\ value$$

6. Creates dead zone to eliminate small vibrations (< 0.1 m/s²)
7. **Double integration:** acceleration → velocity → height
8. Applies velocity damping (×0.999) to prevent drift
9. Prevents negative height (ground constraint)
10. Transmits height data and prints debug info

`calibrateHeightBaseline()`

Purpose: Establishes the sensor's baseline acceleration to compensate for mounting errors and bias

Process:

1. Takes 200 samples of Z-axis acceleration
2. Calculates average baseline acceleration
3. Sets calibration flag when complete
4. Resets all height tracking variables
5. Shows calibration progress percentage

This is critical because accelerometers have bias and the drone may not be perfectly level when powered on.

`transmitHeightData()`

Purpose: Sends height and velocity data to ground station via radio

Process:

1. Stops radio listening mode
2. Switches to transmit pipe (address[1])
3. Sends heightData array containing [height, velocity]
4. Checks transmission success
5. Switches back to receive mode

`resetHeightTracking()`

Purpose: Resets all height-related variables (called when drone lands)

Process:

- Zeros height, velocity, and acceleration
- Clears calibration flag
- Forces recalibration on next flight

`calibrateGyro()`

Purpose: Determines gyroscope zero-point offsets

Process:

1. Takes 1000 samples while drone is stationary
2. Calculates average values for X, Y, Z gyro axes
3. **Note:** This version doesn't store the offsets (incomplete implementation)
4. In a complete system, these offsets would be subtracted from all future readings

`initializeESCs()`

Purpose: Arms the Electronic Speed Controllers

Process:

1. Sends 1000μs pulse width to all ESCs (minimum throttle)
2. Waits 2 seconds for ESCs to recognize and arm
3. This prevents unexpected motor startup

`getMPU6050DataWithFilter()`

Purpose: Processes raw sensor data into usable angles

Process:

1. **Raw Data:** Gets 6-axis data (3 accel + 3 gyro)
2. **Gyro Conversion:** Divides by 131.0 to get degrees/second
3. **Accel to Angles:** Uses trigonometry:
   - $Roll = arctan2(a_y, a_z) \times \frac{180}{2\pi}$
   - $Pitch = arctan2\left(-a_x, \sqrt{a_y^2 + a_z^2}\right) \times \frac{180}{2\pi}$
4. **Complementary Filter:** Combines gyro and accel data:

   $angle = (0.98 \times gyroscope\ integration) + (002 \times accelerometer\ angle)$

5. **Safety:** Constraints angles to ±45°

`applyImprovedPIDControl()`

Purpose: Main flight stabilization algorithm

Process:

1. **Error Calculation:** `error = target - actual`
2. **Integral Term:** Accumulates error over time, with windup protection
3. **Derivative Term:** Calculates rate of error change, applies noise filtering
4. **PID Output:** Combines all three terms with gains (KP=1.2, KI=0.02, KD=0.8)
5. **Motor Mixing:** Distributes control signals to 4 motors in X-configuration:
   - Motor 1: `throttle + roll + pitch - yaw`
   - Motor 2: `throttle + roll - pitch + yaw`
   - Motor 3: `throttle - roll - pitch - yaw`
   - Motor 4: `throttle - roll + pitch + yaw`
6. **Safety:** Only applies stabilization if throttle > 1100, otherwise stops motors and resets

`voltRead()`

Purpose: Measures battery voltage for monitoring

Process:

1. Takes 100 analog readings for accuracy
2. Converts ADC value to voltage:

$$Voltage = 4 \times ADC \times \frac{4.49}{1023}$$

3. Calculates battery percentage using min/max voltages
4. Displays voltage and percentage

`transmitBatteryData()`

Purpose: Sends battery telemetry to ground station

Process:

1. Takes 50 samples (faster than voltRead)
2. Calculates voltage and percentage
3. Transmits via radio
4. Uses same voltage conversion formula

`loop()`

Purpose: Main control loop running

Process:

1. **Time Management:** Calculates deltaTime for integration, caps at 100ms to prevent errors
2. **Height Calculation:** Calls heightCalc() every loop
3. **Radio Communication:**
    - Checks for incoming joystick data
    - Reads 4-channel joystick array [throttle, roll, pitch, yaw]
    - Maps joystick values to usable ranges
4. **Flight Control:**
    - Gets filtered sensor data
    - Applies PID control
    - Outputs motor commands
5. **Telemetry:** Transmits battery data back to controller

ADVANTAGES

- **Modular Design**: Easily upgradable with sensors and modules (e.g., GPS, camera, Wi-Fi).
- **Wireless Control (RF):** Reliable real-time control in short to medium range.
- **Arduino Nano Compatibility:** Cost-effective, open-source, and highly customizable.
- **Inertial Measurement Unit:** Better flight stability through real-time pitch, roll, yaw monitoring.
- **Telemetry Feedback:** Displays battery voltage and height on OLED for live monitoring.
- **DIY-Friendly**: Built using readily available and affordable components.

DISADVANTAGES

- **Limited Range (RF):** RF communication is suitable only for short to medium distances.
- **No GPS Navigation:** Cannot autonomously navigate or return to home without GPS.
- **Battery Constraints:** Short flight time due to Li-Po battery capacity and motor consumption.
- **Lack of Obstacle Avoidance:** No ultrasonic or IR sensors to detect and avoid obstacles
- **Weather Dependency:** Performance can degrade in wind, rain, or harsh environmental conditions.

ERROR

Error Source	Typical Magnitude	Time to Instability
Gyro Drift (uncalibrated)	0.5-2°/minute	5-10 minutes
PID Derivative Error	20-50% overshoot	Slight oscillation
Complementary Filter	10-30% angle error	1-2 minutes
Voltage Reading	10-15% error	Gradual (affects flight time)

PRECAUTIONS

Flying a drone responsibly is crucial to ensure both safety and legal compliance. One of the most important rules is to always keep your drone within your line of sight—this helps prevent accidents and allows for immediate intervention if something goes wrong. Never fly your drone beyond visual range. Before taking off, check the weather conditions thoroughly. Flying in strong winds, rain, or during storms can result in loss of control and potential damage to your drone or harm to people and property.

It's essential to stay well away from sensitive locations such as airports, crowded public areas, or any form of restricted airspace. Not only is it dangerous to fly in these zones, but it may also violate airspace laws. Always inspect your drone before each flight to ensure there are no physical damages, and confirm that the batteries are fully charged and in good condition to avoid mid-air failure.

Choose open areas for flying where there are minimal obstacles like power lines, trees, or buildings that could interfere with your flight path. Be mindful of others' privacy and avoid flying over private property without prior permission. In many regions, drone registration is mandatory—make sure to register your device as required by your local aviation authority.

Plan your flight with safety in mind, including identifying a suitable landing area free from people, animals, and moving vehicles. Lastly, take the time to learn and follow all local drone laws and regulations to ensure your drone operations remain safe, legal, and respectful of your surroundings. Responsible flying protects you, your equipment, and the community.

CONCLUSION

This project successfully demonstrates the fundamental principles of quadcopter flight using Arduino Nano and RF communication. Through effective use of PID control, IMU sensor data, and real-time telemetry, the drone achieves stable, controlled flight and basic maneuvering. While it may not match commercial drones in terms of range or autonomy, this prototype offers a valuable learning platform for embedded systems, control theory, and wireless communication.

It not only strengthens the practical understanding of electronics and instrumentation but also opens doors to future innovations such as GPS integration, camera modules, autonomous navigation, and AI-based obstacle detection. With further development, this system has the potential to serve as a robust base for educational, industrial, and research-based drone applications.

BIBLIOGRAPHY

1. Arduino NANO Hardware Documentation - https://docs.arduino.cc/hardware/nano/
2. Getting Started with Arduino IDE 2 - Arduino Documentation - https://docs.arduino.cc/software/ide-v2/tutorials/getting-started-ide-v2/
3. Language Reference - Arduino Documentation - https://docs.arduino.cc/language-reference/
4. Quadcopter Drones: A Comprehensive Beginner's Guide - https://www.t-drones.com/blog/quadcopter-drones.html

1

2

3

4