Stair Climbing Transporter with a Self-Balancing Platform using Dual Core processors in ESP32 Development board.

Project Report

Embedded Systems and Internet of Things – CSC 376 1.5

Index No – AS2019459

University of Sri Jayewardenepura.

Table of Contents

Project Report 1
Embedded Systems and Internet of Things – CSC 376 1.5 1
Introduction and Background. 3
  Introduction 3
  Background 3
The Problem and Solution 4
  The Problem 4
  Proposed Solution 4
  Assumptions 4
The Functions of the Project 5
  Functions Description 5
    The Wheel Design 5
  Block Diagram 7
  Functional Diagram 7
The Hardware 8
  The Microcontroller 8
    Dual-Core Architecture 9
    Other Advantages 10
  Sensors 11
    MPU6050 Gyroscope Sensor 11
    Bluetooth 12
  Actuators 12
    Servo Motors 12
    DC motors 13
    Motor Driver – BTS7960 High Current Motor Driver 13
  Others 14
    LM2596S Buck Converter 14
    Spur Gears 14
  Power Requirements 14
Programming 15
  Approach 15
  Challenges faced. 15
    Calibrating MPU6050 Offset Errors 15

Writing Code for ESP32 board 19
Reading errors from MPU6050 with time 20
Problems with making use of Dual Core architecture. 20
The Final Code 22
Testing 22
Versions 26
Production Cost 27
Prototype Cost 27
Conclusion 27
Future Work 27
Project Resources 28

Introduction and Background.

Introduction

The aim of this project is to develop a stair climbing rover with a self-balancing platform capable of carrying items upstairs without spilling or toppling.

The most important thing is to make use of Dual core processors in ESP32 board instead of using 2 separate microcontrollers.

The rover utilizes advanced embedded systems and motor control techniques to achieve this functionality. By automating the process of carrying items upstairs, this rover can be a valuable tool in various applications such as logistics, delivery services, and home assistance.

Background

Traditional methods of carrying heavy items upstairs can be laborious and prone to accidents.

Existing solutions, such as using manual labor or elevators, may not be efficient or practical in certain scenarios.

This project draws inspiration from the field of robotics and aims to leverage modern technologies to address the challenges associated with stair climbing.

It also explores parallel processing with microcontrollers by using dual core processors in ESP32 Development board.

The Problem and Solution

The Problem

Carrying items upstairs manually or using traditional methods can be time-consuming, physically demanding, and pose a risk of accidents or damage to the items being transported. There is a need for an automated solution that can navigate stairs smoothly, maintain balance, and ensure the safety of both the rover and the items being carried.

Most microcontrollers can't do two separate tasks at the same time. Hence using two separate boards will take up more power and space requirements.

Proposed Solution

The proposed solution is a stair climbing rover equipped with a self-balancing platform. The rover uses a combination of micro-SG servo motors for self-balancing and DC motors for stair climbing. The self-balancing platform ensures that the items remain stable and upright during the ascent or descent on stairs, reducing the risk of spillage or toppling.

Using parallel processing, a single microcontroller board will be used to control two separate tasks.

1) Self-Balancing Platform.
2) Stair climbing RC car.

Assumptions

For the purpose of this report, we assume that the rover is designed to carry a maximum load of 200 grams. Additionally, it is assumed that the rover is capable of climbing stairs with a maximum height of 30mm. The project also assumes that the rover is powered by a rechargeable battery pack, providing sufficient power for extended operation.

The Functions of the Project

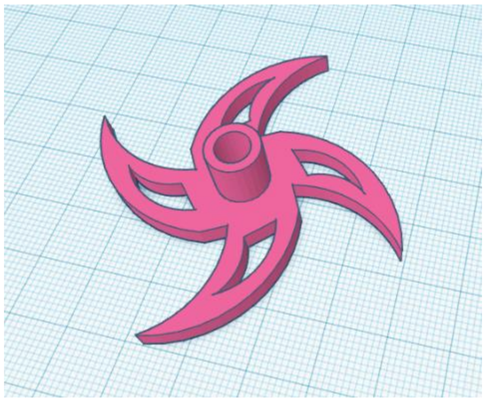
Functions Description

The stair climbing rover is designed to perform the following functions:

Stair Climbing: The rover is equipped with DC motors that enable it to climb stairs. Using appropriate control algorithms, the rover can autonomously navigate the steps. A very important aspect of this was the wheel design.

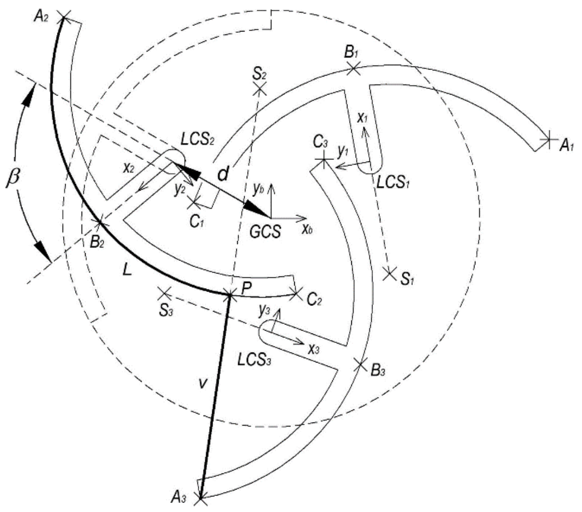
The Wheel Design

A special curved spoke wheel mechanism was used for the wheels. This design ensures that the rover is able to climb stairs. The design was made in sketchup.com and 3D printed.



Reasons for Selecting this wheel design.

- Geometric analysis determines the ideal number of circular turnable segments for a stair design.
- The analysis calculates the distance v and the length of the rolling curve L for the segments.



- Transformation matrices and coordinate systems are used for calculations.

$$\mathbf{T}_{b2} = \begin{bmatrix} \mathrm{c}(S)\cdot\mathrm{c}(\beta) - \mathrm{s}(S)\cdot\mathrm{s}(\beta)] & -\mathrm{c}(S)\cdot\mathrm{s}(\beta) - \mathrm{s}(S)\cdot\mathrm{c}(\beta) & 0 & d\cdot\mathrm{c}(S) \\ \mathrm{c}(S)\cdot\mathrm{s}(\beta) + \mathrm{s}(S)\cdot\mathrm{c}(\beta) & \mathrm{c}(S)\cdot\mathrm{c}(\beta) - \mathrm{s}(S)\cdot\mathrm{s}(\beta) & 0 & d\cdot\mathrm{s}(S) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$S = \frac{\pi\cdot(3s-4)}{2s}$$

sin(S) and cos(S) are abbreviated as s(S) and c(S), respectively. S is the angle of rotation of LCS2 relative to GCS at β = 0. S = π(3s-4)/(2s), where s is the number of segments (s = 3). β is the segment rotation angle, d is the distance between the segment rotation axis and the wheel center, and r is the wheel radius.*

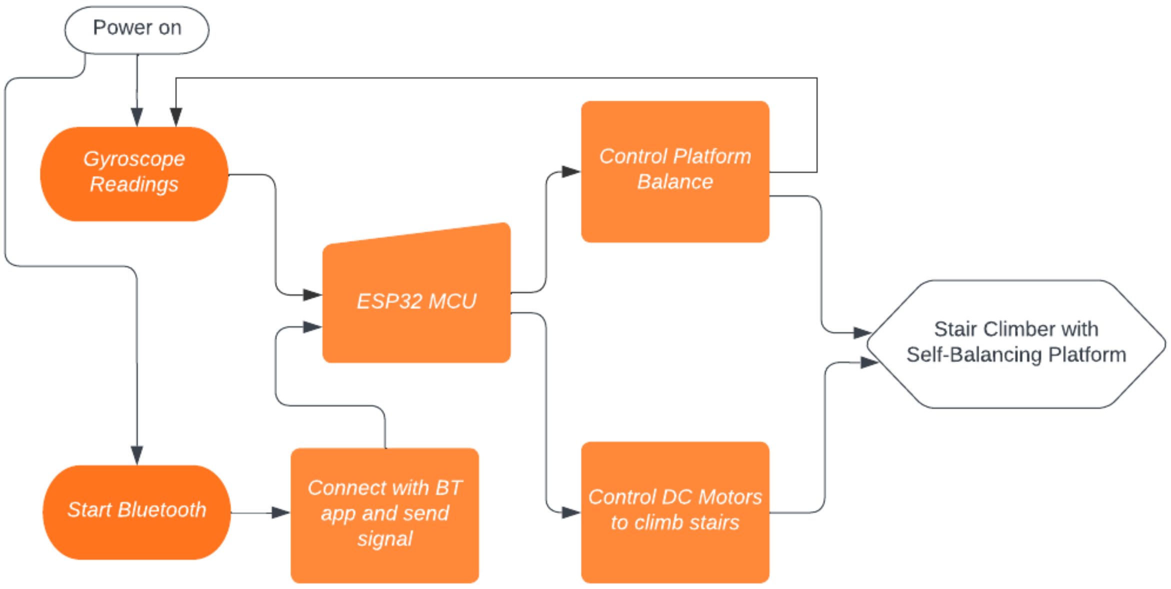
- An iterative optimization method is used to find suitable values for the segment's rotation parameters.
- The range of possible wheel dimensions is determined for different numbers of segments.
- Wheels with four inclinable segments and a diameter of 430 mm provide the best results.
- Wheels with three segments are also satisfactory, with a minimum feasible diameter of 304 mm.

Hence, 4 Segment wheels were used. Since recommended 430mm wheels will be too big for the scope of this project, wheel with diameter of 40mm was used, and tested on stairs of height up to 30mm.

Self-Balancing: The micro-SG servo motors are responsible for maintaining the balance of the rover's platform based on readings from the MPU6050 gyroscope sensor. They continuously monitor and adjust the platform's position to ensure stability, even on uneven surfaces. Further, a basic suspension system was used on the chassis of the rover to prevent overbalancing due to sudden rover movements.

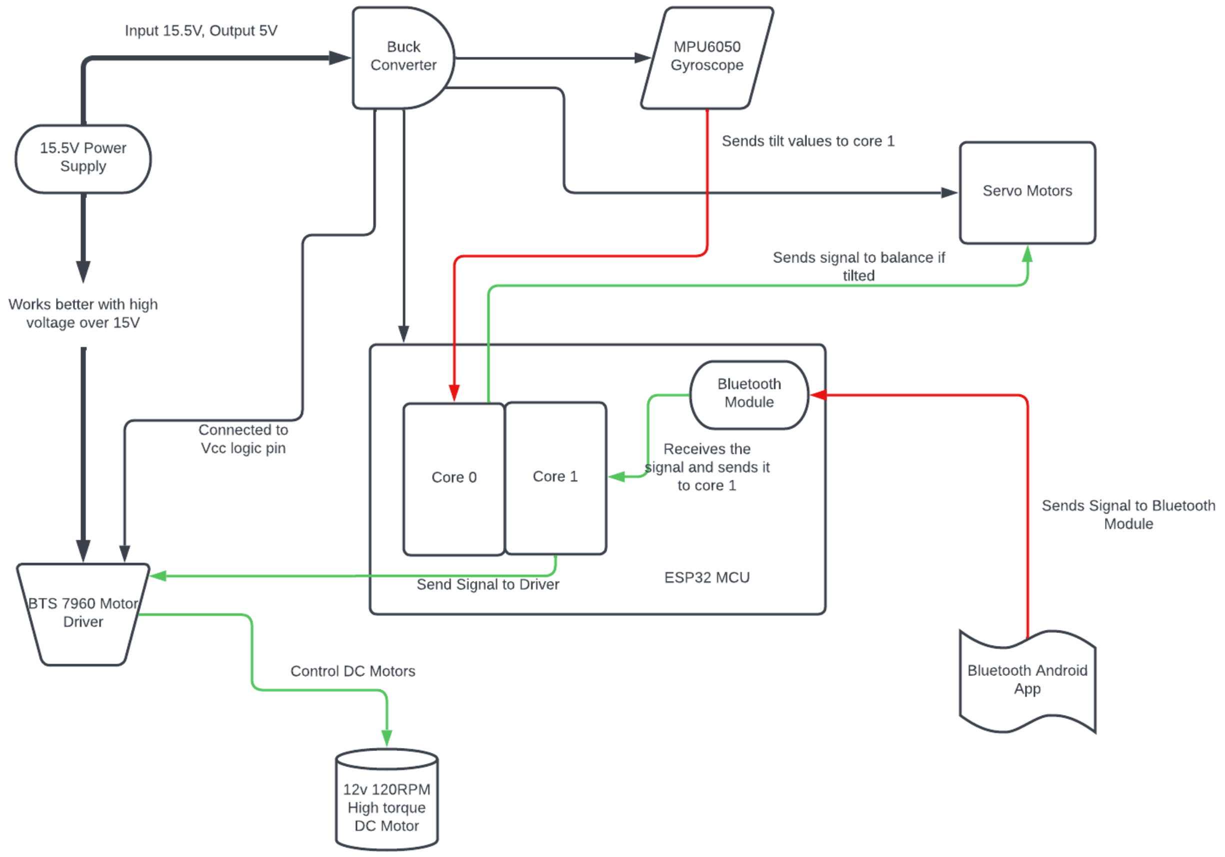
Load Carrying: The rover features a platform specifically designed to securely carry items. The self-balancing platform ensures that the load remains stable during stair climbing, reducing the risk of spillage or toppling.

Block Diagram


Power on
Gyroscope Readings
ESP32 MCU
Control Platform Balance
Control DC Motors to climb stairs
Start Bluetooth
Connect with BT app and send signal
Stair Climber with Self-Balancing Platform

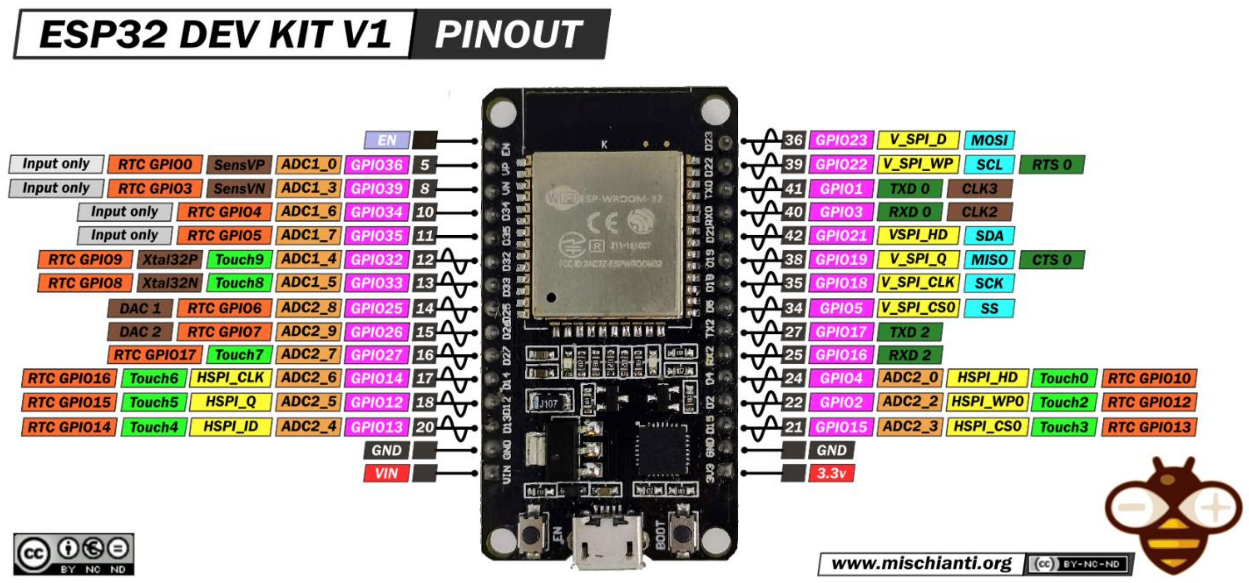
Functional Diagram

This shows the functions and the components of the project and how they interact with each other.


Input 15.5V, Output 5V
Buck Converter
MPU6050 Gyroscope
15.5V Power Supply
Sends tilt values to core 1
Servo Motors
Sends signal to balance if tilted
Works better with high voltage over 15V
Bluetooth Module
Connected to Vcc logic pin
Core 0
Core 1
Receives the signal and sends it to core 1
Sends Signal to Bluetooth Module
Send Signal to Driver
ESP32 MCU
BTS 7960 Motor Driver
Control DC Motors
Bluetooth Android App
12v 120RPM High torque DC Motor

The Hardware

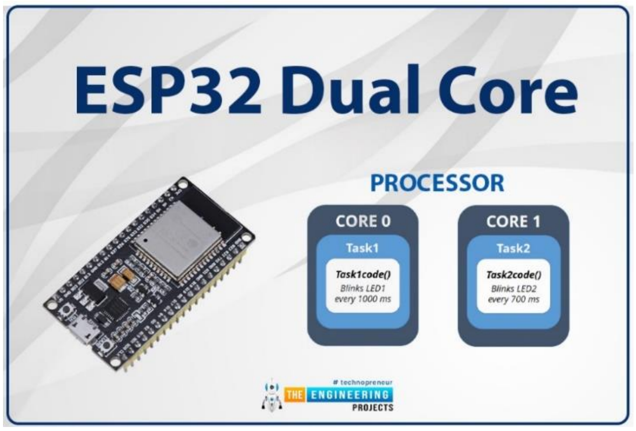
The Microcontroller


ESP32 DEV KIT V1 PINOUT
EN
Input only RTC GPIO0 SensVP ADC1_0 GPIO36 5
Input only RTC GPIO3 SensVN ADC1_3 GPIO39 8
Input only RTC GPIO4 ADC1_6 GPIO34 10
Input only RTC GPIO5 ADC1_7 GPIO35 11
RTC GPIO9 Xtal32P Touch9 ADC1_4 GPIO32 12
RTC GPIO8 Xtal32N Touch8 ADC1_5 GPIO33 13
DAC 1 RTC GPIO6 ADC2_8 GPIO25 14
DAC 2 RTC GPIO7 ADC2_9 GPIO26 15
RTC GPIO17 Touch7 ADC2_7 GPIO27 16
RTC GPIO16 Touch6 HSPI_CLK ADC2_6 GPIO14 17
RTC GPIO15 Touch5 HSPI_Q ADC2_5 GPIO12 18
RTC GPIO14 Touch4 HSPI_ID ADC2_4 GPIO13 20
GND
VIN
36 GPIO23 V_SPI_D MOSI
39 GPIO22 V_SPI_WP SCL RTS 0
41 GPIO1 TXD 0 CLK3
40 GPIO3 RXD 0 CLK2
42 GPIO21 VSPI_HD SDA
38 GPIO19 V_SPI_Q MISO CTS 0
35 GPIO18 V_SPI_CLK SCK
34 GPIO5 V_SPI_CS0 SS
27 GPIO17 TXD 2
25 GPIO16 RXD 2
24 GPIO4 ADC2_0 HSPI_HD Touch0 RTC GPIO10
22 GPIO2 ADC2_2 HSPI_WP0 Touch2 RTC GPIO12
21 GPIO15 ADC2_3 HSPI_CS0 Touch3 RTC GPIO13
GND
3.3v
www.mischianti.org
BY-NC-ND

Microcontroller	***ESP32***
Architecture	Xtensa LX6 dual-core processor
CPU Speed	Up to 240 MHz
Operating Voltage	2.2V to 3.6V
Digital I/O Pins	34 GPIO pins (including GPIO34 to GPIO39 as input-only pins)
Analog Input Pins	18 (ADC channels 0 to 17)
Analog Output Pins	2 (DAC1 and DAC2)
UART	3 (UART0, UART1, and UART2)
SPI	4 (SPI, HSPI, VSPI, and SPIRAM)
I2C	2 (I2C0 and I2C1)
PWM	16 (LED Control - LEDC)
Flash Memory	4MB (32Mb) or 8MB (64Mb)
RAM	520KB (in addition to the 4MB flash)
Wi-Fi	802.11 b/g/n/e/i (2.4GHz)
Bluetooth	Bluetooth v4.2 BR/EDR and BLE
Operating System	FreeRTOS

- The ESP32 microcontroller is chosen for its powerful processing capabilities, built-in Wi-Fi, and extensive GPIO pins.
- It serves as the brain of the system, coordinating components and executing control algorithms.
- The microcontroller communicates with sensors, receives input signals, and controls servo motors and DC motors.
- The choice is based on specific project requirements, including simultaneous self-balancing code, rover control, and Bluetooth communication.
- The ESP32 offers advantages over other microcontrollers like Arduino Uno in terms of performance and features.

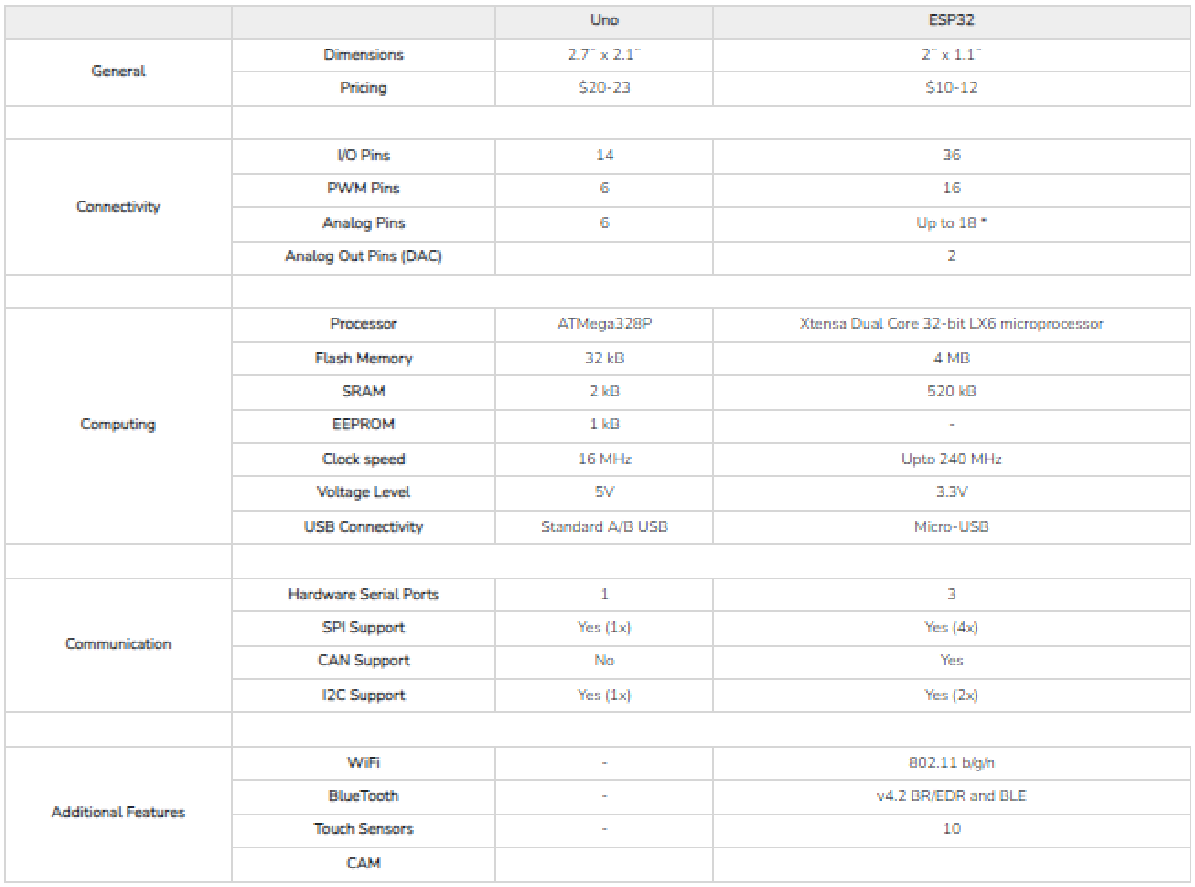
Dual-Core Architecture



- Dual-core architecture allows for concurrent execution of tasks, enabling simultaneous operation of the self-balancing and rover control code.
- Without Dual-core, 2 separate microcontroller boards are needed.
  a) For Self Balancing
  b) For Controlling the Rover DC motors

Other Advantages

- Built-in Bluetooth and Wi-Fi modules simplify wireless communication, enabling remote control and potential integration into wireless networks or cloud-based services. Without built in Bluetooth or Wi-Fi, a separate Bluetooth module will have to be used. This adds weight, wiring, space and other costs to the rover.
- Higher clock speed (up to 240 MHz or more) results in faster processing capabilities compared to the Arduino Uno.
- Larger flash memory (4MB to 16MB or more) and RAM (520KB to 8MB) allow for more complex algorithms, data storage, and future enhancements.
- Comprehensive peripheral support facilitates the integration of sensors, actuators, and communication modules required for the project.
- Overall, the ESP32 microcontroller offers improved performance, increased resources, and enhanced connectivity options for complex embedded systems projects like the stair climbing rover.

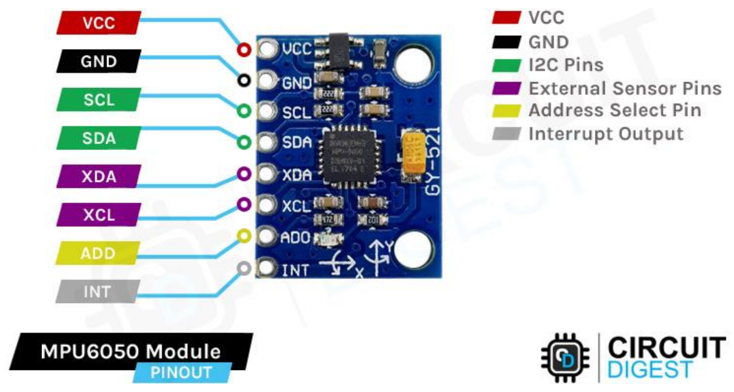


		Uno	ESP32
General	Dimensions	2.7" x 2.1"	2" x 1.1"
	Pricing	$20-23	$10-12
Connectivity	I/O Pins	14	36
	PWM Pins	6	16
	Analog Pins	6	Up to 18 *
	Analog Out Pins (DAC)		2
Computing	Processor	ATMega328P	Xtensa Dual Core 32-bit LX6 microprocessor
	Flash Memory	32 kB	4 MB
	SRAM	2 kB	520 kB
	EEPROM	1 kB	-
	Clock speed	16 MHz	Upto 240 MHz
	Voltage Level	5V	3.3V
	USB Connectivity	Standard A/B USB	Micro-USB
Communication	Hardware Serial Ports	1	3
	SPI Support	Yes (1x)	Yes (4x)
	CAN Support	No	Yes
	I2C Support	Yes (1x)	Yes (2x)
Additional Features	WiFi	-	802.11 b/g/n
	BlueTooth	-	v4.2 BR/EDR and BLE
	Touch Sensors	-	10
	CAM		

Sensors

The rover is equipped with a combination of sensors to facilitate its functionality. These include an inertial measurement unit (IMU) to measure the rover's orientation and acceleration, proximity sensors to detect stairs, and load sensors to monitor the weight on the platform.

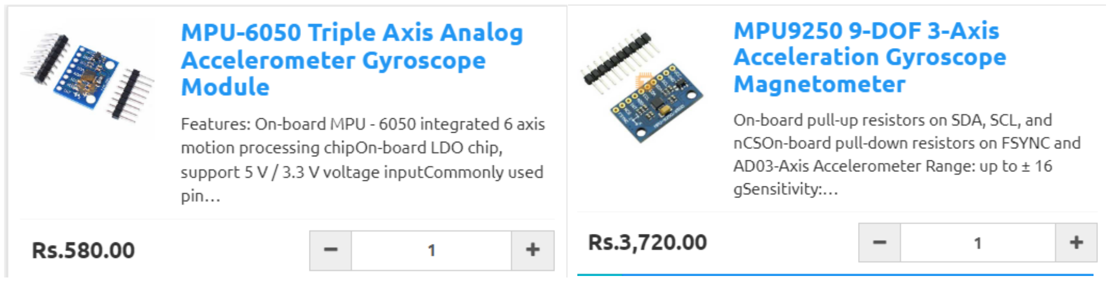
MPU6050 Gyroscope Sensor



- Inertial Measurement Unit (IMU) combining a 3-axis accelerometer and 3-axis gyroscope.
- Measures acceleration along X, Y, and Z axes (-16g to +16g) with the accelerometer.
- Measures angular velocity around X, Y, and Z axes with the gyroscope.
- Utilizes MEMS technology for accurate motion sensing.
- Provides raw digital values for acceleration and angular velocity.
- Communicates with microcontrollers using the I2C interface.
- Calibration is often required to compensate for biases and errors.
- Filtering techniques can be applied to improve data accuracy and stability.
- Widely used for self-balancing robots, drones, and motion tracking systems.

Limitations

The product used for the project is a cheap MPU6050 module, but more accuracy can be got from a more expensive module like MPU9250, which has 9 DOF.



MPU-6050 Triple Axis Analog Accelerometer Gyroscope Module

Features: On-board MPU - 6050 integrated 6 axis motion processing chipOn-board LDO chip, support 5 V / 3.3 V voltage inputCommonly used pin…

Rs.580.00 − 1 +

MPU9250 9-DOF 3-Axis Acceleration Gyroscope Magnetometer

On-board pull-up resistors on SDA, SCL, and nCSOn-board pull-down resistors on FSYNC and AD03-Axis Accelerometer Range: up to ± 16 gSensitivity:…

Rs.3,720.00 − 1 +

Bluetooth

- The Bluetooth module in the ESP32 board is based on the Bluetooth v4.2 specification.
- It supports multiple Bluetooth profiles, including Serial Port Profile (SPP), Generic Attribute Profile (GATT), and more.
- The module operates in the 2.4GHz ISM band and offers a range of communication options.
- It supports both Bluetooth Classic and Bluetooth Low Energy (BLE) modes.
- The Bluetooth stack in the ESP32 board is implemented using the ESP-IDF (Espressif IoT Development Framework) software framework.
- The ESP-IDF provides a set of APIs and libraries for Bluetooth functionality, including device discovery, connection management, and data transmission.

Actuators

Servo Motors

2 x TowerPro – MicroServo SG90

Advantages

Affordability: TowerPro Micro SG90 servo motors are affordable, making them accessible for hobbyists and projects with budget constraints.

Availability: These servo motors are widely available in the market, making them easy to purchase and replace if needed.

Compact Size: TowerPro Micro SG90 servo motors have a compact size, which makes them suitable for projects with limited space or size constraints.

Decent Performance: Despite their low cost, these servo motors offer satisfactory performance for basic applications and tasks.

Compatibility: TowerPro Micro SG90 servo motors are compatible with various microcontrollers and platforms, including Arduino, Raspberry Pi, and other popular development boards.

Wide Operating Voltage Range: They can operate within a broad voltage range, typically from 4.8V to 6V, allowing for flexibility in power supply options.

Disadvantages

Limited Torque: These servo motors have limited torque, which may not be sufficient for applications that require higher force or strength.

Inaccuracy: They may exhibit some degree of inaccuracy or non-linearity in their position control, which can affect precision in certain applications.

Durability: TowerPro Micro SG90 servo motors are not as robust or durable compared to higher-end servo motors. They may be more prone to wear and tear over time or under heavy use.

Noise and Vibrations: Due to their construction and design, these servo motors may generate more noise and vibrations compared to higher-quality servo motors.

Limited Features: TowerPro Micro SG90 servo motors lack advanced features such as position feedback or higher-resolution control, which may be required for more precise applications.

DC motors

DC motors with suitable gearing and traction mechanisms are employed for stair climbing.

- Voltage: 12V DC
- RPM: 120 (rotations per minute)
- Gearbox Type: Gearbox integrated with the motor.
- Motor Type: Brushed DC motor
- Torque: The torque specification will depend on the specific motor model and gear ratio. Higher gear ratios generally provide higher torque output.
- Current Draw: Maximum of 43A, depends on load and requirements.

Motor Driver – BTS7960 High Current Motor Driver

- The BTS7960 is a high current H-bridge motor driver module designed for motor drive applications.
- It features a fully integrated driver IC with logic level inputs, diagnosis with current sense, slew rate adjustment, and dead time generation.
- The module provides protection against overtemperature, overvoltage, undervoltage, overcurrent, and short circuit.
- It operates with an input voltage range of 6 to 27Vdc.
- The BTS7960 has a dual H-bridge configuration, allowing independent control of two motors or combined output for a single motor.
- It can handle a peak current of up to 43A.
- The module supports PWM control with a capability of up to 25 kHz and can also be controlled using level inputs.
- The working duty cycle ranges from 0% to 100%.
- It includes over-voltage lockout and under-voltage shutdown protection.
- The board size is approximately 50mm x 50mm x 43mm, and it weighs around 66g.
- The BTS7960 motor driver provides a cost-optimized solution for high current PWM motor drives while occupying minimal board space.

Others

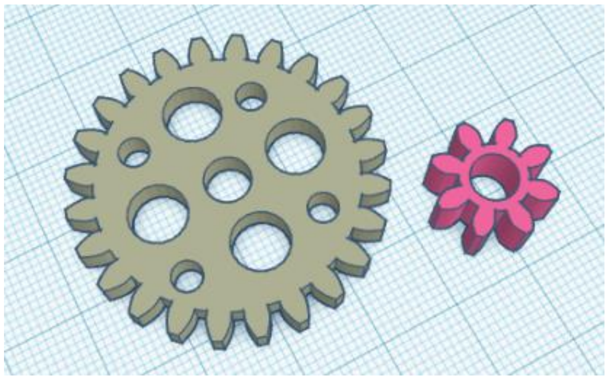
LM2596S Buck Converter

- LM2596S is a popular buck converter for voltage step-down applications.
- Input voltage range: 4.5V to 40V.
- Adjustable output voltage: 1.25V to 35V.
- Maximum output current: 3A.
- Efficient switching topology for high voltage conversion.
- Built-in protection features: overcurrent, thermal shutdown, input reverse polarity.
- Onboard potentiometer for easy voltage adjustment.
- High efficiency (85% to 92%).
- Compact size and low cost.

Spur Gears

To reduce weight and power consumptions, only one DC motor is used. Hence, to rotate two wheels, a gear system was used. The gears were designed in and printed using a 3D printer.

The axle was also printed, but it wasn't strong enough, so it broke. A wooden axle was used afterwards.



Power Requirements

The rover is powered by a rechargeable battery pack, providing the necessary voltage and current to drive the motors and microcontroller.

4 x 18650 3.7 V Rechargeable batteries were used.

DC Motor driver – Requires around 15V to work properly.

ESP32 Microcontroller – Requires 3.3V input but can take up to 9V.

Servo Motors – Require around 5V.

MPU6050 Gyroscope – Requires 5V.

Hence, to satisfy all these conditions, the following approach was taken.

1) Connect 4 batteries and create a power supply with around 15.5V.
2) This is directly supplied to the Motor Driver, and parallelly supplied to a buck converter.
3) The buck converter converts the 15.5V to 5V, and this is supplied parallel to the ESP32 board, Servo Motors, and the gyroscope sensor.

The total capacity of power supply is around 12,000 mAh.

Programming

Approach

At the beginning, 2 separate codes were used.

1. Arduino Uno code for Self-Balancing – Takes in readings from gyroscope sensor and controls 2 MicroSG servo motors to balance the platform.
2. Magicbit code for Rover Control – Takes in signals from a Bluetooth android app and controls the DC motors.

Later, since ESP32 board was used, both these codes were combined and run on both cores of ESP32 board.

1. Core 0 – Self Balancing Code (Named as Task 1 – Gyro)
2. Core 1 – Bluetooth rover control code. This code was included inside the main loop since ESP32 by default uses Core 1 for main loop code.

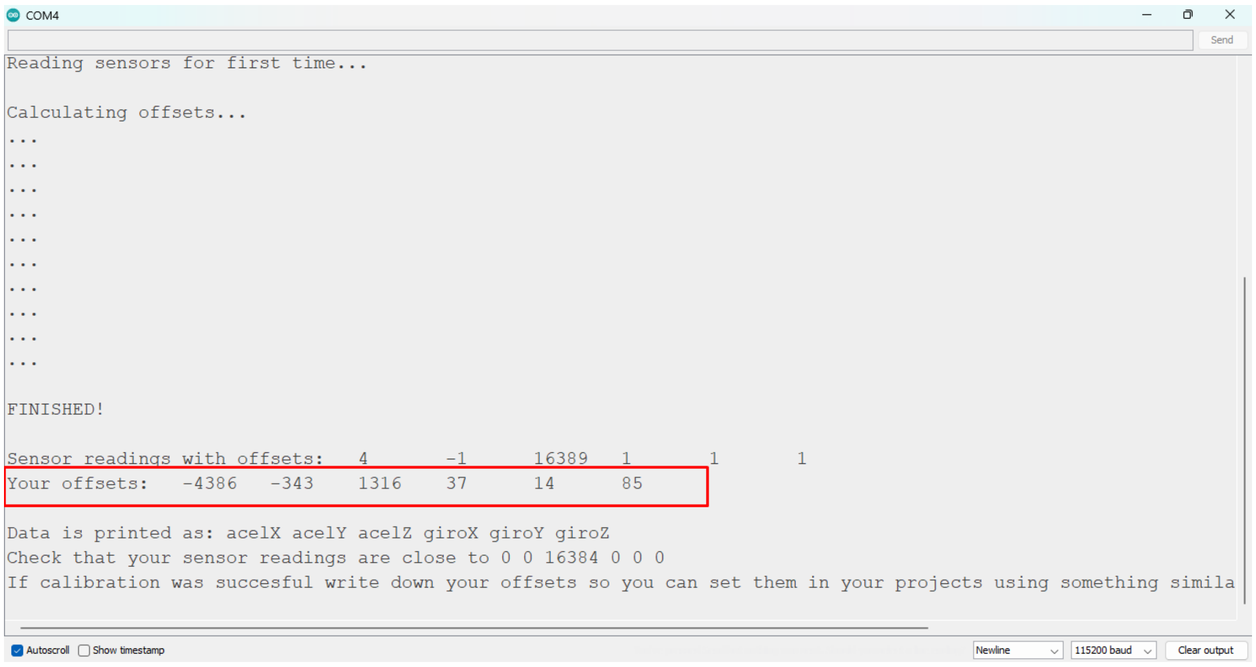
Challenges faced.

Calibrating MPU6050 Offset Errors

First, the balancing didn't work. Then after researching a lot, the issue was found to be offset errors.

All MPU6050 sensors have manufacturing offset errors, and we need to find them and correct them before using the device.

So, a code was run to calculate the 6 offset values (Gyroscope x y z, Accelerometer x y z). After that, the offset values were used in the main code to correct the readings, and then the balancing platform worked well.



```
Reading sensors for first time...

Calculating offsets...
...
...
...
...
...
...
...
...
...
...

FINISHED!

Sensor readings with offsets:   4       -1      16389   1       1       1
Your offsets:   -4386   -343    1316    37      14      85

Data is printed as: acelX acelY acelZ giroX giroY giroZ
Check that your sensor readings are close to 0 0 16384 0 0 0
If calibration was succesful write down your offsets so you can set them in your projects using something simila
```

Offset Calculating Code:

```
#include "I2Cdev.h"
#include "MPU6050.h"
#include "Wire.h"

MPU6050 accelgyro(0x68);

int16_t ax, ay, az,gx, gy, gz;

int mean_ax,mean_ay,mean_az,mean_gx,mean_gy,mean_gz,state=0;
int ax_offset,ay_offset,az_offset,gx_offset,gy_offset,gz_offset;


void setup() {
  // join I2C bus (I2Cdev library doesn't do this automatically)
  Wire.begin();

  // initialize serial communication
  Serial.begin(115200);

  // initialize device
  accelgyro.initialize();

  // wait for ready
  while (Serial.available() && Serial.read()); // empty buffer
  while (!Serial.available()){
    Serial.println(F("Send any character to start sketch.\n"));
    delay(1500);
  }
  while (Serial.available() && Serial.read()); // empty buffer again
  // start message
  Serial.println("\nMPU6050 Calibration Sketch");
  delay(2000);
  Serial.println("\nYour MPU6050 should be placed in horizontal position, with package letters facing up. \nDon't touch it until
you see a finish message.\n");
  delay(3000);
  // verify connection
  Serial.println(accelgyro.testConnection() ? "MPU6050 connection successful" : "MPU6050 connection failed");
  delay(1000);
  // reset offsets
  accelgyro.setXAccelOffset(0);
  accelgyro.setYAccelOffset(0);
  accelgyro.setZAccelOffset(0);
  accelgyro.setXGyroOffset(0);
  accelgyro.setYGyroOffset(0);
  accelgyro.setZGyroOffset(0);
}

void loop() {
  if (state==0){
    Serial.println("\nReading sensors for first time...");
    meansensors();
    state++;
    delay(1000);
  }

  if (state==1) {
    Serial.println("\nCalculating offsets...");
    calibration();
    state++;
    delay(1000);
  }

  if (state==2) {
    meansensors();
    Serial.println("\nFINISHED!");
    Serial.print("\nSensor readings with offsets:\t");
    Serial.print(mean_ax);
    Serial.print("\t");
    Serial.print(mean_ay);
    Serial.print("\t");
    Serial.print(mean_az);
    Serial.print("\t");
    Serial.print(mean_gx);
    Serial.print("\t");
    Serial.print(mean_gy);
    Serial.print("\t");
    Serial.println(mean_gz);
    Serial.print("Your offsets:\t");
    Serial.print(ax_offset);
    Serial.print("\t");
    Serial.print(ay_offset);
    Serial.print("\t");
    Serial.print(az_offset);
    Serial.print("\t");
    Serial.print(gx_offset);
    Serial.print("\t");
    Serial.print(gy_offset);
    Serial.print("\t");
    Serial.println(gz_offset);
    Serial.println("\nData is printed as: acelX acelY acelZ giroX giroY giroZ");
    Serial.println("Check that your sensor readings are close to 0 0 16384 0 0 0");
    Serial.println("If calibration was succesful write down your offsets so you can set them in your projects using something similar
to mpu.setXAccelOffset(youroffset)");
    while (1);
  }
}

void meansensors(){
  long i=0,buff_ax=0,buff_ay=0,buff_az=0,buff_gx=0,buff_gy=0,buff_gz=0;

  while (i<(buffersize+101)){
    // read raw accel/gyro measurements from device
    accelgyro.getMotion6(&ax, &ay, &az, &gx, &gy, &gz);

    if (i>100 && i<=(buffersize+100)){ //First 100 measures are discarded
      buff_ax=buff_ax+ax;
      buff_ay=buff_ay+ay;
      buff_az=buff_az+az;
      buff_gx=buff_gx+gx;
      buff_gy=buff_gy+gy;
      buff_gz=buff_gz+gz;
    }
    if (i==(buffersize+100)){
      mean_ax=buff_ax/buffersize;
      mean_ay=buff_ay/buffersize;
      mean_az=buff_az/buffersize;
      mean_gx=buff_gx/buffersize;
      mean_gy=buff_gy/buffersize;
      mean_gz=buff_gz/buffersize;
    }
    i++;
    delay(2); //Needed so we don't get repeated measures
  }
}

void calibration(){
  ax_offset=-mean_ax/8;
  ay_offset=-mean_ay/8;
  az_offset=(16384-mean_az)/8;

  gx_offset=-mean_gx/4;
  gy_offset=-mean_gy/4;
  gz_offset=-mean_gz/4;
  while (1){
    int ready=0;
    accelgyro.setXAccelOffset(ax_offset);
    accelgyro.setYAccelOffset(ay_offset);
    accelgyro.setZAccelOffset(az_offset);

    accelgyro.setXGyroOffset(gx_offset);
    accelgyro.setYGyroOffset(gy_offset);
    accelgyro.setZGyroOffset(gz_offset);

    meansensors();
    Serial.println("...");

    if (abs(mean_ax)<=acel_deadzone) ready++;
    else ax_offset=ax_offset-mean_ax/acel_deadzone;

    if (abs(mean_ay)<=acel_deadzone) ready++;
    else ay_offset=ay_offset-mean_ay/acel_deadzone;

    if (abs(16384-mean_az)<=acel_deadzone) ready++;
    else az_offset=az_offset+(16384-mean_az)/acel_deadzone;

    if (abs(mean_gx)<=giro_deadzone) ready++;
    else gx_offset=gx_offset-mean_gx/(giro_deadzone+1);

    if (abs(mean_gy)<=giro_deadzone) ready++;
    else gy_offset=gy_offset-mean_gy/(giro_deadzone+1);

    if (abs(mean_gz)<=giro_deadzone) ready++;
    else gz_offset=gz_offset-mean_gz/(giro_deadzone+1);

    if (ready==6) break;
  }
}
```

This printed the offset values in the serial monitor, and these values were later used to correct the readings in the main code.

Writing Code for ESP32 board

ESP32 board is not the same as other common boards like Arduino Uno, hence when coding certain functions don't work.

Ex: analogWrite() function is not available in ESP32. So, it was challenging to control the motors.
Instead of analogWrite(), ESP32 board uses a different functions like ledcSetup(), ledcAttachPin(), ledcWrite().

Reading errors from MPU6050 with time

Initially, MPU6050 values got more and more wrong with time. This happened due to some drift errors.

After researching, it was found that to avoid this issue, the inbuilt Digital Motion Processor (DMP) in the MPU6050 sensor can be used. When this is used, an interrupt using the INT pin is sent to the MicroController, and this prevents the drift errors.

Without using the DMP, the MPU6050 sensor readings can suffer from drift and provide incorrect values.

Drift can occur in the gyroscope readings over time due to various factors.

The accelerometer is sensitive to tilt, leading to inaccurate orientation estimation if not compensated properly.

Sensor noise can introduce errors and affect the accuracy of the measurements.

Sensor calibration is crucial to ensure accurate readings.

Implementing sensor fusion algorithms, calibration techniques, and filtering on the main microcontroller can improve accuracy without relying on the DMP.

The DMP in the MPU6050 performs sensor fusion, calibration, and drift compensation within the sensor itself, providing more accurate motion-related information.

Hence, the code was updated to include the DMP and after that, the sensor accuracy improved a lot.

Problems with making use of Dual Core architecture.

Had to make use of FreeRTOS functions to create Dual Core tasks. FreeRTOS is the firmware for ESP32. Since there is less documentation and resources for this compared to Arduino boards, it was a bit challenging to get it to work.

ESP32 board has 2 cores.

1. Core 0 – Not used usually.
2. Core 1 – The default core used for programs.

Initially, 2 tasks were created and pinned for each core.

- Core 0 – RC Control (with an infinite for loop)
- Core 1 – Self Balancing (with an infinite for loop)
- Main loop – Not used.

But there were some performance issues with this code, so a different approach was taken as follows.

- Core 0 – RC Control (with an infinite for loop)
- Main Loop – Self Balancing
- Core 1 – No pinned task, but Core 1 is used for Main loop by default.

Even after this, there was a slight delay when balancing the platform. After several attempts, the issue was guessed to be that both cores' are running at the same time, so there is a small delay when one is overlapping with other.

To avoid this, a small delay was introduced to the Self Balancing Core0 task using vTaskDelay() function from FreeRTOS.

After this, the dual core process worked properly.

The Final Code

```
TaskHandle_t Task1;
//TaskHandle_t Task2;

#include "BluetoothSerial.h" //Header File for Serial Bluetooth
BluetoothSerial ESP_BT; //Object for Bluetooth
#include <Wire.h>

#include "I2Cdev.h"
#include <ESP32Servo.h>
#include "MPU6050_6Axis_MotionApps20.h"

MPU6050 mpu;

#define OUTPUT_READABLE_YAWPITCHROLL
#define INTERRUPT_PIN 2  // use pin 2 on Arduino Uno & most boards
bool blinkState = false;

// MPU control/status vars
bool dmpReady = false;  // set true if DMP init was successful
uint8_t mpuIntStatus;   // holds actual interrupt status byte from MPU
uint8_t devStatus;      // return status after each device operation (0 = success, !0 = error)
uint16_t packetSize;    // expected DMP packet size (default is 42 bytes)
uint16_t fifoCount;     // count of all bytes currently in FIFO
uint8_t fifoBuffer[64]; // FIFO storage buffer

// orientation/motion vars
Quaternion q;           // [w, x, y, z]         quaternion container
VectorInt16 aa;         // [x, y, z]            accel sensor measurements
VectorInt16 aaReal;     // [x, y, z]            gravity-free accel sensor measurements
VectorInt16 aaWorld;    // [x, y, z]            world-frame accel sensor measurements
VectorFloat gravity;    // [x, y, z]            gravity vector
float euler[3];         // [psi, theta, phi]    Euler angle container
float ypr[3];           // [yaw, pitch, roll]   yaw/pitch/roll container and gravity vector


volatile bool mpuInterrupt = false;     // indicates whether MPU interrupt pin has gone high
void IRAM_ATTR dmpDataReady() {
    mpuInterrupt = true;
}

int  servoPin2  =  25 , servoPin3  =  26 ;
Servo servo2, servo3;

String incoming;// varible store received data

const int BST_LPwm = 27;    // I/O channel setup ESP32 pin
const int BST_RPwm = 12;    // I/O chennel setup ESP32 pin
const int En_LR    = 14;    // I/O pin for BST l_EN & R_EN (enable)
int Speed = 255 ;

const int  Channel_15 = 15;   // & PWM channel 0, for BST pin L_PWM
const int  Channel_14 = 14;   // & PWM channel 1, for BST pin R_PWM
const int freq = 1000;     //  Set up PWM Frequency
const int  res = 8;        //  Set up PWM Resolution

void setup() {

  ledcSetup(Channel_15, freq,res); // setup PWM channel for BST L_PWM
  ledcSetup(Channel_14, freq,res); // setup PWM channel for BST R_PWM
  ledcAttachPin( BST_LPwm , Channel_15); // Attach BST L_PWM
  ledcAttachPin( BST_RPwm , Channel_14); // Attach BST R_PWM
  pinMode(En_LR ,OUTPUT); // declare pin as output

  servo2.attach (servoPin2);
  servo3.attach (servoPin3);

  servo2.write ( 0 ); // Init the servo2 angle to 0
  servo3.write ( 0 ); // Init the servo2 angle to 0

  Serial.begin(115200);
  Wire.begin(21, 22, 100000); // sda, scl, clock speed
  Wire.beginTransmission(0x68);
  Wire.write(0x6B);  // PWR_MGMT_1 register
  Wire.write(0);     // set to zero (wakes up the MPU−6050)
  Wire.endTransmission(true);

  // initialize device
  mpu.initialize();
  pinMode(INTERRUPT_PIN, INPUT);

  // verify connection
  Serial.println(mpu.testConnection() ? F("MPU6050 connection successful") : F("MPU6050 connection failed"));

  // load and configure the DMP
  Serial.println(F("Initializing DMP..."));
  devStatus = mpu.dmpInitialize();

    // supply your own gyro offsets here, scaled for min sensitivity
    mpu.setXGyroOffset(37);
    mpu.setYGyroOffset(14);
    mpu.setZGyroOffset(85);

    mpu.setXAccelOffset(-4386); // 1688 factory default for my test chip
    mpu.setYAccelOffset(-343);
    mpu.setZAccelOffset(1316);

  // make sure it worked (returns 0 if so)
  if (devStatus == 0) {
       // turn on the DMP, now that it's ready
       Serial.println(F("Enabling DMP..."));
       mpu.setDMPEnabled(true);

       // enable Arduino interrupt detection
       Serial.println(F("Enabling interrupt detection (Arduino external interrupt 0)..."));
       attachInterrupt(digitalPinToInterrupt(INTERRUPT_PIN), dmpDataReady, RISING);
       mpuIntStatus = mpu.getIntStatus();
```

```
      // set our DMP Ready flag so the main loop() function knows it's okay to use it
      Serial.println(F("DMP ready! Waiting for first interrupt..."));
      dmpReady = true;

      // get expected DMP packet size for later comparison
      packetSize = mpu.dmpGetFIFOPacketSize();
  } else {
      // ERROR!
      // 1 = initial memory load failed
      // 2 = DMP configuration updates failed
      // (if it's going to break, usually the code will be 1)
      Serial.print(F("DMP Initialization failed (code "));
      Serial.print(devStatus);
      Serial.println(F(")"));
  }


  //create a task that will be executed in the Task1code() function, with priority 1 and executed on core 0
  xTaskCreatePinnedToCore(
                    Task1code,   /* Task function. */
                    "RC_Car",     /* name of task. */
                    10000,       /* Stack size of task */
                    NULL,        /* parameter of the task */
                    1,           /* priority of the task */
                    &Task1,      /* Task handle to keep track of created task */
                    0);          /* pin task to core 0 */

}

//Task1code: Bluetooth RC Car
void Task1code( void * pvParameters ){
  ESP_BT.begin("Stair Climber"); //Name of your Bluetooth Signal
  Serial.print("Task1 running on core ");
  Serial.println(xPortGetCoreID());
  Serial.println("Bluetooth Device is Ready to Pair");

  for(;;){
    digitalWrite(En_LR,HIGH); // enable BST

      if (ESP_BT.available()) //Check if we receive anything from Bluetooth
      {
      incoming = char(ESP_BT.read()); //Read what we received
      }
      Serial.println(incoming);
      if (incoming == "F") { // forward
       ledcWrite(Channel_15,0);       // stop channel 14  (R_PWM)
       ledcWrite(Channel_14,Speed);   // run channel 15 (L_PWM) at "speed"
      }
      else if (incoming == "B") { // reverse
       ledcWrite(Channel_14,0);       // stop channel 15 (L_PWM)
       ledcWrite(Channel_15,Speed);   // run channel 14 (R_PWM) at "speed"
      }
//      else if (incoming == "R") { //turn right
//      Serial.println("Right");
//      }
//      else if (incoming == "L") { // turn left
//      Serial.println("Left");
//      }
//
      else if (incoming == "S") { // turn stop
       ledcWrite(Channel_14,0);      // stop channel 14  (R_PWM)
       ledcWrite(Channel_15,0);      // stop chennel 15  (L_PWM)
      }

      else {
       ledcWrite(Channel_14,0);      // stop channel 14  (R_PWM)
       ledcWrite(Channel_15,0);      // stop chennel 15  (L_PWM)
      }
      vTaskDelay(2); //Very Very Important!
  }
}

void loop() {
   // if programming failed, don't try to do anything
   if (!dmpReady) return;

   // reset interrupt flag and get INT_STATUS byte
   mpuInterrupt = false;
   mpuIntStatus = mpu.getIntStatus();

   // get current FIFO count
   fifoCount = mpu.getFIFOCount();

   // check for overflow (this should never happen unless our code is too inefficient)
   if ((mpuIntStatus & 0x10) || fifoCount == 1024) {
      // reset so we can continue cleanly
      mpu.resetFIFO();
      Serial.println(F("FIFO overflow!"));

   // otherwise, check for DMP data ready interrupt (this should happen frequently)
   } else if (mpuIntStatus & 0x02) {
      // wait for correct available data length, should be a VERY short wait
      while (fifoCount < packetSize) fifoCount = mpu.getFIFOCount();

      // read a packet from FIFO
      mpu.getFIFOBytes(fifoBuffer, packetSize);

      // track FIFO count here in case there is > 1 packet available
      // (this lets us immediately read more without waiting for an interrupt)
      fifoCount -= packetSize;

      #ifdef OUTPUT_READABLE_YAWPITCHROLL
         // display Euler angles in degrees
         mpu.dmpGetQuaternion(&q, fifoBuffer);
         mpu.dmpGetGravity(&gravity, &q);
         mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
         Serial.print("ypr\t");
```

```
        Serial.print(ypr[0] * 180/M_PI);
        Serial.print("\t");
        Serial.print(ypr[1] * 180/M_PI);
        Serial.print("\t");
        Serial.println(ypr[2] * 180/M_PI);
    #endif

    #ifdef  OUTPUT_READABLE_YAWPITCHROLL

        // display Euler angles in degrees

        mpu.dmpGetQuaternion (& q, fifoBuffer);
        mpu.dmpGetGravity (& gravity, & q);
        mpu.dmpGetYawPitchRoll (ypr, & q, & gravity);
        Serial.print ( "ypr \ t" );
        Serial.print (ypr [ 0 ]  *  180 / M_PI);
        Serial.print ( "\ t" );
        Serial.print (ypr [ 1 ]  *  180 / M_PI);
        servo2.write (map (ypr [ 1 ]  *  180 / M_PI,  - 90 ,  90 ,  0 ,  180 )); // Control servo2
        Serial.print ( "\ t" );
        Serial.println (ypr [ 2 ]  *  180 / M_PI);
        servo3.write (map (ypr [ 2 ]  *  180 / M_PI,  90 ,  -90 ,  0 ,  180 )); // Control servo3
    #endif
}
}
```

Testing

Versions

V1

Arduino Uno for Self-Balancing. Magicbit for RC control. RC control working, but self-balancing not working.

V2

Arduino Uno for Self-Balancing. Magicbit for RC control. DMP was used to avoid drift errors. Self-Balancing working better, but still not fully correct

V3

Arduino Uno for Self Balancing. Magicbit for RC control. Offset errors of MPU6050 were calculated and fixed. Both RC control and Self-Balancing worked properly.

V4

ESP32 for Self-Balancing. ESP32 for RC control. Dual core not used. Both RC control and Self-Balancing worked properly.

V5

ESP32 Dual core used, for both RC Control and Self-Balancing. RC Control worked properly; Self-Balancing had a delay.

V6
ESP32 Dual Core used, for both RC Control and Self-Balancing. After adding a delay for Self-Balancing Core Task, both RC Control and Self-Balancing working properly.

V7
2 DC motors used, but issue with weight and power.

V8
1 DC motor with gear and axle used. But rover overbalances due to one side being too heavy.

V9
Counterbalance tail added. Final version, all functions work.

Production Cost

Prototype Cost
The cost breakdown for building the prototype is as follows:

ESP32 Microcontroller:	Rs. 2,200
Micro SG Servo Motors (x2):	Rs. 1,000
DC Motors:	Rs. 2,500
MPU6050 Sensor:	Rs. 650
BTS7960 Motor Driver:	Rs. 1,600
Batteries:	Rs. 3,500

Chassis, Wheels, and Mechanical Components: Rs. 3,500

Wiring, Connectors, and Miscellaneous: Rs. 200

The total cost for building the prototype is estimated to be Rs. 15,150.00/=

Conclusion

Future Work
While the current project prototype successfully demonstrates the stair climbing rover's capabilities, there are several areas for future improvement:

- Ability to turn left and right.
- Be able to carry bigger weights.
- Be able to climb bigger stairs.

- Improve stability on uneven terrains and different load conditions.
- Handle a wider range of stair designs and orientations.
- Use of advanced sensors and machine learning techniques for enhanced obstacle detection and avoidance.

In conclusion, the stair climbing rover with a self-balancing platform presents an innovative solution to the problem of carrying items upstairs. The project successfully demonstrates the integration of embedded systems, motor control, and algorithms to achieve stable and efficient stair climbing functionality. With further development and enhancements, this project has the potential to revolutionize the way items are transported on stairs, offering convenience, safety, and efficiency in various applications.

Project Resources

During the course of the project, the following resources were found valuable:

ESP32 Datasheet - https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf, https://roboeq.ir/files/id/4034/name/ESP32%20MODULE.pdf/

BTS7960 Motor Driver Datasheet - https://www.handsontec.com/dataspecs/module/BTS7960%20Motor%20Driver.pdf

MPU6050 Gyroscope Datasheet - https://invensense.tdk.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

ESP32 Pinout - https://www.mischianti.org/wp-content/uploads/2020/11/ESP32-DOIT-DEV-KIT-v1-pinout-mischianti.png

FreeRTOS resources for coding dual core - https://www.freertos.org/a00127.html

Youtube Video on analogWrite for ESP32 - https://www.youtube.com/watch?v=C_zL2orpj_g

3D designing – https://app.sketchup.com/app?hl=en, https://www.tinkercad.com/, https://www.stlgears.com/generators/3dprint

Stair Climbing Wheel Designs Research - https://journals.sagepub.com/doi/10.1177/1729881417749470