Cre-Aid Labs Pvt Ltd

Assignment

Problem Statement: Write code for the control system for a motor controller that uses the ESP32 microcontroller. The

system must have the following features:

a. The motor speed is the input to the controller.

b. The motor speed shouldn’t change, irrespective of the load applied.

c. The motor uses an H-Bridge driver which, in turn, are driven using PWM by the ESP32.

d. The motor has an incremental hall-effect pulse encoder (See OE-37 Encoder) which is

interfaced with ESP32 for feedback.

e. The controller may communicate to the host device using serial port, where the user can set

the speed and direction in the following format:

i.F<speed> for forward, speed is a value from 0 to 255. E.g. F40

ii.B<speed> for backward, speed is a value from 0 to 255. E.g. E200

iii.S for stop

f. Any kind of mapping from the speed values to angular speed of the motor may be used.

GitHub Link: https://github.com/Vaishd30/Cre-Aid/tree/main

Code

```
// Motor control pins
const int motorPwmPin = 5;              // PWM pin for motor speed control
const int motorDir1Pin = 18;            // H-Bridge input 1
const int motorDir2Pin = 19;            // H-Bridge input 2
const int encoderChannelAPin = 2;           // Encoder channel A pin
```

```
const int encoderChannelBPin = 3;           // Encoder channel B pin

// PID controller variables
float pidProportionalGain = 1.0;            // PID gains
float pidIntegralGain = 0.1;
float pidDerivativeGain = 0.05;
float integralTerm = 0;
float derivativeTerm = 0;
float previousError = 0;
unsigned long previousTime = 0;

// Motor speed variables
int targetSpeed = 0;
volatile int encoderPulseCount = 0;

void setupMotor();
void processSerialCommand();
void runPIDControl();
int parseSpeed();
void setMotorDirection(bool forward);
void setMotorSpeed(int speed);
void stopMotor();
void handleEncoderInterrupt();
```

```
int calculateEncoderSpeed();

void setup() {
  setupMotor();
}

void loop() {
  processSerialCommand();
  runPIDControl();
}

void setupMotor() {
  pinMode(motorPwmPin, OUTPUT);
  pinMode(motorDir1Pin, OUTPUT);
  pinMode(motorDir2Pin, OUTPUT);
  pinMode(encoderChannelAPin, INPUT_PULLUP);
  pinMode(encoderChannelBPin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(encoderChannelAPin),
handleEncoderInterrupt, RISING);
  attachInterrupt(digitalPinToInterrupt(encoderChannelBPin),
handleEncoderInterrupt, RISING);
  Serial.begin(9600);
  stopMotor();
```

```
}


void processSerialCommand() {
  if (Serial.available() > 0) {
    char command = Serial.read();
    switch (command) {
      case 'F':
        setMotorDirection(true);
        targetSpeed = parseSpeed();
        break;
      case 'B':
        setMotorDirection(false);
        targetSpeed = parseSpeed();
        break;
      case 'S':
        stopMotor();
        break;
      default:
        Serial.println("Invalid command");
    }
  }
}
```

```
void runPIDControl() {
  unsigned long currentTime = millis();
  float deltaTime = (currentTime - previousTime) / 1000.0;
  int currentSpeed = calculateEncoderSpeed();
  float error = targetSpeed - currentSpeed;
  integralTerm += error * deltaTime;
  derivativeTerm = (error - previousError) / deltaTime;
  previousError = error;
  previousTime = currentTime;

  // PID output mapping and constraint
  int pwmValue = targetSpeed + pidProportionalGain * error + pidIntegralGain *
integralTerm + pidDerivativeGain * derivativeTerm;
  pwmValue = constrain(pwmValue, 0, 255);
  setMotorSpeed(pwmValue);
}

int parseSpeed() {
  String speedString = Serial.readStringUntil('\n');
  return speedString.toInt();
}

void setMotorDirection(bool forward) {
```

```
  digitalWrite(motorDir1Pin, forward ? HIGH : LOW);
  digitalWrite(motorDir2Pin, forward ? LOW : HIGH);
}

void setMotorSpeed(int speed) {
  analogWrite(motorPwmPin, speed);
}

void stopMotor() {
  setMotorSpeed(0);
}

void handleEncoderInterrupt() {
  if (digitalRead(encoderChannelBPin) == HIGH) {
    encoderPulseCount++;
  }
 Else {
    encoderPulseCount--;
  }
}

int calculateEncoderSpeed() {
  int currentCount = encoderPulseCount;
```

```
  int countsPerSecond = static_cast<int>(currentCount / (millis() - previousTime) *
1000.0);
  return countsPerSecond;
}
```

Assumption:

<u>Hardware configuration :</u>

1. Microcontroller:
    - ESP32 microcontroller board
2. Motor:
    - DC motor with known speed-torque and PWM-to-speed characteristics (consult datasheet)
3. Encoder:
    - Incremental hall-effect pulse encoder (e.g., OE-37 Encoder)
    - Provides two channels (A and B) for quadrature decoding
    - Resolution (pulses per revolution) known from datasheet
4. H-Bridge Driver:
    - Capable of driving the specific motor at its rated voltage and current
    - Controlled by PWM signals from the ESP32
5. Pin Connections:
    ESP32 Pins:
    - **motorPwmPin**: Connected to the H-Bridge driver's PWM input pin
    - **motorDir1Pin, motorDir2Pin**: Connected to H-Bridge driver's direction control pins
    - **encoderChannelAPin, encoderChannelBPin**: Connected to encoder's A and B channels
6. Power Supply:
  - Motor powered by a separate power supply suitable for its voltage and current requirements

- ESP32 powered by a suitable power source (USB, battery, etc.)

Calculations:

Encoder Speed Calculation:

1. Pulses per revolution: Obtained from the encoder's datasheet (e.g., 500 pulses per revolution)
2. Time interval for pulse count: Measured in milliseconds using **millis()** function
3. Counts per second**: counts_per_second = (current_count / (millis() - previous_time)) * 1000.0**
4. RPM: **RPM = (counts_per_second * 60) / pulses_per_revolution**

PID Control:

1. Error: **error = target_speed - current_speed**
2. Integral term: **integral_term = integral_term + error * delta_time**
3. Derivative term: **derivative_term = (error - previous_error) / delta_time**
4. PWM adjustment: **pwm_value = target_speed + Kp * error + Ki * integral_term + Kd * derivative_term**

PWM Mapping:

1. ESP32 PWM range: 0-255
2. Constrain PWM value: **pwm_value = constrain(pwm_value, 0, 255)**

Block Diagram: