Diseño e Implementación de Sistemas Embebidos, Laboratorio 1

Nombre: Joan Esteban Velasco Larrea

Primer reto: Control de Secuencia de LEDs con Botón Único

Análisis de requerimientos:

Objetivo:

Diseñar e implementar un sistema embebido que controle una secuencia de LEDs utilizando un único botón pulsador, permitiendo iniciar, pausar y reanudar la secuencia de forma cíclica. El sistema debe ser simple, determinista y funcionar de manera confiable tanto en simulación como en implementación física.

Requerimientos funcionales:

El sistema debe cumplir los siguientes requisitos funcionales:

1. Control de LEDs:
   - El sistema debe controlar cuatro LEDs conectados a salidas digitales
   - Solo un LED debe estar encendido a la vez
   - Los LEDs deben encenderse secuencialmente.
   - El cambio de LED debe ocurrir cada 500 ms.
2. Control mediante botón:
   - El sistema debe usar un único botón pulsador como entrada.
   - Antes de pulsar por primera vez el botón, todos los leds se encuentran apagados.
   - Al primer pulso del botón, la secuencia de LEDs debe iniciar desde el primer LED.
   - Al segundo pulso, la secuencia debe detenerse y mantener el LED actual encendido.
   - Al tercer pulso, la secuencia debe reanudarse desde el LED donde se detuvo
   - Cada pulso adicional debe alternar entre los estados running y stopped.
3. Comportamiento cíclico
   - La secuencia de LEDs debe reiniciarse automáticamente después del cuarto LED.
   - El sistema debe funcionar indefinidamente mientras esté alimentado.

Requerimientos no funcionales:

1. Confiabilidad
    - El sistema no debe presentar comportamientos erráticos ante pulsaciones normales del botón.
    - El estado del sistema debe mantenerse consistente.
2. Simplicidad
    - El diseño debe ser claro y fácil de entender.

Restricciones ambientales y de usuario:

1. Entorno
    - Alimentación eléctrica estable.
    - No se consideran condiciones extremas de temperatura o humedad.
2. Usuario
    - Usuario interactúa únicamente mediante un botón.
    - No se requiere conocimiento técnico para operar el sistema.

Análisis y diseño del sistema (Arquitectura y componentes):

Objetivo:

Analizar y diseñar la arquitectura lógica y física de un sistema embebido que controla una secuencia de 4 LEDs usando un solo botón, ejecutando la secuencia de forma cíclica con estados RUN y PAUSE.

- Microncontrolador ESP32
- 1 x botón
- 4 x LEDs (Cualquier color)
- Protoboard y Cables
- Entorno de desarrollo Arduino IDE.

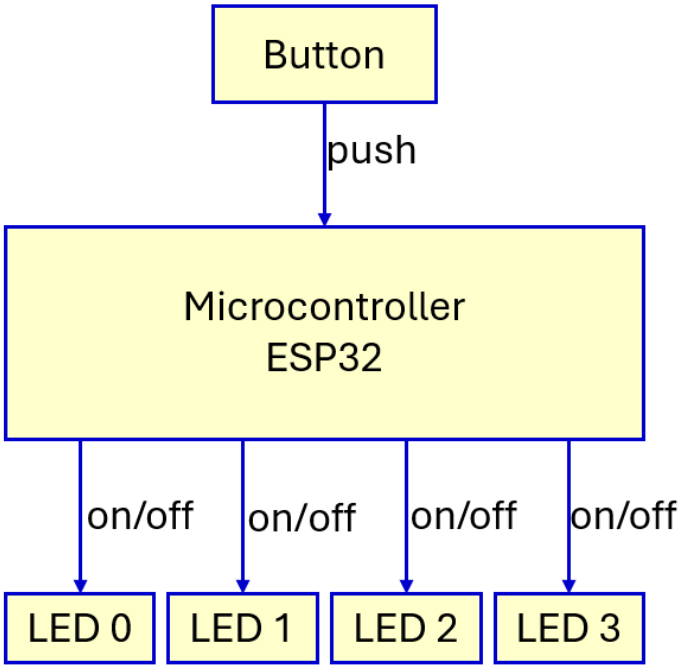


Figura 1. Esquema de funcionamiento.

Diseño e Integración de Hardware:

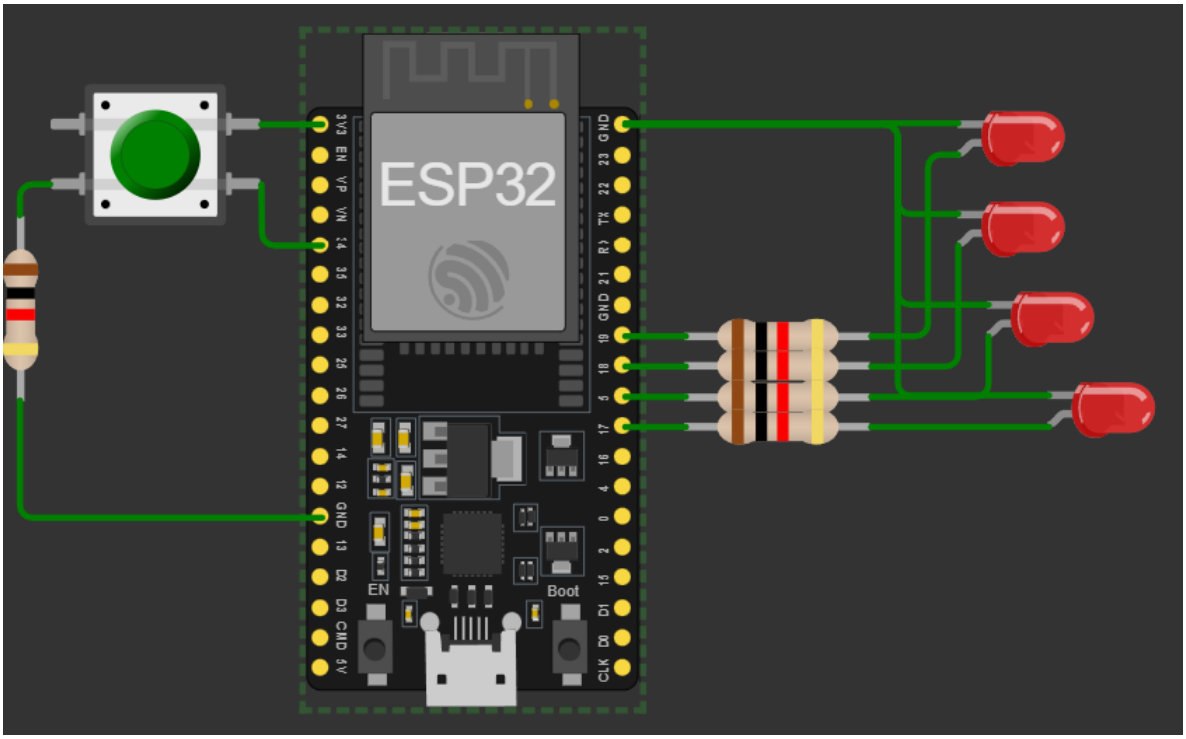


Figura 2. Arquitectura de hardware utilizada.

Desarrollo del Firmware:

El firmware desarrollado es el siguiente:

```
#define LED0 19
#define LED1 18
#define LED2 5
#define LED3 17
#define BUTTON 34
#define BOUNCE_WAIT 50
#define LED_WAIT 500


enum state {
  INIT,
  FIRST_PRSS_WAIT,
  RUN,
  SECOND_PRSS_WAIT,
  PAUSE,
};

bool led0,led1,led2,led3 = 0;

state current_state = INIT;
unsigned long last_time = millis();
```

Figura 3a. Diseño firmware, definir constantes, estados y variables globales.

```
void setup() {
  Serial.begin(9600);
  pinMode(LED0, OUTPUT);
  pinMode(LED1, OUTPUT);
  pinMode(LED2, OUTPUT);
  pinMode(LED3, OUTPUT);

  pinMode(BUTTON, INPUT);

  digitalWrite(LED0, led0);
  digitalWrite(LED1, led0);
  digitalWrite(LED2, led0);
  digitalWrite(LED3, led0);
}
```

Figura 3b. Diseño firmware, configuración de pines y estado inicial.

```
38  void loop() {
39    bool aux = led0;
40    switch(current_state)
41    {
42      case INIT:
43        led0 = true;
44        if (digitalRead(BUTTON)) current_state = FIRST_PRSS_WAIT;
45        Serial.println("INIT");
46
47        break;
48      case FIRST_PRSS_WAIT:
49        current_state = RUN;
50        delay(BOUNCE_WAIT);
51        Serial.println("FIRST_PRSS_WAIT");
52        while(digitalRead(BUTTON));
53        delay(BOUNCE_WAIT);
54        break;
55      case RUN:
56        digitalWrite(LED0, led0);
57        digitalWrite(LED1, led1);
58        digitalWrite(LED2, led2);
59        digitalWrite(LED3, led3);
```

Figura 3c. Diseño firmware, ciclo de ejecución primera parte.

```
61        if(millis() - last_time > LED_WAIT)
62        {
63          led0 = led1;
64          led1 = led2;
65          led2 = led3;
66          led3 = aux;
67          last_time = millis();
68        }
69
70        if (digitalRead(BUTTON)) current_state = SECOND_PRSS_WAIT;
71        Serial.println("RUN");
72        break;
73      case SECOND_PRSS_WAIT:
74        current_state = PAUSE;
75        delay(BOUNCE_WAIT);
76        Serial.println("SECOND_PRSS_WAIT");
77        while(digitalRead(BUTTON));
78        delay(BOUNCE_WAIT);
79        break;
80      case PAUSE:
81        if (digitalRead(BUTTON)) current_state = FIRST_PRSS_WAIT;
82        Serial.println("PAUSE");
83        break;
84    }
85  }
```

Figura 3d. Diseño de firmware, ciclo de ejecución segunda parte.

Documentación:

- Inputs: botón (GPIO 34)

- Outputs: LED0, LED1, LED2 y LED3 (GPIO 19, 18, 5 y 17).
- Estados:
    - Init: Todos los leds apagados esperando al primr pulso
    - Run: Secuencia de LEDs active
    - PAUSE: Secuencia detenida
    - FIRST_PRSS_WAIT y SECOND_PRSS_WAIT: debounce y transición de estados.

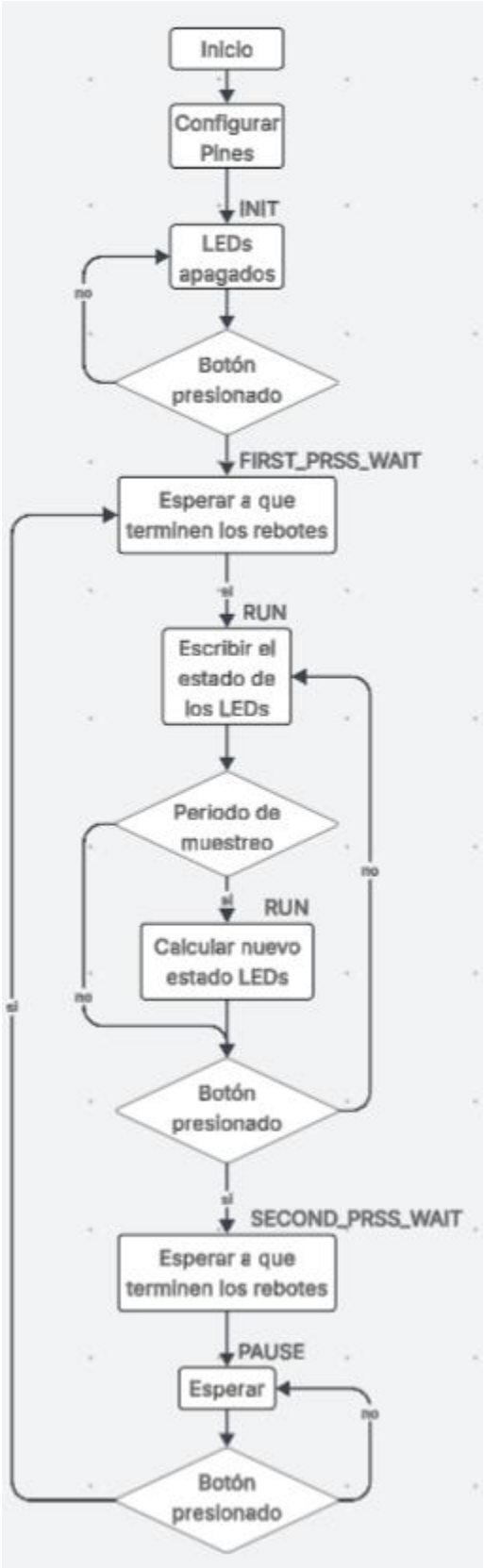


Figura 4. Diagrama de flujo.

Validación y Testeo:

Se realizó el montaje del circuito con ESP32, cuatro LEDs y un botón pulsador, siguiendo las conexiones definidas en el diseño. Posteriormente, se realizaron pruebas para verificar el comportamiento ante múltiples pulsaciones del botón.

Resultados:

- Al primer pulso del botón, la secuencia de LEDs inició correctamente, encendiendo un LED a la vez, con un intervalo de 500 ms.
- Al segundo pulso, la secuencia se detuvo manteniendo el LED actual encendido.
- Al tercer pulso, la secuencia se reanudó desde el LED donde se había pausado.
- Se verificó el comportamiento cíclico, alternando entre los estados RUN y PAUSE en cada pulsación adicional.
- Se constató que los tiempos de cambio de LED y la respuesta al botón cumplen con los requerimientos establecidos.

Por lo tanto, el sistema cumple con los requerimientos funcionales y no funcionales definidos en el análisis, demostrando un comportamiento estable y confiable bajo condiciones normales de operación.

Reto 2: Medición de Peso con Célula de Carga, HX711 y LCD

Análisis de requerimientos:

Objetivo:

Diseñar e implementar un sistema embebido que mida el peso de hasta 50 kg usando una célula de carga conectada a un amplificador HX711, muestre el valor en tiempo real en un LCD 16x2 y lo imprima en el Serial Monitor.

Requerimientos funcionales:

El sistema debe cumplir los siguientes requisitos funcionales:

1. Medición de peso:
   - Leer datos de la célula de carga a través del HX711
   - Escalar los datos según la sensibilidad definida para obtener el peso real.
   - El sistema debe medir pesos de hasta 50 Kg.
2. Visualización:
   - Mostrar el peso en tiempo real en un LCD 16x2.
   - Mostrar el valor en el Serial Monitor cada segundo.

Requerimientos no funcionales:

1. Confiabilidad
   - Lectura estable en la célula de carga.
2. Claridad
   - La interfaz (LCD y Serial) debe mostrar valores legibles.
3. Simplicidad:
   - Diseño compacto, apto para simulación o implementación física.

Restricciones ambientales y de usuario:

El sistema diseñado está hecho para solo correr en simulación.

Análisis y diseño del sistema (Arquitectura y componentes):

Objetivo:

Analizar y diseñar la arquitectura lógica y física de un sistema embebido para medir peso y mostrarlo en LCD y Serial.

- Microncontrolador ESP32
- Célula de carga
- Amplificador HX711
- LCD I2C 16x2
- Protoboard y Cables
- Entorno de simulación WOKWI.

Figura 5. Esquema de funcionamiento.

Diseño e Integración de Hardware:

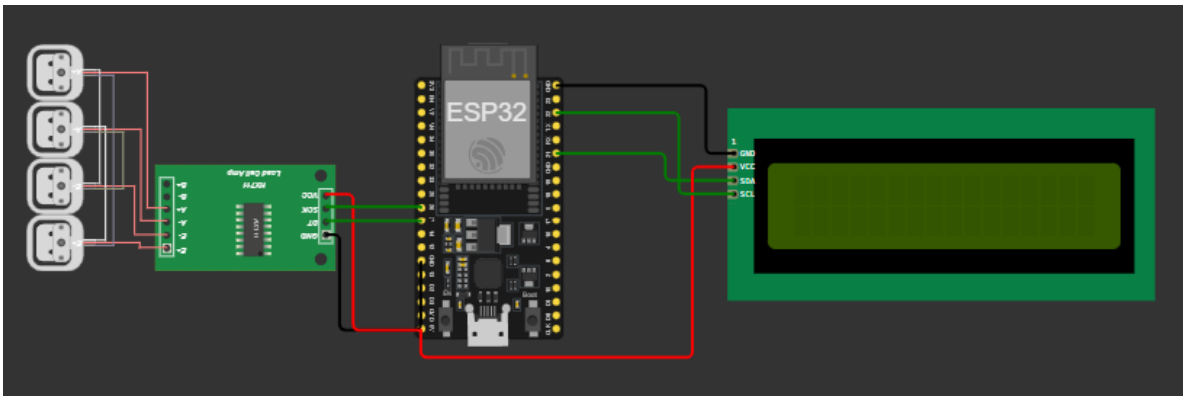


Figura 6. Arquitectura de hardware utilizada

Desarrollo del Firmware:

El firmware desarrollado es el siguiente:

```
#include "HX711.h"
#include <LiquidCrystal_I2C.h>
#include <Wire.h>
LiquidCrystal_I2C lcd(0x27,16,2);  // set the LCD address to 0x27

#define SENSITIVITY 0.002381
#define TIME_REPORT 1000

// HX711 circuit wiring
const int LOADCELL_DOUT_PIN = 27;
const int LOADCELL_SCK_PIN = 26;

unsigned long last_report = 0;

HX711 scale;

enum state {
  measure,
  report
};

state current_state = measure;

double reading = 0.0;
```

Figura 7a. Diseño firmware, importar librerías, definir constantes, estados y variables globales.

```
26  void setup() {
27    Serial.begin(115200);
28    lcd.init();                      // initialize the lcd
29    scale.begin(LOADCELL_DOUT_PIN, LOADCELL_SCK_PIN);
30    lcd.backlight();
31    lcd.clear();
32  }
```

Figura 7b. Diseño de firmware, configuración de comunicación serial, lcd y HX711.

```
34  void loop() {
35
36    switch(current_state)
37    {
38      case measure:
39        reading = scale.read()*SENSITIVITY;
40        if (millis() - last_report >= TIME_REPORT) current_state = report;
41        break;
42      case report:
43        lcd.home();
44        Serial.print("HX711 reading: ");
45        lcd.print("Weight: ");
46        Serial.println(reading);
47        lcd.print(reading, 2);
48        current_state = measure;
49        last_report = millis();
50        break;
51      default:
52        break;
53    }
54
55  }
```

Figura 7c. Diseño de firmware, ciclo de ejecución.

Documentación:

- Inputs: HX711 (GPIO 27 y 26).
- Outputs: LCD I2C y Serial monitor (GPIO 21 y 22, puerto usb).

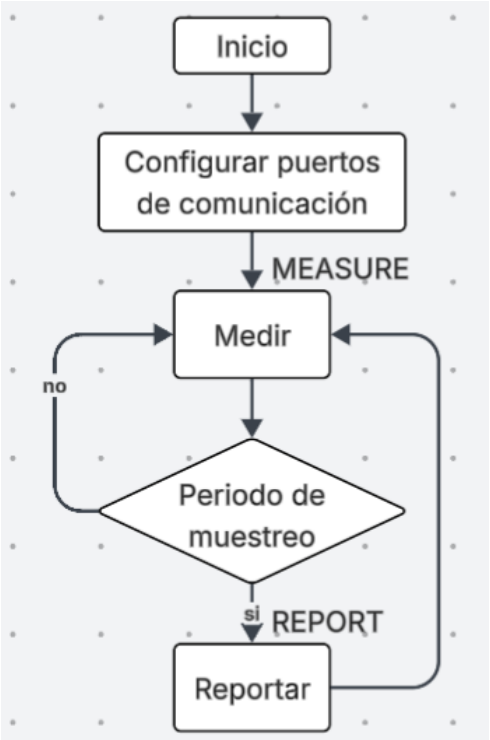


Figura 8. Diagrama de flujo.

Validación y Testeo:

Simulación en WOKWI:

- La lectura de peso se actualiza en tiempo real.
- Valores se muestran correctamente en LCD 16x2.
- Valores impresos en Serial Monitor coinciden con LCD.
- FSM funciona correctamente alternando entre adquisición y reporte.

Por lo tanto, el sistema cumple con los requerimientos funcionales y no funcionales, mostrando valores de peso precisos y en tiempo real en la interfaz visual y en el Serial Monitor.

Reto 3: Estación Meteorológica con DHT22 y LM35

Análisis de requerimientos:

Objetivo:

Diseñar e implementar un sistema embebido que mida temperatura y humedad usando dos sensores: DHT22 y LM35, mostrando los valores en un LCD OLED local y enviándolos por

Serial Monitor para monitoreo remoto. La lectura debe realizarse cada 5 segundos usando un enfoque no bloqueante.

Requerimientos funcionales:

El sistema debe cumplir los siguientes requisitos funcionales:

1. Medición de temperatura y humedad:
   - Sensor DHT22: medir temperatura y humedad ambiental.
   - Sensor LM35: medir temperatura adicional en rango analógico.
2. Visualización:
   - Mostrar el peso en tiempo real en un LCD OLED.
   - Mostrar valores en el monitor serial cada 5 segundos.
3. Enfoque no bloqueante:
   - Usar millis() para temporización en lugar de delay();
   - Prmite lecturas continuas y despliegue de información sin bloquear el MCU.

Requerimientos no funcionales:

1. Confiabilidad
   - Lectura estable y sin errores frecuentes.
2. Claridad
   - Valores legibles en OLED y Serial.
3. Simplicidad:
   - Sistema compacto, fácil de entender y ampliar.

Restricciones ambientales y de usuario:

1. Entorno:
   - Alimentación estable y condiciones normales de laboratorio.
2. Usuario:
   - No requiere interacción directa; sólo observar medidas.

Análisis y diseño del sistema (Arquitectura y componentes):

Objetivo:

Analizar y diseñar la arquitectura lógica y física de una estación meteorológica con sensores DHT22 y LM35, desplegando información en OLED y Serial Monitor cada 5 segundos.

- Microncontrolador ESP32
- Sensor DHT22 (Tempratura y humedad)
- Sensor LM35 (Temperatura analógica)
- Pantalla OLED 128x64 I2C (SSD1306)

- Protoboard y Cables
- Entorno de desarrollo Arduino IDE.

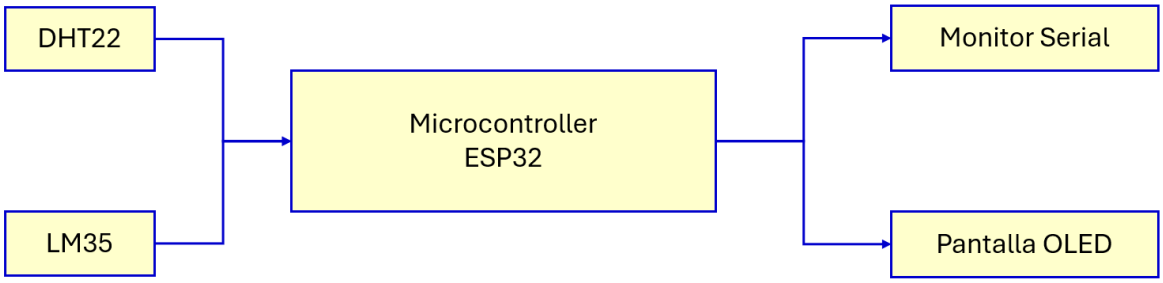


Figura 9. Esquema de funcionamiento.

Diseño e Integración de Hardware:

En la figura 10, el sensor lm35 es reemplazado por el potenciómetro que se observa.

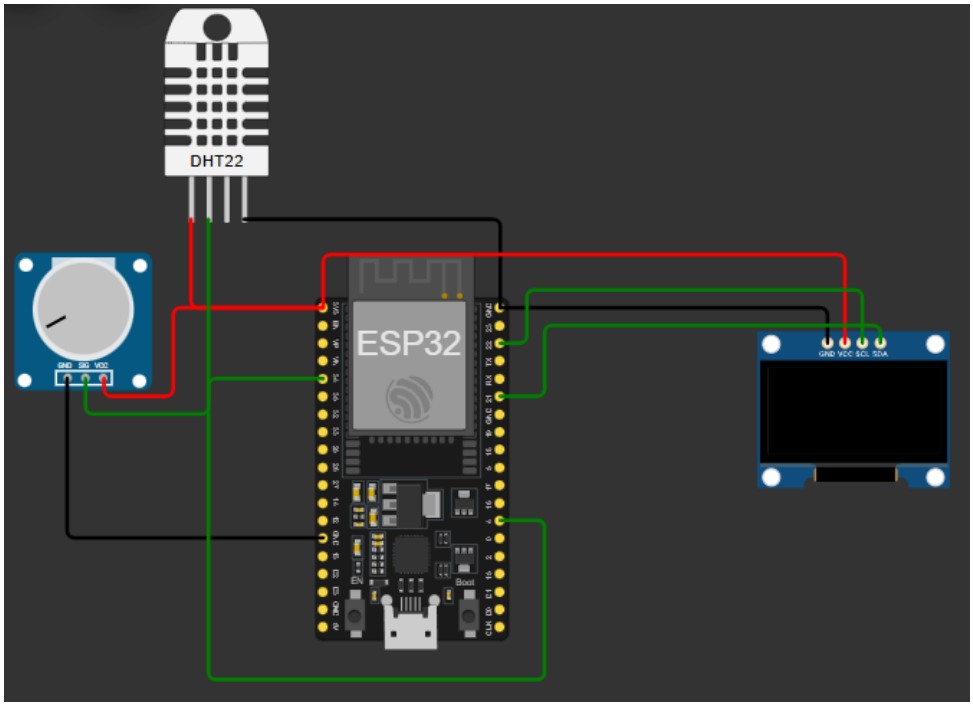


Figura 10. Arquitectura de hardware utilizada.

Desarrollo del Firmware:

El firmware desarrollado es el siguiente:

```
#include <DHT.h>
#include <Wire.h>
#include <Adafruit_GFX.h>
#include <Adafruit_SSD1306.h>

// ---------- OLED ----------
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_ADDR 0x3C

Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);

// ---------- DHT ----------
#define DHT_PIN 4
#define DHT_TYPE DHT22

DHT dht(DHT_PIN, DHT_TYPE);

// ---------- OTROS ----------
#define report_period 5000

unsigned long last_report = 0;
```

Figura 11a. Diseño firmware, importar librerías, definir constantes, objetos, estados y variables globales.

```
void setup() {
  Wire.begin(21, 22);
  Serial.begin(115200);

  dht.begin();

  if (!display.begin(SSD1306_SWITCHCAPVCC, OLED_ADDR)) {
    Serial.println("Error al iniciar OLED");
    while (true);
  }

  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);
}
```

Figura 11b. Diseño de firmware, configuración de puertos de comunicación.

```
40  void loop() {
41    static float val = 0;
42    static unsigned int num_samples = 0;
43
44    // Lectura analógica (ej. LM35 u otro)
45    val += analogRead(34) * 330.0 / 4095.0;
46    num_samples++;
47
48    if (millis() - last_report >= report_period) {
49      last_report = millis();
50
51      float avg_val = val / num_samples;
52
53      float humidity = dht.readHumidity();
54      float temperature = dht.readTemperature(); // Celsius
55
56      display.clearDisplay();
57      display.setCursor(0, 0);
58
59      if (isnan(temperature) || isnan(humidity)) {
60        Serial.println("Error leyendo DHT22");
61        display.println("Error DHT22");
62      } else {
63        Serial.println("Temp: " + String(temperature, 1) + " °C");
64        Serial.println("Humidity: " + String(humidity, 1) + " %");
65
66        display.println("Temp: " + String(temperature, 1) + " C");
67        display.println("Humidity: " + String(humidity, 1) + " %");
68      }
```

Figura 11c. Diseño de firmware, ciclo de ejecución primera parte.

```
70        Serial.println("---");
71        Serial.print("Temp LM75:");
72        Serial.println(String(avg_val, 2) + " C");
73        display.println("---");
74        display.print("Temp LM75:");
75        display.println(String(avg_val, 2) + " C");
76
77        display.display();
78
79        val = 0;
80        num_samples = 0;
81      }
82    }
```

Figura 11d. Diseño de firmware, ciclo de ejecución segunda parte.

Documentación:

- Inputs: DHT22 y LM35 (GPIO 4 y 34).
- Outputs: OLED 128x64 I2C y Serial monitor (GPIO 21 y 22, puerto usb).

Validación y Testeo:

Resultados:

- La lectura de temperatura y humedad se actualiza cada 5s.
- Los valores se muestran correctamente en OLED y Serial Monitor.
- En caso de error con DHT22, se despliega un mensaje de error.
- Promediado de lecturas del LM35 proporciona estabilidad en lecturas analógicas.
- Sistema no bloquea el MCU y permite lecturas continuas.

Por lo tanto, el sistema cumple con los requerimientos funcionales y no funcionales, permitiendo la observación de condiciones ambientales de manera confiable y en tiempo real.

Reto 3: Estación Meteorológica con DHT22 y LM35

Análisis de requerimientos:

Objetivo:

Diseñar e implementar un sistema embebido que permita ajustar y controlar la velocidad de un motor con encodificador (MOTORENCODER) conectado al ESP32 mediante un H-bridge.

Requerimientos funcionales:

El sistema debe cumplir los siguientes requisitos funcionales:

1. Control de arranque y parada:
   - El motor debe iniciar detenido al arrancar o reiniciar el programa.
   - Debe garantizarse que no haya movimiento involuntario al encender el sistema.
2. Entrada del setpoint:
   - El usuario ingresa la velocidad deseada desde la consola Serial.
   - Solo una tecla definida (KEY) activa el modo de ingreso de set point.
   - El set point puede variar de 0 a ±150 RPM, el signo define la dirección del motor.
   - Al presionar ENTER, la velocidad ingresada se hace efectiva.
3. Control PI:
   - La velocidad del motor se ajusta mediante un controlador proporcional-integral (PI) que calcula la señal de control y regula el H-bridge mediante PWM.
4. Visualización y reporte:
   - Mostrar RPM actual, set point, error y señal de control en pantalla OLED y monitor serial.

Requerimientos no funcionales:

1. Confiabilidad
   - Arranque seguro, respuesta estable a cambios de set point.
4. Claridad
   - Información de RPM, set point y error visible en OLED y Serial.
5. Simplicidad:
   - Código organizado por clases y archivos separados (Motor.h, Motor.ino, Motor_Control.ino).

Restricciones ambientales y de usuario:

2. Entorno:
   - Alimentación estable, motor con H-bridge compatible con PWM y dirección.
3. Usuario:
   - Interacción mediante consola Serial; no requiere hardware adicional para ingreso de set point.

Análisis y diseño del sistema (Arquitectura y componentes):

Objetivo:

Definir la arquitectura lógica y física de un sistema embebido de control de velocidad de motor con encodificador y H-bridge.

- Microncontrolador ESP32
- Motor DC con encodificador
- H-bridge para control de dirección y PWM
- Pantalla OLED 128x64 I2C (SSD1306)
- Protoboard y Cables
- Entorno de desarrollo Arduino IDE.

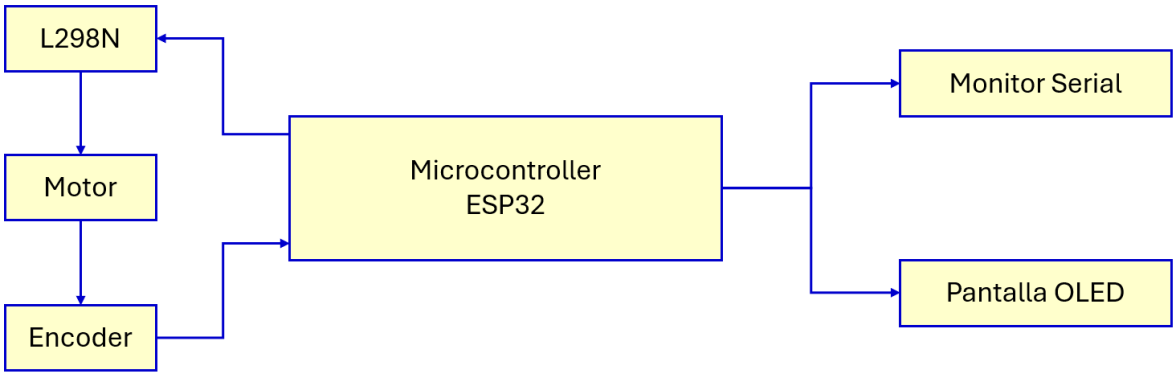


Figura 12. Esquema de funcionamiento.

Diseño e Integración de Hardware:

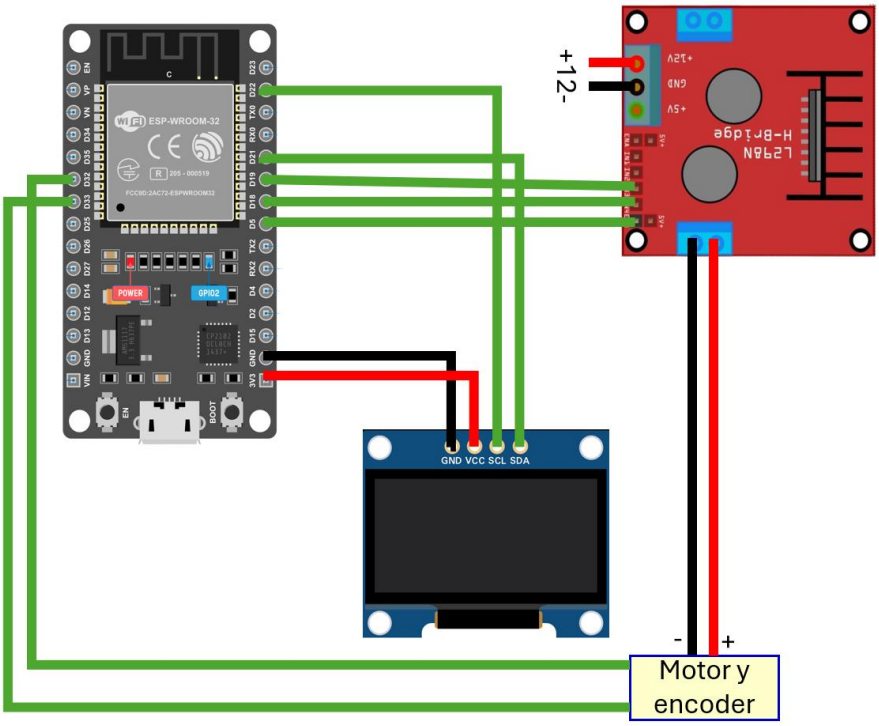


Figura 13. Arquitectura de hardware utilizada.

Desarrollo del Firmware:

El firmware desarrollado es el siguiente:

```
#ifndef MOTOR_H
#define MOTOR_H

#include "driver/pcnt.h"
#define ENC_A 32
#define ENC_B 33
#define PCNT_UNIT PCNT_UNIT_0
#define COUNTS_PER_REVOLUTION 480.0f // Example value, adjust as necessary
#define POS_PIN 19
#define NEG_PIN 18
#define PWM_PIN 5
```

Figura 14a. Diseño de firmware. Motor.h, definición de constantes e inclusión de librerías.

```
13  class Motor {
14      private:
15          // PID control parameters and state variables
16          float RPM;                  // Current revolutions per minute
17          float setpoint;             // Desired RPM
18          float error;                // Difference between setpoint and actual RPM
19          float Kp;                   // Proportional gain
20          float Ki;                   // Integral gain
21          float controlSignal;        // Control signal to the motor
22
23          // Private methods
24          void setRPM(unsigned long last_report);
25          void setError();
```

Figura 14b. Diseño de firmware. Motor.h, clase motor, declaración de características y métodos privados.

```
28          // Constructor to initialize PID parameters and state variables
29          Motor(float kp, float ki);
30
31          // Setter methods
32
33          // Method to compute the control signal using PID control
34          void computeControlSignal();
35          // Method to set a new RPM setpoint
36          void setSetpoint(float sp);
37
38          // Method to set rpm directly (for testing purposes)
39          void setRPMDirect(float rpm);
40
41          // Method to clear the integral term (if needed)
42          void clearIntegral();
43
44          // Getter methods
45
46          // Method to get the current RPM
47          float getRPM(unsigned long last_report);
48          // Method to get the current control signal
49          float getControlSignal();
50          // Method to get the current error
51          float getError();
52          // Method to get the current setpoint
53          float getSetpoint();
54  };
```

Figura 14c. Diseño de firmware. Motor.h, declaración de métodos públicos.

```cpp
#include "Motor.h"

volatile static float integral = 0.0f; // Moved integral to file scope to retain its value across calls

Motor::Motor(float kp, float ki)
    : RPM(0.0f), setpoint(0.0f), error(0.0f), Kp(kp), Ki(ki), controlSignal(0) {}

void Motor::setRPM(unsigned long last_report) {
    // Assuming counts is the number of encoder counts in the given time interval (in seconds)
    // and that we know the counts per revolution (CPR) of the motor encoder.
    int16_t counts;
    pcnt_get_counter_value(PCNT_UNIT, &counts);
    unsigned long current_time = millis();
    float timeInterval = (current_time - last_report) / 1000.0f; // Convert ms to seconds
    RPM = (counts / (COUNTS_PER_REVOLUTION*2)) * (60.0f / timeInterval);
    pcnt_counter_clear(PCNT_UNIT); // Clear counter after reading
}

void Motor::setError() {
    error = setpoint - RPM;
}
```

Figura 14d. Diseño de firmware. Motor.ino, inluir librerías, definición de variables globales y métodos primera parte.

```cpp
void Motor::computeControlSignal() {
    setError();
    // Simple PI control
    float P = Kp * error;
    integral += error; // Accumulate the integral
    float I = Ki * integral;

    float output = P + I;
    controlSignal = output;

    // Clamp control signal to valid range (e.g., 0 to 255 for an 8-bit PWM)
    if(output < 0 ) {
        output = -output;
        digitalWrite(POS_PIN, LOW);
        digitalWrite(NEG_PIN, HIGH);
    } else {
        digitalWrite(POS_PIN, HIGH);
        digitalWrite(NEG_PIN, LOW);
    }
    if (output > 255) {
        output = 255;
    }
    analogWrite(PWM_PIN, static_cast<unsigned int>(output));
}
```

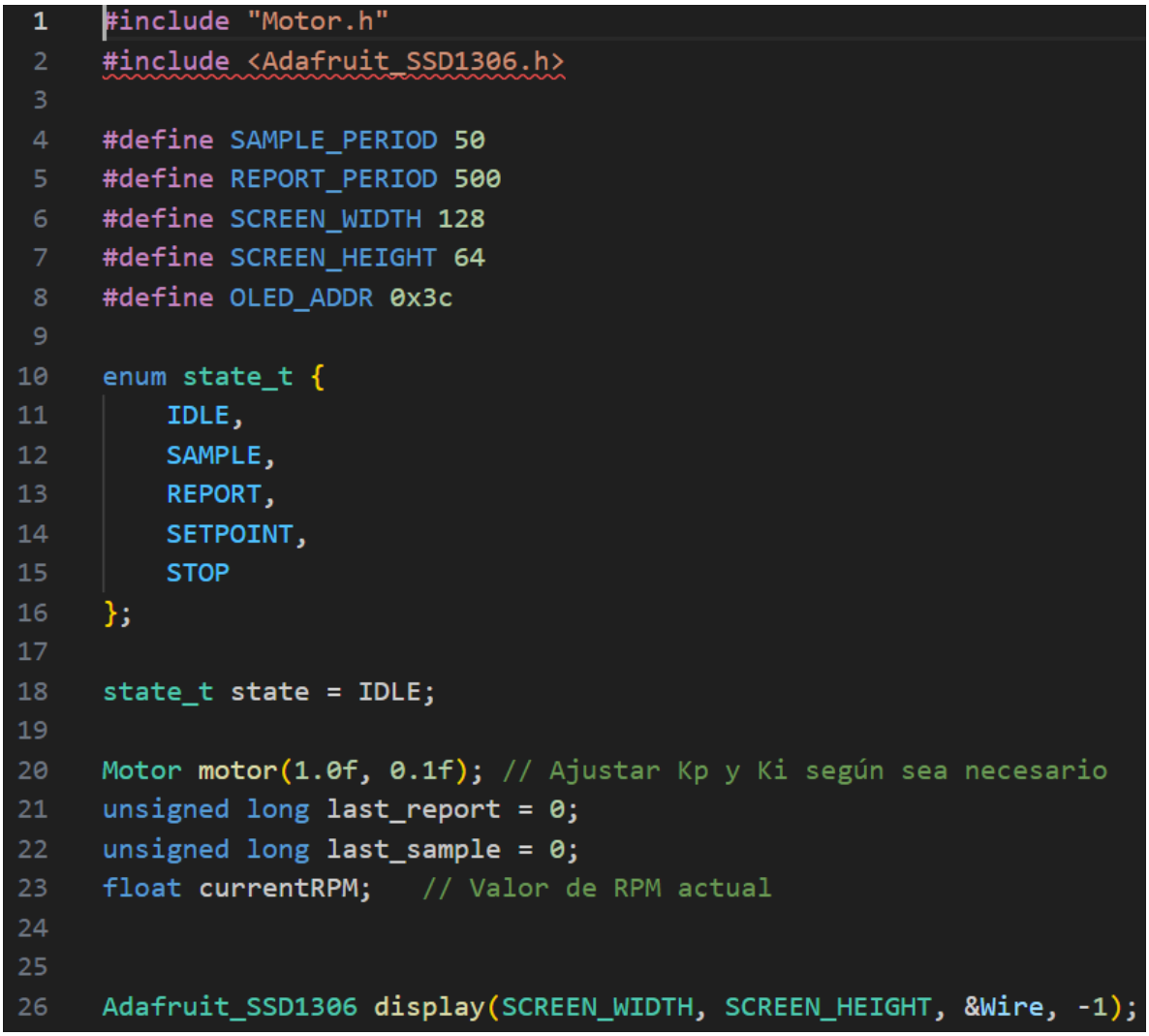
Figura 14e. Diseño de firmware. Motor.ino, definición de funciones segunda parte.

```
48  void Motor::setSetpoint(float sp) {
49      setpoint = sp;
50  }
51
52  void Motor::setRPMDirect(float rpm) {
53      if (rpm < 0) {
54          rpm = -rpm;
55          digitalWrite(POS_PIN, LOW);
56          digitalWrite(NEG_PIN, HIGH);
57      } else {
58          digitalWrite(POS_PIN, HIGH);
59          digitalWrite(NEG_PIN, LOW);
60      }
61      RPM = rpm;
62      analogWrite(PWM_PIN, static_cast<unsigned int>(rpm / 150 * 255)); // Assuming rpm max is 150 for testing purposes
63  }
64
65  float Motor::getRPM(unsigned long last_report) {
66      setRPM(last_report);
67      return RPM;
68  }
69
70  float Motor::getControlSignal() {
71      return controlSignal;
72  }
```

Figura 14f. Diseño de firmware. Motor.ino, definición de funciones tercera parte.

```
74  float Motor::getError() {
75      setError();
76      return error;
77  }
78
79  float Motor::getSetpoint() {
80      return setpoint;
81  }
82
83  void Motor::clearIntegral() {
84      // Clear the integral term (if needed)
85      // This function can be expanded if integral is made a member variable
86      integral = 0.0f;
87  }
```

Figura 14g. Diseño de firmware. Motor.ino, definición de funciones cuarta parte.



```
#include "Motor.h"
#include <Adafruit_SSD1306.h>

#define SAMPLE_PERIOD 50
#define REPORT_PERIOD 500
#define SCREEN_WIDTH 128
#define SCREEN_HEIGHT 64
#define OLED_ADDR 0x3c

enum state_t {
    IDLE,
    SAMPLE,
    REPORT,
    SETPOINT,
    STOP
};

state_t state = IDLE;

Motor motor(1.0f, 0.1f); // Ajustar Kp y Ki según sea necesario
unsigned long last_report = 0;
unsigned long last_sample = 0;
float currentRPM;   // Valor de RPM actual


Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, -1);
```

Figura 14h. Diseño de firmware. Motor_control.ino, incluir librerías, definir variables, estados y objetos.

```
28  void setup() {
29      Wire.begin(21, 22);
30      Serial.begin(115200);
31
32      pinMode(ENC_A, INPUT_PULLUP);
33      pinMode(ENC_B, INPUT_PULLUP);
34      pinMode(POS_PIN, OUTPUT);
35      pinMode(NEG_PIN, OUTPUT);
36      pinMode(PWM_PIN, OUTPUT);
37
38      pcnt_config_t pcnt_config{};
39      pcnt_config.pulse_gpio_num = ENC_A;
40      pcnt_config.ctrl_gpio_num  = ENC_B;
41      pcnt_config.channel        = PCNT_CHANNEL_0;
42      pcnt_config.unit           = PCNT_UNIT;
43      pcnt_config.pos_mode       = PCNT_COUNT_INC;
44      pcnt_config.neg_mode       = PCNT_COUNT_DEC;
45      pcnt_config.lctrl_mode     = PCNT_MODE_REVERSE;
46      pcnt_config.hctrl_mode     = PCNT_MODE_KEEP;
47      pcnt_config.counter_h_lim  = 32767;
48      pcnt_config.counter_l_lim  = -32768;
49
50      pcnt_unit_config(&pcnt_config);
51
52      pcnt_set_filter_value(PCNT_UNIT, 200);  // filtra rebotes
53      pcnt_filter_enable(PCNT_UNIT);
54
55      pcnt_counter_clear(PCNT_UNIT);
56      pcnt_counter_resume(PCNT_UNIT);
```

Figura Fgirua 14i. Diseño de firmware. Motor_control.ino, configuración de puertos de comunicación primera parte.

```
58      display.begin(SSD1306_SWITCHCAPVCC, OLED_ADDR);
59      display.clearDisplay();
60
61      display.setTextSize(1);
62      display.setTextColor(SSD1306_WHITE);
63  }
```

Figura 14j. Diseño de firmware. Motor_Control.ino, configuración de puertos de comunicación segunda parte.

```
65  void loop() {
66      float controlSignal;
67      float sp;
68      // Finite state machine for motor control
69      switch (state) {
70          case IDLE:
71              // Esperar comando para iniciar muestreo
72              if (Serial.available() > 0) {
73                  char c = Serial.read();
74                  if (c == 's') { // Comando para iniciar muestreo
75                      state = SETPOINT;
76                  } else if (c == 'x'){
77                      state = STOP;
78                  }
79              } else if(millis() - last_sample > SAMPLE_PERIOD) {
80                  state = SAMPLE;
81              } else if(millis() - last_report > REPORT_PERIOD) {
82                  state = REPORT;
83              }
84              break;
```

Figura 14k. Diseño de firmware. Motor_Control.ino, ciclo de ejecución primera parte.

```
85          case SAMPLE:
86              // Muestrear RPM y calcular señal de control
87              currentRPM = motor.getRPM(last_sample);
88              motor.getError();
89
90              motor.computeControlSignal();
91              state = IDLE;
92              last_sample = millis();
93              break;
```

Figura 14l. Diseño de firmware. Motor_Control.ino, ciclo de ejecución segunda parte.

```
94          case REPORT:
95              // Reportar datos por Serial y pantalla OLED
96              controlSignal = motor.getControlSignal();
97              display.clearDisplay();
98              display.setCursor(0, 0);
99              Serial.print("RPM: ");
100             Serial.print(currentRPM);
101             Serial.print(" | Set point: ");
102             Serial.print(motor.getSetpoint());
103             Serial.print(" | Error: ");
104             Serial.println(motor.getError());
105             Serial.print("Control Signal: ");
106             Serial.println(controlSignal);
107             display.print("RPM: ");
108             display.println(currentRPM);
109             display.print("Set point: ");
110             display.println(motor.getSetpoint());
111             display.print("Err: ");
112             display.println(motor.getError());
113             display.display();
114             state = IDLE;
115             last_report = millis();
116             break;
```

Figura 14m. Diseño de firmware. Motor_Control.ino, ciclo de ejecución tercera parte.

```
117         case SETPOINT:
118             // Esperar comando para cambiar setpoint o detener
119             while (Serial.available() == 0); // Esperar hasta que haya datos
120             sp = Serial.parseFloat();
121             motor.setSetpoint(sp);
122             state = IDLE;
123             break;
124 
125         case STOP:
126             // Detener motor y limpiar estados
127             motor.setRPMDirect(0);
128             motor.setSetpoint(0);
129             motor.clearIntegral();
130             state = IDLE;
131             break;
132     }
133 }
```

Figura 14n. Diseño de firmware. Motor_Control.ino, ciclo de ejecución cuarta parte.

Documentación:

- Estados:
    - IDLE: Espera de comandos y temporización para muestreo/reporte
    - SETPOINT: Espera ingreso de velocidad por serial
    - SAMPLE: Muestreo de RPM actual mediante encodificador y generación de señal de control.
    - REPORT: despliegu de datos en OLED y Serial
    - STOP: deteción del motor y limpieza de integral.
- Inputs: ENC_A yENC_B (encoder), consola serial (set point).
- Outputs: POS_PIN, NEG_PIN (dirección del motor), PWM_PIN (velocidad), OLED I2C, serial monitor.

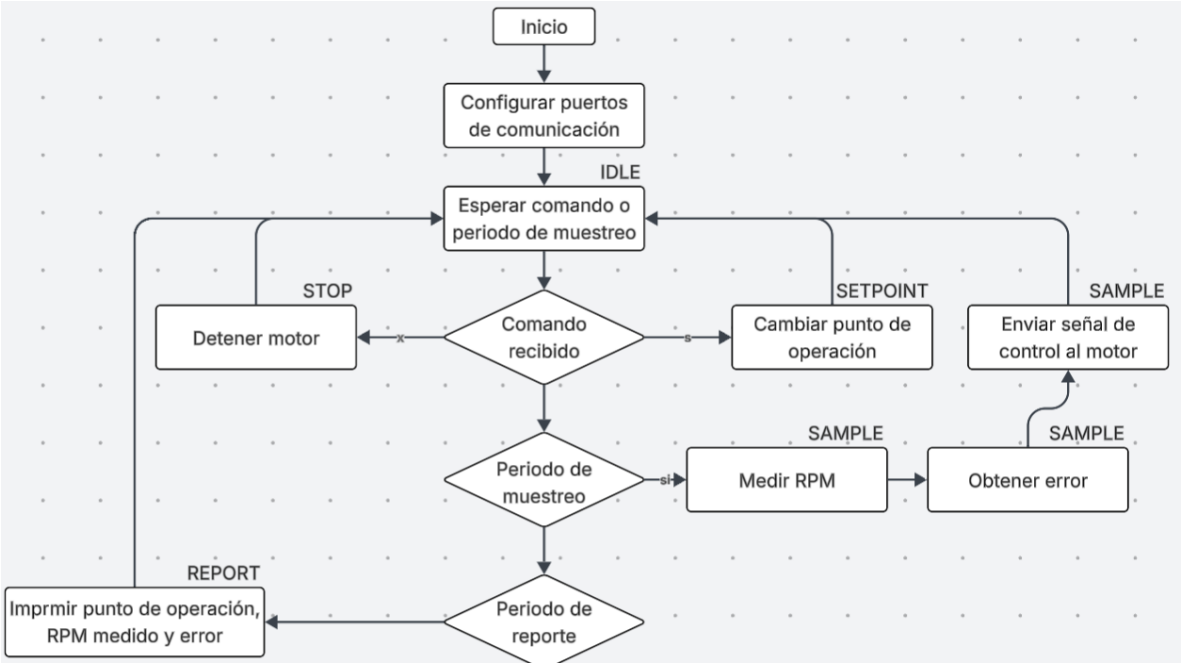


Figura 15. Diagrama de flujo.

Validación y Testeo:

Resultados:

- Arranque seguro: motor permanece detenido al encender ESP32.
- Ingreso de set point: el motor alcanza la velocidad ingresada correctamente.
- Manejo de errores: valores fuera de rango generan mensaje de error y se solicita reingreso.
- Dirección de giro correcta según signo del set point.
- Señal de control PI ajusta PWM de manera estable, evitando overshoot excesivo.
- Reporte en OLED y Serial Monitor muestra RPM, set point, error y control signal en tiempo rea.

Por lo tanto, el sistema cumple con todos los requerimientos funcionales y no funcionales, garantizando seguridad, confiabilidad y facilidad de uso para controlar la velocidad de un motor con encodificador mediante ESP32.

Referencias:

Repositorio de github con todos los códigos:

https://github.com/joanvel/Lab01_Diseno_e_Implementacion_de_Sistemas_Embebidos