Instituto Politécnico Nacional

Unidad Profesional Interdisciplinaria de Ingeniería Campus Zacatecas

Ingeniería en Mecatrónica

Unidad de Aprendizaje:
Sistemas Operativos en Tiempo Real

Nombre del Docente:
Ramón Jaramillo Martínez

Nombre del alumno:
Diego Esteban Cabrera Soto

Proyecto final “Secador de filamento con RTOS”

Zacatecas, Zac. Lunes 12 de enero del 2026

1.Introducción

En la impresión 3D, el filamento plástico es altamente sensible a la humedad del ambiente. Materiales como PLA, PETG, ABS y Nylon absorben vapor de agua, lo que provoca defectos durante la impresión, tales como burbujas, mala adhesión entre capas y una disminución en la resistencia mecánica de las piezas impresas. Por esta razón, es fundamental mantener el filamento seco antes y durante su uso.

Como proyecto final se desarrolló un secador de filamento, capaz de regular la temperatura y la ventilación interna con el fin de eliminar la humedad del material. El sistema está basado en un microcontrolador ESP32, el cual permite implementar algoritmos avanzados de control y ejecutar múltiples tareas simultáneamente mediante un sistema operativo en tiempo real (RTOS).

El secador emplea un calefactor controlado por un relevador de estado sólido (SSR) bajo un controlador PID clásico, y un ventilador de corriente alterna gobernado por un relevador electromecánico con lógica difusa. La supervisión del proceso se realiza mediante un sensor BME280 y una pantalla OLED de 128×64 píxeles.

2. Marco Teórico

2.1 Sistema térmico de un secador de filamento

Un secador de filamento para impresión 3D es un sistema térmico cuyo objetivo es mantener el material plástico (PLA, PETG, ABS, Nylon, etc.) a una temperatura y humedad controladas con el fin de eliminar la humedad absorbida del ambiente. La presencia de agua en el filamento provoca defectos de impresión como burbujas, hilos y baja resistencia mecánica.

Desde el punto de vista dinámico, el secador puede modelarse como un sistema térmico de primer orden, donde la temperatura del aire dentro de la cámara responde de forma progresiva a la potencia suministrada al calefactor. La dinámica del sistema está determinada por la capacidad térmica del aire, del filamento y de la estructura del secador, así como por las pérdidas de calor hacia el ambiente.

Este comportamiento permite aplicar técnicas clásicas de control para regular la temperatura de manera estable.

2.2 Microcontrolador ESP32

El ESP32 es un microcontrolador de alto rendimiento que integra un procesador de 32 bits, memoria, periféricos de comunicación y capacidades de control en tiempo real. Su arquitectura permite ejecutar múltiples tareas de manera simultánea mediante un sistema operativo en tiempo real (RTOS).

En este proyecto, el ESP32 se encarga de leer los sensores de temperatura y humedad, ejecutar los algoritmos de control PID y control difuso, las señales de control para el calefactor y el ventilador y administrar la interfaz de usuario mediante una pantalla OLED.

El uso del RTOS permite dividir el sistema en tareas independientes, mejorando la organización del software, la estabilidad del sistema y el tiempo de respuesta ante cambios en las variables físicas.

2.3 Sensor de temperatura y humedad BME280

El BME280 es un sensor digital que mide temperatura, humedad relativa y presión atmosférica. En este sistema se utiliza para monitorear las condiciones dentro de la cámara del secador de filamento.

La temperatura medida permite evaluar el estado térmico del sistema, mientras que la humedad relativa indica la cantidad de vapor de agua presente en el aire. Esta información es fundamental para determinar si el filamento está siendo correctamente secado.

El sensor se comunica con el ESP32 mediante un bus digital (I^2C), lo que permite una adquisición de datos confiable y precisa.

2.4 Relevador de estado sólido (SSR)

El relevador de estado sólido (SSR) es un dispositivo electrónico que permite conmutar cargas de corriente alterna sin partes mecánicas móviles. En este proyecto, el SSR se utiliza para controlar el calefactor del secador de filamento, el cual opera con alimentación de corriente alterna.

El SSR es activado por una señal digital proveniente del ESP32 y proporciona aislamiento eléctrico entre la etapa de control y la etapa de potencia. Al ser controlado mediante una señal modulada en el tiempo (derivada del controlador PID), el SSR regula la energía entregada al calefactor y, por lo tanto, la temperatura de la cámara.

2.5 Control PID clásico

El control Proporcional–Integral–Derivativo (PID) es una estrategia de control ampliamente utilizada en sistemas térmicos. En el secador de filamento, el PID se emplea para regular la temperatura del aire dentro de la cámara.

El controlador calcula el error como la diferencia entre la temperatura deseada y la temperatura medida por el sensor BME280. A partir de este error, el PID genera una señal de control que ajusta la potencia del calefactor a través del SSR.

- La acción proporcional corrige el error instantáneo.
- La acción integral elimina el error acumulado.
- La acción derivativa anticipa cambios bruscos en la temperatura.

La combinación de estas acciones permite mantener una temperatura estable con bajo sobreimpulso y buena precisión.

2.6 Control del ventilador mediante lógica difusa

El ventilador del secador se controla mediante un relevador electromecánico que conmuta una carga de corriente alterna. A diferencia del calefactor, el ventilador es gobernado por un **control difuso (fuzzy logic)**.

La lógica difusa permite tomar decisiones basadas en reglas lingüísticas, por ejemplo:

1) Si la temperatura es alta y la humedad es baja, entonces reducir ventilación.
2) Si la humedad es alta, entonces aumentar ventilación.

Este tipo de control es especialmente adecuado cuando el comportamiento del sistema no es estrictamente lineal, como ocurre con la circulación de aire y la evaporación de humedad dentro de la cámara.

El controlador difuso evalúa variables como temperatura, humedad y error térmico, y determina si el ventilador debe encenderse o apagarse para optimizar el proceso de secado.

2.7 Interfaz de usuario con pantalla OLED

El sistema cuenta con una pantalla OLED de 128×64 píxeles que permite mostrar en tiempo real la información relevante del proceso. En esta pantalla se visualizan la temperatura, la humedad y el error de temperatura, lo que permite al usuario supervisar el estado del secador.

La pantalla se comunica con el ESP32 mediante el bus I²C, lo que reduce el número de pines necesarios y facilita su integración en el sistema.

2.8 Programación basada en RTOS

El firmware del secador de filamento fue desarrollado bajo un sistema operativo en tiempo real (RTOS). Este enfoque permite dividir el programa en tareas independientes, como:

- Lectura del sensor BME280.
- Ejecución del control PID.
- Ejecución del control difuso.
- Actualización de la pantalla OLED.

Cada tarea se ejecuta de manera concurrente y con prioridades definidas, lo que garantiza que el sistema responda de forma rápida y estable ante cambios de temperatura y humedad.

El uso de RTOS incrementa la confiabilidad, la modularidad y la escalabilidad del sistema, permitiendo una operación más robusta que en sistemas basados en programación secuencial.

3. Objetivo

Implementar un sistema electrónico de secado de filamento para impresión 3D utilizando un microcontrolador ESP32, capaz de regular la temperatura y la humedad dentro de una cámara mediante un controlador PID, lógica difusa y programación en tiempo real (RTOS), garantizando condiciones óptimas para el almacenamiento y uso del filamento.

4.Desarrollo

El desarrollo del proyecto comenzó con el diseño del sistema electrónico y la definición de los bloques funcionales. Se establecieron cuatro etapas principales: adquisición de datos, control térmico, control de ventilación e interfaz de usuario.

El sensor BME280 y la pantalla OLED de 128×64 píxeles fueron conectados al bus de comunicación I^2C del ESP32, permitiendo la adquisición de datos de temperatura y humedad, así como la visualización de la información utilizando únicamente dos líneas de comunicación.
El ESP32 dispone de pines específicos para el control de los actuadores. Un pin configurado como salida PWM gobierna el relevador de estado sólido (SSR), el cual controla el comportamiento del calefactor de corriente alterna mediante el controlador PID, ajustando la potencia entregada según el error térmico.
Asimismo, un pin digital de salida se utiliza para activar un módulo de relevador electromecánico encargado de conmutar el ventilador de corriente alterna. La activación de este relevador es determinada por el controlador difuso, el cual regula la ventilación en función de la temperatura y la humedad dentro de la cámara.
La programación del sistema se realizó bajo un entorno RTOS, lo que permitió dividir el funcionamiento en tareas independientes, mejorando la estabilidad y la respuesta del sistema.

5. Resultados

El sistema logró mantener la temperatura y la humedad dentro de los rangos establecidos para el secado de filamento. La acción del controlador PID proporcionó una regulación térmica estable, mientras que el control difuso permitió una ventilación adecuada para la eliminación de humedad.

La pantalla OLED permitió supervisar el proceso en tiempo real, mostrando valores coherentes y estables durante la operación.

Figura 1. visualización de variables en la pantalla OLED

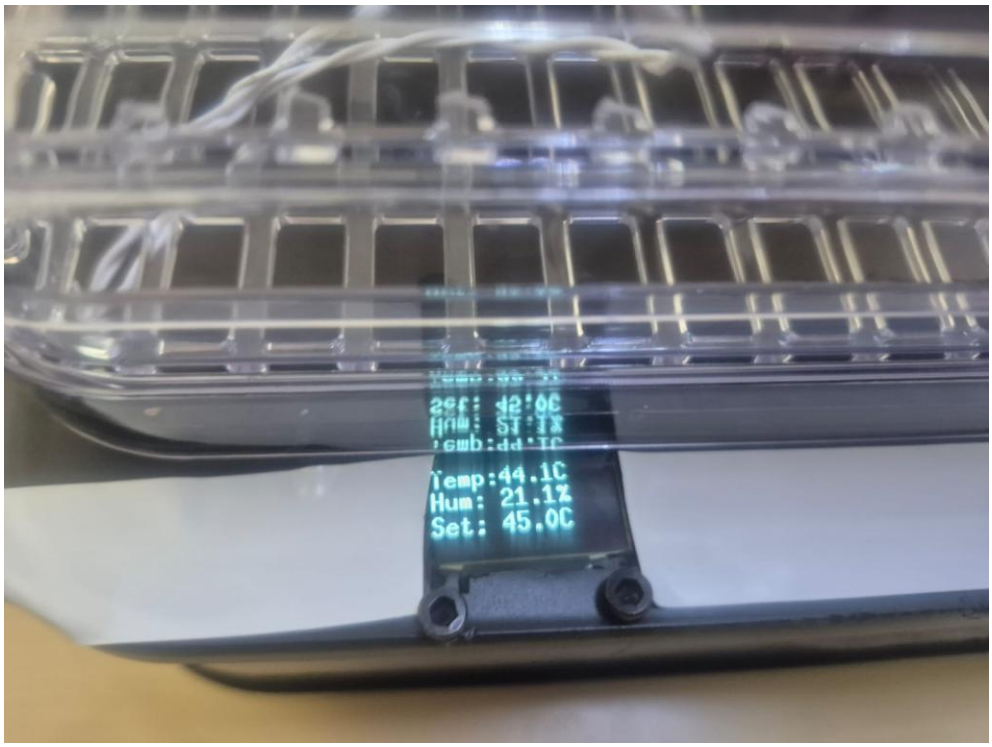

Figura 2. Secador de filamento instalado en banco de pruebas

6. Conclusiones

El secador de filamento desarrollado demostró que el ESP32 es una plataforma adecuada para aplicaciones de control térmico en tiempo real. La combinación de control PID y control difuso permitió mejorar la estabilidad térmica y la eficiencia del proceso de secado.

La implementación mediante RTOS facilitó la organización del software y garantizó una respuesta confiable del sistema ante cambios en las variables físicas.

7.Anexos

Código de programación en RTOS para la ESP32

```
#include <Wire.h>

#include <Adafruit_Sensor.h>

#include <Adafruit_BME280.h>

#include <U8g2lib.h>

#include <WiFi.h>          // Añadido para conexión WiFi

#include <FirebaseESP32.h>  // Añadido para Firebase


U8G2_SSD1306_128X64_NONAME_1_HW_I2C myScreen(U8G2_R0, U8X8_PIN_NONE);

Adafruit_BME280 bme;


// Credenciales WiFi

const char* WIFI_SSID = "INFINITUM5D9B_2.4";

const char* WIFI_PASSWORD = "CVXmG6gWNF";


// Configuración Firebase

const char* FIREBASE_HOST = "https://esp32-dex-dryer-default-rtdb.firebaseio.com/";

const char* FIREBASE_AUTH = "AIzaSyBj7Xppjmqawd1Lho2xcyhRX_j7Et3qDXU";


FirebaseData fbdo;      // Objeto para datos Firebase

FirebaseAuth auth;      // Objeto para autenticación

FirebaseConfig config;  // Configuración de Firebase


// Pines del hardware

const uint8_t relay_pin = 25;  // Relé para ventilador

const uint8_t ssr_pin   = 17;  // SSR para calefacción
```

```
const uint8_t encoder_clk = 19;  // Canal A del encoder
const uint8_t encoder_dt  = 18;  // Canal B del encoder
const uint8_t button      = 23;   // Botón pulsador

// Configuración PWM para SSR
const int SSR_PWM_CHANNEL = 0;
const int SSR_PWM_FREQ    = 100;  // 100 Hz
const int SSR_PWM_RES     = 8;    // 8 bits (0-255)
const int SSR_MAX_DUTY    = 255;

// Variables del sistema
float setTemp = 45.0;    // Temperatura objetivo
float currentTemp = 0;   // Temperatura actual leída
float currentHum  = 0;   // Humedad actual leída

SemaphoreHandle_t dataMutex;  // Mutex para proteger datos compartidos

// Parámetros del controlador PID
float kp = 20.0, ki = 1.25, kd = 0.2;
float pidIntegral = 0;  // Acumulador del término integral
float pidLastErr  = 0;  // Último error para derivada

// Variables del encoder (volatile para ISR)
volatile int8_t encoderDirection = 0;  // Dirección: -1, 0, +1
volatile int8_t stepAccumulator  = 0;  // Acumulador de pasos
volatile uint8_t lastEncoderState = 0; // Estado anterior del encoder
```

```
float lastTemp = 0;      // Temperatura anterior para calcular tasa
bool fanState = false;   // Estado actual del ventilador

// Intervalo para actualizar Firebase (milisegundos)
const unsigned long FB_UPDATE_INTERVAL = 5000;  // Enviar cada 5 segundos
unsigned long lastFirebaseUpdate = 0;

// Prototipos de funciones para nuevas tareas
void TaskFirebaseUpdate(void *pvParameters);
void TaskWiFiManager(void *pvParameters);

// Rutina de servicio de interrupción para encoder
void IRAM_ATTR readEncoder() {
  // Lee estado actual del encoder (valor de 2 bits)
  uint8_t state = (digitalRead(encoder_clk) << 1) | digitalRead(encoder_dt);
  uint8_t transition = (lastEncoderState << 2) | state;  // Código de transición de 4 bits

  // Determina dirección basada en secuencia de código Gray
  switch (transition) {
    case 0b0001: case 0b0111: case 0b1110: case 0b1000:
      stepAccumulator++; break;  // Sentido horario
    case 0b0010: case 0b0100: case 0b1101: case 0b1011:
      stepAccumulator--; break;  // Sentido antihorario
  }

  lastEncoderState = state;  // Guarda estado actual
```

```
  // Verifica si se completó un detente (4 pasos por detente)
  if (stepAccumulator >= 4) {
    encoderDirection = +1;  // Detente horario completado
    stepAccumulator = 0;
  } else if (stepAccumulator <= -4) {
    encoderDirection = -1;  // Detente antihorario completado
    stepAccumulator = 0;
  }
}

// Cálculo del controlador PID
uint8_t computePID(float temp) {
  float error = setTemp - temp;  // Error = objetivo - actual

  pidIntegral += error;  // Acumula error para término integral
  pidIntegral = constrain(pidIntegral, -50, 50);  // Anti-windup

  // Salida PID: P + I + D
  float output = kp * error + ki * pidIntegral + kd * (error - pidLastErr);
  pidLastErr = error;  // Guarda error para siguiente derivada

  return constrain(output, 0, 100);  // Limita a 0-100%
}

// Controla SSR con PWM basado en salida PID
void driveSSR(uint8_t pidPercent) {
```

```
  // Mapea 0-100% a duty cycle 255-0 (invertido porque SSR es activo-bajo)
  uint8_t duty = map(pidPercent, 0, 100, 255, 255 - SSR_MAX_DUTY);
  ledcWrite(SSR_PWM_CHANNEL, duty);
}

// Lógica difusa para control del ventilador
bool fuzzyFan(float temp) {
  float error = temp - setTemp;  // Positivo = por encima del objetivo
  float dTemp = temp - lastTemp; // Tasa de cambio de temperatura

  // Reglas difusas:
  if (error > 2.0 && dTemp > 0.05) return false;  // Caliente y aumentando
  if (error > 1.0 && dTemp > 0.0)  return false;  // Encima del objetivo y subiendo
  if (error < -1.5)                return true;   // Muy por debajo del objetivo
  if (abs(error) < 0.5 && abs(dTemp) < 0.03) return true;  // Estable cerca del objetivo

  return fanState;  // Mantiene estado actual
}

// Controla relé del ventilador
void driveFan(bool on) {
  fanState = on;
  // LOW = ENCENDIDO (relé activo-bajo)
  digitalWrite(relay_pin, on ? LOW : HIGH);
}

// Tarea 1: Maneja entrada del encoder
```

```
void TaskEncoder(void *pvParameters) {
  for (;;) {
    if (encoderDirection != 0) {
      xSemaphoreTake(dataMutex, portMAX_DELAY);
      setTemp += encoderDirection * 0.5;  // Cambia temp en 0.5°C por detente
      setTemp = constrain(setTemp, 30, 70);  // Limita rango 30-70°C
      encoderDirection = 0;  // Reinicia dirección
      xSemaphoreGive(dataMutex);
    }
    vTaskDelay(pdMS_TO_TICKS(20));  // Verifica cada 20ms
  }
}

// Tarea 2: Lee sensores y controla sistema
void TaskSensorControl(void *pvParameters) {
  for (;;) {
    // Lee datos del sensor
    float t = bme.readTemperature();
    float h = bme.readHumidity();

    xSemaphoreTake(dataMutex, portMAX_DELAY);
    currentTemp = t;
    currentHum  = h;

    // Calcula y aplica control PID
    uint8_t pid = computePID(t);
    driveSSR(pid);
```

```
    // Controla ventilador con lógica difusa
    bool fanCmd = fuzzyFan(t);
    driveFan(fanCmd);

    lastTemp = t;  // Guarda para siguiente cálculo de tasa
    xSemaphoreGive(dataMutex);

    vTaskDelay(pdMS_TO_TICKS(250));  // Actualiza cada 250ms
  }
}

// Tarea 3: Actualiza pantalla OLED
void TaskDisplay(void *pvParameters) {
  char b1[8], b2[8], b3[8];  // Buffers para cadenas formateadas

  for (;;) {
    xSemaphoreTake(dataMutex, portMAX_DELAY);
    // Convierte floats a strings
    dtostrf(currentTemp, 4, 1, b1);
    dtostrf(currentHum,  4, 1, b2);
    dtostrf(setTemp,     4, 1, b3);
    xSemaphoreGive(dataMutex);

    // Muestra en OLED
    myScreen.firstPage();
    do {
```

```
      myScreen.drawStr(0, 15, "Temp:");
      myScreen.drawStr(50, 15, b1);
      myScreen.drawStr(90, 15, "C");

      myScreen.drawStr(0, 32, "Hum:");
      myScreen.drawStr(50, 32, b2);
      myScreen.drawStr(90, 32, "%");

      myScreen.drawStr(0, 50, "Set:");
      myScreen.drawStr(50, 50, b3);
      myScreen.drawStr(90, 50, "C");
    } while (myScreen.nextPage());

    vTaskDelay(pdMS_TO_TICKS(500));  // Actualiza pantalla cada 500ms
  }
}

// NUEVA Tarea 4: Envía datos a Firebase
void TaskFirebaseUpdate(void *pvParameters) {
  for (;;) {
    unsigned long currentTime = millis();

    // Verifica si es momento de actualizar Firebase
    if (currentTime - lastFirebaseUpdate >= FB_UPDATE_INTERVAL) {
      float temp, hum, target;

      // Obtiene datos protegidos
```

```
  xSemaphoreTake(dataMutex, portMAX_DELAY);
  temp = currentTemp;
  hum = currentHum;
  target = setTemp;
  xSemaphoreGive(dataMutex);

  // Prepara datos JSON
  String path = "/sensor_data";
  FirebaseJson json;

  json.set("temperature", temp);
  json.set("humidity", hum);
  json.set("target_temperature", target);
  json.set("timestamp", millis() / 1000);  // Marca de tiempo Unix
  json.set("fan_state", fanState ? "ON" : "OFF");

  // Envía a Firebase
  if (Firebase.updateNode(fbdo, path, json)) {
    Serial.println("Firebase: Actualización exitosa");
  } else {
    Serial.println("Firebase: Error - " + fbdo.errorReason());
  }

  lastFirebaseUpdate = currentTime;
}

vTaskDelay(pdMS_TO_TICKS(1000));  // Verifica cada segundo
```

```
  }
}

// NUEVA Tarea 5: Maneja conexión WiFi
void TaskWiFiManager(void *pvParameters) {
  for (;;) {
    if (WiFi.status() != WL_CONNECTED) {
      Serial.println("Conectando a WiFi...");
      WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

      int attempts = 0;
      while (WiFi.status() != WL_CONNECTED && attempts < 20) {
        vTaskDelay(pdMS_TO_TICKS(500));
        attempts++;
        Serial.print(".");
      }

      if (WiFi.status() == WL_CONNECTED) {
        Serial.println("\n¡WiFi conectado!");
        Serial.print("IP: ");
        Serial.println(WiFi.localIP());
      } else {
        Serial.println("\nError conectando WiFi");
      }
    }

    vTaskDelay(pdMS_TO_TICKS(10000));  // Verifica conexión cada 10 segundos
```

```
  }
}

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

  // Inicializa WiFi
  WiFi.mode(WIFI_STA);
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

  // Configura Firebase
  config.host = FIREBASE_HOST;
  config.signer.tokens.legacy_token = FIREBASE_AUTH;
  Firebase.begin(&config, &auth);
  Firebase.reconnectWiFi(true);  // Reintenta si WiFi se desconecta

  // Inicializa pines de hardware
  pinMode(encoder_clk, INPUT_PULLUP);
  pinMode(encoder_dt,  INPUT_PULLUP);
  pinMode(button, INPUT_PULLUP);

  pinMode(relay_pin, OUTPUT);
  digitalWrite(relay_pin, HIGH);  // Inicia con ventilador APAGADO

  // Configura PWM para SSR
  ledcSetup(SSR_PWM_CHANNEL, SSR_PWM_FREQ, SSR_PWM_RES);
  ledcAttachPin(ssr_pin, SSR_PWM_CHANNEL);
```

```
  ledcWrite(SSR_PWM_CHANNEL, 255);  // Inicia con SSR APAGADO

  // Inicializa encoder
  lastEncoderState = (digitalRead(encoder_clk) << 1) | digitalRead(encoder_dt);
  attachInterrupt(digitalPinToInterrupt(encoder_clk), readEncoder, CHANGE);
  attachInterrupt(digitalPinToInterrupt(encoder_dt),  readEncoder, CHANGE);

  // Inicializa OLED
  myScreen.begin();
  myScreen.setFont(u8g2_font_10x20_tf);
  myScreen.setContrast(255);

  // Inicializa sensor BME280
  if (!bme.begin(BME280_ADDRESS_ALTERNATE)) {
    Serial.println("¡BME280 no encontrado!");
    while (1);
  }

  lastTemp = bme.readTemperature();  // Lectura inicial de temperatura

  // Crea mutex para protección de datos
  dataMutex = xSemaphoreCreateMutex();

  // Crea tareas RTOS con prioridades y núcleos asignados
  xTaskCreatePinnedToCore(TaskWiFiManager, "WiFi", 4096, NULL, 1, NULL, 0);    // Prioridad
baja
  xTaskCreatePinnedToCore(TaskEncoder, "Encoder", 2048, NULL, 3, NULL, 1);      // Prioridad
alta
```

```
  xTaskCreatePinnedToCore(TaskSensorControl, "Control", 4096, NULL, 4, NULL, 1); // Prioridad más alta
  xTaskCreatePinnedToCore(TaskDisplay, "Display", 4096, NULL, 2, NULL, 0);      // Prioridad media
  xTaskCreatePinnedToCore(TaskFirebaseUpdate, "Firebase", 4096, NULL, 1, NULL, 0); // Prioridad baja

  // Nota: Las tareas se asignan a diferentes núcleos:
  // Núcleo 0: WiFi, Display, Firebase (menos críticas en tiempo)
  // Núcleo 1: Encoder, Control (tareas críticas en tiempo)
}

void loop() {
  // Vacío - toda la funcionalidad está en tareas RTOS
  vTaskDelay(portMAX_DELAY);
}
```