Análisis de Implementación IoT con MicroPython y FreeRTOS

Tu Nombre

2 de julio de 2025

Índice

1 Resumen **3**

2 Introducción **3**
2.1 Motivación 3
2.2 Objetivos 3
2.2.1 Descripción del Problema 3
2.3 Estructura del proyecto 3

3 Estado del Arte **4**
3.1 Trabajos Relacionados 4
3.2 Conceptos teóricos 4
3.2.1 Sistemas Operativos de Tiempo Real (RTOS) 4
3.2.2 MicroPython 4
3.2.3 Concurrencia y Multitarea 4
3.2.4 ESP32 4
3.2.5 Protocolos de Comunicación IoT 5
3.2.6 Otras herramientas utilizadas 5
3.3 Conclusiones del Estado del Arte 5

4 Desarrollo **5**
4.1 Descripción del caso de estudio 5
4.1.1 Problemática 5

4.1.2 Objetivo del caso de estudio 5
4.1.3 Evaluación de solución del caso de estudio 6
4.2 Metodología de desarrollo de la solución 6
4.2.1 Especificación de requerimientos 6
4.2.2 Herramientas 6
4.3 Diseño de arquitectura de desarrollo 7
4.3.1 Integración e interfaces 7
4.3.2 Análisis de datos 7
4.4 Módulos 8
4.4.1 Diseño de módulos 8

5 Resultados **9**
5.1 Pruebas y Análisis 9
5.1.1 Mejoras de desarrollo 10

6 Conclusiones y Trabajo Futuro **10**
6.1 Conclusiones 10
6.2 Trabajo Futuro 10

7 Bibliografía **11**

8 Estructura de tablas de la Aplicación **11**
8.1 Tablas de entrada a la Aplicación 11
8.2 Tablas salida de la Aplicación 11

1. Resumen

Este informe presenta un análisis técnico de un proyecto IoT basado en un emulador ESP32, desarrollado en MicroPython para el curso de Sistemas Operativos. Se enfatiza el estudio de la concurrencia y la interacción entre MicroPython y FreeRTOS, identificando los mecanismos de multitarea, sincronización y gestión de recursos en el contexto de sistemas embebidos.

2. Introducción

2.1. Motivación

La creciente demanda de dispositivos IoT con capacidades de procesamiento en tiempo real ha impulsado la necesidad de comprender cómo los lenguajes de alto nivel como MicroPython pueden aprovechar las capacidades de sistemas operativos de tiempo real como FreeRTOS, especialmente en plataformas con recursos limitados como el ESP32.

2.2. Objetivos

El objetivo principal de este proyecto es analizar y documentar la implementación de un sistema IoT basado en ESP32 utilizando MicroPython sobre FreeRTOS, con énfasis en los mecanismos de concurrencia y gestión de recursos.

2.2.1. Descripción del Problema

Los sistemas IoT modernos requieren gestionar múltiples tareas simultáneas (lectura de sensores, control de actuadores, comunicación en red) con recursos limitados. El desafío consiste en implementar estas funcionalidades manteniendo la fiabilidad, eficiencia y capacidad de respuesta en tiempo real, utilizando herramientas que faciliten el desarrollo rápido como MicroPython.

2.3. Estructura del proyecto

El proyecto se estructura en torno a un sistema de monitoreo y control ambiental que integra sensores de temperatura y humedad, actuadores (LEDs

y ventiladores) y comunicación mediante WiFi y MQTT para la interacción con servicios en la nube.

3. Estado del Arte

3.1. Trabajos Relacionados

La integración de lenguajes interpretados de alto nivel con sistemas operativos de tiempo real ha sido objeto de diversos estudios. Proyectos como CircuitPython y MicroPython han demostrado la viabilidad de ejecutar código Python en microcontroladores, mientras que implementaciones como Zerynth han explorado específicamente la integración con RTOS.

3.2. Conceptos teóricos

3.2.1. Sistemas Operativos de Tiempo Real (RTOS)

Un RTOS es un sistema operativo diseñado para aplicaciones con requisitos temporales estrictos. FreeRTOS, utilizado en este proyecto, es un RTOS ligero y de código abierto que proporciona servicios de multitarea, sincronización y gestión de recursos.

3.2.2. MicroPython

MicroPython es una implementación eficiente del lenguaje Python 3 optimizada para microcontroladores y sistemas embebidos. Incluye un subconjunto del estándar Python junto con módulos específicos para acceso a hardware.

3.2.3. Concurrencia y Multitarea

La concurrencia permite la ejecución aparentemente simultánea de múltiples tareas. En sistemas embebidos, esto se implementa mediante multitarea cooperativa o preemptiva, siendo esta última característica de FreeRTOS.

3.2.4. ESP32

El ESP32 es un microcontrolador de bajo costo y bajo consumo con WiFi y Bluetooth integrados, ampliamente utilizado en aplicaciones IoT. Su arqui-

tectura dual-core permite ejecutar aplicaciones complejas con requisitos de concurrencia.

3.2.5. Protocolos de Comunicación IoT

MQTT (Message Queuing Telemetry Transport) es un protocolo ligero de mensajería para dispositivos IoT, basado en el patrón publicación-suscripción, ideal para entornos con ancho de banda limitado.

3.2.6. Otras herramientas utilizadas

El proyecto utiliza herramientas adicionales como DHT22 para sensores de temperatura y humedad, PWM para control de actuadores, y APIs RESTful para la integración con servicios en la nube.

3.3. Conclusiones del Estado del Arte

La combinación de MicroPython y FreeRTOS representa un equilibrio entre facilidad de desarrollo y capacidades de tiempo real, aunque con limitaciones en el acceso directo a las funcionalidades avanzadas del RTOS desde el código Python.

4. Desarrollo

4.1. Descripción del caso de estudio

4.1.1. Problemática

Los sistemas de monitoreo y control ambiental requieren respuesta en tiempo real a cambios en las condiciones ambientales, mientras mantienen comunicación constante con servicios en la nube. Esto plantea desafíos de concurrencia, gestión de recursos y sincronización.

4.1.2. Objetivo del caso de estudio

Implementar un sistema IoT que demuestre la integración efectiva entre MicroPython y FreeRTOS, permitiendo el control de actuadores, lectura de sensores y comunicación en red de manera concurrente y eficiente.

4.1.3. Evaluación de solución del caso de estudio

La solución se evalúa en términos de respuesta a eventos, eficiencia en el uso de recursos, robustez ante fallos y facilidad de mantenimiento y extensión.

4.2. Metodología de desarrollo de la solución

4.2.1. Especificación de requerimientos

Requerimientos Funcionales:

- Lectura periódica de temperatura y humedad mediante sensor DHT22
- Control de LEDs y ventiladores mediante PWM
- Conexión a red WiFi y broker MQTT
- Envío periódico de datos de telemetría a la nube
- Recepción y procesamiento de comandos remotos
- Control local mediante botones físicos

Requerimientos No Funcionales:

- Respuesta en tiempo real a cambios en condiciones ambientales
- Operación continua sin interrupciones
- Bajo consumo de recursos (memoria, CPU)
- Sincronización temporal precisa
- Recuperación ante fallos de red

4.2.2. Herramientas

MicroPython Entorno de ejecución Python para ESP32, proporcionando acceso a hardware y abstracciones de alto nivel.

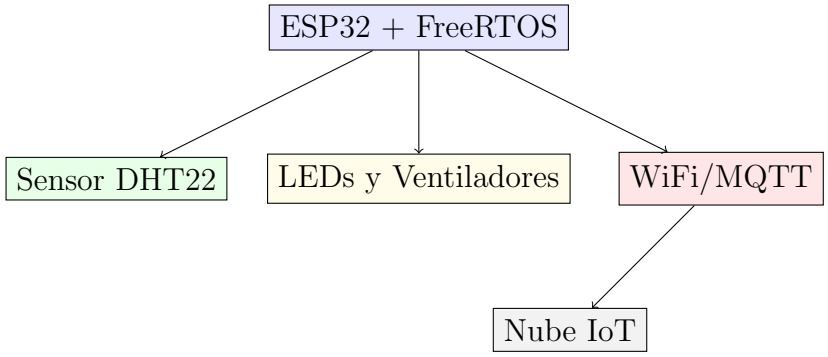
FreeRTOS Sistema operativo subyacente que gestiona la multitarea y recursos del sistema.

ESP32 Plataforma hardware con capacidades WiFi y procesamiento dual-core.

MQTT Protocolo de comunicación para la interacción con servicios en la nube.

4.3. Diseño de arquitectura de desarrollo

El sistema implementa una arquitectura en capas:



4.3.1. Integración e interfaces

La integración entre componentes se realiza mediante:

- APIs de MicroPython para acceso a hardware
- Callbacks para manejo de eventos asíncronos
- Clases de abstracción para dispositivos y comunicación
- Temporizadores para tareas periódicas

4.3.2. Análisis de datos

Los datos de sensores se procesan localmente para control de actuadores y se envían a la nube para análisis histórico y visualización.

4.4. Módulos

4.4.1. Diseño de módulos

Módulo de Control de Dispositivos: Gestiona la interacción con LEDs, ventiladores y otros actuadores mediante PWM.

```
class LED:
    def __init__(self, pin_num, freq=1500):
        self.pin_num = pin_num
        self.freq = freq
        self.pwm = PWM(Pin(pin_num), freq=freq)
        self.is_on = False
        self.percent = 0

    def on(self, percentage = 0):
        if percentage > 0:
            self.percent = percentage
            self.duty = int(percentage / 100 * 1023)
        self.pwm.duty(self.duty)
        self.is_on = True
```

Listing 1: Clase LED para control PWM

Módulo de Comunicación: Gestiona la conexión WiFi y MQTT para interacción con servicios en la nube.

```
class MqttConnector:
    def __init__(self, client_id, broker, user, password):
        self.client_id = client_id
        self.broker = broker
        self.user = user
        self.password = password

    def connect(self, callback):
        self.client = MQTTClient(self.client_id, self.broker)
        self.client.set_callback(callback)
        self.client.connect()

    def publish(self, topic, data):
        self.client.publish(topic, data)
```

Listing 2: Conector MQTT

Módulo de Sensores: Gestiona la lectura de sensores de temperatura y humedad.

Módulo de Gestión de Tareas: Implementa temporizadores y callbacks para simular multitarea en MicroPython.

```
from helpers import DelayedMethod

def push_data():
    # Envia datos de sensores por MQTT
    ...

push_device_data_delay = DelayedMethod(push_data, 10)

while True:
    push_device_data_delay.run()
    mqtt_connector.check_incoming_msg()
    time.sleep(0.1)
```

Listing 3: Ejemplo de tarea concurrente

5. Resultados

5.1. Pruebas y Análisis

El sistema demuestra capacidad para gestionar múltiples tareas concurrentes, manteniendo la respuesta en tiempo real a eventos externos. La abstracción de MicroPython sobre FreeRTOS proporciona un equilibrio entre facilidad de desarrollo y rendimiento.

Reto	Solución Implementada	Apoyo FreeRTOS
Gestión de memoria	Uso de MicroPython con recolección de basura	FreeRTOS gestiona heap y stack por tarea
Latencia crítica	Temporizadores y polling para eventos	Planificador de prioridades y preemptivo
Conmutación de contexto	Bucle principal y callbacks asincrónicos	Context switching eficiente entre tareas

5.1.1. Mejoras de desarrollo

Se identifican posibles mejoras:

- Implementación de uasyncio para mejor manejo de concurrencia en MicroPython
- Optimización de uso de memoria para tareas críticas
- Implementación de mecanismos de recuperación ante fallos

6. Conclusiones y Trabajo Futuro

6.1. Conclusiones

El uso de MicroPython sobre FreeRTOS en ESP32 permite una rápida prototipación y manejo sencillo de concurrencia para aplicaciones IoT. Sin embargo, la abstracción limita el acceso directo a mecanismos avanzados de RTOS, lo que puede ser relevante en aplicaciones de tiempo real estricto. FreeRTOS aporta robustez en multitarea, gestión de recursos y escalabilidad, mientras que MicroPython facilita el desarrollo y la integración de componentes.

6.2. Trabajo Futuro

Trabajo 1: Implementación de uasyncio Migrar el sistema a utilizar la biblioteca uasyncio de MicroPython para mejor gestión de concurrencia sin perder la simplicidad de Python.

Trabajo 2: Optimización de rendimiento Identificar puntos críticos y optimizar el rendimiento mediante módulos nativos en C para operaciones intensivas.

Trabajo 3: Seguridad Implementar mecanismos de seguridad para la comunicación MQTT y la autenticación de dispositivos.

7. Bibliografía

1. MicroPython Documentation. `https://docs.micropython.org/`
2. FreeRTOS Real-Time Operating System. `https://www.freertos.org/`
3. Espressif ESP32 Technical Reference Manual. `https://www.espressif.com/`
4. MQTT Protocol Specification. `https://mqtt.org/`

8. Estructura de tablas de la Aplicación

8.1. Tablas de entrada a la Aplicación

No aplicable en este proyecto al no utilizarse base de datos.

8.2. Tablas salida de la Aplicación

No aplicable en este proyecto al no utilizarse base de datos.