TAREA DE SISTEMASEMBEBIDOS-TEÓRICO

TAREA #5

ESP32 – UART, FreeRTOS y Gestión de Energía

1. OBJETIVO GENERAL

· Desarrollar aplicaciones avanzadas sobre el microcontrolador ESP32 que integren comunicación serial UART, ejecución concurrente de tareas mediante FreeRTOS y técnicas de ahorro de energía, utilizando el entorno de desarrollo PlatformIO en Visual Studio Code, fomentando el diseño eficiente, modular y profesional de sistemas embebidos.

2. OBJETIVOS ESPECÍFICOS

· Implementar comunicación serial utilizando el puerto UART2 del ESP32 bajo el framework ESP-IDF.

· Diseñar aplicaciones multitarea haciendo uso de FreeRTOS, gestionando concurrencia, sincronización y temporización.

· Aplicar modos de ahorro de energía del ESP32, analizando su impacto en el consumo energético.

· Integrar UART y FreeRTOS en un sistema completo y funcional.

· Fortalecer el uso de PlatformIO como entorno profesional de desarrollo para sistemas embebidos.

· Diferenciar entre simulación virtual y pruebas sobre hardware real.

3. INTRODUCCIÓN

En el desarrollo de sistemas embebidos modernos, los microcontroladores no solo deben ejecutar instrucciones básicas, sino también comunicarse eficientemente con otros dispositivos, gestionar múltiples procesos de forma simultánea y optimizar el consumo energético, especialmente en aplicaciones alimentadas por baterías. El ESP32 se ha consolidado como una de las plataformas más completas en este ámbito, ya que integra múltiples periféricos de comunicación, un sistema operativo en tiempo real y diversos mecanismos avanzados de gestión de energía.

Esta tarea práctica tiene como finalidad que el estudiante adquiera experiencia real en el uso de estas capacidades, abordando tres pilares fundamentales del diseño de sistemas embebidos: la comunicación serial mediante UART, la ejecución concurrente de tareas con FreeRTOS y la implementación de modos de ahorro energético. Cada uno de estos temas es esencial para el desarrollo de aplicaciones robustas, escalables y eficientes.

Comunicación Serial UART en Sistemas Embebidos

La comunicación serial es uno de los mecanismos más utilizados en los sistemas embebidos para el intercambio de información entre microcontroladores, sensores, módulos externos y

sistemas de supervisión. El protocolo UART (Universal Asynchronous ReceiverTransmitter) es particularmente popular debido a su simplicidad, confiabilidad y bajo requerimiento de hardware, ya que únicamente necesita dos líneas de comunicación: transmisión (TX) y recepción (RX).

Ilustración 1. Comunicación Serial entre Microcontroladores.

Ilustración 1. Comunicación Serial entre el Microcontrolador y el Ordenador.

A diferencia de otros protocolos como SPI o I2C, UART es asíncrono, lo que significa que no utiliza una señal de reloj compartida. En su lugar, la sincronización se realiza mediante una velocidad de transmisión previamente acordada, conocida como baud rate. Esto hace que UART sea ideal para depuración, configuración de dispositivos y comunicación con terminales o módulos externos como GPS, Bluetooth o módulos de audio.

El ESP32 dispone de tres interfaces UART (UART0, UART1 y UART2), lo que permite separar la comunicación de depuración de la comunicación funcional del sistema. En esta tarea, se hace énfasis en el uso del UART2, obligando al estudiante a configurar explícitamente los pines, los buffers de transmisión y recepción, y el manejo de datos entrantes.

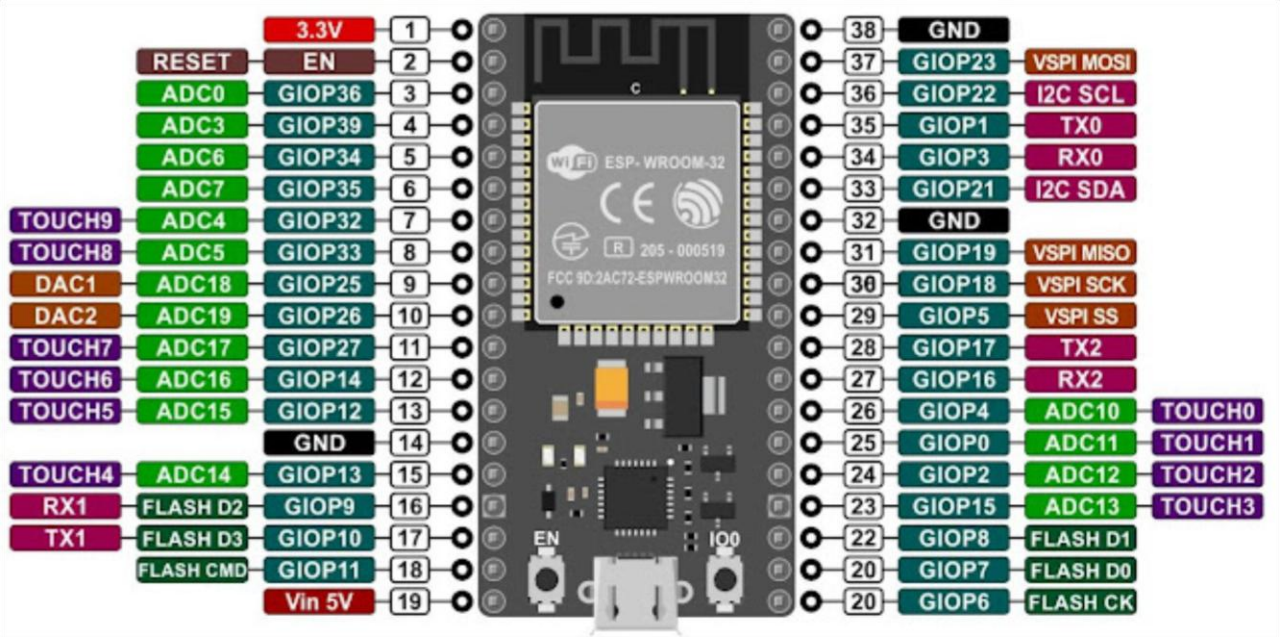


Ilustración 2. Puertos UART de la ESP32.

TX0 >> GPIO1 TX1 >> GPIO10 TX2 >> GPIO17 RX0 >>
GPIO3 RX1 >> GPIO9 RX2 >> GPIO16

Mediante la implementación de un sistema de comandos por UART, el estudiante aprenderá a:

- Interpretar datos recibidos en forma de cadenas o bytes.
- Diseñar protocolos simples de comunicación.
- Gestionar buffers de entrada y salida.
- Evitar bloqueos en la recepción de datos.
- Responder de manera estructurada a solicitudes externas.

Estas habilidades son fundamentales en aplicaciones reales como sistemas de monitoreo, control industrial, dispositivos IoT y sistemas de configuración remota.

FreeRTOS y Ejecución Concurrente de Tareas

En aplicaciones sencillas, un microcontrolador puede operar correctamente utilizando un único ciclo principal (loop). Sin embargo, a medida que la complejidad del sistema aumenta, este enfoque se vuelve limitado, especialmente cuando se requiere ejecutar múltiples procesos de forma simultánea o con diferentes prioridades.

FreeRTOS (Real-Time Operating System) es un sistema operativo en tiempo real integrado de forma nativa en el ESP32, que permite dividir la aplicación en tareas independientes que se ejecutan de manera concurrente. Cada tarea puede tener su propia prioridad, periodo de ejecución y función específica, lo que mejora la organización del código y la capacidad de respuesta del sistema.

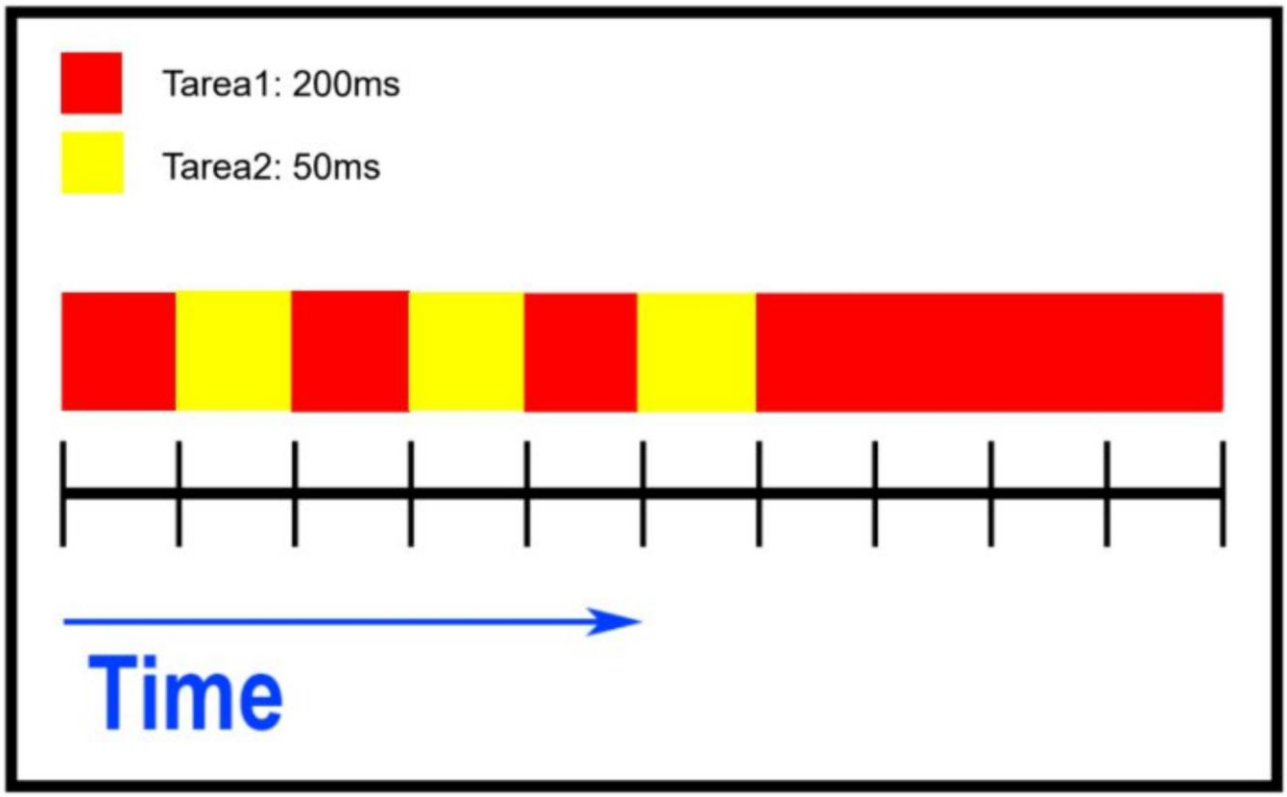


Ilustración 3. Distribución de tiempo entre tareas con FreeRTOS.

El uso de FreeRTOS permite:

- Ejecutar múltiples tareas en paralelo.
- Asignar prioridades según la importancia de cada proceso.
- Evitar bloqueos del sistema.
- Sincronizar tareas mediante semáforos, colas y eventos.
- Aprovechar los dos núcleos del ESP32.

En esta tarea, el estudiante deberá diseñar un sistema multitarea donde distintas funciones se ejecuten simultáneamente, como la lectura de sensores, el control de salidas y la comunicación serial. Esto permitirá comprender conceptos clave como:

- Planificación de tareas (task scheduling).
- Temporización con vTaskDelay().
- Comunicación entre tareas.
- Diferencias entre programación secuencial y concurrente.

El dominio de FreeRTOS es una competencia esencial en el desarrollo profesional de sistemas embebidos, especialmente en aplicaciones críticas como automatización, robótica, sistemas médicos e industriales.

Gestión de Energía y Modos de Ahorro en el ESP32

El consumo energético es un factor crítico en el diseño de sistemas embebidos, particularmente en dispositivos portátiles, sensores inalámbricos y aplicaciones IoT. Un sistema mal optimizado puede agotar rápidamente la batería, reduciendo su autonomía y vida útil.

El ESP32 incorpora múltiples modos de ahorro de energía, que permiten reducir significativamente el consumo cuando el sistema no se encuentra realizando tareas activas. Entre los modos más importantes se encuentran:

- **Moderm Sleep:** En este modo la CPU está operativa y el reloj es configurable. La banda base Wi-Fi/Bluetooth y la radio están desactivadas.
- **Light Sleep:** En este modo la CPU está en pausa. La memoria RTC y los periféricos RTC, así como el coprocesador ULP, están en funcionamiento. Cualquier evento de activación (MAC, host SDIO, temporizador RTC o interrupciones externas) activará el chip.
- **Deep Sleep:** En este modo solo se alimentan la memoria RTC y los periféricos RTC. Los datos de conexión Wi-Fi y Bluetooth se almacenan en la memoria RTC. El coprocesador ULP está operativo.
- **Modo de hibernación:** El oscilador interno de 8 MHz y el coprocesador ULP están desactivados. La memoria de recuperación del RTC está apagada. Solo un temporizador RTC en el reloj lento y ciertos GPIO RTC están activos. El temporizador RTC o los GPIO RTC pueden activar el chip desde el modo de hibernación.

Power mode	Description			Power Consumption
Active (RF working)	Wi-Fi Tx packet			Please refer to Table 5-4 for details.
	Wi-Fi/BT Tx packet			
	Wi-Fi/BT Rx and listening			
Modem-sleep	The CPU is powered up.	240 MHz *	Dual-core chip(s)	30 mA ~ 68 mA
			Single-core chip(s)	N/A
		160 MHz *	Dual-core chip(s)	27 mA ~ 44 mA
			Single-core chip(s)	27 mA ~ 34 mA
		Normal speed: 80 MHz	Dual-core chip(s)	20 mA ~ 31 mA
			Single-core chip(s)	20 mA ~ 25 mA
Light-sleep	-			0.8 mA
Deep-sleep	The ULP coprocessor is powered up.			150 μA
	ULP sensor-monitored pattern			100 μA @1% duty
	RTC timer + RTC memory			10 μA
Hibernation	RTC timer only			5 μA
Power off	CHIP_PU is set to low level, the chip is powered down.			1 μA

Ilustración 4. Tabladeconsumo de energía por modos de alimentación de la ESP32.

El uso adecuado de estos modos permite diseñar sistemas que alternan entre periodos de actividad y reposo, maximizando la eficiencia energética sin comprometer la funcionalidad.

En esta tarea, el estudiante deberá implementar al menos un modo de ahorro energético y analizar:

- Cuándo y por qué el sistema debe entrar en reposo.
- Qué eventos provocan el despertar del microcontrolador.
- Cómo se comporta el sistema antes y después del modo de bajo consumo.
- Las diferencias entre simulación y pruebas reales en hardware.

Este ejercicio refuerza la importancia de probar los sistemas embebidos en condiciones reales, ya que el comportamiento energético no puede evaluarse completamente en entornos de simulación.

Los sistemas embebidos modernos rara vez utilizan estos conceptos de manera aislada; por el contrario, deben combinarse para lograr soluciones eficientes, escalables y confiables.

A través de esta tarea, el estudiante desarrollará no solo habilidades técnicas, sino también criterios de diseño, estructuración del código y toma de decisiones, preparándolo para enfrentar proyectos más complejos dentro del ámbito de los sistemas embebidos.

4. TAREA PRÁCTICA

La tarea consta de cuatro ejercicios obligatorios, los cuales deberán ser desarrollados en PlatformIO (Visual Studio Code). Cada ejercicio deberá contar con su propio proyecto y repositorio, correctamente documentado.

EJERCICIO 1 – Comunicación Serial Avanzada con UART2 (ESP-IDF)

Tema principal: Comunicación UART
Framework: ESP-IDF (PlatformIO)
Simulación: Permitida en Wokwi (VSC) + Serial Monitor Hardware
real: Opcional

Descripción del ejercicio

Desarrollar una aplicación en ESP32 que utilice el puerto UART2 para implementar un sistema de comandos seriales.

El sistema deberá:

- Inicializar correctamente el UART2 (baud rate configurable).
- Recibir comandos de texto desde un terminal serial externo.
- Interpretar los comandos recibidos.
- Ejecutar acciones internas según el comando.
- Enviar respuestas estructuradas por el mismo puerto UART.

Ejemplos de comandos obligatorios

- **status**: devuelve el estado actual del sistema.
- **led on / led off**: controla un LED virtual o físico.
- **info**: muestra información del sistema (baud rate, puerto, contador de comandos).
- **reset**: reinicia variables internas (no el microcontrolador).

Requisitos técnicos

- Uso exclusivo de UART2.
- Implementación con drivers UART nativos del ESP-IDF.
- Manejo de buffers y lectura no bloqueante.
- Código estructurado en funciones.

EJERCICIO 2 – Sistema Multitarea con FreeRTOS

Tema principal: FreeRTOS
Framework: Arduino o ESP-IDF (PlatformIO)
Simulación: Permitida en Wokwi Hardware
real: Opcional

Descripción del ejercicio

Diseñar un sistema multitarea que ejecute al menos tres tareas concurrentes, utilizando FreeRTOS**.**

Ejemplo de tareas sugeridas:

- Tarea 1: Lectura periódica de un sensor virtual o temporizador.
- Tarea 2: Control de LED o salida digital con diferente frecuencia.
- Tarea 3: Envío periódico de información al monitor serial.

Requisitos técnicos

- Uso de xTaskCreate() o xTaskCreatePinnedToCore().
- Configuración de prioridades diferentes para las tareas.
- Uso de vTaskDelay() para temporización.
- Justificación del uso de FreeRTOS frente a un loop tradicional.
- El sistema no debe bloquearse.

EJERCICIO 3 – Ahorro de Energía en el ESP32

Tema principal: Gestión de energía
Framework: Arduino o ESP-IDF (PlatformIO)
Simulación: No permitida
Hardware real: Obligatorio

Descripción del ejercicio

Implementar un sistema que utilice uno o más modos de ahorro de energía del ESP32, tales como:

- Modem Sleep
- Light Sleep
- Deep Sleep
- Hibernation

El sistema deberá:

- Ejecutar una tarea activa durante un tiempo determinado.
- Entrar en modo de bajo consumo automáticamente.
- Despertar mediante un evento (temporizador o pin externo).
- Indicar mediante LED o mensajes seriales los estados del sistema.

Requisitos técnicos

- Configuración explícita del modo de ahorro utilizado.
- Uso de temporizadores de despertar (esp_sleep_enable_timer_wakeup()).
- Evidencia de prueba en hardware real.
- Análisis breve del comportamiento del sistema.

EJERCICIO 4 – Sistema Integrado UART + FreeRTOS

Tema principal: Integración de sistemas
Framework: Arduino o ESP-IDF (PlatformIO)
Simulación: Permitida en Wokwi + Serial Monitor
Hardware real: Opcional

Descripción del ejercicio

Desarrollar un sistema completo que integre:

- Comunicación UART (comandos).
- Múltiples tareas FreeRTOS.

El sistema deberá:

- recibir comandos por UART.
- Procesarlos en una tarea dedicada.
- Ejecutar acciones concurrentes en otras tareas.
- Utilizar colas (Queues) o semáforos para comunicación entre tareas.

Ejemplo de funcionamiento

- Una tarea recibe comandos UART.
- Otra tarea controla salidas o estados.
- Otra tarea reporta el estado del sistema periódicamente.

Requisitos técnicos

- Uso obligatorio de FreeRTOS + UART.
- Comunicación entre tareas mediante mecanismos de FreeRTOS.
- Código modular y documentado.

- Funcionamiento estable y verificable.

Entregables:

Documento PDF con:

- Descripción del ejercicio.
- Explicación del funcionamiento del sistema.
- Diagrama de flujo o arquitectura.
- Capturas de simulación (si aplica).
- Evidencia de pruebas en hardware real (Ejercicio 3 obligatorio).
- Enlace al repositorio GitHub con el código.

Repositorio GitHub:

- Código fuente organizado.
- Archivo README.md explicativo.
- Instrucciones de compilación y ejecución.

VÍNCULO GITHUB: EstraRobert/TAREA5-Sistemas-Embebidos: Se encuentra todo lo solicitado en la Tarea4

Video en YouTube mostrando:

- Demostración del funcionamiento.
- Explicación general del código.

VÍNCULO VIDEO: Dentro del GitHub se encontrarán los videos por cada actividad junto a su código.

5. CONCLUSIONES Y RECOMENDACIONES

Adjunte Tres conclusiones relacionadas con el uso de UART, FreeRTOS y ahorro de energía. Dos recomendaciones sobre diseño de sistemas embebidos eficientes.

Ejercicio 1: Sistema de Comandos UART2

Asignatura: Sistemas Embebidos (ESP32)

Framework: ESP-IDF

Entorno: PlatformIO + Visual Studio Code

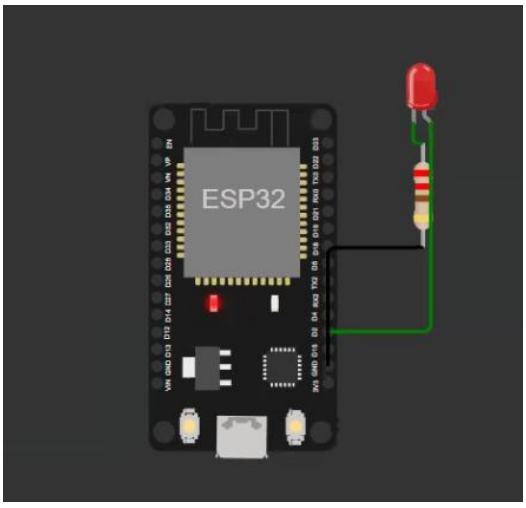
Descripción

Se ha desarrollado un **sistema avanzado de gestión serial** utilizando el periférico **UART2** del ESP32. La aplicación emplea los controladores nativos del framework **ESP-IDF** para capturar cadenas de texto desde un terminal externo, procesarlas mediante un analizador de comandos y ejecutar acciones específicas en el hardware, garantizando una comunicación fluida y eficiente.

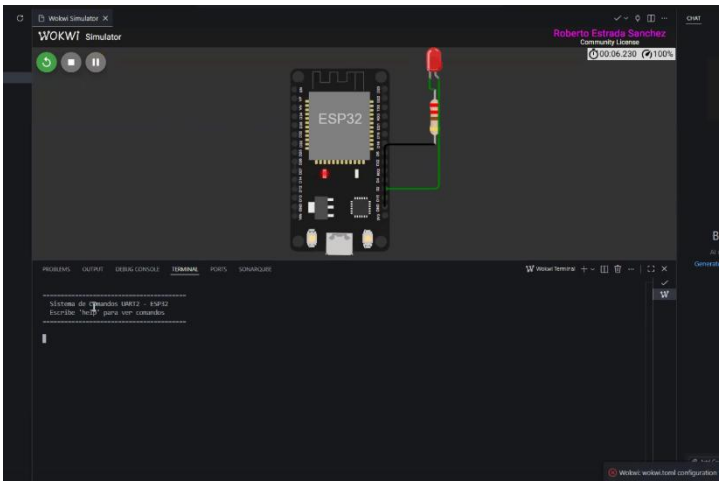
Objetivos Cumplidos

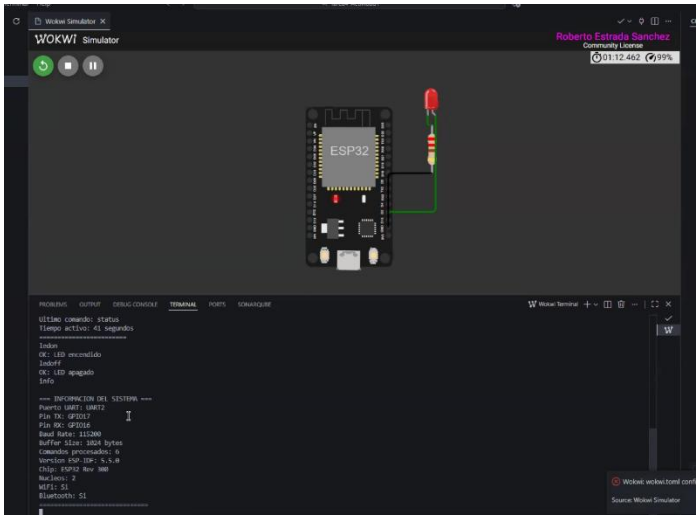
- **Configuración Precisa:** Inicialización del periférico UART2 con parámetros de comunicación industrial.
- **Procesamiento en Tiempo Real:** Recepción e interpretación de comandos de texto de forma asíncrona.
- **Interacción Activa:** Implementación de un sistema de respuestas estructuradas para retroalimentación al usuario.
- **Eficiencia de Memoria:** Uso de buffers circulares y lectura **no bloqueante** para optimizar el ciclo de CPU.
- **Arquitectura Profesional:** Código modular, escalable y debidamente documentado.

Diagrama Esquemático:



Simulación en Wokwi:





Código:

```c
//Actividad1

#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/uart.h"
#include "driver/gpio.h"
#include "esp_log.h"
#include "esp_system.h"
#include "esp_chip_info.h"

// DEFINICIONES Y CONFIGURACIÓN

#define UART_NUM            UART_NUM_2
#define UART_TX_PIN         GPIO_NUM_17
#define UART_RX_PIN         GPIO_NUM_16
#define UART_BAUD_RATE      115200
#define UART_BUF_SIZE       1024
#define RX_BUFFER_SIZE      256
#define LED_GPIO            GPIO_NUM_2

static const char *TAG = "UART_CMD";

// VARIABLES GLOBALES

typedef struct {
    bool led_state;
    uint32_t command_counter;
    uint32_t baud_rate;
    char last_command[64];
    uint32_t uptime_seconds;
} system_status_t;

static system_status_t sys_status = {
    .led_state = false,
    .command_counter = 0,
    .baud_rate = UART_BAUD_RATE,
    .last_command = "none",
    .uptime_seconds = 0
};
```

```c
// PROTOTIPOS DE FUNCIONES

void uart2_init(void);
void led_init(void);
void process_command(const char *cmd);
void send_response(const char *response);
void cmd_status(void);
void cmd_led_on(void);
void cmd_led_off(void);
void cmd_info(void);
void cmd_reset(void);
void cmd_help(void);
void uptime_task(void *pvParameters);
void uart_rx_task(void *pvParameters);

// INICIALIZACIÓN DE PERIFÉRICOS

void uart2_init(void) {
    uart_config_t uart_config;
    uart_config.baud_rate = UART_BAUD_RATE;
    uart_config.data_bits = UART_DATA_8_BITS;
    uart_config.parity = UART_PARITY_DISABLE;
    uart_config.stop_bits = UART_STOP_BITS_1;
    uart_config.flow_ctrl = UART_HW_FLOWCTRL_DISABLE;
    uart_config.rx_flow_ctrl_thresh = 122;
    uart_config.source_clk = UART_SCLK_DEFAULT;

    // Configurar parámetros UART
    ESP_ERROR_CHECK(uart_param_config(UART_NUM, &uart_config));

    // Configurar pines UART
    ESP_ERROR_CHECK(uart_set_pin(UART_NUM, UART_TX_PIN, UART_RX_PIN, UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE));

    // Instalar driver UART
    ESP_ERROR_CHECK(uart_driver_install(UART_NUM, UART_BUF_SIZE * 2, UART_BUF_SIZE * 2, 0, NULL, 0));

    ESP_LOGI(TAG, "UART2 inicializado: TX=GPIO%d, RX=GPIO%d, Baud=%d", UART_TX_PIN, UART_RX_PIN, UART_BAUD_RATE);
}
```

```c
void led_init(void) {
    gpio_reset_pin(LED_GPIO);
    gpio_set_direction(LED_GPIO, GPIO_MODE_OUTPUT);
    gpio_set_level(LED_GPIO, 0);
    ESP_LOGI(TAG, "LED inicializado en GPIO%d", LED_GPIO);
}

// FUNCIONES DE COMUNICACIÓN

void send_response(const char *response) {
    uart_write_bytes(UART_NUM, response, strlen(response));
    uart_write_bytes(UART_NUM, "\r\n", 2);
}

// PROCESAMIENTO DE COMANDOS

void process_command(const char *cmd) {
    while (*cmd == ' ' || *cmd == '\t') cmd++;

    strncpy(sys_status.last_command, cmd, sizeof(sys_status.last_command) - 1);
    sys_status.command_counter++;

    ESP_LOGI(TAG, "Comando recibido: '%s'", cmd);

    if (strcmp(cmd, "status") == 0) {
        cmd_status();
    }
    else if (strcmp(cmd, "ledon") == 0 || strcmp(cmd, "led on") == 0) {
        cmd_led_on();
    }
    else if (strcmp(cmd, "ledoff") == 0 || strcmp(cmd, "led off") == 0) {
        cmd_led_off();
    }
    else if (strcmp(cmd, "info") == 0) {
        cmd_info();
    }
    else if (strcmp(cmd, "reset") == 0) {
        cmd_reset();
    }
    else if (strcmp(cmd, "help") == 0) {
        cmd_help();
    }
    else {
        char response[128];
        snprintf(response, sizeof(response),
                 "ERROR: Comando desconocido: '%s'. Escribe 'help' para ver comandos.", cmd);
        send_response(response);
    }
```

```c
// IMPLEMENTACIÓN DE COMANDOS

void cmd_status(void) {
    char response[256];
    snprintf(response, sizeof(response),
             "\r\n=== ESTADO DEL SISTEMA ===\r\n"
             "LED: %s\r\n"
             "Comandos ejecutados: %lu\r\n"
             "Ultimo comando: %s\r\n"
             "Tiempo activo: %lu segundos\r\n"
             "========================",
             sys_status.led_state ? "ENCENDIDO" : "APAGADO",
             (unsigned long)sys_status.command_counter,
             sys_status.last_command,
             (unsigned long)sys_status.uptime_seconds);
    send_response(response);
}

void cmd_led_on(void) {
    sys_status.led_state = true;
    gpio_set_level(LED_GPIO, 1);
    send_response("OK: LED encendido");
    ESP_LOGI(TAG, "LED encendido");
}

void cmd_led_off(void) {
    sys_status.led_state = false;
    gpio_set_level(LED_GPIO, 0);
    send_response("OK: LED apagado");
    ESP_LOGI(TAG, "LED apagado");
}
```

```c
void cmd_info(void) {
    esp_chip_info_t chip_info;
    esp_chip_info(&chip_info);

    char response[512];
    snprintf(response, sizeof(response),
             "\r\n=== INFORMACION DEL SISTEMA ===\r\n"
             "Puerto UART: UART2\r\n"
             "Pin TX: GPIO%d\r\n"
             "Pin RX: GPIO%d\r\n"
             "Baud Rate: %lu\r\n"
             "Buffer Size: %d bytes\r\n"
             "Comandos procesados: %lu\r\n"
             "Version ESP-IDF: %s\r\n"
             "Chip: ESP32 Rev %d\r\n"
             "Nucleos: %d\r\n"
             "WiFi: %s\r\n"
             "Bluetooth: %s\r\n"
             "============================",
             UART_TX_PIN, UART_RX_PIN,
             (unsigned long)sys_status.baud_rate,
             UART_BUF_SIZE,
             (unsigned long)sys_status.command_counter,
             esp_get_idf_version(),
             chip_info.revision,
             chip_info.cores,
             (chip_info.features & CHIP_FEATURE_WIFI_BGN) ? "Si" : "No",
             (chip_info.features & CHIP_FEATURE_BT) ? "Si" : "No");
    send_response(response);
}


void cmd_reset(void) {
    sys_status.command_counter = 0;
    sys_status.uptime_seconds = 0;
    strcpy(sys_status.last_command, "reset");
    sys_status.led_state = false;
    gpio_set_level(LED_GPIO, 0);

    send_response("OK: Variables internas reiniciadas");
    ESP_LOGI(TAG, "Variables reiniciadas");
}
```

```c
void cmd_help(void) {
    const char *help_text =
        "\r\n=== COMANDOS DISPONIBLES ===\r\n"
        "status     - Muestra el estado actual del sistema\r\n"
        "ledon      - Enciende el LED\r\n"
        "ledoff     - Apaga el LED\r\n"
        "info       - Muestra informacion del sistema\r\n"
        "reset      - Reinicia variables internas\r\n"
        "help       - Muestra esta ayuda\r\n"
        "============================";
    send_response(help_text);
}

// TAREAS DE FREERTOS


void uptime_task(void *pvParameters) {
    while (1) {
        vTaskDelay(pdMS_TO_TICKS(1000));
        sys_status.uptime_seconds++;
    }
}
```

```c
void uart_rx_task(void *pvParameters) {
    uint8_t data[RX_BUFFER_SIZE];
    char command_buffer[RX_BUFFER_SIZE];
    int cmd_index = 0;

    // Mensaje de bienvenida
    vTaskDelay(pdMS_TO_TICKS(100));
    send_response("\r\n========================================");
    send_response("  Sistema de Comandos UART2 - ESP32");
    send_response("  Escribe 'help' para ver comandos");
    send_response("========================================\r\n");

    while (1) {
        // Leer datos del UART (no bloqueante)
        int len = uart_read_bytes(UART_NUM, data, RX_BUFFER_SIZE - 1, pdMS_TO_TICKS(100));

        if (len > 0) {
            for (int i = 0; i < len; i++) {
                char c = (char)data[i];

                // Echo del carácter (opcional)
                uart_write_bytes(UART_NUM, (const char*)&c, 1);

                // Procesar carácter
                if (c == '\r' || c == '\n') {
                    if (cmd_index > 0) {
                        command_buffer[cmd_index] = '\0';
                        uart_write_bytes(UART_NUM, "\r\n", 2);
                        process_command(command_buffer);
                        cmd_index = 0;
                    }
                }
                else if (c == 127 || c == 8) { // Backspace
                    if (cmd_index > 0) {
                        cmd_index--;
                        uart_write_bytes(UART_NUM, "\b \b", 3);
                    }
                }
                else if (cmd_index < RX_BUFFER_SIZE - 1) {
                    command_buffer[cmd_index++] = c;
                }
            }
        }
    }
}
```

```c
// FUNCIÓN PRINCIPAL

extern "C" void app_main(void) {
    // Inicializar periféricos
    led_init();
    uart2_init();

    ESP_LOGI(TAG, "Sistema iniciado");

    // Crear tareas
    xTaskCreate(uart_rx_task, "uart_rx", 4096, NULL, 5, NULL);
    xTaskCreate(uptime_task, "uptime", 2048, NULL, 3, NULL);

    ESP_LOGI(TAG, "Tareas creadas. Sistema listo.");
}
```

Ejercicio 2 - Puente de Comunicación UART2 y Control de Periféricos

Tarea de Sistemas Embebidos - ESP32 Framework: ESP-IDF

Plataforma: PlatformIO + Visual Studio Code

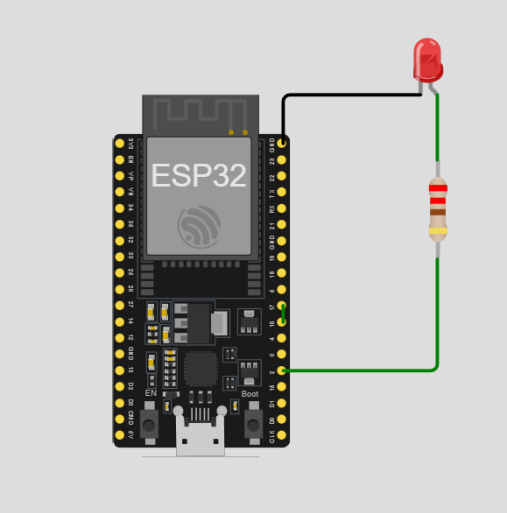
Descripción

Este ejercicio implementa un sistema de comunicación bidireccional que actúa como un puente (bridge) entre la consola (UART0) y el puerto UART2. El sistema utiliza **FreeRTOS** para gestionar dos tareas concurrentes: una que simula la lectura de un sensor y otra que monitorea el flujo de datos seriales. El código permite el control de un LED físico mediante comandos recibidos específicamente a través del pin de recepción (RX2), validando la integridad del enlace físico (loopback o comunicación externa).

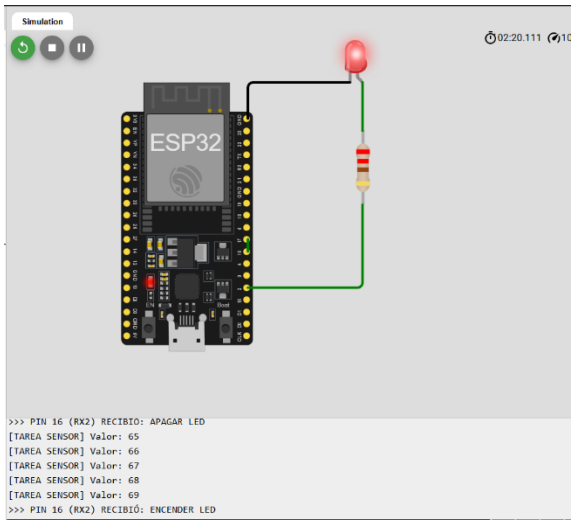
Objetivos Cumplidos

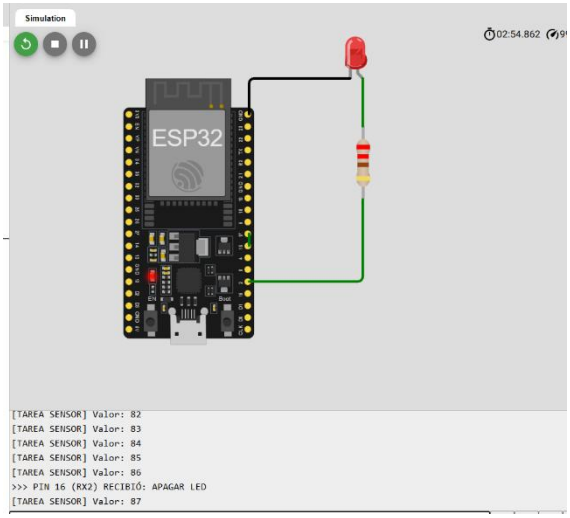
- **Gestión Multitarea:** Implementación de tareas independientes mediante FreeRTOS (tarea_sensor y tarea_puente_uart).
- **Puente Serial (UART Bridge):** Retransmisión de datos desde el terminal (UART0) hacia el puerto UART2.
- **Control por Hardware:** Procesamiento de señales entrantes en el pin 16 (RX2) para conmutar el estado del GPIO 2.
- **Lectura No Bloqueante:** Uso de uart_read_bytes con tiempos de espera (*timeouts*) optimizados para no detener la ejecución del sistema.
- **Configuración Nativa:** Uso exclusivo de los controladores oficiales de ESP-IDF para la gestión de UART y GPIO.

Diagrama Esquemático:



Simulación en Wokwi:





Código:

```c
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/uart.h"
#include "driver/gpio.h"

#define LED_PIN GPIO_NUM_2
#define UART_FISICA UART_NUM_2 // Usaremos los pines 17 y 16

void tarea_sensor(void *pvParameters) {
    int contador = 0;
    while (1) {
        printf("[TAREA SENSOR] Valor: %d\n", contador++);
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

void tarea_puente_uart(void *pvParameters) {
    uint8_t teclado_data[128];
    uint8_t rx_data[128];

    while (1) {
        // 1. Leemos del teclado (UART0)
        int len_teclado = uart_read_bytes(UART_NUM_0, teclado_data, 127, pdMS_TO_TICKS(20));
        if (len_teclado > 0) {
            // 2. RETRANSMITIMOS por el pin 17 (TX2) para que viaje por el cable
            uart_write_bytes(UART_FISICA, (const char*)teclado_data, len_teclado);
        }
```

```
30
31          // 3. LEEMOS del pin 16 (RX2). Solo si el cable está conectado, esto funcionará.
32          int len_rx = uart_read_bytes(UART_FISICA, rx_data, 127, pdMS_TO_TICKS(20));
33          if (len_rx > 0) {
34              rx_data[len_rx] = '\0';
35              if (strstr((char*)rx_data, "on")) {
36                  gpio_set_level(LED_PIN, 1);
37                  printf(">>> PIN 16 (RX2) RECIBIÓ: ENCENDER LED\n");
38              } else if (strstr((char*)rx_data, "off")) {
39                  gpio_set_level(LED_PIN, 0);
40                  printf(">>> PIN 16 (RX2) RECIBIÓ: APAGAR LED\n");
41              }
42          }
43          vTaskDelay(pdMS_TO_TICKS(50));
44      }
45  }
46
47  void app_main(void) {
48      gpio_reset_pin(LED_PIN);
49      gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);
50
51      // Configurar UART0 (Consola)
52      uart_config_t uart_cfg0 = {.baud_rate = 115200,
53       .data_bits = UART_DATA_8_BITS,
54       .parity = UART_PARITY_DISABLE,
55       .stop_bits = UART_STOP_BITS_1,
56       .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
57       .source_clk = UART_SCLK_DEFAULT};
58      uart_param_config(UART_NUM_0, &uart_cfg0);
59      uart_driver_install(UART_NUM_0, 1024, 0, 0, NULL, 0);
60
```

```
47  void app_main(void) {
48      gpio_reset_pin(LED_PIN);
49      gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);
50
51      // Configurar UART0 (Consola)
52      uart_config_t uart_cfg0 = {.baud_rate = 115200,
53       .data_bits = UART_DATA_8_BITS,
54       .parity = UART_PARITY_DISABLE,
55       .stop_bits = UART_STOP_BITS_1,
56       .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
57       .source_clk = UART_SCLK_DEFAULT};
58      uart_param_config(UART_NUM_0, &uart_cfg0);
59      uart_driver_install(UART_NUM_0, 1024, 0, 0, NULL, 0);
60
61      // Configurar UART2 (Pines 17 y 16)
62      uart_config_t uart_cfg2 = {.baud_rate = 115200,
63      .data_bits = UART_DATA_8_BITS,
64      .parity = UART_PARITY_DISABLE,
65      .stop_bits = UART_STOP_BITS_1,
66      .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
67      .source_clk = UART_SCLK_DEFAULT};
68      uart_param_config(UART_FISICA, &uart_cfg2);
69      uart_set_pin(UART_FISICA, 17, 16, -1, -1);
70      uart_driver_install(UART_FISICA, 1024, 0, 0, NULL, 0);
71
72      xTaskCreate(tarea_sensor, "Sensor", 2048, NULL, 1, NULL);
73      xTaskCreate(tarea_puente_uart, "Puente_UART", 2048, NULL, 2, NULL);
74  }
75
76  void setup() { app_main(); }
77  void loop() {}
```

Ejercicio 3: Ahorro de Energía en el ESP32 (Deep Sleep)

Tema: Gestión de Energía y Modos de Bajo Consumo

Framework: Arduino / ESP-IDF (PlatformIO)

Plataforma: Hardware Real (ESP32 DevKit V1)

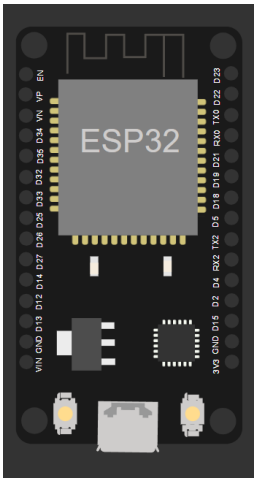
Descripción

Este ejercicio demuestra la implementación del modo Deep Sleep, el estado de mayor ahorro de energía del ESP32 (excluyendo la hibernación). El sistema ejecuta una tarea activa (encendido de LED y envío de logs), configura un temporizador de despertar mediante el coprocesador de ultra bajo consumo (ULP) y entra en sueño profundo, apagando la CPU, la memoria RAM y los periféricos digitales para minimizar el consumo de corriente.

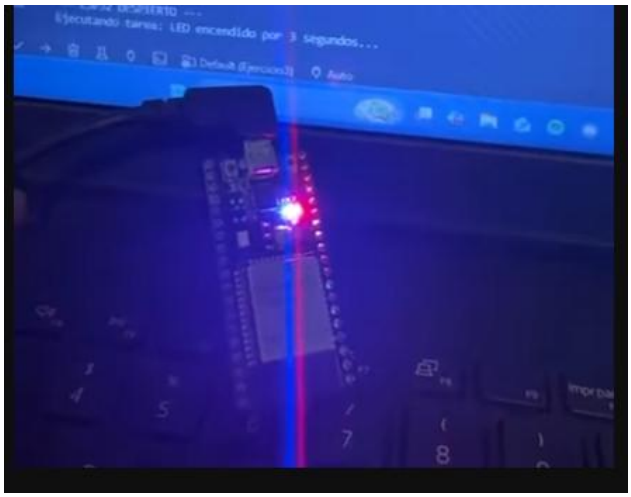
Objetivos Cumplidos

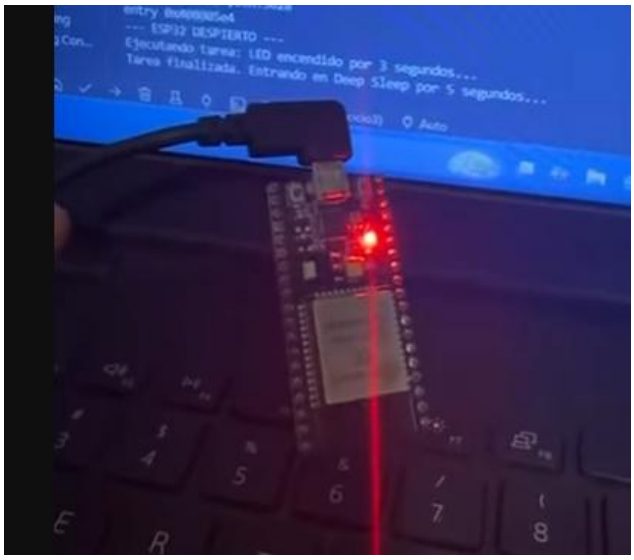
- **Gestión de Ciclos de Vida: Implementación del ciclo *Wakeup -> Active -> Deep Sleep*.**
- **Configuración de Despertar: Uso exitoso de la API esp_sleep_enable_timer_wakeup() para definir despertares automáticos.**
- **Indicadores de Estado: Uso de retroalimentación visual (LED en GPIO 2) y serial para identificar la transición entre modos.**
- **Optimización de Periféricos: Uso de Serial.flush() para asegurar la integridad de los datos antes de apagar el controlador UART.**
- **Hardware Real: Verificación del reinicio del sistema (el código en *Deep Sleep* reinicia el setup() al despertar).**

Diagrama Esquemático:



Implementado en la vida real:





Código:

```
#include <Arduino.h>
#define SEGUNDOS_A_MICROSEGUNDOS 1000000ULL
#define TIEMPO_SUENO  5

void setup() {
  Serial.begin(115200);
  pinMode(2, OUTPUT); 2

  Serial.println("--- ESP32 DESPIERTO ---");
  Serial.println("Ejecutando tarea: LED encendido por 3 segundos...");
  digitalWrite(2, HIGH);
  delay(3000);
  digitalWrite(2, LOW);
  esp_sleep_enable_timer_wakeup(TIEMPO_SUENO * SEGUNDOS_A_MICROSEGUNDOS);

  Serial.println("Tarea finalizada. Entrando en Deep Sleep por 5 segundos...");
  Serial.flush();
  esp_deep_sleep_start();
}
void loop() {}
```

Ejercicio 4: Sistema Integrado UART + FreeRTOS

Tema: Integración de Sistemas y Comunicación entre Tareas

Framework: ESP-IDF

Herramientas: Queues (Colas), Multitasking, Gestión de Energía

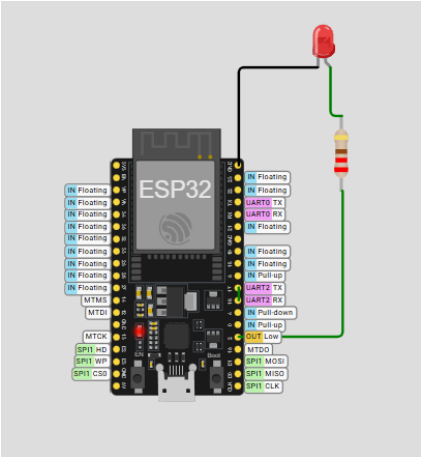
Descripción

Este sistema representa una solución integral de ingeniería embebida que implementa un modelo **Productor-Consumidor** mediante el uso de **FreeRTOS Queues**. El firmware gestiona tres tareas concurrentes con diferentes niveles de prioridad, permitiendo que la recepción de comandos, el control de actuadores (LED) y el reporte de telemetría funcionen de manera independiente y sincronizada sin bloquear el procesador.

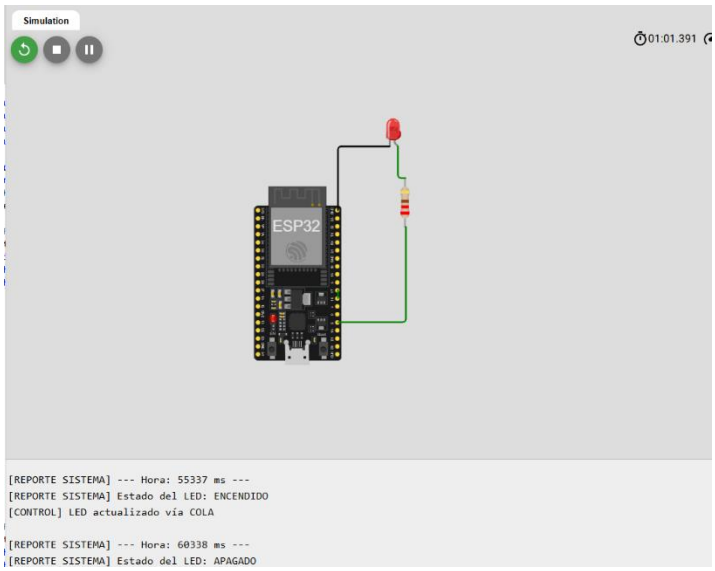
Objetivos Cumplidos

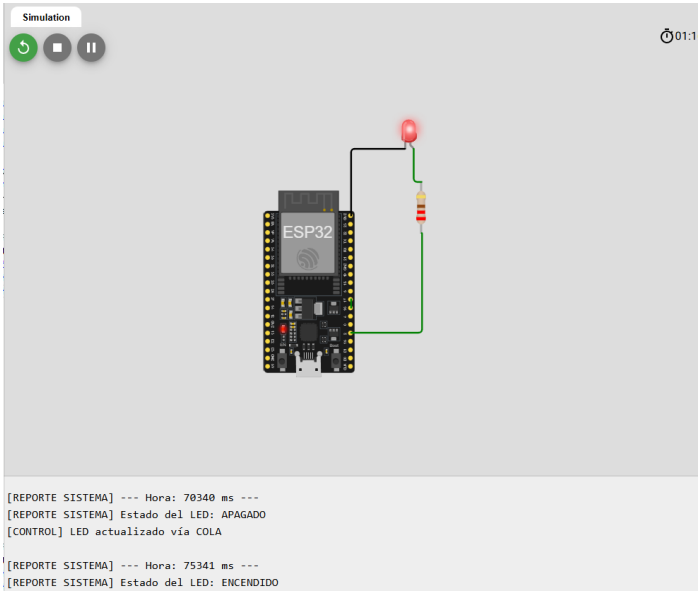
- **Comunicación Inter-Tareas:** Uso de una cola (QueueHandle_t) para enviar datos de control de forma segura entre hilos de ejecución.
- **Procesamiento Concurrente:** Ejecución simultánea de la recepción serial, el control de periféricos y un reportero de estado.
- **Desacoplamiento de Código:** Separación lógica entre la capa de comunicación (UART) y la capa de aplicación (Control del LED).
- **Monitoreo en Tiempo Real:** Implementación de una tarea de diagnóstico que reporta el *uptime* y el estado del sistema cada 5 segundos.
- **Sincronización Eficiente:** La tarea consumidora utiliza el estado portMAX_DELAY, lo que significa que no consume ciclos de CPU hasta que llega un dato a la cola.

Diagrama Esquemático:



Simulación en Wokwi:





Código:

```c
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/uart.h"
#include "driver/gpio.h"

#define LED_PIN GPIO_NUM_2
#define UART_FISICA UART_NUM_2
QueueHandle_t xCola;
bool estado_led = false; // Variable para que el reportero sepa el estado

// TAREA 1: RECIBE COMANDOS UART (Pines 17 -> 16)
void tarea_productora_pines(void *pvParameters) {
    uint8_t teclado[128], rx_fisico[128];
    char cmd_cola;
    while (1) {
        int len_t = uart_read_bytes(UART_NUM_0, teclado, 127, pdMS_TO_TICKS(20));
        if (len_t > 0) uart_write_bytes(UART_FISICA, (const char*)teclado, len_t);

        int len_r = uart_read_bytes(UART_FISICA, rx_fisico, 127, pdMS_TO_TICKS(20));
        if (len_r > 0) {
            rx_fisico[len_r] = '\0';
            if (strstr((char*)rx_fisico, "on")) {
                cmd_cola = '1';
                xQueueSend(xCola, &cmd_cola, 0);
            } else if (strstr((char*)rx_fisico, "off")) {
                cmd_cola = '0';
                xQueueSend(xCola, &cmd_cola, 0);
            }
        }
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}
```

```c

// TAREA 2: CONTROLA SALIDAS (LED)
void tarea_consumidora_led(void *pvParameters) {
    char recibido;
    while (1) {
        if (xQueueReceive(xCola, &recibido, portMAX_DELAY)) {
            estado_led = (recibido == '1');
            gpio_set_level(LED_PIN, estado_led ? 1 : 0);
            printf("[CONTROL] LED actualizado vía COLA\n");
        }
    }
}

// TAREA 3: REPORTA ESTADO PERIÓDICAMENTE (Cada 5 segundos)
void tarea_reportero(void *pvParameters) {
    while (1) {
        printf("\n[REPORTE SISTEMA] --- Hora: %lld ms ---\n", esp_timer_get_time() / 1000);
        printf("[REPORTE SISTEMA] Estado del LED: %s\n", estado_led ? "ENCENDIDO" : "APAGADO");
        printf("[REPORTE SISTEMA] Memoria libre: %d bytes\n\n", esp_get_free_heap_size());
        vTaskDelay(pdMS_TO_TICKS(5000)); // Reporte cada 5 segundos
    }
}
```

```
59  void app_main(void) {
60      gpio_reset_pin(LED_PIN);
61      gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);
62
63      uart_config_t cfg = {.baud_rate = 115200,
64      .data_bits = UART_DATA_8_BITS,
65      .parity = UART_PARITY_DISABLE,
66      .stop_bits = UART_STOP_BITS_1,
67      .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
68      .source_clk = UART_SCLK_DEFAULT};
69      uart_param_config(UART_NUM_0, &cfg);
70      uart_driver_install(UART_NUM_0, 1024, 0, 0, NULL, 0);
71      uart_param_config(UART_FISICA, &cfg);
72      uart_set_pin(UART_FISICA, 17, 16, -1, -1);
73      uart_driver_install(UART_FISICA, 1024, 0, 0, NULL, 0);
74
75      xCola = xQueueCreate(5, sizeof(char));
76
77      // Lanzamos las 3 tareas solicitadas
78      xTaskCreate(tarea_productora_pines, "ReceptorUART", 2048, NULL, 3, NULL);
79      xTaskCreate(tarea_consumidora_led, "ControlSalidas", 2048, NULL, 2, NULL);
80      xTaskCreate(tarea_reportero, "Reportero", 2048, NULL, 1, NULL);
81  }
82
83  void setup() { app_main(); }
84  void loop() {}
```

Conclusiones y Recomendaciones

Conclusiones:

Sobre la Comunicación UART: La implementación de la UART2 mediante drivers nativos de ESP-IDF permite una gestión de datos más robusta y eficiente que los métodos tradicionales. Al utilizar buffers circulares y lecturas no bloqueantes, el sistema puede procesar comandos externos sin interrumpir el flujo principal del programa, lo que es crítico en aplicaciones industriales de mecatrónica.

Sobre FreeRTOS y la Gestión de Tareas: El uso de Queues (Colas) para la comunicación entre tareas demuestra ser superior al uso de variables globales. Este mecanismo garantiza la integridad de los datos y permite una sincronización precisa entre una tarea productora (UART) y una consumidora (Control de LED), optimizando el uso del procesador al mantener las tareas en estado *Blocked* hasta que sea necesario actuar.

Sobre el Ahorro de Energía: La arquitectura multitarea de FreeRTOS es fundamental para la eficiencia energética. Al emplear vTaskDelay() en lugar de retardos bloqueantes (delay), se permite que la tarea IDLE del sistema operativo tome el control, facilitando que el ESP32 entre en modos de bajo consumo de manera automática cuando no hay procesamiento activo.

Recomendaciones para Diseño Eficiente:

Implementación de Modos de Sueño (Sleep Modes): Para proyectos alimentados por batería, se recomienda integrar el uso de esp_light_sleep_start() o esp_deep_sleep_start(). Configurar interrupciones externas (como un flanco de subida en el pin RX de la UART) puede despertar al microprocesador solo cuando hay datos entrantes, reduciendo drásticamente el consumo promedio de corriente.

Optimización de Prioridades y Stack: Es vital asignar correctamente las prioridades de las tareas y dimensionar el stack de memoria (en palabras, no bytes). Una tarea de comunicación UART debe tener una prioridad más alta que una tarea de reporte periódico para evitar la pérdida de caracteres en el buffer, mientras que el monitoreo del stack ayuda a prevenir el desbordamiento de memoria (*stack overflow*) en sistemas complejos.

Link de Github: EstraRobert/TAREA5-Sistemas-Embebidos: Se encuentra todo lo solicitado en la Tarea4

INTEGRANTES DEL GRUPO: Josue Vera, Josue Bajaña y Roberto Estrada