Efficient Inter-Task Communication with ESP32 Using the Event Loop Library

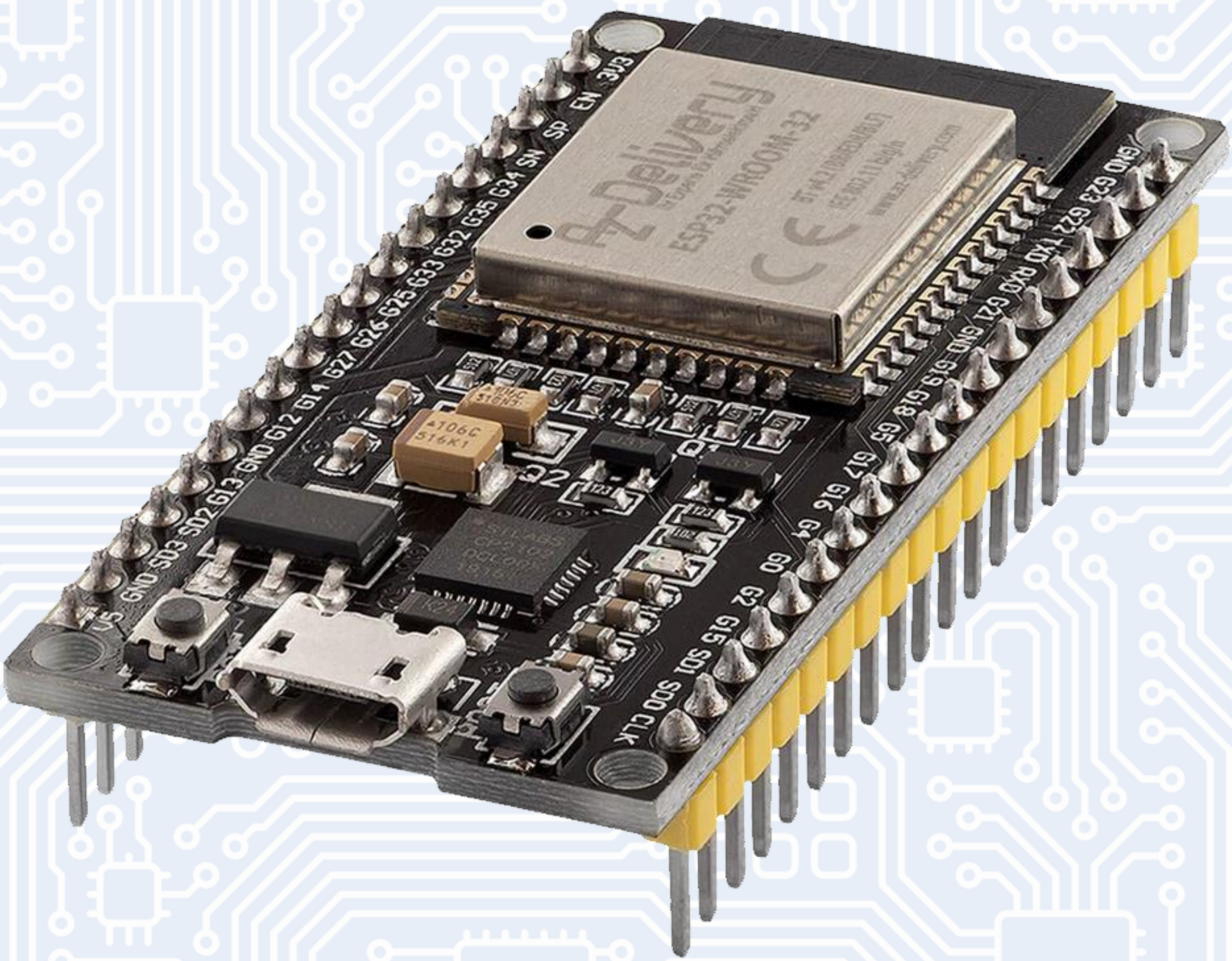

Table of Contents

1. Introduction
2. What Is the Event Loop Pattern?
3. Why Use esp_event for Inter-Task Communication?
4. Setting Up Custom Event Bases and IDs
5. Creating Sensor Tasks (Event Producers)
6. Designing the Event Handler (Consumer)
7. Putting It All Together: Full Code Example
8. Advantages of the Event Loop Architecture
9. Potential Extensions
10. Conclusion

1. Introduction

Modern IoT applications on microcontrollers like the ESP32 often involve multiple parallel tasks such as sensor reading, data processing, and communication. In such systems, effective inter-task communication is crucial for maintaining modularity, scalability, and responsiveness.While FreeRTOS provides powerful primitives like queues, semaphores, and event groups, the ESP-IDF Event Loop Library (esp_event) introduces an elegant alternative based on the Event Loop pattern — a well-known software architecture that promotes decoupling through Inversion of Control (IoC).

1. Introduction

In this article, we explore how to use ESP-IDF’s event loop to establish communication between multiple tasks. We'll implement a real-life scenario where simulated temperature and light sensor tasks post events to a shared handler, enabling clean and scalable task collaboration.

2. What Is the Event Loop Pattern?

The **Event Loop Pattern** is a software design model where actions (handlers) are triggered in response to events instead of being called directly. This inversion of control shifts the execution logic from “caller-driven” to “event-driven.”

In the context of ESP32 and ESP-IDF:

- Producers (tasks, interrupts, timers) **post events** using `esp_event_post()`.
- Consumers **register handlers** for specific events via `esp_event_handler_register()`.
- The **event loop** (either default or custom) delivers the events and invokes the handlers.

2. What Is the Event Loop Pattern?

In the context of ESP32 and ESP-IDF:

- Producers (tasks, interrupts, timers) **post events** using `esp_event_post()`.
- Consumers **register handlers** for specific events via `esp_event_handler_register()`.
- The **event loop** (either default or custom) delivers the events and invokes the handlers.

This pattern eliminates tight coupling between tasks and allows more flexible system designs.

3. Why Use esp_event for Inter-Task Communication?

Unlike traditional FreeRTOS methods like queues or shared buffers, the esp_event system provides:

- **Modularity**: Producers and consumers are decoupled.
- **Thread safety**: You can post events from any context (even ISRs with esp_event_isr_post()).
- **Scalability**: Supports multiple event bases and handlers.
- **Low overhead**: Efficient internal dispatch mechanism.
- **Inversion of Control**: Consumers react only when events occur.

4. Setting Up Custom Event Bases and IDs

To build a modular event system, you define:

```
ESP_EVENT_DEFINE_BASE(SENSOR_EVENT);
```

This defines a new event base (SENSOR_EVENT) which will group related events.

Then, create an enumeration of event IDs:

```
typedef enum {
    SENSOR_EVENT_TEMP_READY,
    SENSOR_EVENT_LIGHT_READY
} sensor_event_id_t;
```

These IDs allow the event loop to distinguish between different types of data.

5. Creating Sensor Tasks (Event Producers)

In a real-world IoT application, tasks often collect data from sensors. Here, we simulate two such producers:

```c
static void temp_sensor_task(void *arg) {
    while (1) {
        int temp = esp_random() % 100;
        esp_event_post(SENSOR_EVENT, SENSOR_EVENT_TEMP_READY,
                       &temp, sizeof(temp), portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(2000));
    }
}

static void light_sensor_task(void *arg) {
    while (1) {
        int light = esp_random() % 1000;
        esp_event_post(SENSOR_EVENT, SENSOR_EVENT_LIGHT_READY,
                       &light, sizeof(light), portMAX_DELAY);
        vTaskDelay(pdMS_TO_TICKS(3000));
    }
}
```

These tasks simulate sensor readings and post them as events every 2–3 seconds.

6. Designing the Event Handler (Consumer)

The event handler receives posted data and performs processing:

```c
static void sensor_event_handler(void *handler_arg,
                                 esp_event_base_t base,
                                 int32_t id,
                                 void *event_data) {
    if (base == SENSOR_EVENT) {
        switch (id) {
            case SENSOR_EVENT_TEMP_READY:
                ESP_LOGI("EVENT_HANDLER", "Temperature: %d°C",
                    *(int *)event_data);
                break;
            case SENSOR_EVENT_LIGHT_READY:
                ESP_LOGI("EVENT_HANDLER", "Light level: %d lux",
                    *(int *)event_data);
                break;
        }
    }
}
```

6. Designing the Event Handler (Consumer)

You can register this handler to listen to all events under SENSOR_EVENT using:

```
esp_event_handler_register(SENSOR_EVENT, ESP_EVENT_ANY_ID,
                           &sensor_event_handler, NULL);
```

7. Putting It All Together: Full Code Example

Here's the complete implementation:

```c
#include <stdio.h>
#include <stdlib.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_event.h"
#include "esp_log.h"
#include "esp_random.h"  // required for esp_fill_random()

// Define a custom event base for sensor-related events
ESP_EVENT_DEFINE_BASE(SENSOR_EVENT);

// Enumeration of sensor event IDs.
typedef enum {
    SENSOR_EVENT_TEMP_READY,    // Temperature data ready
    SENSOR_EVENT_LIGHT_READY    // Light sensor data ready
} sensor_event_id_t;

/**
* @brief Simulated temperature sensor task.
*         Reads (simulates) temperature data and posts it as an event.
*
* @param arg Pointer to user argument (not used here).
*/
static void temp_sensor_task(void *arg) {
    while (1) {
        // Simulate a temperature value (0-99 °C)
        uint32_t temp;
        esp_fill_random(&temp, sizeof(temp));
        temp %= 100;

        ESP_LOGI("TEMP_TASK", "Posting temperature: %lu°C", (unsigned long)temp);

        // Post temperature event to the default event loop
        esp_event_post(SENSOR_EVENT, SENSOR_EVENT_TEMP_READY,
                       &temp, sizeof(temp), portMAX_DELAY);

        vTaskDelay(pdMS_TO_TICKS(2000)); // Wait 2 seconds
    }
}
```

7. Putting It All Together: Full Code Example

```c
                      &temp, sizeof(temp), portMAX_DELAY);

/**
* @brief Simulated light sensor task.
*        Reads (simulates) light sensor data and posts it as an event.
*
* @param arg Pointer to user argument (not used here).
*/
static void light_sensor_task(void *arg) {
    while (1) {
        // Simulate light level (0-999 lux)
        uint32_t light;
        esp_fill_random(&light, sizeof(light));
        light %= 1000;

        ESP_LOGI("LIGHT_TASK", "Posting light: %lu lux", (unsigned long)light);

        // Post light level event to the default event loop
        esp_event_post(SENSOR_EVENT, SENSOR_EVENT_LIGHT_READY,
                       &light, sizeof(light), portMAX_DELAY);

        vTaskDelay(pdMS_TO_TICKS(3000)); // Wait 3 seconds
    }
}

/**
* @brief Common event handler for all sensor events.
*        Processes both temperature and light events.
*
* @param handler_arg Argument passed during registration (unused).
* @param base The event base (SENSOR_EVENT).
* @param id Event ID (TEMP_READY or LIGHT_READY).
* @param event_data Pointer to the event data (temperature/light value).
*/
static void sensor_event_handler(void *handler_arg,
                                 esp_event_base_t base,
                                 int32_t id,
                                 void *event_data) {
    if (base == SENSOR_EVENT) {
        switch (id) {
            case SENSOR_EVENT_TEMP_READY: {
                int temp = (int)*(uint32_t *)event_data;
                ESP_LOGI("EVENT_HANDLER", "Temperature received: %d°C", temp);
                // Optional: take action if temp exceeds threshold
                break;
```

7. Putting It All Together: Full Code Example

```c
 * @param handler_arg Argument passed during registration (unused).
static void sensor_event_handler(void *handler_arg,
                                  esp_event_base_t base,
                                  int32_t id,
                                  void *event_data) {
    if (base == SENSOR_EVENT) {
        switch (id) {
            case SENSOR_EVENT_TEMP_READY: {
                int temp = (int)*(uint32_t *)event_data;
                ESP_LOGI("EVENT_HANDLER", "Temperature received: %d°C", temp);
                // Optional: take action if temp exceeds threshold
                break;
            }
            case SENSOR_EVENT_LIGHT_READY: {
                int light = (int)*(uint32_t *)event_data;
                ESP_LOGI("EVENT_HANDLER", "Light level received: %d lux", light);
                // Optional: take action if light level is too low/high
                break;
            }
            default:
                ESP_LOGW("EVENT_HANDLER", "Unknown event ID: %ld", id);
        }
    }
}


/**
* @brief Application entry point.
*        Initializes the default event loop, registers the sensor event handler,
*        and launches the temperature and light reading tasks.
*/
void app_main(void) {
    ESP_LOGI("APP_MAIN", "Starting Event Loop Inter-Task Communication Example");

    // Step 1: Create the default event loop (global, shared across all components)
    ESP_ERROR_CHECK(esp_event_loop_create_default());

    // Step 2: Register the sensor event handler for all sensor events
    ESP_ERROR_CHECK(esp_event_handler_register(SENSOR_EVENT, ESP_EVENT_ANY_ID,
                                               &sensor_event_handler, NULL));

    // Step 3: Launch the simulated sensor tasks
    xTaskCreate(temp_sensor_task, "temp_sensor_task", 2048, NULL, 5, NULL);
    xTaskCreate(light_sensor_task, "light_sensor_task", 2048, NULL, 5, NULL);
}
```

8. Advantages of the Event Loop Architecture

Feature	Benefit
Inversion of Control	Tasks only post; handlers decide the action
Thread-safe	Safe to use from ISRs or multiple tasks
Modularity	Easy to scale, maintain, or replace handlers
Efficiency	Minimal memory and CPU usage

9. Conclusion

The **Event Loop Library in ESP-IDF** is a powerful, elegant solution for inter-task communication in FreeRTOS-based applications. It brings flexibility, modularity, and cleaner code through the **Inversion of Control principle**.

By using **esp_event**, developers can decouple producers and consumers, scale applications without redesigning communication flows, and integrate clean architectures into their IoT designs. Whether you're working on sensors, actuators, or cloud communication, the event loop architecture offers a robust foundation for modern embedded systems.