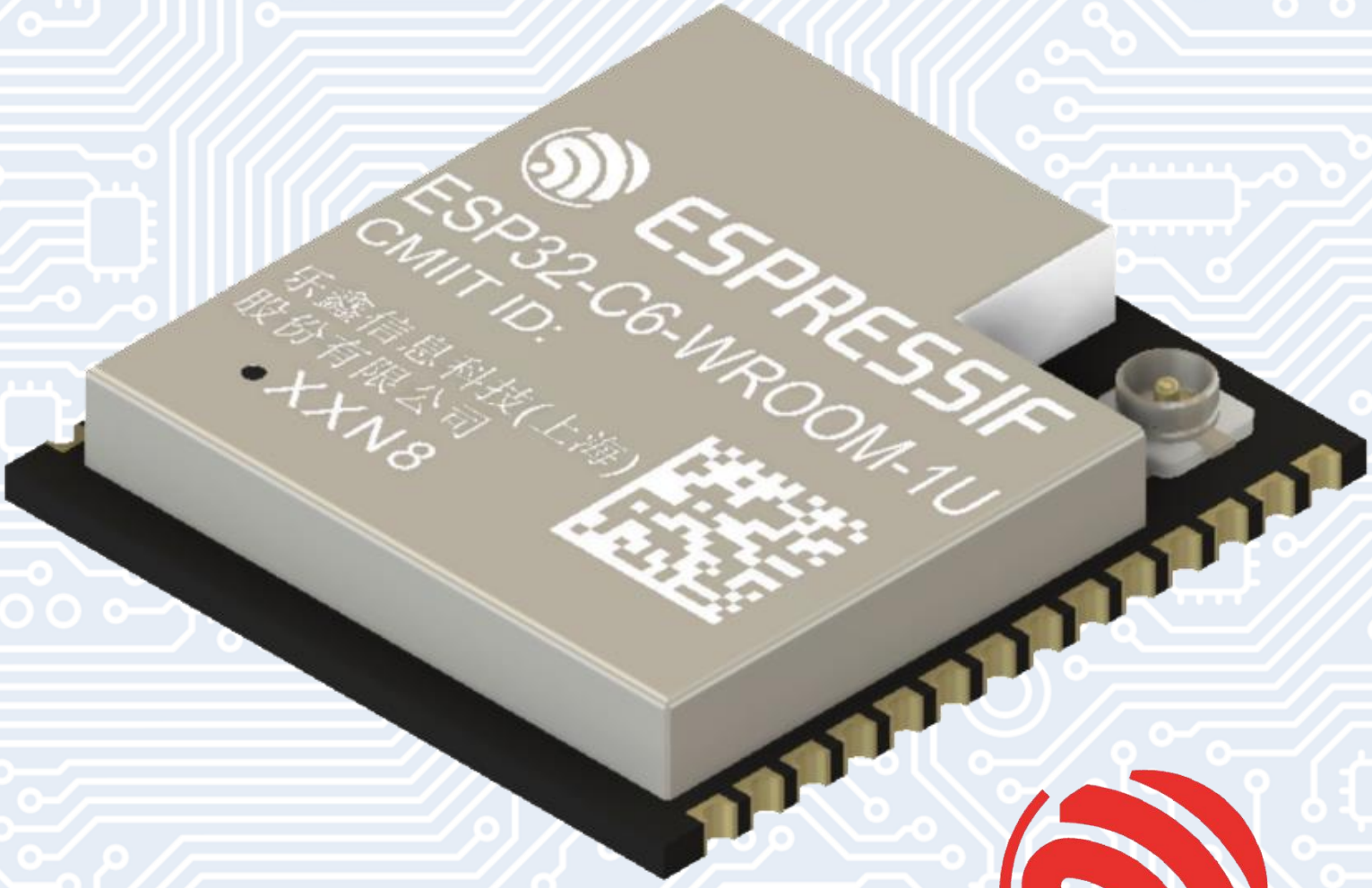
How to Create and Use a Software Timer with ESP32 and ESP-IDF Framework



freeRTOS

ESP-IDF

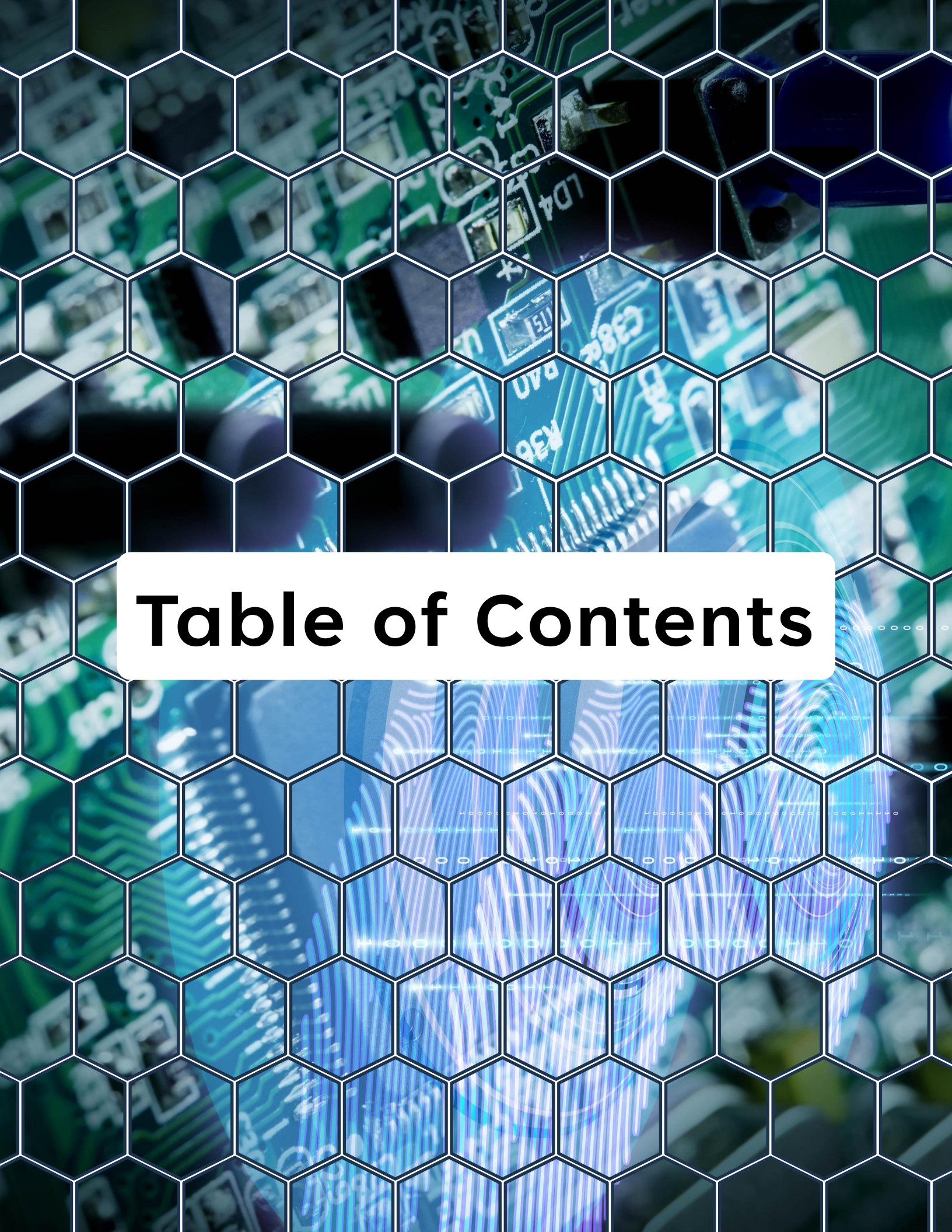
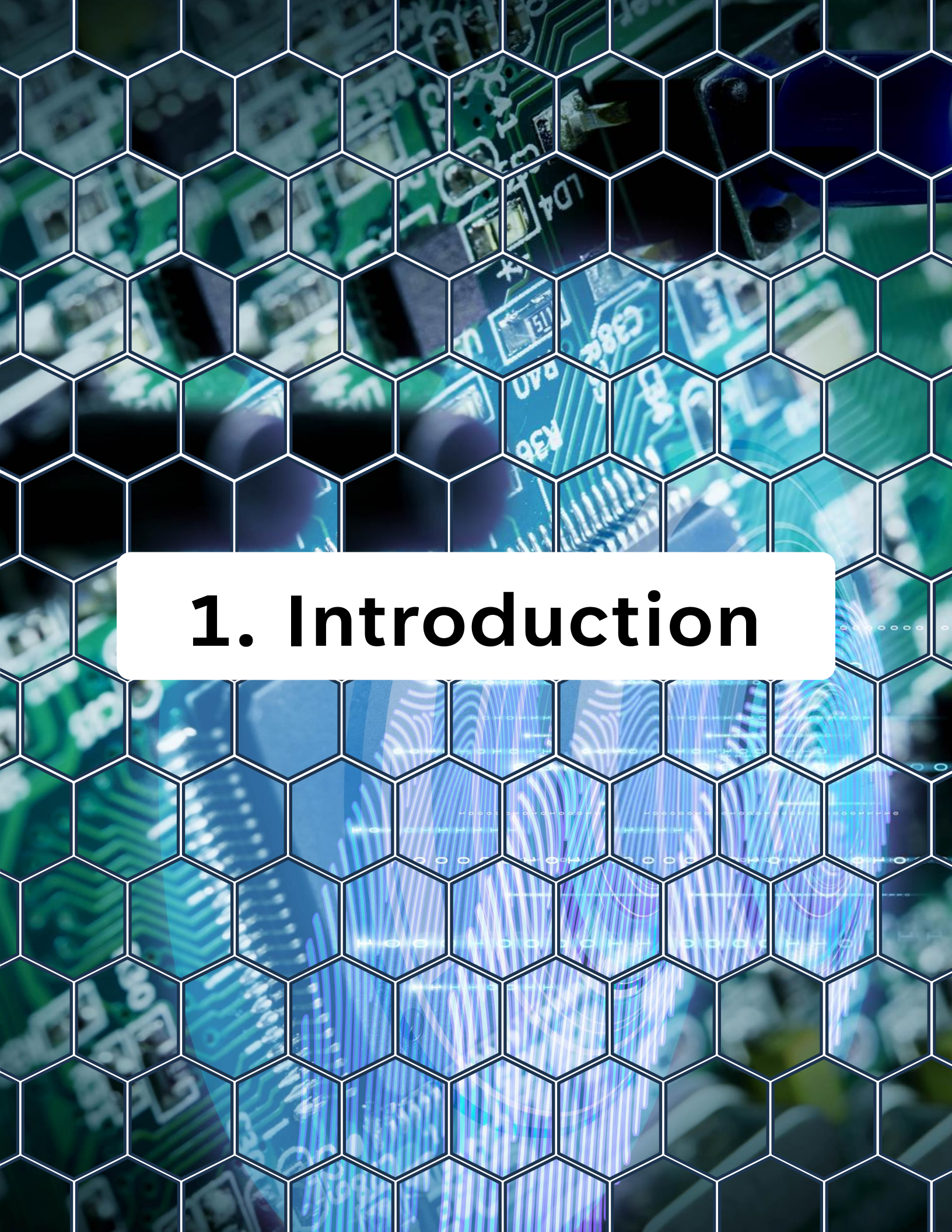


Table of Contents

Table of Contents



1. Introduction

1. Introduction

In modern IoT applications, timing is everything. Whether it's blinking LEDs, reading sensors periodically, or managing communication timeouts, software timers are a critical building block.

In this article, we will explore how to create and use software timers in the **ESP-IDF** framework on the **ESP32** microcontroller — a platform renowned for its versatility and robustness in IoT development.

2. What is a Software Timer?

2. What is a Software Timer?

A **Software Timer** is a timer managed by the **FreeRTOS kernel** rather than by hardware peripherals. Instead of using a dedicated hardware timer, software timers allow you to schedule the execution of a callback function after a specified period — either once or periodically — without consuming critical hardware resources.

In ESP-IDF, FreeRTOS software timers are extensively used to implement non-blocking periodic or delayed tasks.

3. Why Use Software Timers in IoT Applications?

3. Why Use Software Timers in IoT Applications?

Software timers provide:

- **Resource efficiency**: Save precious hardware timers for time-critical operations.
- **Flexibility**: Easy to create, start, stop, and manage dynamically.
- **Simplicity**: No need to handle interrupts manually.
- **Portability**: Code written with software timers is easier to migrate across platforms.

3. Why Use Software Timers in IoT Applications?

Typical use cases include:

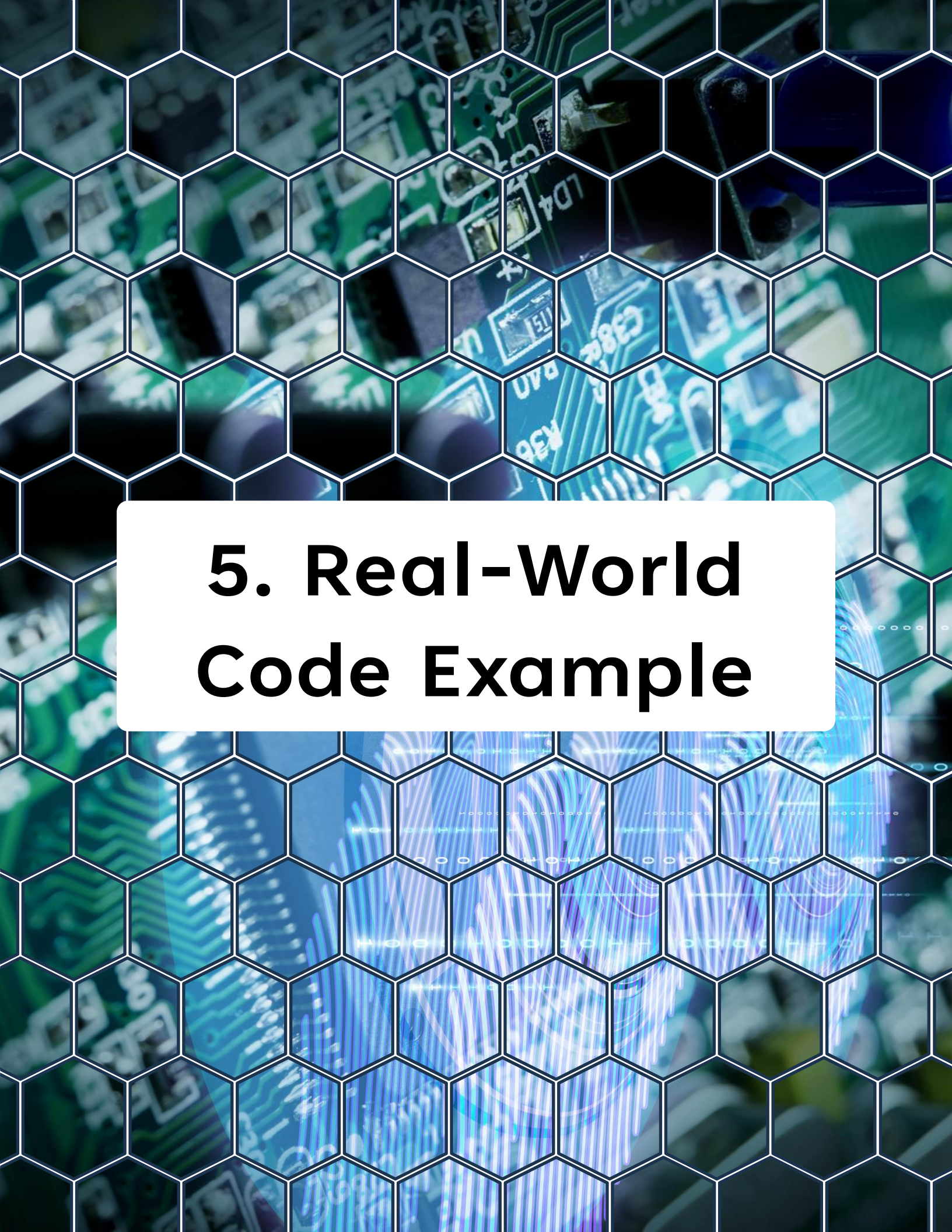
- Periodic sensor data reading
- Triggering watchdog resets
- Managing communication timeouts
- Delaying execution of non-critical operations

4. Steps to Create and Use a Software Timer

4. Steps to Create and Use a Software Timer

Here’s the basic process when working with software timers in ESP-IDF:

- **Define the timer callback function**: The function to execute when the timer expires.
- **Create the timer**: Using `xTimerCreate()`.
- **Start the timer**: Using `xTimerStart()`, optionally with a timeout.
- **Optionally stop/reset/change period**: Using corresponding FreeRTOS timer APIs like `xTimerStop()`, `xTimerReset()`, etc.



5. Real-World Code Example

5. Real-World Code Example

Let’s implement a software timer that toggles an LED every 2 seconds.

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

// Define LED GPIO
#define LED_GPIO GPIO_NUM_2

// Global variable declaration
TimerHandle_t blink_timer;  // Timer Handle
static const char *TAG = "Software Timer Example";

// Timer Callback Function
void blink_timer_callback(TimerHandle_t xTimer) {
    // Toggle the LED
    static bool led_on = false;
    gpio_set_level(LED_GPIO, led_on);
    led_on = !led_on;
    ESP_LOGI(TAG, "LED is now %s", led_on ? "ON" : "OFF");
}

// Program entry point
void app_main(void) {
    // Initialize the LED GPIO
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << LED_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = 0,
        .pull_down_en = 0,
```

5. Real-World Code Example

```
// Program entry point
void app_main(void) {
    // Initialize the LED GPIO
    gpio_config_t io_conf = {
        .pin_bit_mask = (1ULL << LED_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = 0,
        .pull_down_en = 0,
        .intr_type = GPIO_INTR_DISABLE,
    };
    gpio_config(&io_conf);

    // Create the software timer
    // The minimum software timer granularity is 1 ms
    blink_timer = xTimerCreate(
        "BlinkTimer",               // Name of timer
        pdMS_TO_TICKS(2000),        // Timer period in ticks (2000ms)
        pdTRUE,                     // Auto-reload (pdTRUE means periodic)
        (void *)0,                  // Timer ID (not used here)
        blink_timer_callback        // Callback function
    );

    if (blink_timer == NULL) {
        printf("Failed to create timer\n");
        return;
    }

    // Start the timer
    if (xTimerStart(blink_timer, 0) != pdPASS) {
        printf("Failed to start timer\n");
    }
}
```

5. Real-World Code Example

Code Breakdown:

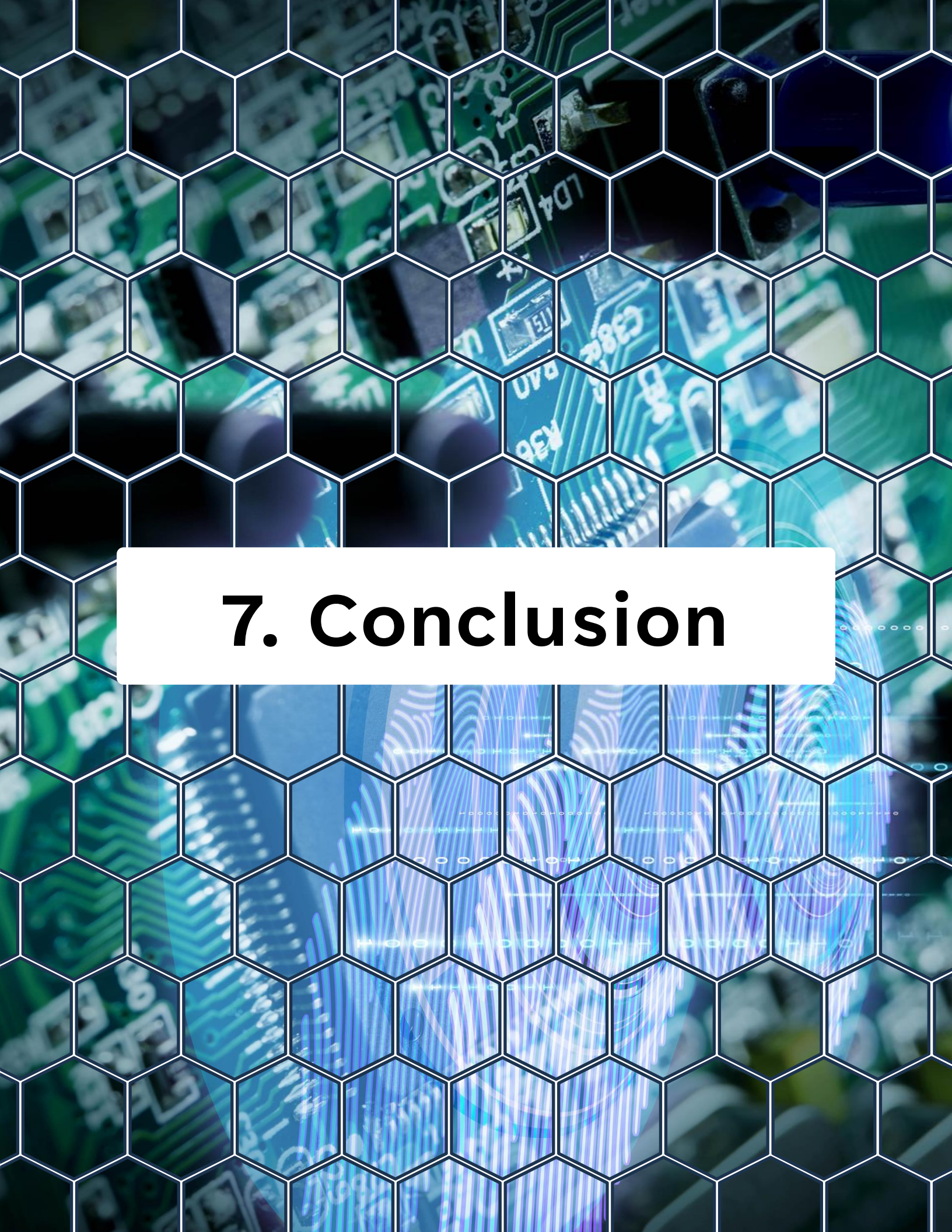
- **GPIO Setup**: Configures the LED pin as an output.
- **Timer Creation**: xTimerCreate sets up a 2-second periodic timer linked to a callback function.
- **Timer Start**: xTimerStart starts the timer immediately.
- **Callback Function**: blink_timer_callback toggles the LED state every time the timer fires.

6. Best Practices

6. Best Practices

When using software timers in ESP32 projects:

- **Keep callbacks short**: Timer callbacks should execute quickly to avoid delaying other timers.
- **Use proper synchronization**: If the callback touches shared resources, protect them with mutexes or other synchronization primitives.
- **Always check return values**: Functions like `xTimerCreate()` and `xTimerStart()` can fail — never assume success.
- **Use meaningful timer names**: Helps in debugging using the ESP-IDF tracing tools.
- Free resources if needed: Use `xTimerDelete()` if you dynamically create timers that will no longer be used.



7. Conclusion

7. Conclusion

Software timers are an indispensable feature for managing time-driven events in IoT applications with ESP32 and ESP-IDF. By integrating them properly, developers can design efficient, non-blocking, and scalable systems without consuming scarce hardware resources. Whether it's blinking LEDs, reading sensors, or triggering maintenance tasks, software timers provide a lightweight and flexible solution that fits perfectly into FreeRTOS-based projects.

In the dynamic world of IoT development, mastering tools like software timers separates a good design from an outstanding one.